Programming Language Syntax

- LR parsing -

Chapter 2 , Section 2.3

LR Parsing

- LR parsers
  - maintain a forest of subtrees of the parse tree
  - join trees together when recognizing a RHS
  - keeps the roots of subtrees in a stack
  - *shift*: tokens from scanner into the stack
  - *reduce*: when recognizing a RHS, pop it, push LHS
  - discovers a right-most derivation in reverse

Stack contents (roots of partial trees)	Remaining input
ε	`A, B, C;`
`id (A)`	`, B, C;`
`id (A),`	`B, C;`
`id (A), id (B)`	`, C;`
`id (A), id (B),`	`C;`
`id (A), id (B), id (C)`	`;`
`id (A), id (B), id (C)` *id_list_tail*	
`id (A),` *id_list_tail*	
`id (A)` *id_list_tail*	
id_list	

LR Parsing

- Example: LR(1) grammar for calculator language

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* → `/`

- Compare with previous LL(1)
  - left recursive prod. is better
  - keeps operands together

program → *stmt list* `$$`

stmt_list → *stmt stmt_list* | ε

stmt → `id :=` *expr* | `read` *id* | `write` *expr*

expr → *term term_tail*

term_tail → *add op term term_tail* | ε

term → *factor fact_tail*

fact_tail → *mult_op fact fact_tail* | ε

factor → `(` *expr* `)` | `id` | `number`

add_op → `+` | `-`

mult_op → `*` | `/`

LR Parsing

- LR parser
  - recognizes right-hand sides of productions
    - keep track of productions we might be in the middle of
    - and where: represent the location in an RHS by a '•'
  - Example:

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

LR Parsing

- start with:

  program → • *stmt_list* `$$` – this is called an **LR-item**

- '•' in front of *stmt_list* means we may be about to see the yield of *stmt_list,* that is, we could also be at the beginning of a production with *stmt_list* on LHS:

  stmt_list → • *stmt_list stmt*
  stmt_list → • *stmt*

- similarly, we need to include also:

  stmt → • `id :=` *expr*
  stmt → • `read id`
  stmt → • `write` *expr*

- Only terminals follow, so we stop

LR Parsing

- the state we have obtained is:

  program → • *stmt_list* `$$` (the basis) (state 0)
  stmt_list → • *stmt_list stmt* (closure …
  stmt_list → • *stmt* …
  stmt → • `id :=` *expr* …
  stmt → • `read id` …
  stmt → • `write` *expr* …)

- next token: `read` - the next state is:

  stmt → `read` • `id` (empty closure) (state 1)

- next token: `A` - the next state is:

  stmt → `read id` • (state 1')

- '•' at the end means we can reduce
  - what is the new state?

LR Parsing

- replace `read id` with *stmt*

  stmt_list → • *stmt* becomes

  stmt_list → *stmt* • (state 0')

- we reduce again: replace *stmt* with *stmt_list*
- this means shifting a *stmt_list* in state 0:

  program → *stmt_list* • `$$` (basis … (state 2)

  stmt_list → *stmt_list* • *stmt* …)

  stmt → • `id :=` *expr* (closure …

  stmt → • `read id` …

  stmt → • `write` *expr* …)

- Complete states on next slides

LR Parsing

	State	Transitions
0.	*program* ⟶ • *stmt_list* `$$`	on *stmt_list* shift and goto 2
	stmt_list ⟶ • *stmt_list stmt*	
	stmt_list ⟶ • *stmt*	on *stmt* shift and reduce (pop 1 state, push *stmt_list* on input)
	stmt ⟶ • `id :=` *expr*	on `id` shift and goto 3
	stmt ⟶ • `read id`	on `read` shift and goto 1
	stmt ⟶ • `write` *expr*	on `write` shift and goto 4
1.	*stmt* ⟶ `read` • `id`	on `id` shift and reduce (pop 2 states, push *stmt* on input)
2.	*program* ⟶ *stmt_list* • `$$`	on `$$` shift and reduce (pop 2 states, push *program* on input)
	stmt_list ⟶ *stmt_list* • *stmt*	on *stmt* shift and reduce (pop 2 states, 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
3.	*stmt* ⟶ `id` • `:=` *expr*	on `:=` shift and goto 5

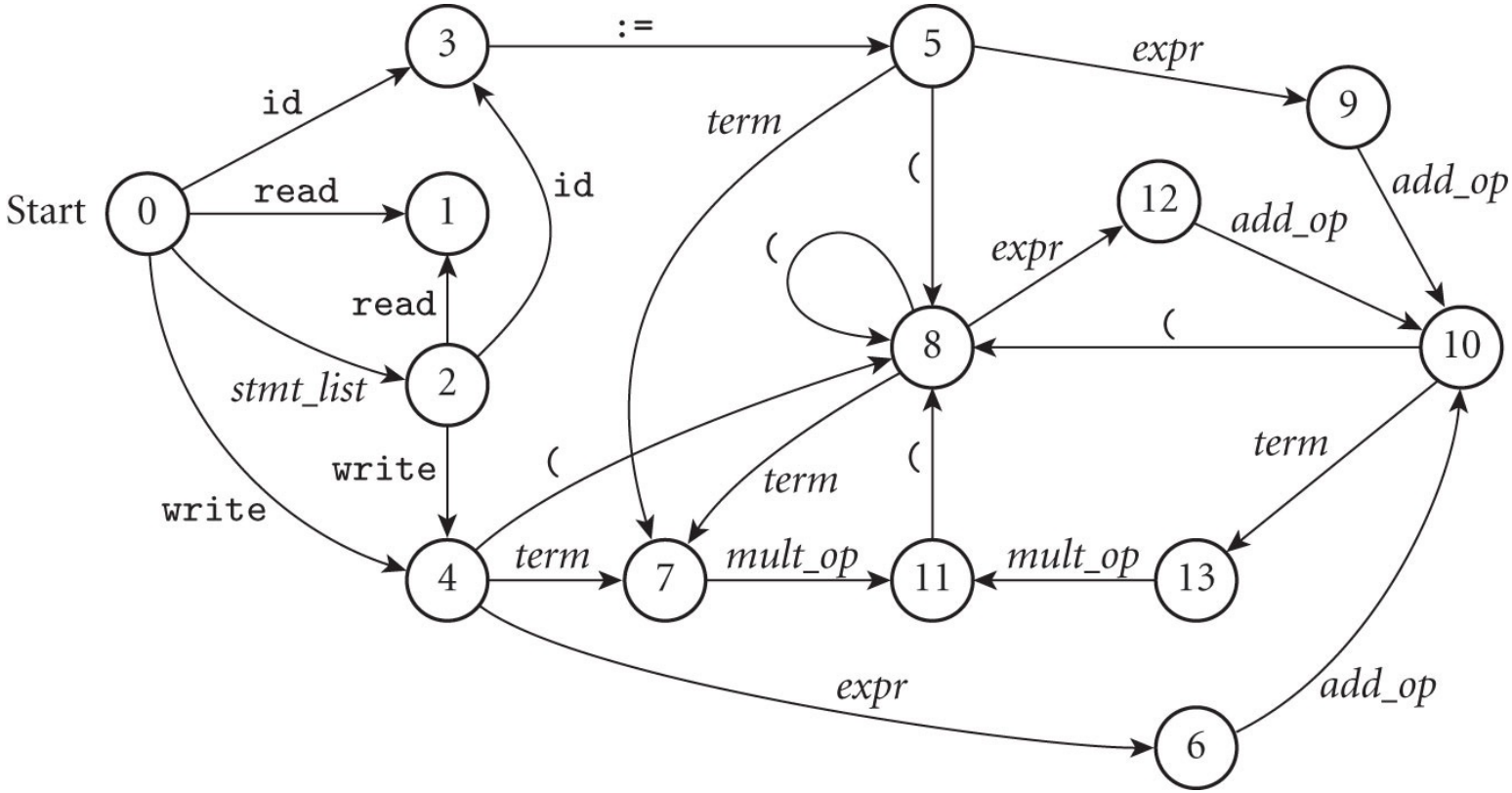
	State	Transitions
4.	*stmt* ⟶ `write` • *expr*	on *expr* shift and goto 6
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)
5.	*stmt* ⟶ `id :=` • *expr*	on *expr* shift and goto 9
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)
6.	*stmt* ⟶ `write` *expr* •	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} reduce (pop 2 states, push *stmt* on input)
	expr ⟶ *expr* • *add_op term*	on *add_op* shift and goto 10
	add_op ⟶ • `+`	on `+` shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ • `-`	on `-` shift and reduce (pop 1 state, push *add_op* on input)

	State	Transitions
7.	*expr* ⟶ *term* • *term* ⟶ *term* • *mult_op factor* ————————	on FOLLOW(*expr*) = {`id`, `read`, `write`, `$$`, `)`, `+`, `-`} reduce (pop 1 state, push *expr* on input) on *mult_op* shift and goto 11
	mult_op ⟶ • `*` *mult_op* ⟶ • `/`	on `*` shift and reduce (pop 1 state, push *mult_op* on input) on `/` shift and reduce (pop 1 state, push *mult_op* on input)
8.	*factor* ⟶ `(` • *expr* `)` ————————	on *expr* shift and goto 12
	expr ⟶ • *term* *expr* ⟶ • *expr add_op term* *term* ⟶ • *factor* *term* ⟶ • *term mult_op factor* *factor* ⟶ • `(` *expr* `)` *factor* ⟶ • `id` *factor* ⟶ • `number`	on *term* shift and goto 7 on *factor* shift and reduce (pop 1 state, push *term* on input) on `(` shift and goto 8 on `id` shift and reduce (pop 1 state, push *factor* on input) on `number` shift and reduce (pop 1 state, push *factor* on input)
9.	*stmt* ⟶ `id :=` *expr* • *expr* ⟶ *expr* • *add_op term* ————————	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} reduce (pop 3 states, push *stmt* on input) on *add_op* shift and goto 10
	add_op ⟶ • `+` *add_op* ⟶ • `-`	on `+` shift and reduce (pop 1 state, push *add_op* on input) on `-` shift and reduce (pop 1 state, push *add_op* on input)

	State	Transitions
10.	*expr* ⟶ *expr add_op* • *term*	on *term* shift and goto 13
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)
11.	*term* ⟶ *term mult_op* • *factor*	on *factor* shift and reduce (pop 3 states, push *term* on input)
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)
12.	*factor* ⟶ `(` *expr* • `)`	on `)` shift and reduce (pop 3 states, push *factor* on input)
	expr ⟶ *expr* • *add_op term*	on *add_op* shift and goto 10
	add_op ⟶ • `+`	on `+` shift and reduce (pop 1 state, push *add_op* on input)
	add_op ⟶ • `-`	on `-` shift and reduce (pop 1 state, push *add_op* on input)
13.	*expr* ⟶ *expr add_op term* •	on FOLLOW(*expr*) = {`id`, `read`, `write`, `$$`, `)`, `+`, `-`} reduce (pop 3 states, push *expr* on input)
	term ⟶ *term* • *mult_op factor*	on *mult_op* shift and goto 11
	mult_op ⟶ • `*`	on `*` shift and reduce (pop 1 state, push *mult_op* on input)
	mult_op ⟶ • `/`	on `/` shift and reduce (pop 1 state, push *mult_op* on input)

LR Parsing

- LL(1) parser: decides using nonterminal + token
- LR(1) parser: decides using state + token
  - CFSM: Characteristic Finite State Machine
  - Almost always table-driven



LR Parsing

- Parse table `parse_tab`
  - shift (s) followed by state
  - reduce (r), shift + reduce (b) followed by production

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	–	–	–	–	–	s3	–	s1	s4	–	–	–	–	–	–	–	b1
3	–	–	–	–	–	–	–	–	–	–	–	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	–	–	–	s13	b9	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
11	–	–	–	–	b10	–	–	b12	b13	–	–	–	s8	–	–	–	–	–	–
12	–	–	–	–	–	s10	–	–	–	–	–	–	–	b11	b14	b15	–	–	–
13	–	–	–	–	–	–	s11	r8	–	r8	r8	–	–	r8	r8	r8	b16	b17	r8

LR Parsing

- Algorithm
- uses the `parse_tab` (previous slide) and `prod_tab` (not shown)
- example after algorithm for:

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

```
state = 1 . . number_of_states
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions
action_rec = record
        action : (shift, reduce, shift_reduce, error)
        new_state : state
        prod : production

parse_tab : array [symbol, state] of action_rec
prod_tab : array [production] of record
        lhs : symbol
        rhs_len : integer
-- these two tables are created by a parser generator tool

parse_stack : stack of record
        sym : symbol
        st : state
```

LR Parsing

```
parse_stack.push(⟨null, start_state⟩)
cur_sym : symbol := scan()                          -- get new token from scanner
loop
    cur_state : state := parse_stack.top().st   -- peek at state at top of stack
    if cur_state = start_state and cur_sym = start_symbol
        return                                      -- success!
    ar : action_rec := parse_tab[cur_state, cur_sym]
    case ar.action
        shift:
            parse_stack.push(⟨cur_sym, ar.new_state⟩)
            cur_sym := scan()                       -- get new token from scanner
        reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len)
        shift_reduce:
            cur_sym := prod_tab[ar.prod].lhs
            parse_stack.pop(prod_tab[ar.prod].rhs_len−1)
        error:
            parse_error
```

LR Parsing

Parse stack	Input stream	Comment
0	`read A read B` ...	
0 `read` *1*	`A read B` ...	shift `read`
0	*stmt* `read B` ...	shift `id(A)` & reduce by *stmt* ⟶ `read id`
0	*stmt_list* `read B` ...	shift *stmt* & reduce by *stmt_list* ⟶ *stmt*
0 *stmt_list* *2*	`read B sum` ...	shift *stmt_list*
0 *stmt_list* *2* `read` *1*	`B sum :=` ...	shift `read`
0 *stmt_list* *2*	*stmt* `sum :=` ...	shift `id(B)` & reduce by *stmt* ⟶ `read id`
0	*stmt_list* `sum :=` ...	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 *stmt_list* *2*	`sum := A` ...	shift *stmt_list*
0 *stmt_list* *2* `id` *3*	`:= A +` ...	shift `id(sum)`
0 *stmt_list* *2* `id` *3* `:=` *5*	`A + B` ...	shift `:=`
0 *stmt_list* *2* `id` *3* `:=` *5*	*factor* `+ B` ...	shift `id(A)` & reduce by *factor* ⟶ `id`
0 *stmt_list* *2* `id` *3* `:=` *5*	*term* `+ B` ...	shift *factor* & reduce by *term* ⟶ *factor*
0 *stmt_list* *2* `id` *3* `:=` *5* *term* *7*	`+ B write` ...	shift *term*
0 *stmt_list* *2* `id` *3* `:=` *5*	*expr* `+ B write` ...	reduce by *expr* ⟶ *term*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9*	`+ B write` ...	shift *expr*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9*	*add_op* `B write` ...	shift `+` & reduce by *add_op* ⟶ `+`
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9* *add_op* *10*	`B write sum` ...	shift *add_op*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9* *add_op* *10*	*factor* `write sum` ...	shift `id(B)` & reduce by *factor* ⟶ `id`
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9* *add_op* *10*	*term* `write sum` ...	shift *factor* & reduce by *term* ⟶ *factor*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9* *add_op* *10* *term* *13*	`write sum` ...	shift *term*
0 *stmt_list* *2* `id` *3* `:=` *5*	*expr* `write sum` ...	reduce by *expr* ⟶ *expr add_op term*
0 *stmt_list* *2* `id` *3* `:=` *5* *expr* *9*	`write sum` ...	shift *expr*
0 *stmt_list* *2*	*stmt* `write sum` ...	reduce by *stmt* ⟶ `id :=` *expr*
0	*stmt_list* `write sum` ...	shift *stmt* & reduce by *stmt_list* ⟶ *stmt*

LR Parsing

Parse stack	Input stream	Comment
0 stmt_list 2	`write sum ...`	shift *stmt_list*
0 stmt_list 2 `write` *4*	`sum write sum ...`	shift `write`
0 stmt_list 2 `write` *4*	*factor* `write sum ...`	shift `id(sum)` & reduce by *factor* ⟶ `id`
0 stmt_list 2 `write` *4*	*term* `write sum ...`	shift *factor* & reduce by *term* ⟶ *factor*
0 stmt_list 2 `write` *4 term 7*	`write sum ...`	shift *term*
0 stmt_list 2 `write` *4*	*expr* `write sum ...`	reduce by *expr* ⟶ *term*
0 stmt_list 2 `write` *4 expr 6*	`write sum ...`	shift *expr*
0 stmt_list 2	*stmt* `write sum ...`	reduce by *stmt* ⟶ `write` *expr*
0	*stmt_list* `write sum ...`	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 stmt_list 2	`write sum / ...`	shift *stmt_list*
0 stmt_list 2 `write` *4*	`sum / 2 ...`	shift `write`
0 stmt_list 2 `write` *4*	*factor* `/ 2 ...`	shift `id(sum)` & reduce by *factor* ⟶ `id`
0 stmt_list 2 `write` *4*	*term* `/ 2 ...`	shift *factor* & reduce by *term* ⟶ *factor*
0 stmt_list 2 `write` *4 term 7*	`/ 2 $$`	shift *term*
0 stmt_list 2 `write` *4 term 7*	*mult_op* `2 $$`	shift `/` & reduce by *mult_op* ⟶ `/`
0 stmt_list 2 `write` *4 term 7 mult_op 11*	`2 $$`	shift *mult_op*
0 stmt_list 2 `write` *4 term 7 mult_op 11*	*factor* `$$`	shift `number(2)` & reduce by *factor* ⟶ `number`
0 stmt_list 2 `write` *4*	*term* `$$`	shift *factor* & reduce by *term* ⟶ *term mult_op factor*
0 stmt_list 2 `write` *4 term 7*	`$$`	shift *term*
0 stmt_list 2 `write` *4*	*expr* `$$`	reduce by *expr* ⟶ *term*
0 stmt_list 2 `write` *4 expr 6*	`$$`	shift *expr*
0 stmt_list 2	*stmt* `$$`	reduce by *stmt* ⟶ `write` *expr*
0	*stmt_list* `$$`	shift *stmt* & reduce by *stmt_list* ⟶ *stmt_list stmt*
0 stmt_list 2	`$$`	shift *stmt_list*
0	*program*	shift `$$` & reduce by *program* ⟶ *stmt_list* `$$`
[done]		

LR Parsing

- *Shift/reduce conflict*
  - two items in a state:
    - one with '•' in front of terminal (shift)
    - one with '•' at the end (reduce)
  - SLR (simple LR)
    - conflict can be resolved using `FIRST` and `FOLLOW`
  - Example: state 6
    - *stmt* → `write` *expr* •
    - *expr* → *expr* • *add_op term*
    - `FIRST`(*add_op*) ∩ `FOLLOW`(*stmt*) = ∅

LL(1) vs SLR(1)

- LL(1)
  - For any productions $A \rightarrow u \mid v$:
    - $\texttt{FIRST}(u) \cap \texttt{FIRST}(v) = \emptyset$
    - at most one of u and v can derive the empty string ε
    - if $v \Rightarrow^* \varepsilon$, then $\texttt{FIRST}(u) \cap \texttt{FOLLOW}(A) = \emptyset$
- SLR(1)
  - No shift/reduce conflict: cannot have in the same state:

    $A \rightarrow u \bullet xv$, $B \rightarrow w \bullet$, with $x \in \texttt{FOLLOW}(B)$
  - No reduce/reduce conflict: cannot have in the same state:

    $A \rightarrow u \bullet$, $B \rightarrow v \bullet$, with $\texttt{FOLLOW}(A) \cap \texttt{FOLLOW}(B) \neq \emptyset$

Unambiguous vs LL(1) vs SLR(1)