# CS 421 Final Exam review session

- Outline
  - Overview
  - Your questions
    - General
    - Sample exam problems

# Overview

- Format
  - Comprehensive, but heavy emphasis on 2nd half
  - 2 hours (120 minutes)
  - Closed-book, closed-notes
  - No calculators, no phones, no computers, no talking
  - No clarifications

- Content:
  - MPs
  - Lecture examples
  - Lecture slides
  - Midterm exam
  - Mostly analysis + synthesis, not recall

# Lecture 11-12 – Code generation

- Basic idea: given a statement or expression in the language we are compiling, generate equivalent "virtual machine" instructions

- Example: while loop (with break/continue support)

```
[ while e do S ] = let L1,L2,L3 = genlabel()
                     and (I, t) = [ e ]
                     in
                       JUMP L2
                   L1: [ S ]L3,L2
                   L2: I
                       CJUMP t,L1,L3
                   L3:
```

# Lecture 17-18 – scoping and environments

- Basic idea: which *declaration* of a variable name does each *use* of a variable name correspond to?

- OCaml – static (lexical) scope
  - "Closest enclosing definition"

- Example:
  - ```
    let x = 2
    in let y = x
       in let f z = let x=3 in y+z
          in f x
    ```
    ```

# Lecture 17-18 – scoping and environments

- Implementing scope: environment/closure model
  - Put free variables in an "environment" data structure (set of name -> value pairs)
  - Update the environment as we evaluate expressions

- Closures needed for let expressions and abstraction
  - Actually, let expressions *are* abstractions
  - "let x = a in e"  is just "(fun x -> e) a"
  - <expr, env>

- Inside the body of the function (e)
  - Get *free* variables from the application environment (actual arg)
  - Get *bound* (non-free) variables from the closure environment

# Lecture 17-18 – scoping and environments

- **Example:** `(fun x -> fun y -> x y) (fun y -> y 4) (fun z -> z+1)`

```
parse order:
(f a) b
     f = (fun x -> fun y -> x y), a = (fun y -> y 4), b = (fun z -> z+1)

evaluation order: the same
1. evaluate (f a)
   a) evaluate a = (fun y -> y 4) - cannot simplify further
   b) replace x by a in body of f:
     f' = fun y -> (fun y -> y 4) y
   c) evaluate f' = fun y -> (fun y -> y 4) y
     1) replace all "free" occurences of y by … y
     2) evaluate f'' = fun y -> y 4 - cannot simplify further
2. evaluate (f'' b)
   a) evaluate b = (fun z -> z+1) - cannot simplify further
   b) replace y by b in the body of f'': (fun z -> z+1) 4
   … 4+1 = 5
```

# Lecture 17-18 – parser combinators

- Basic idea:
  - Define some basic top-down parser functions (token, epsilon, …)
  - Define higher-order functions for combining parser functions
  - Build more complex parsers out of simpler parser functions
- Parser to recognize a single token:

```
let token s = fun cl -> if cl=[] then None
                  else if s=hd cl then Some (tl cl)
                  else None;;


let parsex = token ‘x’;;
```

# Lecture 17-18 – parser combinators

- "Combinators" to combine parsers into larger parsers:

```
let (++) p q = fun cl -> match p cl with None -> None
                       | Some cl' -> q cl';;


let (||) p q = fun cl -> match p cl with None -> q cl
                       | Some cl' -> Some cl';;


let rec parseA cl = ((token 'a' ++ parseB) || token 'b') cl
   and parseB cl = ((token 'c' ++ parseB) || parseA) cl;;
```

# Lecture 19 – function objects

- Basic idea:
  - Write objects that behave like functions (stateless, n args -> 1 output, operate on other function objects, etc.)
  - Implement functional programming style in imperative O-O languages

- What's involved:
  - Interface – defines "type signature", e.g., int -> int -> bool
  - Function object – implements the interface (body of function)
  - Anonymous inner class = anonymous function (fun x -> …)
  - Operator overloading
    - Rather than defining the .apply() method, redefine the () operator
    - new Incr(2) instead of (new Incr).apply(2)

# Lecture 19 – function objects

- Function objects in Java

```
interface IntFun {
    int apply(int x);
}
interface IntFun2 {
    int apply(int x, int y);
}

IntFun compose2 (IntFun2 f, IntFun g, IntFun h) {
    return new IntFun {
        int apply(int x) {
            return f.apply(g.apply(x), h.apply(x));
        }
    };
}
```

In Ocaml:

```
let compose2 f g h = fun x -> f(g x, h x)
```

# Lecture 20 – Proof systems

- Basic idea: build a framework for writing proofs without "handwaving"
  - Should be understandable to a computer program
- Example
  - Bad: x > 5, therefore x > 0
  - Good:

$$\text{(Trans)}\quad \frac{x > 5 \quad 5 > 0}{x > 0}$$

  - Read: "If (x > 5) is true and (5 > 0) is true, then by the Transitivity rule (x > 0) is true."
  - If x > 5 and 5 > 0 are axioms, we are done. Otherwise, prove x > 5 and 5 > 0.

# Lecture 20 – Proof systems

- Proof system
  - Judgments – logical propositions we want to test
    - Judgments are boolean predicates (evaluate to true or false)
  - Axioms – judgments assumed to be true without proof
  - Inference rules – relations between judgments
- Proofs
  - Sequence (tree) of inference rules and axioms
  - Rooted at the judgment we want to prove
  - Internal nodes = inference rules
  - Leaves = axioms (if judgment is true)
- We are interested in two particular PSs:
  - Type systems – type checking & type inference
  - Semantics – correctness

# Lecture 20 – Proof systems

- Example: (fun x -> + x x) (+ 3 4) ⇓ 14
- AST

```
                app
               /    \
         abstr        app
        /   |           |
       x   app          +
          /    \       / \
         x      x     3   4
```

- Proof

```
(const)   ------        ------
          3 ⇓ 3         4 ⇓ 4       3+4 ⇓ 7
 (delta)  --------------------------------
      …             (+ 3 4) ⇓ 7           …
(App)  -------------------------------------
          (fun x -> + x x) (+ 3 4) ⇓ 14
```

# Lecture 21 – type systems

- Basic idea: prove expression *e* has type *t*
  - Complication: polymorphic types
- Types contain variables (notated $\alpha$, $\beta$, …)
  - E.g., ‘a list, (‘a * ‘b) list, …
- Variables can be generalized in some circumstances; types with generalized variables are written $\forall \alpha, \beta, \ldots . \tau$, and called *type schemes*

# Lecture 21 – type systems

Application and abstraction rules are the same as in $T_{simp}$. Also add rules for tuples.

(Application) $$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

(Abstraction) $$\frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \text{fun } x \to e : \tau \to \tau'}$$

(Tuple) $$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

# Lecture 21 – type systems

let and letrec are new:

(let)
$$\frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma[x : GEN_\Gamma(\tau')] \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}$$

(letrec)
$$\frac{\Gamma[x : \tau'] \vdash e_1 : \tau' \qquad \Gamma[x : GEN_\Gamma(\tau')] \vdash e_2 : \tau}{\Gamma \vdash \text{let rec } x = e_1 \text{ in } e_2 : \tau}$$

# Lecture 22 – operational semantics

- Basic idea: prove expression *e* has value *v*
  - Evaluate in the same order as the expression is parsed
  - Structure of the AST determines structure of the proof

$$\mathbf{B} = \frac{\overline{[x:4,y:3],\ x \Downarrow 4} \qquad \overline{[x:4,y:3],\ y \Downarrow 3}}{[x:4,y:3],\ x+y \Downarrow 7}$$

$$\mathbf{A} = \frac{\overline{[x:4],\ (\text{fun } y \rightarrow x+y) \Downarrow \langle \text{fun } y \rightarrow x+y\ ,[x:4] \rangle} \qquad \overline{[x:4],\ 3 \Downarrow 3} \qquad \mathbf{B}}{[x:4],\ (\text{fun } y \rightarrow x+y)3 \Downarrow 7}$$

$$\frac{\overline{\varnothing,\ (\text{fun } x \rightarrow (\text{fun } y \rightarrow x+y)3) \Downarrow \langle \text{fun } x \rightarrow (\text{fun } y \rightarrow x+y)3, \varnothing \rangle} \qquad \overline{\varnothing,\ 4 \Downarrow 4} \qquad \mathbf{A}}{\varnothing,\ (\text{fun } x \rightarrow (\text{fun } y \rightarrow x+y)3)4 \Downarrow 7}$$

# Lecture 24 – Hoare logic

- Basic idea: prove correctness of imperative programs
  - Can no longer simply evaluate expressions; have sequence of statements instead
- Hoare formulas (judgments)
  - P {A} Q
  - Precondition-program-postcondition
- Hard part: finding and proving the loop invariant
  - P { while b … } P & !b
- And termination:
  - Define phi(state) s.t. $phi(state_{i+1}) < phi(state_i)$

# Lecture 24 – Hoare logic

$$x = 0 \wedge y = 1 \wedge z = 0 \wedge 1 \leq n \Rightarrow y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n$$

$$y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n \wedge \neg(z < n) \Rightarrow y = \text{fib } n$$

$$y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n \wedge z < n \Rightarrow \quad ?$$

$$x+y = \text{fib } (z+1) \wedge x+y-x = \text{fib } (z+1-1) \wedge z + 1 \leq n$$

$$\{y := x + y\} \quad y = \text{fib } (z+1) \wedge y-x = \text{fib } (z+1-1) \wedge z + 1 \leq n$$

$$\{x := y - x\} \quad y = \text{fib } (z+1) \wedge x = \text{fib } (z+1-1) \wedge z + 1 \leq n$$

$$\{z := z + 1\} \quad y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n$$

---

$$? \quad \{y := x + y;\ x := y - x;\ z := z + 1\} \quad y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n$$

---

$$y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n \wedge z < n \quad \{y := x + y;\ x := y - x;\ z := z + 1\} \quad y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n$$

---

$$y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n \quad \{\text{While } z < n \ldots\} \quad y = \text{fib } z \wedge x = \text{fib } (z-1) \wedge z \leq n \wedge \neg(z < n)$$

---

$$x = 0 \wedge y = 1 \wedge z = 0 \wedge 1 \leq n \quad \{\text{While} \ldots\} \quad y = \text{fib } n$$

# Lecture 25 – Lambda calculus

- Basic idea: minimal set of constructs needed to implement a sequential functional language
- Need:
  - Expressions: vars, abstraction ($\lambda$x.e), application ($e_1 e_2$)
    - What we want to evaluate
  - Values: closed abstractions
    - What we want to evaluate *to*
  - Operational semantics: $\beta$-reduction (and others, but ignore them)
    - *How* we evaluate

- "Everything is a function"
  - No constants, data structures, etc.
  - Define everything as a function

# Lecture 25 – Lambda calculus

- Beta reduction (similar to function application in Ocaml)
  - Replace expression ($\lambda$x.e) e’ by e[e’/x]
  - A.K.A. replace (fun x -> e) e’ by e, with e’ replacing any free occurrences of x in e
  - Similar to Ocaml application rule, except replace the expression before evaluating e’ $\Downarrow$ v
    - Lazy evaluation

- Example: ($\lambda$x.$\lambda$y.x) 1 2   Ocaml: (fun x -> fun y -> x) 1 2
  - Apply beta-reduction 1:
    - e = fun y -> x, e’ = 1, e[e’/y] = fun y -> x
  - Apply beta-reduction 2:
    - e = x, e’ = 2, e[e’/x] = 2

# Lecture 25 – Lambda calculus

```
let pair x y = λf. f x y
let fst p = p (λx. λy. x)
let snd p = p (λx. λy. y)
```

- Example: `fst (pair 4 5)`

$= (\lambda p.\ p\ (\lambda x.\ \lambda y.\ x))\ ((\lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y)\ 4\ 5)$

$\equiv_\beta (\lambda p.\ p\ (\lambda x.\ \lambda y.\ x))\ (\lambda f.\ f\ 4\ 5)$

$\equiv_\beta (\lambda f.\ f\ 4\ 5)\ (\lambda x.\ \lambda y.\ x)$

$\equiv_\beta (\lambda x.\ \lambda y.\ x)\ 4\ 5$

$\equiv_\beta (\lambda y.\ 4)\ 5$

$\equiv_\beta 4$

# Lecture 25 – Lambda calculus

- Church numerals
- Represent *n* by expression:

$$\lambda f.\lambda x.f(f(\ldots(fx)\ldots)) = \lambda f.f \circ f \circ \ldots \circ f = \lambda f.f^n$$

- Example:

```
0 = λf. λx. x
1 = λf. λx. f x ≡η λf. f
2 = λf. λx. f (f x) = λf. f ∘ f
3 = λf. f ∘ f ∘ f
```

# Lecture 25 – Lambda calculus

- Define "paradoxical combinator"

$$Y = \lambda f.(\lambda x.f\ (x\ x))\,(\lambda x.f\ (x\ x))$$

- For any f:

$$Y\ f = f\,(Y\ f)$$ (apply β-reduction twice)

- Consider OCaml definition:

```
let rec sum x = if x = 0 then 0 else x+sum(x-1)
```

then consider this definition:

```
let Sum = Y(λsum. λx. if x=0 then 0 else x+Sum(x-1))
```

- Note that definition of Y is not recursive.

# Spring 08 final – problem 2

```
type stmt = Assign of string * expr
          | If of expr * stmt * stmt
          | While of expr * stmt
          | Block of stmt list

and expr = Var of string | Const of int
         | Plus of expr * expr | Less of expr * expr | Not of expr
```

Write a function `trans: stmt → stmt` that makes the following transformations:

- `if (!e) then s1 else s2` ⇒ `if (e) then s2 else s1`
- `{ s }` ⇒ `s` (i.e. a block with a single statement doesn't need to be a block)

These transformations should be performed recursively throughout the term – inside the body of a while, the statements in a block (as well as the block itself), and the true and false branches of an if (as well as the if itself).

```
let rec transform s = match s with
    Assign(x,e) -> s
  | If(Not e,s1,s2) -> If(e, transform s2, transform s1)
  | If(e,s1,s2) -> If(e, transform s1, transform s2)
  | While(e,s) -> While(e, transform s)
  | Block [s] -> transform s
  | Block sl -> Block (map transform sl);
```

# Spring 08 final – problem 4

```
type expr = Int of int  |  Add of expr * expr

let rec fold (f,g) e = match e with
    Int i  -> f i
  | Add(e1,e2) -> g (fold (f,g) e1, fold (f,g) e2)
```

fill in the blanks in the following OCaml session. (Recall that `string_of_int` is the OCaml function to convert an int to a string.):

```
val e1 = Add(Int 3, Add(Int 4, Int 5))

let evaluate e = fold (_(fun n -> n)________________________,

                        _(fun (x,y) -> x+y) ________________) e;;
evaluate e1;;
-: int = 12
let prettyprint e = fold

              (____string_of_int __________________________,

               ___ fun (x,y) -> "("^x^"+"^y^")" ______________) e;;
prettyprint e1;;
-: string = "(3+(4+5))"
```

# Spring 08 final – problem 5b

- Original

```
S → id int
    | id id int
    | D int
D → ε
    | D $
```

- LL(1)

```
S → id T
    | D int
T → int | id int
    | D int            <--- typo
D → ε
    | $ D
```

- Does T include “D int”?  No!
- Is “S -> id T | D int” left-recursive? No!

# Spring 08 final – problem 9c

- This problem is about using higher-order functions

```
let rec fold_right f lis accu =
     match lis with
            [] -> accu
          | h::t -> f h (fold_right f t accu)
```

Write the following OCaml functions:

(c) graph_fun: $(\alpha \rightarrow \beta) \rightarrow \alpha$ list $\rightarrow (\alpha * \beta)$ list, where graph_fun *f [x1; x2; …; xn] = [(x1, f x1); (x2, f x2); …]*

```
let rec graph_fun f x =
 if x=[] then [] else (hd x, f (hd x)):: graph_fun f (tl x)
```

# Spring 08 final – problem 13

- Explain “fun () -> (cnt := !cnt + 1; !cnt)”

- This is about references (lecture 22)
  - () is “unit” – it is the datatype of the := operator
  - := is reference assignment
  - !cnt is dereference variable cnt

- This is a function that takes a unit as argument, and performs the following, in order:
  - Dereference cnt
  - Compute (!cnt + 1)
  - Store this value back in cnt
  - Dereference cnt (and return its value)

# Spring 08 final – problem 10

- This deals with environment updates (lecture 18)

let x = 4;;

$\rho_0$: $\{x \rightarrow 4\}$

---

let f y = fun z -> x + y + z;;

$\rho_1$: $\rho_0[f \rightarrow \langle y, z \text{ -> } x + y + z, \rho_0 \rangle]$

---

let x = 8;;

$\rho_2$: $\rho_1[x \rightarrow 8]$

---

let g = f 6;;

$\rho_3$: $\rho_2[g \rightarrow \langle z, x + y + z, \rho_0[y \rightarrow 6] \rangle]$

---

let x = g x;;

$\rho_4$: $\rho_3[x \rightarrow 18]$

---

# Spring 08 final – problem 11

- This is an operational semantics proof in OSclo
- Similar to the lecture example given above

# Spring 08 final – problem 12a,b

- a) we didn't cover dynamic semantics; use OSsubst or OSclo
- b) we just apply the type rules (from the exam, not the lecture)

a. Give a dynamic semantics rule for this expression:

$$\frac{\rho, e_1 \Downarrow v_1 \qquad \rho[x \to v_1], e2 \Downarrow v_2 \qquad \rho[x \to v_1, y \to v_2], e \Downarrow v}{\rho, \texttt{let x = e1 then y = e2 in e} \Downarrow v}$$

b. Give a type rule for this expression (in the non-polymorphic type system):

$$\frac{\Gamma \vdash \mathbf{e1}: \tau_1 \qquad \Gamma[x: \tau_1] \vdash \mathbf{e2}: \tau_2 \qquad \Gamma[x: \tau_1, y: \tau_2] \vdash \mathbf{e}: \tau}{\Gamma \vdash \texttt{let x = e1 then y = e2 in e}: \tau}$$

# Spring 08 final – problem 14

- This is a straight-forward type proof
  - Gamma implies "let x = 1 in cons x nil" has type "int list"

Γ= {cons: int → int list → int list, nil: int list }.

Give the proof tree for the type judgment below, using the lines provided. On each line, give the name of the inference rule being used. Recall that axioms have a line with nothing above it. The axioms and rules of inference for the system are given at the end of the exam.

______________________ *Variable* ___ ______ *Variable* __

**Γ[x:int]** |- **cons : int → int list → int list** **Γ[x:int]** |- **x : int**

*App* ______________________________________ _________ *Variable* _

**Γ[x:int]** |- cons x : int list → int list **Γ[x:int]** |- nil : int list

____ *Const* ______________________________________ *App*

**Γ** |- 1:int **Γ[x:int]** |- cons x nil : int list

__________________________________________________ *Let*

Γ |- let x = 1 in cons x nil : int list

# Spring 08 final – problem 15

- We didn't cover this; ignore

# Spring 08 final – problem 16

- Hoare logic problem: give invariant and prove termination

```
i = 0; j = n-1;
while (i < j) {
   if (a[i] <= x)
      i = i+1;
   else if (a[j] > x)
      j = j-1;
   else {
      temp = a[i]
      a[i] = a[j]
      a[j] = temp
      i = i+1
      j = j-1
   }
}
```

(a) Give the loop invariant for the loop.

$$\exists i, j.\ (0 \le i \le j \le n\ \wedge (\forall m.\ 0 \le m < i \Rightarrow a[m] \le x) \wedge (\forall m.\ j \le m < n \Rightarrow a[m] > x))$$

(b) Give a well-founded ordering on the variables that proves the termination

**Numerical ordering on j-i. (Declines on every iteration; cannot g**

The correctness formula for this statement is:

true { `i=0; j=n-1; while ...` } $\exists k.\ (0 \le k \le n\text{-}1 \wedge (\forall m.\ 0 \le m < k \Rightarrow a[m] \le x) \wedge (\forall m.\ k < m < n \Rightarrow a[m] > x))$

# Outline

- Spring 08 final:
  - 14-17