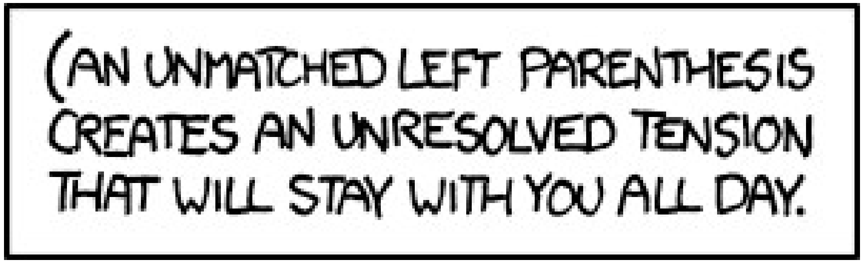
CS 432
Fall 2018

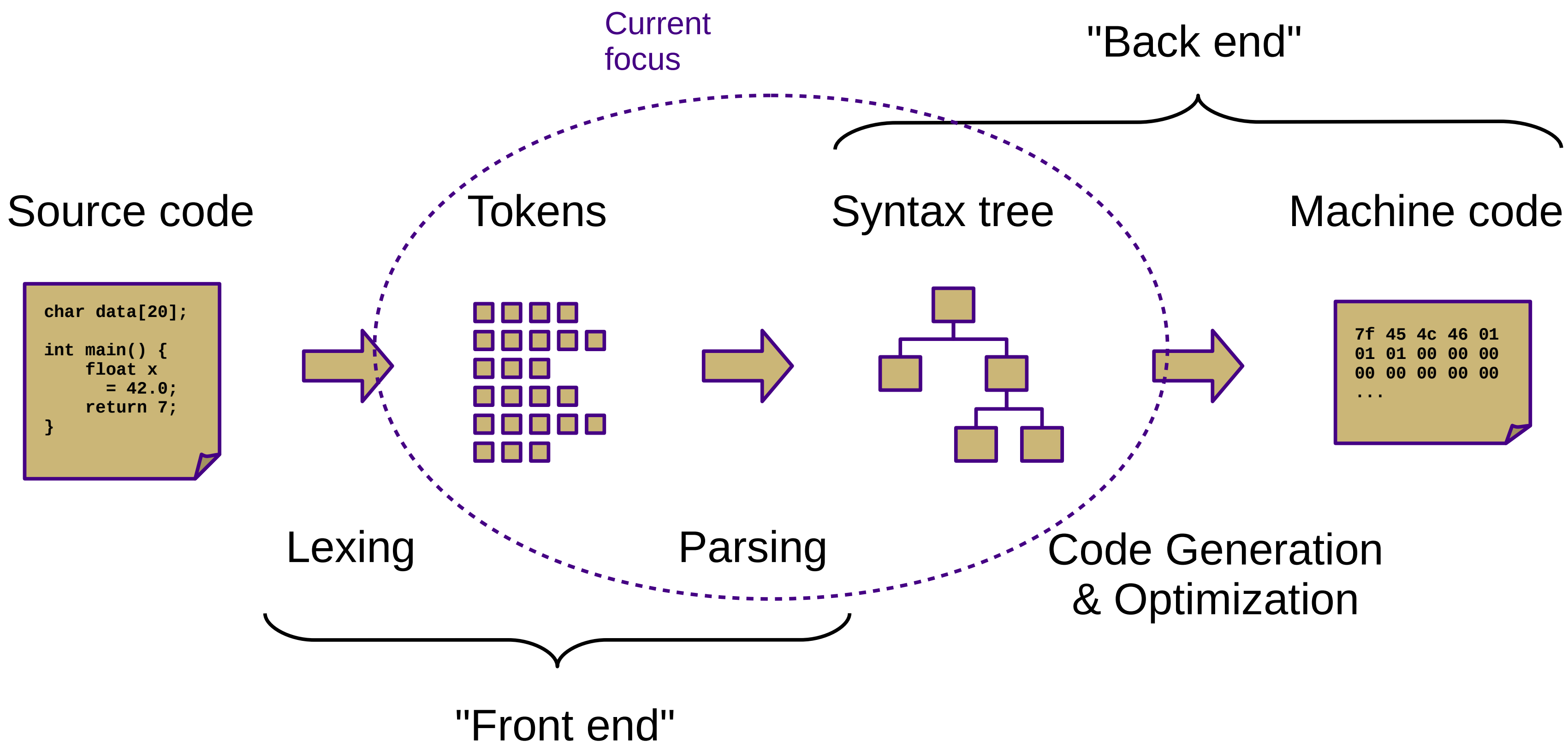
Mike Lam, Professor



https://xkcd.com/859/

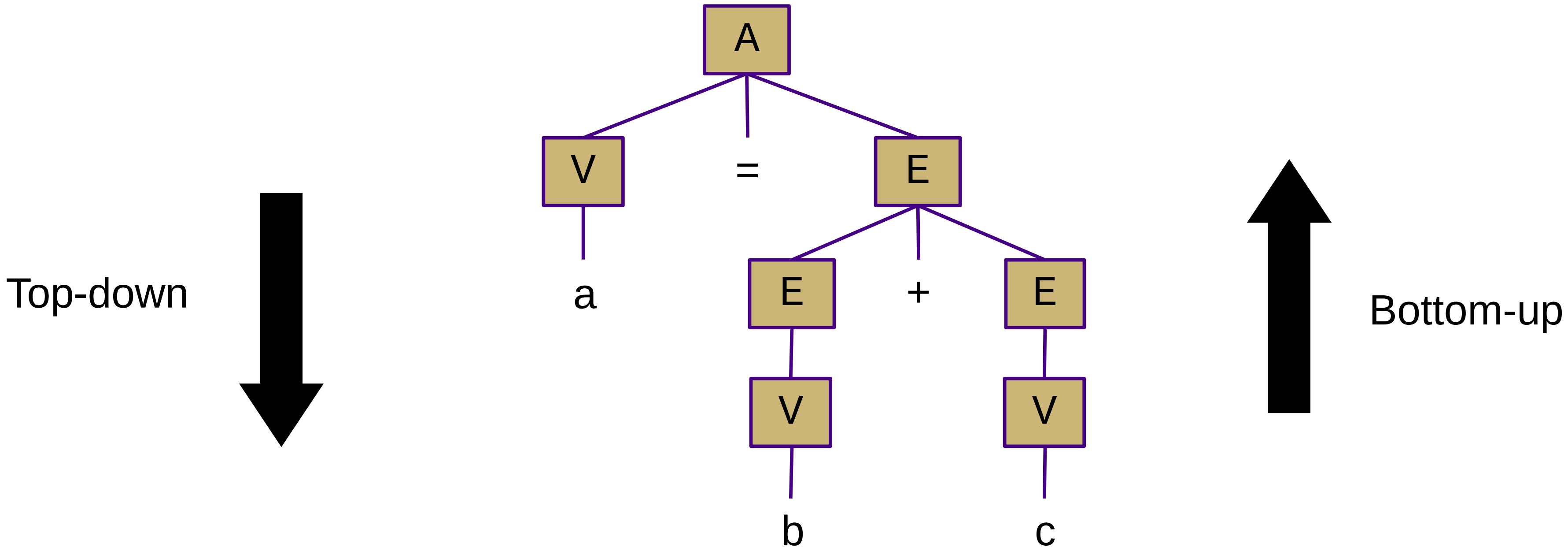
Bottom-Up (LR) Parsing

Compilation



Overview

- Two general parsing approaches
  - Top-down: begin with start symbol (root of parse tree), and gradually expand non-terminals
  - Bottom-up: begin with terminals (leaves of parse tree), and gradually connect using non-terminals



Shift-Reduce Parsing

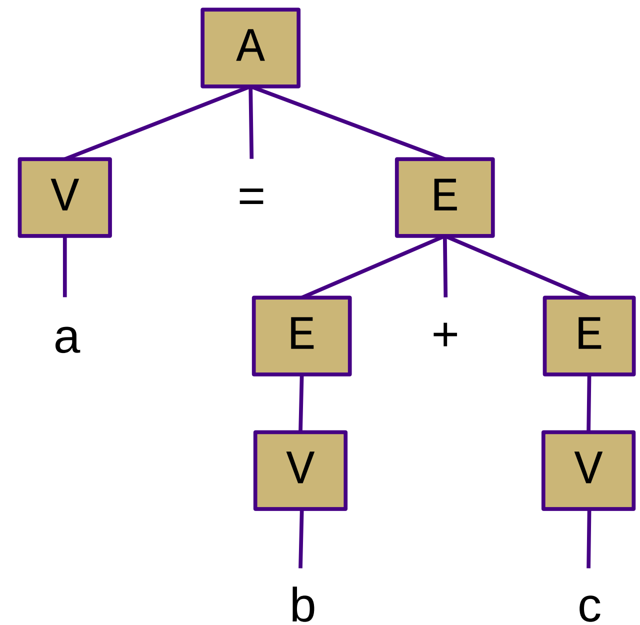
- Top-down (LL) parsers
  - Left-to-right scan, Leftmost derivation
  - Recursive routines, one per non-terminal (*recursive descent*)
  - Implicit stack (system call stack)
  - Requires more restrictive grammars
  - Simpler to understand and possible to hand-code
- Bottom-up (LR) parsers
  - Left-to-right scan, (reverse) Rightmost derivation
  - "Shift"/push terminals and non-terminals onto a stack
  - "Reduce"/pop to replace *handles* with non-terminals
  - Less restrictive grammars
  - Harder to understand and nearly always auto-generated
  - Very efficient

Shift-Reduce Parsing

-
  - shift 'a'
- a
  - reduce (V → a)
- V
  - shift '='
- V =
  - shift 'b'
- V = b
  - reduce (V → b)
- V = V
  - reduce (E → V)
- V = E
  - shift '+'
- V = E +
  - shift 'c'
- V = E + c
  - reduce (V → c)
- V = E + V
  - reduce (E → E + V)
- V = E
  - reduce (V = E)
- A
  - accept

(handles are underlined)

shift = push, reduce = popN



A → V = E
E → E + V
  | V
V → a | b | c

LR Parsing

- Creating an LR parser (pushdown automaton)
  - Build item sets
    - An item uses a dot (•) to represent parser status: "A → a • S b"
      - Dots on the left end: "possibilities"
      - Dots in the middle: "partially-complete"
      - Dots on the right end: "complete"
    - Item sets represent closures of parser states
    - Similar to NFA state collections in subset construction
  - Build **ACTION** / **GOTO** tables
    - Encodes stack and transition decisions (replaces δ in FA)
    - **ACTION**(state, terminal) = { *shift/push*, *reduce/pop*, *accept* }
    - **GOTO**(state, non-terminal) = state

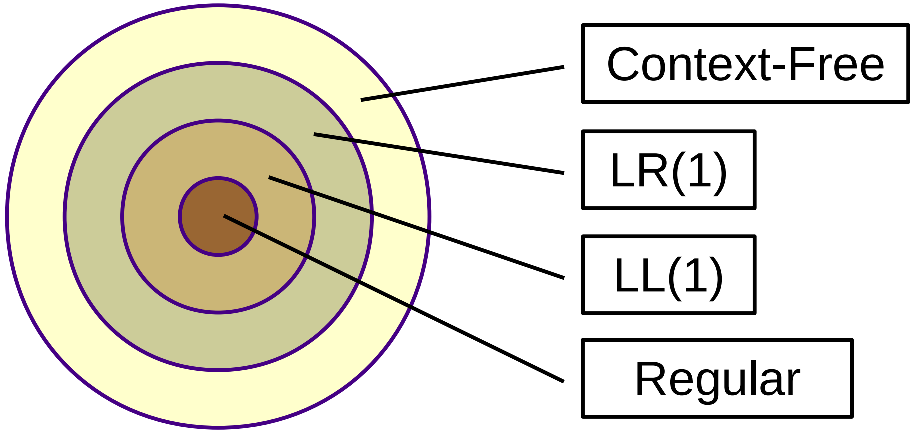
LR Parsing

- How much lookahead do we need?
  - Depends on how complicated the grammar is
  - LR(k) – multiple lookaheads (not necessary)
  - **LR(1)** – single lookahead (*our textbook covers this!*)
    - Very general and very powerful
  - LR(0) – no lookahead
    - Severely restricted; most "interesting" grammars aren't LR(0)

LR Parsing

- LR(1) grammars and parsers
  - **Left-to-right** scan of the input string
  - **Rightmost** derivation
  - **1 symbol** of lookahead
  - Less restricted form of context-free grammar
    - Support for most language features
    - Efficient parsing

Context-Free Hierarchy

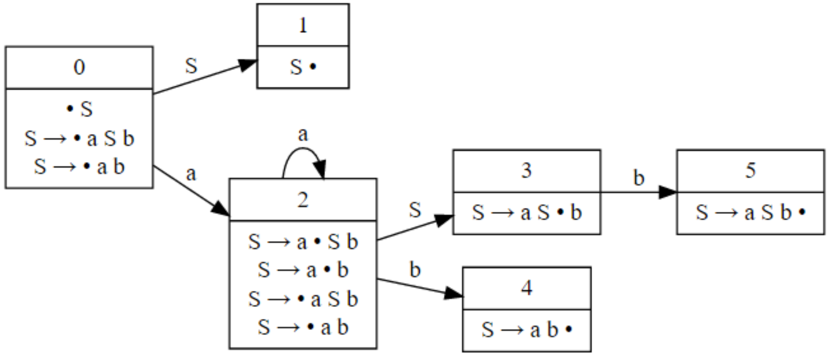


LR Parser Variants

- LR(1) – single lookahead (*our textbook covers this!*)
  - Very general and very powerful
  - Lots of item sets; tedious to construct by hand
  - Overkill for most practical languages
- LALR(1) – special case of LR(1) that merges some states
  - Less powerful, but easier to manage
- **SLR(1)** – special case of LR(1) w/o explicit lookahead
  - Uses **FOLLOW** sets to disambiguate
  - Even less powerful, but much easier to understand
  - Slightly counterintuitive: all LR(1) languages have SLR(1) grammars
  - So SLR(1) is sufficiently general for our purposes
  - Use LR(0) item sets and generate SLR(1) ACTION/GOTO tables

LR(0) Item Sets

- LR(0) item sets and automaton
  - Start with an item representing "• S"
  - Form new sets by "moving the dot"
  - Take the closure to add more states if the dot lies to the left of a non-terminal
    - (Denoted here in blue)
  - Convert to finite automaton for recognizing handles by adding transitions
    - Each set becomes a state
    - "Moving the dot" = state transition + stack push



S → a S b
  | a b

CC_0: • S
S → • a S b
S → • a b

CC_1: S •

CC_2: S → a • S b
S → a • b
S → • a S b
S → • a b

CC_3: S → a S • b

CC_4: S → a b •

CC_5: S → a S b •

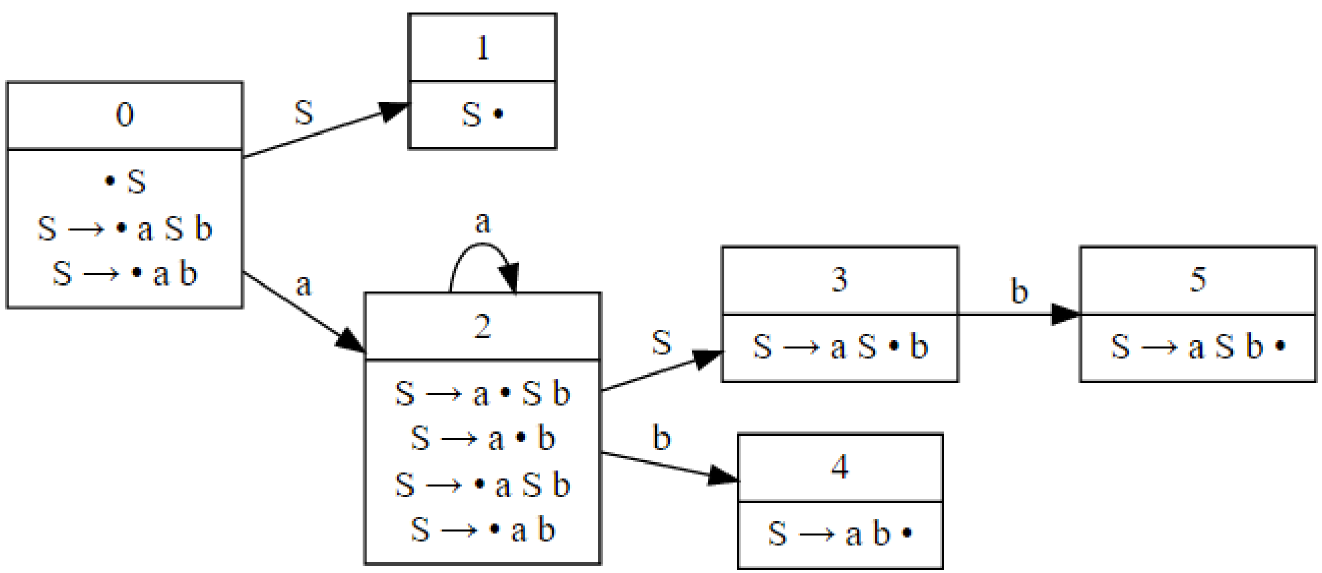
SLR(1) Tables

- Create **ACTION** and **GOTO** tables
  - For each item set i
    - If an item matches **A → β • c γ**
      - **ACTION**(i, c) = "shift" to corresponding item set ("move the dot")
    - If an item matches **A → β •**
      - **ACTION**(i, x) = "reduce A → β" for all x in **FOLLOW**(A) (“backtrack in FA”)
    - If an item matches **A → β • B γ**
      - **GOTO**(i, B) = corresponding item set ("move the dot")
  - **ACTION**(S', $) = "accept"

SLR(1) parsing

S → a S b
  | a b

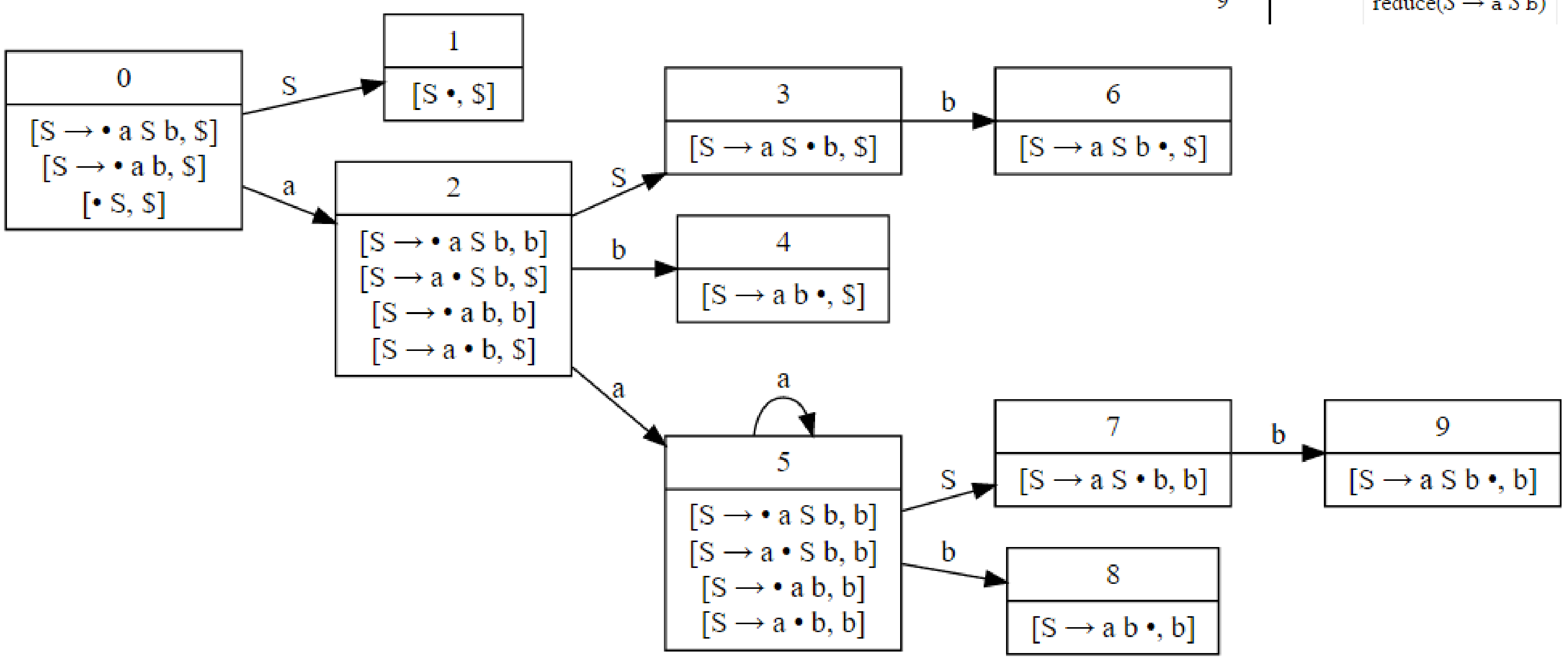
State	ACTION: a	ACTION: b	ACTION: $	GOTO: S
0	shift(2)			1
1			accept	
2	shift(2)	shift(4)		3
3		shift(5)		
4		reduce(S → a b)	reduce(S → a b)	
5		reduce(S → a S b)	reduce(S → a S b)	



LR(1) parsing

S → a S b
  | a b

State	a	b	$	S
0	shift(2)			1
1			accept	
2	shift(5)	shift(4)		3
3		shift(6)		
4			reduce(S → a b)	
5	shift(5)	shift(8)		7
6			reduce(S → a S b)	
7		shift(9)		
8		reduce(S → a b)		
9		reduce(S → a S b)		



LR Conflicts

- Shift/reduce
  - Can be resolved by always shifting or by grammar modification
- Reduce/reduce
  - Requires grammar modification to fix

```
A -> V = E
E -> E + V
E -> V
V -> a | b | c
```

Shift/reduce conflict in LR(0)

```
A -> x A x
A ->
```

Shift/reduce conflict (all LR)

```
A -> B | C
B -> x
C -> x
```

Reduce/reduce conflict (all LR)

Observation: none of these languages are LL(1) either!