Programming 3
Functional Programming Challenges

Elliot Purvis
Student ID: 28561716

13th December 2017

1 Approach

My approach to exercises 1-5 largely combined pattern matching and guards, reusing functions where I could. Most of my functions involved a recursive definition for the main function, and usually some form of auxiliary function. In order to test my functions, I wrote a variety of test inputs and outputs, and tested their results by hand as well as comparing with my peers. This proved to be inefficient, but helped iron out small issues. For some, such as free variables, I wrote a small java program to generate random integers between 1 and 4, relating to each element of the `Expr` Data type, then substituted in Strings for each element. With a few tweaks, and the addition of lambda integer values by hand, this proved a reasonably effective way of generating a variety of inputs quickly, as I found writing them out by hand to be time consuming. Ive included an example of some of my tests for each exercise in appendix 1. The disadvantage of this method was that while a range of inputs were generated, edge cases were rarely tested.

An additional method I used for testing was splitting elements of my function into subfunctions, then testing these with individual use cases using GHCi. This again proved to be inefficient, but was essential in diagnosing small bugs with my code that were causing problems inside larger combinations of functions. This was particularly useful for the translate function as well as exercise substitute, in which my implementation required a lot of sub-functions. Testing allvars and nextint on individual inputs was invaluable.

When approaching the pretty printer, I began with the problem by attempting to generalize my function as much as possible, and attempted to avoid hard coding combinations of expressions in order to satisfy bracketing requirements. With regard to Exercise 4, I initially drew a flow chart of the processes the values would have to go through. I initially failed to include `HLam`, which allows for later conversion of Lam $->$ K/S types, but realised this issue reasonably quickly. This served me well, as I was able to identify the three key stages (represented in my code as `toH`, `ft`, and `out`). I was disappointed that I needed to implement another freeVariables method, but I felt it was easier than parsing output from my existing implementation.

2 Composition Law Proof

Given the law, we can translate the left hand side to equal the right.

```
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

Taking only the left hand side, and applying the definition of pure for Maybe:

```
Pure x = Just x
```

The left hand side is left as:

```
(Just (.)) <*> u <*> v <*> w
```

We know that for the Maybe type, $< * >$ is defined as:

1

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

Hence if either argument for $< * >$ is nothing, the entire expression reduces to Nothing, so we can ignore these and presume u, v and w are something. Using this, we can then apply $< * >$ (being careful to keep terms left-applicative) to the first two terms.

```
(Just (.) f) <*> (Just g) <*> (Just x)
```

Repeating this:

```
(Just (.) f g) <*> (Just x)
```

And then applying $< * >$ a final time:

```
(Just (f.g) x)
```

Which expands to:

```
(Just f (g x))
```

From here, we can apply:

```
pure f <*> pure x = pure (f x)
```

We apply the homomorphism law once, and then again to the second generated term.

```
(Just f) <*> Just (g x)
```

```
(Just f) <*> ( (Just g) <*> (Just x) )
```

Finally, we can remove the earlier definitions of u, v, w to give the right side of our original equation.

```
u <*> (v <*> w)
```

3 Appendix

3.1 Example test cases

```
-- Exercise 1
freeVariables (Lam 1(Lam 2 (App (Var 1) (App (Var 2) (Var 3))))) =>
    [3]
freeVariables (App (Var 1) (Lam 1 (App (Var 2) (App (App (Lam 3 (
    App (Var 4) (Var 3))) (Var 2)) (Var 1))))) => [1,2,4,2]
freeVariables (Var 1) => 1
freeVariables (Lam 1 (App (Var 1) (Var 1))) => []
freeVariables (Lam 1 (Var 1)) = []
freeVariables (Lam 1 (App (Var 1) (Var 2))) = [2]

-- Exercise 2
rename (Lam 1 (App (Lam 1 (Var 1)) (Var 1))) 1 3 => Lam 3 (App (Lam
    1 (Var 1)) (Var 3))
rename (Lam 1 (App (Lam 2 (App (Var 3) (Var 2))) (Var 2))) 2 5 =>
    Lam 1 (App (Lam 5 (App (Var 3) (Var 5))) (Var 5))
rename (Lam 1 (Var 2)) 2 3 => Lam 1 (Var 3)
rename (App (Lam 2 (App (Var 3) (Var 5))) (Var 5))
rename (Lam 1 (Var 1)) 1 2 = (Lam 2 (Var 2))
rename (Lam 1 (Lam 1 (Var 1))) 1 2 = (Lam 2 (Lam 1 (Var 1)))



-- Exercise 3
alphaEquivalent (Var 1) (Var 2) = False
alphaEquivalent (App (Lam 2 (App (Var 3) (Var 5))) (Var 5)) (App (
    Lam 5 (App (Var 5) (Var 3))) (Var 5)) = False
alphaEquivalent (Lam 1 (Var 2)) (Lam 2 (Var 1)) = True
alphaEquivalent (Lam 2 (Var 2)) (Lam 2 (Var 2)) = True
alphaEquivalent (Lam 2 (Var 3)) (Lam 1(Var 3)) = True
alphaEquivalent  (Lam 2 (Var 2)) (Lam 1 (Var 1)) = True
alphaEquivalent  (App (Var 1) (Lam 1 (Var 2))) (App (Var 1) (Var 2)
    ) = False
alphaEquivalent  (Lam 1 (Var 1)) (Lam 2 (Var 2)) = True


-- Exercise 4
hasRedex  (App (Var 1) (Lam 1 (Var 2))) = True
hasRedex (App (Var 1) (Var 2)) = False
hasRedex (App (Lam 1(Var 1)) (Var 2)) = True
hasRedex (App (App (App (Var 1) (Lam 1 (Var 2))) (Var 3)) (Var 4))
    = True
hasRedex (Lam 1 (Var 2)) = False
hasRedex (App (Lam 1 (Var 1)) (Var 1)) = True
hasRedex (Lam 1 (App (Var 1) (Var 2))) = False
hasRedex (App (Lam 1 (App (Var 1) (Var 2))) (Lam 3 (Var 5))) = True

-- exercise 5
substitute (App (Var 1) (Lam 5 (Var 1))) 1 (App (Var 2) (Var 3)) =
    App (App (Var 2) (Var 3)) (App (Var 2) (Var 3))
substitute (Var 1) 1 (Var 2) = (Var 2)
substitute (Lam 1 (Var 1)) 1 (Var 2) = (Lam 1 (Var 1))
```

```
substitute (Lam 2 (Var 1)) 1 (Var 2) = (Lam 3 (Var 2))


-- Section 2 (Pretty print)
prettyPrint (Lam 1 (Lam 2 (App (Var 1) (Var 3)))) = "\\x1x2->x1x3"
prettyPrint (Lam 1 (Lam 2 (Lam 3 (App (Var 1) (App (Var 2) (Var 3))
    )))) = "\\x1x2x3->x1(x2x3)"


-- Section 4 (Translate)
translate (Lam 1 (Lam 2 (Var 1))) = AppCL (AppCL S (AppCL K K)) (
    AppCL (AppCL S K) K)
translate (App (Lam 1 (Var 1)) (Var 2)) = AppCL (AppCL (AppCL S K)
    K) (VarCL 2)
```