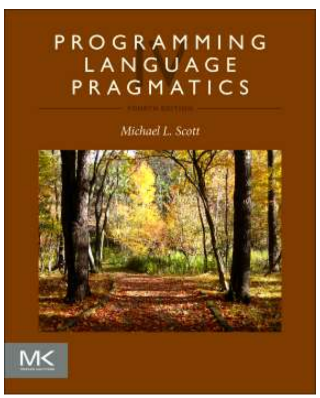
Bottom-Up LR Parsing

17-363/17-663: Programming Language Pragmatics



Reading: PLP section 2.3

Top-Down vs. Bottom-Up Parsing

- Top-Down/LL Parsing Intuition

program

Start trying to parse a program

stmt_list $$$

Based on lookahead, refine to *stmt_list* then to *stmt stmt_list*

stmt stmt_list $$$

Stack tracks predicted future parsing

. . .

- Bottom-Up/LR Parsing Intuition

`read A`

Start by shifting a few tokens

stmt

Reduce tokens to a *stmt*, then to a *stmt_list*

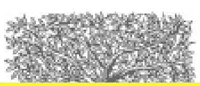
stmt_list

stmt_list `read B`

Continue to shift and reduce tokens tokens to recognize another *stmt*

stmt_list stmt

Stack shows what constructs have been recognized so far



Example Program and SLR(1) Grammar

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

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* ⟶ `/`

Modeling a Parse with LR Items

- Initial parse state captured by an *item*

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

  – includes start symbol, production, and current location

- What we see next might be inside *stmt_list*
  – So we expand *stmt_list* and add more items to the set:

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

Modeling a Parse with LR Items

- We can likewise expand *stmt* to get the item set:

$$
\begin{aligned}
\textit{program} &\longrightarrow \bullet\ \textit{stmt\_list}\ \texttt{\$\$} \\
\textit{stmt\_list} &\longrightarrow \bullet\ \textit{stmt\_list}\ \textit{stmt} \\
\textit{stmt\_list} &\longrightarrow \bullet\ \textit{stmt} \\
\textit{stmt} &\longrightarrow \bullet\ \texttt{id := }\ \textit{expr} \\
\textit{stmt} &\longrightarrow \bullet\ \texttt{read id} \\
\textit{stmt} &\longrightarrow \bullet\ \texttt{write}\ \textit{expr}
\end{aligned}
$$

- This is an SLR parser *state*
  - We'll call it state 0

Modeling a Parse with LR Items

- Our starting stack has state 0 on it:

0

- Input: `read A read B …`

program ⟶ • *stmt_list* $$
stmt_list ⟶ • *stmt_list* *stmt*
stmt_list ⟶ • *stmt*
stmt ⟶ • `id := ` *expr*
stmt ⟶ • `read id`
stmt ⟶ • `write` *expr*

- From state 0, we *shift* `read` onto the stack and move to state 1:

0 `read` 1

- State 1 represents the following item:

stmt ⟶ `read` • `id`

ELSEVIER

Modeling a Parse with LR Items

- stack / item: 0 `read` 1
- input: `A read B ...`

stmt ⟶ **read** • **id**

- From state 1, we shift `id` onto the stack
- stack / item: 0 `read` 1 `id` 1'
- input: `read B ...`

stmt ⟶ **read id** •

- Now we reduce to *stmt*, and put *stmt* into the input
- stack / item: 0
- input: *stmt* `read B ...`

program ⟶ • *stmt_list* **$$**
stmt_list ⟶ • *stmt_list stmt*
stmt_list ⟶ • *stmt*
stmt ⟶ • **id** := *expr*
stmt ⟶ • **read id**

Modeling a Parse with LR Items

- stack / item: 0
- input: *stmt* `read B …`

program ⟶ • *stmt_list* $$
stmt_list ⟶ • *stmt_list* *stmt*
stmt_list ⟶ • *stmt*
stmt ⟶ • `id := ` *expr*
stmt ⟶ • `read id`
stmt ⟶ • `write` *expr*

- We now shift *stmt*
- stack / item: 0 *stmt* 0'
- input: `read B …`

stmt_list ⟶ *stmt* •

- Next we reduce to *stmt_list*
- stack / item: 0
- input: *stmt_list* `read B …`

program ⟶ • *stmt_list* $$
stmt_list ⟶ • *stmt_list* *stmt*
stmt_list ⟶ • *stmt*
stmt ⟶ • `id := ` *expr*
stmt ⟶ • `read id`
stmt ⟶ • `write` *expr*

Modeling a Parse with LR Items

- stack / item: 0
- input: *stmt_list* `read B …`

program ⟶ • *stmt_list* `$$`
stmt_list ⟶ • *stmt_list* *stmt*
stmt_list ⟶ • *stmt*
stmt ⟶ • `id := ` *expr*
stmt ⟶ • `read id`
stmt ⟶ • `write` *expr*

- Now we shift *stmt_list*

- stack / item: 0 *stmt_list* 2
- input: `read B …`

program ⟶ *stmt_list* • `$$`
stmt_list ⟶ *stmt_list* • *stmt*
stmt ⟶ • `id := ` *expr*
stmt ⟶ • `read id`
stmt ⟶ • `write` *expr*

The Characteristic Finite State Machine (CFSM)

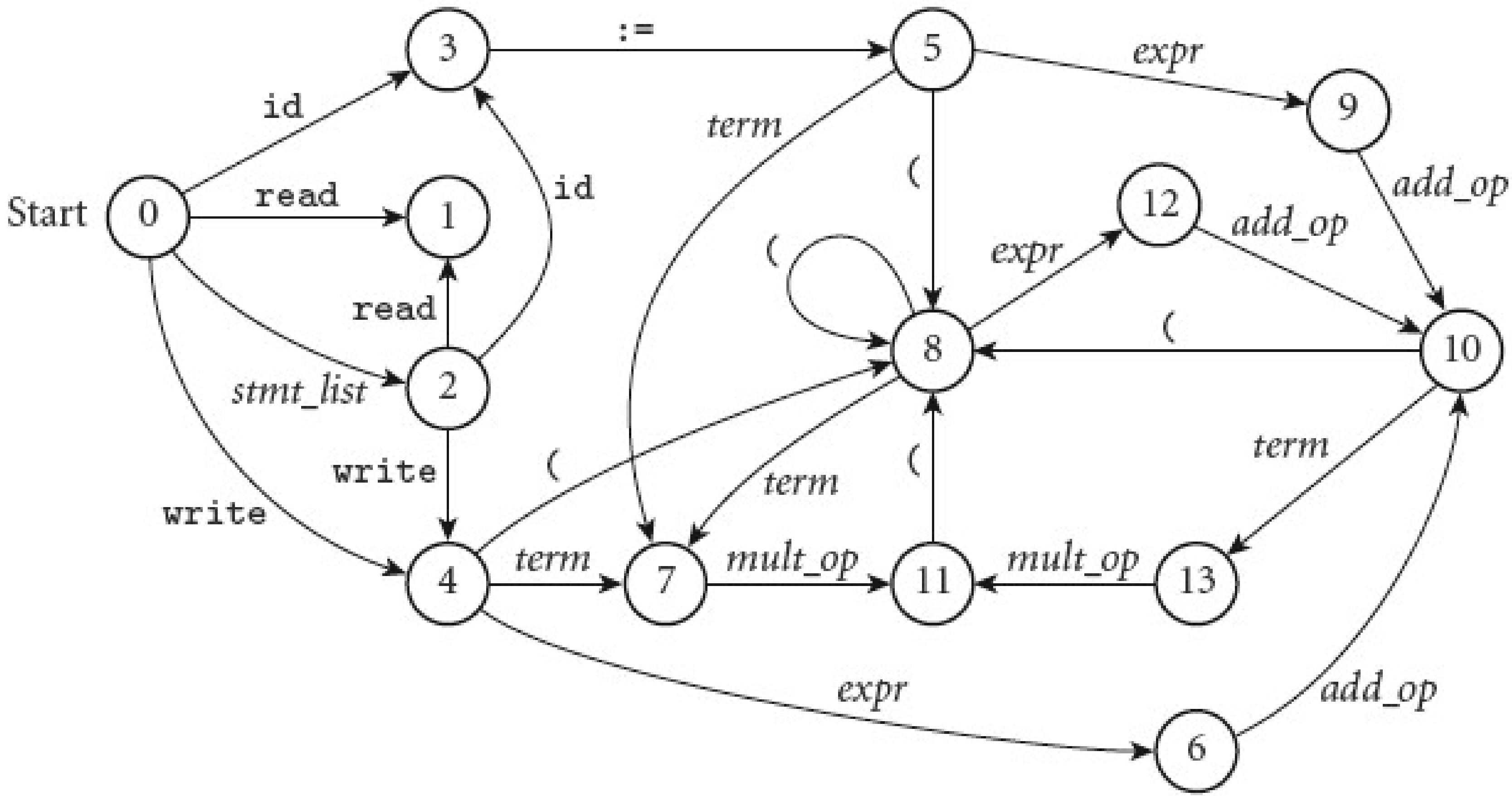


Figure 2.27 Pictorial representation of the CFSM of Figure 2.26. Reduce actions are not shown.

There are also shift-reduce actions. So our states 0', 1' aren't shown here: they are "in between" states within a shift-reduce action

The CFSM as a Table

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

Figure 2.28 SLR(1) parse table for the calculator language. Table entries indicate whether to shift (s), reduce (r), or shift and then reduce (b). The accompanying number is the new state when shifting, or the production that has been recognized when (shifting and) reducing. Production numbers are given in Figure 2.25. Symbol names have been abbreviated for the sake of formatting. A dash indicates an error. An auxiliary table, not shown here, gives the left-hand-side symbol and right-hand-side length for each production.

A Detailed Explanation of the CFSM

	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
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)

A Detailed Explanation of the CFSM

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 `$$` shift and reduce (pop 2 states, push *program* on input)
	stmt_list ⟶ *stmt_list* • *stmt*	on *stmt* shift and reduce (pop 2 states, push *stmt_list* on input)
	stmt ⟶ • `id` `:=` *expr*	on `id` shift and goto 3
	stmt ⟶ • `read` `id`	on `read` shift and goto 1
	stmt ⟶ • `write` *expr*	on `write` shift and goto 4
3.	*stmt* ⟶ `id` • `:=` *expr*	on `:=` shift and goto 5
4.	*stmt* ⟶ `write` • *expr*	on *expr* shift and goto 6
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)
5.	*stmt* ⟶ `id` `:=` • *expr*	on *expr* shift and goto 9
	expr ⟶ • *term*	on *term* shift and goto 7
	expr ⟶ • *expr add_op term*	
	term ⟶ • *factor*	on *factor* shift and reduce (pop 1 state, push *term* on input)
	term ⟶ • *term mult_op factor*	
	factor ⟶ • `(` *expr* `)`	on `(` shift and goto 8
	factor ⟶ • `id`	on `id` shift and reduce (pop 1 state, push *factor* on input)
	factor ⟶ • `number`	on `number` shift and reduce (pop 1 state, push *factor* on input)
6.	*stmt* ⟶ `write` *expr* •	on FOLLOW(*stmt*) = {`id`, `read`, `write`, `$$`} 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)

Figure 2.26 CFSM for the calculator grammar (Figure 2.25). Basis and closure items in each state are separated by a horizontal rule. Trivial reduce-only states have been eliminated by use of "shift and reduce" transitions. *(continued)*

ELSEVIER

A Detailed Explanation of the CFSM

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

Figure 2.26 *(continued)*

Exercise: LR Parsing

- Assume you are in parsing state 0
  and the token stream is `write sum / 2`
- Show how the parse stack changes as the token stream is consumed
- We’ll do the first two actions together

Exercise: LR Parsing

Parse stack	Input stream	Action
0	`write sum / 2 $$`	(starting configuration)

Exercise: LR Parsing

Parse stack	Input stream	Action
0	`write sum / 2 $$`	(starting configuration)
0 `write` 4	`sum / 2 $$`	shift `write`
0 `write` 4	*factor* / 2 $$	shift `id(sum)` and reduce by *factor* → `id`
0 `write` 4	*term* / 2 $$	shift *factor* and reduce by *term* → *factor*
0 `write` 4 *term* 7	/ 2 $$	shift *term* and reduce by *term* → *factor*
0 `write` 4 *term* 7	*mult_op* 2 $$	shift / and reduce by *mult_op* → /
… 4 *term* 7 *mult_op* 11	2 $$	shift *mult_op*
… 4 *term* 7 *mult_op* 11	*factor* $$	shift 2 and reduce by *factor* → `num_lit(2)`
0 `write` 4	*term* $$	shift *factor* and reduce by *term* → *term mult_op factor*
0 `write` 4 *term* 7	$$	shift *term*
0 `write` 4	*expr* $$	*reduce by expr* → *term*
0 `write` 4 *expr 6*	$$	shift *expr*

Exercise: LR Parsing

Parse stack	Input stream	Action
…		
0 `write` 4 *expr 6*	$$	shift *expr*
0	*stmt* $$	reduce by *stmt* → `write` *expr*
0	*stmt_list* $$	shift *stmt* and reduce by *stmt_list* → *stmt*
0 *stmt_list* 2	$$	shift *stmt_list*
0	program	shift $$ and reduce by *program* → *stmt_list* $$
[done]		

Parsing if-then-else Statements

- A famous parsing challenge (from Algol) involves if-then-else, where else is optional:

stmt ::= `if` *exp* `then` *stmt*

  | `if` *exp* `then` *stmt* `else` *stmt*

- Consider the phrase:

`if` *exp* `then` `if` *exp* `then` *stmt* `else` *stmt*

- Which `then` does the `else` belong to?

Shift/Reduce Conflicts

- This is a shift-reduce conflict

`if` *exp* `then` `if` *exp* `then` *stmt* `.` `else` *stmt*

- When the `else` appears
  - we can *shift*, treating it as part of the inner `if` statement, or
  - we can *reduce* the inner `if` statement,
    treating the `else` as part of the outer `if` statement
- How to solve?
  - Many existing tools prioritize shift over reduce
    - This corresponds to the traditional solution to the `if` problem

Shift/Reduce Conflicts

- This is a shift-reduce conflict

`if` *exp* `then if` *exp* `then` *stmt* `.` `else` *stmt*

- When the `else` appears
  - we can *shift*, treating it as part of the inner `if` statement, or
  - we can *reduce* the inner `if` statement,
    treating the `else` as part of the outer `if` statement
- How to solve?
  - Many existing tools prioritize shift over reduce
  - You can declare productions with *precedence*
    - E.g. giving the if-then-else production higher precedence than the if-then production

ELSEVIER

Shift/Reduce Conflicts

- This is a shift-reduce conflict

`if` *exp* `then` `if` *exp* `then` *stmt* `.` `else` *stmt*

- When the `else` appears
  - we can *shift*, treating it as part of the inner `if` statement, or
  - we can *reduce* the inner `if` statement,
    treating the `else` as part of the outer `if` statement
- How to solve?
  – Many existing tools prioritize shift over reduce
  – You can declare productions with *precedence*
  – Rewrite the grammar to make it LR(1)

An LR(0) If-Then-Else Grammar

stmt → *balanced_stmt* | *unbalanced_stmt*

balanced_stmt → `if` *cond* `then` *balanced_stmt* `else` *balanced_stmt*
| *other_stuff*

unbalanced_stmt → `if` *cond* `then` *stmt*
| `if` *cond* `then` *balanced_stmt* `else` *unbalanced_stmt*

Invariant: *balanced_stmt*s may be inside *unbalanced_stmt*s

– but not vice versa

Unfortunately, this grammar is LR(0) but not LL(0)

– Have to use precedence in LL parsers
or add custom code to a recursive-descent parser

Advice for Managing Conflicts

- Start with a simple grammar that works
  - Even if it doesn't parse the whole language
- Add constructs incrementally
  - Save and compile the grammar after each change
  - If there was a conflict, adjust the grammar to avoid it before proceeding
  - Error messages are sometimes helpful, but don't actually try to understand/reconstruct the LR parsing table (even experts typically don't go this route)

Connections to Theory

- A scanner is a Deterministic Finite Automaton (DFA)
  - it can be specified with a state diagram
- An LL or LR parser is a Pushdown Automaton (PDA)
  - a PDA can be specified with a state diagram and a stack
    - the state diagram looks just like a DFA state diagram, except the arcs are labeled with <input symbol, top-of-stack symbol> pairs, and in addition to moving to a new state the PDA has the option of pushing or popping a finite number of symbols onto/off the stack
  - For LL(1) parsers the state machine has only two states: processing and accepted
    - All the action is in the input symbol and top of stack
  - LR(1) parsers are richer (and more expressive)

Error Reporting

- Error reporting is relatively simple
- If you get a token for which there's no entry in the current parsing state / top of stack element, signal an error
  - Can tell the user what tokens *would* be OK here

Error Recovery

- Nice to report more than one error to the user
  - Rather than stopping after the first one
- Simple idea: Panic mode
  - In C-like languages, semicolons are good recovery spots
  - So on an error:
    - read tokens until you get to a semicolon
    - discard the parser's stack (predictions in an LL parser, states in an LR parser) until you come to a production that has a semicolon
    - assume you've parsed the semicolon-containing construct, and continue parsing
  - There are ways to do substantially better – see the online supplement to the textbook

Other Parsing Tools

- Generalized LR (GLR) parser generators
  - Accept any grammar – even ambiguous ones!
    - This can be good if you have grammars written by nonexperts, as in SASyLF
    - But for a compiler-writer it is dangerous—you may not even know your grammar is ambiguous, and then your poor users get ambiguity errors when the parser runs
  - Works like an LR parser, but on ambiguity considers all possible parses in parallel
  - Still O(n) if the grammar is LR (or "close")

ELSEVIER

Other Parsing Tools

- Parsing Expression Grammar (PEG) parser generators
  - Sidestep ambiguity by always favoring the first production
  - Same danger as GLR parsers – you may not know your grammar is ambiguous
  - Still used some in practice (e.g. in Python)
    - About as efficient as LL or LR in practice
    - Like LR, PEG grammars can be cleaner than LL grammars
    - Requires extreme care to get right – must think algorithmically instead of declaratively
      - Guido van Rossum, the developer of Python, saw this as an advantage