LR5: Precedence and Associativity

# LR Parsing

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

# S/R & ambiguous grammars

- Lx(k) Grammar vs. Language
  - Grammar is Lx(k) if it can be parsed by Lx(k) method – according to criteria that is specific to the method.
  - A Lx(k) grammar may or may not exist for a language.
- Even if a given grammar is not LR(k), shift/reduce parser can *sometimes* handle them by accounting for ambiguities
  - Example: ‘dangling’ else
    - Preferring shift to reduce means matching inner ‘if’

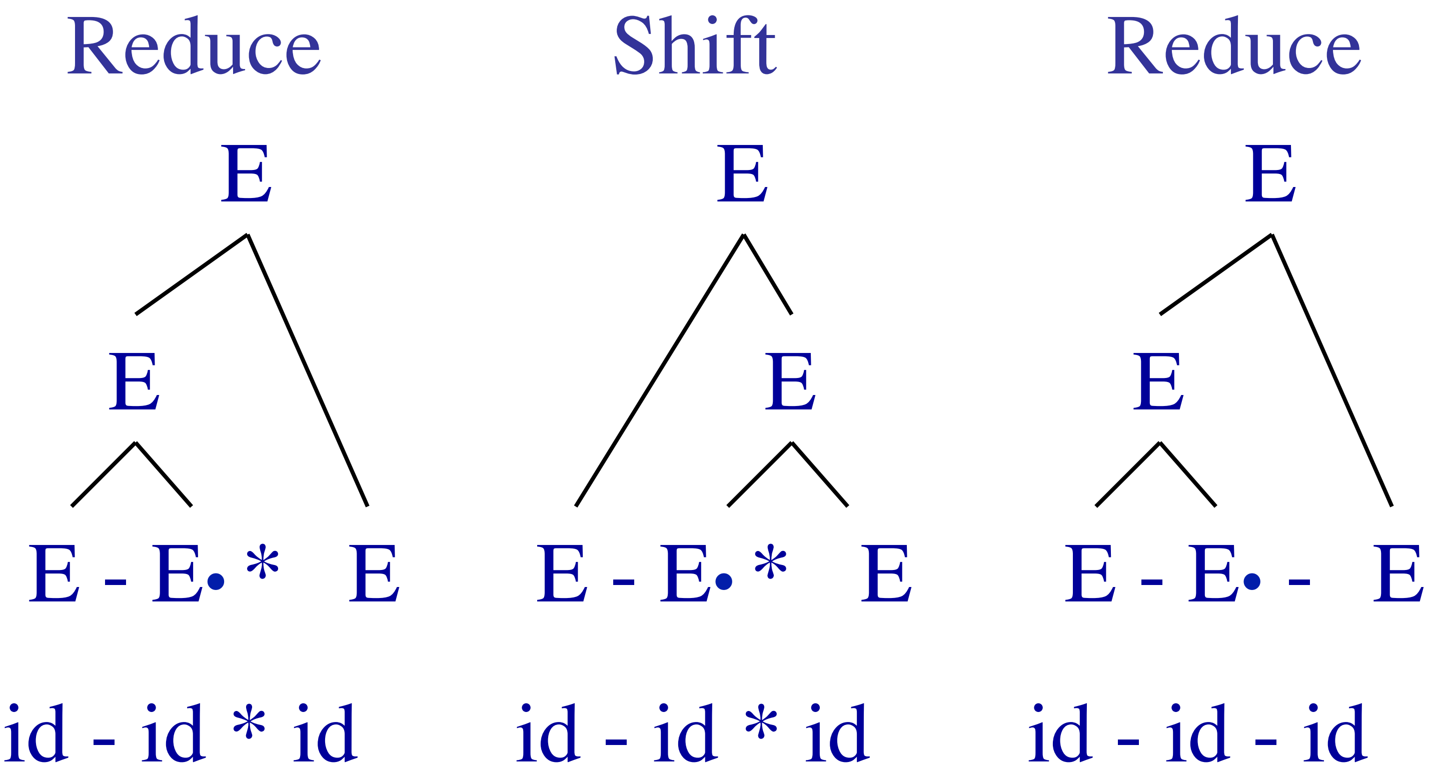
# Dangling ‘else’

1. S → if E then S
2. S → if E then S else S

- Viable prefix “if E then if E then S”
  - Then read else
- Shift “else” (means go for 2)
- Reduce (reduce using production #1)
- NB: dangling else as written above is ambiguous
  - NB: Ambiguity can be resolved, but there’s still no LR(k) grammar

# Precedence & Associativity

- Consider $E \rightarrow E - E \mid E * E \mid id$



Reduce

E

E

E - E• * E

id - id * id

Shift

E

E

E - E• * E

id - id * id

Reduce

E

E

E - E• - E

id - id - id

# Precedence Relations

- Let A → *w* be a rule in the grammar
- And *b* is a terminal
- In some state *q* of the LR(1) parser there is a shift-reduce conflict:
  - either reduce with A → *w* or shift on *b*
- Write down a rule, either:

  A → *w*, < *b* or A → *w*, > *b*

# Precedence Relations

- A → *w*, < *b* means rule has less precedence and so we shift if we see *b* in the lookahead
- A → *w*, > *b* means rule has higher precedence and so we reduce if we see *b* in the lookahead
- If there are multiple terminals with shift-reduce conflicts, then we list them all:

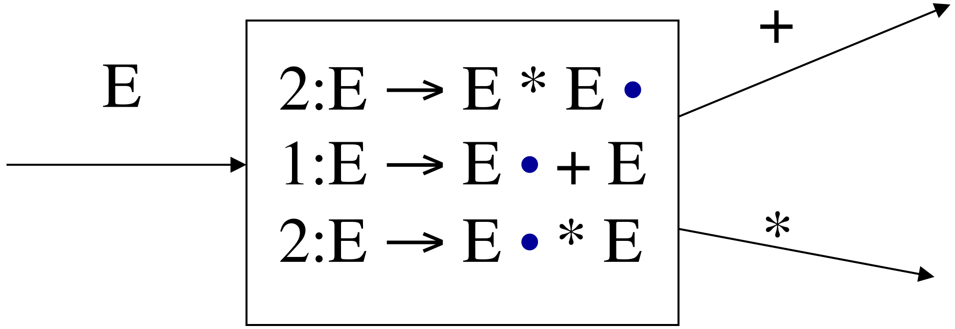
  A → *w*, > *b*, < *c*, > *d*

# Precedence Relations

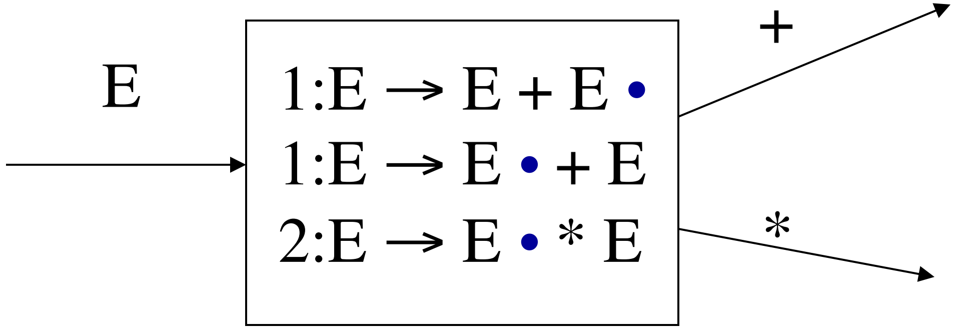
- Consider the grammar
  $E \rightarrow E + E \mid E * E \mid ( E ) \mid a$
- Assume left-association so that E+E+E is interpreted as (E+E)+E
- Assume multiplication has higher precedence than addition
- Then we can write precedence rules/relns:
  $E \rightarrow E + E$, > +, < *
  $E \rightarrow E * E$, > +, > *

# Precedence & Associativity

10:



7:



E → E + E, > +, < *
E → E * E, > +, > *

| | + | * |
|---|---|---|
| 7 | R1 | Shift |
| 10 | R2 | R2 |

# Handling S/R & R/R Conflicts

- Have a conflict?
  - No? – Done, grammar is compliant.
- Already using most powerful parser available?
  - No? – Upgrade and goto 1
- Can the grammar be rearranged so that the conflict disappears?
  - While preserving the language!

# Conflicts revisited (cont'd)

- Can the grammar be rearranged so that the conflict disappears?
  - Yes: Is it worth it?
    - Yes, resolve conflict.
    - No: live with default or specified conflict resolution (precedence, associativity)

# Compiler (parser) compilers

- Rather than build a parser for a particular grammar (e.g. recursive descent), write down a grammar as a text file
- Run through a compiler compiler which produces a parser for that grammar
- The parser is a program that can be compiled and accepts input strings and produces user-defined output

# Compiler (parser) compilers

- For LR parsing, all it needs to do is produce action/goto table
  - Yacc (yet another compiler compiler) was distributed with Unix, the most popular tool. Uses LALR(1).
  - Many variants of yacc exist for many languages
- As we will see later, translation of the parse tree into machine code (or anything else) can also be written down with the grammar
- Handling errors and interaction with the lexical analyzer have to be precisely defined

# Parsing - Summary

- Top-down vs. bottom-up
- Lookahead: FIRST and FOLLOW sets
- LL(1) – Parsing: $O(n)$ time complexity
  - recursive-descent and table-driven predictive parsing
- LR(k) – Parsing : $O(n)$ time complexity
  - LR(0), SLR(1), LR(1), LALR(1)
- Resolving shift/reduce conflicts
  - using precedence, associativity