In the name of God

PL homework #3

PanteA Habibi - 9131010

3) Use lambda calculus reduction to find a shorter expression for (λx.λy.xy)(λx.xy). Begin by renaming bound variables. You should do all possible reductions to get the shortest possible expression. What goes wrong if you do not rename bound variables?

(λx.λy.xy)(λx.xy)=(λx.λy.xy)(λz.zm)=λy.(λz.zm)y=λy.ym

if we do not rename, finally we have λy.yy but m is a free variable not a bounded variable.

4) The Algol-like program fragment

```
function f(x)
        return x+4
end;
function g(y)
        return 3-y
end;
f(g(1));
```

can be written as the following lambda expression:

((λf.λg.f(g1)) (λx.x+4)) (λy.3−y).

Reduce the expression to a normal form in two different ways, as described below.

(a) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the left as possible.

(b) Reduce the expression by choosing, at each step, the reduction that eliminates a λ as far to the right as possible.

a.

((λf.λg.f(g 1))(λx.x+4))(λy.3-y)

(λg.(λx.x+4)(g 1))(λy.3-y)

((λx.x+4)(λy.3-y 1))

(λy.3-y 1)+4

(3-1)+4

b.

((λf.λg.f(g 1))(λx.x+4))(λy.3-y)

(λg.(λx.x+4)(g 1))(λy.3-y)

(λg.(g 1)+4)(λy.3-y)

((λy.3-y) 1)+4

(3-1)+4

6) A programmer is having difficulty debugging the following C program. In theory, on an "ideal" machine with infinite memory, this program would run forever. (In practice, this program crashes because it runs out of memory, as extra space is required every time a function call is made.)

```
int f(int (*g)(. . .)){ /* g points to a function that returns an int */
        return g(g);
}
int main(){
        int x;
        x = f(f);
        printf("Value of x = %d\n", x);
        return 0;
}
```

Explain the behavior of the program by translating the definition of f into lambda calculus and then reducing the application f(f). This program assumes that the type checker does not check the types of arguments to functions.

f=λg.gg

f f = λg.gg f —>(ff—>ff—>…—>ff)(infinite)

this call will never terminate and this expression will not reduce to another and it continues forever.

8) The text describes a denotational semantics for the simple imperative language given by the grammar

P::=x:=e|P_1;P_2|if e then P_1 else P_2|while e do P.

Each program denotes a function from states to states, in which a state is a function from variables to values.

(a) Calculate the meaning C[x:=1;x:=x+1;](s_0) in approximately the same detail as that of the examples given in the text, where s_0 = λv ∈ variables. 0, giving every variable the value 0.

(b) Denotational semantics is sometimes used to justify ways of reasoning about programs. Write a few sentences, referring to your calculation in part (a), explaining why
C[x := 1; x := x + 1;](s) = C[x := 2;](s) for every state s.

a.

E[[x]](S0)=0

E[[x]](S1)=1

E[[x+1]](S1)=E[[x]](S1)+E[[1]](S1)=1+1=2

c[[x:=1;x:=x+1;]](S0)

c[[x:=x+1;]](c[[x:=1]])

c[[x:=x+1;]](modify(S0,x.E[[1]](S0)))

c[[x:=x+1;]](modify(S0,x,1))

modify(S1,x,2)

b.

in every state, variables such as x can have different value and the expression show us that at first we have x=1 and then "1" plus it.

c[[x:=x+1]](c[[x:=1]]) : c[[x:=1]] returns the state that x is equivalent to 1. So after c[[x:=x+1]](c[[x:=1]]) value of x will change to 2.

c[[x:=2](S)] : it means that in every state that we are in, value of x changes to 2, so it is equal to previous, because both of them return the state that value of x is 2 and other variables are the same as before.

because in the expression after these changes, value of x=1 increases 1 unit and the other variables are not change so we can show this change in 1 step and just modify the state.

c[[x:=1;x:=x+1;]](S)=C[[x:=2]](S)=modify(S,x,2)

9) A nonstandard denotational semantics describing initialize-before-use analysis is presented in the text.

(a) What is the meaning of
C[x := 0; y := 0; if x = y then z := 0 else w := 1](s_0)
in the state s_0 = λy ∈ variables.uninit? Show how to calculate your answer.

(b) Calculate the meaning
C[if x = y then z := y else z := w](s)
in state s with s(x) = init, s(y) = init, and s(v) = uninit or every other variable v.

a.

c[[x:=0]](S0) = if E[[0]](S0)=ok then modify(S0,x,init)

else error

S1=modify(S0,x,init)

c[[y=0]](S1) (same as above) S2=modify(S1,y,init)

c[[if x=y then z:=0 else w:=1]](S2) =

if E[[x=y]](S2)=error or c[[z:=0]](S2)=error or c[[w:=1]](S2)=error then error

else c[[z:=0]](S2) (+) c[[w:=1]](S2)=c[[z:=0]](S2) (+) c[[w:=1]](S2) =

if c[[z:=0]](S2)(v)=c[w:=1]](S2)(v)=init for every v then init else uninit

so z,w: uninit

b.

s {x=init, y=init, others=uninit}

if E[[e]](S)=error or c[[z:=y]](S)=error or c[[z:=w]]=error then error
c[[z:=w]]=if E[[w]](S)=ok then modify(S,x,init) else error
z and v do not init in if so uninit

13) Many more lines of code are written in imperative languages than in functional ones. This question asks you to speculate about reasons for this. First, however, an explanation of what the reasons are not is given:
It is not because imperative languages can express programs that are not possible in functional languages. Both classes can be made Turing complete, and indeed a denotational semantics of an imperative language can be used to translate it into a functional language.
It is not because of syntax. There is no reason why a functional language could not have a syntax similar to that of C, for example.
It is not because imperative languages are always compiled whereas functional languages are always interpreted. Basic is imperative, but is usually interpreted, whereas Haskell is functional and usually compiled.
For this problem, consider general properties of imperative and functional languages. Assume that a functional language supports higher-order functions and garbage collection, but not assignment. For the purpose of this question, an imperative language is a language that supports assignment, but not higher-order functions or garbage collection. Use only these assumptions about imperative and functional languages in your answer.

(a) Are the re inherent reasons why imperative languages are superior to functional ones for the majority of programming tasks? Why?

(b) Which variety (imperative or functional) is easier to implement on machines with limited disk and memory sizes? Why?

(c) Which variety (imperative or functional) would require bigger executables when compiled? Why?

(d) What consequence might these facts have had in the early days of computing?

(e) Are these concerns still valid today?

a. *imperative languages are easy to understand because they are imperative ! and they have more similarity to machine language. maybe for parallel processes, functional languages are better.*

b. compiles of functional languages codes are easier than imperative and they have fewer overhead.

c. functionals, because it is possible that in the middle of their compiles we have code generation and they are more complex than imperative languages.

d. they used functional languages less than others.

e. because the memory limit does not a problem nowadays.

14) It can be difficult to write programs that run on several processors concurrently because a task must be decomposed into independent subtasks and the results of subtasks often must be combined at certain points in the computation. Over the past 20 years, many researchers have tried to develop programming languages that would make it easier to write concurrent programs. In his Turing Lecture, Backus advocated functional programming because he believed functional programs would be easier to reason about and because he believed that functional programs could be executed efficiently in parallel.
Explain why functional programming languages do not provide a complete solution to the problem of writing programs that can be executed efficiently in parallel. Include two specific reasons in your answer.

One of the problems is that when two threads want to init in a critical section simultaneously. and sometimes it is possible that we have common states. another problem is that the output of them are not optimum as much as other languages and it is because of incoherence between functional languages and machine languages that are imperative.

5.2.2) Find another way to define the successor function on Church numerals.

λn. λs. λz.(n s (s z))

5.2.4) Define a term for raising one number to the power of another.

λm.λn (m (Times n) c1)

is equivalent to n^m

5.2.8) A list can be represented in the lambda- calculus by its fold function. (OCaml's name for this function is fold_left; it is also sometimes called reduce .) For example, the list [x,y,z] becomes a function that takes two arguments c and n and returns c x (c y (c z n))). What would the representation of nil be? Write a function cons that takes an element h and a list (that is, a fold function) t and returns a similar representation of the list formed by prepending h to t. Write isnil and head functions, each taking a list parameter. Finally, write a tail function for this representation of lists (this is quite a bit harder and requires a trick analogous to the one used to define prd for numbers).

nil := λx.tru equivalent to pair tru tru

isnil := fst

head := λz.fst(scd z)

tail:= λz.scd(scd z)

cons := λh.λt.pair fls(pair h t)

nill = λc.λn.n

cons = λh.λt.λc.λn.c h (t c n)

head = λg.g(λt.λf.t) (any expression)

for example:

head = λg.g(λt.λf.t)(fls)

isnil = λg.g(λa.λb.fls)tru

tail = λg.fst(g(λh.λf.pair(snd f)(cons h(snd f))))(pair)

5.2.11) Use fix and the encoding of lists from Exercise 5.2.8 to write a function that sums lists of Church numerals.

g = λf.λl.ifisnill(l)then C0 else plus f(tail(l)) head(l)

function = yg

y = λf.(λx.f(xx)(λxx.f(xx))) call by name fixed point combinator

5.3.6) Adapt these rules to describe the other three strategies for evaluation—full beta-reduction, normal-order, and lazy evaluation.

full beta reduction :

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}$$

$$\frac{t_2 \to t_2'}{tt_2 \to tt_2'}$$

$$(\lambda x.\, t)t_1 \to [x \to t_1]t$$

normal form :

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}$$

$$\frac{t_2 \to t_2'}{t t_2 \to t t_2'}$$

$$(\lambda x.t) t_1 \to [x \to t_1] t$$

$$\lambda x.t \to \lambda x.t'$$

Lazy:

$$\frac{t_1 \to t_1'}{t_1 t_2 \to t_1' t_2}$$

$$(\lambda x.t) t_1 \to [x \to t_1] t$$