# CMSC330 Spring 2009 Final Exam (Solutions)

| | Problem | Score | Max Score |
|---|---|---|---|
| 1 | Programming languages | | 14 |
| 2 | Regular expressions & CFGs | | 8 |
| 3 | Finite automata | | 10 |
| 4 | Parsing | | 12 |
| 5 | OCaml types & type inference | | 12 |
| 6 | OCaml programming | | 10 |
| 7 | Scoping | | 8 |
| 8 | Polymorphism | | 9 |
| 9 | Multithreading | | 20 |
| 10 | Lambda calculus | | 16 |
| 11 | Lambda calculus encodings | | 16 |
| 12 | Operational semantics | | 8 |
| 13 | Markup languages | | 8 |
| 14 | Garbage collection | | 9 |
| | Total | | 160 |

1. (1) Programming languages
   a. (6 pts) List 3 different design choices for *parameter passing* in a programming language. Which choice is seldom used in modern programming languages? Explain why.
      **Design choices (need 3) = call-by-value, call-by-reference,**
      **call-by-name, call-by-result, call-by-value-result, call-by-need**
      **Seldom used = everything except call-by-value or call-by-reference**
      **Reason = highly complex, inefficient, can be confusing**
   b. List 2 different design choices for *type declarations* in a programming language. Which choice is seldom used in modern programming languages? Explain why.
      **Design choices (need 2) = explicit, implicit**
      **Seldom used = implicit**
      **Reason = requires static types, requires type inference,**
      **error messages can be confusing**
   c. List 2 different design choices for determining *scoping* in a programming language. Which choice is seldom used in modern programming languages? Explain why.
      **Design choices (need 2) = static lexical, dynamic**
      **Seldom used = dynamic**
      **Reason = can be confusing**

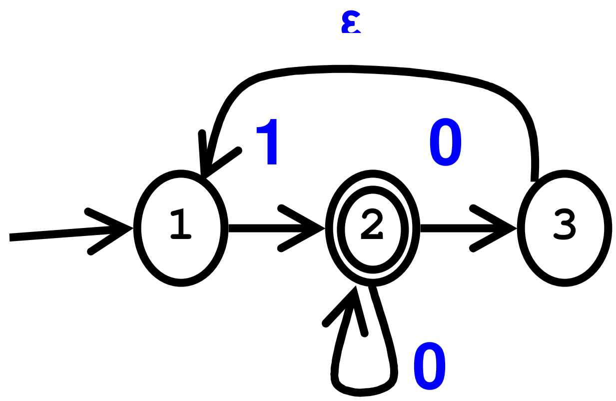
2. (8 pts) Regular expressions and context free grammars
   Give a
   a. Regular expression for binary numbers with an even number of 1s.
      **(0*10*10*)*| 0***

   b. Context free grammar for binary numbers with twice as many 1s as 0s
      **S → S1S1S0S | S1S0S1S | S0S1S1S | epsilon**

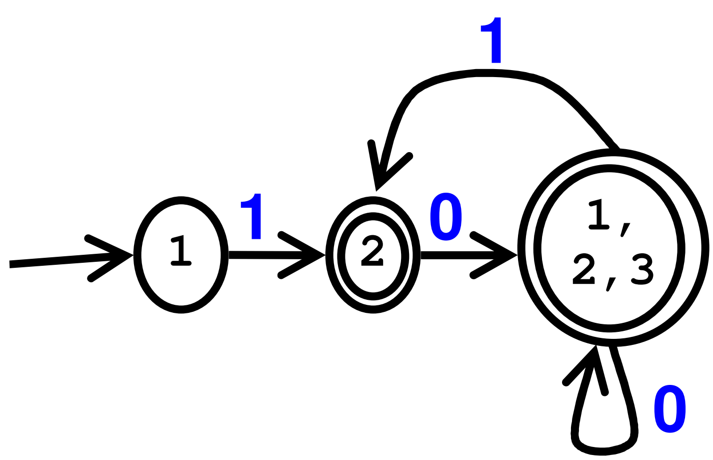
      **Many possible answers, one possible solution above.**

3. (10 pts) Finite automata

Apply the subset construction algorithm to convert the following NFA to a DFA.
Show the NFA states associated with each state in your DFA.




**Answer**



4. (1) Parsing

Consider the following grammar:

$S \rightarrow Ac \mid a$

$A \rightarrow bS \mid epsilon$

a. (6 pts) Compute First sets for S and A

**First(S) = { a, b, c }** //

**First(A) = { b, epsilon }** //

**Partial credit for**

**First(S) = { a }** //

**First(S) = { a, b }** //

**First(S) = { a, b, epsilon }** //

**First(A) = { b }** //

b. (6 pts) Write the parse_A( ) function for a predictive, recursive descent parser for the grammar (You may assume parse_S( ) has already been written, and match( ) is provided).

```
parse_A( ) {
  if (lookahead == 'b') {      // for correct lookahead
    match('b');                // for correct body
    parse_S( );
  }
  else ;       //    just return for other lookaheads
}
```

5. (1) OCaml Types and Type Inference

a. Give the type of the following OCaml expression

let f x y z = y (x z)

**Type = ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c** //

b. (6 pts) Write an OCaml expression with the following type

int -> (int * int -> 'a) -> 'a

**Code = let f x y = y (2, x+1)**

**Many possible answers, use OCaml interpreter if unclear. Partial credit**

**() function with 2 curried arguments. E.g., let f x y = …**

**() 1st argument is an int. E.g., let f x y = x+1**

**() 2nd argument is a function. E.g., let f x y = y …**

c. Give the value of the following OCaml expression. If an error exists, describe the error.

let x y = x in 3

**Value = error, unbound symbol x** // **if unbound x error**

6. (10 pts) OCaml Programming

Consider the OCaml type *bst* implementing a binary tree:

```
type tree =
    Empty
    | Node of int * tree * tree;;

let rec equal = …                (* type =  (tree * tree) -> bool *)
```

Implement a function *equal* that takes a tuple argument (t1, t2) that returns true if the two trees t1 and t2 are of the same shape *and* equivalent nodes in the trees have the same value, else returns false.

```
let rec equal = function
   (Empty, Empty) -> true                          //   true if both empty
 | (Node(m1, l1, r1), Node(m2, l2, r2)) ->         //   pull apart both trees
     m1 = m2 &&                                    //   check values
     (equal (l1, l2)) &&                           //   check subtrees
     (equal (r1, r2))
 | _ -> false                                      //   false otherwise
```

**Other possible answers using "match" to pull apart parts of tree**

7. (8 pts) Scoping

Consider the following OCaml code.

```
let app f y = let x = 5 in let y = 7 in let a = 9 in f y ;;
let add x y = let incr a = a+y in app incr x ;;
 (add 1 (add 2 3)) ;;
```

a. What value is returned by (add 1 (add 2 3)) with static scoping? Explain.

**17, since the y in incr is bound to the formal parameter y for add**

**The sequences of calls & resulting values bound to the formal parameters is as follows.**

i. **First evaluate (add 2 3) since arguments are evaluated first**
ii. **add (x=2,y=3) calls app (f=incr, y=2) binds (x=5, y=7, a=9) calls incr (a=7, since when incr is called the argument y is bound to 7)**
iii. **In the body of incr y is free and refers to the y in add x y (y=3), leading to a+y=7+3=10**
iv. **(add 1 (add 2 3)) is evaluated next, with the 2nd argument (add 2 3) having value 10**
v. **add (x=1,y=10) calls app (f=incr, y=1) binds (x=5, y=7, a=9) calls incr (a=7, since when incr is called the argument y is bound to 7)**
vi. **In the body of incr y is free and refers to the y in add x y (y=10), leading to a+y=7+10=17**

b. (6 pts) What value is returned by (add 1 (add 2 3)) with dynamic scoping? Explain.

**14, since the y in incr is bound to the y=7 in app**

**The sequences of calls & resulting values bound to the formal parameters is as follows.**

**Note "let z=5 in …" is really "(fun z-> …) 5" and adds a dynamic scope.**

i. **First evaluate (add 2 3) since arguments are evaluated first**
ii. **add (x=2,y=3) calls app (f=incr, y=2) binds (x=5, y=7, a=9) calls incr (a=7, since when incr is called the argument y is bound to 7)**
iii. **In the body of incr y is free and refers to the y in let y = 7 (y=7) in the body of app, leading to a+y=7+7=14**
iv. **(add 1 (add 2 3)) is evaluated next, with the 2nd argument (add 2 3) having value 14**
v. **add (x=1,y=14) calls app (f=incr, y=1) binds (x=5, y=7, a=9) calls incr (a=7, since when incr is called the argument y is bound to 7)**
vi. **In the body of incr y is free and refers to the y in let y = 7 (y=7) in the body of app, leading to a+y=7+7=14**

(9 pts) Polymorphism

Consider the following Java classes:

class A { public void a( ) { … } }
class B extends A { public void b( ) { … } }
class C extends B { public void c( ) { … }}

( each) Explain why the following code is or is not legal

c. int count(Set<B> s) { … } … count(new TreeSet<C>( ));

**() Illegal**

**Actual parameter type (Set<C>) is not a subclass of formal parameter type (Set<B>), even though C is a subclass of B.**

d. int count(Set<? extends B> s) { … } … count(new TreeSet<C>());

**() Legal**

**Actual parameter type (Set<C>) matches formal parameter type (Set<? extends B>), since "? extends B" can match B and its subclass C**

e. int count(Set<? super C> s) { for (A x : s) x.a( ); … }

**() Illegal**

**Elements of s may be objects of class Object.**

8. (20 pts) Multithreading

Using Ruby monitors and condition variables, you must implement a multithreaded simulation of factories producing chopsticks for philosophers. Factories continue to *produce* chopsticks one at a time, placing them in a single shared market. The market can only hold 10 chopsticks at a time. Philosophers enter the market to *acquire* 2 chopsticks.

Helpful functions:

```
m = Monitor.new         // returns monitor
m.synchronize { … }     // only 1 thread can execute code block at a time
c = m.new_cond          // returns conditional variable for monitor
c.wait_while { … }      // sleeps while code in condition block is true
c.broadcast             // wakes up all threads sleeping on condition var
t = Thread.new {… }     // creates thread, executes code block in new thread
t.join                  // waits until thread t exits
```

a. (1) Implement a thread-safe class Market with methods initialize, produce, and acquire that can support multiple multi-threaded factories and philosophers.

```
require "monitor.rb"
class Market
  def initialize
     # initialize synchronization, number of chopsticks
     @current = 0
     @myLock = Monitor.new
     @myCondition = @myLock.new_cond
  end
  def produce
     # produces 1 chopstick if market is not full ( < 10 )
     # increases number of chopsticks in market by 1
     @myLock.synchronize {
          @myCondition.wait_while { @current >= 10 }
          @current = @current + 1
          @myCondition.broadcast
      }

  end
  def acquire
     # acquires 2 chopsticks if market has 2 or more chopsticks
     # decreases number of chopsticks in market by 2
     @myLock.synchronize {
          @myCondition.wait_while { @current < 2 }
          @current = @current - 2
          @myCondition.broadcast
      }
  end
end

     // – producers increment chopstick count by 1 when < 10
     // – acquire decrements chopstick count by 2 when > 2
     //  – initialize inits count, allocates monitor, condition var
     //  – proper use of synchronization
     //  – proper use of condition variable
     //  – proper use of broadcast
```

b. (6 pts) Write a simulation with 2 factories and 2 philosophers using the market. Each factory and philosopher should be in a separate thread. The simulation should exit *after* both philosophers acquire a pair of chopsticks.

```
market = Market.new

factory1 = Thread.new {
        while true
                market.produce
        end
}
factory2 = Thread.new {
        while true
                market.produce
        end
}

philosopher1 = Thread.new { market.acquire }
philosopher2 = Thread.new { market.acquire }

philosopher1.join
philosopher2.join

// Max 6 pts
// 2 pt – new Market
//  – factory keep calling Market.produce
//  – philosophers call Market.acquire
//  – join
```

9. (16 pts) Lambda calculus

Find all free (unbound) variables in the following λ-expressions

a. (λa. c b) λb. a

**(λa. c b) λb. a** **// rightmost a**

**// () b in body of 1st λ**

**// () c**

( each) Evaluate the following λ-expressions as much as possible

b. (λx.λy.y x) a b

**(λx.λy.y x) a b → (λy.y a) b → b a** **// if stopped halfway**

c. (λz.z x) (λy.y x)

**(λz.z x) (λy.y x) → (λy.y x) x → x x** **// if stopped halfway**

d. Write a small λ-expression which requires alpha-conversion to evaluate properly.

**(λx.λy.x) y // argument of 1st λ matches formal parameter of 2nd λ in body**

10. (16 pts) Lambda calculus encodings

Prove the following using the appropriate λ-calculus encodings, given**:**

1 = λf.λy.f y
2 = λf.λy.f (f y)
3 = λf.λy.f (f (f y))
4 = λf.λy.f (f (f (f y)))
M * N = λx.(M (N x))
Y = λf.(λx.f (x x)) (λx.f (x x))
succ = λz.λf.λy.f (z f y)

a. (10 pts) 2 * 2 = 4

| | |
|---|---|
| **(2 * 2)** | **// replacing * w/ encoding** |
| **= λx.(2 (2 x))** | **// () replacing 2 w/ encoding** |
| **= λx.(2 ((λf.λy.f (f y)) x))** | **// (s) β-reduction: f → x** |
| **= λx.(2 (λy.x (x y)))** | **// () replacing 2 w/ encoding** |
| **= λx.((λf.λy.f (f y)) (λy.x (x y)))** | **// a-conversion: y → a** |
| **= λx.((λf.λa.f (f a)) (λy.x (x y)))** | **// () β-reduction: f → λy.x (x y)** |
| **= λx.(λa. (λy.x (x y)) ((λy.x (x y))a))** | **// () β-reduction: 2nd y → a** |
| **= λx.(λa. (λy.x (x y)) (x (x a)))** | **// () β-reduction: y → x (x a)** |
| **= λx.(λa. x (x (x (x a))))** | **// () apply encoding for 4** |
| **= 4** | **// () result** |

b. (6 pts) (Y succ) x = succ (Y succ) x // you do not need to expand succ

| | |
|---|---|
| **(Y succ) x** | **// replace Y w/ encoding** |
| **= (λf.(λx.f (x x)) (λx.f (x x)) succ) x** | **// 1st f → succ** |
| **= (λx.succ (x x)) (λx.succ (x x)) x** | **// 1st x → λx.succ (x x)** |
| **= (succ ((λx.succ (x x)) (λx.succ (x x)))) x** | **// (s) encoding for (Y succ)** |
| **= (succ (Y succ)) x** | **// (s) result** |

11. (8 pts) Operational semantics

Use operational semantics to determine the values of the following OCaml codes:

(fun x = + 4 x ) 2

| | |
|---|---|
| **• ; (fun x = + 4 x) → (• , λx.+ 4 x)** | **// evaluate function to closure** |
| **• ; 2 → 2** | **// evaluate argument** |
| **( x:2, + 4 x) → 6** | **// evaluate body in extended env** |
| **• ; (fun x = + 4 x) 2 → 6** | **// result of proof** |

12. (8 pts) Markup languages

Creating your own XML tags, write an XML document that organizes the following information: Yoda is a 900 year old Jedi with rank Grandmaster, Obi-Wan is a 36 year old Jedi with rank Master, Anakin is an 9 year old Jedi with rank Padawan.

```
<JediList>
    <Jedi>
        <name>Yoda</name>
        <age>900</age>
        <rank>Grandmaster</rank>
    </Jedi>
    <Jedi>
        <name>Obi-Wan</name>
        <age>36</age>
        <rank>Master</rank>
    </Jedi>
    <Jedi>
        <name>Anakin</name>
        <age>9</age>
        <rank>Padawan</rank>
    </Jedi>
</JediList >
```

**// Top level tag <JediList> around all info**
**// Individual Jedi tags <Jedi>**
**// Name, age, rank information**
**// All tags balanced <tag> … </tag>**

13. (9 pts) Garbage collection

Consider the following Java code.

```
Jedi Darth, Anakin;
private void plotTwist( ) {
        Anakin = new Jedi( );  // object 1
        Darth = new Jedi( );     // object 2
        Darth = Anakin;
        Anakin = Darth;
}
```

a. What object(s) are garbage when plotTwist ( ) returns? Explain why.

**Object 2 is garbage because it is no longer reachable (once the reference to it is overwritten by "Darth = Anakin;" )**

b. (3pts) Explain why stop-and-copy has to copy live objects.

**Live objects must be moved to a new semi-space since all objects in the current semi-space will be freed.**

c. How can garbage collection take advantage of the fact an object is from an older generation?

**Objects from older generations are presumed longer-lasting and do not need to be processed as frequently (i.e., can be moved to a separate semi-space that is not checked as frequently by garbage collection)**