CS 430
Spring 2020

Mike Lam, Professor

```
selectionStatement
    :   'if' '(' expression ')' statement ('else' statement)?
    |   'switch' '(' expression ')' statement
    ;

iterationStatement
    :   While '(' expression ')' statement
    |   Do statement While '(' expression ')' ';'
    |   For '(' forCondition ')' statement
    ;
```

Syntax and Parsing

Consider the following code

Language A

```
if (a < 5) {
    printf(“%d\n”, a);
}
```

Language B

```
if a < 5:
    print a
```

Language C

```
if [ $a -lt 5 ]; then
    echo $a
fi
```

Language D

```
puts a if a < 5
```

Syntax

- Textbook: syntax is "the form of [a language's] expressions, statements, and program units."
- In other words: the **appearance** of code
- Semantics deal with the **meaning** of code
  - Syntax and semantics are (ideally) closely related
- Goals of syntax analysis:
  - Checking for program validity or correctness
  - Facilitate translation or execution of a program

Syntax Analysis

- Context-free grammar
  - Description of a language's syntax
    - Usually written in Backus-Naur Form
  - Encodes hierarchy and structure of code
    - Usually represented using a tree
  - Provide ways to control ambiguity, associativity, and precedence in a language

Grammars

- Non-terminals vs. terminals
  - Terminals are essentially tokens (described using regular expressions)
  - Non-terminals represent units of program structure
  - One special non-terminal: the start symbol
- Production rules
  - Left hand side: single non-terminal
  - Right hand side: sequence of terminals and/or non-terminals
  - LHS is replaced by the RHS during derivation
  - Colloquially: "is composed of"

```
<assign> ::=  <var> = <expr>
<var>    ::=  a | b | c
<expr>   ::=  <expr> + <expr>
           |  <var>
```

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

Derivation

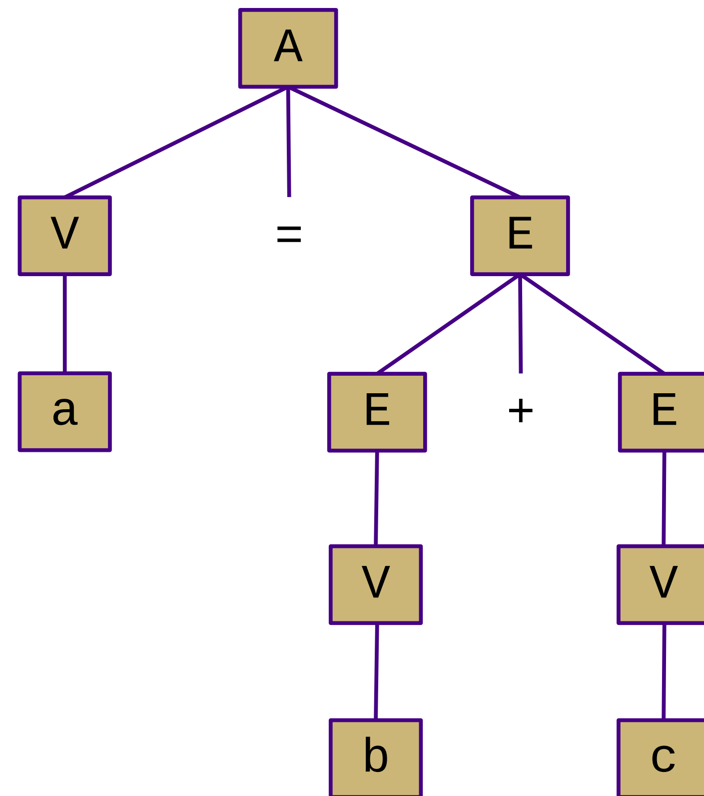
- Derivation: a series of grammar-permitted transformations leading to a sentence (sequence of terminals)
  - Each transformation applies exactly one rule
  - Each intermediate string of symbols is a sentential form
  - Leftmost vs. rightmost derivations
    - Which non-terminal do you expand first?
  - Parse tree represents a derivation in tree form
    - Built from the start symbol (root) down during derivation
    - Final parse tree is called complete parse tree
    - The sentence is the sequence of all leaf nodes (terminals)
    - Interior nodes represent non-terminals
    - Represents a program, executed from the bottom up

Example

- Show the **leftmost** derivation and parse tree of the sentence "a = b + c" using this grammar:

$$
\begin{aligned}
A &\rightarrow V = E \\
V &\rightarrow a \mid b \mid c \\
E &\rightarrow E + E \\
  &\mid V
\end{aligned}
$$

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



Ambiguous Grammars

- An ambiguous grammar allows multiple derivations (and therefore parse trees) for the same sentence
  - The semantics may be similar or identical, but there is a difference syntactically
  - It is important to be precise!
- Can usually be eliminated by rewriting the grammar
  - Usually by making one or more rules more restrictive
- Example: derive "a = x + y + z" and show the parse tree

$$
\begin{aligned}
A &\rightarrow V = E \\
V &\rightarrow a \mid b \mid c \\
E &\rightarrow E + E \\
&\mid V
\end{aligned}
$$

Operator Associativity

- The previous ambiguity resulted from an unclear associativity
- Does x+y+z = **(x+y)**+z or x+**(y+z)**?
  - Former is left-associative (E → E + V)
  - Latter is right-associative (E → V + E)
- Can be enforced explicitly in a grammar
  - The problem is the E → E + E production
    - Need to remove one possible interpretation
  - Left-associative: change to (E → E + V)
  - Right-associative: change to (E → V + E)
  - Sometimes just noted with annotations

Operator Precedence

- Precedence determines the relative priority of operators in a single production
  - Another source of ambiguity
- Does x+y*z = **(x+y)***z or x+**(y*z)**?
  - Former: "+" has higher precedence
  - Latter: "*" has higher precedence
- Can be enforced explicitly in a grammar
  - Separate into two non-terminals (e.g., E and T)
  - Non-terminals closer to the start symbol have lower precedence
    - E.g., for “normal” precedence: $E \rightarrow E + T \mid T$ $T \rightarrow T * V \mid V$
  - Sometimes just noted with annotations

Extended BNF

- New constructs
  - Optional: []
  - Closure: {}
  - Multiple-choice: |
- All of these can be expressed using regular BNF
  - (exercise left to the reader)
- So these are really just conveniences

Summary

- Context-free languages
  - Described by context-free grammars (using BNF)
  - Often used to describe a programming language's syntax
- Lots of very nice language theory
  - We won't dig too deeply in this course
  - You have seen (or will see) a bit in CS 327
  - Take CS 432 if you're interested in digging deeper

Examples

- ANTLR grammars:
  - C
  - C++14
  - Java 8
  - Ruby
  - Prolog

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
```

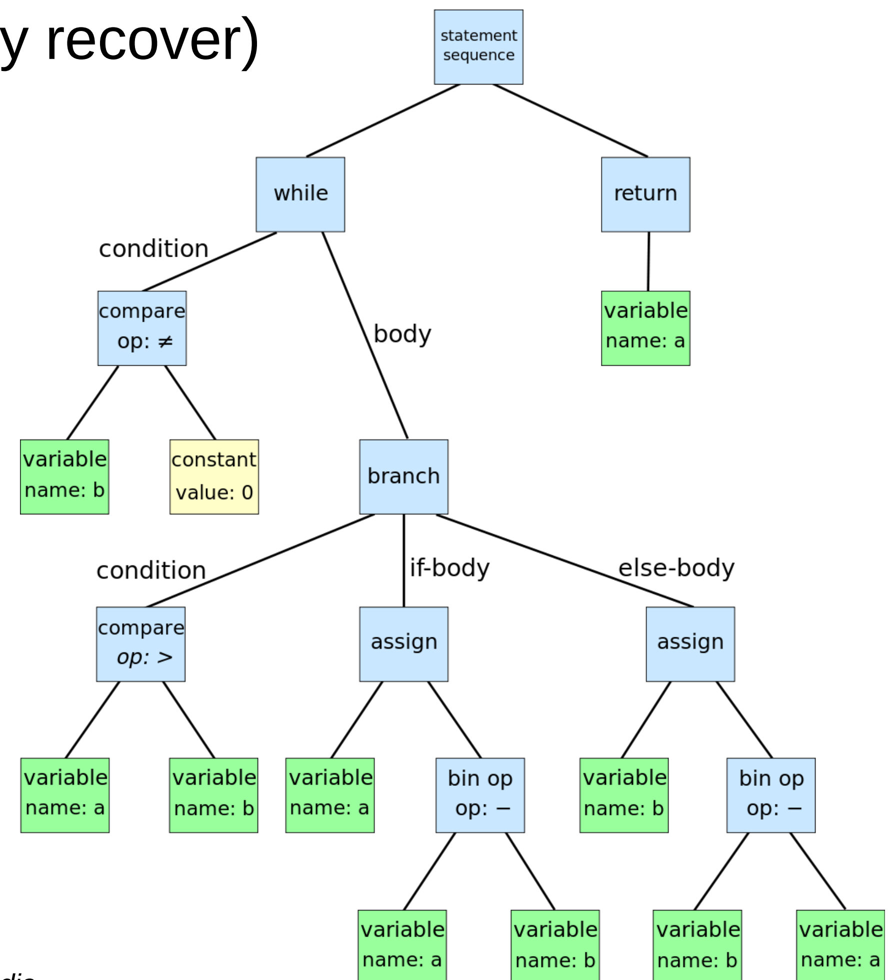
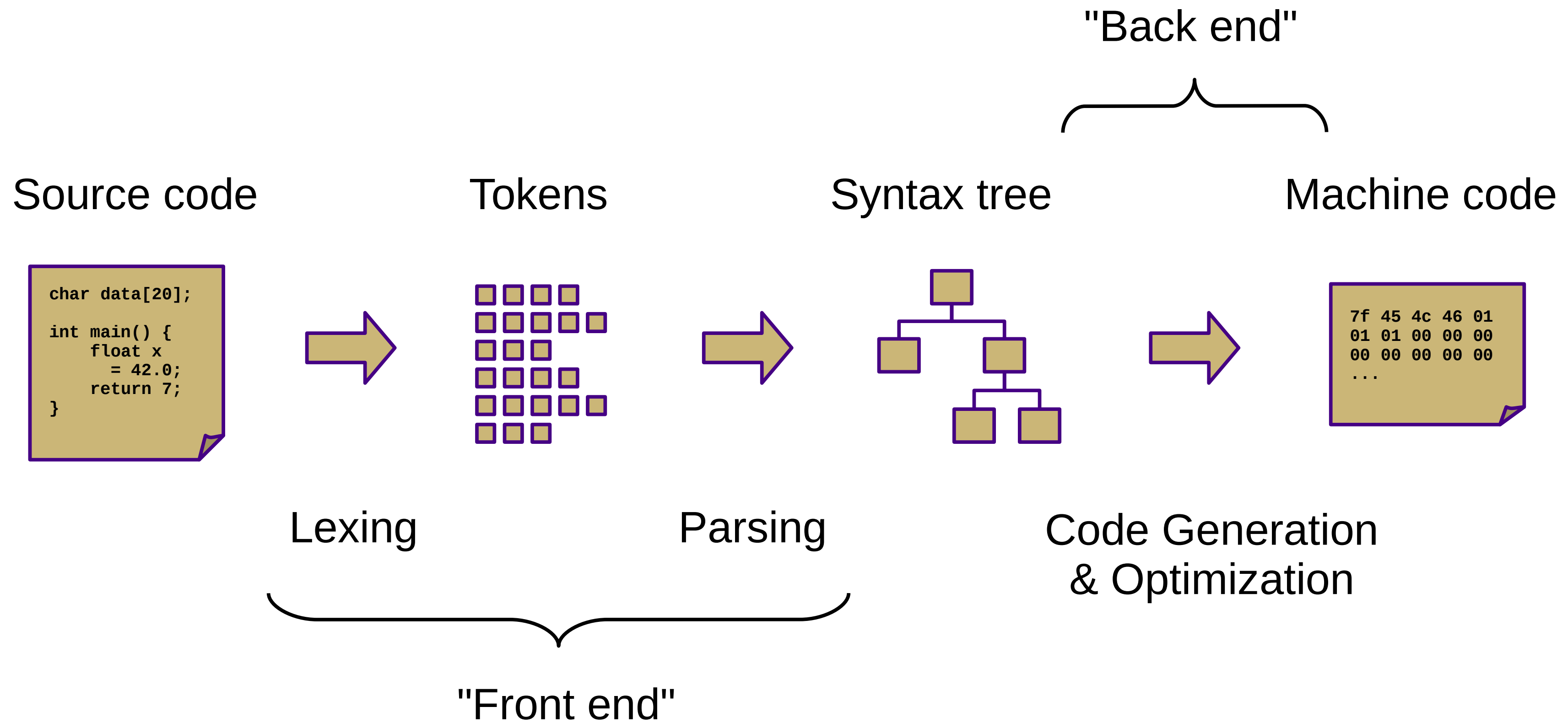


Image taken from Wikipedia

Syntax Analysis

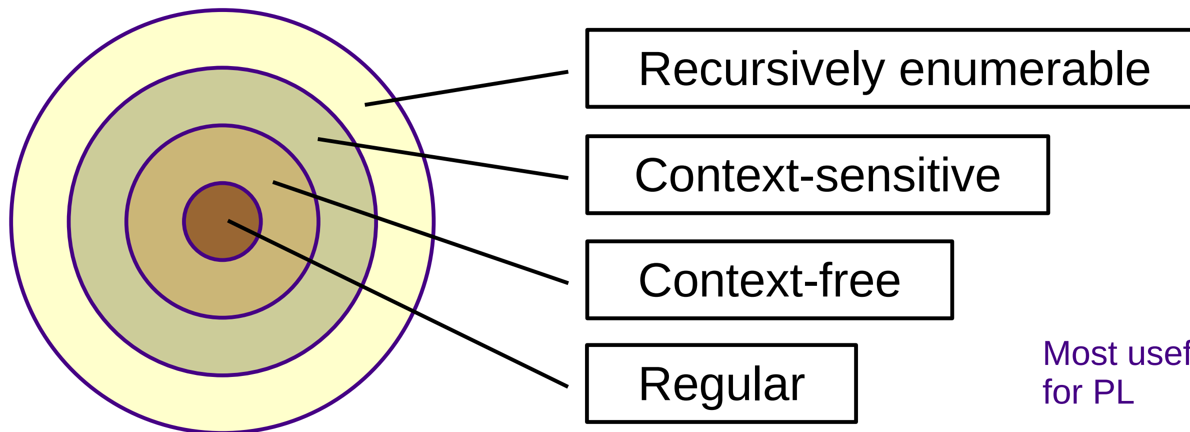
- 1) Lexical analysis
  - Scanning: text → tokens
  - Regular languages (described by regular expressions)
- 2) Syntax analysis
  - Parsing: tokens → syntax tree
  - Context-free languages (described by context-free grammars)
- Often implemented separately
  - For simplicity (lexing is simpler), efficiency (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



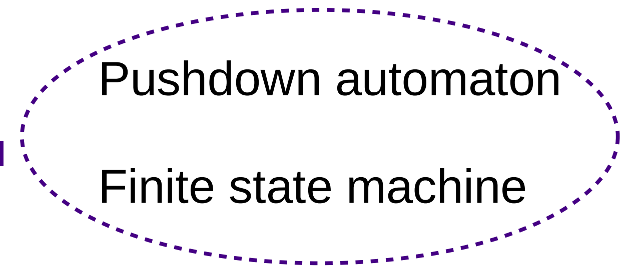
Lexical Analysis

Chomsky Hierarchy of Languages | **Deciding machine**



Turing machine

Linear bounded automaton

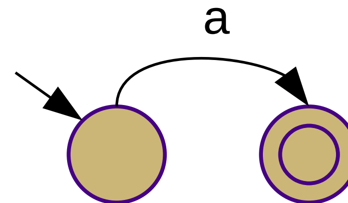


Most useful for PL

Lexical Analysis

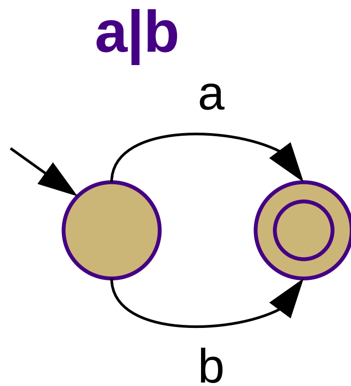
- Regular languages are recognized by state machines (finite automata)
  - Set of states with a single start state
  - Transitions between states on inputs (+ implicit dead states)
  - Some states are final or accepting

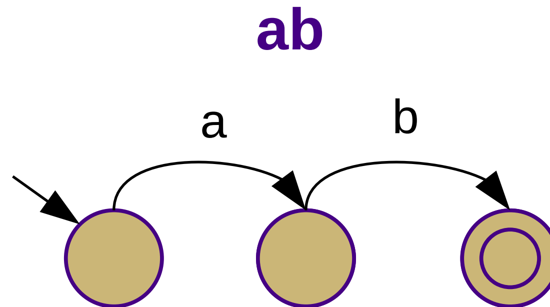
Regex: a

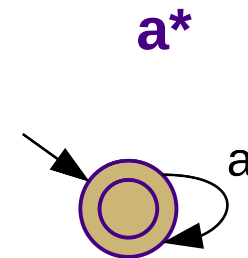


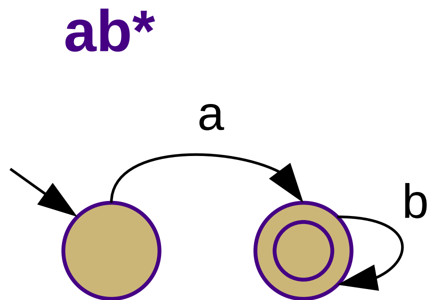
Lexical Analysis

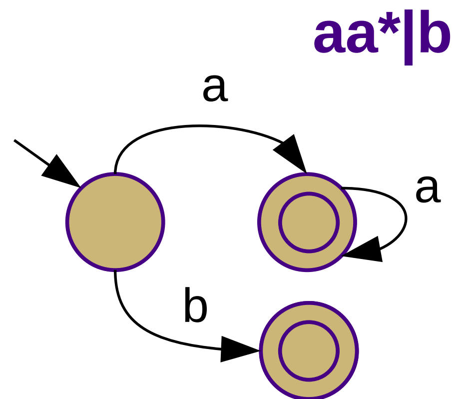
- More examples:

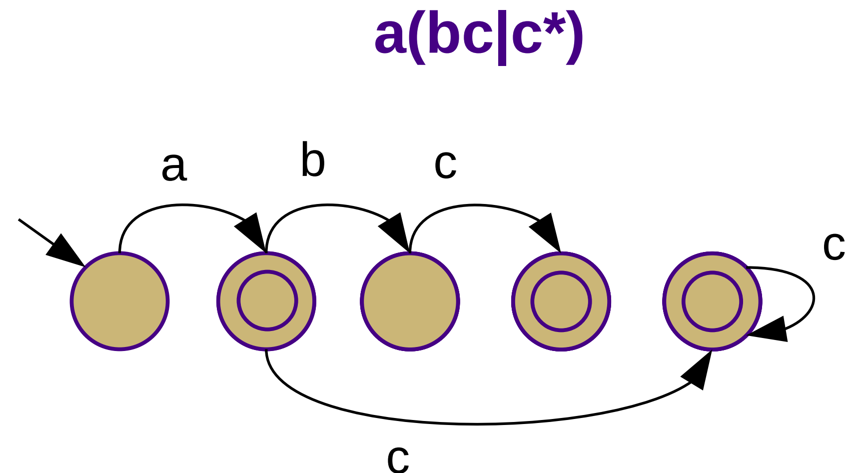












Lexing

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

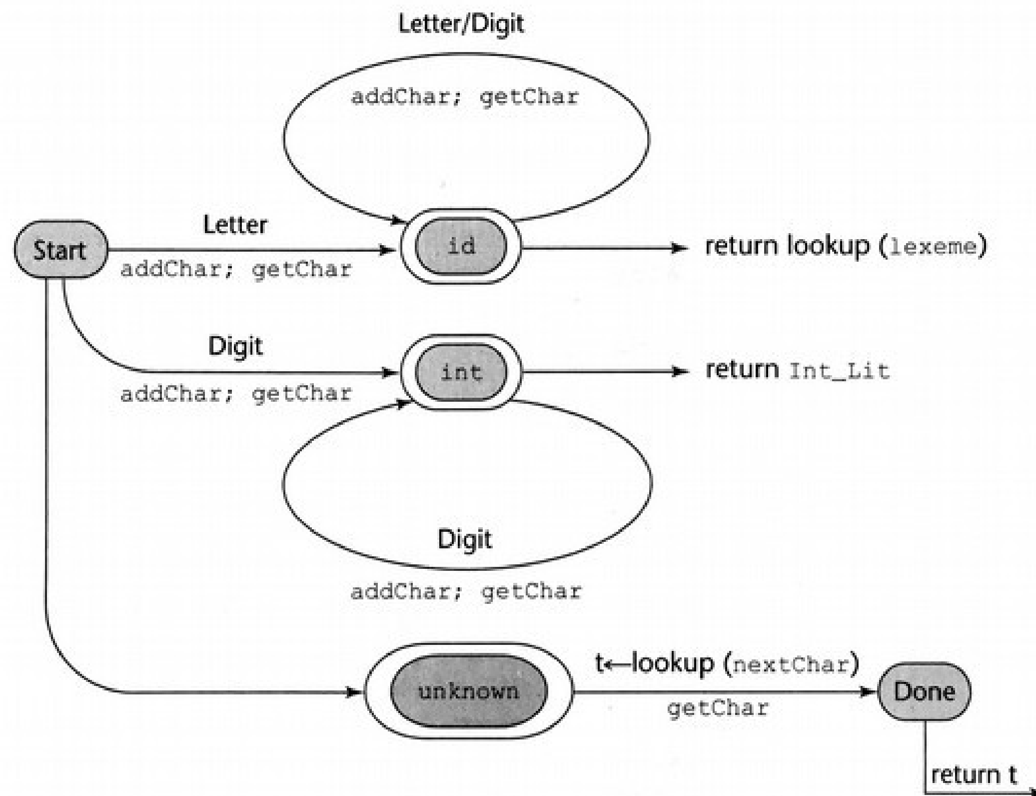


Figure from CPL 11th Ed.

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('#')
```

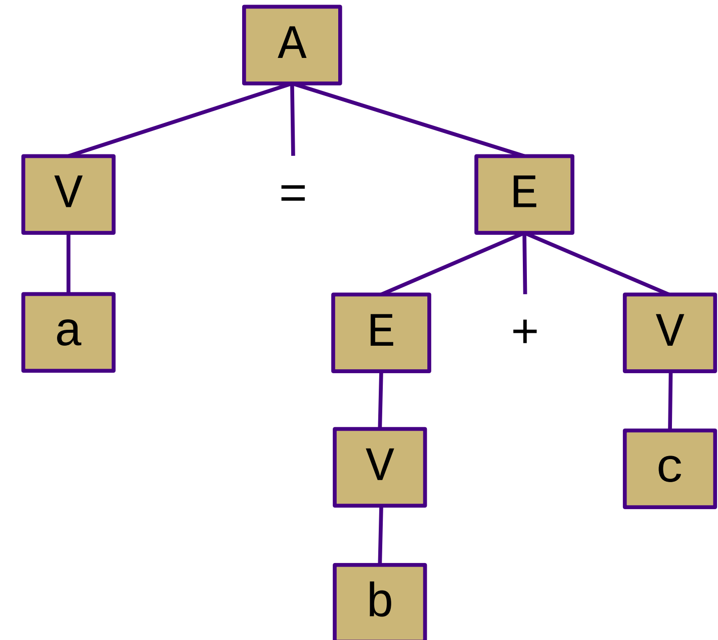
```
parseB():
    if peek() == 'x':
        consume('x')
    elif peek() == 'y':
        consume('y')
    else:
        error "Bad input: "
              + peek()
```

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!