LR Parsing pt. 2: LR(0) and SLR(1) Parsers

CS 440: Programming Languages and Translators, Spring 2020

3/25 v.3, 3/25 21:16 p.7

1. Review of LR Parsers

- LR parsers (read input Left-to-right and build reversed Rightmost derivations) are a form of shift-reduce parsers. I.e., they work bottom-up, building small parse trees from terminal symbols and combine those into larger and larger parse trees, ending with a parse tree for the whole input.
- An LR parser uses a stack to hold the input that hasn't been completely processed (the terminal and/or nonterminal symbols that have been encountered up to this point in the input string).
- An LR parser is controlled by a finite state automaton[1]. Though the basic LR parser automaton is nondeterministic, we can build a deterministic version using the set-of-states transformation we saw with finite state machines.
  - When we make a nondeterministic jump from one state to another, we don't change the stack, so we don't have to worry about trying to do a set-of-states-**and-stacks** transformation, which is good because we can't.
- In our initial LR parsers, the CFSM states are **LR(0) items**: Grammar rules augmented with position dots indicating where we are in the parse of the rules. (E.g., $A \rightarrow \bullet$ `a` B, A $\rightarrow$ `a` $\bullet B$, $A \rightarrow$ `a` $B \bullet$.)[2]
  - Input symbols get **shifted** onto the stack and cause the automaton to move forward in the rule we're parsing. (E.g., shifting an `a` causes us to go from $A \rightarrow \bullet$ `a` B to $A \rightarrow$ `a` $\bullet B$.)
  - If we're in a point in a rule where we expect a nonterminal, then we nondeterministically jump to the beginning of every rule for that nonterminal. (E.g., from $A \rightarrow$ `a` $\bullet B$, we might jump to $B \rightarrow \bullet$ `b c` and $B \rightarrow \bullet$ `c`.)
  - If we reach the end of a rule, we can **reduce** the stack using that rule by popping the rhs of the rule off the stack and pushing on the lhs nonterminal. (E.g., from $B \rightarrow \bullet$ `b c`, if we shift a `b` and a `c`, then we're at $B \rightarrow$ `b c` $\bullet$ and can reduce by popping off the `b` and `c` and pushing on B.)
- After reduction, go back to the earlier rule.
  - Once we reduce, we go back to the rule we were in before and progress through that rule using the nonterminal we just reduced to.
    - E.g., if we were at $B \rightarrow$ `a` $\bullet A\ \beta$, jumped to $A \rightarrow \bullet$ `b c`, parsed `b c` and reduced via $A \rightarrow$ `b c` $\bullet$, then we go back to $B \rightarrow$ `a` $\bullet A\ \beta$ and move forward to $B \rightarrow$ `a` $A \bullet \beta$.
  - To keep track of what item we're in when we jump to start parsing a different rule, we store the state names on the stack along with the sentential form entries.
  - The top of the stack always contains the current state number.

---

[1] The typical name for the control automaton is CFSM: the Characteristic Finite State Machine.

[2] When we get to full LR(1) parsers, we'll use LR(1) items, which extend LR(0) items.

- A state number inside the stack falls just under the (terminal or nonterminal) symbol that got pushed on when we were in that state.
- When we reduce, we uncover the state we were in before going off to parse the rule we just reduced. That tells us what state to go to after we push the nonterminal we just reduced to.

Action and GoTo Tables

- To represent the CFSM transitions in table form, we use an **Action table**, which maps states and terminal symbols to the states we shift to; and a **GoTo table**, which maps states and nonterminals to the new state to go to after a reduction.
- In the Action table
  - An "s *nbr*" action indicates a shift. We push the current input symbol onto the stack, push the new state number onto the stack, and go to that state. So shifts handle transitions like $A \to \alpha \bullet x\ \beta$ to $A \to \alpha\ x \bullet \beta$, where $x \in \Sigma$.
  - An "r *nbr*" action indicates a reduction of the named rule, say $A \to \alpha \bullet$. First, we pop the symbols of α off the stack (along with the state numbers above each rhs symbol). This exposes a state number, which we note. (It's the state we were in when we decided to go to $A \to \bullet\ \alpha$.). Then we push on the lhs nonterminal A, and push on the new state (found by using the GoTo table).
- The GoTo table maps states and nonterminals to the new states to go to. Consider the transition $B \to \gamma \bullet A\ \delta$ to $B \to \gamma\ A \bullet \delta$. The state number exposed during reduction is for $B \to \gamma \bullet A\ \delta$. Indexing into the GoTo table with the state number and A gives us the state number for $B \to \gamma\ A \bullet \delta$, so we push it onto the stack and make it the new state.
- Be sure of the difference between the numbers attached to shift and reduce operations: For shift we get the new state to go to; for reduce, we get the rule number to reduce with, and we find the new state to go to in the GoTo table.

Example 1: An LR(0) Parser, with Action and GoTo tables

- Let's reuse the LR(0) grammar example from the previous lecture: rule 0 is $S' \to S$ `$`, rule 1 is $S \to$ `a` A, and rule 2 is $A \to$ `c`.
- For the deterministic automaton, we get state 0 is $\{S' \to \bullet\ S$ `$`, $S \to \bullet$ `a` $A\}$, state 1 is $\{S \to$ `a` $\bullet\ A$, $A \to \bullet$ `c`$\}$, state 2 is $\{A \to$ `c` $\bullet\}$, state 3 is $\{S \to$ `a` $A \bullet\}$, state 4 is $\{S' \to S \bullet$ `$`$\}$, and state 5 is $\{S' \to S$ `$` $\bullet\}$.
- The CFSM transitions are:
  - In state 0 (the initial state), if we see an S, we go to state 4; if we see an `a` we go to state 1.
  - In state 1, if we see an A we go to state 3; if we see a `c` we go to state 2.
  - In state 2, we reduce $A \to$ `c` $\bullet$.
  - In state 3, we reduce $S \to$ `a` $A \bullet$
  - In state 4, if we see a `$`, we go to state 5.
  - In state 5, we accept the input. (We don't reduce the rule for the unique start symbol.)

- Below are the Action and GoTo tables for this grammar along with a trace of a successful parse of `a c $`.

Action & GoTo Tables for Example 1

State	*Action*			*GoTo*	
	`a`	`c`	`$`	*S*	*A*
0	s1			4	
1		s2			3
2	r2	r2	r2		
3	r1	r1	r1		
4			s5		
5	accept input				

Successful Parse of `a c $`

Stack (top at right)	Input	Action
0	`a c $`	s3
0 `a` 1	`c $`	s5
0 `a` 1 `c` 2	`$`	r2
0 `a` 1 *A* 3	`$`	r1
0 *S* 4	`$`	s2
0 *S* 4 `$` 5	ε	accept

- With an LR(0) parser, if there's a reduction operation in some Action table row, then *all* the Action table entries for that row have the same reduction. This is because in an LR(0) parser the next input symbol is irrelevant for deciding whether or not to reduce: An LR(0) parser always reduces if it can.
- Note that each entry of the Action and GoTo tables is either empty (indicating an error) or it has exactly one action. If there's more than one action in an entry, we say that that entry contains a **conflict**.

2. Conflicts in LR Parsers

- There are two kinds of conflicts.
  - **Reduce-Reduce conflict**: $\{A \to \alpha \bullet,\ B \to \beta \bullet\}$ indicates we have two possible reductions, $A \to \alpha \bullet$ and $B \to \beta \bullet$. We can only do one or the other, so we have a conflict.
  - **Shift-Reduce conflict**: With $\{P \to \alpha \bullet \gamma,\ R \to \alpha \bullet\}$, we can reduce using $R \to \alpha \bullet$ or we can shift the next input symbol (in the hope that it begins a yield of γ). Again, we can only do one or the other.
  - **No Shift-Shift conflicts**: LR parsers don't have shift-shift conflicts because of the set-of-states transformation. Say nondeterministically we can be in two states $C \to \alpha \bullet \texttt{c}\ \beta$ or $D \to \gamma \bullet \texttt{c}\ \delta$. The set-of-states transforms combines these into a state that includes both of them. The nondeterministic transitions on input `c` become a deterministic transition to a state that contains both $C \to \alpha\ \texttt{c} \bullet \beta$ and $D \to \gamma\ \texttt{c} \bullet \delta$.

Handling Conflicts

- It's possible for a grammar to cause a conflict in one kind of LR parser but not another; in that case, we can handle the problem by changing parsers. But it's also possible for all LR parsers to have the conflict, in which case we need to revise the grammar (and possibly the language).
- The reason it's possible to have one LR parser that has a conflict and another parser that doesn't conflict is because different LR parsers use different techniques for deciding when to reduce. E.g., LR(0) parsers always

reduce when it's possible to reduce, but non-LR(0) parsers use the next input symbol to decide whether or not to reduce.

- LR parsers all use the same shift operation, so they differ basically only in when they decide to reduce.

3. When do LR Parsers Reduce?

- One way to characterize the difference between families of LR parsers is to study the lookahead symbols they check for. I.e., for $A \rightarrow \alpha \bullet$, one reduces when the next input symbol [fill in the blank].
- **LR(0) parsers** always reduce. They use no lookahead symbol (the zero in LR(0)), so the next input is irrelevant. (Or you can characterize it as "Reduce if next symbol $\in \Sigma$ (the alphabet)", which is always true.)
- **SLR(1) parsers (Simple LR parsers)** reduce $A \rightarrow \alpha \bullet$ if the next input $\in$ *Follow*(A). This is the same follow set we used with LL(1) parsers.
- **"Canonical" LR(1) parsers** refine the follow sets by remembering the rule the lhs nonterminal appeared in.
  - **Preview** (We'll study this in detail later): Say we have rules $X \rightarrow \bullet\ A$ `a` and $Y \rightarrow \bullet\ A$ `b`. We reduce $A \rightarrow \alpha \bullet$ if we came from the $X \rightarrow \bullet\ A$ `a` rule and the next input symbol is `a`; we also reduce if we came from the $Y \rightarrow \bullet\ A$ `b` rule and the next input symbol is `b`. In contrast, an SLR(1) parser reduces $A \rightarrow \alpha \bullet$ if the next input symbol is `a` or `b`, regardless of whether we came from the X rule or Y rule.
- **LALR(1) parsers (Look Ahead LR parsers)** are a condensed version of canonical LR(1) parsers. (We'll study this in detail later.)

4. The difference between LR(0) and SLR(1) Parsers

- The zero in LR(0) indicates we don't use any lookahead at all. As a result, we always reduce when we reach the end of a rule. This makes for a simple parser, but relatively few grammars are LR(0).
- The 1 in SLR(1) ("Simple LR") indicates we use one symbol of lookahead (i.e., the next input symbol). When we get to the end of a rule $A \rightarrow \alpha \bullet$, we reduce if the next symbol $\in$ Follow(A).
- Since SLR(1) checks for *lookahead* $\in$ Follow(*lhs nonterminal*) and LR(0) in effect checks for *lookahead* $\in \Sigma$ (the alphabet), we can illustrate the difference by giving a grammar where some nonterminal's follow set is not the whole alphabet.
- An SLR(1) parser can be turned into the corresponding LR(0) parser by setting it to reduce regardless of the lookahead.

Notation for LR(0) Items and States containing them

- **Notation**: Let's number grammar rules 0, 1, 2, etc., with rule 0 always being the rule for the unique start symbol (e.g., $S' \rightarrow S$ `$`). The LR(0) items for each rule will be named a, b, c, … according to where the dot position is in the item. (E.g., 0a is $S' \rightarrow \bullet\ S$ `$`, 0b is $S' \rightarrow S \bullet$ `$`, and 0c is $S' \rightarrow S$ `$` $\bullet$.) Note different rule-letter combinations refer to different LR(0) items.
- (**Exception**: We haven't looked at ε rules, but $X \rightarrow \bullet\ \varepsilon$ and $X \rightarrow \varepsilon \bullet$ are the same. (We use $X \rightarrow \varepsilon \bullet$ when we're talking about situations where we do want to reduce $X \rightarrow \varepsilon$ and $X \rightarrow \bullet\ \varepsilon$ when we don't want to reduce.)

An Example of a non-LR(0) grammar

- **Example 2**: Consider the grammar with rules 0: $S' \rightarrow S$ `$`, 1: $S \rightarrow A$ `a`, 2: $S \rightarrow B$ `b`, 3: $A \rightarrow$ `c`, and 4: $B \rightarrow$ `c`.
- The follow sets are Follow(S') = ∅, Follow(S) = {`$`}, Follow(A) = {`a`}, Follow(B) = {`b`}.

The SLR(1) Parser

- The CFSM for the SLR(1) parser for this grammar has the following states and next-symbol transitions.
  Notes: The row for state {0b} doesn't include r0 because Follow(S') = ∅. I've used the states in the Action and GoTo tables instead of the state numbers, just to make things more explicit.

SLR(1) CFSM [3/24]

State #	State Items	Actions				GoTo		
		`a`	`b`	`c`	`$`	*S*	*A*	*B*
0	{0a: $S' \rightarrow \bullet S$ `$`, 1a: $S \rightarrow \bullet A$ `a`, 2a: $S \rightarrow \bullet B$ `b`, 3a: $A \rightarrow \bullet$ `c`, 4a: $B \rightarrow \bullet$ `c`}			{3b, 4b}		{0b}	{1b}	{2b}
1	{0b: $S' \rightarrow S \bullet$ `$`}				{0c}			
2	{0c: $S' \rightarrow S$ `$` $\bullet$}	accept						
3	{1b: $S \rightarrow A \bullet$ `a`}	{1c}						
4	{1c: $S \rightarrow A$ `a` $\bullet$}				r1			
5	{2b: $S \rightarrow B \bullet$ `b`}		{2c}					
6	{2c: $S \rightarrow B$ `b` $\bullet$}				r2			
7	{3b: $A \rightarrow$ `c` $\bullet$, 4b: $B \rightarrow$ `c` $\bullet$}	r3	r4					

- The grammar is SLR(1) because there are no shift-reduce or reduce-reduce conflicts; i.e., no transition entries with a state and reduction or with two reductions.
  - The only row that contains two different kinds of reductions is row {3b, 4b}. It has reduce operations for $A \rightarrow$ `c` $\bullet$ and $B \rightarrow$ `c` $\bullet$ but the follow sets for A and B have no intersection, so we don't have both reductions for the same entry.

The LR(0) Parser (has conflicts)

- The LR(0) parser for this grammar has the same shift operations as the SLR(1) parser, but it has more generous reduction rows. Since LR(0) parsers reduce regardless of lookahead, the row below for state {3b, 4b} has multiple reduce entries, so the grammar is **not LR(0)**.

LR(0) CFSM — has conflicts [3/24]

State #	State Items	Actions				GoTo		
		`a`	`b`	`c`	`$`	*S*	*A*	*B*
…	…							
4	{1c: $S \rightarrow A$ `a` $\bullet$}	r1	r1	r1	r1			
…	…							
6	{2c: $S \rightarrow B$ `b` $\bullet$}	r2	r2	r2	r2			
7	{3b: $A \rightarrow$ `c` $\bullet$, 4b: $B \rightarrow$ `c` $\bullet$}	r3, r4	r3, r4	r3, r4	r3, r4			

Example 3: An SLR(1) parser with delayed error detection

- In an SLR(1) grammar, we reduce if the next symbol is in the follow set for the lhs nonterminal.
- It turns out that this is not ideal because, depending on earlier parts of the derivation, we might want to reduce only if the lookahead is in a subset of the full follow set.
- The grammar has three rules 0: $S' \rightarrow S$ `$`, 1: $S \rightarrow A$ `b` A `c`, 2: $A \rightarrow$ `a`.
  - It only generates one string, `a b a c`; the derivation is $S' \rightarrow S$ `$` $\rightarrow A$ `b` A `c` `$` $\rightarrow A$ `b a c` `$` $\rightarrow$ `a b a c` `$`.
  - In state terms, this is {0a, 1a, 2a} → (on `a`) {2b} → (reduce 2) {1b} → (on `b`) {1c, 2a} → (on `a`) {2b} → (reduce 2) {1d} → (on `c`) → {1e} → (reduce 1) {0b} → (on `$`) {0c}.
- An SLR(1) parser will (correctly) reject the input `a c` … `$` (we don't care what comes after the `c`).
  - Tracing the states, we get {0a, 1a, 2a} → (on `a`) {2b} → (reduce 2) {1b: $S \rightarrow A \bullet$ `b a c $`}, but the next input symbol is `c`, so the parse fails.
  - Ideally, we'd like the parser to realize there's an error as early as possible, which is when it looks at input symbol `c` to decide whether or not to reduce $A \rightarrow$ `a` $\bullet$.
  - But the parser reduced $A \rightarrow$ `a` $\bullet$, continued with $A \rightarrow A \bullet$ `b` A `c` and then it flagged the input symbol `c` as an error (since it $\not\equiv$ `b`).
- But because it checks for input $\in$ Follow(A) = {`b`, `c`} to okay a reduction, an SLR(1) parser can't distinguish between the transitions
  - {2b} → {1b} → (on `b`) {1c, 2a}
  - {2b} → {1d} → (on `c`) {1e}
- So we reduced $A \rightarrow$ `a` $\bullet$ (for {2b} → {1b}) even though the next symbol was `c`, not `b`.
  - Similarly, we can reduce $A \rightarrow$ `a` $\bullet$ (for {2b} → {1d}) even if the next symbol is `b`, not `c`.
  - See Figure 1 for the Action/Goto tables and traces of parsing `a b a c` (successfully) and `a c` (unsuccessfuly).
- At one level, you can argue that doing the reduction isn't that harmful because we figure out that the input is bad when we try to shift the next symbol.
- Though it doesn't happen in this example, in general, one bad reduction can cause a cascade of bad reductions. E.g., we might reduce an expression, then an assignment statement, then a statement, then a sequence of statements. When we do stop the parse and generate an error message, we'll say that there was something wrong with the sequence of statements, not with the original expression.

SLR Automaton for rules 0: $S' \rightarrow S$ $, 1: $S \rightarrow A$ b A c, 2: $A \rightarrow$ a

State	Items	Action				Go To	
		a	b	c	$	*S*	*A*
0	0a: $S' \rightarrow \bullet S$ $ 1a: $S \rightarrow \bullet A$ b A c 2a: $A \rightarrow \bullet$ a	s6: {2b}				s1: {1a}	s2: {1b}
1	0b: $S' \rightarrow S \bullet$ $				accept		
2	1b: $S \rightarrow A \bullet$ b A c		s3: {1c, 2a}				
3	1c: $S \rightarrow A$ b $\bullet A$ c 2a: $A \rightarrow \bullet$ a	s6: {2b}					s4: {1d}
4	1d: $S \rightarrow A$ b $A \bullet$ c			s5: {1e}			
5	1e: $S \rightarrow A$ b A c $\bullet$				r1		
6	2b: $A \rightarrow$ a $\bullet$		r2	r2			

Successful Parse of a b a c $

Stack (top at right)	Input	Action
0	a b a c $	s6
0 a 6	b a c $	r2
0 *A* 2	b a c $	s3
0 *A* 3 b 3	a c $	s6
0 *A* 3 b 3 a 6	c $	r2
0 *A* 3 b 3 *A* 4	c $	s5
0 *A* 3 b 3 *A* 5 c 5	$	r1
0 *S* 1	$	accept

Unsuccessful Parse of a c $

Stack (top at right)	Input	Action
0	a c $	s6
0 a 6	c $	r2 *
0 *A* 2	$	error

* An SLR(1) parser reduces even though an error is inevitable. A full LR(1) parser will announce an error here.

Figure 1: SLR(1) parser that delays error detection [3/25]

Activity Problems For Lecture 18

Lecture 18: LR Parsers (background)

1. LR Parsers
   a. In general, how are LR parsers different from each other?
   b. In LR parsers, what are shift-reduce and reduce-reduce conflicts? Are there shift-shift conflicts?
   c. How do the Action table and GoTo table work? How are they connected to the CFSM?

Lecture 18: LR(0) and SLR(1) Parsers

2. Study this grammar: Rule 0: $S' \rightarrow X$ `$`, 1: $X \rightarrow$ `a` X `a`, 2: $X \rightarrow$ `b` X `b`, 3: $X \rightarrow Y$, 4: $Y \rightarrow Y$ `y`, 5: $Y \rightarrow$ `y`.
   a. What are the Follow sets for X and Y?
   b. Fill in the action/go-to tables below for an SLR(1) parser for the language.
   c. Is this grammar SLR(1)? If not, why not?
   d. Is this grammar LR(0)? If not, why not?
   e. Does this grammar announce a parse error as soon as possible? (I.e., the first time we look at a lookahead character?) If it does, argue briefly for why that is. If not, give a sample input that causes this symptom.
   f. Trace the action of the parser on some inputs, such as `y`, `a y a`, and `a b y y b a`.

(Incomplete Action/Go-to tables to fill in for part b)

State	LR(0) Items	`a`	`b`	`y`	`$`	*X*	*Y*
0	{0a, 1a, 2a, 3a, 4a, 5a}	s2					
1	{0b}						
2	{1b, 1a, 2a, 3a, 4a, 5a}						
3	{1c}						
4	{1d}						
5	{2b, 1a, 2a, 3a, 4a, 5a}						
6	{2c}						
7	{2d}						
8	{3b, 4b}						
9	{4c}						
10	{5b}						

3. The rules $Y \rightarrow \mathtt{y}\ Y$ and $Y \rightarrow Y\ \mathtt{y}$ (combined with $Y \rightarrow \mathtt{y}$) generate the same strings (namely, $\mathtt{y}^+$). Why do we use the first rule for an LL(1) parser and the second for an SLR(1) parser? (Or equivalently, what happens if we swap them and use the first rule with an SLR(1) parser and the second with an LL(1) parser?)

Selected Solutions to Activity Problems For Lecture 18

Lecture 18: LR Parsers (background)

1. (See the notes).

Lecture 18: LR(0) and SLR(1) Parsers

2. (CFSM for SLR(1) parser)
   a. Follow(X) = {`a`, `b`, `$`}, Follow(Y) = {`a`, `b`, `y`, `$`}
   b. Here are the filled-in action and go-to tables. (I've included the items along with the new state numbers; you didn't have to do that.)

SLR(1) Parser Action/Go-To tables

State	LR(0) Items	`a`	`b`	`y`	`$`	*X*	*Y*
0	{0a, 1a, 2a, 3a, 4a, 5a}	s2: {1b, …}	s5: {2b, …}	s10: {5b}		1: {0b}	8: {3b, 4b}
1	{0b}				accept		
2	{1b, 1a, 2a, 3a, 4a, 5a}	s2: {1b, …}	s5: {2b, …}	s10: {5b}		3: {1c}	8: {3b, 4b}
3	{1c}	s4: {1d}					
4	{1d}						
5	{2b, 1a, 2a, 3a, 4a, 5a}	s2: {1b, …}	s5: {2b, …}	s10: {5b}		6: {2c}	8: {3b, 4b}
6	{2c}		s7: {2d}				
7	{2d}						
8	{3b, 4b}	r3	r3	s9: {4c}			
9	{4c}	r4	r4	r4			
10	{5b}	r5	r5	r5			

   c. The grammar is SLR(1); we know because no entry in the Action/Go-To tables has > 1 action
   d. The grammar is not LR(0). For an LR(0) parser, the columns for `a`, `b`, `y`, and `$` contain the same reductions (if there are any). The entry for state 8 on input symbol y would have a shift-reduce conflict: s9 and r3.
   e. The input `a y b $` causes an error. The partial reverse rightmost derivation `a y` … ← `a` Y `a` ← `a` X `a` ← … shows that $Y \rightarrow$ `y` • and then $X \rightarrow Y$ • get reduced when `y` is followed by `a`. However, since `b` ∈ Follow(X) and Follow(Y), the input `a y b` … would also cause $Y \rightarrow$ `y` • and then $X \rightarrow Y$ • to be reduced, but then a parse error would occur because our state $X \rightarrow$ `a` X • `a` will reject the `b`.

f. (Trace the parser) Omitted.

3. If we use $Y \rightarrow Y$ `y` with an LL(1) parser, the parser diverges (gets infinite recursion or an infinite loop). If we use $Y \rightarrow$ `y` Y with an SLR(1) parser, then we would have a state that includes $Y \rightarrow$ `y` $\bullet$ Y and $Y \rightarrow$ `y` $\bullet$, which is a shift-reduce error.

