

CMSC330 – Organization of Programming Languages
Fall 2022
Final Exam

CMSC330 Course Staff
University of Maryland
Department of Computer Science

December 14th, 2022

- You get 5 points if you do not remove the staple or any sheets from your exam packet.
- Do not tear out any individual sheets from the exam packet
- Write your name and UID in the header of each page.
- Refrain from bending or folding the exam in any place except near the staple, this helps us when scanning your exams.
- **Read *all* questions carefully before starting.**
- Table of points-per-question is on the back of the packet (if you want to strategize with your time).

NAME: _____

UID: _____

UID:

1. For each code listing, fill in the blank such that the code would result in the given output:

(a) The following should print 15

```
arr = [2, 4, 6, 8, 10]
sum = 0
arr.each _____blank 1_____
puts sum          # prints 15
```

Blank1:

(b) The following should print {2=>16, 4=>36, 6=>64, 8=>100}

```
def mycons(x,h)
  for i in x
    h[i] = _____blank 2_____
  end
  h
end
hsh = mycons([2,4,6,8], Hash.new){|x| x^2}
puts hsh # prints {2=>16, 4=>36, 6=>64, 8=>100}
```

Blank2:

UID:

- (c) The following should print [1, 3, 5]

```
arr2 = [1, 2, 3, 4, 5, 6]
result = arr2.select _____blank 3_____
puts result      # prints [1, 3, 5]
```

Blank3:

- (d) The following should print ‘NY => New York; MD => Maryland; VA => Virginia’

```
result_string = ""
hsh = {"NY": "New York", "MD": "Maryland", "VA": "Virginia" }
hsh.each _____blank 4_____
puts result_string      # prints ‘NY => New York; MD => Maryland; VA => Virginia; ’
```

Blank4:

UID:

Here are the type signatures for some useful OCaml functions:

```
map : ('a -> 'b) -> 'a list -> 'b list
```

```
fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
cons: 'a -> 'a list -> 'a list
```

```
append: 'a list -> 'a list -> 'a list
```

2. Typing OCaml Expressions

Write the types of the following OCaml expressions. If the expression doesn't type check, just write "type error" with no explanation required.

(a) `let x a b = fun a b -> a :: (a = b)`

(b) `fun x a -> fun x b -> x :: [x+1] :: a :: [b]`

(c) `fun x a -> (x a) :: a :: ["cmsc330"]`

UID:

3. Writing instances of OCaml Types

Without any additional type information, write an expression with the following provided types, if it is not possible write “impossible”. Any pattern matches in your answers must be exhaustive.

(a) (`'a -> 'b`) -> `'a -> 'b -> 'b list`

(b) `int -> int -> bool`

(c) (`'a * 'b`) -> (`'a -> 'b -> 'c`) -> `'c -> bool list`

UID:

4. 10 points Implement the `zip` function that takes in two lists (`list1` and `list2`) and returns a list of tuples where the i^{th} tuple in the returned list contains the i^{th} element from `list1` and the i^{th} element from `list2`. If the lists are not the same length, the resulting list should have the length of the shorter of the input lists.

`zip` should have the following type: `zip : 'a list -> 'b list -> ('a * 'b) list` and its behavior should match the following examples:

```
zip [] [] == []
zip [1;2;3] ["a";"b";"c"] == [(1,"a"); (2,"b"); (3,"c")]
zip [1;2] ["a";"b";"c"] == [(1,"a"); (2,"b")]
```

Note: you should not need all this space, but some of you have very big hand-writing. If you're writing a lot of code, rethink your solution.

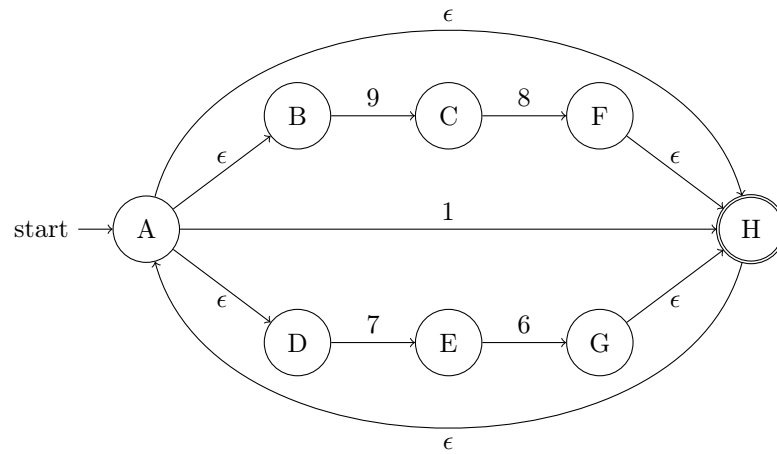
UID:

Page Intentionally Left Blank. You can use it for scratch space, but then it wouldn't be as blank as it currently is. Your call.

UID:

5. Non-deterministic Finite Automata

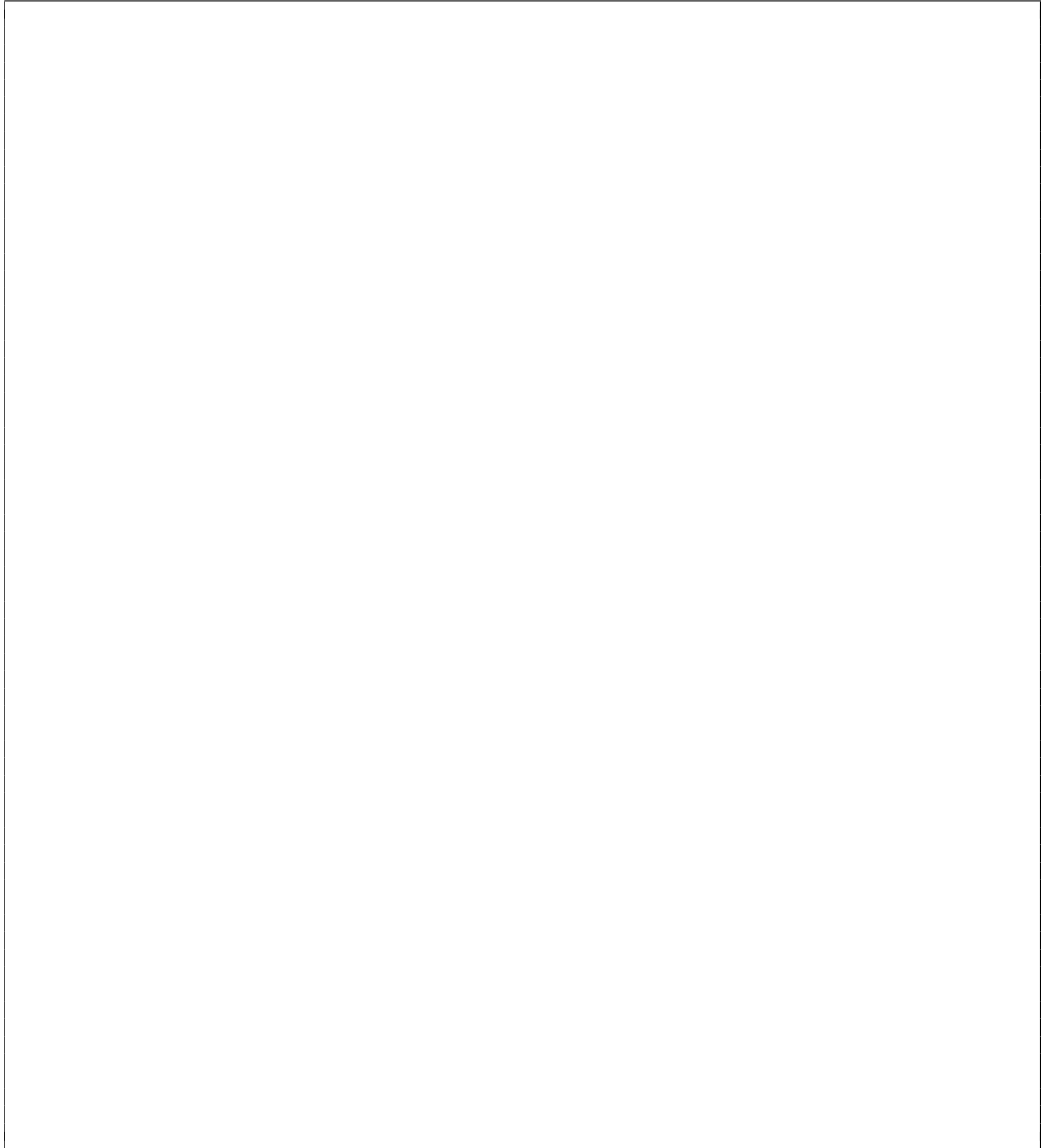
- (a) 8 points Convert the following NFA into a DFA:



Scratch space for Question 5:

UID:

Your answer here (There is scratch space on the previous page):



(b) What is an equivalent Regular Expression for the NFA in Question 5(a)?



6. Consider the following OCaml types and code:

```

type expr = Int of int
          | Var of String
          | Add of expr * expr
          | Square of expr
          | Let of String * Expr * Expr

let rec lookup env x =
  match env with
  | [] -> failwith "Variable not found in the environment"
  | (y,v)::env -> if x = y then v else lookup env x

let extend env var val = (var,val)::env

let rec eval env e =
  match e with
  | Int i -> i
  | Var v -> lookup env v
  | Square expr -> let v = eval env e in v * v
  | Add (e1,e2) -> let v1 = eval env e1 in
                    let v2 = eval env e2 in
                    v1 + v2
  | Let (v,def,body) -> let d = eval env def in
                          let env2 = extend env v d in
                          eval env2 body

```

Now consider the following operation semantics for the language we are implementing above:

$$\begin{array}{c}
 \text{num} \frac{}{A; n \rightarrow n} \quad \text{lookup} \frac{A(x) = v}{A; x \rightarrow v} \quad \text{let} \frac{A; e_1 \rightarrow v_1 \quad A, x : v_1; e_2 \rightarrow v_2}{A; \text{let } x = e_1 \text{ in } e_2 \rightarrow v_2} \\
 \\
 \text{add} \frac{A; e_1 \rightarrow v_1 \quad A; e_2 \rightarrow v_2 \quad v_3 \text{ is } v_1 + v_2}{A; e_1 + e_2 \rightarrow v_3}
 \end{array}$$

- (a) 1 point Which constructor from our `expr` is missing a corresponding rule in the operational semantics?

UID:

- (b) 4 points Write the missing rule for our operational semantics that is consistent with the behavior from our `eval` function above.

- (c) 5 points We then add a `Print` and an `If` constructor to our `expr` type, with the following definitions:

```
...  
| Print of expr  
| If of expr * expr * expr  
...
```

We add the following lines to our interpreter for `Print`:

```
| Print e -> let v = eval env e in print_int v
```

The intention is to add a conditional expression with the same sort of behavior as OCaml's `if`. Assume that we've added booleans in the usual way (like in project 4), and we want the following program to print out 84, for example:

```
if false then print 42 else print 84
```

In order to delegate this task to a colleague, I write the following rules for our Operational Semantics:

$$\text{if-true} \frac{A; e_1 \rightarrow \text{true} \quad A; e_2 \rightarrow v_2 \quad A; e_3 \rightarrow v_3}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow v_2} \quad \text{if-false} \frac{A; e_1 \rightarrow \text{false} \quad A; e_2 \rightarrow v_2 \quad A; e_3 \rightarrow v_3}{A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow v_3}$$

If our colleague implements the rules exactly as written, will we get the behavior (OCaml-like) that we want? Explain why:

UID:

7. Lambda Calculus

- (a) 4 points Consider the following lambda expression:

$$((\lambda x. (\lambda x. (\lambda x.x (\lambda x.x))))\lambda x.x)x$$

Provide a valid alpha-renaming of the above Lambda Expression where no variable name is used more than once:

8. Rust

- (a)
- 3 points
- Consider the following code:

```
let a = String::from("cm330");
{ let b = a;
  { let c = b;
    let d = &c;
  }
  //HERE
}
```

At the point in the program marked **HERE** what variable, if any, 'owns' the string created on the first line:

- (b)
- 5 points
- Code Surgery

The following code will not compile.

```
1: fn main() {
2:     let s = String::from("Hello");
3:     fun(s);
4:     println!("{}", s);
5: }
6:
7: fn fun(s: String) {
8:     println!("{}", s);
9: }
```

The intended behavior was for the program to print the string twice, once via **fun** and once in **main**. Replace a maximum of two lines in order to fix this program. Ensure that you specify which line(s) you are replacing.

Correction 1

Correction 2

UID:

General Scratch Space and Useful Information

Ruby

Helpful Array Functions:

each {|item| block} → array

each → Enumerator

Calls the given block once for each element in self, passing that element as a parameter.

Returns the array itself.

If no block is given, an Enumerator is returned.

```
a = [ "a", "b", "c" ]
a.each {|x| print x, " -- " }
produces:
a -- b -- c --
```

select {|item| block} → new_array

select → Enumerator

Returns a new array containing all elements of the array for which the given block returns a true value.

If no block is given, an Enumerator is returned instead.

```
[1,2,3,4,5].select {|num| num.even? }      #=> [2, 4]
a = %w[ a b c d e f ]
a.select {|v| v =~ /[aeiou]/ }           #=> ["a", "e"]
```

each {| key, value | block } → hash

each_pair {| key, value | block } → hash

each → an_enumerator

each_pair → an_enumerator

Calls block once for each key in the hash, passing the key-value pair as parameters.

If no block is given, an enumerator is returned instead.

```
h = { "a" => 100, "b" => 200 }
h.each {|key, value| puts "#{key} is #{value}" }

produces:
a is 100
b is 200
```

Regular Expression Documentation:

Creating a Regex:

<code>/pat/</code>	<code>/hay/ =~ "haystack"</code>
<code>%r{pat}</code>	<code>%r{hay} =~ "haystack"</code>

<code>[]</code>	range specification (e.g., <code>[a-z]</code> means a letter in the range a to z)	<code>{m,n}</code>	at least m and at most n repetitions of the preceding
<code>\w</code>	letter or digit; same as <code>[0-9A-Za-z]</code>	<code>{m,}</code>	at least m repetitions of the preceding
<code>\W</code>	neither letter or digit	<code>(a b)</code>	a or b
<code>\s</code>	space character; same as <code>[\t\n\r\f]</code>	<code>(...)</code>	grouping; capture everything enclosed
<code>\S</code>	non-space character	<code>^</code>	Start of line
<code>\d</code>	digit character; same as <code>[0-9]</code>	<code>\$</code>	End of line
<code>\D</code>	non-digit character	<code>[^abc]</code>	Any single character except: a, b, or c
<code>*</code>	zero or more repetitions of the preceding	<code>[abc]</code>	A single character of: a, b, or c
<code>+</code>	one or more repetitions of the preceding	<code>[a-z]</code>	Any single character in the range a-z
<code>?</code>	at most one repetition of the preceding; same as <code>{0,1}</code>	<code>a{3}</code>	Exactly 3 of a

Matching a Pattern:

<code>/regexp/ =~ string</code>	<code>/hay/ =~ "haystack" #=> 0</code> <code>/needle/.match('haystack') #=> nil</code>
<code>/regexp/.match(string)</code>	<code>/y/.match('haystack') #=> #<MatchData "y"></code> <code>/I(n)ves(ti)ga\2ons/.match("Investigations")</code> <code>#=> #<MatchData "Investigations" 1:"n" 2:"ti"></code>

case... when...end

case Starts a case statement definition. Take the variable you are going to work with.

when Every condition that can be matched is one when statements.

else If nothing matches then do this. Optional.

Ruby Case & Ranges

```
case capacity
when 0
  "You ran out of gas."
when 1..20
  "The tank is almost empty. Quickly, find a gas station!"
when 21..70
  "You should be ok for now."
when 71..100
  "The tank is almost full."
else
  "Error: capacity has an invalid value (#{capacity})"
end
```

UID:

NFA to DFA Algorithm:

NFA (input): $(\Sigma, Q, q_0, F_n, \sigma)$, DFA (output): $(\Sigma, R, r_0, F_d, \sigma_n)$

```
 $R \leftarrow \{ \}$   
 $r_0 \leftarrow \epsilon - \text{closure}(\sigma, q_0)$   
while  $\exists$  an unmarked state  $r \in R$  do  
  mark  $r$   
  for all  $a \in \Sigma$  do  
     $E \leftarrow \text{move}(\sigma, r, a)$   
     $e \leftarrow \epsilon - \text{closure}(\sigma, E)$   
    if  $e \notin R$  then  
       $R \leftarrow R \cup \{e\}$   
    end if  
     $\sigma_n \leftarrow \sigma_n \cup \{r, a, e\}$   
  end for  
end while  
 $F_d \leftarrow \{r \mid \exists s \in r \text{ with } s \in F_n\}$ 
```

Grammar for the Lambda Calculus:

```
e ::= v  
   | e e  
   |  $\lambda v . e$ 
```

Grammar for Regular Expressions:

```
r ::=  $\sigma$   
   |  $\epsilon$   
   | rr  
   | r | r  
   |  $r^*$ 
```

Question	Points
1	13
2	8
3	7
4	10
5	10
6	10
7	4
8	8
Total:	70