

# CPSC-354 Report

Jackson Goldberg  
Chapman University

December 11, 2022

## Abstract

My Report.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Homework</b>	<b>1</b>
2.1	Week 1 . . . . .	1
2.2	Week 2 . . . . .	2
2.3	Week 3 . . . . .	3
2.4	Week 4 . . . . .	3
2.5	Week 5 . . . . .	3
2.6	Week 6 . . . . .	7
2.7	Week 7 . . . . .	7
2.8	Week 8 . . . . .	7
2.9	Week 9 . . . . .	7
2.10	Week 10 . . . . .	8
2.11	Week 11 . . . . .	8
<b>3</b>	<b>Project</b>	<b>8</b>
3.1	Specification . . . . .	8
3.2	Prototype . . . . .	8
3.3	Documentation . . . . .	8
3.4	Critical Appraisal . . . . .	8
<b>4</b>	<b>Conclusions</b>	<b>9</b>

## 1 Introduction

Hi there. My name is Jackson Goldberg. I am a senior here at chapman University and I am taking programming languages. I like playing chess. Enjoy reading my report.

## 2 Homework

### 2.1 Week 1

I have added my program into this github repo. It is a simple python program that loops infinitely until a and b are equal. I ran it with python 3 and manually input 9 and 33 yielding the answer 3.

## 2.2 Week 2

This week we were tasked with creating simple recursive programs in Haskell. These are my solutions for all of the assigned functions. These functions can also be found in a Haskell file titled "Main.hs".

---

```
-- Takes a list of char and returns a list of char. This works by assigning the element at odd
positional values of a list into an empty list using the zip function which is what it
returns.
select_evens :: [[a]] -> [[a]]
select_evens xs = [x | (x,i) <- zip xs [0..], odd i]

-- Example:
select_evens ["a","b","c","d","e"] =
  [] : (select_evens ["b","c","d","e"]) =
    ["b"] : (select_evens ["c","d","e"]) =
      ["b"] : (select_evens ["d","e"]) =
        ["b","d"] : (select_evens ["e"]) =
          ["b","d"] : (select_evens []) =
            ["b","d"]
```

---

I referenced [this](#).

---

```
-- Same logic as above just assigns based on even position.
select_odds :: [[a]] -> [[a]]
select_odds xs = [x | (x,i) <- zip xs [0..], even i]

-- Example:
select_odds ["a","b","c","d","e"] =
  ["a"] : (select_odds ["b","c","d","e"]) =
    ["a"] : (select_odds ["c","d","e"]) =
      ["a","c"] : (select_odds ["d","e"]) =
        ["a","c"] : (select_odds ["e"]) =
          ["a","c","e"] : (select_odds []) =
            ["a","c","e"]
```

---

```
-- Uses the filter to function to create a list of all matching elements to the ones provided.
If the length is greater than 0 then it's a member.
member :: Int -> [Int] -> IO Bool
member x li = do
  let xs = filter(== x) li
  if length xs == 0
  then
    return (False)
  else do
    return (True)

-- Example:
member 2 [5,2,6] =
1 = filter(== 2) li
if length xs == 0
  else do
    return (True)
True
```

---

```
-- Uses ++ to concatenate lists.
append :: [Int] -> [Int] -> [Int]
append l1 l2 = l1 ++ l2
```

```
--Example:
append [1,2] [3,4,5] =
  [1,2] ++ [3,4,5] =
  [1,2,3,4,5]
```

---

```
-- Recursively assigns element to back of new list, creating a reverse list.
revert :: [Int] -> [Int]
revert [] = []
revert (x:xs) = revert xs ++ [x]
```

```
--Example:
revert [1,2,3] =
  [3]: revert[1,2] =
  [3,2]: revert[1] =
  [3,2,1]
```

---

```
-- Compares two strings using <=.
less_equal :: [Int] -> [Int] -> IO Bool
less_equal l1 l2 = do
  if last l1 <= last l2
  then
    return (True)
  else do
    return (False)
```

```
--Example:
less_equal [1,2,3] [2,3,2] =
  3 > 2
  False
```

---

## 2.3 Week 3

This week we were tasked with finishing the hanoi file supplied to us. The full file can be found in Hanoi.txt. Hanoi is used 31 times in the file. You can express this as formula because it doesn't matter the starting blocks since the process will always be the exact same. you will always need to reduce to the highest block so the number is nonconsequential. There are many cool visuals of this online which show what it looks like using a graphical interpretation. There is a simple break down for however many N you have.

## 2.4 Week 4

This week we worked with context free grammars and parse trees. Part one can be found [here](#), part 2 can be found [here](#).

## 2.5 Week 5

This week we are doing a lot of lambda calc stuff with abstract syntax trees.

Using the parser to generate linearized abstract syntax trees:

---

Input:

\ x . x a

[Abstract Syntax]

Prog (EApp (EAbs (Id "x") (EVar (Id "x")))) (EVar (Id "a")))

Output:

a

---

Input:

\ x . x a

[Abstract Syntax]

Prog (EAbs (Id "x") (EApp (EVar (Id "x")) (EVar (Id "a"))))

Output:

\ x . x a

---

Input:

\ x . \ y . x a b

[Abstract Syntax]

Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x")))) (EVar (Id "a"))) (EVar (Id "b")))

Output:

a

---

Input:

\ x . \ y . y a b

[Abstract Syntax]

Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "y")))) (EVar (Id "a"))) (EVar (Id "b")))

Output:

b

---

Input:

\ x . \ y . x a b c

[Abstract Syntax]

Prog (EApp (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x")))) (EVar (Id "a"))) (EVar (Id "b"))) (EVar (Id "c")))

Output:

a c

---

---

Input:  
\ $x$  . \ $y$  .  $y$   $a$   $b$   $c$

[Abstract Syntax]

```
Prog (EApp (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "y"))))) (EVar (Id "a"))) (EVar (Id "b"))) (EVar (Id "c")))
```

Output:  
 $b$   $c$

---

---

Input:  
\ $x$  . \ $y$  .  $x$   $a$  ( $b$   $c$ )

[Abstract Syntax]

```
Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x"))))) (EVar (Id "a"))) (EApp (EVar (Id "b"))) (EVar (Id "c")))
```

Output:  
 $a$

---

---

Input:  
\ $x$  . \ $y$  .  $y$   $a$  ( $b$   $c$ )

[Abstract Syntax]

```
Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "y"))))) (EVar (Id "a"))) (EApp (EVar (Id "b"))) (EVar (Id "c")))
```

Output:  
 $b$   $c$

---

---

Input:  
\ $x$  . \ $y$  .  $x$  ( $a$   $b$ )  $c$

[Abstract Syntax]

```
Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x"))))) (EApp (EVar (Id "a"))) (EVar (Id "b")))) (EVar (Id "c")))
```

Output:  
 $a$   $b$

---

---

Input:  
\ $x$  . \ $y$  .  $y$  ( $a$   $b$ )  $c$

[Abstract Syntax]

```
Prog (EApp (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "y"))))) (EApp (EVar (Id "a"))) (EVar (Id "b")))) (EVar (Id "c")))
```

---

Output:  
c

---

Input:  
 $\backslash x . \backslash y . x (a b c)$

[Abstract Syntax]

Prog (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "x"))))) (EApp (EApp (EVar (Id "a")) (EVar (Id "b")))) (EVar (Id "c"))))

Output:  
 $\backslash yx0 . a b c$

---

Input:  
 $\backslash x . \backslash y . y (a b c)$

[Abstract Syntax]

Prog (EApp (EAbs (Id "x") (EAbs (Id "y") (EVar (Id "y"))))) (EApp (EApp (EVar (Id "a")) (EVar (Id "b")))) (EVar (Id "c"))))

Output:  
 $\backslash yy0 . yy0$

---

Evaluate using pen-and-paper the following expressions:

---

$(\backslash x.M) N$

$N = \text{argument}$

$M = \text{function operation}$

$(\backslash x.x) a = a$

$\backslash x.x a = \backslash x.x a$  cannot be reduced further because there are no parentheses

$(\backslash x.\backslash y.x) a b = (\backslash x.(\backslash y.x) a) b$   
 $= (\backslash x.x) a$   
 $= a$

$(\backslash x.\backslash y.y) a b = (\backslash x.(\backslash y.y) a) b$   
 $= (\backslash y.y) b$   
 $= b$

$(\backslash x.\backslash y.x) a b c = ((\backslash x.(\backslash y.x) a) b) c$   
 $= ((\backslash y.a) b) c$   
 $= (\backslash y.a) b c$   
 $= a c$

$(\backslash x.\backslash y.y) a b c = ((\backslash x.(\backslash y.y) a) b) c$   
 $= ((\backslash y.y) b) c$   
 $= b c$

$(\backslash x.\backslash y.x) a (b c) = ((\backslash x.(\backslash y.x)) a) (b c)$   
 $= (\backslash y.a) (b c)$   
 $= a$

$(\backslash x.\backslash y.y) a (b c) = ((\backslash x.(\backslash y.y)) a) (b c)$   
 $= (\backslash y.y) (b c)$   
 $= b c$

$(\backslash x.\backslash y.x) (a b) c = ((\backslash x.(\backslash y.x)) (a b)) c$

$$\begin{aligned}
&= (\backslash y. \ a \ b) \ c \\
&= a \ b \\
(\backslash x. \backslash y. y) \ (a \ b) \ c &= ((\backslash x. (\backslash y. y)) \ (a \ b)) \ c \\
&= (\backslash y. y) \ c \\
&= c \\
(\backslash x. \backslash y. x) \ (a \ b \ c) &= (\backslash x. (\backslash y. x)) \ (a \ b \ c) \\
&= (\backslash y. \ a \ b \ c) \\
(\backslash x. \backslash y. y) \ (a \ b \ c) &= (\backslash x. (\backslash y. y)) \ (a \ b \ c) \\
&= (\backslash y. y)
\end{aligned}$$


---

All of the hand written notes can be found [Here](#).

## 2.6 Week 6

This week for homework we did lambda calc beta reduction on exponentiation. The file containing my solution can be found in week6.hs [Here](#) (Has .hs for sake of parentheses convenience).

## 2.7 Week 7

Line 5: e1 and e2 are being bound by the = and the scope is until line 7  
Line 6: i and e3 are being bound by the -λ and the scope goes until line 6  
Line 7: e3 is being bound by -λ and the scope is line 7  
Line 8: x is bound by = and the scope is line 8  
Line 18: id s id1 e1 are all bound by = and the scope is to the end of line 22  
line 20: f is bound by = and the scope goes until the end of line 22  
line 21: e2 is bound by = and the scope goes until the end of line 22  
The Rest of this weeks homework can be found [here](#).

## 2.8 Week 8

The ARS does not terminate because lines 3 and 4 are a loop.  
a is a normal form of aa and b is a normal form of bb  
No because by definition lines 3 and 4 are infinitely reducing into it's self so you would need to modify the equivalence in order for the ars to achieve normal forms. If you removed lines 3 or 4 the ARS would have normal forms.  
The normal forms provide termination, this is essentially like a base case. The two functions would be:  
 $F(aa) = a;$   
 $F(bb) = b;$

## 2.9 Week 9

1) The deadlines for my project:  
a. The first deadline November 20th, I will have a written guide on the rust language breaking down the language.  
b. The second deadline November 27th, I will have a scripted server in rust to show the backend capabilities.  
c. The third deadline December 4th, I will have both a Rust server and a C server and compare and contrast the differences.  
2) The assignment  
This week for our assignment we were asked to evaluate the ARS found in "Homework 9" on canvas. We were tasked with finding the function which corresponds with the set off rules very similar to a puzzle. One of the first things I noticed is that there were only 3 normal forms: the empty string (""), a, and b. I also noticed there were not equivalence cases, such that a can be equal to b and so on. With that in mind I set a=1, b=2, and c=3 with the empty string equal to 0 (" "=0). From this I could see that the only normal

forms were 1 and 2. Given that I knew this ARS couldn't be calculating something like addition, subtraction, or multiplication. I first attempted division. However there was not a case in which the answer could be zero with natural numbers. I eventually landed on Mod being the most likely answer since given the normal forms it is a function that given the right denominator can yield 0, 1, and 2 as the only answers. I then checked it against many examples of combinations of a, b, and c. I found that it was not consistent if the exact string was converted to a number but I did figure out that if you add the numbers together instead of directly translating them that you do end up getting only the remainders 0, 1, and 2 when you divide by 4. Therefore my answer to the puzzle is that this ARS is the ruleset for SUM of any combination (a,b,c) MOD 4.

## 2.10 Week 10

The homework can be found [here](#).

## 2.11 Week 11

My Question:

I asked this question in class but since I saw that it wasn't added to the list of questions I thought I would still post it:

"What can a DSL not do?"

Essentially I am asking what are the limitations of a DSL since they are inherently going to be more restricted than an object oriented programming language given their nature. Is this solution given those limitations something that actually provides more value than making something like a parser which can do something similar? I was really happy with this question and I thought I would share it.

My response to Meghna:

So I am not sure if I have the correct answer but I can offer maybe one line of thinking. I think that to your first point it all depends on the implementation. Contracts are tricky pieces of legally binding paper and the dynamic nature of them might prove to be challenging to cover for the implementers of the DSL. To your second question I think of the European option as a separate entity from the American option and it makes more sense than them being related for the sake of uniqueness.

My response to Andriana:

This is where I think it might speed up the process because once you have something that can be typed that has a strict set of rules the only thing that follows is memorization which I think is a good trade off for the efficiency that one might acquire by picking up these skills.

## 2.12 Week 12

What is the invariant? Indicate the reasoning steps in which you apply the rules of Hoare Logic.

while (x!=0) do z:=z\*y; x:= x-1 done

We can write out our logic like:

$x \neq 0$  while (x!=0) do z:=z\*y; x:= x-1 done

We can see the variables:

x = 100

y = 2

z = 1

We can also initialize a variable t to 0 and use it to keep track of loop execution.

With that we get:

t x y z

0 100 2 1

1 99 2 2

2 98 2 4

... ..



99 1 2 2<sup>9</sup>9  
100022<sup>1</sup>00  
 $t + x = 100 - >$   
 $t = 100 - x$   
 $z = y^t$   
*The invariant*  $z = y^{(100 - x)}$

### 3 Project

Introductory remarks ... Over the summer I had an internship and I primarily worked using rust. I didn't get to learn it very well but the little I did learn got me really interested in it. For my project I would like to do a deep dive into rust and look at the advantages it has over a similar language like C then show off a project with it.

#### 3.1 Specification

#### 3.2 Prototype

#### 3.3 Documentation

#### 3.4 Critical Appraisal

...

### 4 Conclusions

(approx 400 words)

In the conclusion, I want a critical reflection on the content of the course. Step back from the technical details. How does the course fit into the wider world of programming languages and software engineering?

### References

[PL] [Programming Languages 2022](#), Chapman University, 2022.