

# CS440: Programming Languages and Translators

Lecture 26: What did we do, why is it important, and what's next?

Spring 2023

Stefan Muller

# Logistics/Reminders

- HW6 due tonight
- Course eval open through Sunday
  - Bonus points for everyone:  $2 * (\text{response rate})^2$
- Review session: Monday 11-12, SB 106 (and Zoom and recorded)
- Final: **Tuesday, May 2, 10:30am-12:30pm, SB 104**

# Content

- Simple answer: everything!
- All lectures, from the beginning of the semester until this Thursday
  - More emphasis on material since midterm
  - Only high-level questions about post-HW6 material
- Written questions from HW5, HW6 and the midterm are good examples of the types of questions I might ask

# Format

- 120 minutes, 100 points
- Approx. 50%:
  - A few short answer questions
  - Give the value of an OCaml expression or say it doesn't evaluate (like on midterm)
  - Write a proof tree for a big-step semantics or typing derivation (like HW5, 2.1 and HW6, 1.2)
  - Evaluate a lambda calculus term to a normal form (like HW5, 3.3, but you only have to do one)
- Approx. 50%: 2-3 more long questions

# Rules, etc.

- Write in whatever you want (please no red/green/purple pen though)
- You can bring **two** double-sided 8.5x11" sheets of notes
  - Written or typed, can contain anything you want
  - One can be the one from the midterm
- Provided reference material (I will give this to you at the exam, no need to print it or put it on your note sheets):
  - Signatures for OCaml list functions
  - IMP syntax and big-step rules
  - STLC syntax and typing rules

# Practice, review

- Practice exam posted on Blackboard today or tomorrow, with reference material
  - Same basic format as real exam, but I make no promises about exact difficulty, length
- Review session
  - Monday, 5/1 11am-12pm (instead of office hours)
  - SB 106
  - Will also be streamed and recorded – I'll send out the link
  - Come with questions!

# Schedule

- Intro (1 week)
- Learn OCaml (~4 weeks)
- Interpreters (~2 weeks)
- *Midterm*
- Type checking (~2 weeks)
- *Spring break*
- Formal semantics (~2 weeks)
- Formal type systems (~2 weeks)
- Other topics and wrap-up (~3 weeks)

} Programming Languages

} Implementing PLs

} Reasoning about PLs

# Knowing the right paradigm to use can make programming easier

Task: Sort a linked list (using merge sort)

```
/* Sort a linked list using merge sort */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Node structure */
struct Node {
    int data;
    struct Node *next;
};

/* Function to create a new node */
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

/* Function to insert a new node at the end of the linked list */
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

/* Function to print the linked list */
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

/* Function to sort a linked list using merge sort */
struct Node* mergeSort(struct Node* head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }
    struct Node* slow = head;
    struct Node* fast = head->next;
    while (fast != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    struct Node* mid = slow;
    struct Node* left = head;
    struct Node* right = mid->next;
    mid->next = NULL;
    left = mergeSort(left);
    right = mergeSort(right);
    return mergeTwoSortedLists(left, right);
}

/* Function to merge two sorted linked lists */
struct Node* mergeTwoSortedLists(struct Node* list1, struct Node* list2) {
    struct Node* mergedList = NULL;
    struct Node* temp = NULL;
    while (list1 != NULL || list2 != NULL) {
        if (list1 == NULL) {
            temp = list2;
            list2 = list2->next;
        } else if (list2 == NULL) {
            temp = list1;
            list1 = list1->next;
        } else if (list1->data < list2->data) {
            temp = list1;
            list1 = list1->next;
        } else {
            temp = list2;
            list2 = list2->next;
        }
        mergedList = insertAtEnd(&mergedList, temp->data);
        temp = temp->next;
    }
    return mergedList;
}

int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 1);
    insertAtEnd(&head, 3);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 4);
    insertAtEnd(&head, 5);
    printList(head);
    head = mergeSort(head);
    printList(head);
    return 0;
}
```

C

```
#!/usr/bin/perl
use strict;
use warnings;

# Node structure
my $Node = {
    data => 0,
    next => undef,
};

# Function to create a new node
sub createNode {
    my ($data) = @_;
    my $node = $Node->{data => $data, next => undef};
    return $node;
}

# Function to insert a new node at the end of the linked list
sub insertAtEnd {
    my ($head, $data) = @_;
    my $newNode = createNode($data);
    if ($head == undef) {
        $head = $newNode;
    } else {
        my $temp = $head;
        while ($temp->{next} != undef) {
            $temp = $temp->{next};
        }
        $temp->{next} = $newNode;
    }
}

# Function to print the linked list
sub printList {
    my ($head) = @_;
    my $temp = $head;
    while ($temp != undef) {
        print "$temp->{data} ";
        $temp = $temp->{next};
    }
    print "\n";
}

# Function to sort a linked list using merge sort
sub mergeSort {
    my ($head) = @_;
    if ($head == undef || $head->{next} == undef) {
        return $head;
    }
    my $slow = $head;
    my $fast = $head->{next};
    while ($fast != undef) {
        $slow = $slow->{next};
        $fast = $fast->{next->{next}};
    }
    my $mid = $slow;
    my $left = $head;
    my $right = $mid->{next};
    $mid->{next} = undef;
    $left = mergeSort($left);
    $right = mergeSort($right);
    return mergeTwoSortedLists($left, $right);
}

# Function to merge two sorted linked lists
sub mergeTwoSortedLists {
    my ($list1, $list2) = @_;
    my $mergedList = undef;
    my $temp = undef;
    while ($list1 != undef || $list2 != undef) {
        if ($list1 == undef) {
            $temp = $list2;
            $list2 = $list2->{next};
        } else if ($list2 == undef) {
            $temp = $list1;
            $list1 = $list1->{next};
        } else if ($list1->{data} < $list2->{data}) {
            $temp = $list1;
            $list1 = $list1->{next};
        } else {
            $temp = $list2;
            $list2 = $list2->{next};
        }
        $mergedList = insertAtEnd($mergedList, $temp->{data});
        $temp = $temp->{next};
    }
    return $mergedList;
}

int main() {
    my $head = undef;
    insertAtEnd($head, 1);
    insertAtEnd($head, 3);
    insertAtEnd($head, 2);
    insertAtEnd($head, 4);
    insertAtEnd($head, 5);
    printList($head);
    $head = mergeSort($head);
    printList($head);
    return 0;
}
```

Python

```
#!/usr/bin/ocaml
let Node = {
  data : int;
  next : t option;
}

let createNode data = {
  data = data;
  next = None;
}

let insertAtEnd head data =
  let newNode = createNode data in
  if head = None then
    Some newNode
  else
    let temp = head in
    while temp.next = Some _ do
      temp = temp.next
    done;
    temp.next = Some newNode

let printList head =
  let rec aux head =
    match head with
    | None -> ()
    | Some node -> (print_endline (string_of_int node.data);
                    aux node.next)
  in
  aux head

let rec mergeSort head =
  if head = None || head.next = None then
    head
  else
    let slow = head in
    let fast = head.next in
    while fast != None do
      slow = slow.next;
      fast = fast.next.next
    done;
    let mid = slow in
    let left = head in
    let right = mid.next in
    mid.next = None;
    let left = mergeSort left in
    let right = mergeSort right in
    mergeTwoSortedLists left right

let rec mergeTwoSortedLists list1 list2 =
  if list1 = None || list2 = None then
    list1
  else
    if list1.data < list2.data then
      list1
    else
      list2
  end;
  let temp = mergeTwoSortedLists list1 list2 in
  mergeTwoSortedLists temp temp.next

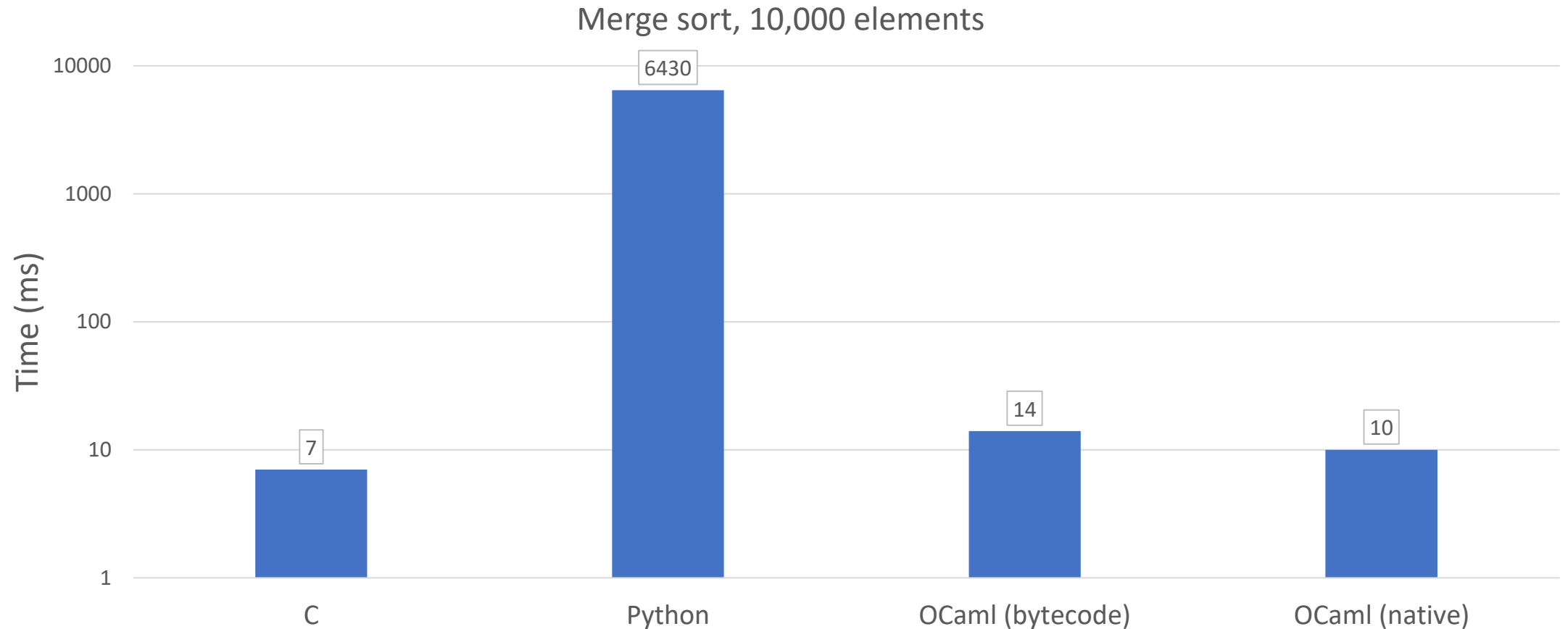
int main () =
  let head = None in
  insertAtEnd head 1;
  insertAtEnd head 3;
  insertAtEnd head 2;
  insertAtEnd head 4;
  insertAtEnd head 5;
  printList head;
  head = mergeSort head;
  printList head;
  0
```

OCaml

Try writing even a minimal working web server in C in an hour!



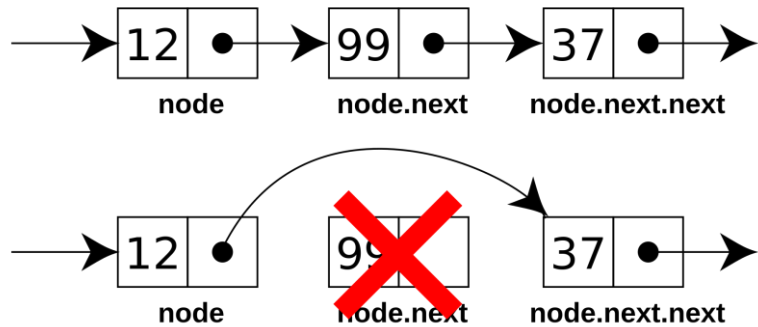
# Knowing about the language and how it's translated can help you write faster code



# Type systems can express different levels of guarantees

- **C** `node *mergesort(node *list)`
  - Takes a pointer to a node and returns a pointer to a node.
- **OCaml** `mergesort : int list -> int list`
  - Takes an integer list and returns an integer list.
- **Haskell** `mergesort :: IO ([int] -> [int])`
  - Takes an integer list, returns an integer list and performs I/O (e.g., printing).
- **Coq** `mergesort : forall (l1 : list int), exists (l2: int list),  
Sorted l2 /\ Permutation l1 l2`
  - Takes an integer list and returns a sorted permutation of it.

# Different languages are up to different tasks



 OCaml ?

C?

 Rust?

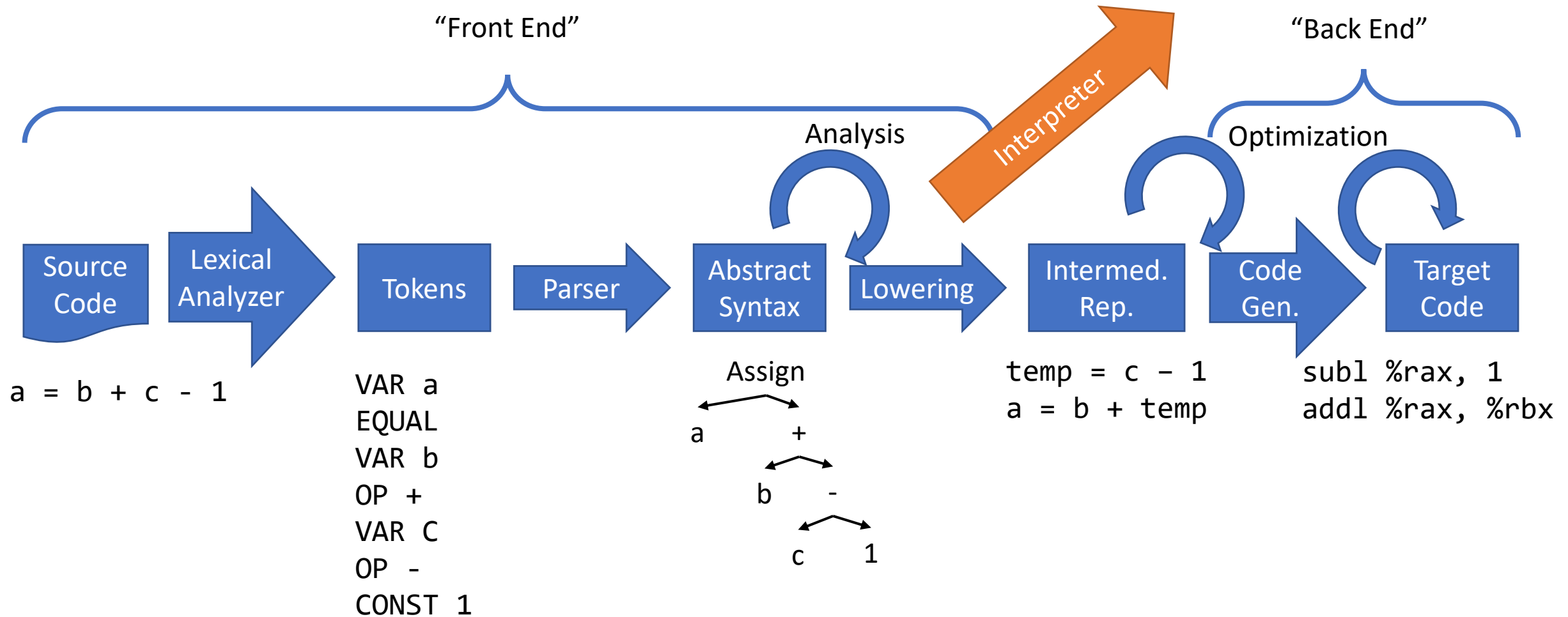
# Schedule

- Intro (1 week) } Programming Languages
- Learn OCaml (~4 weeks)
- Interpreters (~2 weeks)
- *Midterm*
- Type checking (~2 weeks) } Implementing PLs
- *Spring break*
- Formal semantics (~2 weeks)
- Formal type systems (~2 weeks)
- Other topics and wrap-up (~3 weeks) } Reasoning about PLs

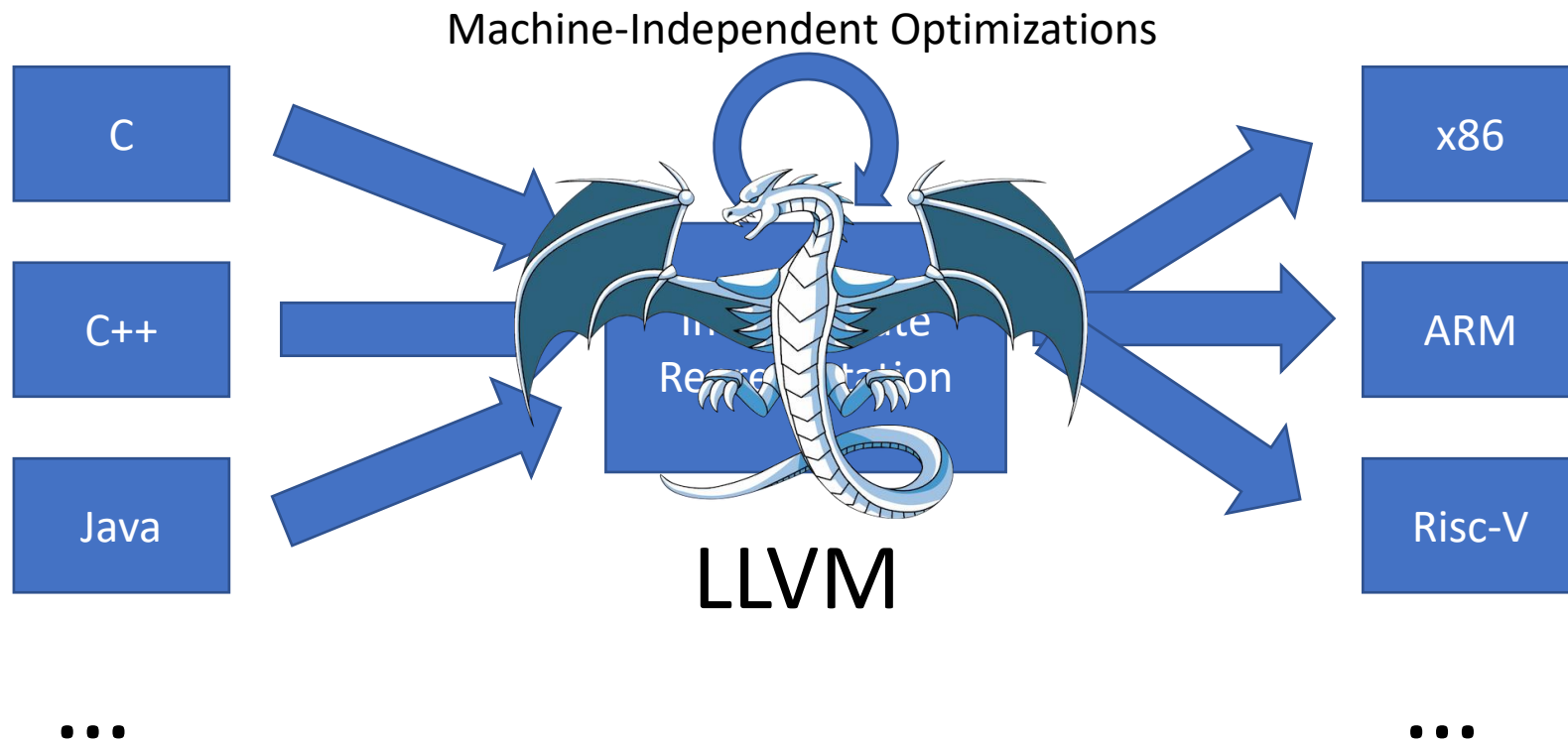
# Compilers vs. interpreters

- Compiler
  - Translates the program to a form executable by the machine (or assembly)
  - Compile, then can run the executable: compiler no longer involved
- Interpreter
  - Doesn't translate to machine-readable format
    - Might compile to bytecode or intermediate representation
  - Runs (“interprets”) program directly
  - Can't run without the interpreter

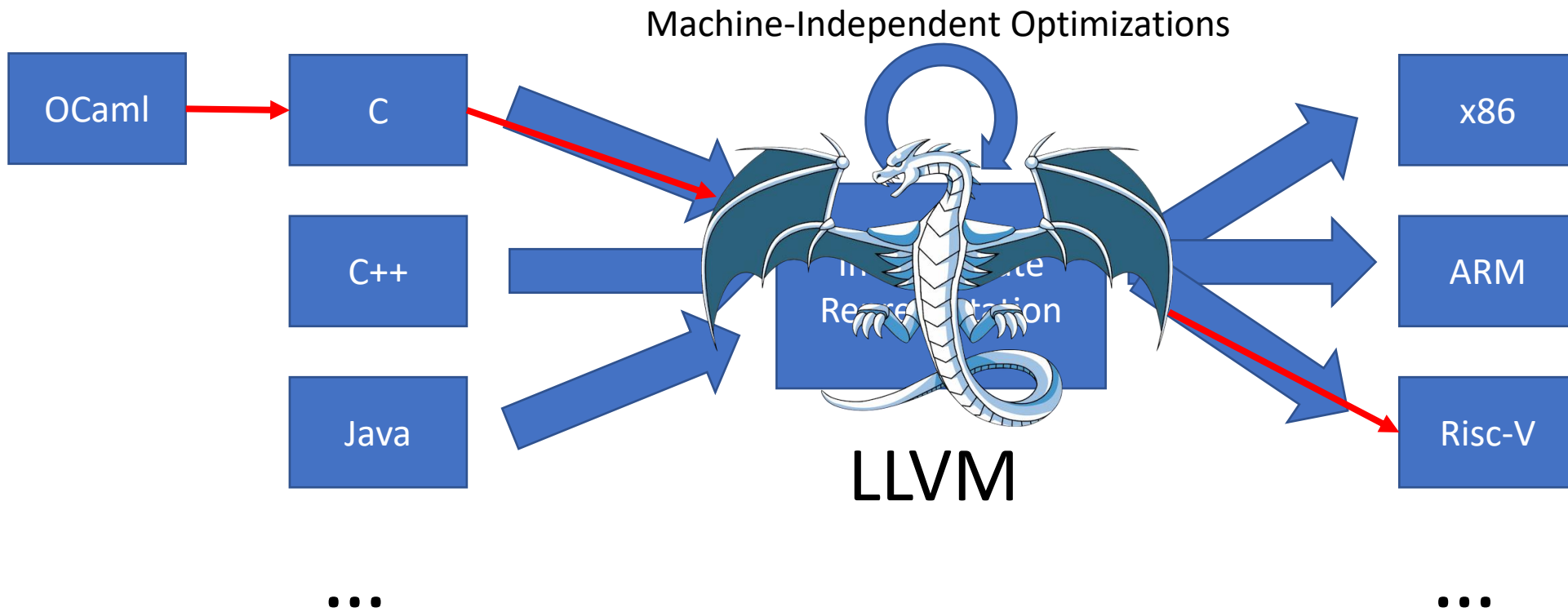
# Compilers translate code in phases



# Compiler collections also swap out front ends for different languages



# Want to see more? Take CS443 (Compiler Construction)





# Schedule

- Intro (1 week) } Programming Languages
- Learn OCaml (~4 weeks)
- Interpreters (~2 weeks)
- *Midterm*
- Type checking (~2 weeks) } Implementing PLs
- *Spring break*
- Formal semantics (~2 weeks)
- Formal type systems (~2 weeks) } Reasoning about PLs
- Other topics and wrap-up (~3 weeks)

Type safety: well-typed programs don't "go wrong"

- Progress: A well-typed program isn't wrong (in STLC: stuck)
- Preservation: If a well-typed program takes a step, it's still well-typed

“Go wrong” can mean lots of other things

- One application we haven't talked (much) about: parallelism

# Functional languages are **great** for parallelism

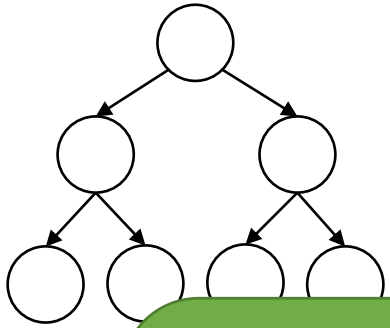
```
let (a, b) = (f (), g ())
```

- If f and g are functional, it can't matter what order we execute them in...
- so why not do them at the same time?



Cilk, Go, Parallel ML, Parallel Haskell, ...

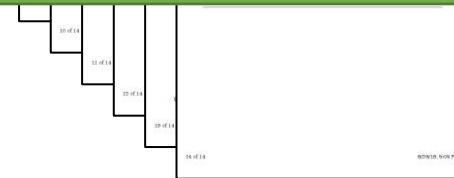
$$\text{Theorem: } T(P) \leq \frac{W}{P} + S$$



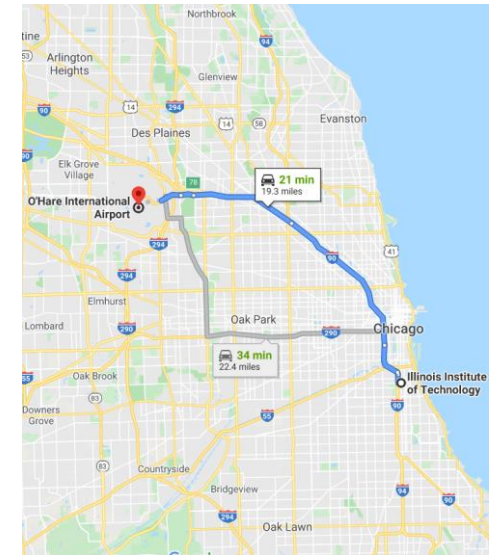
So why don't people  
use the abstractions?

(Short answer: user interaction)

POSIX threads



# Use multiple threads to do a lot of things



# With parallelism, stops being responsive



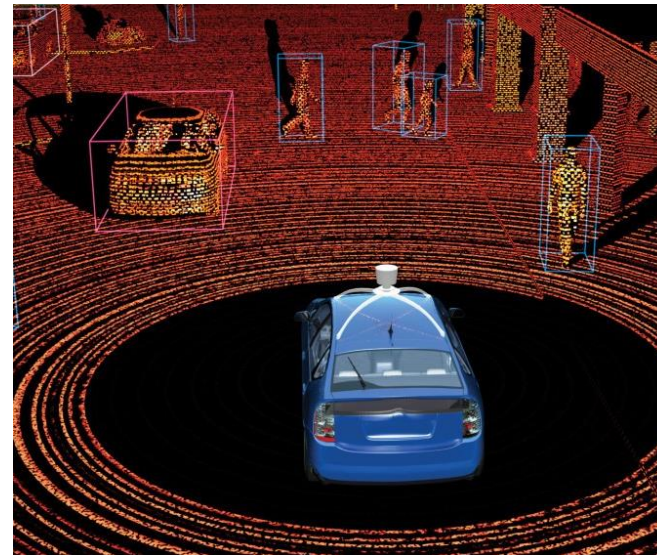
One thread listening  
for events

Many lightweight threads  
running AI

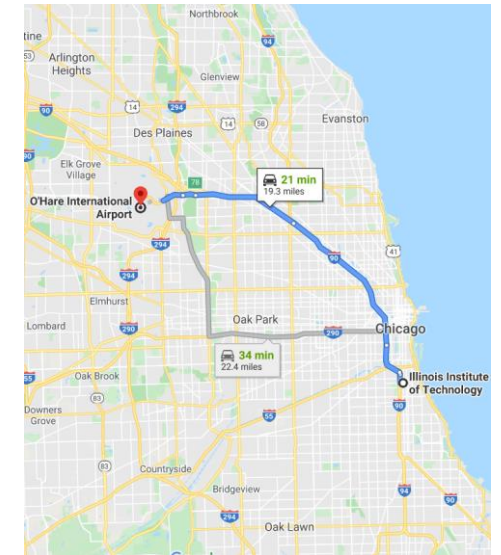
# Some tasks have higher priority than others



>



>

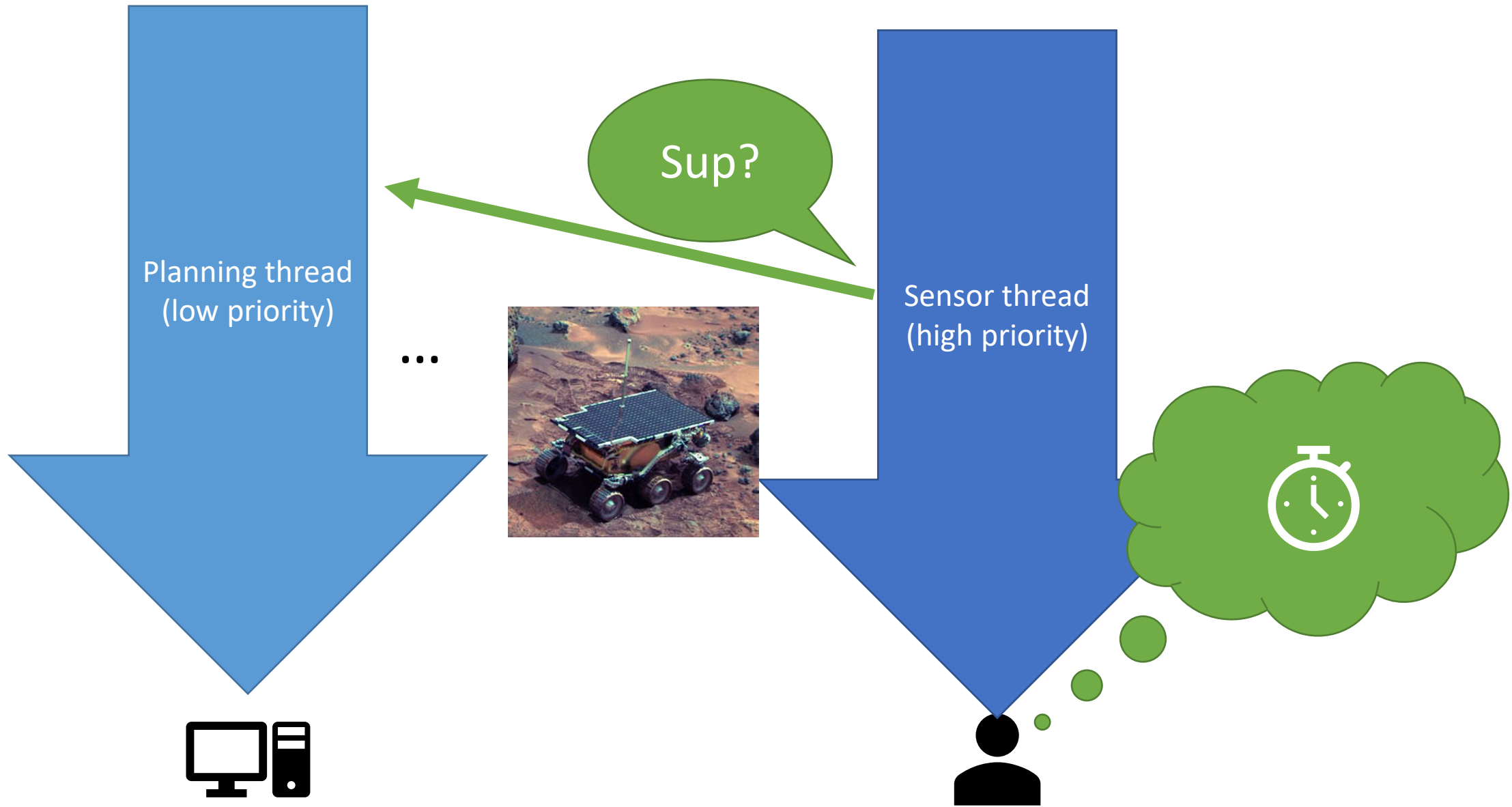




# Simple priority syntax

```
priority sensors  
priority short_term_planning  
priority long_term_planning  
order long_term_planning < short_term_planning  
order short_term_planning < sensors
```

```
sensethread <- spawn[sensors] { ... };  
plan1 <- spawn[short_term_planning] { ... };  
plan2 <- spawn[long_term_planning] { ... };
```



# The program went wrong

- How do we stop programs from going wrong?

# We track priorities through code in **types**

order low < high

```
cmd[high]
```

```
{
```

```
  t <- spawn[high] { ... };
```

```
  ...
```

```
  sync(t)
```

```
}
```

- This thread is high-priority
- Spawn a high-priority thread
- Sync on it

```
constraint violated at example.prm:5.1-5.8 : high <= low  
Type error: constraint violated
```

W

$e$  is a handle to a thread of priority  $\rho$

$$\frac{\Gamma \vdash e : \tau \text{ thread}[\rho] \quad \rho \geq \rho'}{\Gamma \vdash \text{sync}(e) : \tau @ \rho'}$$

$e$  is higher priority than the current thread

# What if a thread wants to change its priority?

```
priority sensors
priority short_term_planning
priority long_term_planning
order long_term_planning < short_term_planning
order short_term_planning < sensors
```

```
plan1 <- spawn[long_term_planning]
  { ...
    if time > deadline - 5ms then
      change[short_term_planning];
    ... }
};
```

Extension to type system being  
done currently by a CS440  
Spring 2021 student!

Want to learn

- How to prove progress and preservation?
- More advanced type systems that can express more complex programs?
- How to design new type systems for things you want to express about programs?

Take CS534 (Types and Programming Languages)

# Hoare Logic can verify other properties

- Remember:  $\models \{P\} S \{Q\}$
- “if P holds before and S terminates, Q holds after”
- $\models \{n \geq 0\} x := \text{fact}(n) \{x = n!\}$
- How do we prove this?



With inference rules!

$$SKIP \frac{}{\models \{P\} \text{ skip } \{P\}}$$

$$ASSIGN \frac{}{\models \{[E/x] P\} x := E \{P\}}$$

$$\frac{}{\models \{[(x + 1)/x](x = 1)\} x := x + 1 \{x = 1\}}$$

# With inference rules!

“Loop invariant”

$$\text{SEQ} \frac{\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}}{\models \{P\} S_1; S_2 \{R\}} \quad \text{WHILE} \frac{\{P \wedge B\} S \{P\}}{\models \{P\} \text{while } B \text{ do } S \{P \wedge \neg B\}}$$

$$\frac{\dots \quad \overline{\{x = (i-1)! \wedge i \leq n\} (x := x * i; i := i + 1) \{x = (i-1)!\}}}
 {\overline{\{n \geq 0\} x := 1; i := 2 \{x = (i-1)!\} \quad \{x = (i-1)!\} \text{while } i < n \text{ do } (x := x * i; i := i + 1) \{x = (i-1)!\} \wedge i > n}}
 \models \{n \geq 0\} x := 1; i := 2; \text{while } i \leq n \text{ do } (x := x * i; i := i + 1) \{x = n!\}$$

```
9 method ComputeFib(n: nat) returns (r: nat)
10     ensures r == fib(n)
11 {
12     var a, b := 0, 1;
13     var temp := 0;
14     var i := 0;
15     while (i < n)
16     invariant (a == fib(i)) && (b == fib(i+1)) && (i <= n)
17     {
18         temp := a + b;
19         assert temp == fib(i+2);
20         a := b;
21         b := temp;
22         i := i + 1;
23     }
24     return a;
25 }
```

WSL: Ubuntu udaya 1 0 Spaces: 3 UTF-8 CRLF Dafny

Want to learn

- How to use Hoare Logic to prove real things about real programs?
- About total correctness (proving programs terminate)?
- About verifying concurrent programs?

Take CS536 (Science of Programming)

# What to take next?

Like this stuff (especially the priority type system) and want to do it more hands-on?  
I'm looking for research assistants!  
Email me!

