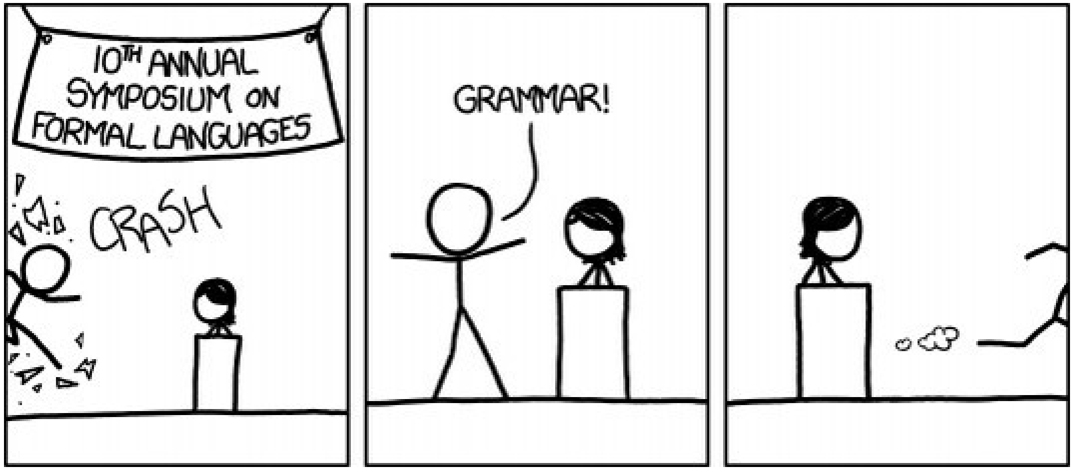
CS 432
Fall 2018

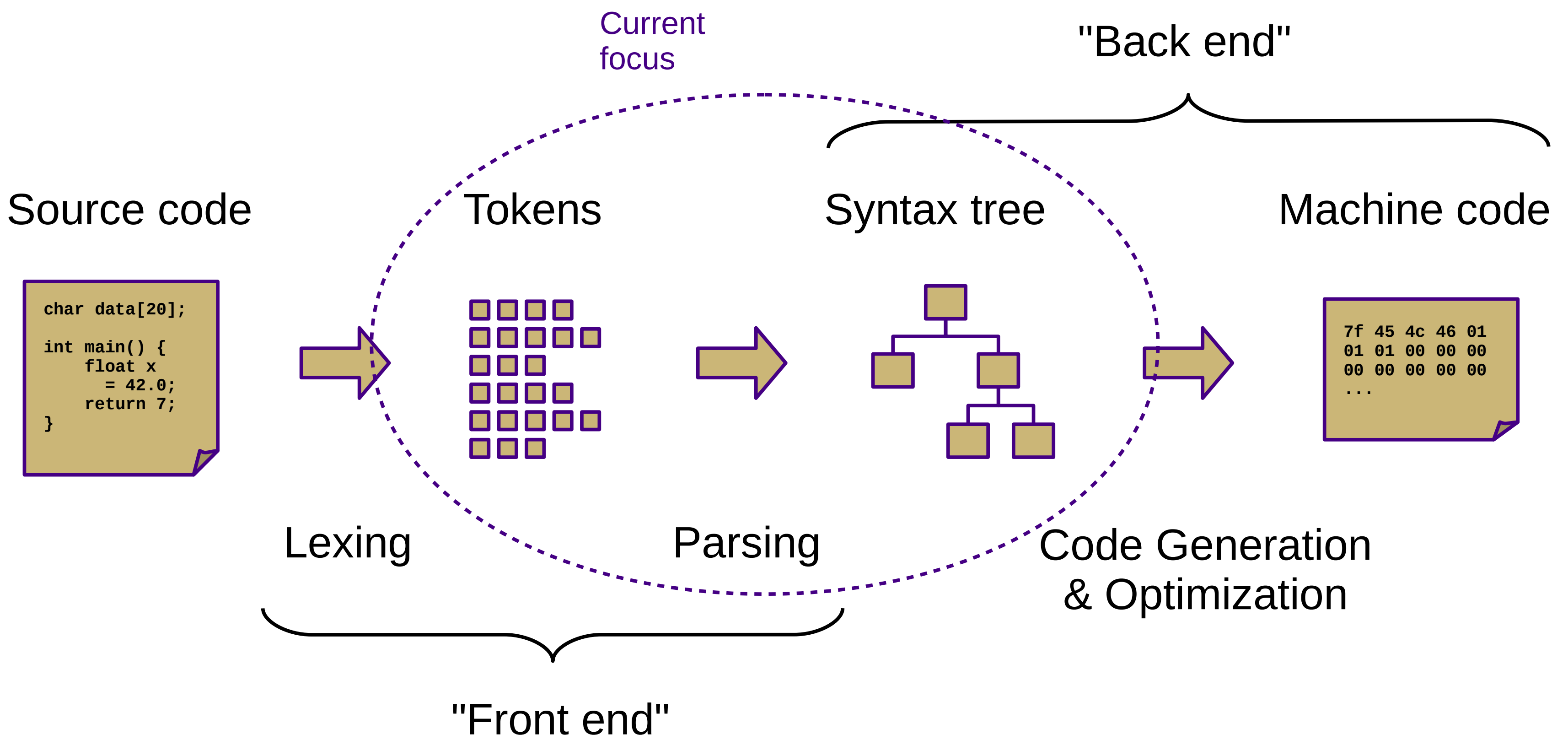
Mike Lam, Professor



[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

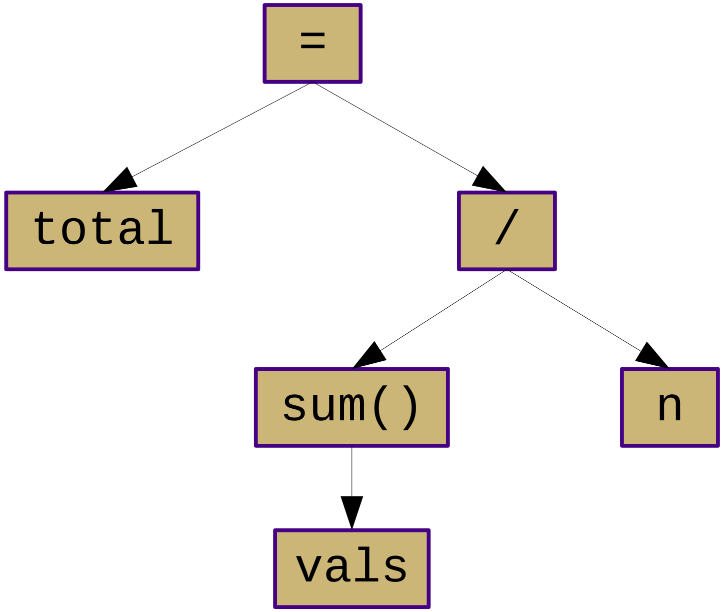
- Textbook: "the form of [a language's] expressions, statements, and program units."
  - In other words, the **form** or **structure** of the code
- Goals of syntax analysis:
  - Checking for program validity or correctness
  - Facilitate translation (compiler) or execution (interpreter) of a program

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

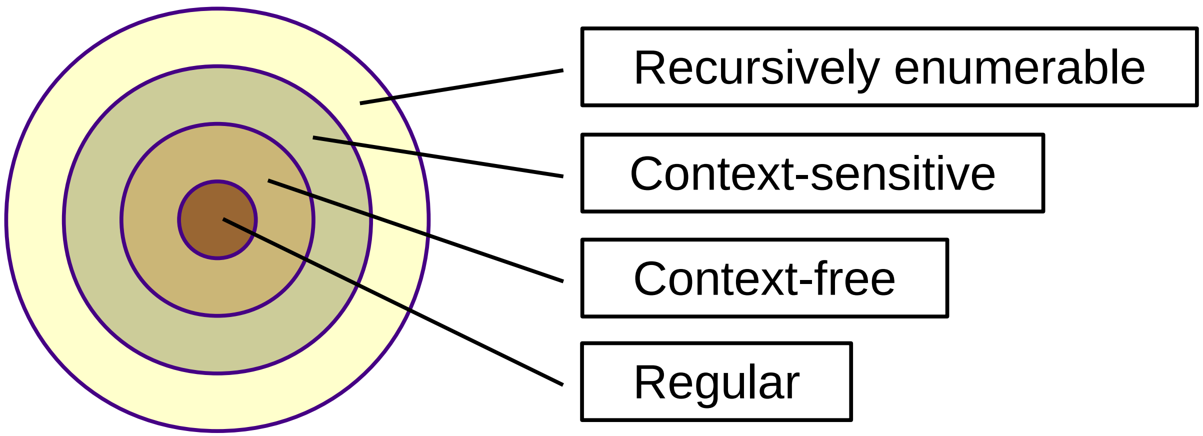
```
total = sum(vals) / n
```

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



Languages

Chomsky Hierarchy of Languages



$\alpha \rightarrow \gamma$

$\alpha A \beta \rightarrow \alpha \gamma \beta$

$A \rightarrow \gamma$

$A \rightarrow a \mid a\,B$

NOTE: Greek letters (α,β,γ) indicate arbitrary strings of terminals and/or non-terminals

- Regular languages are not sufficient to describe programming languages
  - Core issue: finite DFAs can't "count:" no way to express $a^m b^n$ where $n = f(m)$
  - Consider the language of all matched parentheses $(^n\,)^n$
  - How can we solve this to make it feasible to write a compiler?

Add memory! (and move up the language hierarchy)

Syntax Analysis

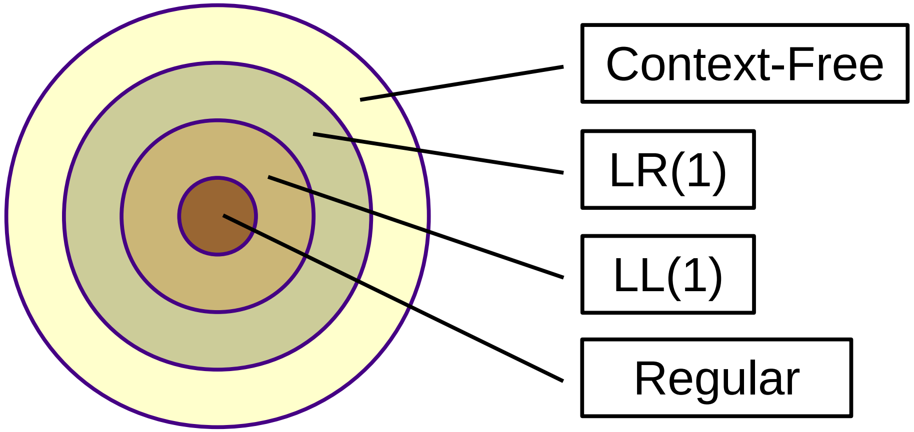
- Context-free language
  - More expressive than regular languages
  - Encodes hierarchy and structure of language tokens
    - Usually represented using a tree
  - Described by *context-free grammars*
    - Recursive description of the language's form
    - Usually written in Backus-Naur Form
  - Recognized 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

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 ∪ NT)*

Example:

S → x S x
S → y



Context-Free Hierarchy

Context-Free Grammars

- *Non-terminals* vs. *terminals*
  - Terminals are single tokens, non-terminals are aggregations
  - 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")
  - RHS can be empty (or “ε”), meaning it can be composed of nothing
- *Sentence*: a sequence of terminals
  - A sentence is a member of a language if and only if it can be derived using the language’s grammar

Context-Free Grammars

- *Derivation*: a series of grammar-permitted transformations leading to a sentence
  - Begin with the grammars start symbol (a non-terminal)
  - Each transformation applies exactly one rule
    - Expand one non-terminal to a string of terminals and/or non-terminals
    - 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
    - For a compiler: represents a program, executed from the bottom up

Context-Free Grammars

- Backus-Naur Form: list of context-free grammar rules
  - Usually beginning with start symbol
  - Convention: non-terminals start with upper-case letters
  - Combine rules using “|” operator:

```
E →  E + E
E →  V
```

```
E →  E + E
  |  V
```

```
E →  E + E | V
```

  - Several formatting variants:

```
<Assign> ::=  <Var> = <Expr>
<Var>    ::=  a | b | c
<Expr>   ::=  <Expr> + <Expr>
           |  <Var>
```

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

Example

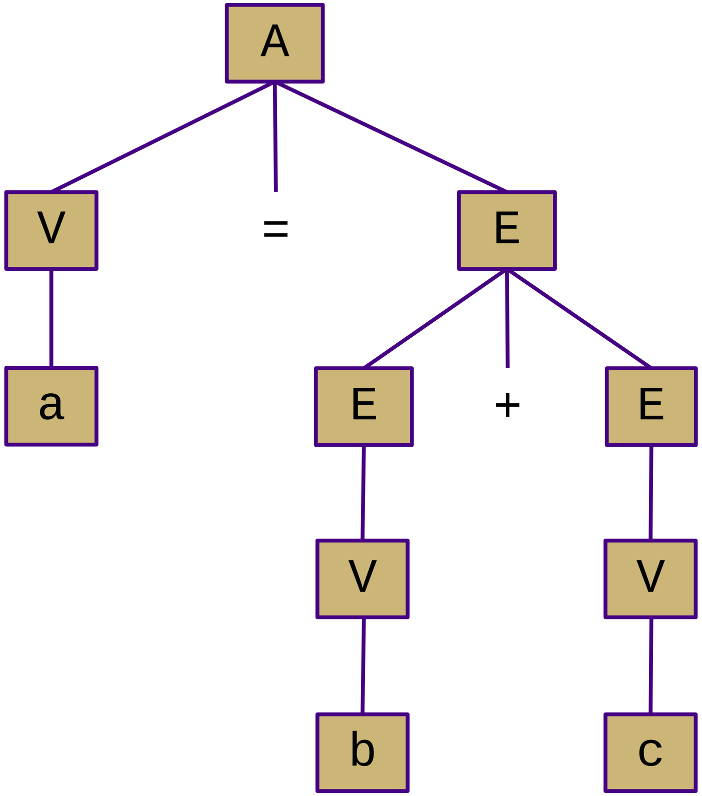
- Show the leftmost derivation and parse tree of the sentence "a = b + c" using this grammar:

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

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

Example

- Let's revisit the "matched parentheses" problem
  - Cannot write a regular expression for $\underline{(}^n\,\underline{)}^n$
  - How about a context-free grammar?
  - First attempt:

$$S \rightarrow \underline{(}\ S\ \underline{)}$$
$$S \rightarrow \varepsilon$$

Use underlining to indicate literal terminals when ambiguous

  - Second attempt:

$$S \rightarrow \underline{(}\ S\ \underline{)}\ S$$
$$S \rightarrow \varepsilon$$

What (if anything) is wrong with this:

$$S \rightarrow S\ \underline{(}\ S\ \underline{)}\ S$$
$$S \rightarrow \varepsilon$$

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
  | A * A
  | x
```

Ambiguous
(Associativity/Precedence)

```
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

Operator Precedence

- Precedence determines the relative priority of operators
- 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
    - Each level contains references to the next level
  - Sometimes just noted with annotations
  - Same approach for unary and binary operators
    - For binary operators: left or right associativity?
    - For unary operators: prefix or postfix?
    - For unary operators: is repetition allowed?

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

Precedence

+ (lowest)

* (middle)

! (highest)

Grammar Examples

```
A → A x
  | x
```

Left Recursive

```
A → x A
  | x
```

Right Recursive

```
A → A + x
  | x
```

Left Associative

```
A → x + A
  | x
```

Right Associative

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

Associativity/Precedence
+ (lowest, left-associative)
* (middle, right-associative)
! (highest, postfix unary)

```
A → A + A
  | A * A
  | x
```

Ambiguous
(Associativity/Precedence)

```
A → B | C
B → x
C → x
```

Ambiguous
(Ad-hoc)

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

Ambiguous
("Dangling Else" Problem)