CS 430
Spring 2022

Mike Lam, Professor

Parsing

Syntax Analysis

- We can now formally describe a language's syntax
  - Using regular expressions and context-free 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
```

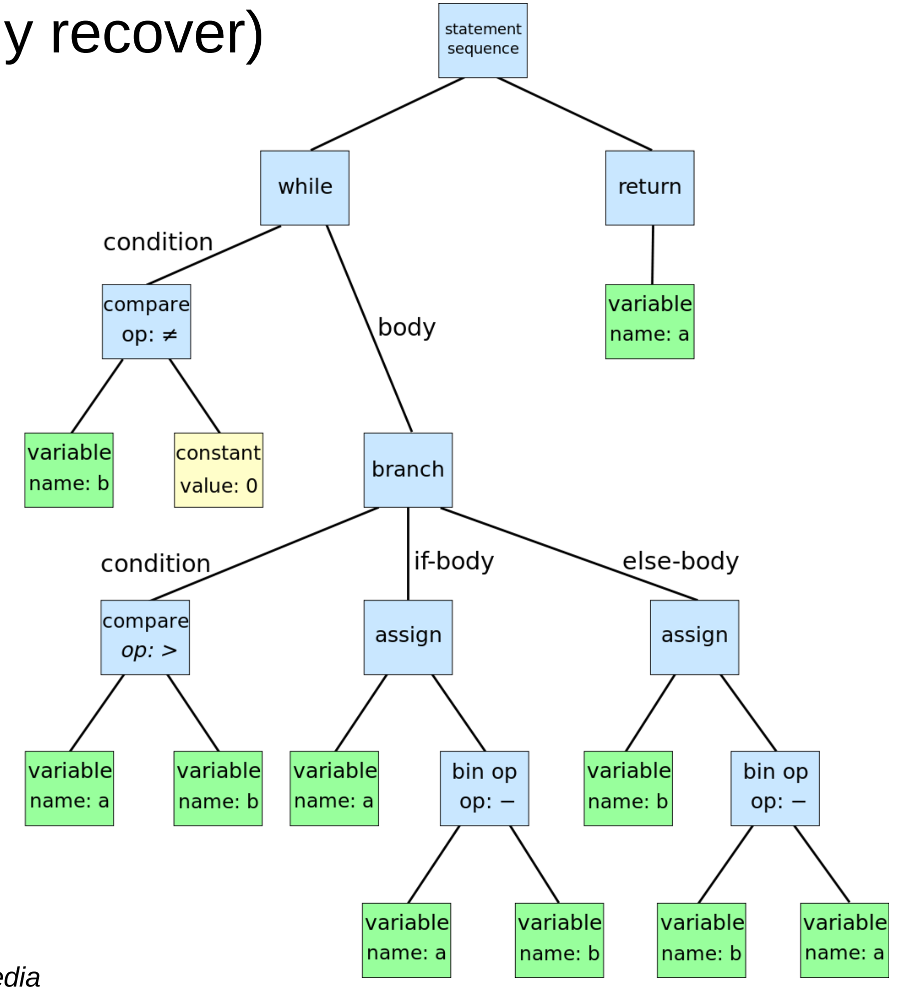
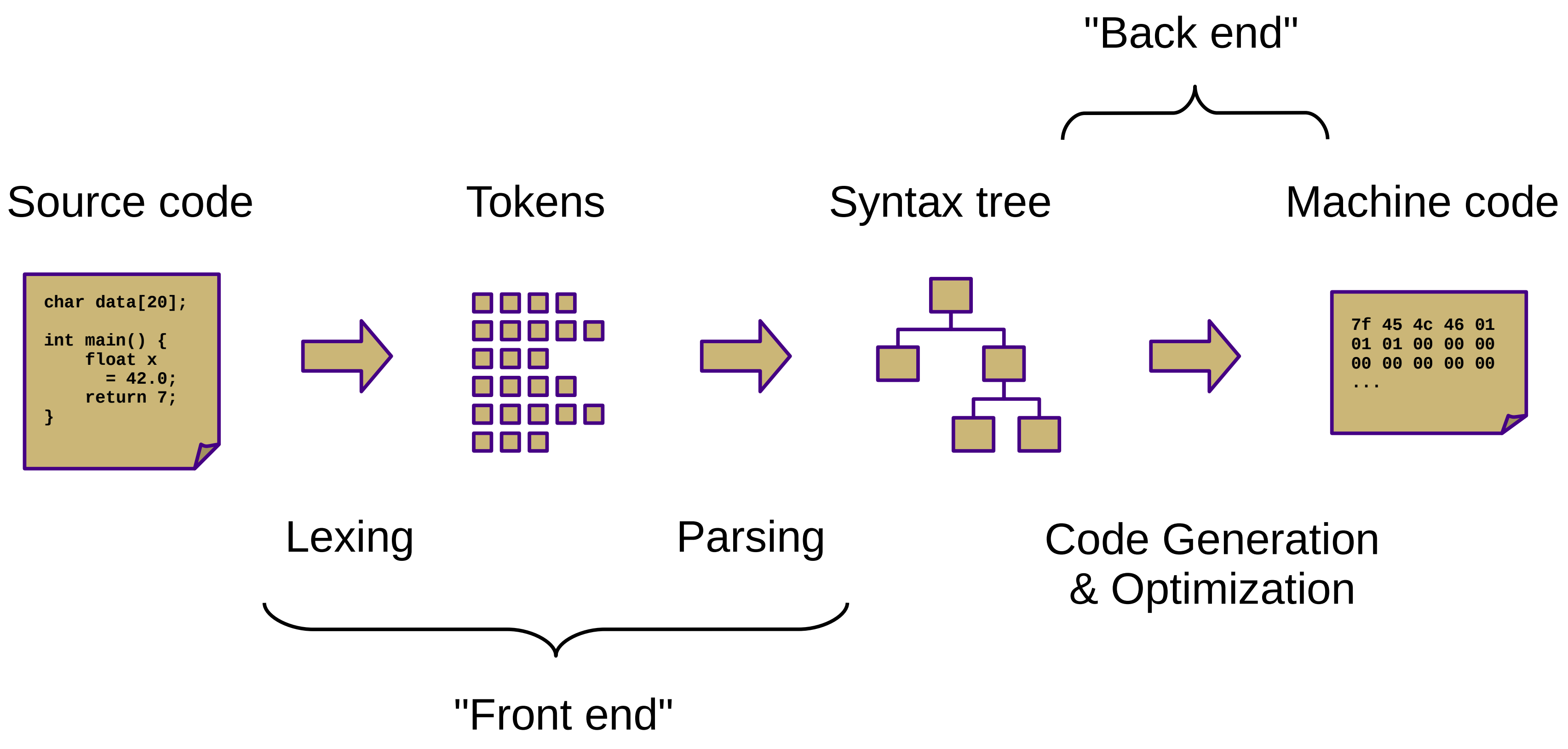


Image taken from Wikipedia

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
  - Referred to as the front end of a compiler

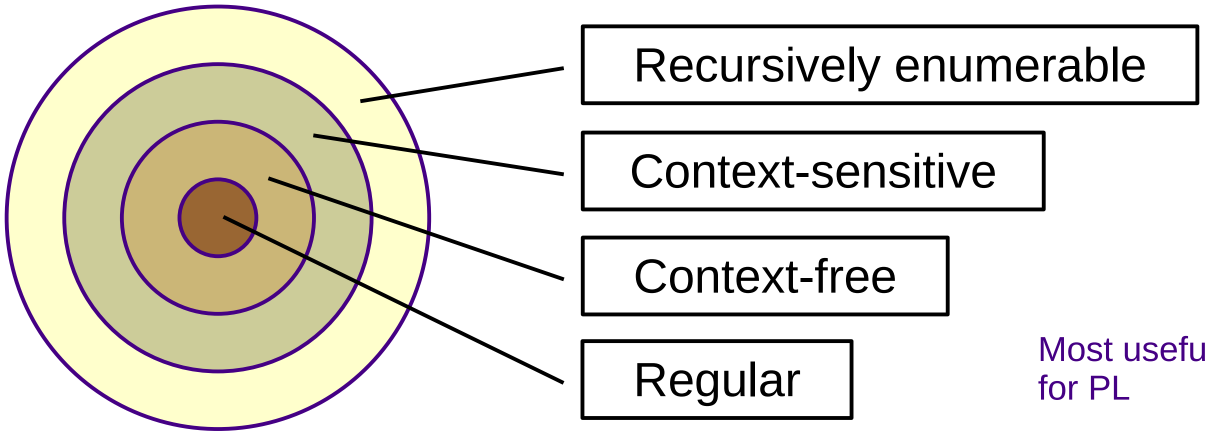
Compilation



Lexical Analysis

Chomsky Hierarchy of Languages

Deciding machine



Turing machine

Linear bounded automaton

Pushdown automaton

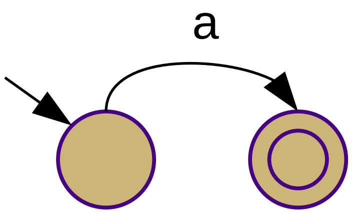
Finite state machine

Most useful for PL

Lexical Analysis

- Regular languages are recognized by state machines (finite automata)
  - Set of states with a single start state
  - Transitions between states on inputs (+ implicit dead states)
  - Some states are final or accepting

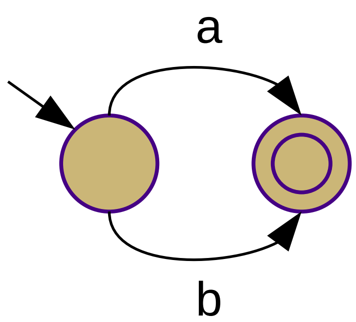
Regex: a



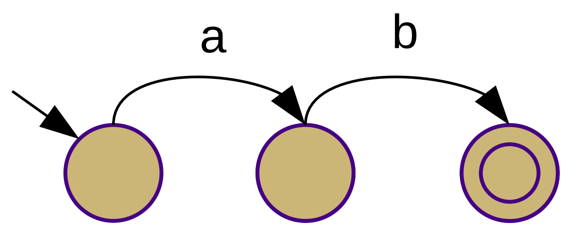
Lexical Analysis

- More examples:

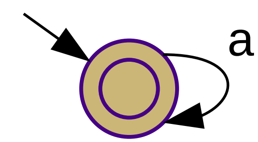
a|b



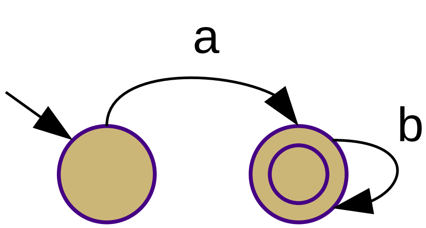
ab



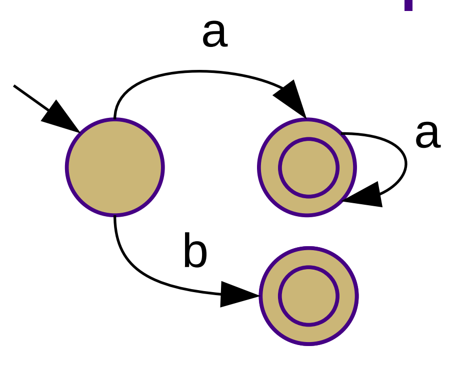
a*



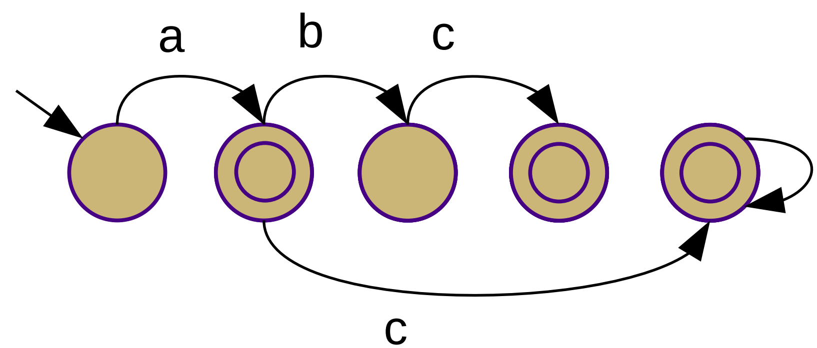
ab*



aa*|b



a(bc|c*)



Lexing

- Combine finite automata from multiple regular expressions
  - Read as much as possible
  - Return token and reset automaton

Figure 4.1

A state diagram to recognize names, parentheses, and arithmetic operators

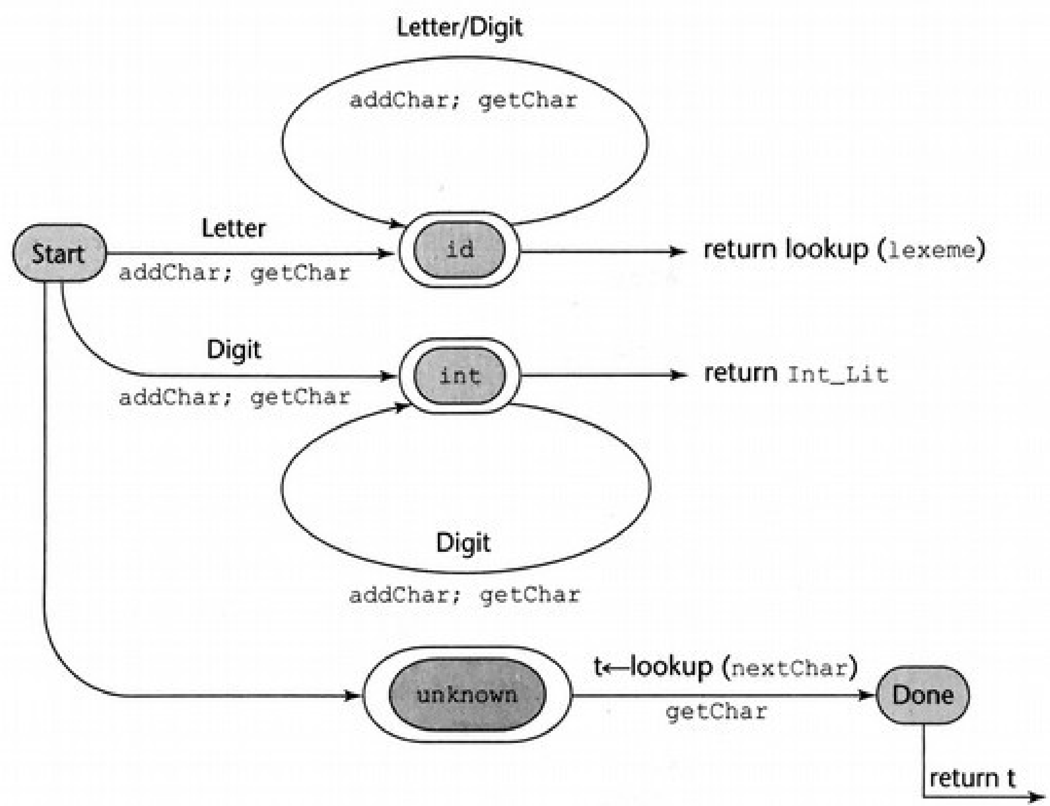


Figure from CPL 11th Ed.

Parsing

- Implemented using a finite automaton + a **stack**
  - Formally: pushdown automata
- Two major types of parsers:
  - Recursive-descent parsers
    - Implicit stack: system call stack
    - Sometimes called top-down parsers
    - Left to right token input, Leftmost derivation (LL)
  - Shift/reduce parsers
    - Explicit stack
    - Sometimes called bottom-up parsers (w/ explicit stack)
    - Left to right token input, Rightmost derivation (LR)

Recursive Descent (LL) Parsing

- Collection of parsing routines that call each other
  - Uses a stack implicitly (i.e., system 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
- Advantage
  - 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
  - Push (or shift) tokens onto a stack
  - Pattern-match top of stack to a RHS (called a handle) and reduce to corresponding LHS (pop RHS and push LHS)
- Advantage
  - Much more general than LL parsers
- Disadvantage
  - Very difficult to construct by hand
    - Usually constructed using automated tools

Recursive Descent Parsing

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('#')
```

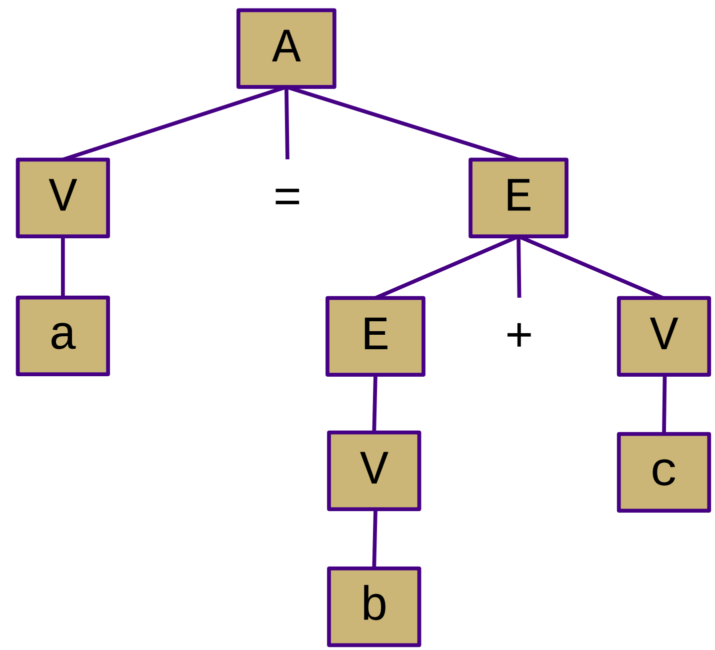
```
parseB():
    if peek() == 'x':
        consume('x')
    elif peek() == 'y':
        consume('y')
    else:
        error "Bad input: "
              + peek()
```

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

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

Conclusion

- Parsers convert code to a syntax tree
  - First part of compilation or interpretation
  - Largely considered a “solved” problem now
  - CPL Ch.4 provides a brief overview
  - For a deeper dive, take CS 432!