Foundations 1: Assignment 2019

Andrea Pavan

Throughout the assignment, assume the terms and definitions given in the DATA SHEET.

1. Just like I defined translation functions T: M’ → M’’ and ω : M → Λ, give translation functions from U : M → M’’ and V : M → M’ and ω’ : M’ → Λ’. Your translation functions need to be complete with all sub functions and needed information (just like T and ω were complete with all needed information). Submit all these functions here:

- *U : M → M”:*

U(v) = v *U(λv.A) = f(v,U(A))* *U(PQ) = (U(P)U(Q))*

where f takes a variable and a combinator-term and returns a combinator term according to the following numbered clauses:
1. f(v, v) = I”
2. f(v,P”) = K” P” if v ∉ FV (P”)
3. $f(v, P''_1 P''_2) = \begin{cases} P''_1 & \text{if } v \notin FV(P''_1) \text{ and } P''_2 \equiv v \\ S''\, f(v, P''_1)\, f(v, P''_2) & \text{otherwise.} \end{cases}$

- *V : M → M’ :*

V(v) = v *V(λv.M) = [v] V(M)* *V(QP) = <V(P)> V(Q)*

- *ω'[x₁,⋯,xₙ] : M’ → Λ’*

For $[x_1 \dots x_n]$ *a list (not a set) of variables, we define* $\omega'[x_1 \dots x_n] : M' \to \Lambda'$ *inductively by:*
$\omega[x_1 \dots x_n](v) = \min\{j : v \equiv x_j\}$
$\omega[x_1 \dots x_n](<P>Q) = \omega[x_1 \dots x_n](P)\ \omega[x_1 \dots x_n](Q)$
$\omega[x_1 \dots x_n]([x]A) = \lambda\omega[x, x_1 \dots x_n](A)$

2. For each of the SML terms vx, vy, vz, t1… t9 translate them into the corresponding terms of M, M', M", Λ and Λ' using the translation functions V , U, ω and ω'.

	M	V
vx	x	x
vy	y	y
vz	z	z
t1	λx.x	[x]x
t2	λy.x	[y]x
t3	(((λx.x) (λy.x)) z)	<z><[y]x>[x]x
t4	((λx.x) z)	<z>[x]x
t5	((((λx.x) (λy.x)) z) (((λx.x) (λy.x)) z))	<<z><[y]x>[x]x><z><[y]x>[x]x
t6	(λx.(λy.(λz.((x z) (y z)))))	[x][y][z]<<z>y><z>x
t7	(((λx.(λy.(λz.((x z) (y z))))) (λx.x)) (λx.x))	<[x]x><[x]x>[x][y][z]<<z>y><z>x
t8	(λz.(z ((λx.x) z)))	[z]<<z>[x]x>z
t9	((λz.(z ((λx.x) z))) (((λx.x) (λy.x)) z))	<<z><[y]x>[x]x>[z]<<z>[x]x>z

	U	ω	ω'
vx	x	1	1
vy	y	2	2
vz	z	3	3
t1	I	λ1	[]1
t2	Kx	λ2	[]2
t3	I(Kx)z	(λ1) (λ2) 3	<3><[]2>[]1
t4	Iz	(λ1) 3	<3>[]1
t5	(I(Kx)z) (I(Kx)z)	((λ1) (λ2) 3) ((λ1) (λ2) 3)	<<3><[]2>[]1><3><[]2>[]1
t6	S	λ λ λ 3 1 (2 1)	[][][]<<1>2><1>3
t7	SII	(λ λ λ 3 1 (2 1)) (λ1) (λ1)	<[]1><[]1>[][][]<<1>2><1>3
t8	SII	λ1((λ1) 1)	[]<<1>[]1>1
t9	SII(I(Kx)z)	(λ1((λ1) 1)) ((λ1) (λ2) 3)	<<3><[]2>[]1>[]<<1>[]1>1

3. Just like I introduced SML terms vx, vy, vz, t1, t2, … t9 which implement terms in M, please implement the corresponding terms each of the other sets M', Λ, Λ', M''.

- M

```
val vx = (ID "x");
val vy = (ID "y");
val vz = (ID "z");
val t1 = (LAM("x",vx));
val t2 = (LAM("y",vx));
val t3 = (APP(APP(t1,t2),vz));
val t4 = (APP(t1,vz));
val t5 = (APP(t3,t3));
val t6 = (LAM("x",(LAM("y",(LAM("z",(APP(APP(vx,vz),(APP(vy,vz))))))))));
val t7 = (APP(APP(t6,t1),t1));
val t8 = (LAM("z", (APP(vz,(APP(t1,vz))))));
val t9 = (APP(t8,t3));
```

- M'

```
val ivx = IID "x";
val ivy = IID "y";
val ivz = IID "z";
val it1 = ILAM ("x",ivx);
val it2 = ILAM ("y",ivx);
val it3 = IAPP (ivz, IAPP(it2, it1));
val it4 = IAPP (ivz,it1);
val it5 = IAPP (it3,it3);
val it6 = ILAM("x",(ILAM("y",(ILAM("z",(IAPP((IAPP(ivz,ivy),IAPP(ivz,ivx)))))))));
val it7 = IAPP(it1,IAPP(it1,it6));
val it8 = ILAM ("z", (IAPP(IAPP(ivz,it1),ivz)));
val it9 = IAPP(it3,it8);
```

- M''

```
val cvx = (CID "x");
val cvy = (CID "y");
val cvz = (CID "z");
val ct1 = (CI);
val ct2 = (CAPP (CK,cvx));
val ct3 = (CAPP (CAPP (ct1,ct2),cvz));
val ct4 = (CAPP (ct1,cvz));
val ct5 = (CAPP(ct3,ct3));
val ct6 = (CS);
val ct7 = (CAPP (CAPP(ct6,ct1), ct1));
val ct8 = (CAPP((CAPP(CS,CI)),CI));
val ct9 = (CAPP (ct8,ct3));
```

- Λ

```
val bv1 = (BID 1);
val bv2 = (BID 2);
val bv3 = (BID 3);
val bt1 = (BLAM bv1);
val bt2 = (BLAM bv2);
val bt3 = (BAPP(BAPP(bt1,bt2),bv3));
val bt4 = (BAPP(bt1,bv3));
val bt5 = (BAPP(bt3,bt3));
val bt6 = (BLAM((BLAM((BLAM((BAPP(BAPP(bv3,bv1),(BAPP(bv2,bv1))))))))));
val bt7 = (BAPP(BAPP(bt6,bt1),bt1));
val bt8 = (BLAM(BAPP(bv1,BAPP(bt1,bv1))));
val bt9 = (BAPP(bt8,bt3));
```

- Λ’

```
val ibv1 = (IBID 1);
val ibv2 = (IBID 2);
val ibv3 = (IBID 3);
val ibt1 = (IBLAM ibv1);
val ibt2 = (IBLAM ibv2);
val ibt3 = (IBAPP(ibv3, IBAPP(ibt2, ibt1)));
val ibt4 = (IBAPP(ibv3,ibt1));
val ibt5 = (IBAPP(ibt3,ibt3));
val ibt6 = (IBLAM((IBLAM((IBLAM((IBAPP(IBAPP(ibv1,ibv2),(IBAPP(ibv1,ibv3))))))))));
val ibt7 = (IBAPP(ibt1, IBAPP(ibt1,ibt6)));
val ibt8 = (IBLAM((IBAPP((IBAPP(ibv1,ibt1)),ibv1))));
val ibt9 = (IBAPP(ibt3,ibt8));
```

4. For each of M’, Λ, Λ’, M’’, implement a printing function that prints its elements nicely and you need to test it on every one of the corresponding terms vx, vy, vz, t1, t2, … t9.

- (*Prints a term in M notation*)

```
fun printLEXP (ID v) =
    print v
  | printLEXP (LAM (v,e)) =
    (print "(λ";
     print v;
     print ".";
     printLEXP e;
     print ")")
  | printLEXP (APP(e1,e2)) =
    (print "(";
     printLEXP e1;
     print " ";
     printLEXP e2;
     print ")");
```

Printing these M terms yields:

```
- printLEXP(vx);
xval it = () : unit
- printLEXP(vy);
yval it = () : unit
- printLEXP(vz);
zval it = () : unit
- printLEXP(t1);
(λx.x)val it = () : unit
- printLEXP(t2);
(λy.x)val it = () : unit
- printLEXP(t3);
(((λx.x) (λy.x)) z)val it = () : unit
- printLEXP(t4);
((λx.x) z)val it = () : unit
- printLEXP(t5);
((((λx.x) (λy.x)) z) (((λx.x) (λy.x)) z))val it = () : unit
- printLEXP(t6);
(λx.(λy.(λz.((x z) (y z)))))val it = () : unit
- printLEXP(t7);
(((λx.(λy.(λz.((x z) (y z))))) (λx.x)) (λx.x))val it = () : unit
- printLEXP(t8);
(λz.(z ((λx.x) z)))val it = () : unit
- printLEXP(t9);
((λz.(z ((λx.x) z))) (((λx.x) (λy.x)) z))val it = () : unit
```

- (* prints a term in M' notation *)

```
fun printIEXP (IID v) =
        print v
 | printIEXP (ILAM (v, e)) =
 ( print "[";
        print v;
        print "]";
        printIEXP e )
 | printIEXP (IAPP(e1,e2)) =
 ( print "<";
        printIEXP e1;
        print ">";
        printIEXP e2 );
```

Printing these M’ terms yields:

```
- printIEXP(ivx);
xval it = () : unit
- printIEXP(ivy);
yval it = () : unit
- printIEXP(ivz);
zval it = () : unit
- printIEXP(it1);
[x]xval it = () : unit
- printIEXP(it2);
[y]xval it = () : unit
- printIEXP(it3);
<z><[y]x>[x]xval it = () : unit
- printIEXP(it4);
<z>[x]xval it = () : unit
- printIEXP(it5);
<<z><[y]x>[x]x><z><[y]x>[x]xval it = () : unit
- printIEXP(it6);
[x][y][z]<<z>y><z>xval it = () : unit
- printIEXP(it7);
<[x]x><[x]x>[x][y][z]<<z>y><z>xval it = () : unit
- printIEXP(it8);
[z]<<z>[x]x>zval it = () : unit
- printIEXP(it9);
<<z><[y]x>[x]x>[z]<<z>[x]x>zval it = () : unit
```

- (* prints a term in Λ notation *)

```
fun printBEXP (BID n) =
        print (Int.toString n)
  | printBEXP (BLAM(e)) =
  ( print "(";
        print "λ";
        printBEXP e;
        print ")" )
  | printBEXP (BAPP(e1,e2)) =
  ( print "(";
        printBEXP e1;
        print " ";
        printBEXP e2;
        print ")" );
```

Printing these Λ terms yields:

```
- printBEXP(bv1);
1val it = () : unit
- printBEXP(bv2);
2val it = () : unit
- printBEXP(bv3);
3val it = () : unit
- printBEXP(bt1);
(λ1)val it = () : unit
- printBEXP(bt2);
(λ2)val it = () : unit
- printBEXP(bt3);
(((λ1) (λ2)) 3)val it = () : unit
- printBEXP(bt4);
((λ1) 3)val it = () : unit
- printBEXP(bt5);
((((λ1) (λ2)) 3) (((λ1) (λ2)) 3))val it = () : unit
- printBEXP(bt6);
(λ(λ(λ((3 1) (2 1)))))val it = () : unit
- printBEXP(bt7);
(((λ(λ(λ((3 1) (2 1))))) (λ1)) (λ1))val it = () : unit
- printBEXP(bt8);
(λ(1 ((λ1) 1)))val it = () : unit
- printBEXP(bt9);
((λ(1 ((λ1) 1))) (((λ1) (λ2)) 3))val it = () : unit
```

- (* prints a term in Λ' notation *)

```
fun printIBEXP (IBID n) =
        print (Int.toString n)
  | printIBEXP (IBLAM(e)) =
  ( print "[]";
        printIBEXP e )
  | printIBEXP (IBAPP(e1,e2)) =
  ( print "<";
        printIBEXP e1;
        print ">";
        printIBEXP e2 );
```

Printing these terms in Λ' yields:

```
- printIBEXP(ibv1);
1val it = () : unit
- printIBEXP(ibv2);
2val it = () : unit
- printIBEXP(ibv3);
3val it = () : unit
- printIBEXP(ibt1);
[]1val it = () : unit
- printIBEXP(ibt2);
[]2val it = () : unit
- printIBEXP(ibt3);
<3><[]2>[]1val it = () : unit
- printIBEXP(ibt4);
<3>[]1val it = () : unit
- printIBEXP(ibt5);
<<3><[]2>[]1><3><[]2>[]1val it = () : unit
- printIBEXP(ibt6);
[][][]<<1>2><1>3val it = () : unit
- printIBEXP(ibt7);
<[]1><[]1>[][][]<<1>2><1>3val it = () : unit
- printIBEXP(ibt8);
[]<<1>[]1>1val it = () : unit
- printIBEXP(ibt9);
<<3><[]2>[]1>[]<<1>[]1>1val it = () : unit
```

- (* prints a term in M" notation *)

```
fun printCEXP (CID v) =
          print v
  | printCEXP (CK) =
          print "K""
  | printCEXP (CS) =
          print "S""
  | printCEXP (CI) =
          print "I""
  | printCEXP (CAPP(e1,e2)) =
  ( print "(";
          printCEXP e1;
          print " ";
          printCEXP e2;
          print ")" );
```

Printing these terms in M'' yields:

```
- printCEXP(cvx);
xval it = () : unit
- printCEXP(cvy);
yval it = () : unit
- printCEXP(cvz);
zval it = () : unit
- printCEXP(ct1);
I"val it = () : unit
- printCEXP(ct2);
(K" x)val it = () : unit
- printCEXP(ct3);
((I" (K" x)) z)val it = () : unit
- printCEXP(ct4);
(I" z)val it = () : unit
- printCEXP(ct5);
(((I" (K" x)) z) ((I" (K" x)) z))val it = () : unit
- printCEXP(ct6);
S"val it = () : unit
- printCEXP(ct7);
((S" I") I")val it = () : unit
- printCEXP(ct8);
((S" I") I")val it = () : unit
- printCEXP(ct9);
(((S" I") I") ((I" (K" x)) z))val it = () : unit
```

5. Implement in SML the translation functions T, U and V and give these implemented functions here.

- (* Translation function (V: M → M') *)

```
fun itran (ID v) = IID(v)
  | itran (LAM (v, e)) = ILAM(v, itran(e))
  | itran (APP(e1,e2)) = IAPP(itran(e2),itran(e1));
```

- (* Auxiliary functions for T and U *)

```
fun cfree v1 (CID v2) = (v1 = v2)
  | cfree v (CAPP(e1,e2)) = (cfree v e1) orelse (cfree v e2)
  | cfree v (CI) = false
  | cfree v (CK) = false
  | cfree v (CS) = false;

fun comfun v1 (CID v2) = if (v1 = v2)
            then CI
            else CAPP(CK, (CID v2))
  | comfun v (CAPP(e1,e2)) = if not (cfree v (CAPP(e1,e2)))
            then CAPP(CK, CAPP(e1,e2))
            else if not (cfree v e1) andalso ((CID v) = e2)
                        then e1
                        else CAPP(CAPP(CS, (comfun v e1)), (comfun v e2))
  | comfun v e = e;
```

- (* Translation function (U: M → M") *)

```
fun ltoc (ID v) = (CID v)
  | ltoc (LAM(v,e)) = comfun v (ltoc e)
  | ltoc (APP(e1,e2)) = CAPP(ltoc e1,ltoc e2);
```

- (* Translation function (T: M' → M") *)

```
fun itoc (IID v) = (CID v)
  | itoc (ILAM(v,e)) = (comfun v (itoc e))
  | itoc (IAPP(e1,e2)) = (CAPP((itoc e2),(itoc e1)));
```

6. Test these functions on all possible translations between these various sets for all the given terms vx, vy, vz, t1, … t9 and give your output clearly. For example, my itran translates from M to M’ and my printIEXP prints expressions in M’. You need to show how all your terms are translated in all these sets and how you print them.

- (* Translation function (V: M → M') and printing function for M’ *)

```
- printIEXP(itran vx);
xval it = () : unit
- printIEXP(itran vy);
yval it = () : unit
- printIEXP(itran vz);
zval it = () : unit
- printIEXP(itran t1);
[x]xval it = () : unit
- printIEXP(itran t2);
[y]xval it = () : unit
- printIEXP(itran t3);
<z><[y]x>[x]xval it = () : unit
- printIEXP(itran t4);
<z>[x]xval it = () : unit
- printIEXP(itran t5);
<<z><[y]x>[x]x><z><[y]x>[x]xval it = () : unit
- printIEXP(itran t6);
[x][y][z]<<z>y><z>xval it = () : unit
- printIEXP(itran t7);
<[x]x><[x]x>[x][y][z]<<z>y><z>xval it = () : unit
- printIEXP(itran t8);
[z]<<z>[x]x>zval it = () : unit
- printIEXP(itran t9);
<<z><[y]x>[x]x>[z]<<z>[x]x>zval it = () : unit
```

- (* Translation function (U: M → M'') and printing function for M'' *)

```
- printCEXP(ltoc vx);
xval it = () : unit
- printCEXP(ltoc vy);
yval it = () : unit
- printCEXP(ltoc vz);
zval it = () : unit
- printCEXP(ltoc t1);
I''val it = () : unit
- printCEXP(ltoc t2);
(K'' x)val it = () : unit
- printCEXP(ltoc t3);
((I'' (K'' x)) z)val it = () : unit
- printCEXP(ltoc t4);
(I'' z)val it = () : unit
- printCEXP(ltoc t5);
(((I'' (K'' x)) z) ((I'' (K'' x)) z))val it = () : unit
- printCEXP(ltoc t6);
S''val it = () : unit
- printCEXP(ltoc t7);
((S'' I'') I'')val it = () : unit
- printCEXP(ltoc t8);
((S'' I'') I'')val it = () : unit
- printCEXP(ltoc t9);
(((S'' I'') I'') ((I'' (K'' x)) z))val it = () : unit
```

- (* Translation function (T: M' → M'') and printing function for M'' *)

```
- printCEXP(itoc ivx);
xval it = () : unit
- printCEXP(itoc ivy);
yval it = () : unit
- printCEXP(itoc ivz);
zval it = () : unit
- printCEXP(itoc it1);
I''val it = () : unit
- printCEXP(itoc it2);
(K'' x)val it = () : unit
- printCEXP(itoc it3);
((I'' (K'' x)) z)val it = () : unit
- printCEXP(itoc it4);
(I'' z)val it = () : unit
- printCEXP(itoc it5);
(((I'' (K'' x)) z) ((I'' (K'' x)) z))val it = () : unit
- printCEXP(itoc it6);
S''val it = () : unit
- printCEXP(itoc it7);
((S'' I'') I'')val it = () : unit
- printCEXP(itoc it8);
((S'' I'') I'')val it = () : unit
- printCEXP(itoc it9);
(((S'' I'') I'') ((I'' (K'' x)) z))val it = () : unit
```

7. Define the subterms in M’’ and implement this function in SML. You should give below the formal definition of subterm’’, its implementation in SML and you need to test on finding the subterms for all combinator terms that correspond to vx, vy, vz, t1, … t9.

- Definition of subterms in M”

subterms”(v)	=	{v}
subterms”(I)	=	{I}
subterms”(K)	=	{K}
subterms”(S)	=	{S}
subterms”(PQ)	=	{PQ} ∪ subterms”(P) ∪ subterms”(Q)

- Implementation of the function csubterms (*subterms”*) in SML

```
(* Gives the list of subterms *)
fun csubterms (CID v) = [(CID v)]
  | csubterms (CI) = [(CI)]
  | csubterms (CK) = [(CK)]
  | csubterms (CS) = [(CS)]
  | csubterms (CAPP(e1,e2)) = (csubterms e1) @ (csubterms e2) @ [(CAPP(e1,e2))];

(* Checks if term is present in the list *)
fun cfind (e:COM) [] = false
  | cfind e1 [e2] = (e1 = e2)
  | cfind e l = (e = (hd l)) orelse cfind e (tl l);

(* Removes all duplicates from a COM list *)
fun cremoveduplicates [e] = [e]
  | cremoveduplicates l = if not (cfind (hd l) (tl l))
                                    then [hd l] @ cremoveduplicates (tl l)
                                    else cremoveduplicates (tl l);

(* prints the elements of a COM list in a nicely formatted way *)
fun printprettylist [e] = (printCEXP e; print ", \n" )
  | printprettylist l = (printCEXP (hd l); print ", \n"; printprettylist (tl l));

(* Removes duplicates and prints them out nicely *)
fun printlistcomb l = printprettylist(cremoveduplicates l);

(* Finds the subterms of a COM term, removes duplicates and prints them out nicely *)
fun printcsubterms t = printlistcomb(csubterms t);
```

- Printing subterms of the terms in M’’, with repetitions

```
- csubterms cvx;
val it = [CID "x"] : COM list
- csubterms cvy;
val it = [CID "y"] : COM list
- csubterms cvz;
val it = [CID "z"] : COM list
- csubterms ct1;
val it = [CI] : COM list
- csubterms ct2;
val it = [CK,CID "x",CAPP (CK,CID "x")] : COM list
- csubterms ct3;
val it =
  [CI,CK,CID "x",CAPP (CK,CID "x"),CAPP (CI,CAPP (CK,CID "x")),CID "z",
   CAPP (CAPP (CI,CAPP (CK,CID "x")),CID "z")] : COM list
- csubterms ct4;
val it = [CI,CID "z",CAPP (CI,CID "z")] : COM list
- csubterms ct5;
val it =
  [CI,CK,CID "x",CAPP (CK,CID "x"),CAPP (CI,CAPP (CK,CID "x")),CID "z",
   CAPP (CAPP (CI,CAPP (CK,CID "x")),CID "z"),CI,CK,CID "x",CAPP (CK,CID "x"),
   CAPP (CI,CAPP (CK,CID "x")),...] : COM list
- csubterms ct5;
val it =
  [CI,CK,CID "x",CAPP (CK,CID "x"),CAPP (CI,CAPP (CK,CID "x")),CID "z",
   CAPP (CAPP (CI,CAPP (CK,CID "x")),CID "z"),CI,CK,CID "x",CAPP (CK,CID "x"),
   CAPP (CI,CAPP (CK,CID "x")),...] : COM list
- csubterms ct6;
val it = [CS] : COM list
- csubterms ct7;
val it = [CS,CI,CAPP (CS,CI),CI,CAPP (CAPP (CS,CI),CI)] : COM list
- csubterms ct8;
val it = [CS,CI,CAPP (CS,CI),CI,CAPP (CAPP (CS,CI),CI)] : COM list
- csubterms ct9;
val it =
  [CS,CI,CAPP (CS,CI),CI,CAPP (CAPP (CS,CI),CI),CI,CK,CID "x",
   CAPP (CK,CID "x"),CAPP (CI,CAPP (CK,CID "x")),CID "z",
   CAPP (CAPP (CI,CAPP (CK,CID "x")),CID "z"),...] : COM list
```

- Printing the same subterms nicely and without repetitions

```
- printcsubterms cvx;
x
val it = () : unit
```

```
- printcsubterms cvy;
y
val it = () : unit

- printcsubterms cvz;
z
val it = () : unit

- printcsubterms ct1;
I"
val it = () : unit

- printcsubterms ct2;
K",
x,
(K" x)
val it = () : unit

- printcsubterms ct3;
I",
K",
x,
(K" x),
(I" (K" x)),
z,
((I" (K" x)) z)
val it = () : unit

- printcsubterms ct4;
I",
z,
(I" z)
val it = () : unit

- printcsubterms ct5;
I",
K",
x,
(K" x),
(I" (K" x)),
z,
((I" (K" x)) z),
(((I" (K" x)) z) ((I" (K" x)) z))
val it = () : unit

- printcsubterms ct6;
S"
val it = () : unit
```

```
- printcsubterms ct7;
S",
(S" I"),
I",
((S" I") I")
val it = () : unit

- printcsubterms ct8;
S",
(S" I"),
I",
((S" I") I")
val it = () : unit

- printcsubterms ct9;
S",
(S" I"),
((S" I") I"),
I",
K",
x,
(K" x),
(I" (K" x)),
z,
((I" (K" x)) z),
(((S" I") I") ((I" (K" x)) z))
val it = () : unit
```

8. Implement the combinatory reduction rules =c given in the data sheets and use your implementation to reduce all combinator terms that correspond to vx, vy, vz, t1, … t9 showing all reduction steps.

```
(* Checks if term is combinatory redex. *)
fun is_credex (CAPP(CI,_)) = true
  | is_credex (CAPP(CAPP(CK,_),_)) = true
  | is_credex (CAPP(CAPP(CAPP(CS,_),_),_)) = true
  | is_credex _ = false;

(* Checks if a term has combinatory redexes in it. *)
fun has_credex (CID v) = false
  | has_credex (CI) = false
  | has_credex (CK) = false
  | has_credex (CS) = false
  | has_credex (CAPP(e1,e2)) = (is_credex (CAPP(e1,e2)))
                                              orelse has_credex e1
                                              orelse has_credex e2;
```

```
(* This is the function that implement the actual combinatory reduction. *)
fun cred (CAPP(CI,e1)) = e1
|     cred (CAPP(CAPP(CK,e1),e2)) = e1
|     cred (CAPP(CAPP(CAPP(CS,e1),e2),e3)) = (CAPP(CAPP(e1,e3),CAPP(e2,e3)))
|     cred e = e;

(* This function does one combinatory reduction step*)
fun one_creduce (CAPP(e1,e2)) = if is_credex (CAPP(e1,e2))
                                 then cred (CAPP(e1,e2))
                                 else if (has_credex e1)
                                        then (CAPP(one_creduce e1,e2))
                                        else if (has_credex e2)
                                               then (CAPP(e1,one_creduce e2))
                                               else (CAPP(e1,e2))
  | one_creduce e = e;

(* Returns a list with all combinatory reduction of one term *)
fun creduce e = if(has_credex e)
                                 then [e] @ creduce(one_creduce e)
                                 else [e];

(* Prints a COM list in a nicely formatted way *)
fun printcredexlist [] = print "\n"
  | printcredexlist [e] =
  ( printCEXP e;
        print "\n" )
  | printcredexlist l =
  ( printCEXP (hd l);
        print " → \n";
        printcredexlist (tl l) );

(* This function creates a list of combinatory redex from a term and then prints it out nicely. *)
fun printcreduce e = printcredexlist (creduce e);
```

- Testing the combinatory reduction function:

```
- printcreduce cvx;
x
val it = () : unit

- printcreduce cvy;
y
val it = () : unit

- printcreduce cvz;
z
val it = () : unit
```

```
- printcreduce ct1;
I"
val it = () : unit

- printcreduce ct2;
(K" x)
val it = () : unit

- printcreduce ct3;
((I" (K" x)) z) →
((K" x) z) →
x
val it = () : unit

- printcreduce ct4;
(I" z) →
z
val it = () : unit

- printcreduce ct5;
(((I" (K" x)) z) ((I" (K" x)) z)) →
(((K" x) z) ((I" (K" x)) z)) →
(x ((I" (K" x)) z)) →
(x ((K" x) z)) →
(x x)
val it = () : unit

- printcreduce ct6;
S"
val it = () : unit

- printcreduce ct7;
((S" I") I")
val it = () : unit

- printcreduce ct8;
((S" I") I")
val it = () : unit

- printcreduce ct9;
(((S" I") I") ((I" (K" x)) z)) →
((I" ((I" (K" x)) z)) (I" ((I" (K" x)) z))) →
(((I" (K" x)) z) (I" ((I" (K" x)) z))) →
(((K" x) z) (I" ((I" (K" x)) z))) →
(x (I" ((I" (K" x)) z))) →
(x ((I" (K" x)) z)) →
(x ((K" x) z)) →
(x x)
val it = () : unit
```

9. For creduce in the above question, implement a counter that counts the number of –>’s used to reach a normal form.

- Function that calculates the number of reduction steps for M’’

```
fun countprintcreduce e = ( let val templist = (creduce e) in
  ( printcredexlist templist;
          print (Int.toString((length templist) - 1));
          print " steps\n" ) end);
```

- Testing the function on terms vx,vy,vz,t1, t9

```
- countprintcreduce cvx;
x
0 steps
val it = () : unit

- countprintcreduce cvy;
y
0 steps
val it = () : unit

- countprintcreduce cvz;
z
0 steps
val it = () : unit

- countprintcreduce ct1;
I"
0 steps
val it = () : unit

- countprintcreduce ct2;
(K" x)
0 steps
val it = () : unit

- countprintcreduce ct3;
((I" (K" x)) z) →
((K" x) z) →
x
2 steps
val it = () : unit
```

```
- countprintcreduce ct4;
(I'' z) →
z
1 steps
val it = () : unit



- countprintcreduce ct5;
(((I'' (K'' x)) z) ((I'' (K'' x)) z)) →
(((K'' x) z) ((I'' (K'' x)) z)) →
(x ((I'' (K'' x)) z)) →
(x ((K'' x) z)) →
(x x)
4 steps
val it = () : unit

- countprintcreduce ct6;
S''
0 steps
val it = () : unit

- countprintcreduce ct7;
((S'' I'') I'')
0 steps
val it = () : unit

- countprintcreduce ct8;
((S'' I'') I'')
0 steps
val it = () : unit

- countprintcreduce ct9;
(((S'' I'') I'') ((I'' (K'' x)) z)) →
((I'' ((I'' (K'' x)) z)) (I'' ((I'' (K'' x)) z))) →
(((I'' (K'' x)) z) (I'' ((I'' (K'' x)) z))) →
(((K'' x) z) (I'' ((I'' (K'' x)) z))) →
(x (I'' ((I'' (K'' x)) z))) →
(x ((I'' (K'' x)) z)) →
(x ((K'' x) z)) →
(x x)
7 steps
val it = () : unit
```

10. Implement η-reduction on M and test it on many examples of your own. Give the implementation as well as the test showing all the reduction steps one by one until you reach a η-normal form.

- eta reduction function (with all auxiliary needed for the implementation)

```
(* Return true if a variable is free in another term, otherwise return false *)
fun free v1 (ID v2) = (v1 = v2) |
   free v1 (APP(e1,e2))= (free v1 e1) orelse (free v1 e2) |
   free v1 (LAM(v2, e1)) = (free v1 e1) andalso not (v1 = v2);

(* Checks if term is eta redex *)
fun is_eredex (LAM(v1,(APP(e,v2)))) = ((ID v1) = v2) andalso (free v1 (APP(e,v2)))
  | is_eredex _ = false;

(* Checks if term has eta redex *)
fun has_eredex (ID v) = false
  | has_eredex (LAM(v,e)) = is_eredex (LAM(v,e)) orelse has_eredex e
  | has_eredex (APP(e1,e2)) = (has_eredex e1) orelse (has_eredex e2);

(* This function does one eta reduction step *)
fun one_ereduce (ID v) = (ID v)
  | one_ereduce (LAM(v,(APP(e1,e2)))) = if (is_eredex (LAM(v,(APP(e1,e2)))))
                                              then (e1)
                                              else if (has_eredex (APP(e1,e2)))
                                                      then (one_ereduce (APP(e1,e2)))
                                                      else (LAM(v,(APP(e1,e2))))
  | one_ereduce (LAM(v,e)) = if(has_eredex e)
                                              then (LAM(v,(one_ereduce e)))
                                              else (LAM(v,e))
  | one_ereduce (APP(e1,e2)) = if (has_eredex e1)
                                              then (APP(one_ereduce e1,e2))
                                              else if (has_eredex e2)
                                                      then (APP(e1, one_ereduce e2))
                                                      else  (APP(e1,e2));

(* Creates a list of the sequence of eta redexes for one term *)
fun ereduce e = if (has_eredex e)
                                              then [e] @ ereduce (one_ereduce e)
                                              else [e];

(* Prints out the list of eta redexes in a pretty way *)
fun printeredexlist [] = print "\n"
  | printeredexlist [e] =
  ( printLEXP e;
         print "\n" )
  | printeredexlist l =
  ( printLEXP (hd l);
         print " → η\n";
         printeredexlist (tl l) );
```

```
(* This function uses all functions above and prints out a nicely formatted eta reduction sequence. *)
fun printereduce e = printeredexlist (ereduce e);
```

- Testing the eta reduction function

- t10

```
val t10 = (LAM("x",APP(LAM("y",APP(vz,vy)),vx)));
val t10 = LAM ("x",APP (LAM ("y",APP (ID "z",ID "y")),ID "x")) : LEXP

- printLEXP(t10);
(λx.((λy.(z y)) x))val it = () : unit

- printereduce t10;
(λx.((λy.(z y)) x)) → η
(λy.(z y)) → η
z
val it = () : unit
```

- t11

```
- val t11 = (LAM("z",APP(t10,vz)));
val t11 =
  LAM ("z",APP (LAM ("x",APP (LAM ("y",APP (ID "z",ID "y")),ID "x")),ID "z")) : LEXP

- printLEXP(t11);
(λz.((λx.((λy.(z y)) x)) z))val it = () : unit

- printereduce t11;
(λz.((λx.((λy.(z y)) x)) z)) → η
(λx.((λy.(z y)) x)) → η
(λy.(z y)) → η
z
val it = () : unit
```

- t12

```
- val t12 = (APP(APP(t2,t10),t1));
val t12 = APP (APP (LAM ("y",ID "x"),LAM ("x",APP (LAM ("y",APP (ID "z",ID "y")),ID "x"))),
     LAM ("x",ID "x")) : LEXP

- printLEXP(t12);
(((λy.x) (λx.((λy.(z y)) x))) (λx.x))val it = () : unit

- printereduce t12;
(((λy.x) (λx.((λy.(z y)) x))) (λx.x)) → η
(((λy.x) (λy.(z y))) (λx.x)) → η
(((λy.x) z) (λx.x))
val it = () : unit
```

- t13

```
- val t13 = (LAM("x",APP(t12,vx)));
val t13 = LAM("x", APP (APP (APP (LAM ("y",ID "x"),
          LAM ("x",APP (LAM ("y",APP (ID "z",ID "y")),ID "x"))),
        LAM ("x",ID "x")),ID "x")) : LEXP

- printLEXP(t13);
(λx.((((λy.x) (λx.((λy.(z y)) x))) (λx.x)) x))val it = () : unit

printereduce t13;
(λx.((((λy.x) (λx.((λy.(z y)) x))) (λx.x)) x)) → η
(((λy.x) (λx.((λy.(z y)) x))) (λx.x)) → η
(((λy.x) (λy.(z y))) (λx.x)) → η
(((λy.x) z) (λx.x))
val it = () : unit
```

11. Give an implementation of leftmost reduction in M and test it on a number of rich examples

- All auxiliary function needed in order to do leftmost reduction

```
(*the function below adds lambda id to a list of terms *)
fun addlam id [] = [] |
   addlam id (e::l) = (LAM(id,e))::(addlam id l);

(*Finds a beta redex*)
fun is_redex (APP(LAM(_,_),_)) =
      true
  | is_redex _ =
      false;

(* checks whether a variable is free in a term *)
fun free id1 (ID id2) = if (id1 = id2) then true else false|
   free id1 (APP(e1,e2))= (free id1 e1) orelse (free id1 e2) |
   free id1 (LAM(id2, e1)) = if (id1 = id2) then false else (free id1 e1);

(* finds new variables which are fresh  in l and different from id*)
fun findme id l = (let val id1 = id^"1"  in if not (List.exists (fn x => id1 = x) l) then id1 else (findme id1 l)
end);

(* finds the list of free variables in a term *)
fun freeVars (ID id2)       = [id2]
  | freeVars (APP(e1,e2))   = freeVars e1 @ freeVars e2
  | freeVars (LAM(id2, e1)) = List.filter (fn x => not (x = id2)) (freeVars e1); (* Filter out of the list *)
```

```
(*does substitution avoiding the capture of free variables*)
fun subs e id (ID id1) =  if id = id1 then e else (ID id1) |
    subs e id (APP(e1,e2)) = APP(subs e id e1, subs e id e2)|
    subs e id (LAM(id1,e1)) = (if id = id1 then LAM(id1,e1) else
                                   if (not (free id e1) orelse not (free id1 e))
                                                  then LAM(id1,subs e id e1)
                                   else (let val id2 = (findme id ([id1]@ (freeVars e) @ (freeVars e1)))
                                                       in LAM(id2, subs e id (subs (ID id2) id1 e1)) end));

(*beta-reduces a redex*)
fun red (APP(LAM(id,e1),e2)) = subs e2 id e1;

(* Checks that the term has at least one beta redex in it *)
fun has_redex (ID id) = false |
    has_redex (LAM(id,e)) = has_redex e|
    has_redex (APP(e1,e2)) = if (is_redex  (APP(e1,e2))) then true else
                              ((has_redex e1) orelse (has_redex e2));

(* Does one leftmost beta reduction step *)
fun one_loreduce (ID id) = (ID id)|
    one_loreduce (LAM(id,e)) = LAM(id, (one_loreduce e))|
    one_loreduce (APP(e1,e2)) = if (is_redex (APP(e1,e2))) then (red (APP(e1,e2))) else
                                 if (has_redex e1) then APP(one_loreduce e1, e2) else
                                 if (has_redex e2) then APP(e1, (one_loreduce e2)) else (APP(e1,e2));


(* Leftmost outermost reduction function *)
fun loreduce (ID id) =  [(ID id)] |
    loreduce (LAM(id,e)) = (addlam id (loreduce e)) |
    loreduce (APP(e1,e2)) = (let val l1 = if (is_redex (APP(e1,e2))) then  (loreduce (red (APP(e1,e2)))) else
                                         if (has_redex e1) then (loreduce (APP(one_loreduce e1, e2))) else
                                         if (has_redex e2) then  (loreduce (APP(e1, (one_loreduce e2)))) else []
                                         in [APP(e1,e2)]@l1
                                    end);

(*prints elements from a list putting an arrow in between*)
fun printlistreduce [] = ()|
    printlistreduce (e::[]) = (printLEXP e) |
    printlistreduce (e::l) = (printLEXP e; print "-->\n" ; (printlistreduce l));
```

- Leftmost is printloreduce in the data-files.sml

```
(* Prints out the leftmost beta reduction sequence *)
fun printloreduce e = (let val tmp =  (loreduce e)
                          in (printlistreduce tmp; print "\n") end);
```

- Testing leftmost reduction using omega

```
- val half_omega = (LAM("x",APP(vx,vx)));
val half_omega = LAM ("x",APP (ID #,ID #)) : LEXP

- val omega = (APP(half_omega,half_omega));
val omega = APP (LAM ("x",APP #),LAM ("x",APP #)) : LEXP
```

- t14 (Should terminate)

```
- val t14 = (APP(t2,omega));
val t14 = APP (LAM ("y",ID #),APP (LAM #,LAM #)) : LEXP
- printloreduce t14;
((λy.x) ((λx.(x x)) (λx.(x x))))-->
x
val it = () : unit
```

- t15 (Should not terminate)

```
- val t15 = (APP(omega,t2));
val t15 = APP (APP (LAM #,LAM #),LAM ("y",ID #)) : LEXP
- printloreduce t15;
#############
```

- t16 (Should terminate)

```
- val t16 = (LAM("x", APP(t2, APP(t3,omega))));
val t16 = LAM ("x",APP (LAM #,APP #)) : LEXP

- printloreduce t16;
(λx.((λy.x) ((((λx.x) (λy.x)) z) ((λx.(x x)) (λx.(x x))))))-->
(λx.x)
val it = () : unit
```

- t17 (Should not terminate)

```
- val t17 = (LAM("x", APP(t3,omega)));
val t17 = LAM ("x",APP (APP #,APP #)) : LEXP
- printloreduce t17;
############
```

FINITO