CSE 262 Final Exam

Fall 2019

**FIRST:** Please read the following carefully:

* I have not received, I have not given, nor will I give or receive, any assistance to another student taking this exam, including discussing the exam with students in another section of the course. Do not discuss the exam after you are finished until final grades are submitted.
* I will not plagiarize someone else’s work and turn it in as my own.
* I understand that acts of academic dishonesty may be penalized to the full extent allowed by the Lehigh University Code of Conduct, including receiving a **failing** grade for the course. I recognize that I am responsible for understanding the provisions of the Lehigh University Code of Conduct as they relate to this academic exercise.

If you agree with the above, write or type your name in the following space along with the date. Your exam will not be graded without this assent.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Ryan McGuiness |  | Date | 12/12/19 |

**Instructions**

* The exam is open book, open internet, open notes, open compiler.
* Use any resources you see fit but cite every source you use and include the references on the last page of the exam. Use IEEE format for citations *e.g.* [1]
* Cite links if it’s on the internet, and reference book and page numbers if you’re using published material like a book, essay, or research paper.
* There are 10 questions to the exam. Answer all 10 questions.
* You may type your answers in the document itself. If you want to include drawings that is fine. You may also print this out and write directly on the exam, but you’ll have to re-digitize the pages because submission will be on Course Site.
* All questions are weighted equally.

**Due: 12/16 @ 11:55pm on Course Site**

You must rely solely on yourself though. No asking for help on the internet or from any other student, professor, inside or outside Lehigh. For the next 10 days, you are under **STRICT** non-disclosure agreement regarding the contents of this exam. You are not to discuss this exam with anyone else. No one. **No exceptions.**

**Question 1**

Consider the following statement of valid Rust code:

let (i, j) = foo(bar)(i)?;

Here is the function signature for foo():

pub fn foo<I: Clone, O, E: ParseError<I>, F>(f: F) -> impl Fn(I) ->IResult<I, Option<O>, E>

where

F: Fn(I) -> IResult<I, O, E>,

Answer the following:

1. What type does foo() return?

foo takes in f of type F, where F is defined as a function that takes in input I and returns IResult. Therefore, foo will return IResult.

1. What is the type of bar?

bar is of type F

1. What is the return type of the following expression?

foo(bar)(i)?

Upon success the ‘?’ operator will take the IResult ,unwrap it, and return its type ; upon failure it will return from the function that its being called from.

1. What is the type of j?

j is what gets returned from the function foo after the input bar is parsed. Which will be type O.

**Question 2**

Given: Y = (λx . λy . y (x x y))( λx . λy . y (x x y))

Evaluate: Y z

* Simplify as much as possible. Show each step. Show each alpha conversion and beta reduction explicitly.
* Indicate an alpha conversion with: [a/a’]
* Indicate a beta reduction with: [x -> y]

A close up of text on a whiteboard

Description automatically generated

**Question 3**

What is tail recursion? How does a tail recursive algorithm avoid overflowing the stack? Write a recursive summation algorithm in Rust in both tail call optimized and non-optimized forms.

**150 words minimum (1/4 page)**

Tail recursion is when the function calls itself in the last statement of the function. It needs a base case so that it doesn’t call itself infinitely. Then it has its body where the data gets changed to how the user needs it to be changed. After that, in its return statement the function calls itself. This process keeps repeating until the base case is reached. Once the base case is reached it returns the last expected value and never reaches the recursive part of the function. Then the function goes all the back and provides the user with the information desired. Tail recursion prevents overflow of the stack, because every function call does not go onto the stack. Then when it is done it does not need to pop from the stack. In a recursive function all the types are the same, which allows for optimization of the stack. This happens because they can recursively use the same row in the stack. [4]

**Recursive**

fn sum(mut v: Vec<u32>) -> u32 {

let last = v.pop().unwrap();

if v.len() == 0{

return last;

}

return last + sum(v);

}

**Tail Recursive**

fn sum(v: Vec<u32>) -> u32 {

return sum\_tail(v.pop().unwrap(), v );

}

fn sum\_tail(r: u32, v: Vec<u32>) -> u32 {

if v.len()==0{

return r;

}

return (sum\_tail(v.pop().unwrap(), v) + r);

}

**Question 4**

For this question, I’d like to test one of the central purposes of the course, which was to make you more comfortable in approaching new programming languages and familiarizing yourself with them. So, I want you to learn something new:

* Choose a langue with which you aren’t familiar. I might suggest any of the languages on Course Site under “Other Languages”.
* Find an implementation of the quicksort algorithm in that language. A Google search along the lines of “[Language Name] + Quicksort” will likely lead you to Stack Overflow or some other solution. Reproduce it below (be sure to include a citation). If you can’t find a quicksort implementation, you can look for any of the other classic CSE017 sorting or searching algorithms. You’re bound to find something. If not, choose another language, the one you picked is too obscure!
* For each line of code, give me your best interpretation of what it’s doing. It doesn’t matter if you’re 100% correct. I’m more interested in your analysis of what the code is doing relative to the algorithm it’s purported to implement (you know how quicksort works, right?). Some implementations are shorter than others, by the way. You might recall which class of languages tend to produce terser source code.

LISP [1]

(defun quickSort (arr low high) // Quick sort that takes an array, a low value, and a high value

(if (< low high) // if low is less than high

(let((pair(partition arr low high))) //define pair as two numb

(quicksort arr low (car pair)) //quicksort the left side of list (car deconstructs a list and takes first pointer)

(quicksort arr (cdr pair) high)) //quicksort the right side of list (cdr takes the pointer of second pointer)

))

(defun partition (arr low high) // Partition is a function that take an arr and the low and high values

(let((piv (aref arr low))(cur low)) // this defines the pivot and the starting value as low

(loop while (<= low high) do // do while loop for when low is less than high

(cond

((< pivot (aref arr low)) // if pivot is less than arr[low] then->

(swap arr low high)(decf end)) // swap arr[low]. With arr[high]

((> pivot (aref arr low)) // if pivot is greater than arr[low] then->

(swap arr cur low)(incf cur)(incf low)) // swap pivot with arr[low]

(t // else increment the low pointer

(incf low))))

(cons (decf cur) low))) //cons creates a list of low and current val and reutns it to quick sort

(defun swap (arr i j) // swap function that takes two values and swaps them

(let ((temp (aref arr i))) // define the temp variable as the reference to the first given int

(setf(aref arr i)(aref arr j))) // setf replaces i with j

(setf (aref arr j) temp))) // then replaces j with temp variable

**Question 5**

Compare and contrast lazy and strict evaluation schemes. Use Haskell and Rust code to demonstrate examples of each. What are the advantages of lazy evaluation? What are the downsides?

**150 words minimum (1/4 page)**

Lazy evaluation is when the value of variable is retrieved from the computer upon query for it. This varies from strict evaluation schemes, because strict evaluation solves every expression even when the function is not called explicitly. Therefore, when you run a program, with lazy evaluation the functions that are not called do not get computed, and in strict evaluation all functions are computed even when not called explicitly. Strict evaluations have longer run times but are better for testing code, because if there are bugs in your code strict evaluation will run into those bugs. Lazy evaluation helps save speed when dealing with large amounts of inputs but makes it harder to test code. [2]

A simple example of lazy evaluation is the use of an “if-statement”. Here’s an example in rust and racket:

Haskell: assuming inc() increments a function by 1,and dec() decrements a function by 1

if (x>0) then inc(x) else dec(x)

In rust the ‘.collect’ function is eager, because it iterates through everything and makes a collection. An example of it being used would be:

let chars = [‘a’, ‘b’, ‘c’];

let str : String = chars.iter().map(|&x| x as u8).collect();

**Question 6**

Consider the following valid list comprehension in Haskell:

product [x^2 | x<-[1..100]]

Answer the following:

1. Explain in English what this list comprehension computes.

Take every number 1 through 100 to the power of 2 and multiply them all together.

1. Write a program in Rust that would produce the same result. Do you foresee any problem with evaluating this program?

let mut total = 1;

for i in 1..100{

total = (i) \*(i) \* total;

}

println!(“{}”, total);

This code should produce the same result, because it squares each integer 1..100 and multiplies them by each other, however when I try to test my code in Rust playground it gives me a stack over flow error.

1. Evaluate the list comprehension in GHCI. What is the result?

8709782489089480079416590161944485865569720643940840134215932536243379996346583325877967096332754920644690380762219607476364289411435920190573960677507881394607489905331729758013432992987184764607375889434313483382966801515156280854162691766195737493173453603519594496000000000000000000000000000000000000000000000000

This result is a very big Integer

4. How does Haskell succeed in generating such a large number?

Haskell succeeds in generating such a large number, because Haskell has overloaded types of numbers. Therefore, the integer type is like all other numeric types. Haskell leaves an undetermined amount of space on the stack for integers, and therefore cannot overflow it. [5]

**Question 7**

What is shared mutable state? Explain why shared mutable state is so dangerous. What kind of errors are caused by the misuse of shared mutable state? What are some strategies languages use to minimize the deleterious effects of shared mutable state? Use code snippets (in any language) to motivate your answer *i.e.* use code to show either an error caused by shared mutable state or a language feature preventing its misuse.

**150 words (1/4 page) minimum**

Shared mutable state is when references to variables are the same. This is when a variable uses the same reference as another. Therefore, when one gets changed, so does the other. This can cause a lot of problems because variables may get changed when they are not supposed to be changed. This can cause a lot of problems when working with a team, because some people may have references to variable that are used in other parts of the code. They may change it in their code, and it will work properly, but it could change variables somewhere else where they wouldn’t want it to change. This can lead to out of bounds errors and a whole bunch of other errors, this could be avoided if the variables were immutable. Therefore, it is preferred to use immutable states, because they are much less problematic, and easier to control. Rust variables must be declared mutable and a lack of declaration would mean that the variable is immutable. Example of an error in Rust:

let a = 5;

a = 6;

This is implemented by Rust, because if you want to us mutable states, you must be aware of what you are doing. Other languages like python and java have a default mutable state and if the user is unaware, they could accidently pass pointers to other variables unknowingly, and may lead to some difficult debugging. However, if one is proficient at using mutable states it could make things easier but requires a strong understanding of how states are passed.

**Question 8**

Consider the following Prolog program:

slithers(snake).

swims(fish).

swims(shark).

crawl(lizard).

scales(snake).

scales(lizard).

scales(fish).

scales(shark).

flies(bug).

% eats(A,B) is true when A eats B

eats(shark, fish).

eats(lizard, bug).

eats(snake, lizard).

eats(fish, fish).

eats(bug, bug).

Use the above facts and rules to answer the following questions. Provide the prolog query that would resolve the answer.

* What swims?

| ?- listing(swims)

* Do lizards have scales?

| ?- scales(lizard)

* What eats fish?

|?- eats(x, fish)

* What do things with scales eat?

|?- scales(x),eats(x, y)

Add the following rules to the program:

* Reptiles are things with scales that slither or crawl

Reptiles(x) :- slithers(x),(crawls(x);slithers(x))

* Fish (capital F) are things with scales that swim and eat fish

Fish(x) :- scales(x), swims(x), eats(x, fish)

* Insects fly and are eaten by Reptiles and bugs

Insects(x) :- flies(x) , eats((Reptiles(y);bug), x)

**Question 9**

Summarize the main points of the two essays “No Silver Bullet” by Fred Brooks and “Out of the Tar Pit” by Ben Mosley and Peter Marks. Be sure to reference passages from the essays. As your summaries should show, the two essays are a bit at odds with one another. Both essays make good points and it’s hard to know who is correct. Take a position as which side you think is more persuasive and explain why, again making specific reference to passages in the essay.

At least 3 citations from each essay are required. Don’t forget to add references in the works cited on the last page of the exam.

**350 words (1/2 page) minimum**

In both essays, they recognize that software engineering includes essentials and accidentals. In “No Silver Bullet” Brooks says that previous simplifications of software focused on making the accidental tasks easier to eliminate or get around. Brook’s says that no matter how much we shrink these accidental tasks down, there will be no significant improvement in software. He says this because developments to minimize these accidents are happening in a linear fashion, but as the size of software projects scale up “elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly.”(Silver Bullet pg.3) Brooks thinks that the only way to make software easier is to minimize the essential tasks. He says the main problems of software are caused by: complexity, conformity, changeability, and invisibility. As these projects get larger the data added to them connects into each other in a different way each time. Brooks says, “Much of the complexity he must master is arbitrary complexity, forced without rhyme or reason by many human institutions and systems to which his interfaces must confirm.” (Silver Bullet pg.4) Therefore, the more sporadic information that comes in the more complexity grows exponentially. [6]

In rebuttal the essay “Out of the Tar Pit”, says that Brook’s is wrong about how focusing on minimizing essential complexity will solve the problem. Mosely and Marks say that accidental complexity is significantly larger than essential complexity, and the way to minimize complexity the most is to solve the larger problem. They say that mutable state is the cause of most problems. Since most problems are accidentals, “this gives us hope that it may be possible to significantly reduce the complexity of real large systems.” (Tarpit pg.27) They go on to say that, even in an ideal world, there is a need for some complexities. Therefore, the way to get rid of these complexities is to, “avoid it where possible, and to separate it where not.” (Tarpit pg. 64) Which requires an advanced knowledge of the program being developed, before even starting. As projects get more intricate then this may become less realistic of an approach. [7]

Although I agree with Mosley and Mark(M&M) that the best way to make significant progress is to focus on the larger problem at hand, I believe the Brook’s argument has a long-term vision that is more likely to succeed. Mosely and Mark said it themselves that accidental complexities are sometimes necessary to make a program function. This plays in favor to Brook’s theory that says as a data set gets larger, accidental complexity grows exponentially. M&M say that the only thing Brook’s was right about was complexity, and the other three arguments were just different forms of complexity. They were right that they were just different forms of complexity, but I think those are the only forms that could be greatly reduced, unlike accidental complexity. Brooks explains, “That conceptual structures we construct today are too complicated to be accurately specified in advance, and too complex to be built faultlessly, then we must take a radically different approach.”(Silver Bullet pg.14) The approach that he goes on to further explain is to grow a program, instead of building one. I find this to be a better solution, because projects that are being built today require teams, rather than individuals. This further greatens the complexity gap with adding communication errors. If we could simplify the way programs are developed through simplifying essential complexities, this is where we will see greatest potential for a silver bullet. Both essays agree that simplicity is the key to getting rid of complexity, but M&M, “believe the focus should be on avoiding state, avoiding explicit control where possible, and striving at all costs to get rid of code.” (pg.64) Brooks thinks it’s not about simplifying code; the most growth can be seen by changing the way we think about writing code and changing the process of writing code as a whole. Having an easier flow of logic is more important than have less words on the screen. I believe this to be true, because with less words there is less explanation how things work, and therefore can make it more difficult for groups to work on these larger projects.

**Question 10**

Remember back to the first reading I assigned. It was a recording of a talk titled: “A human view of programming languages” by Amy J. Ko. You can go back and watch it again, but she summarizes her main points as follows:

*The mathematical view of programming languages is powerful and productive.*

*However, that view is also narrow, limiting the research questions that we ask.*

*Moreover, it limits who participates in computing, because the dominant culture of computing only projects interest in the mathematical view.*

*If embracing other views is important for discovery and equity, what other views exist and how can we explore them?*

*These views including PL as power, interface, design, notation, language, communication, but also other surprising lenses such as glue, legalese, infrastructure, and even a path out of poverty.*

*These views, however, also embody values, meaning that by investigating these metaphors for programming languages, we also embrace new values.*

Now fast forward to the end of the semester when we started talking about the future of programming languages. We spent some time talking about how programming languages in the future will be more declarative in the style of functional or logic languages for reasons of complexity. But we did not examine potential future programming languages from other perspectives, like the ones laid out by the author above.

For this question, I want you to explain your vision of the future of programming languages in the next 10 years. Frame your vision in the context of one of the twelve views of programming languages presented in the video essay (indicate which one you chose). As you answer this question, you might want to think about the problems you face every day working with machines. What makes that experience frustrating? What would make it more enjoyable? What do you think other people who are not programmers currently misunderstand about programming? What would make programming easier for non programmers? A new interface? More expressibility? Natural language recognition? The ability to make inferences with artificial intelligence? Something else?

There’s no correct answer to this question (that doesn’t mean you automatically get 100%!), but your response should be well-reasoned and persuasive.

**350 words (1/2 page) minimum**

I predict that the future of programming languages should focus on the programming interface. Interfaces have to be useable, and the current problem with computer languages is the perceived complexity of the languages. This is good for current computer scientist, as it helps our reputation as people with superior intellect but is detrimental for the future growth and innovation of programming. What scares people most is a complete program written out. They think that is read like a book, top to bottom. They do not realize instead it is a flow of logic that takes you through several different pitstops and changes dozens of variables as it goes along.

The interfaces of programming languages will change completely, by making it more user friendly. I think this will be done through better visuals in the language, I think representing a language in a 3D model would help beginners not only understand how to write a language but will help them understand the logical flow when running a program. Representing things as 3D-objects instead using vague unfamiliar terms like foo, vector, static, etc. will make people more willing to dip their toe in the ocean of programming. There have been advancements to this already, but I believe that this visualization aspect should be a part of all the big-name languages that developers use daily. The 3D- representation could use shapes, colors, and pointers to show classes and directional flow.

Another aspect will be changing up the documentation for code. As of right now it is very dense and not appealing to read or get to know. People would much rather google their problem, than figure it out using the resources provided by the language. Also, the examples of code are either too vague or complex for beginners to understand. I believe the documentation should keep its excessive commenting throughout the documentation. People struggle to keep every single short-cut that the documentation mentions, so when they get to the end of the documentation important concepts tend to go straight over their head. Also, these complex concepts should not build off of other complex concepts, rather there should be several examples showing a beginner version of such code with incrementing steps to show the most efficient way of doing things.

Another part of the programming interface that will improve would be error messages and compiling messages. When a beginner uses a terminal for the first time, they are frightened by the fantically scrolling screen. Rather those messages should be provided in a drop-down table for the people that actually need those messages and keep the output and errors very presentable. Visual Studio Code has done a great job developing error messages with quick fixes that helps get the accidental bugs out code. However, there are more logical errors that are not as easy to fix, that should be better annotated.

As the interface becomes more user friendly, this will invite more people to program, and as more people join more ideas will be circulated. Therefore, more new bright ideas will be presented, or more problems with these languages will be highlighted. This easy to follow logic will also make it easier for experienced programmers to develop even more complex programs and push the limits of programming. My future prediction is that interfaces will become easy enough for them to start teaching concepts in middle school as a fundamental class, like English and math. Instead of having kids placing shaped blocks into like sized holes, they will be placing pre-programmed classes, represented by blocks, and linking them to create unique and fun patterns. Basically, programming will become easy enough to write and debug that a child will be able to make simple programs.

**Works Cited**

Add references for the works you’ve cited in your exam below:

[1] https://stackoverflow.com/questions/19082032/quicksort-in-lisp

[2] http://homepages.inf.ed.ac.uk/wadler/topics/call-by-need.html#lazyvsstrict

[3] https://docs.rs

[4] https://medium.com/@vijeshsalian/recursion-how-to-overflow-the-stack-and-how-not-to-b9dcffdfab27

[5] https://wiki.haskell.org/Haskell

[6] http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf

[7] http://curtclifton.net/papers/MoseleyMarks06a.pdf

Good luck and have a safe and restful winter break!

Thank you, I hope you have a great break as well!