CS 430
Spring 2015

Mike Lam, Professor

Parsing

Syntax Analysis

- We can now formally describe a language's syntax
  - Using regular expressions and BNF grammars
- How does that help us?

Syntax Analysis

- We can now formally describe a language's syntax
  - Using regular expressions and BNF grammars
- How does that help us?

It allows us to program a computer to recognize and translate programming languages automatically!

Parsing

- General goal of syntax analysis: turn a program into a form usable for automated translation or interpretation
  - Report syntax errors (and optionally recover)
  - Produce a *parse tree* / *syntax tree*

```
while b != 0:
    if a > b:
        a = a − b
    else:
        b = b − a
return a
```

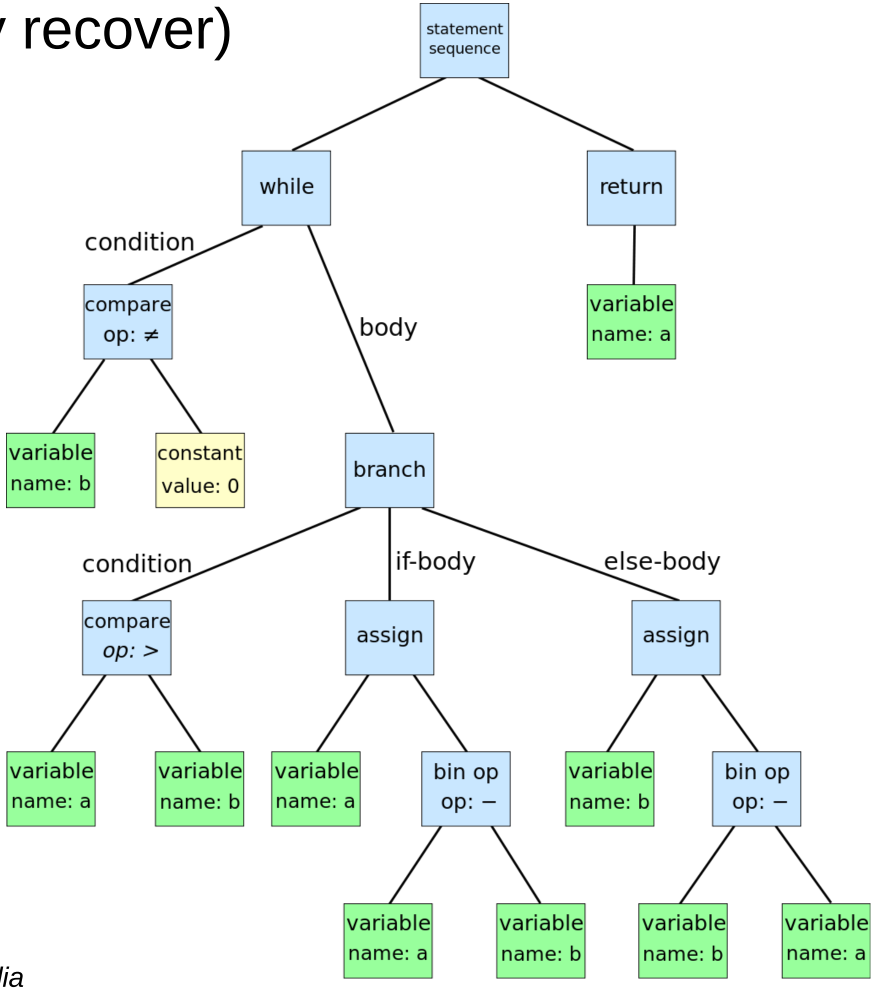
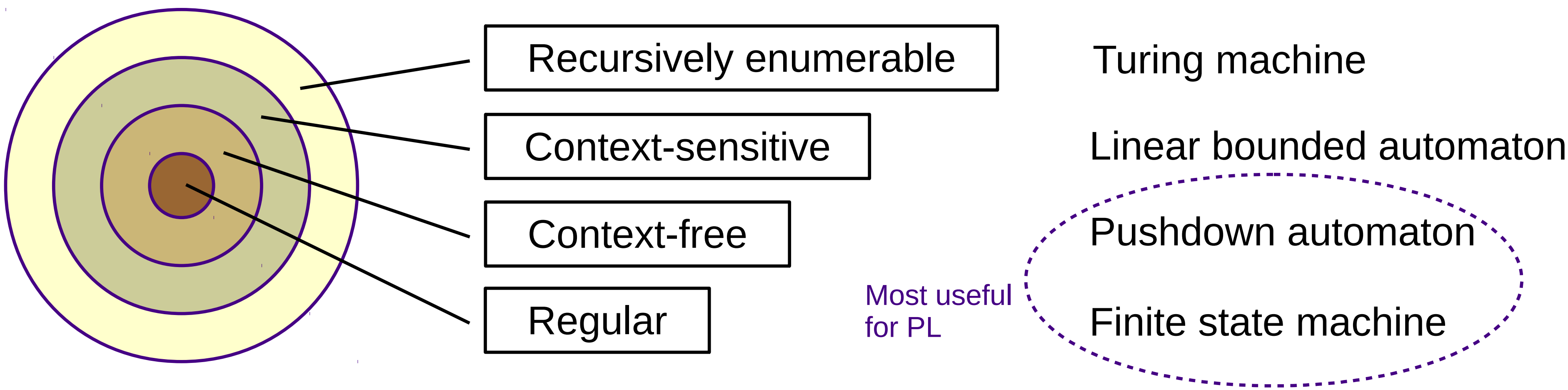


Image taken from Wikipedia

Languages

Chomsky Hierarchy of Languages | **Deciding machine**



- Language:
  - L = { set of sequences of characters from alphabet Σ }
  - Colloquially: "set of all valid sentences in the language"

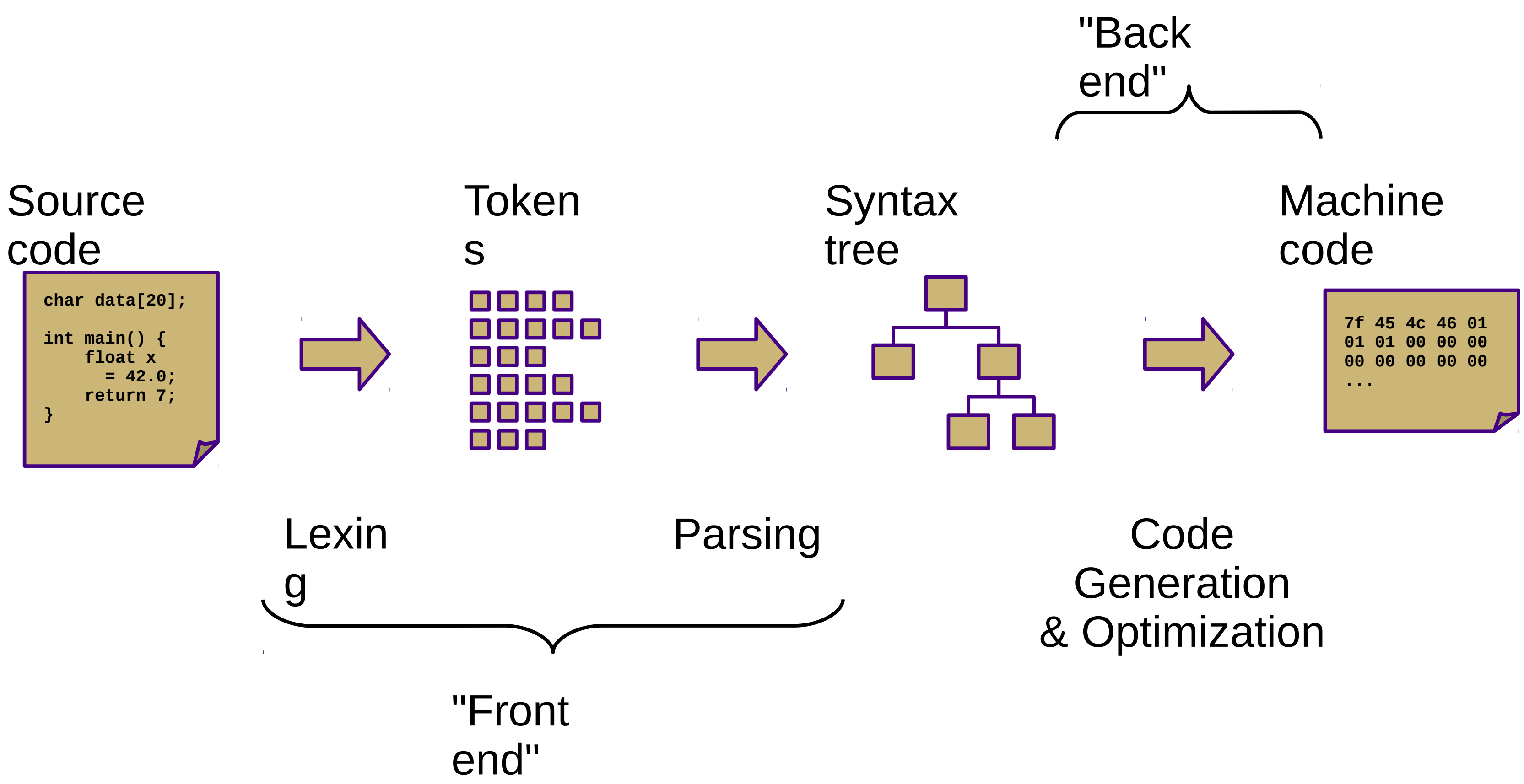
Challenge: Write a regular expression to check for matched parentheses.

Valid: "", "()", "(())", "()()"
Invalid: "(", ")", "())", "(()()"

Syntax Analysis

- 1) Lexical analysis
  - *Scanning*: text → tokens
  - Regular languages (described by regular expressions)
- 2) Syntax analysis
  - *Parsing*: tokens → syntax tree
  - Context-free languages (described by context-free grammars)
- Often implemented separately
  - For simplicity (lexing is simpler), efficiency (lexing is expensive), and portability (lexing can be platform-dependent)
- Together, they represent the first "phase" of compilation or interpretation
  - Referred to as the "front end" of a compiler

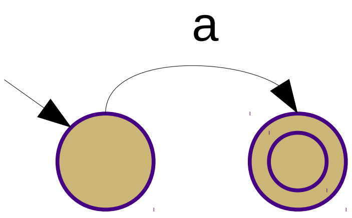
Compilation



Lexical Analysis

- Performed automatically by state machines (*finite state automata)*
  - Set of states with a single *start state*
  - Transitions between states on inputs (+ implicit *dead states*)
  - Some states are *final* or *accepting*
- Deterministic vs. non-deterministic
  - Non-deterministic: multiple possible states for given sentence
  - One edge from each state per character (deterministic)
  - Multiple edges from each state per character (non-deterministic)
  - Empty or ε-transitions (non-deterministic)

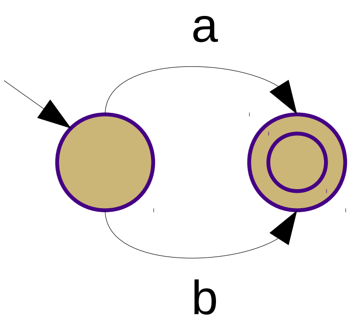
Regex: a



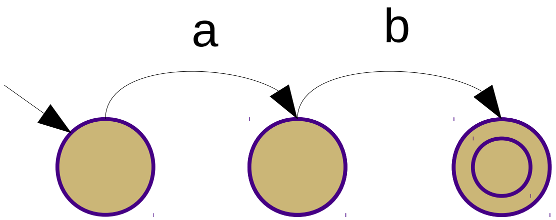
Lexical Analysis

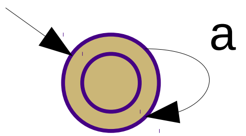
- Examples:

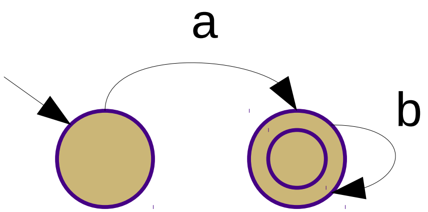
a|b



ab



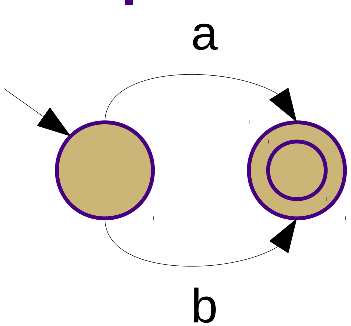




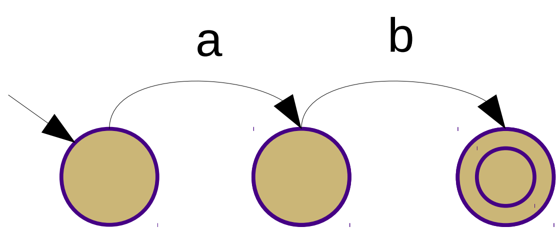
Lexical Analysis

- Examples:

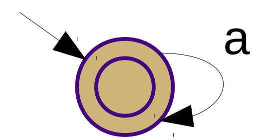
a|b



ab

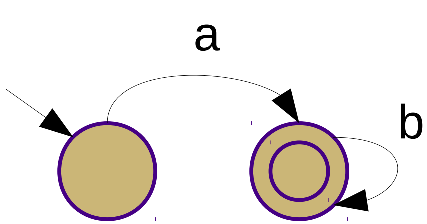


a*



aa*|b

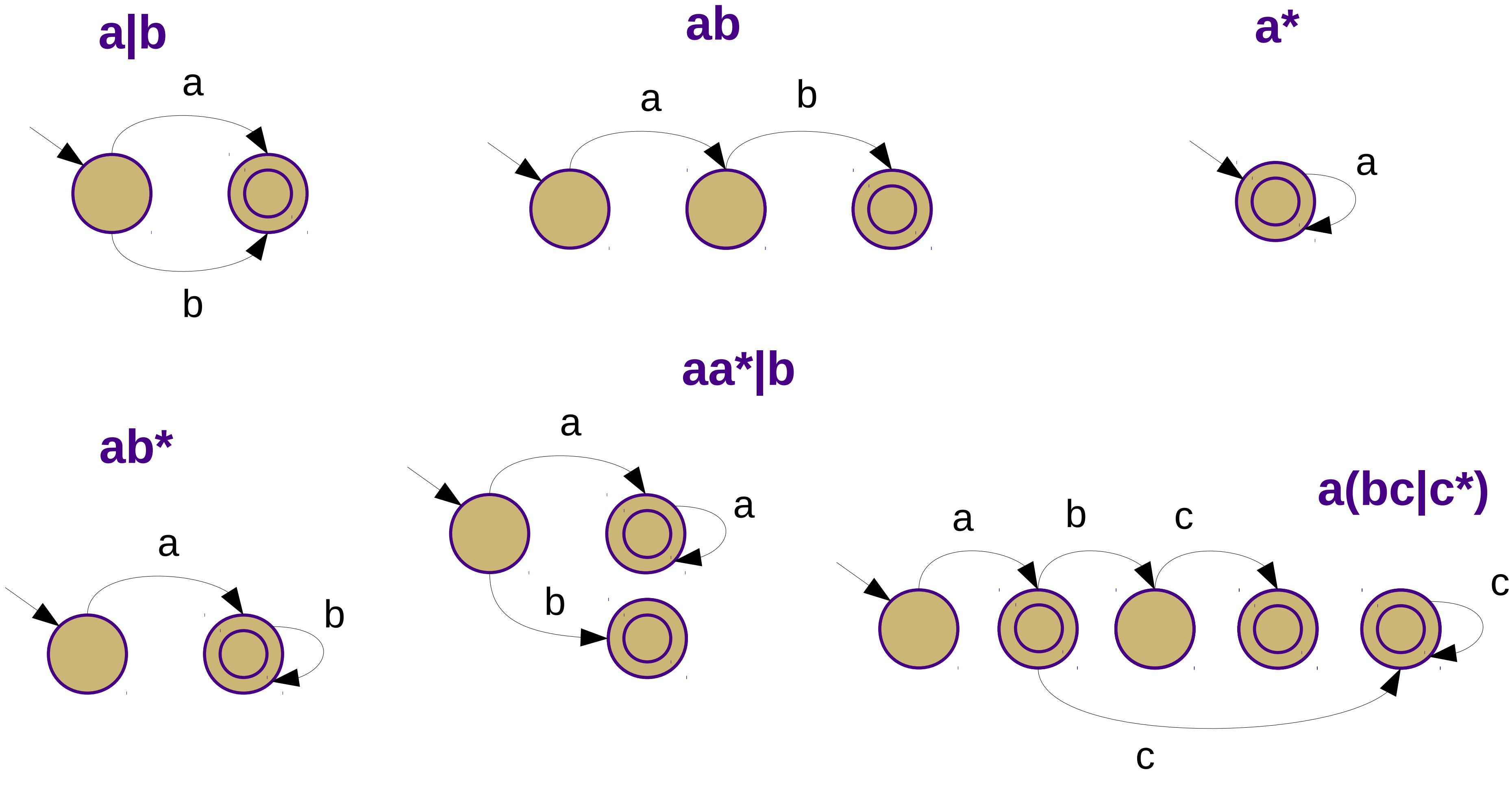
ab*



a(bc|c*)

Lexical Analysis

- Examples:



Parsing

- Implemented using stacks
  - Formally: pushdown automata
- Two major types of parsers:
  - Recursive-descent parsers
    - Sometimes called *top-down* parsers
    - Left to right token input, Leftmost derivation (LL)
  - Shift/reduce parsers
    - Sometimes called *bottom-up* parsers
    - Left to right token input, Rightmost derivation (LR)

Recursive Descent (LL) Parser

A → # B & B #
| # B #

B → x | y

Assuming the following methods are implemented:

`bool consume(char c)`
Consumes a character of input and verifies that it matches the given character (returns "false" if it does not).

`char peek()`
Returns a copy of the next character of input to be consumed, but does not consume it.

```
parseA():
    consume('#')
    parseB()
    if peek() == '&':
        consume('&')
        parseB()
    consume('#')
```

```
parseB():
    if peek() == 'x':
        consume('x')
    elif peek() == 'y':
        consume('y')
    else:
        error "Bad input: "
              + peek()
```

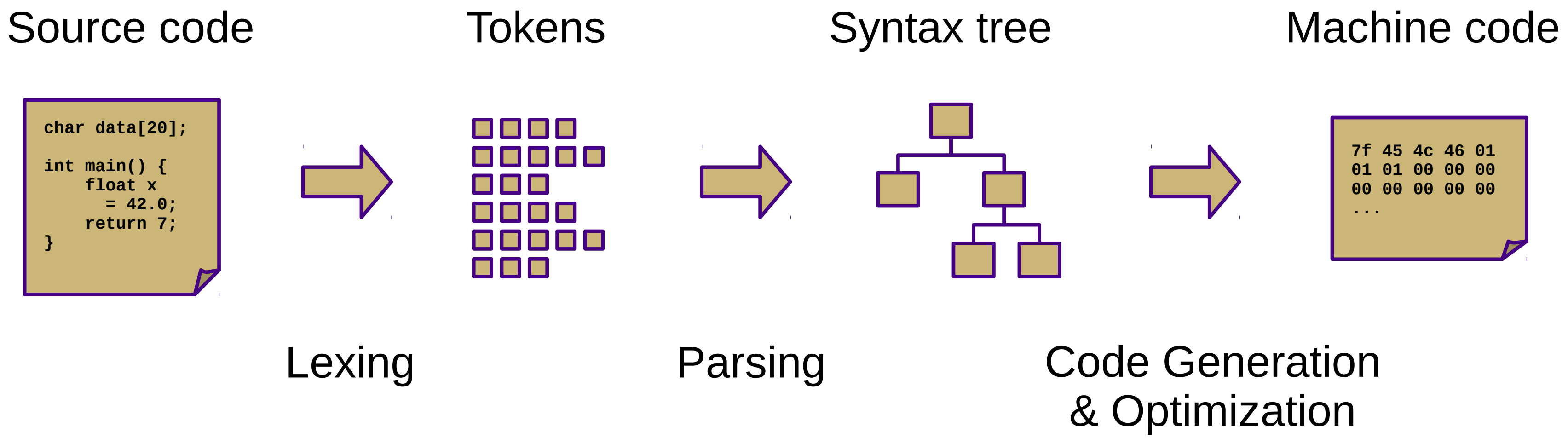
Recursive Descent (LL) Parsing

- Collection of parsing routines that call each other
  - Uses a stack implicitly (call stack)
  - Usually one routine per non-terminal in the grammar
  - Each routine builds a subtree of the parse tree associated with the corresponding non-terminal
- Advantages
  - Relatively simple to write by hand
- Disadvantage
  - Doesn't work with left-recursive grammars and non-pairwise-disjoint grammars
    - This can sometimes be fixed (e.g., with left factoring)

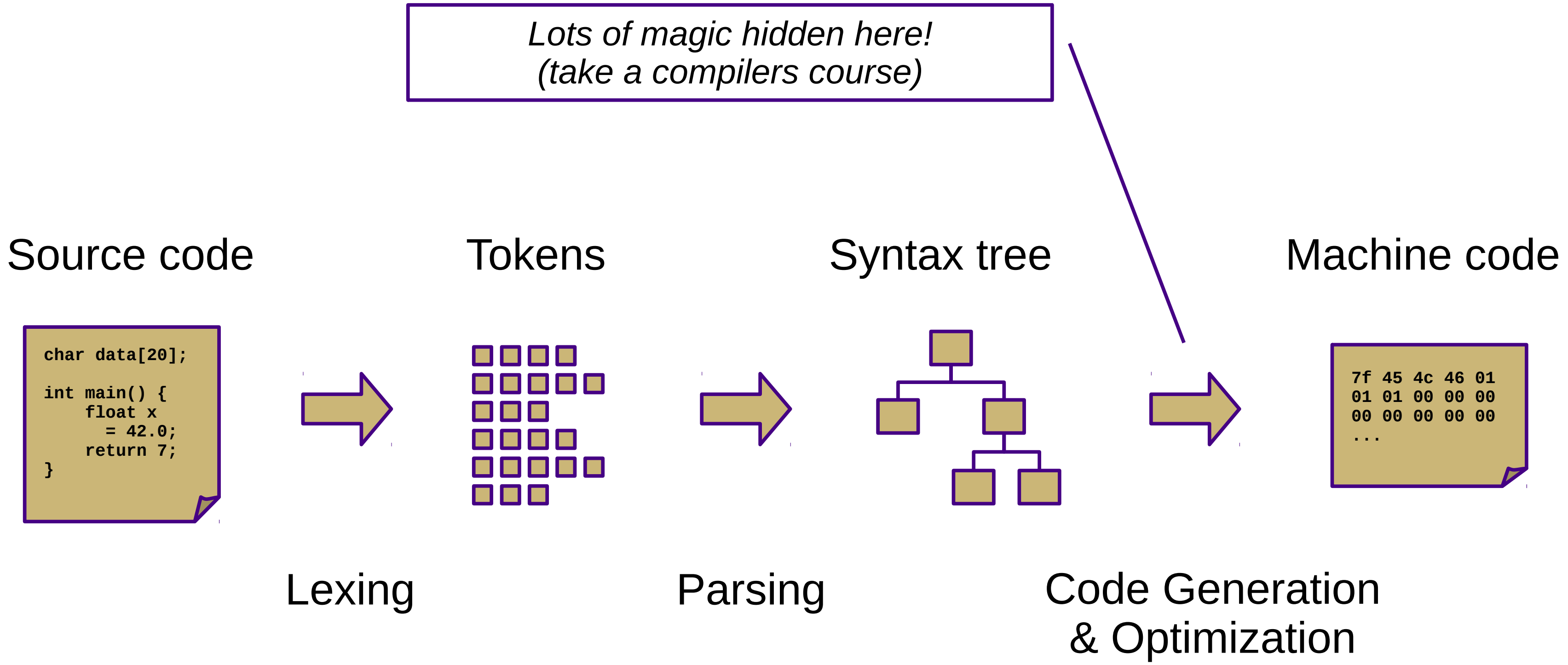
Shift/Reduce (LR) Parsing

- Based on a table of states and actions
  - Explicitly stack-based
  - *Shift* tokens onto a stack
  - Pattern-match top of stack to a RHS and *reduce* to corresponding LHS (pop RHS and push LHS)
- Advantages
  - Much more general than LL parsers
- Disadvantages
  - Very difficult to construct by hand
    - Usually constructed using automated tools

Compilation



Compilation



Compiler Tools

- Creating a parser can be somewhat automated by lexer/parser generators
  - Classic: lex and yacc
  - Modern: flex and bison (C) or ANTLR (Java, Python, etc.)
- Input: language description in regular expressions and BNF
- Output: hard-coded lexing and parsing routines
  - Can be re-generated if the grammar needs to be changed
  - Still have to manually write the translation or execution code

Activity

- Construct state machines for the following regular expressions:

x*yz* **1(1|0)*** **1(10)*** **(a|b|c)(ab|bc)**

(dd*.d*)|(d*.dd*) ← ε-transitions may make this one slightly easier

- Write recursive-descent parsing routines for the following grammar:

```
A → V = E ;
E → T + E
  | T
T → V * T
  | V
V → a | b | c
```

You may assume the following methods are implemented:

```
bool consume(char c)
```

Consumes a character of input and verifies that it matches the given character (returns "false" if it does not).

```
char peek()
```

Returns a copy of the next character of input to be consumed, but does not consume it.