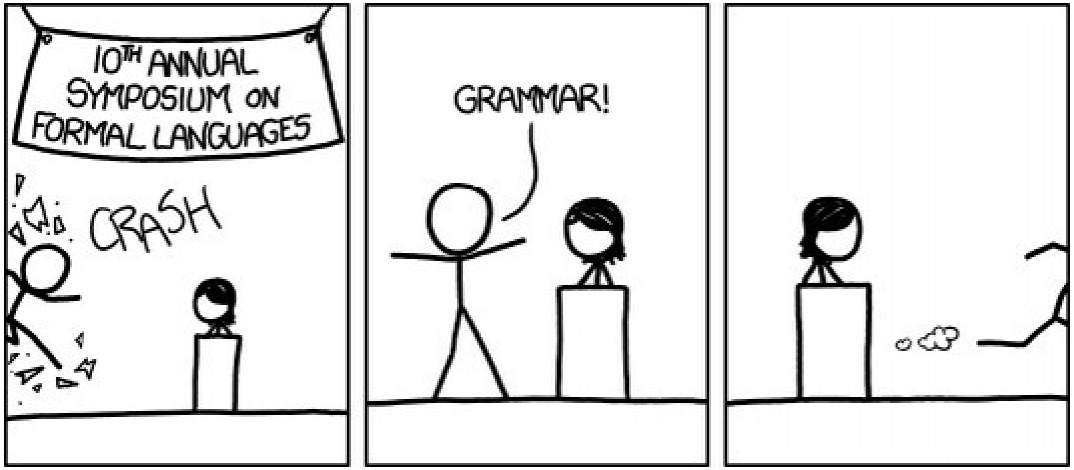
CS 432
Fall 2024

Mike Lam, Professor

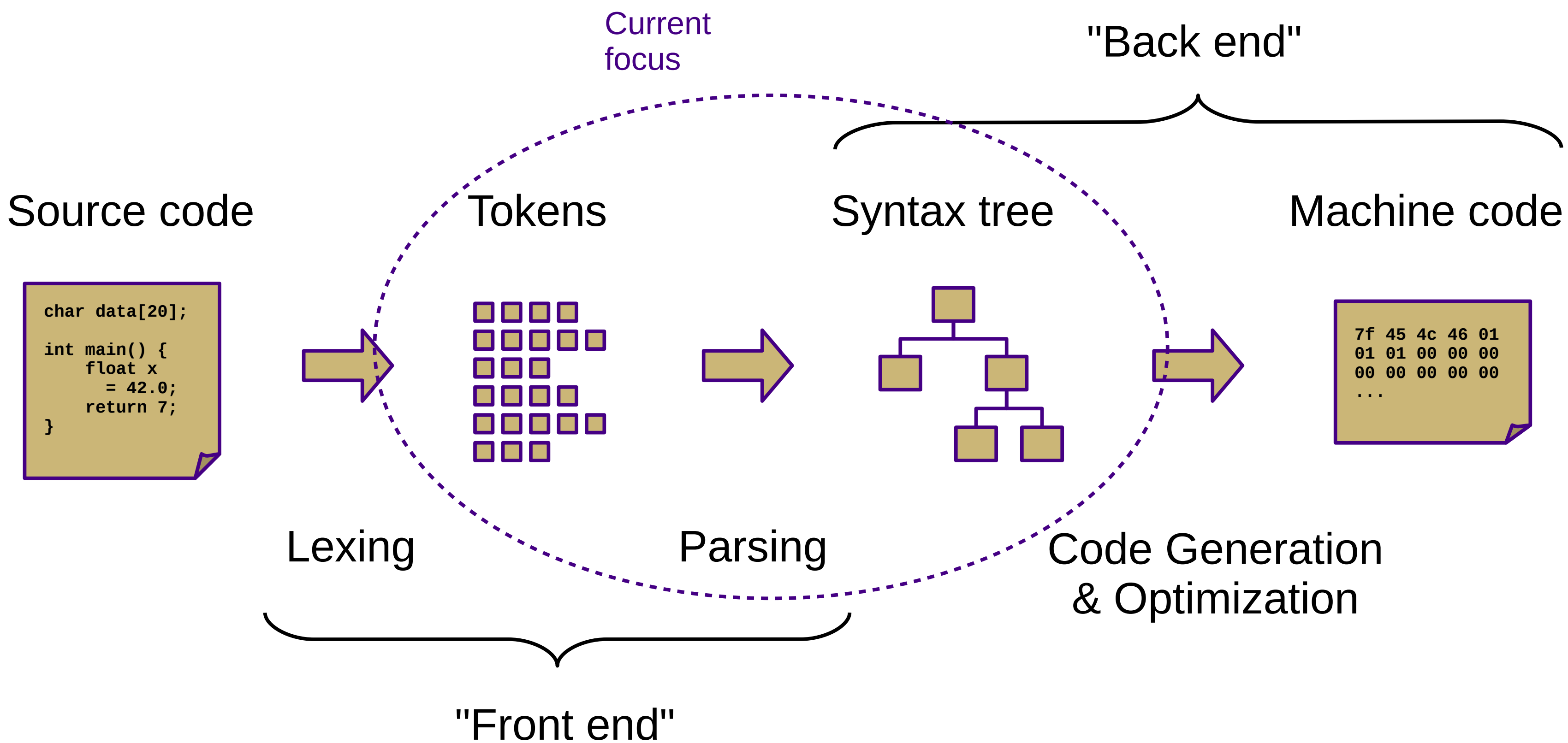
https://xkcd.com/1090/



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

Context-free Grammars

Compilation



Overview

- General programming language topics (e.g., CS 430)
  - Syntax (what a program looks like)
  - Semantics (what a program means)
  - Implementation (how a program executes)

Syntax

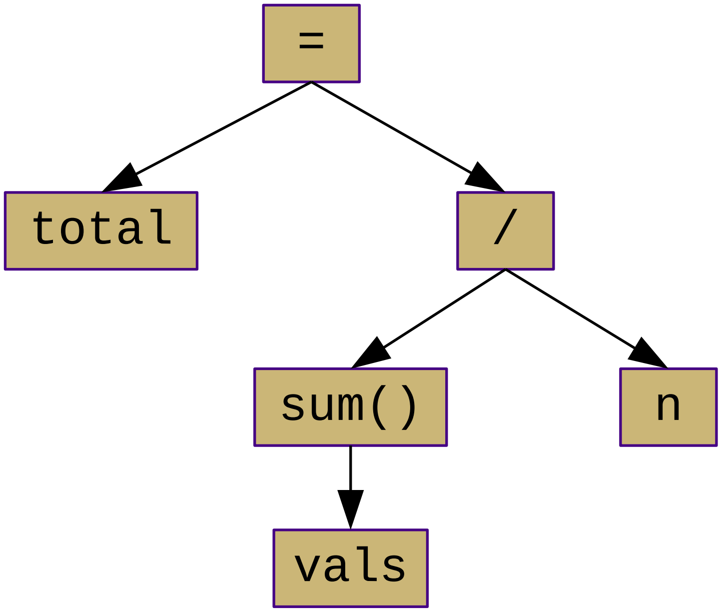
- Textbook: "the form of [a language's] expressions, statements, and program units."
  - In other words, the **form** or **structure** of the code
- Goals of syntax analysis:
  - Checking for program validity or correctness
  - Encode semantics (meaning of program)
  - Facilitate translation (compiler) or execution (interpreter)
  - We’ve already seen the first step (lexing/scanning)

Syntax Analysis

- Problem: tokens have no structure
  - No inherent relationship between each other
  - Need to make hierarchy of tokens explicit
  - Closer to the *semantics* of the language

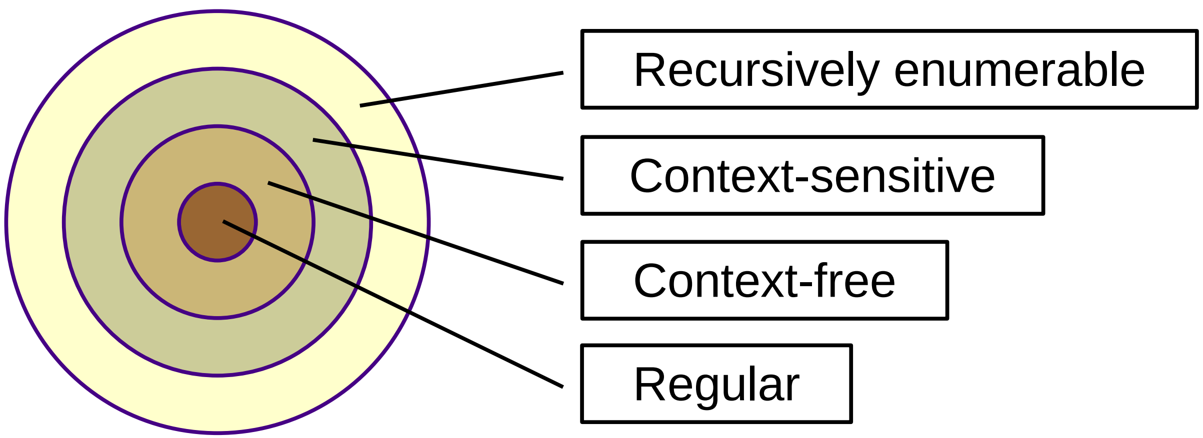
`total = sum(vals) / n`

```
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)

Languages

- Chomsky-Schützenberger representation theorem
  - A language L over the alphabet Σ is **context-free** if and only if there exists
    - a matched alphabet $T \cup \overline{T}$
    - a regular language R over $T \cup \overline{T}$
    - a mapping $h : T \cup \overline{T} \rightarrow \Sigma^*$
  - such that $L = h\,(D_T \cap R)$
  - where $D_T = \{ w \in T \cup \overline{T} \mid$ w is **a correctly-nested sequence of parenthesis** $\}$

https://en.wikipedia.org/wiki/Chomsky–Schützenberger_representation_theorem

Basically, all context-free languages can be expressed as the combination of two simpler languages: one being regular and one being composed of correctly-nested sequences of parentheses.

KEY OBSERVATION: Context-free grammars describe a wider range of languages than regular expressions, with the primary new feature being the ability to count

Languages

- Context-free languages
  - More expressive than regular languages
    - Expressive enough for “real” programming languages
  - Described by *context-free grammars*
    - Recursive description of the language’s form
    - Encodes hierarchy and structure of language tokens
    - Usually written in Backus-Naur Form
  - Recognized by *pushdown automata*
    - Finite automata + stack
    - Two major approaches: top-down and bottom-up
    - Produces a tree-based intermediate representation of a program
  - Provide ways to eliminate ***ambiguity*** and control ***associativity*** and ***precedence*** in a language’s operators

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) – usually the first non-terminal listed
  - P: set of productions or rules:
    - $NT \rightarrow (T \cup NT)^*$

Example:

$A \rightarrow x \; A \; x$
$A \rightarrow y$

T = { x, y }
NT = { A }
S = A
P = { A → x A x, A → y }

Strings in language:

y
xyx
xxyxx
xxxyxxx
(etc.)

Context-Free Grammars

- *Non-terminals* vs. *terminals*
  - Terminals are single tokens, non-terminals are aggregations
  - One special non-terminal: the *start symbol*
- Production *rules*
  - Meta-symbol operator “→” with left- and right-hand sides
  - 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

Context-Free Grammars

- *Derivation*: a series of grammar-permitted transformations leading to a sentence
  - Begin with the grammar's 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 “|” meta-symbol operator:

```
E →  E + E          E →  E + E         E →  E + E | V
E →  V                 |  V
```

  - Several formatting variants:

```
<Assign> ::=  <Var> = <Expr>            A →  V = E
<Var>    ::=  a | b | c                 V →  a | b | c
<Expr>   ::=  <Expr> + <Expr>           E →  E + E
           |  <Var>                       |  V
```

Example

- Identify parts of the following grammar:
  - Non-terminals
  - Terminals
  - Meta-symbols

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

Example

- Identify parts of the following grammar:
  - Non-terminals
  - Terminals
  - Meta-symbols

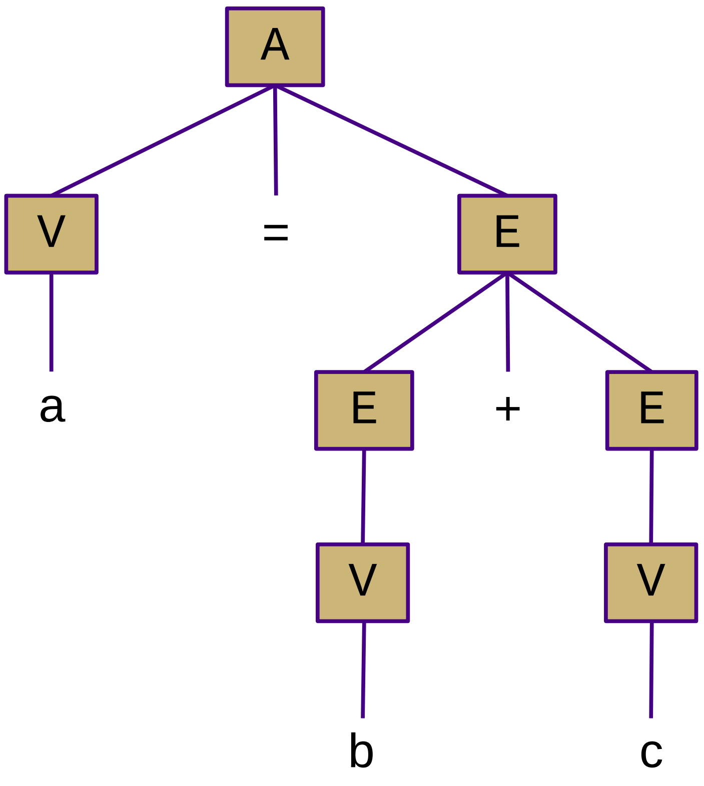
$$
\begin{array}{llllllll}
A & \rightarrow & V & = & E & & \\
V & \rightarrow & a & | & b & | & c \\
E & \rightarrow & E & + & E & & \\
 & | & V & & & &
\end{array}
$$

Example

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

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

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:

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

empty string!

Use underlining to indicate literal terminals when ambiguous

  - Second attempt:

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

What is wrong with this?

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

Example

What is wrong with this grammar? (Hint: try deriving "<u>()()</u>")

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

$S \rightarrow \varepsilon$

Ambiguous Grammars

- An ambiguous grammar allows multiple derivations (and therefore parse trees) for the same sentence
  - The syntax may be similar, but there is a difference semantically!
  - 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
- Can be enforced explicitly for binary operators in a grammar
  - Different non-terminals on left- and right-hand sides of the operator
  - Sometimes just noted with annotations

Ambiguous	Left Associative	Right Associative
$A \rightarrow A + A$ $\quad \mid x$	$A \rightarrow A + x$ $\quad \mid x$	$A \rightarrow x + A$ $\quad \mid x$

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? (! D vs. D !)
    - For unary operators: is repetition allowed? (C ! vs. D !)

$$
\begin{array}{lcl}
A & \rightarrow & A + B \\
  & | & B \\
B & \rightarrow & B * C \\
  & | & C \\
C & \rightarrow & D\ ! \\
  & | & D
\end{array}
$$

Precedence
+ (lowest)
***** (middle)
! (highest)

Grammar Examples

```
A → A x
  | x
```

Left Recursive

```
A → x A
  | x
```

Right Recursive

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

Associativity/Precedence

+ (lowest, binary, left-associative)

* (middle, binary, right-associative)

! (highest, unary, postfix, non-repeatable)

```
A → A + x
  | x
```

Left Associative

```
A → x + A
  | x
```

Right Associative

```
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)