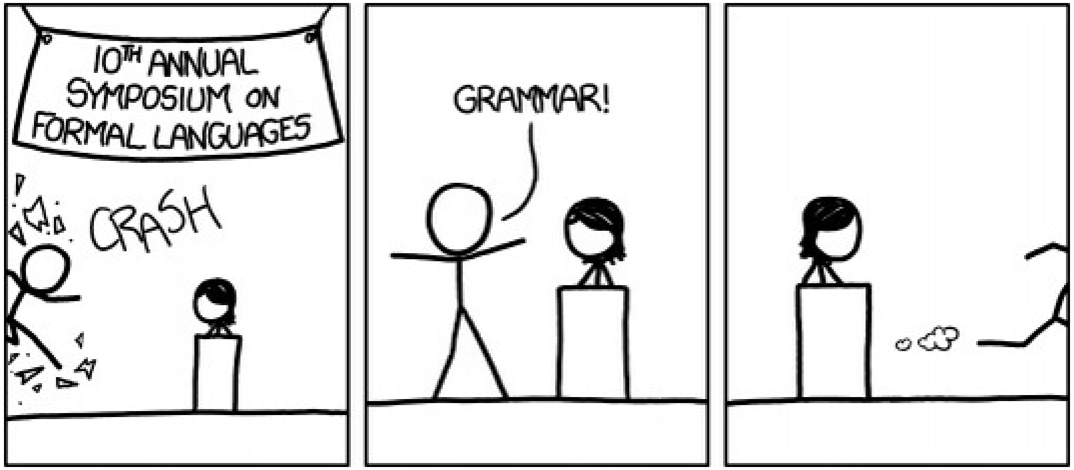
CS 432
Fall 2017

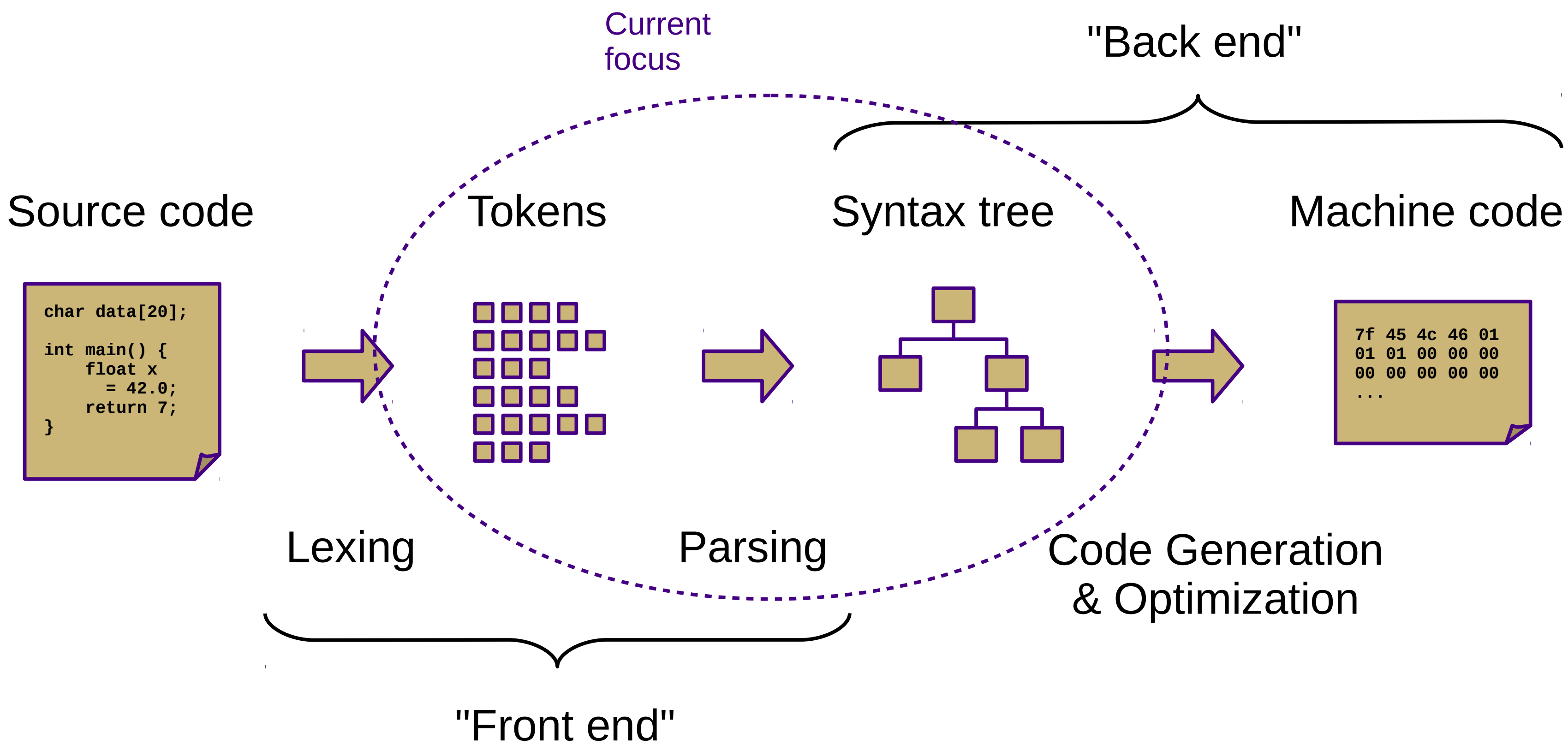
Mike Lam, Professor



[audience looks around] "What just happened?"
"There must be some context we're missing."

Context-free Grammars

Compilation



Overview

- General programming language topics
  - Syntax (what a program looks like)
  - Semantics (what a program means)
  - Implementation (how a program executes)

Syntax

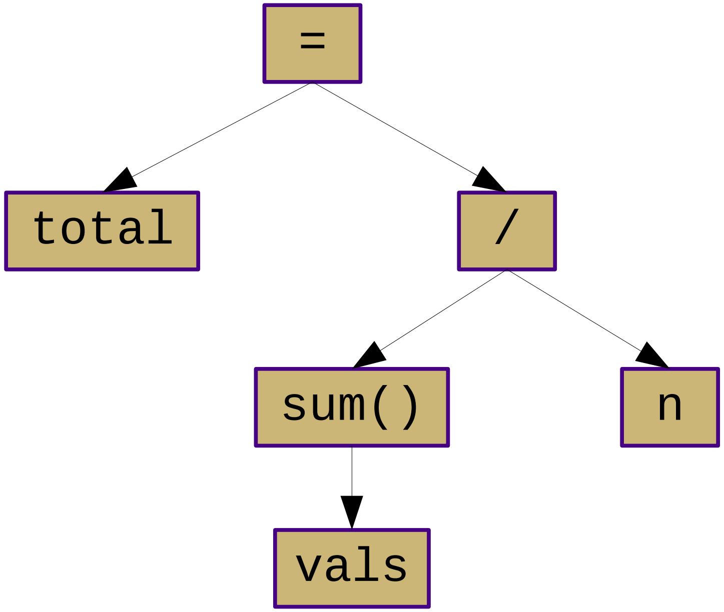
- Textbook: "the form of [a language's] expressions, statements, and program units."
  - In other words, the **form** or **structure** of the code
- Goals of syntax analysis:
  - Checking for program validity or correctness
  - Facilitate translation (compiler) or execution (interpreter) of a program

Syntax Analysis

- Tokens have no structure
  - No inherent relationship between each other
  - Need a way to describe hierarchy in a way that is closer to the *semantics* of the language

```
total = sum(vals) / n
```

```
total     identifier
=         equals_op
sum       identifier
(         left_paren
vals      identifier
)         right_paren
/         divide_op
n         identifier
```

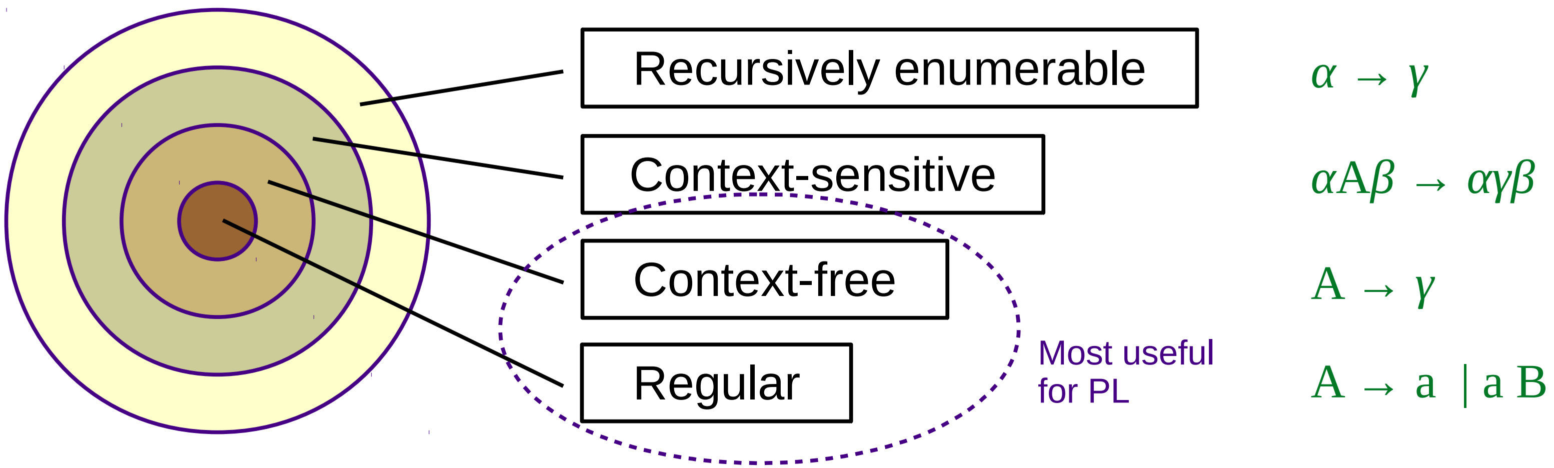


Syntax Analysis

- Context-free language
  - Description of a language's syntax
  - Encodes hierarchy and structure of language tokens
    - Usually represented using a tree
  - Described by *context-free grammars*
    - Usually written in Backus-Naur Form
  - Recognized by *pushdown automata*
    - Two major approaches: top-down and bottom-up
    - Next two weeks
  - Provide ways to control ***ambiguity***, ***associativity***, and ***precedence*** in a language

Languages

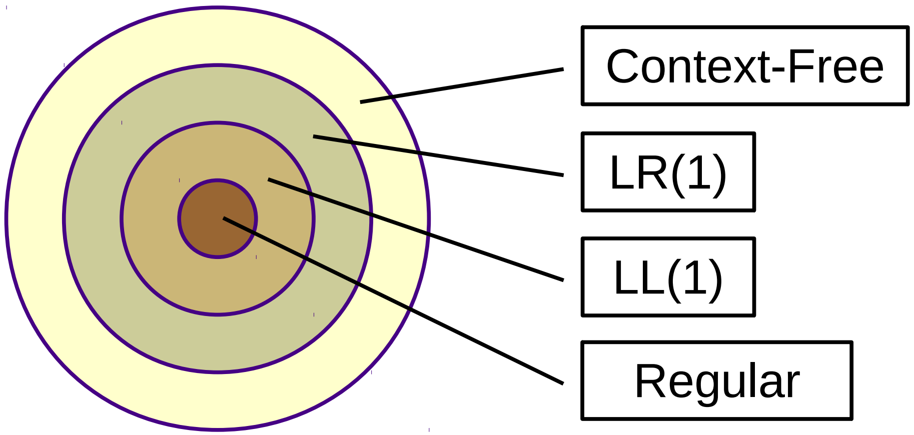
Chomsky Hierarchy of Languages



- Regular languages are not sufficient to describe programming languages
  - Core issue: DFAs can't count
  - Consider the language of all matched parentheses

Context-Free Grammars

- A context-free grammar is a 4-tuple (T, NT, S, P)
  - T: set of terminal symbols (tokens)
  - NT: set of nonterminal symbols
  - S: start symbol (S ϵ NT)
  - P: set of productions or rules:
    - NT → (T U NT)+



Context-Free Hierarchy

Backus-Naur Form

- *Non-terminals* vs. *terminals*
  - Terminals are essentially tokens
  - 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 can be replaced by the RHS (colloquially: "is composed of")
- *Sentence*: a sequence of terminals
  - A sentence is *valid* in a language if it can be derived using the grammar

```
<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
  - 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 (the sentence is the sequence of all leaf nodes)
    - Built from the top down during derivation
    - Final parse tree is called *complete* parse tree
    - 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:

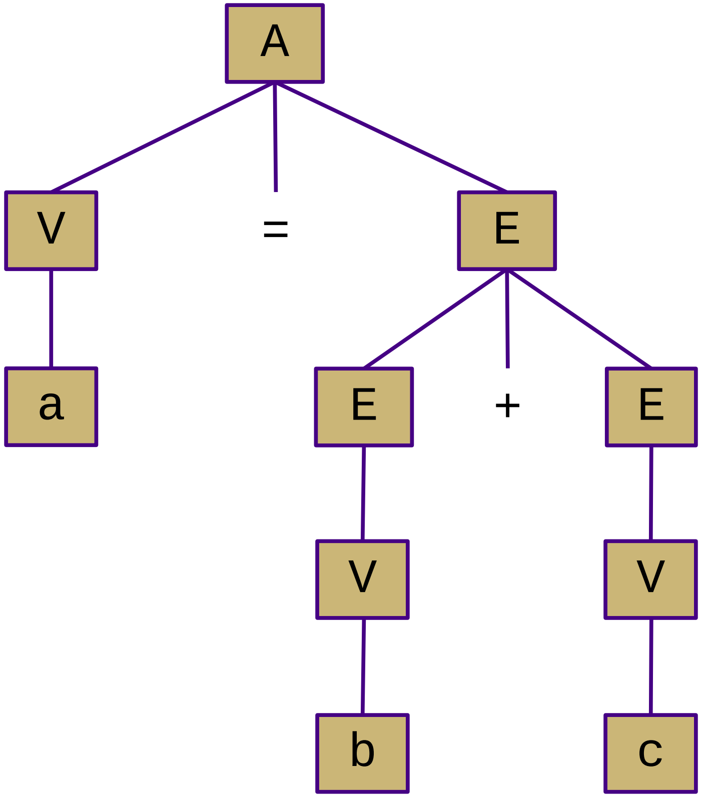
$$
\begin{aligned}
A &\rightarrow V = E \\
V &\rightarrow a \mid b \mid c \\
E &\rightarrow E + E \\
&\mid V
\end{aligned}
$$

Example

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

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

```
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, but there is a difference syntactically!
  - Example: if/then/else construct
  - It is important to be precise!
- Often can be eliminated by rewriting the grammar
  - Usually by making one or more rules more restrictive

```
A → A + A
  | x
```

Ambiguous
(Associativity)

```
A → B | C
B → x
C → x
```

Ambiguous
(Ad-hoc)

```
A → ifthen A else A
  | ifthen A
  | stmt
```

Ambiguous
("Dangling Else" Problem)

Operator Associativity

- Does x+y+z = **(x+y)**+z or x+**(y+z)**?
  - Former is left-associative
  - Latter is right-associative
- Closely related to recursion
  - Left-hand recursion → left associativity
  - Right-hand recursion → right associativity
- Sometimes enforced explicitly in a grammar
  - Different non-terminals on left- and right-hand sides of an operator
  - Sometimes just noted with annotations

Left Associative	Right Associative

A → A + x
     | x
	A → x + A
	     | x

Left Associative   **Right Associative**

Operator Precedence

- Precedence determines the relative priority of operators in a single production
- Does x+y*z = **(x+y)***z or x+**(y*z)**?
  - Former: "+" has higher precedence
  - Latter: "*" has higher precedence
- Sometimes enforced explicitly in a grammar
  - One non-terminal for each level of precedence
  - Sometimes just noted with annotations

$$A \rightarrow A + B \mid B$$
$$B \rightarrow B * x \mid x$$

Precedence
+ (lower)
* (higher)

Grammar Examples

```
A → A x
  | x
```

Left Recursive

```
A → x A
  | x
```

Right Recursive

```
A → A + B
  | B
B → B * x
  | x
```

Precedence
+ (lower)
* (higher)

```
A → A + x
  | x
```

Left Associative

```
A → x + A
  | x
```

Right Associative

```
A → A + A
  | x
```

Ambiguous
(Associativity)

```
A → B | C
B → x
C → x
```

Ambiguous
(Ad-hoc)

```
A → ifthen A else A
  | ifthen A
  | stmt
```

Ambiguous
("Dangling Else" Problem)