Homework 4

Task 1

1. What is the output for test “if succ(0) then true else (if true the succ(2) else succ(3));” in untyped implementation? Why? Why term in brackets won’t reduce?

```
ubuntu@ubuntu2004:~/Documents/plt/fulluntyped$ cat a.txt
if succ(0) then true else (if true then succ(2) else succ(3));
ubuntu@ubuntu2004:~/Documents/plt/fulluntyped$ ./f a.txt
(if 1 then true else if true then 3 else 4)
```

   It will be “if 1 then true else (if true the succ(2) else succ(3));” because by the rules the guard is computed first, and then there are no rules when the guard is not “true” or “false”

2. Perform same for typed implementation and explain result

```
ubuntu@ubuntu2004:~/Documents/plt/fullsimple$ cat a.txt
if succ(0) then true else (if true then succ(2) else succ(3));
ubuntu@ubuntu2004:~/Documents/plt/fullsimple$ ./f a.txt
/home/ubuntu/Documents/plt/fullsimple/a.txt:1.0:
guard of conditional not a boolean
```

3. What evaluation strategy is implemented? Reason it, show how you understood it.
   Typed lambda calculus uses the same main rules from the previous chapters, thus it uses the same evaluation strategy which is *call-by-value.*

4. eval function is implemented as repeatedly calling eval1 of term. How it knows when it reached the normal form (value or stuck state)?
   eval1 works in such a way that it raises exception *NoRuleApplies* when the term is in normal form. This way *eval* knows when *eval1* has finished. The left screenshot shows eval function, the right one shows the end of the eval1 function

```
let rec eval ctx t =
  try let t' = eval1 ctx t
      in eval ctx t'
  with NoRuleApplies -> t
```

```
    let t1' = eval1 ctx t1 in
    TmLet(fi, x, t1', t2)
| _ ->
    raise NoRuleApplies
```

5. Exc 8.3.5
   Technically, we cand do this, but we should not, because it will cause *leq* and *eq,* which relies on *leq*, functions to break

```
minus = lambda a. lambda b. b predok a;

is0 = lambda n. n (lambda x. fls) tru;
leq = lambda a. lambda b. is0 (minus a b);
isequal = lambda a. lambda b.c_to_bool (and (leq a b) (leq b a));
```

6. Exc 8.3.6.
   Expansion property does not hold
   “If true then 1 else false” – this statement is completely fine and will be reduced to *1*, but the *if term* is not of any type, however *1* is of *Nat* type.

Task 2

I have added TyWrong that represents the wrong type and accepts string which is an error message, then I changed how errors are created in typeof function. Since there are no numbers, succ, pred, iszero, there is not much that can be tested. By the wat, it can be implemented in fullsimple in a similar way.

```
[jiklopo@linux-pc hw4]$ cat test.f
/* Examples for testing */
if lambda x:Bool.x then true else false;
if true then lambda x:Bool.x else false;
[jiklopo@linux-pc hw4]$ make test
./f test.f
(if lambda x:Bool.x then true else false)
  : Wrong: guard of conditional not a boolean
(lambda x:Bool. x)
  : Wrong: arms of conditional have different types
[jiklopo@linux-pc hw4]$
```

What I have changed:

Syntax.ml:

```
 8  type ty =
 9      TyArr of ty * ty
10    | TyBool
11    | TyWrong of string
```

```
168  and printty_AType outer tyT = match tyT with
169      TyBool -> pr "Bool"
170    | TyWrong(s) -> pr "Wrong: "; print_string s
171    | tyT -> pr "("; printty_Type outer tyT; pr ")"
```

Syntax.mli:

```
 7  type ty =
 8      TyArr of ty * ty
 9    | TyBool
10    | TyWrong of string
```

Core.ml:

```
53              if (=) tyT2 tyT11 then tyT12
54              else TyWrong("parameter type mismatch")
55          | _ -> TyWrong("arrow type expected"))
56    | TmTrue(fi) ->
57        TyBool
58    | TmFalse(fi) ->
59        TyBool
60    | TmIf(fi,t1,t2,t3) ->
61      if (=) (typeof ctx t1) TyBool then
62        let tyT2 = typeof ctx t2 in
63        if (=) tyT2 (typeof ctx t3) then tyT2
64        else TyWrong("arms of conditional have different types")
65      else TyWrong("guard of conditional not a boolean")
```