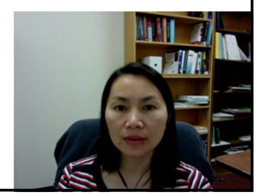
CSCI-C311 Programming Languages

LR(1) Parsers

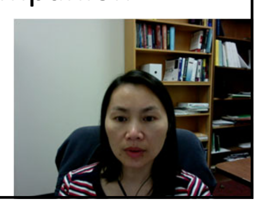
Dr. Hang Dinh



1

Outline and Reading

- After this lecture, you will learn
  - LR(1) parsing
  - Characteristic Finite State Machine (CFSM)
  - SLR(1) parse table
- Reading
  - Scott 4e - Section 2.3.4
  - Scott 4e – Section 2.4.2 Push-down Automata (on the companion site)



2

Overview of LR Parsing

- Recall: **LR** parsing is bottom-up
  - maintain a forest of partially completed subtrees of the parse tree
  - join some subtrees whenever it recognizes the symbols on the right-hand side of some production used in the *right-most derivation* of the input string
  - create a new internal node and make the roots of the joined-together subtrees the children of that node
- Like a table-driven **LL** parser, **LR** parsers are almost always table-driven
  - An **LR** parser uses a big loop in which it repeatedly inspects a two-dimensional table to find out what action to take
  - An **LR** parser also maintains a stack of symbols.

3

Overview of LR Parsing

- Unlike the **LL** parser, the **LR** parser's stack contains a record of what has been seen SO FAR (NOT what is expected)
  - It keeps the roots of partially completed subtrees on the stack
  - When it accepts a new token from the scanner, it shifts the token into the stack
  - When it recognizes that the top few symbols on the stack constitute a right-hand side, it reduces those symbols to their left-hand side by popping them off the stack and pushing the left-hand side in their place.
- Unlike the **LL** parser,
  - the **LR** parsers use the current input token and current parse ***state*** (NOT current nonterminal) to index into the table.

4

LR(1) Grammar Example

- **LR(1)** calculator grammar:

1. *program* ⟶ *stmt_list* `$$`
2. *stmt_list* ⟶ *stmt_list stmt*
3. *stmt_list* ⟶ *stmt*
4. *stmt* ⟶ `id :=` *expr*
5. *stmt* ⟶ `read id`
6. *stmt* ⟶ `write` *expr*
7. *expr* ⟶ *term*
8. *expr* ⟶ *expr add_op term*
9. *term* ⟶ *factor*
10. *term* ⟶ *term mult_op factor*
11. *factor* ⟶ `(` *expr* `)`
12. *factor* ⟶ `id`
13. *factor* ⟶ `number`
14. *add_op* ⟶ `+`
15. *add_op* ⟶ `-`
16. *mult_op* ⟶ `*`
17. *mult_op* ⟶ `/`

5

LR(1) Parsing Example

- Suppose we'll parse the sum-and-average program

```
read A
read B
sum := A + B
write sum
write sum / 2
```

- The key to success is to figure out when the top few symbols on the stack constitute the right-hand side of a production
- The trick is to keep track of:
  - the list of productions we might be "in the middle of" at any particular time. Such a list is called a ***state***, which will also be pushed to stack.
  - an indication (marked by a dot) of where in those productions we might be.
- LR *item:* a production augmented with a dot ·

6

LR(1) Parsing Example: Building Initial State

- Initially, the stack is empty and we are at the beginning of the (only) production for the start symbol *program*

$$program \longrightarrow \bullet\, stmt\_list\ \ \$\$$$

- Since the dot is in front of nonterminal *stmt_list*, we may be about to see the yield of *stmt_list* coming up on the input.
  - Hence, we may be at the beginning of some production with *stmt_list* on the left-hand side:

$$stmt\_list \longrightarrow \bullet\, stmt\_list\ \ stmt$$
$$stmt\_list \longrightarrow \bullet\, stmt$$

7

LR(1) Parsing Example: Building Initial State

- Current list

$$program \longrightarrow \bullet\, stmt\_list\ \ \$\$$$
$$stmt\_list \longrightarrow \bullet\, stmt\_list\ \ stmt$$
$$stmt\_list \longrightarrow \bullet\, stmt$$

- Since the dot is in front of nonterminal *stmt*, we may be at the beginning of some production with *stmt* on the left-hand side

$$stmt \longrightarrow \bullet\, \texttt{id := } expr$$
$$stmt \longrightarrow \bullet\, \texttt{read id}$$
$$stmt \longrightarrow \bullet\, \texttt{write } expr$$

All of these last productions begin with a terminal, no additional items need to be added to the list

8

LR(1) Parsing Example: Building Initial State

- The initial state of the parser:

State 0

program	$\longrightarrow$	$\bullet$ *stmt_list* \$\$
stmt_list	$\longrightarrow$	$\bullet$ *stmt_list* *stmt*
stmt_list	$\longrightarrow$	$\bullet$ *stmt*
stmt	$\longrightarrow$	$\bullet$ `id` `:=` *expr*
stmt	$\longrightarrow$	$\bullet$ `read` `id`
stmt	$\longrightarrow$	$\bullet$ `write` *expr*

basis

closure

  - The original item in the list is called the *basis* of the list.
  - The additional items are its *closure*.

LR(1) Parsing Example: Moving on

- As we shift and reduce,
  - The set of items will change, always indicating which productions may be the right one to use next in the derivation of the input string.
- If we reach a state in which some item has the dot at the end of the right-hand side, then we can reduce by that production
- Otherwise, we must shift
  - If we must shift but the incoming token cannot follow the dot in any item of the current state, an error occurred.
- At the initial state (State 0) in our LR(1) example, we must shift.

LR(1) Parsing Example: Moving from State 0

- At State 0, and incoming token is `read`
  - We shift `read` onto stack
  - Since token `read` comes after the dot in the following item of State 0:

    $stmt \longrightarrow \bullet$`read id`
  - The the new state is:

    State 1 $stmt \longrightarrow$ `read`$\bullet$`id`
- At State 1, we also must shift
  - We shift `id(A)` onto stack and go to the new state:

    State 1' $stmt \longrightarrow$ `read id`$\bullet$

11

LR(1) Parsing Example: Moving on from State 1'

State 1' $stmt \longrightarrow$ `read id`$\bullet$

- At State 1', the dot • is at the end of the right-hand side of an LR item
  - We reduce `read id` to *stmt* (pop two symbols and push *stmt*)
  - Same as: replace `read id` with *stmt* on the input stream then shift *stmt* when at State 0
- What should the new state be?
  - When we shifted `read`, we were in State 0 and the upcoming tokens on the input were `read id` (but we didn't look at them at the time).
  - We now have consumed `read id` and know that they constitute a *stmt*.
  - An item in State 0 was $stmt\_list \longrightarrow \bullet\ stmt$
  - Hence, the new state is **State 0'** $stmt\_list \longrightarrow stmt\ \bullet$

12

LR(1) Parsing Example: Moving on from State 0'

State 0' $\{$ *stmt_list* ⟶ *stmt* •

- At State 0', the dot • is at the end of the right-hand side of an LR item
  - We reduce *stmt* to *stmt_list* (pop *stmt* and push *stmt_list*)
  - Same as: shift *stmt_list* when at State 0
- Two items of State 0 have a *stmt_list* after the dot •

  program ⟶ • *stmt_list* $$

  stmt_list ⟶ • *stmt_list* *stmt*
- New state is State 2:
- Then shift `read` and go to State 1
- Then shift `id(B)` and go to State 1'…

State 2

program ⟶ *stmt_list* • $$

stmt_list ⟶ *stmt_list* • *stmt*

---

stmt ⟶ • `id := ` *expr*

stmt ⟶ • `read id`

stmt ⟶ • `write` *expr*

13

Optimization

- Trivial states: have a single item, with the dot at the end

  State 1' $\{$ *stmt* ⟶ `read id` •

  State 0' $\{$ *stmt_list* ⟶ *stmt* •
- In addition to shift and reduce, allow the "shift-then-reduce" action
  - This serves to eliminate trivial states
  - Example: we can combine the "shift `id(A)`" and "reduce `read id` to *stmt*" into a single shift-then-reduce action → skip state 1'
  - Example: we can combine the "shift *stmt*" and "reduce *stmt* to *stmt_list*" into a single shift-then-reduce action → skip state 0'

14

LR(1) Parse Stack with States

- An **LR** parser keeps track of the states it has traversed by pushing them into the parse stack, along with the grammar symbols.
  - Initialize the stack with the initial state
  - When we shift ***X*** and go to state ***s***,
    - push ***X*** into stack, then push ***s*** into stack
  - When we reduce using production $A \to \alpha$,
    - pop $length(\alpha)$ symbols off the stack together with states we moved through while shifting those symbols
      - These pops expose the state we were in immediately prior to the shifts, allowing us to return to that state and proceed as if we had seen A in the first place.
    - push A into the input stream, so the top of stack is always the current state.

15

LR(1) Parse Stack Example: Unoptimized Version

- SLR(1) parse stack of the sum-and-average program
  - States in the stack are shown in boldface type

Parse Stack (Top is right-most)	Input Stream	Actions
0	`read A read B…`	
0 `read` ***1***	`A read B…`	shift `read` and go to state **1**
0 `read` ***1*** `id` ***1'***	`read B…`	shift `id(A)` and go to state **1'**
0	*stmt* `read B…`	reduce by *stmt* → `read id`
0 *stmt* ***0'***	`read B…`	shift *stmt* and go to state **0'**
0	*stmt_list* `read B…`	reduce by *stmt_list* → *stmt*
0 *stmt_list* ***2***	`read B sum…`	shift *stmt_list* and go to state **2**

16

LL(1) Parse Stack Example: Optimized Version

- SLR(1) parse stack of the sum-and-average program
  - States in the stack are shown in boldface type

Parse Stack (Top is right-most)	Input Stream	Actions
0	`read A read B…`	
0 `read` ***1***	`A read B…`	shift `read` and go to state **1**
0	*stmt* `read B…`	shift `id(A)` and then reduce by *stmt* → `read id`
0	*stmt_list* `read B…`	shift *stmt* and then reduce by *stmt_list* → *stmt*
0 *stmt_list* ***2***	`read B sum…`	shift *stmt_list* and go to state **2**

See full trace of the stack in Figure 2.30 in Scott 4e

17

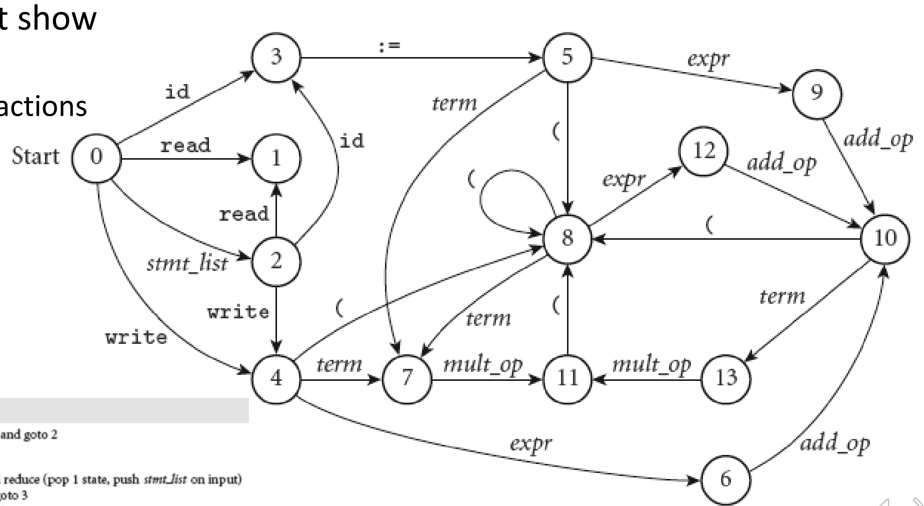
Characteristic Finite State Machine (CFSM)

- The shift rules of an **LR(1)** parser can be thought of as the transition function of a finite automata (like in DFA)
  - The transition for symbol ***X*** (terminal or nonterminal) in input stream moves to a state whose basis consists of items in which the dot has been moved across an ***X*** in the right-hand side, plus whatever items need to be added as closure.
  - These lists are constructed by a bottom-up parser generator in order to build the automaton, but are not needed during parsing.
- Different classes of LR parsers use different type of automata
  - SLR(1) and LALR(1) parsers use *characteristic finite state machine* (CFSM)
  - Full LR parsers use a machine with a much larger number of states.

18

CFSM for LR(1) Calculator Grammar

- This CFSM does not show
  - reduce actions
  - shift-then-reduce actions



	State	Transitions
0.	*program* ⟶ • *stmt_list* `$$`	on *stmt_list* shift and goto 2
	stmt_list ⟶ • *stmt_list* *stmt*	
	stmt_list ⟶ • *stmt*	on *stmt* shift and reduce (pop 1 state, push *stmt_list* on input)
	stmt ⟶ • `id` `:=` *expr*	on `id` shift and goto 3
	stmt ⟶ • `read` `id`	on `read` shift and goto 1
	stmt ⟶ • `write` *expr*	on `write` shift and goto 4
1.	*stmt* ⟶ `read` • `id`	on `id` shift and reduce (pop 2 states, push *stmt* on input)

SLR(1) Parse Table for LR(1) Calculator Grammar

Top-of-stack state	Current input symbol																	
	sl	*s*	*e*	*t*	*f*	*ao*	*mo*	`id`	`lit`	`r`	`w`	`:=`	`(` | `)`	`+`	`-`	`*`	`/`	`$$`
0	s2	b3	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	b5	–	–	–	–	–	–	–	–	–	–	–
2	–	b2	–	[illegible]	[illegible]	[illegible]	[illegible]	s3	–	s1	s4	–	–	–	–	–	–	–	b1
3	–	–	–	[illegible]	[illegible]	[illegible]	[illegible]	–	–	–	–	s5	–	–	–	–	–	–	–
4	–	–	s6	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
5	–	–	s9	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
6	–	–	–	–	–	s10	–	r6	–	r6	r6	–	–	–	b14	b15	–	–	r6
7	–	–	–	–	–	–	s11	r7	–	r7	r7	–	–	r7	r7	r7	b16	b17	r7
8	–	–	s12	s7	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
9	–	–	–	–	–	s10	–	r4	–	r4	r4	–	–	–	b14	b15	–	–	r4
10	–	[illegible]	–	s13	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
11	–	–	–	–	b10	–	–	b12	b13	[illegible]	[illegible]	[illegible]	s8	–	–	–	–	–	–
12	–	–	–	–	–	s10	–	–	–	[illegible]	[illegible]	–	–	b11	b14	b15	–	–	[illegible]
13	–	–	–	–	–	–	s11	r8	–	r8	r8	–	–	r8	r8	r8	b16	b17	[illegible]

shift-then-reduce

shift

reduce

LR(1) Parser Driver

- Different classes of LR(1) differ in the nature of the parse table only.
  - The driver is the same.
- For the optimized version
  - Still need to handle the shift-then-reduce action

```
PUSH start_state
symbol = next_token()            -- get new token
Loop
  s = top_of_stack()             -- current state
  if s = start_state and symbol = start_symbol
       return                    -- success
  if action[s, symbol] = “shift(next_state)”
       PUSH symbol
       PUSH next_state
       symbol = next_token()
  else if action[s, symbol] = “reduce(A→ α)”
       POP 2 * |α| symbols/states
       symbol = A        -- push A into input stream
  else  parse_error()
```

```
else if action[s, symbol] = “shift-then-reduce(A→ α)”
       POP 2 * (|α|-1) symbols/states
       symbol = A
```

21

22