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

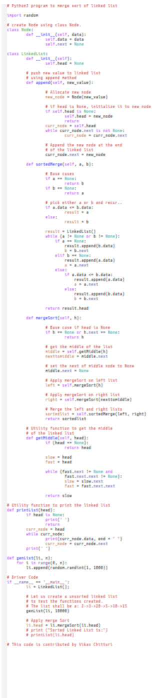
Implementing PLs

Reasoning about PLs

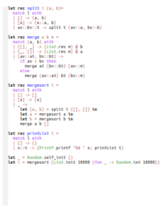
Knowing the right paradigm to use can make programming easier

Task: Sort a linked list (using merge sort)

C



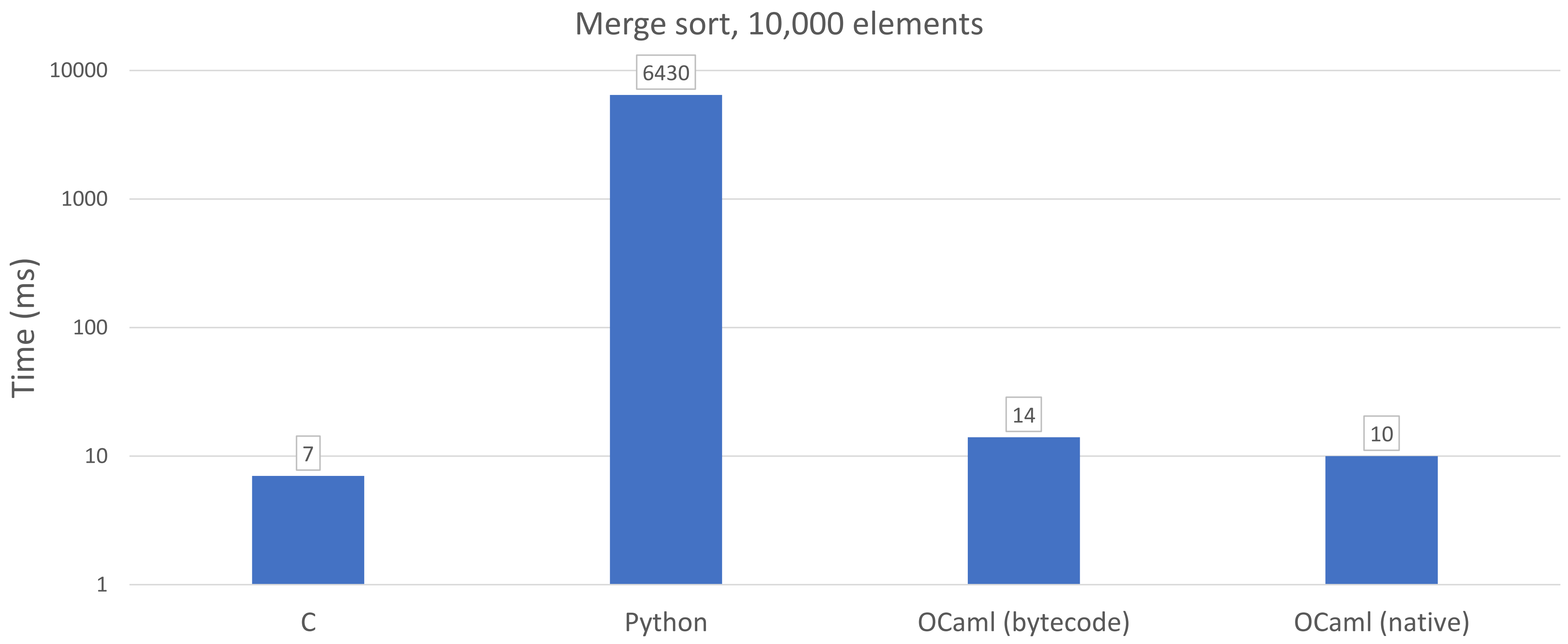
Python



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



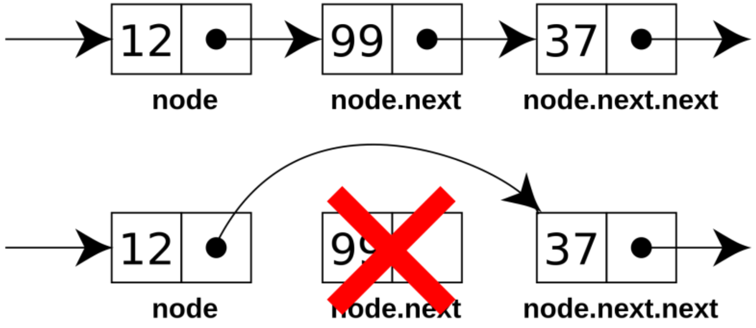
Type systems can express different levels of guarantees

- 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

OCaml ?

C?

Rust?

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

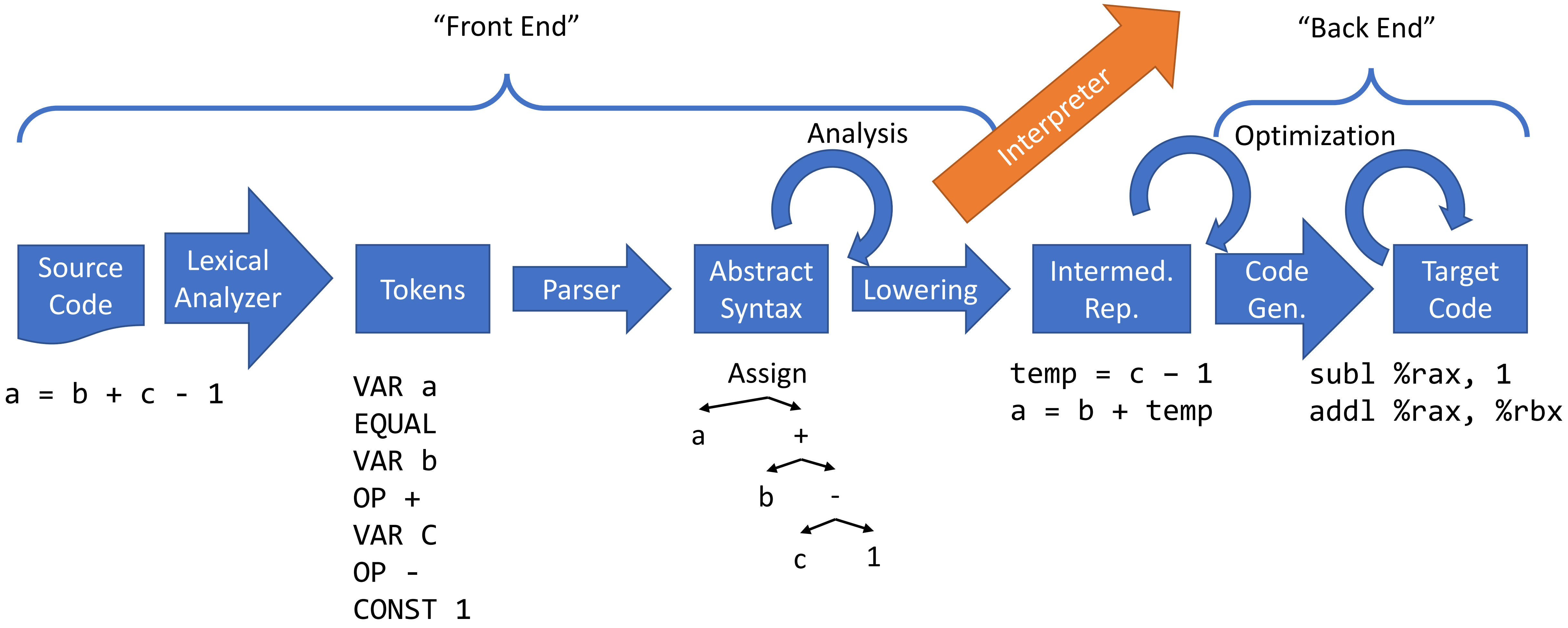
Implementing PLs

Reasoning about PLs

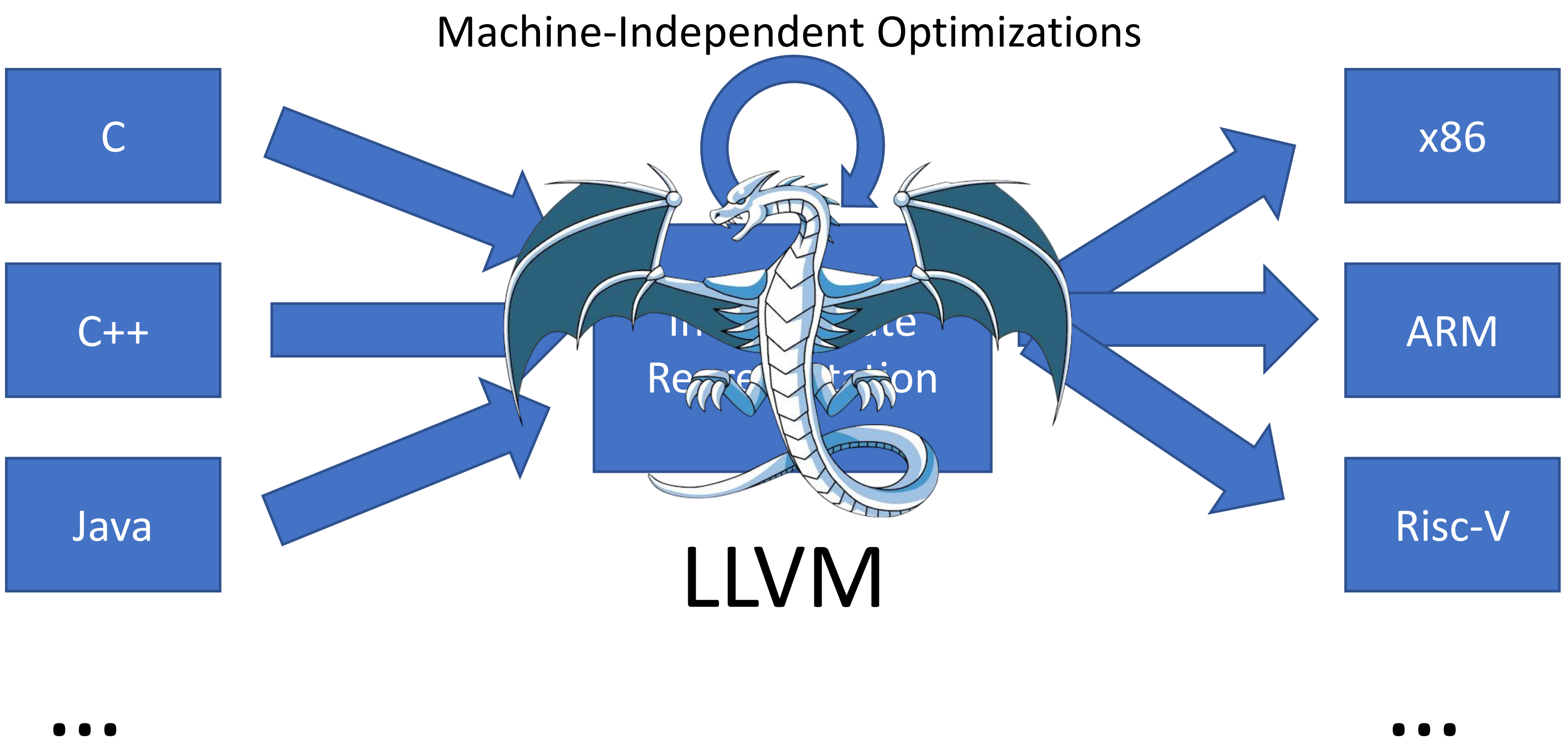
Compilers vs. interpreters

- 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

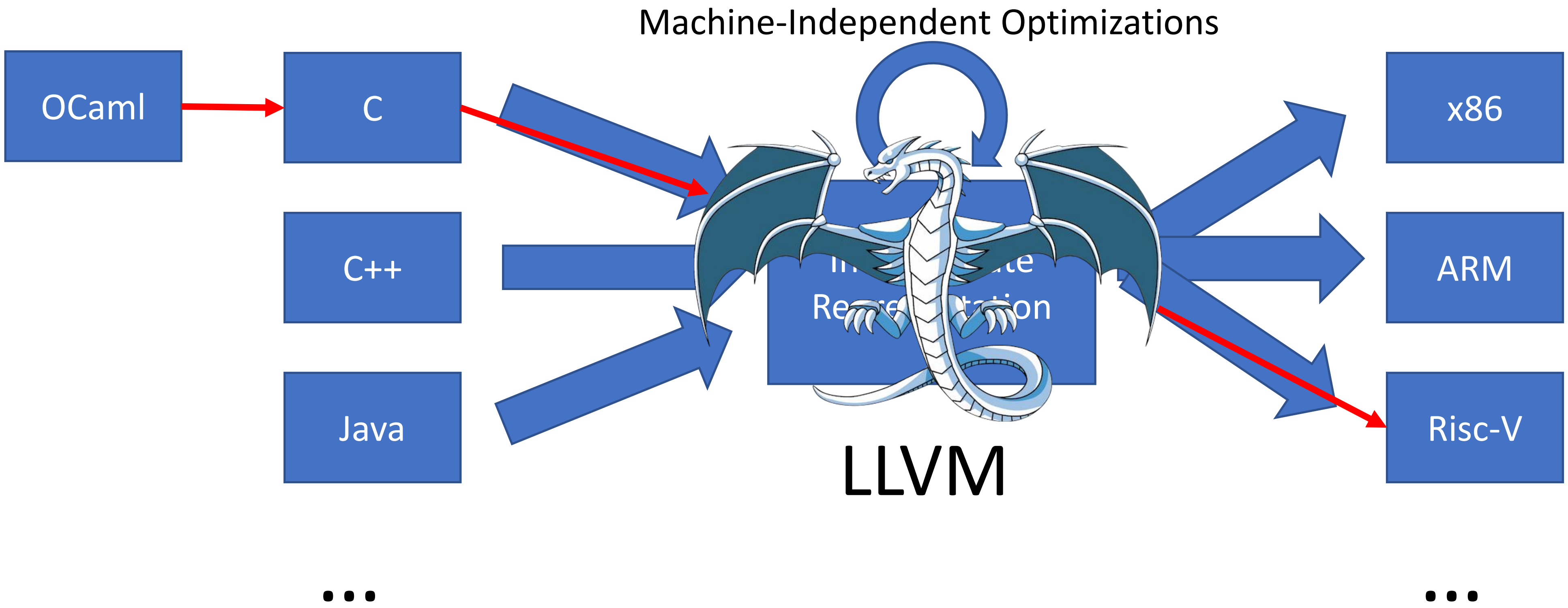
Compilers translate code in phases



Compiler collections also swap out front ends for different languages



Want to see more?
Take CS443 (Compiler Construction)



Schedule

- Intro (1 week)
- 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

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

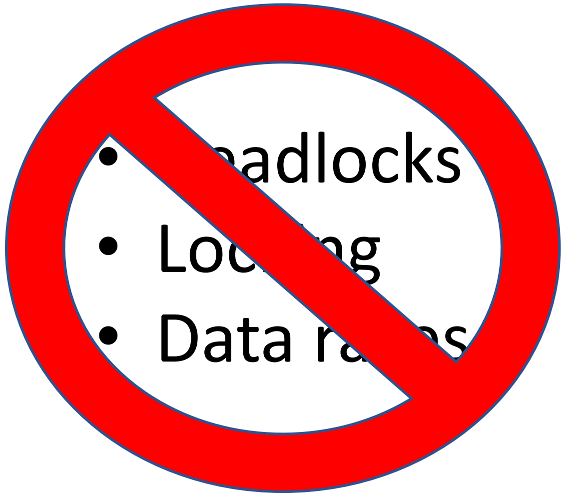
“Go wrong” can mean lots of other things

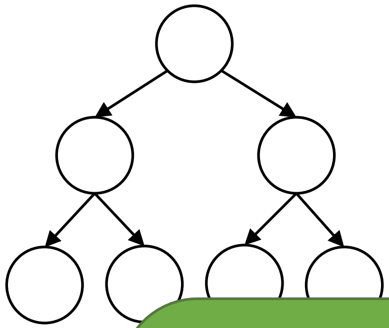
- One application we haven’t talked (much) about: parallelism

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?



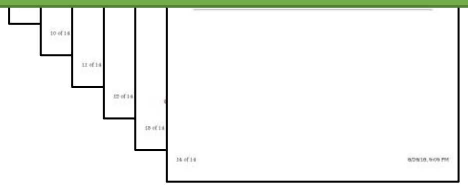


Cilk, Go, Parallel ML, Parallel Haskell, …

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

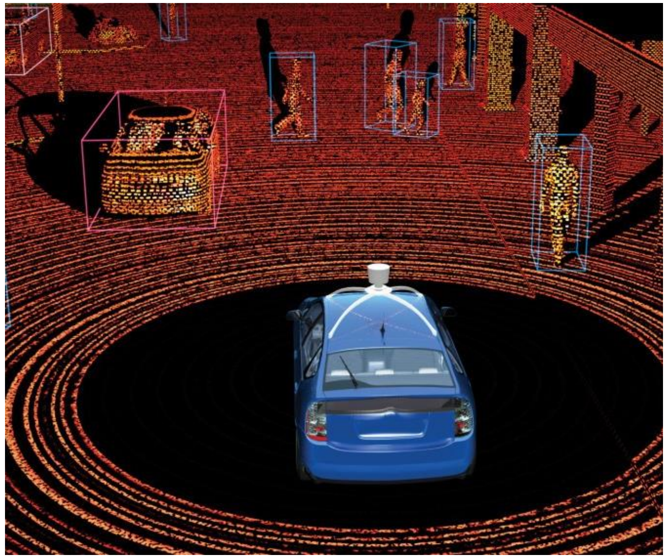
So why don't people use the abstractions?

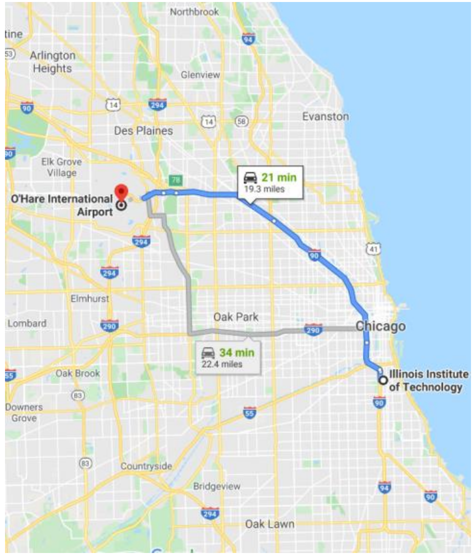
(Short answer: user interaction)



POSIX threads

Use multiple threads to do a lot of things





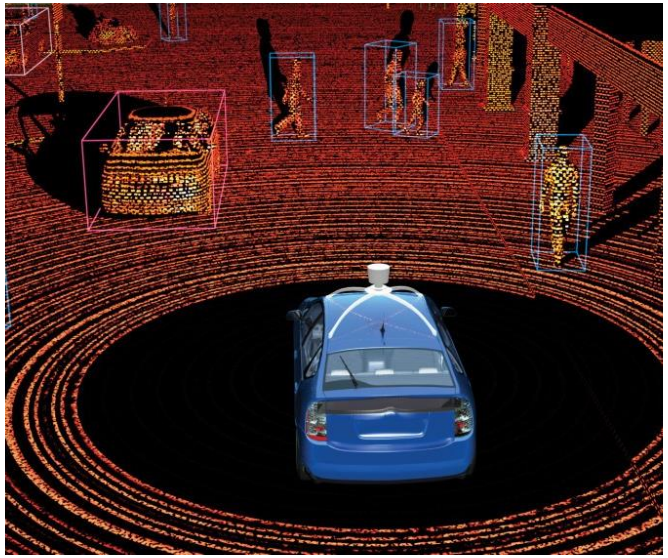
With parallelism, stops being responsive

One thread listening for events

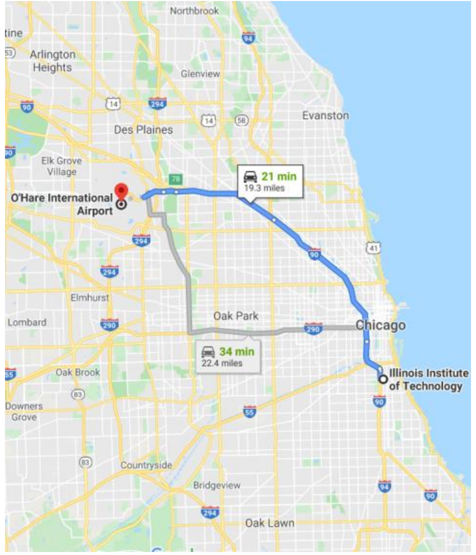
Many lightweight threads running AI

Some tasks have higher priority than others

>



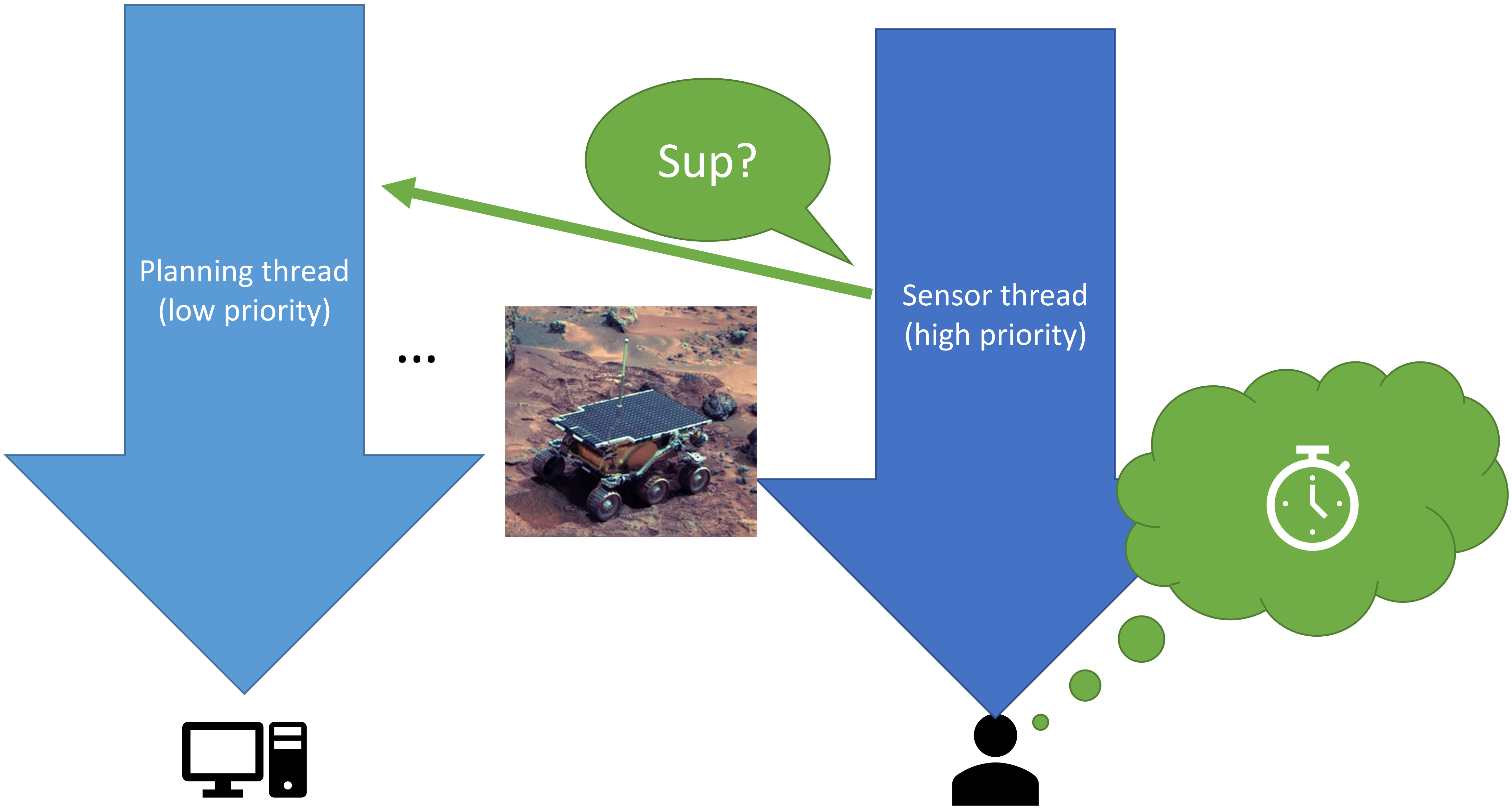
>



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] { … };
```



Planning thread
(low priority)
...
Sup?
Sensor thread
(high priority)

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 ***ρ***

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

W

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: $\vDash \{P\}\, S\, \{Q\}$
- "if P holds before and S terminates, Q holds after"

- $\vDash \{n \geq 0\}\, x := \mathrm{fact}(n)\, \{x = n!\}$

- How do we prove this?

With inference rules!

$$SKIP\ \frac{}{\vDash\ \{P\}\ skip\ \{P\}} \qquad ASSIGN\ \frac{}{\vDash\ \{[E/x]\ P\}\ x\ := E\ \{P\}}$$

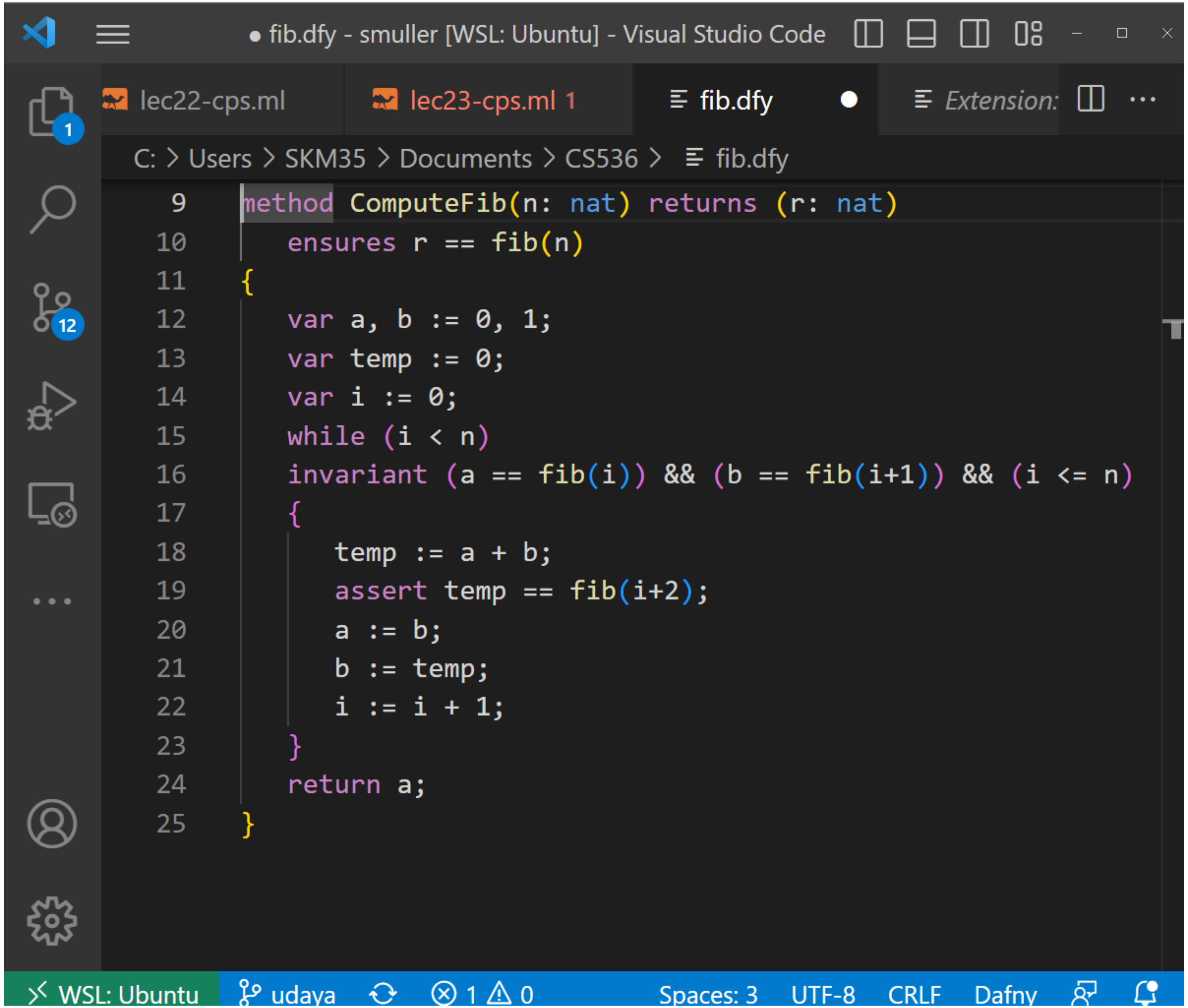
$$\frac{}{\vDash\ \{[(x+1)/x](x=1)\}\ x\ := x+1\ \{x=1\}}$$

With inference rules!

"Loop invariant"

$$SEQ\ \frac{\{P\}\,S_1\,\{Q\}\qquad \{Q\}\,S_2\,\{R\}}{\vDash\ \{P\}\,S_1;S_2\{R\}}\qquad WHILE\ \frac{\{P\wedge B\}\,S\,\{P\}}{\vDash\ \{P\}\ while\ B\ do\ S\ \{P\wedge\neg B\}}$$

$$\frac{\dfrac{\dots}{\{n\geq 0\}\ x\ \coloneqq 1;i\ \coloneqq 2\ \{x=(i-1)!\}}\qquad \dfrac{\dfrac{\dots}{\{x=(i-1)!\wedge i\leq n\}(x\ \coloneqq x\ *i;i\ \coloneqq i+1)\ \{\ x=(i-1)!\ \}}}{\{x=(i-1)!\}\ while\ i<n\ do\ (x\ \coloneqq x\ *i;i\ \coloneqq i+1)\ \{x=(i-1)!\wedge\ i>n\}}}{\vDash\{n\geq 0\}\ x\coloneqq 1;i\coloneqq 2;while\ i\leq n\ do\ (x\ \coloneqq x\ *i;i\ \coloneqq i+1)\ \{x=n!\}}$$



fib.dfy - smuller [WSL: Ubuntu] - Visual Studio Code
lec22-cps.ml
lec23-cps.ml 1
fib.dfy
Extension:
C: > Users > SKM35 > Documents > CS536 > fib.dfy
method ComputeFib(n: nat) returns (r: nat)
ensures r == fib(n)
{
var a, b := 0, 1;
var temp := 0;
var i := 0;
while (i < n)
invariant (a == fib(i)) && (b == fib(i+1)) && (i <= n)
{
temp := a + b;
assert temp == fib(i+2);
a := b;
b := temp;
i := i + 1;
}
return a;
}
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!