322 Compilers
Why take this course?

• Understanding tools better; what does the compiler really do?

• Computer Engineering & Architecture people: the compiler is your lens to the world

• Phil Greenpun’s 10th rule of programming:

   Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.
Interpreters vs Compilers

**interpreter** : program $\rightarrow$ answer

**compiler** : program $\rightarrow$ program

// no answer!

no interpreter $\Rightarrow$ programs don’t run
Why Compile?

Performance. That’s the *only* reason.
Why Compile?: Interpreter overhead

\[ \text{addl} \ %rax,\%rbx \]

vs

\[
(\text{define} \ (\text{interp} \ \text{exp})
  (\text{type-case} \ \text{FAE} \ \text{exp})
  \begin{array}{l}
    \text{[num} \ (n) \ \text{(num} \ n)] \\
    \text{[add} \ (\text{lhs} \ \text{rhs}) \\
      (\text{let} \ ([\text{lv} \ (\text{interp} \ \text{lhs})] \\
        [\text{rv} \ (\text{interp} \ \text{rhs})])
        (\text{type-case} \ \text{FAE-Value} \ \text{lv})
        \begin{array}{l}
          \text{[numV} \ (\text{ln}) \\
            (\text{type-case} \ \text{FAE-Value} \ \text{rv})
            \begin{array}{l}
              \text{[numV} \ (\text{rn}) \ (+ \ \text{ln} \ \text{rn})] \\
              \text{[else} \ (\text{error} \ '\text{interp}')])
            \end{array}
            \text{[else} \ (\text{error} \ '\text{interp}')]])
        \end{array}
      \end{array}
  \end{array}
)\]
Why Compile?: Automate Transformations

- Bad maintenance practices, yet profitable transformations

For example unrolling loops; when the chip sees straight-line code it can go faster:

It can “look ahead” and thus make good guesses about what is going to happen next,

filling in caches early, keeping the pipeline full, etc
Why Compile?: Automate Transformations

• Lower-level details are exposed in destination language

For example, variables might live on the stack or in registers; want to use registers as much as possible
Goalposts

Build a compiler accepting a language (L5) that has:

• Higher-order functions
• Safe, mutable arrays
• Arithmetic on (bounded) integers
• Recursive binding form
• Conditionals

and producing x86-64 assembly
Fib in L5

(letrec ([fib
            (lambda (n)
                (if (= n 0)
                    0
                    (if (= n 1)
                        1
                        (+ (fib (- n 1))
                            (fib (- n 2))))))])

(print (fib 10)))
Fib in L4

no more higher-order functions

```
((print (:fib 10))
 (:fib (n)
   (if (= n 0)
     0
     (if (= n 1)
       1
       (+ (:fib (- n 1)) (:fib (- n 2)))))
```
Fib in L3

every intermediate result has a name

((let ([fibten (:fib 10)])
    (print fibten))
 (:fib
  (n)
  (let ([niszero (= n 0)])
   (if niszero
    0
     (let ([nisone (= n 1)])
      (if nisone
       1
        (let ([n1 (- n 1)])
         (let ([fn1 (:fib n1)])
          (let ([n2 (- n 2)])
           (let ([fn2 (:fib n2)])
            (+ fn2
             fn1))))))))))}
Fib in L2
no more nested expressions

(:main
  0
  0
  (rdi <- 10)
  ((mem rsp -8) <= :fr)
  (call :fib 1)
  :fr
  (rdi <- rax)
  (rdi *= 2)
  (rdi += 1)
  (call print 1)
  (return))

(:fib
  1
  0
  (cjump rdi = 0 :zero :nonzero)
  :zero
  (rax <- 0)
  (return)
  :nonzero
  (cjump rdi = 1 :one :recur)
  :one
  (rax <- 1)
  (return)
  :recur
  (n <- rdi)
  (rdi -= 1)
  ((mem rsp -8) <= :for)
  (call :fib 1)
  :for
  (result <- rax)
  (n -= 2)
  (rdi <- n)
  ((mem rsp -8) <= :ftr)
  (call :fib 1)
  :ftr
  (rax += result)
  (return))
Fib in L1

no more variables (just registers)

(:main
  0
  0
  (rdi <- 10)
  ((mem rsp -8) <- :fr)
  (call :fib 1)
  :fr
  (rdi <- rax)
  (rdi *= 2)
  (rdi += 1)
  (call print 1)
  (return))

(:fib
  1
  2
  (cjump rdi = 0 :zero :nonzero)
  (zero)
  (rax <- 0)
  (return)
  (one)
  (cjump rdi = 1 :one :recur)
  (recur
   ((mem rsp 0) <- r12)
   ((mem rsp 8) <- r13)
   (r12 <- rdi)
   (rdi -= 1)
   ((mem rsp -8) <- :for)
   (call :fib 1)
   :for
   (r13 <- rax)
   (r12 -= 2)
   (rdi <- r12)
   ((mem rsp -8) <- :ftr)
   (call :fib 1)
   :ftr
   (rax += r13)
   (r12 <- (mem rsp 0))
   (r13 <- (mem rsp 8))
   (return))

13
Implementation/Project overview

L5 → L4 → L3 → L2, each one step

L2 → L1, multiple steps:
  • spilling
  • graph coloring
  • graph construction
  • liveness analysis

Speed test

2 assignments per step: tests & implementation

There is no real “late” code

Use any PL you want (learn a new one!)
http://www.eecs.northwestern.edu/~robbby/courses/322-2015-spring/

Want to pair program? Send me a note (both members), with the promise on the web page

More admin details, including grading rubric, on website; read it