CS 131 Discussion

Week 1: hello(world)

matt wang (he/him)
✉️ matt@matthewwang.me
🔗 mattxw.com/131slides
📣 mattxw.com/131feedback

Discussion Agenda

1. intros!
2. my commitment to you
3. intro survey + hi-chews
4. what to review + content block 1
5. person of the week

   ~ break ~

6. language of the week
7. content block 2
8. hw 1 review + related problems

snacc of the week

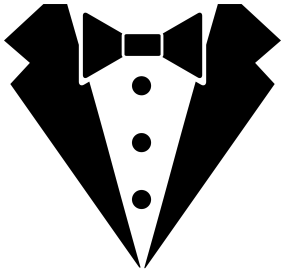
matt wang (he/him)

- 1st-year masters student (CS / ESAP)
- undergrad @ UCLA (B.S. CS, B.S. Math-Econ)
- super super passionate about cs education
  - teaching for last 9 years
  - pedagogy, tooling, EDI, everything!!
- was involved in ACM at UCLA / QWER Hacks / BEAM
- current research: CS Education, Probabilistic PL
- prev interned at CZI, FB, Adobe, AWS, Booz Allen, startup

oh … and I
love programming languages!!!

related work:

Sorbet
types for Ruby



Stylelint
CSS Linter

no logo … yet!

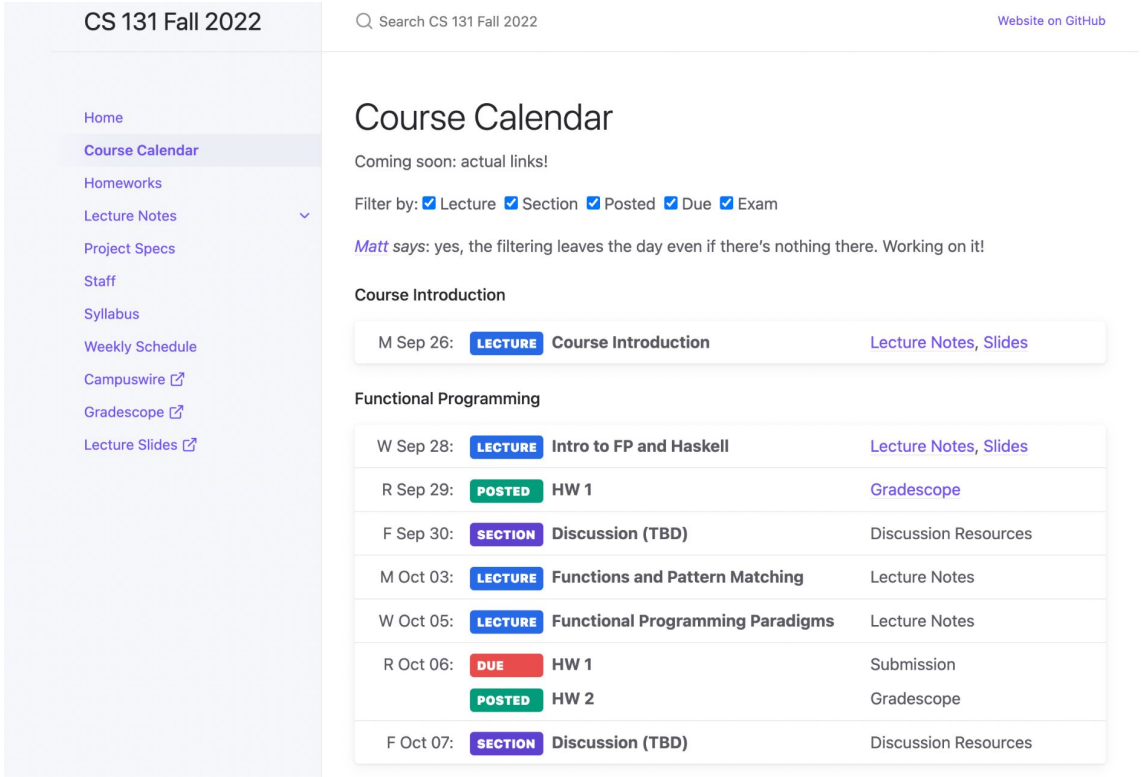
dice/rsdd
Probabilistic PL!

some logistics

- email: matt@matthewwang.me
- slides folder: mattxw.com/131slides
- office hours: 11 AM - 12 PM PT M/W in Boelter 3256-S
  - rare cases, can do by appt: mattxw.com/cal
  - **but, check with me first please!!**
  - considering online OH as well - will let you know!
- always-open feedback form: mattxw.com/131feedback
- **am a mandated reporter**

Re: enrollment

- I cannot help you with PTEs, etc. :(
- I don't *really* care if you're enrolled in my section, *but* that may change if the class physically gets too full.



quick plug - class website

https://ucla-cs-131.github.io/fall-22/

your turn!

name, pronouns (optional),
last thing you read/watched/listened to

My Commitment to You & SLAs

Above all, **I really care about your experience!** What does that mean?

- timely communication
- bi-directional feedback
- encouraging & inclusive environment
- COVID safety

To make these concrete, I'll provide you with some SLAs (service-level agreements).

SLA: Timely Communication

I will:

- answer questions in discussion and office hours
  - all questions are fair game!
  - includes non-course material (ex internships) – but I will answer course material first
- check CampusWire & email at least once a day
- respond to active threads (CW or email) in < 24 hours

SLA: Bi-directional Feedback

I'll take feedback from you and iterate, via:

- (anonymous) weekly check-in form + any-time form
- in-person during section or office hours
- virtually via email or CampusWire

I'll also give you any feedback you ask for (within reason)!

- particularly - includes homework and exams!

Interlude: what has *already* come from feedback?

- transparent autograding (incl. instant feedback)
  - large ask from CS Town Halls over the past few years
  - I wrote a chunk of it :) it's open-source and self-runnable
- "practical" / "breadth" content in discussion
  - anecdotal ask from many students during my undergrad
- lecture notes & "open" content
  - lecture notes – driven by Carey's CS 32 and positive feedback
  - "open content" – helpful for later review, improving on course over time
- this portion of this slide deck!!
  - common course feedback: unclear discussion/TA expectations
  - desire for clear paths of feedback, actively inclusive environments

SLA: Encouraging & Inclusive Environment

- **Inclusive participation**
  - Letting you participate on your own terms, with different avenues!
  - Comfortable and responsive learning environment!
  - Includes: async & digital methods, when possible
- **Digital accessibility -** through alt text, web content, contrast, advocacy :)
- General **DEI** goals!
  - Inclusive language!
  - Embracing & emphasizing diversity!
  - Equity in conduct and grading

SLA: COVID Safety

I:

- am vaccinated + boosted
- do PCR tests twice a week
- will *always* teach with a mask on
- will always accommodate for students who miss discussion due to *any* illness!

What do I expect from you?

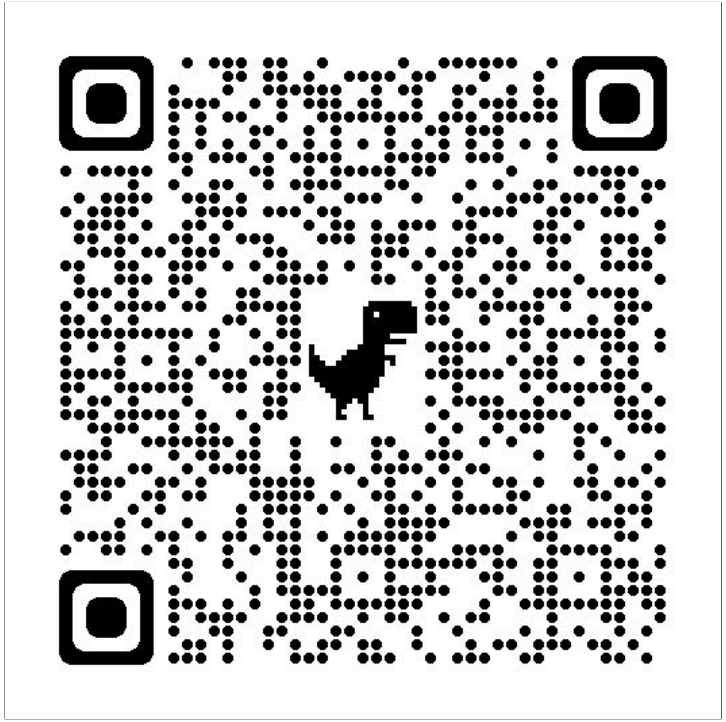
- Biggest thing: **tell me what you want!**
  - Give me feedback :)
  - Show me what works by **actively engaging**
  - Communication is key!
- Help me create an **inclusive environment.**
  - Things I never want to hear:
    - “What a stupid question” or “This person is so dumb”
    - “You only know HTML? That’s not a real programming language”
  - Please be respectful to each other :)
- **Academic honesty**

anything I’m missing?

(or questions, etc.)

~ intro survey ~

once you're done,
I'll throw you a hi-chew



https://forms.gle/PEvhEU8FpP95edZA7

What do we want to cover today?

I have slides / can talk about:

- Overarching PL concepts
  - compilers, interpreters, transpilation, linkers, higher/lower-level languages
- Haskell basics
  - up to lecture content - syntax, lists, functions, pattern matching
- Review other class content
  - 32, 111, 180, etc.
  - **recursion**!

In general (and today), I will **always review this week's homework** and **go over similar problems!**

content block 1

(watch matt fumble as he switches slides)

person of the week

person of the week

(wait ... why??)

with person of the week...

- highlight computer science history
- spotlight the **diversity** of computing pioneers
- hopefully provide a bit of inspiration!!

person of the week

Grace Hopper (1906 – 1992)

- PhD in Math from Yale, prev Professor at Vassar
- Joined US Navy Reserves at 34 (1943)
  - computation project at Harvard
  - 42-year Navy career, highest non-combat honors
- **Pioneer in compilers** (and, programming languages!)
  - Developed A-0 linker for UNIVAC
  - Directed creation of early compiled languages
  - **Largely influential in creation of COBOL;** pushed for **human-readable languages**
  - Credited for “debugging” etymology
- Why now?
  - The namesake of the Grace Hopper Celebration!
  - Soon, you’ll be developing an interpreter for a language – quite related to her early work!

Grace Hopper in front of a UNIVAC, with a COBOL manual. Computer History Museum, 1961.

Grace Hopper (1906 – 1992)

“It’s much easier for most people to write an English statement than it is to use symbols”

“So I decided data processors ought to be able to write their programs in English, **and the computers would translate them into machine code.** That was the beginning of COBOL (common business-oriented language), a computer language for data processors.”

Commodore Grace Hopper.
USN / Wikimedia, 1984.

~ break ~

discussion will resume at 11:05

language of the week

language of the week

(wait ... why??)

with language of the week...

- tie theory & course concepts to real world
- emphasis on rapid development of PLs
- give you conversational familiarity & inspiration!
- matt **loves** programming languages lol

language of the week

ML & OCaml

We just learned about Haskell (and functional programming)!

ML is *the* influential functional programming language!

- Direct influence to Miranda and thus Haskell
- Popularized various functional paradigms, including:
  - **type inference**
  - **pattern matching**
  - currying
  - algebraic data types & parametric polymorphism
- Developed in 1973 by Robin Milner

OCaml is one popular implementation of ML

- Developed by Inria (~ French DARPA) in 1996
- Heavily invested-in by Jane Street

Why ML & OCaml?

History and influence:

- Functional programming is not “new”!
- Languages often have lineages, dialects, and various implementations
- Languages are often developed at companies *and* academia!

OCaml is widely used in PL tooling (compilers, interpreters), formal methods, analysis.

Major projects: Coq, various Facebook-related type retrofitters (Hack, Flow, pyre), Haxe.

We used to cover OCaml in 131!

Compared to Haskell, ML/OCaml are **different approaches to functional programming!**

Compared to Haskell:

- **strict by default (opposed to lazy)**
  - i.e. arguments are always evaluated first
  - what’s the tradeoff here?
- impure
  - allows imperative layers on top, “practical”
- other complicated stuff
  - Polymorphic variants, type classes, module system, parallelism

And … large performance implications!

Quicksort in OCaml – similar to Haskell :)

```
let rec qsort = function
    | [] -> []
    | pivot :: rest ->
      let is_less x = x < pivot in
      let left, right = List.partition is_less rest in
       qsort left @ [pivot] @ qsort right
```

content block 2

(watch matt fumble as he switches slides)

general hw review philosophy

- will not give full answers for this week's hw
  - but! will often give hints :)
- will always give answers for prev hws
- will cover problems in class that:
  - test similar concepts to HW & project
  - matt thinks will be on midterm/exam
- very down to cover things more in-depth in OH!

overview: this week's HW!

- installing ghc/ghci & python
- haskell syntax
- recursion and mutual recursion
- lists & list comprehension
- control flow (emphasis: **pattern matching**)

question 1

Write a Haskell function named `largest` that takes in 2 `String` arguments and returns the longer of the two. If they are the same length, return the first argument.

Example:
`largest “cat” “banana”` should return `“banana”`.
`largest “Carey” “rocks”` should return `“Carey”`.

question 2

Barry Snatchenberg is an aspiring Haskell programmer. He wrote a function named `reflect` that takes in an `Integer` and returns that same `Integer`, but he wrote it in a very funny way:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
  | num < 0 = (-1) + reflect num+1
  | num > 0 = 1 + reflect num-1
```

He finds that when he runs his code, it always causes a stack overflow (infinite recursion) for any non-zero argument! What is wrong with Barry's code (i.e. can you fix it so that it works properly)?

question 3a

Write a Haskell function named `all_factors` that takes in an `Integer` argument and returns a list containing, in ascending order, all factors of that integer. You may assume that the argument is always positive. **Your function's implementation should be a single, one-line list comprehension.**

Example:
`all_factors 1` should return `[1]`.
`all_factors 42` should return `[1, 2, 3, 6, 7, 14, 21, 42]`.

question 3b

A perfect number is defined as a positive integer that is equal to the sum of its proper divisors (where “proper divisors” refers to all of its positive whole number factors, excluding itself). For example, 6 is a perfect number because its proper divisors are 1, 2 and 3 and 1 + 2 + 3 = 6.

Using the `all_factors` function, write a Haskell expression named `perfect_numbers` whose value is a **list comprehension** that generates an infinite list of all perfect numbers (even though it has not been proved yet whether there are infinitely many perfect numbers 😉).

Example:
`take 4 perfect_numbers` should return `[6, 28, 496, 8128]`.

Hint: You may find the init and sum functions useful.

question 4

Write a pair of Haskell functions named `is_odd` and `is_even` that each take in 1 `Integer` argument and return a `Bool` indicating whether the integer is odd or even respectively. You may assume that the argument is always positive.

You may not use any builtin arithmetic, bitwise or comparison operators (including `mod`, `rem` and `div`). You may only use the addition and subtraction operators (+ and -) and the equality operator (==).

You must implement THREE versions of these functions: (1) with regular if statements, (2) using guards, and (3) using pattern matching.

Example:
`is_even 8` should return `True`.
`is_odd 8` should return `False`.

Hint: The functions can call one another in their implementations. (This is called mutual recursion).

question 5

Write a function named `count_occurrences` that returns the number of ways that all elements of list a_1 appear in list a_2 in the same order (though a_1's items need not necessarily be consecutive in a_2). The empty sequence appears in another sequence of length `n` in 1 way, even if `n` is 0.

Examples:

`count_occurrences [10, 20, 40] [10, 50, 40, 20, 50, 40, 30]` should return 1.
`count_occurrences [10, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 2.
`count_occurrences [20, 10, 40] [10, 50, 40, 20, 50, 40, 30]` should return `0`.
`count_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 3.
`count_occurrences [] [10, 50, 40, 20, 50, 40, 30]`
should return 1.
`count_occurrences [] []` should return 1.
`count_occurrences [5] []` should return `0`.

guided exercises

warmup: type bonanza

```
:t [1..10]
:t [x^2 | x <- [1..100]]
let f x = ‘hi ’ ++ x
:t f
:t sum
:t `div`
```

recall: what does this comprehension do?

```
some_list = [
   x | x <- [1..100],
   ((mod x 3) == 0) ||
   ((mod x 5) == 0)
]
```

recall: what about this one?

```
some_list = [
   x - y | x <- [1..100], y <- [1..x]
   x + y == 100
]
```

hw: mutual recursion

```
is_odd x = if x == 1
             then True
             else is_even x - 1

is_even x = if x == 0
             then True
             else is_odd x - 1
```

hw: mutual recursion

```
is_odd x = if x == 1
               then True
               else is_even (x-1)

is_even x = if x == 0
               then True
               else is_odd (x-1)
```

hw: mutual recursion

```
is_odd x = if x == 1
               then True
               else is_even (x-1)

is_even x = if x == 0
               then True
               else is_odd (x-1)
```

hw: mutual recursion

```
is_odd x = if x == 0
              then False
              else is_even (x-1)

is_even x = if x == 0
              then True
              else is_odd (x-1)
```

pattern matching in short

```
factorial :: (Int a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

warmup: naive fibonacci, but ... wrong?

```
fib :: (Int n) => n -> n
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

this breaks ... but how, why, and how do we fix it?

warmup: naive fibonacci, but ... wrong?

```
fib :: (Int n) => n -> n
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

hint: what happens with fib 2?

warmup: naive fibonacci, but ... wrong?

```
fib :: (Int n) => n -> n
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

all good!

challenge problem: do not comprehend

```
f :: Int -> Int -> [Int]
f n d = ...

-- equal to list comprehension
[x | x <- [1..n], (mod x d) == 0]

-- BUT... no comprehensions!!
```

challenge problem: stack overflow

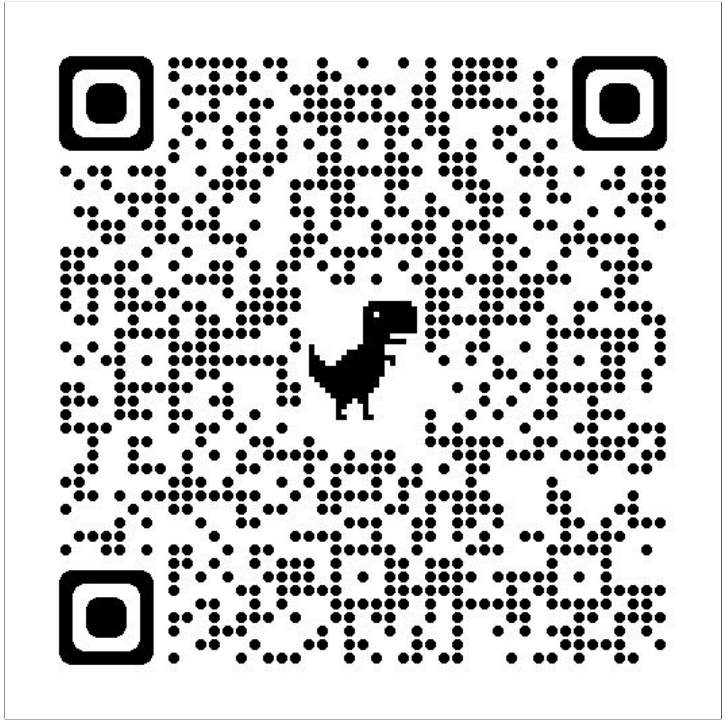
```
fib :: (Int n) => n -> n
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

I lied – not all good, and in particular, not performant.
Why? How would we implement this?
(hint: CS 33)

~ post-discussion survey ~

it's only 2 questions and optional thoughts!!

see you next week <3



https://forms.gle/33gPkKDfajrrQrZ88

appendix

challenge problem solution(s)

challenge problem – single function

```
f n d =
   if n == 0
      then []
      else f (n-1) d ++
         if (mod n d) == 0
            then [n]
            else []
```

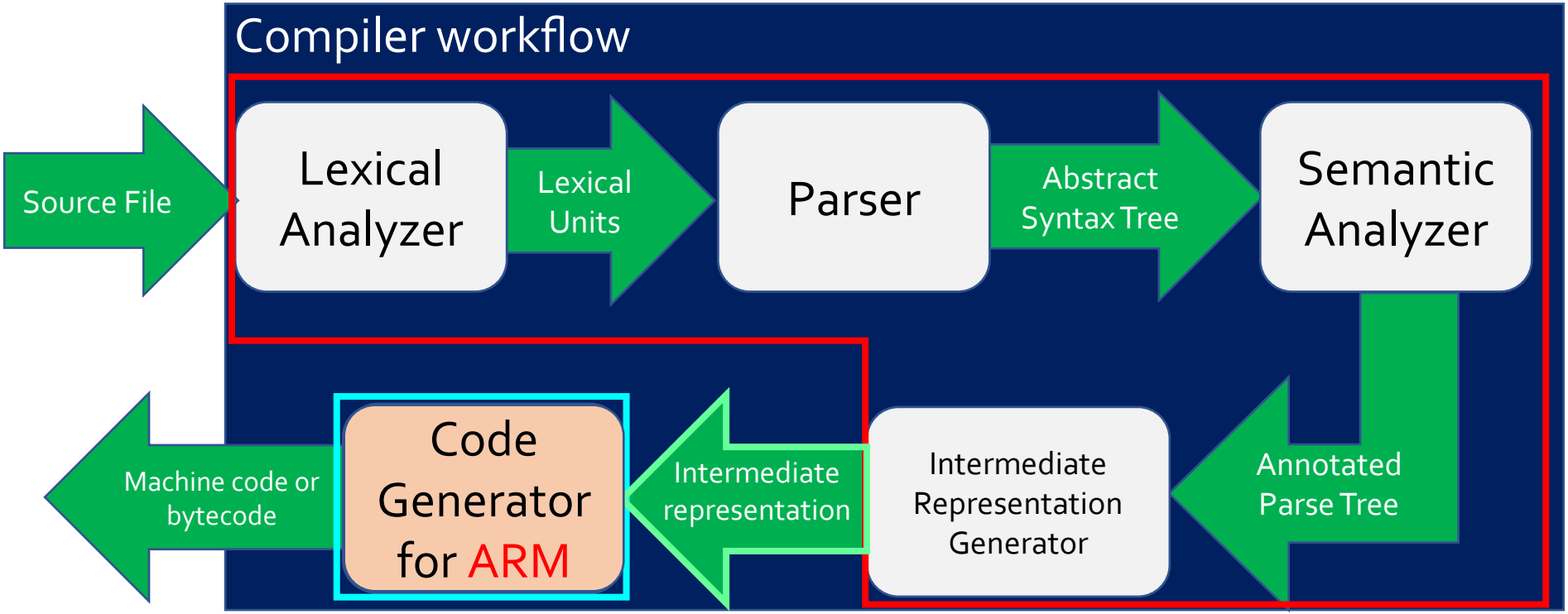
challenge problem – multiple functions

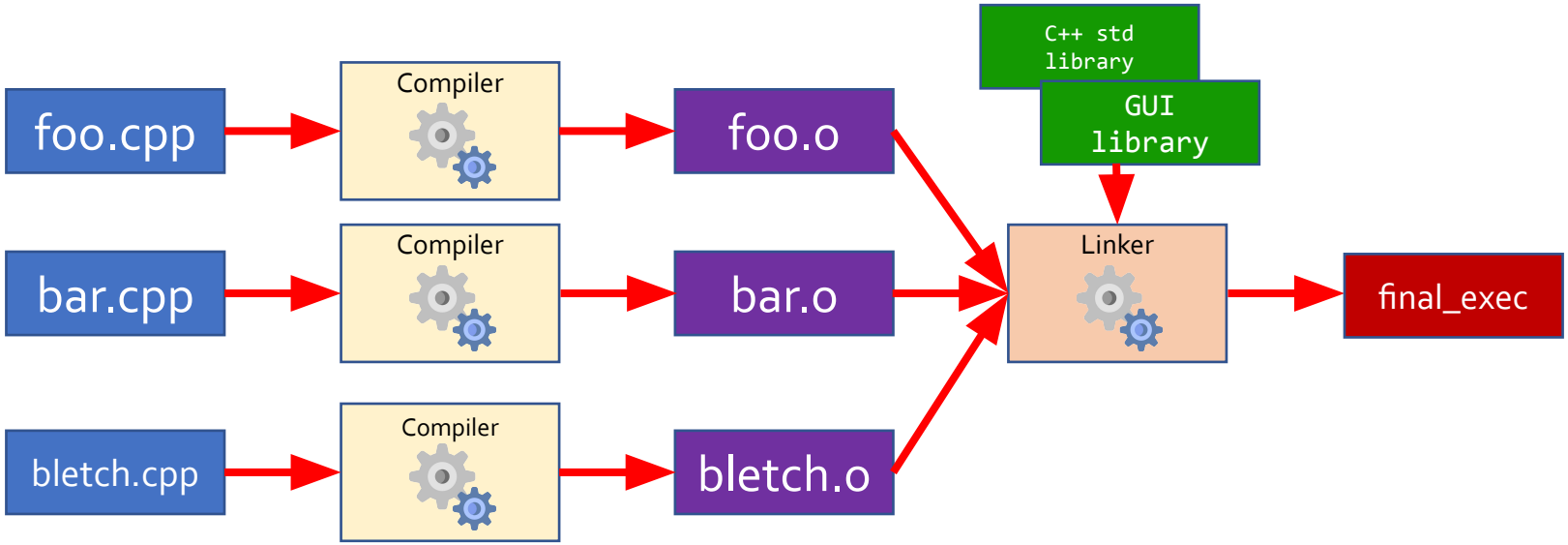
```
g lst d = if null lst
   then []
   else if (mod (head lst) d) == 0
      then (head lst) : g (tail lst) d
      else g (tail lst) d

f n d = g [1..n] d
```

challenge problem – w/ tail recursion

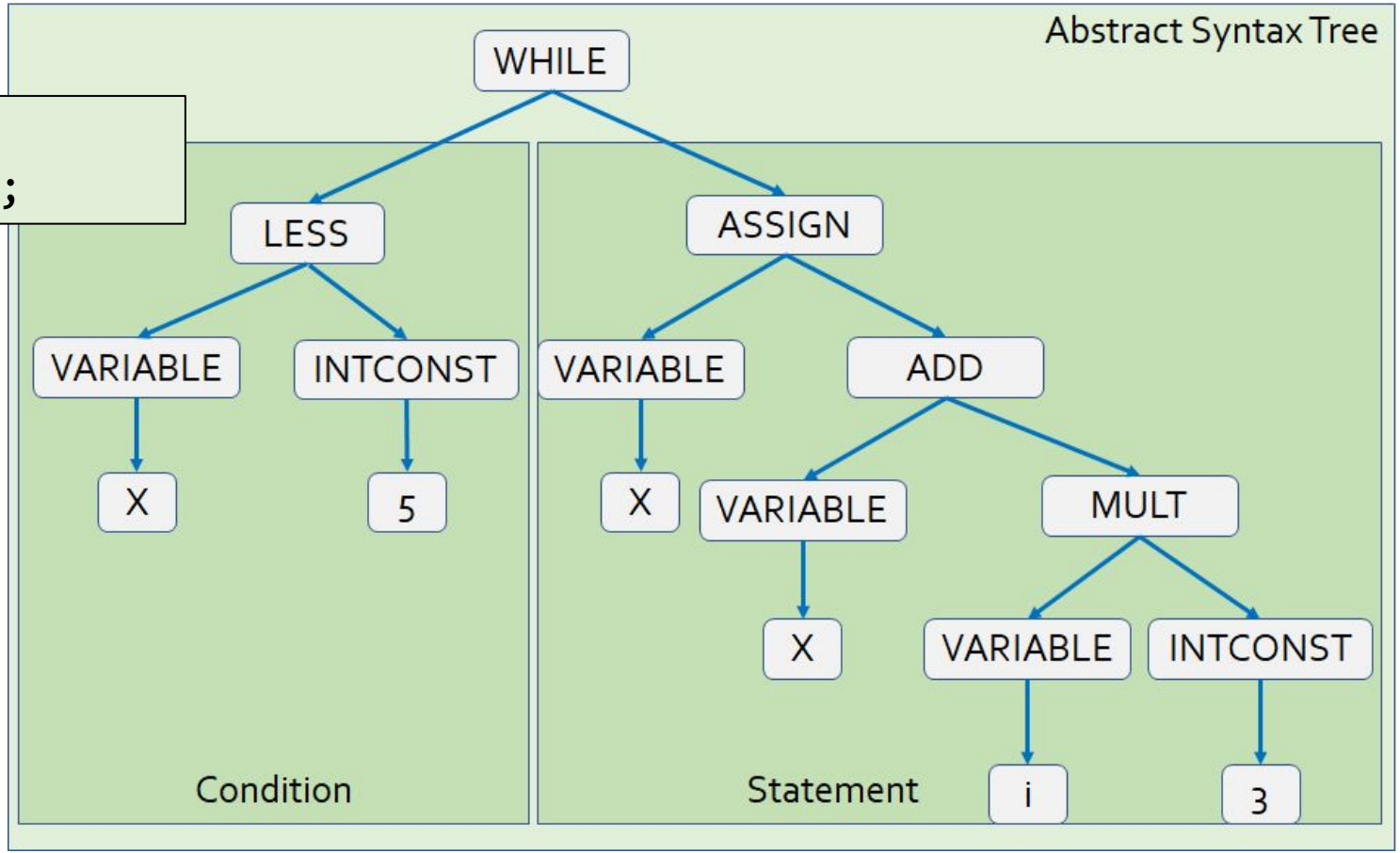
```
g lst accum d = if null lst
  then accum
  else g (tail lst) (accum ++ (
    if (mod (head lst) d) == 0
      then [(head lst)] else []
  )) d
f n d = g [1..n] [] d
```

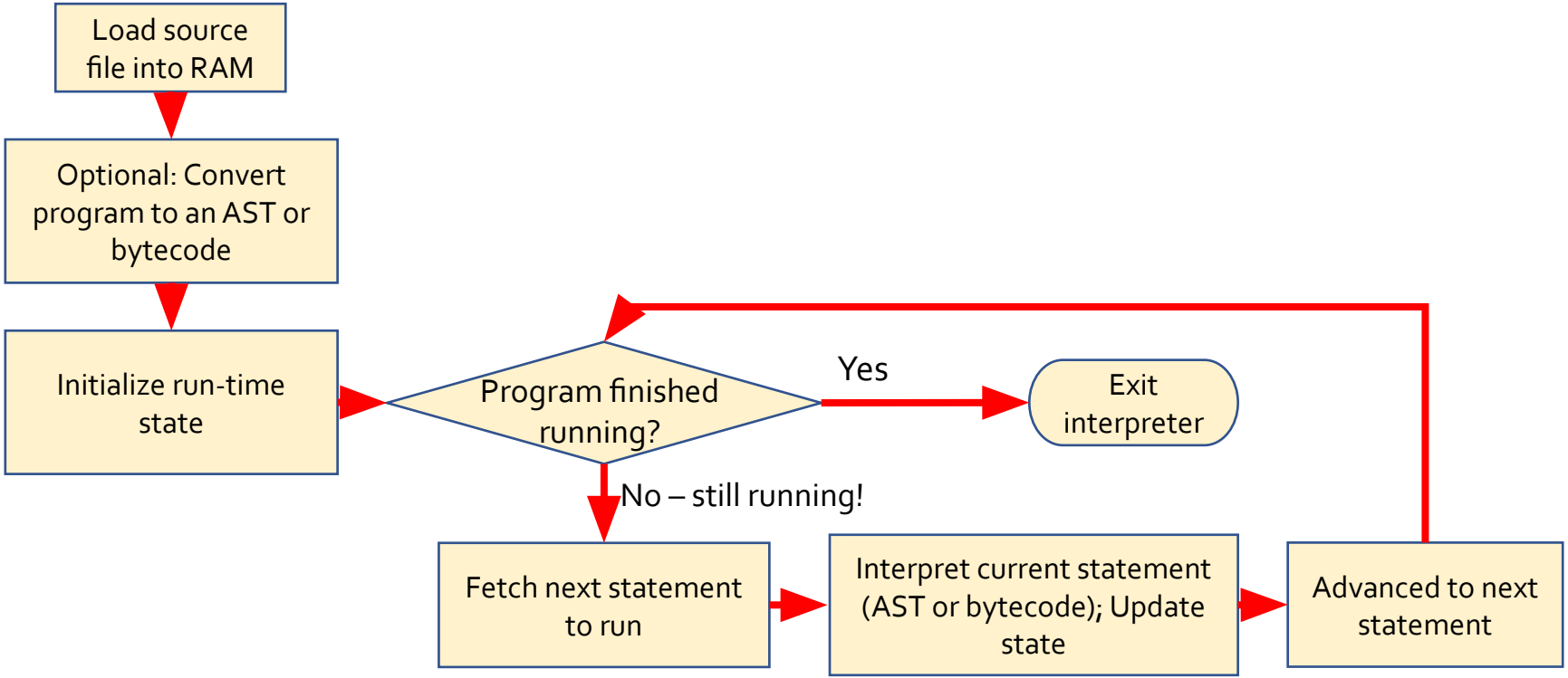
slide diagrams & examples! (lecture 1)


Compiler workflow
Source File
Lexical Analyzer
Lexical Units
Parser
Abstract Syntax Tree
Semantic Analyzer
Machine code or bytecode
Code Generator for ARM
Intermediate representation
Intermediate Representation Generator
Annotated Parse Tree


foo.cpp
Compiler
foo.o
C++ std library
GUI library
bar.cpp
Compiler
bar.o
Linker
final_exec
bletch.cpp
Compiler
bletch.o

```
while(i<5)
 x = x + i*3;
```




Load source file into RAM
Optional: Convert program to an AST or bytecode
Initialize run-time state
Program finished running?
Yes
Exit interpreter
No – still running!
Fetch next statement to run
Interpret current statement (AST or bytecode); Update state
Advanced to next statement

```
class Interpreter {
public:
  Interpreter(const vector<string>& program)
    { program_ = program; }

  void run() {
    cur_line_ = 0;
    terminated_ = false;
    while (!terminated_)
      execute_statement();
  }

private:
  void execute_statement();

  vector<string>     program_;
  int                cur_line_;
  bool               terminated_;
  map<string, int>   variables_;
};
```

```
void Interpreter::execute_statement() {
  const string& stat = program_[cur_line_];
  vector<string> tokens = split(stat,' ');
  if (tokens[0] == "end") {
    terminated_ = true;
    return;
  }
  if (tokens[0] == "set")
    variables_[tokens[1]] = stoi(tokens[2]);
  else if (tokens[0] == "add")
    variables_[tokens[1]] += stoi(tokens[2]);
  else if (tokens[0] == "print")
    cout << variables_[tokens[1]] << endl;
  ++cur_line_;
}
```

```
class Student:
 ...  # details hidden

# student goes through life changes
def life_changes(s: Student):
 s.change_major("Econ")
 # hates life choices, so starts from
scratch
 s = Student("Carence", "Music")

# main program
student1 = Student("Carey", "CompSci")
life_changes(student1)
student1.print_my_details();
```

prints the following:

```
Carey's major is Econ.
```

How does this language pass parameters?

By value? By reference? By pointer?