CS4100: Formal Languages and Compilers

Gordon Stewart
Assistant Professor, School of EECS

Office: Stocker 355
gstewart@ohio.edu

Administrative Stuff

http://ace.cs.ohio.edu/~gstewart/courses/4100-17

Logistics: Lecture T/Th 1:30-2:50pm
ARC 315

Professor: Gordon Stewart
355 Stocker
gstewart@ohio.edu
OH: Tuesdays, Thursdays 11am-12pm

TA: Alex Bagnall (ab667712@ohio.edu)

What's a Formal Language?

- Informally:
  - A set of strings of symbols with precise meaning, designed to be understood, or interpreted, by some sort of computing device (a Turing machine, an Intel X86 chip, a compiler, a bytecode interpreter, etc.)

- Examples:
  - Commonly used programming languages: C, C++, Perl, Haskell, Ocaml, Fortran, Java, Rust, Python, Lisp, Prolog, Ada, etc.
  - Turing machine descriptions (to be interpreted by a universal Turing machine)
  - x86 assembly, MIPS assembly, etc.

What's a Compiler?

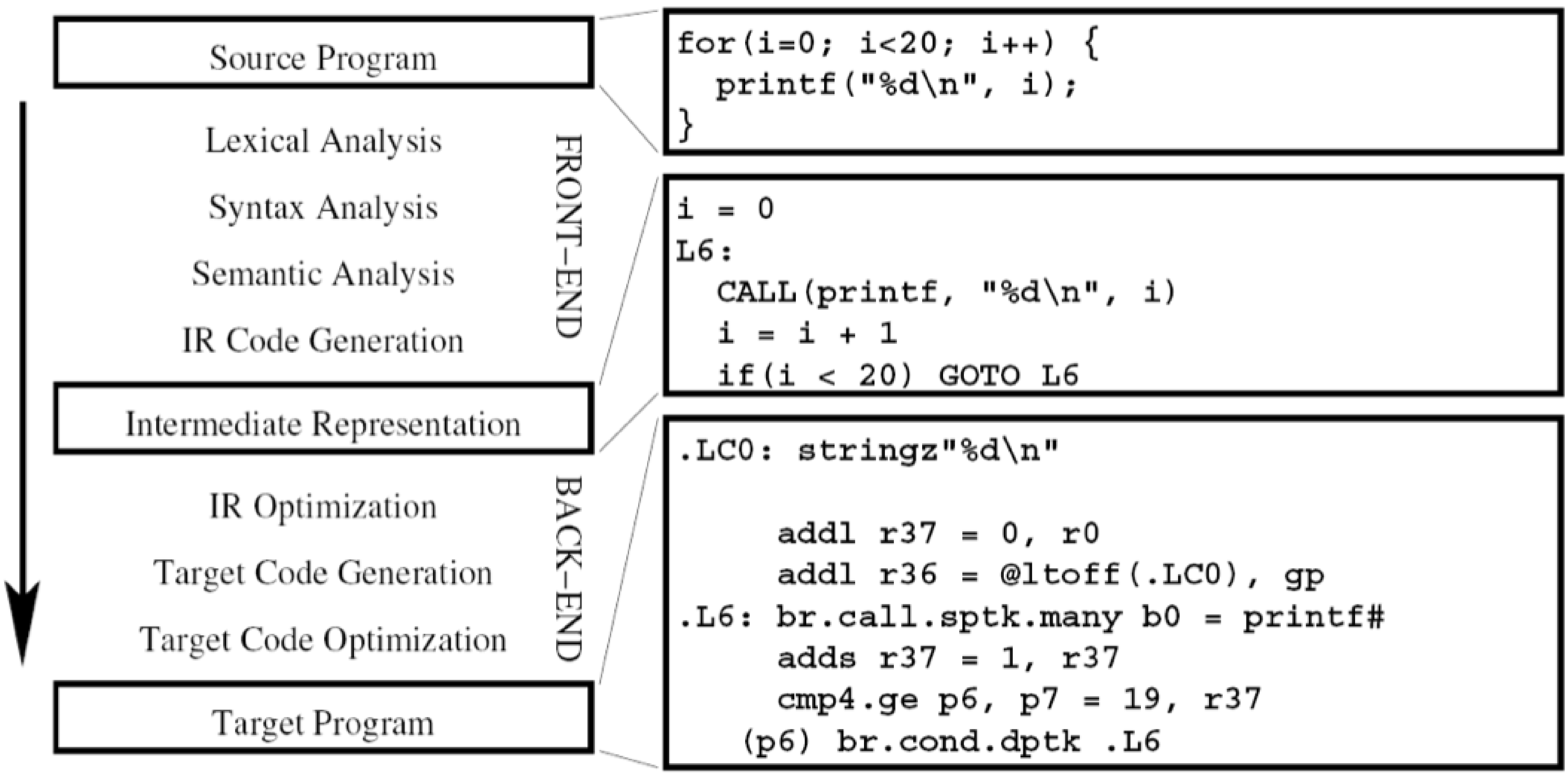
- A piece of software that translates programs in a **source (formal) language** to functionally equivalent programs in a **target (formal) language**

- **Source languages:** C, C++, Python, Java, JavaScript, Haskell, OCaml, VHDL, Verilog, …

- **Target languages:** x86, ARM, PPC, LLVM, C, …

- Compilers also:
  - Optimize your programs
  - Find errors early (syntax errors, type errors, etc.) before they manifest at runtime
  - Save you time and make you a better programmer!

What’s a Compiler?



Why Learn About Compilers?

Compilers are everywhere!

The usual places:

- C, C++ -> Assembly (gcc, icc, clang, …)
- Java, Python -> Bytecode (javac, python)
- Rust, Haskell -> LLVM (rustc, ghc, …)

But also

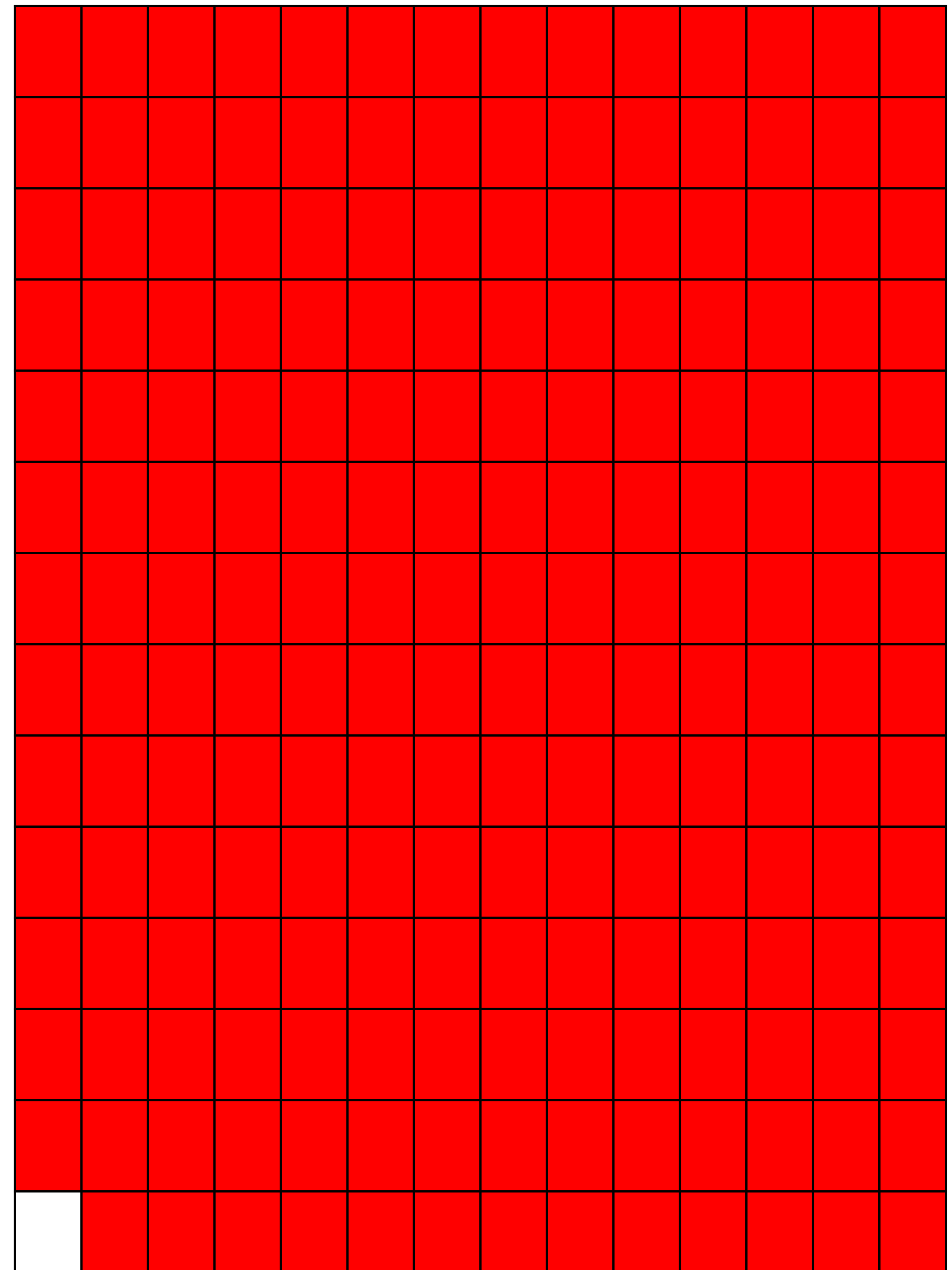
- Assembly -> machine code (assembler/linker)
- Machine instructions -> microcode (e.g., x86)
- SystemVerilog (HDL) -> FPGA layout
- Bluespec (higher-level HDL) -> SystemVerilog
- Publishing: LaTeX -> PDF

Why Learn About Compilers?

Almost all code is compiled

Linux
- C = 2,558,100 lines
- Assembly = 12,164 lines
- Almost 99.5% C!

1 / (14*14) * 100 = ~0.5%



Why Learn About Compilers?

Compiler can teach us about:

Programming Languages

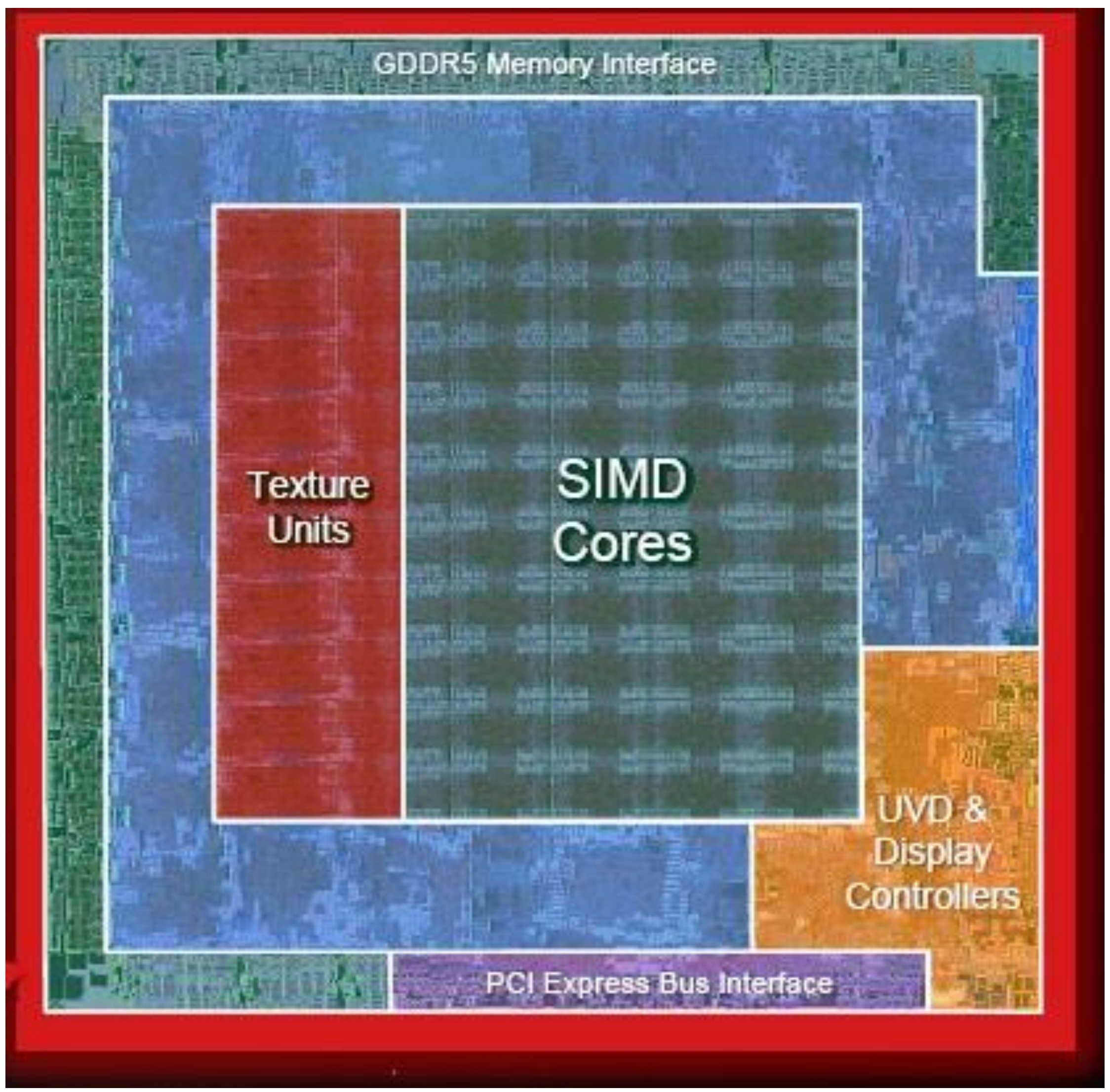
- Why are some PL features easy to implement on current hardware? Why are some difficult?
- How do we develop performant implementations? (Time, energy, memory usage, …)
- Take advantage of parallelism, new hardware, GPUs?

Computer Architecture

- How do instruction set architectures (ISAs) affect compilation? What's the right level of abstraction? CISC vs. RISC (an ongoing debate…cf. RISC-V)?

Why Learn About Compilers?

- Programming Languages: Can we compile existing languages (e.g. C) to take advantage of massively parallel hardware like GPUs?



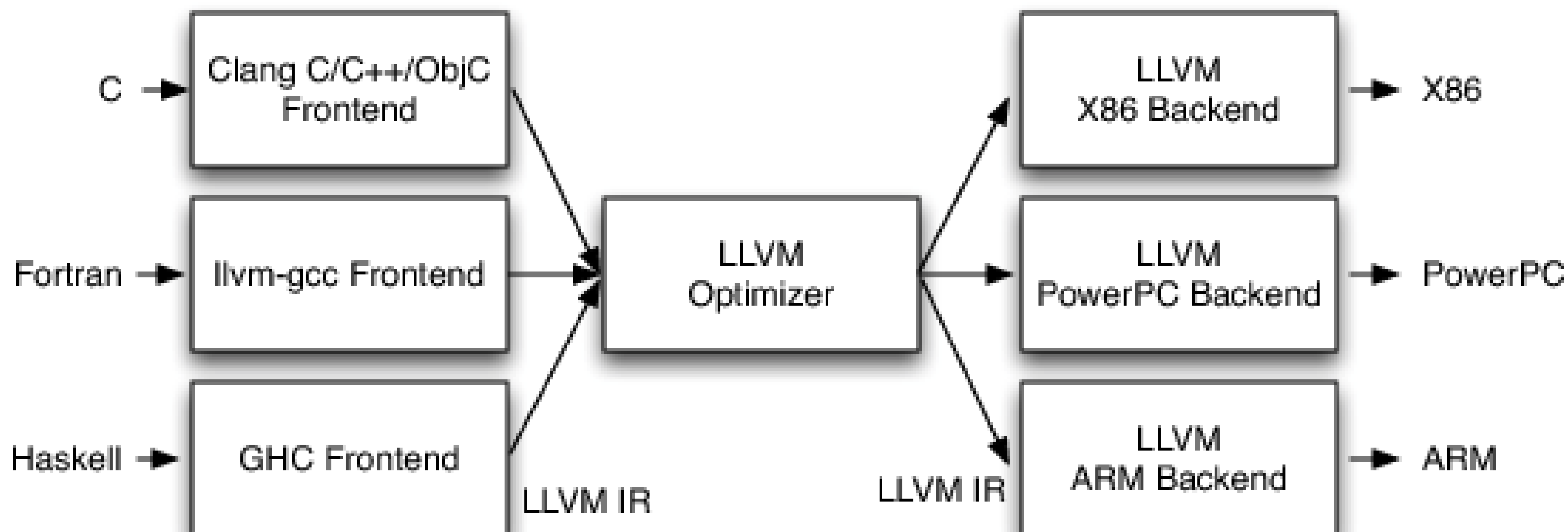
ATI Radeon (~80 stream processing units)

- Which are the right abstractions to expose to the programmer? CUDA? OpenCL? Something new?

- Is it possible to **automatically parallelize** sequential programs to parallel hardware?

Why Learn About Compilers?

- Compilers can also teach us about abstractions and architecture design: **Which abstractions can/should ISAs expose to the language implementer?**
- ISAs aren't always in hardware!



Why Learn About Compilers?

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.10.0"
declare i32 @putchar(i32)
define i32 @show_density(i32 %i) {
    %_t3 = icmp slt i32 %i, 2
    br i1 %_t3, label %true6, label %false8
  true6:
    %_t4 = call i32 @putchar(i32 42)
    br label %true_end7
  true_end7:
    br label %end10
  false8:
    %_t14 = icmp slt i32 %i, 4
    br i1 %_t14, label %true17, label %false19
  true17:
    %_t15 = call i32 @putchar(i32 43)
    br label %true_end18
  true_end18:
  …
}
```

Why Learn About Compilers?

```
…
define i32 @mandelbrot() {
    %x = alloca double
    store double 0.000000, double* %x
    %y = alloca double
    store double 0.000000, double* %y
    br label %branch130
  branch130:
    %_t133 = load double* %y
    %_t128 = fcmp olt double %_t133, 50.000000
    br i1 %_t128, label %body131, label %done132
  body131:
    store double 0.000000, double* %x
    …
}
define i32 @main() {
  %_t1 = call i32 @mandelbrot()
  %_t0 = call i32 @show_newline()
  ret i32 %_t0
}
```

*** test50-fractal.gpy passed ***

```
*** test50-fractal.gpy passed ***
```

Mandelbrot Sets

$$f_c(z) = z^2 + c$$

$$\text{Mandelbrot} = \{\ c \mid \ \ \forall n.\ |iter\ n\ f\,c\ 0| \leq 2\ \}$$

The set of complex c for which $f_c(0)$ does not "escape" to infinity

Mandelbrot in **grumpy**

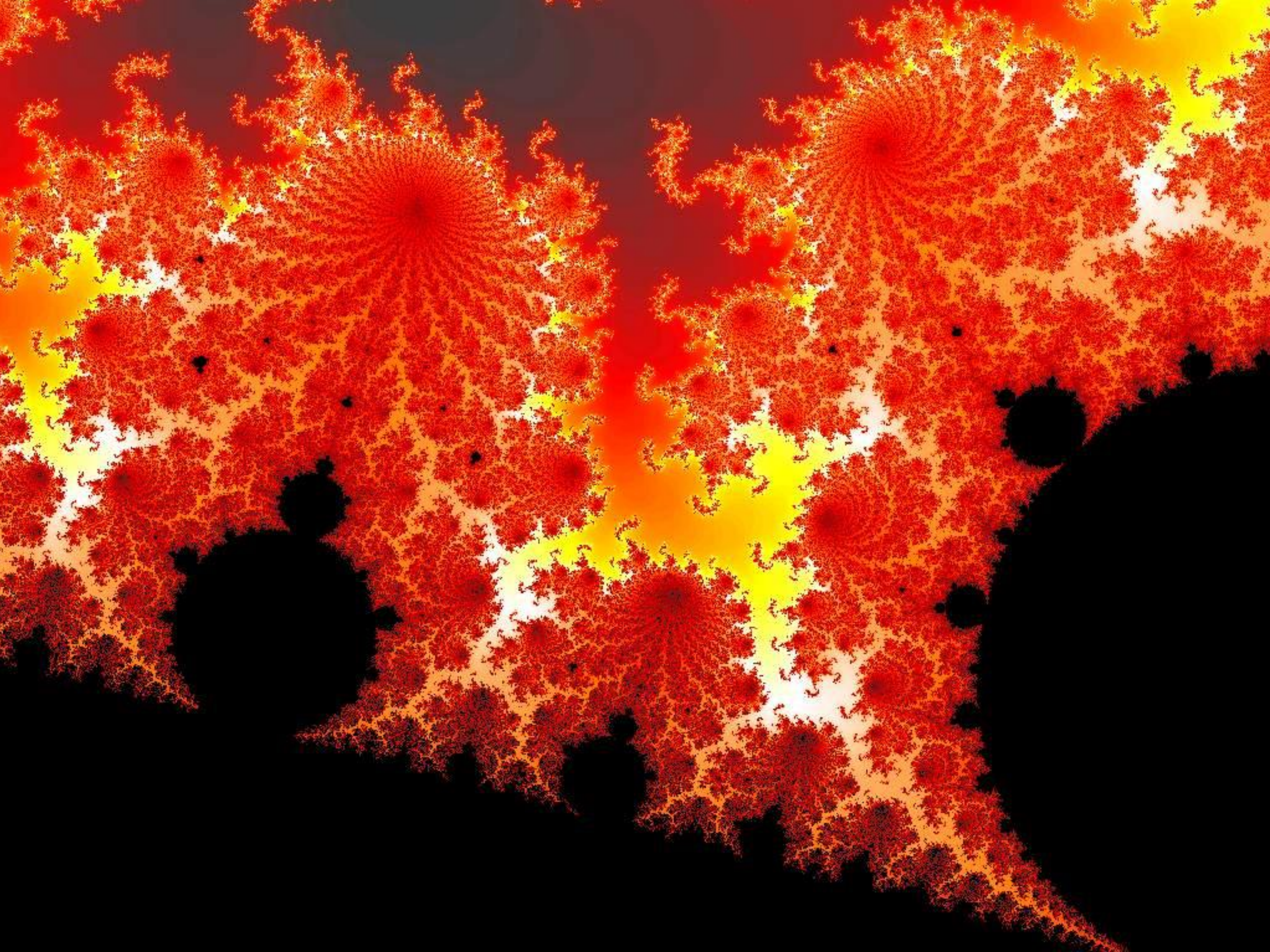
```
def show_pixel(num_cols : float, //resolution in "number of columns", assumed nonzero
               num_rows : float, //resolution in "number of rows", assumed nonzero
               px : float,       //pixel x-coord
               py : float        //pixel y-coord
              ) : unit {
  let xmin = -2.5;
  let xmax = 1.0;
  let ymin = -1.0;
  let ymax = 1.0;

  /* scale [px] and [py] to range [xmin..xmax) and [ymin..ymax) respectively */
  let x0 = xmin + px*(xmax-xmin)/num_cols;
  let y0 = ymin + py*(ymax-ymin)/num_rows;

  let x = ref 0.0;
  let y = ref 0.0;
  let i = ref 0;
  let max_iters = 255;

  while (!i < max_iters && (!x*!x + !y*!y < 4.0)) {
    let xtemp = !x*!x - !y*!y + x0;
    y := 2.0*!x*!y + y0;
    x := xtemp;
    i := !i + 1
  };
  show_density(!i)
}
```

Draw (px,py) with density proportional to #iters before “escape”

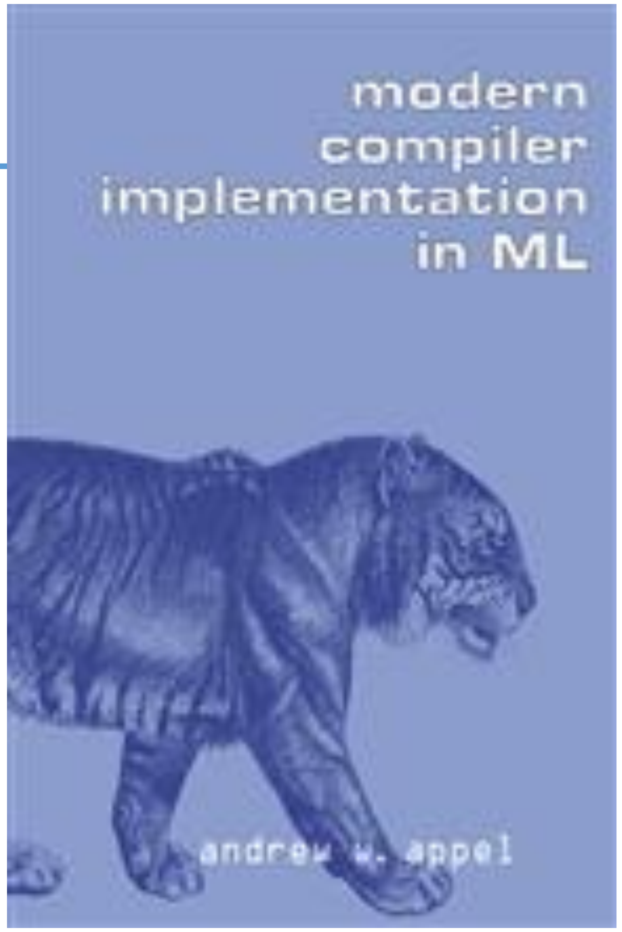
Mandelbrot in **grumpy**

```
def show_newline() : unit {
  let ascii_cr = 13;
  let ascii_lf = 10;
  putchar(ascii_cr);
  putchar(ascii_lf)
}

def mandelbrot() : unit {
  let xend = 50.0;
  let yend = 20.0;
  let x = ref 0.0;
  let y = ref 0.0;

  while (!y < yend) {
    while (!x < xend) {
      show_pixel(xend, yend, !x, !y);
      x := !x + 1.0
    };
    show_newline();
    x := 0.0;
    y := !y + 1.0
  }
}

mandelbrot();
show_newline()
```

Dimensions of “canvas”



Textbook



Modern Compiler Implementation in ML

Andrew W. Appel

Available [free online for OU students]

Supplementary Texts:

- Real World OCaml
- the OCaml language manual
- the OCaml Batteries Included documentation
- the LLVM reference manual
- Types and Programming Languages (TAPL)

Website & Syllabus

http://ace.cs.ohio.edu/~gstewart/courses/4100-17

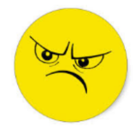
Grading

	%Grade
Course Project Assignments	40
Lecture Attendance and Participation	5
Midterm Exam	15
Final Exam	25
Quizzes	10

+5

?

Course Project

Implement your own compiler for a small imperative language, Grumpy  , over the course of about 12 weeks.

FRONTEND

- Weeks 3-5:
  - **A2:** Lexical Analysis (Lexing)
  - **A3:** Syntax Analysis (Parsing)
  - Building on: REs, DFAs, NFAs, context-free grammars, recursive descent parsing, predictive parsing, …
- Weeks 6-9:
  - **A4:** Typechecking
  - Building on: symbol tables, abstract syntax, type systems, subtyping, …

Course Project

Implement your own compiler for a small imperative language, Grumpy  , over the course of about 12 weeks.

BACKEND

- Weeks 10-12:
  - **A5:** Static Single Assignment (SSA)
  - Stack layout and activation records, control-flow graphs, dominator computation, optimizations
- Weeks 13-15:
  - **A6:** Code generation (targeting LLVM)
  - LLVM assembly and the LLVM compiler toolkit
  - Other stuff: runtimes, garbage collection, instruction selection, register allocation

Late Homework Policy

Up to 24 hours late, **no deduction**

But no more than 2 homeworks late per student over the course of the semester

- $>$ 24 hours late = 0%
- 3rd late homework = 0%

Why only 24 hours? Why not 1 week?

- Late homeworks make it much more difficult to get graded assignments back to you in a timely fashion
- Typically, we'll be grading all the assignments in batch mode, using an automated testsuite

Participation

PROS

+ Showing up to lecture on time, being engaged and asking questions
+ Coming to office hours
  Try to come at least once; I want to meet all of you!
+ Seeking help, if you need it, during lab hours

CONS

- Missing lecture
- Disrupting class (showing up late, sleeping)
- Being rude to the TA (i.e., Alex)

Exams

The usual sort of thing:

Midterm (~15%)

- Projected date: Thursday 2/25 (subject to change)
- The week right before Spring Break

Final Exam (~25%)

- Projected date: Sometime during finals period

Fair game on both exams:

- Anything we cover in class
- Anything in required readings
- Questions related to course assignments

Quizzes

- Every Tuesday, we’ll have a quiz with probability 1/3
- Typically one question; graded leniently

1 2 3

- On weeks when no assignment is due, we’ll also have offline quizzes (in Blackboard)

Quizzes

- Every Tuesday, we'll have a quiz with probability 1/3
- Typically one question; graded leniently

1

2

3

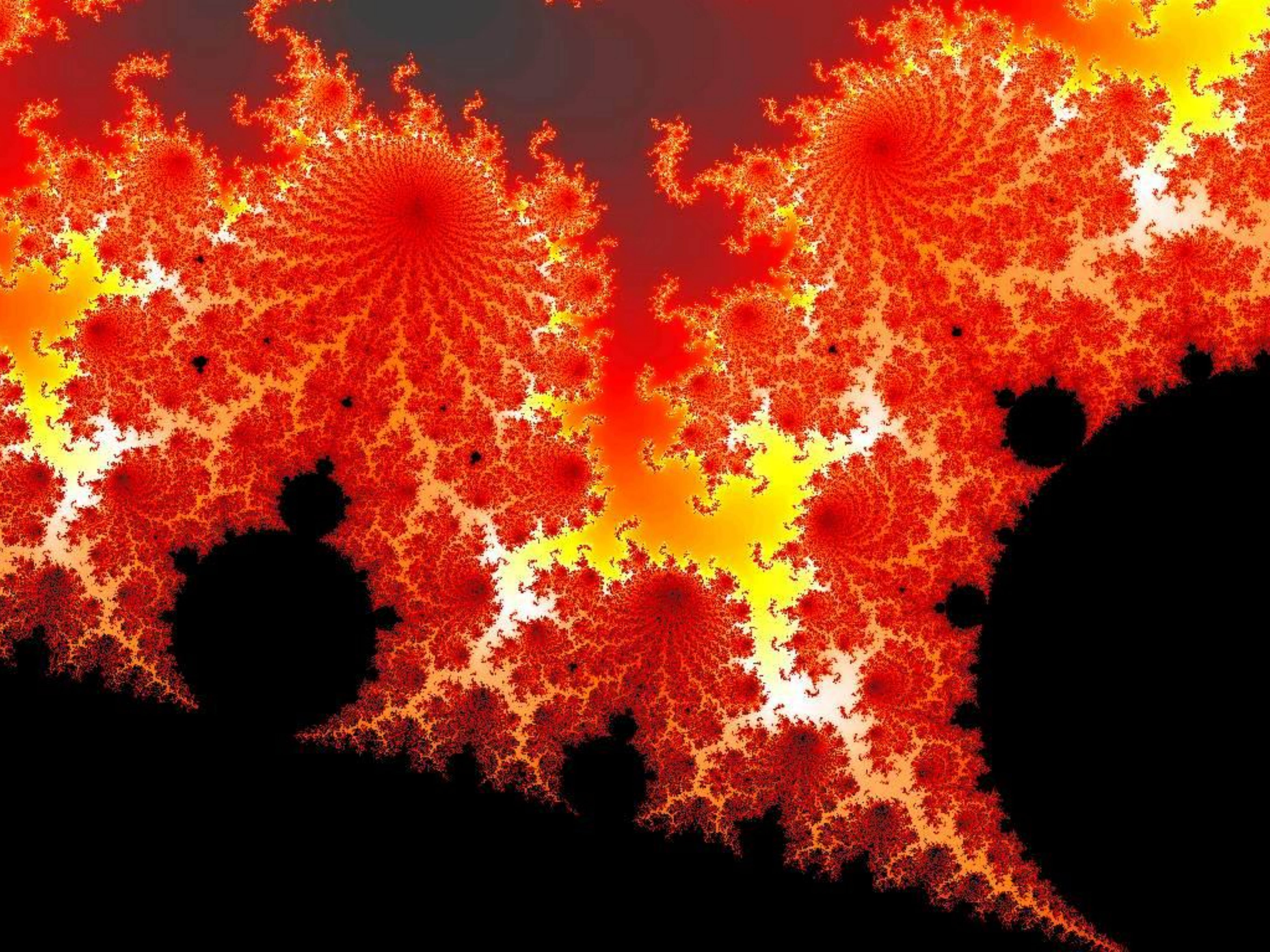
Quiz 0

On the FRONT of your note card, please write:

1. Your name
2. Your year (e.g., junior, senior)
3. Which programming languages are you most comfortable with?
4. Have you used a functional PL before?

On the BACK:

5. It's possible to convert any NFA to an equivalent DFA (True, False, or Wha?!?)



Assignment 0

Due date:

Tuesday 1/17, before the beginning of class (1:30pm)

Goals:

To get you set up with a working environment for the remainder of the course

To get you started with programming in OCaml

Lab Hours

Monday, 1/16

Stocker 307

4-5pm

Goals:

To ***help you*** get set up with a working environment for the remainder of the course

Both Alex and I will be there, at least for lab hours 0

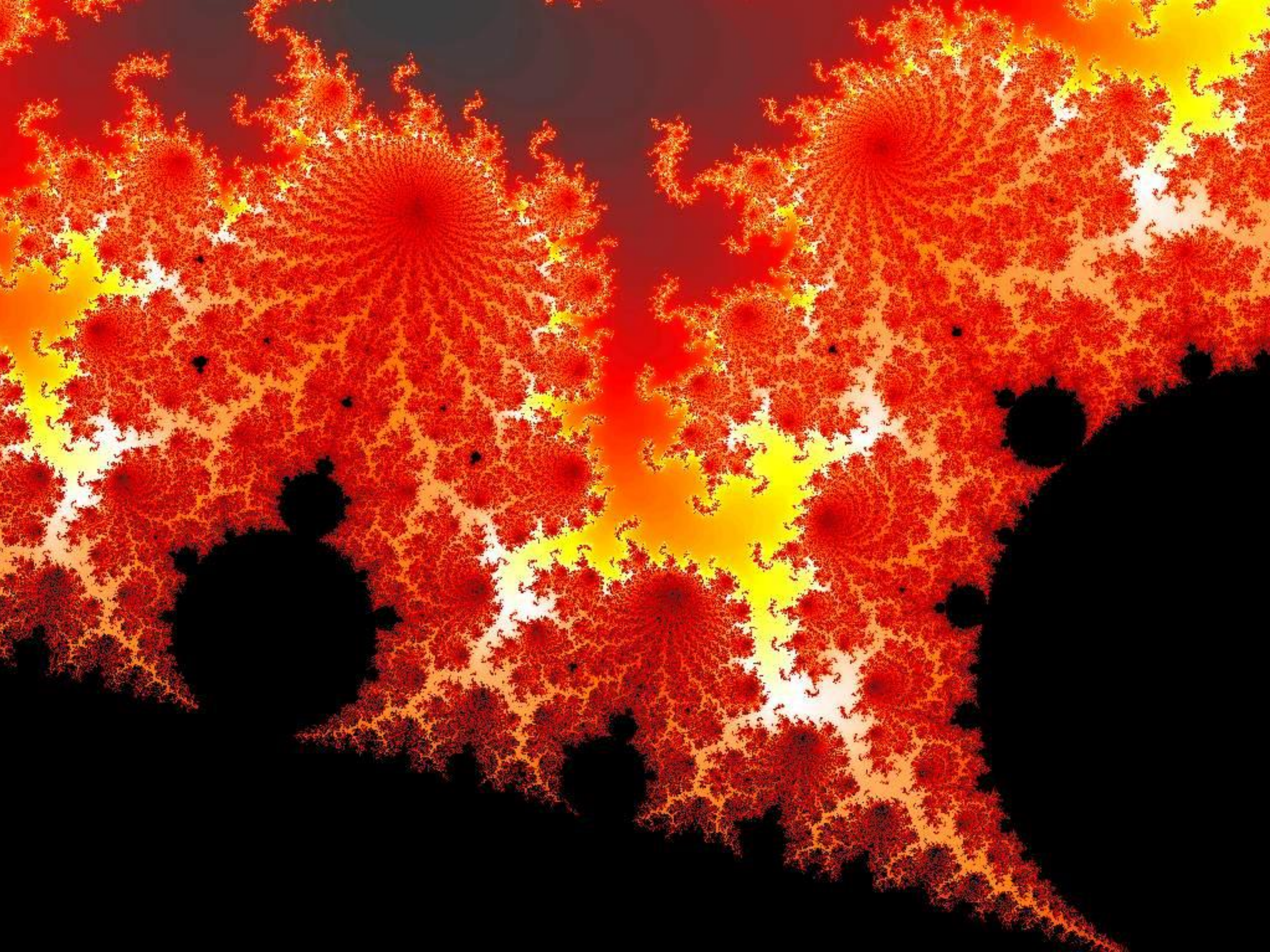
Piazza

This year's Piazza page:

https://piazza.com/ohio/spring2017/cs41005100/home

2016:

https://piazza.com/ohio/spring2016/cs41005100/home



OCaml

- Functional programming language in development since c. 1985
- Higher-order functions

```
twice f x = f (f x)
twice : (int -> int) -> int -> int
```

- Algebraic data types & pattern-matching

```
type intlist = nil
            | cons of int * intlist;;
let l = cons (3, nil) in
match l with
  | nil -> (*do something*)
  | cons (hd, tl) -> (*do something else*)
```

Why OCaml?

- Algebraic data types and pattern matching make it super easy to define the **abstract syntax** of programming languages

```
(** [raw_exp] (and the related [exp]) is the main datatype. *)
type 'a raw_exp =
  | EInt of int32                         (** 32-bit integers *)
  | EFloat of float                       (** Double-precision floats *)
  | EId of id                             (** Program identifiers [x, y, z, ...] *)
  | ESeq of ('a exp) list                 (** [e1; e2; ...; eN] *)
  | ECall of id * ('a exp) list           (** [f(e1, e2, ..., eM)] *)
  | ERef of 'a exp                        (** Allocate a reference cell *)
  | EUnop of unop * 'a exp                (** Apply a unary operation, e.g. [-e] *)
  | EBinop of binop * 'a exp * 'a exp     (** Apply a binary operation e.g., [e1+e2] *)
  | EIf of 'a exp * 'a exp * 'a exp       (** Conditional [if e1 then e2 else e3] *)
  | ELet of id * 'a exp * 'a exp          (** [let x = e1 in e2] *)
  | EScope of 'a exp                      (** \{ e \} *)
```

- Higher-order functions: generic transformations over abstract syntax
- Strong module system: clean interfaces among components