Foundations1 assignment 2017

By Jordan Walker H00222194

November 6, 2017

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

1. Just like I defined translation functions $T : \mathcal{M}' \mapsto \mathcal{M}''$ and $\omega : \mathcal{M} \mapsto \Lambda$, give translation functions from $U : \mathcal{M} \mapsto \mathcal{M}''$ and $V : \mathcal{M} \mapsto \mathcal{M}'$ and $\omega' : \mathcal{M}' \mapsto \Lambda'$. Your translation functions need to be complete with all subfunctions and needed information (just like T and ω were complete with all needed information). Submit all these functions here. (1)

2. For each of the SML terms vx, vy, vz, t1, $\cdots$ t9 in `http://www.macs.hw.ac.uk/~fairouz/foundations-2017/slides/data-files.sml`, let the overlined term represent the corresponding term in $\mathcal{M}$. I.e., $\overline{\text{vx}} = x$, $\overline{\text{vy}} = y$, $\overline{\text{vz}} = z$, $\overline{\text{t1}} = \lambda x.x$, $\overline{\text{t2}} = \lambda y.x$, $\cdots$.
   For each of $\overline{\text{vx}}$, $\overline{\text{vy}}$, $\overline{\text{vz}}$, $\overline{\text{t1}}$, $\overline{\text{t2}}$, $\cdots \overline{\text{t9}}$ in $\mathcal{M}$, translate it into the corresponding terms of $\mathcal{M}'$, $\mathcal{M}''$, Λ and Λ' using the translation functions V, U, ω and ω'.
   Your output should be tidy as follows:

	V	U	ω	ω'
$\lambda x.x$	$[x]x$	I''	$\lambda 1$	$[\,]1$

   (1)

3. Just like I introduced SML terms vx, vy, vz, t1, t2, $\cdots$ t9 which implement terms in $\mathcal{M}$, please implement the corresponding terms each of the other sets $\mathcal{M}'$, Λ, Λ', $\mathcal{M}''$. Your output must be like my output in `http://www.macs.hw.ac.uk/~fairouz/foundations-2017/slides/data-files.sml`, for the implementation of these terms of $\mathcal{M}$. I.e., your output for each set must be similar to the following: (1)

   The implementation of terms in $\mathcal{M}$ is as follows:

```
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));
```

4. For each of $\mathcal{M}'$, Λ, Λ', $\mathcal{M}''$, 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, $\cdots$ t9. Your output for each such set must be similar to the one below (1)

```
(*Prints a term in classical lambda calculus*)
fun printLEXP (ID v) =
    print v
  | printLEXP (LAM (v,e)) =
    (print "(\\";
     print v;
     print ".";
     printLEXP e;
     print ")")
  | printLEXP (APP(e1,e2)) =
    (print "(";
     printLEXP e1;
     print " ";
     printLEXP e2;
     print ")");
```

Printing these $\mathcal{M}$ terms yields:

```
printLEXP vx;
xval it = () : unit

printLEXP vy;
yval it = () : unit

printLEXP vz;
zval it = () : unit
```

```
 printLEXP t1;
(\x.x)val it = () : unit

printLEXP t2;
(\y.x)val it = () : unit

printLEXP t3;
(((\x.x) (\y.x)) z)val it = () : unit

 printLEXP t4;
((\x.x) z)val it = () : unit

printLEXP t5;
((((\x.x) (\y.x)) z) (((\x.x) (\y.x)) z))val it = () : unit

printLEXP t6;
(\x.(\y.(\z.((x z) (y z)))))val it = () : unit

printLEXP t8;
(\z.(z ((\x.x) z)))val it = () : unit

printLEXP t9;
((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))val it = () : unit
```

5. Implement in SML the translation functions T, U and V and give these implemented functions here. (1)

6. Test these functions on all possible translations between these various sets for all the given terms vx, vy, vz, t1, $\cdots$ t9 and give your output clearly.

   For example, my itran translates from $\mathcal{M}$ to $\mathcal{M}'$ and my printIEXP prints expressions in $\mathcal{M}'$. Hence,

   ```
   - printIEXP (itran t5);
   <<z><[y]x>[x]x><z><[y]x>[x]xval it = () : unit
   ```

   You need to show how all your terms are translated in all these sets and how you print them. (2)

7. Define the subterms in $\mathcal{M}''$ 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, $\cdots$ t9. For example, if ct1 and ct2 are the terms that correspond to t1 and t2 then

```
- subterm2 ct1;
val it = [CI] : COM list
- subterm2 ct2;
val it = [CAPP (CK,CID "x"),CK,CID "x"] : COM list
```

(1)

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, $\cdots$ t9 showing all reduction steps. For example,

```
-creduce ct3;
ct3 =
I(Kx)z -->
Kxz -->
x
-creduce ct5;
ct5 =
I(Kx)z(I(Kx)z)-->
K x z(I(Kx)z)-->
x(I(Kx)z)-->
x(Kxz) -->
xx
```

(1)

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

```
-creduce ct3;
ct3 =
I(Kx)z -->
Kxz -->
x
2 setps
```

```
-creduce ct5;
ct5 =
I(Kx)z(I(Kx)z)-->
K x z(I(Kx)z)-->
x(I(Kx)z)-->
x(Kxz) -->
xx
4 steps
```

(1)

10. Implement η-reduction on $\mathcal{M}$ 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. (1)

11. Translate $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ in each of $\mathcal{M}'$, $\mathcal{M}''$, Λ and Λ' and give the SML implementation of all these translations. (1)

12. Assume comega is your SML implementation of the term that corresponds to Ω. Run -creduce comega; and say what happens. (1)

13. Give an implementation of leftmost reduction and rightmost reduction in $\mathcal{M}$ and test them on a number of examples that show which terminates more and which is more efficient. (2)

DATA SHEET

At `http://www.macs.hw.ac.uk/~fairouz/foundations-2017/slides/data-files.sml`, you find an implementation in SML of the set of terms $\mathcal{M}$ and many operations on it. You can use all of these in your assignment. You can also use any other help SML functions I have given you. Anything you use from elsewhere has to be well cited/referenced.

† The syntax of the classical λ-calculus is given by $\mathcal{M} ::= \mathcal{V} \mid (\lambda\mathcal{V}.\mathcal{M}) \mid (\mathcal{M}\mathcal{M})$.
We assume the usual notational conventions in $\mathcal{M}$ and use the reduction rule:
$\underline{(\lambda v.P)Q} \to_\beta P[v := Q]$.

† The syntax of the λ-calculus in item notation is given by $\mathcal{M}' ::= \mathcal{V} \mid [\mathcal{V}]\mathcal{M}' \mid \langle\mathcal{M}'\rangle\mathcal{M}'$.
We use the reduction rule: $\underline{\langle Q'\rangle[v]}P' \to_{\beta'} [x := Q']P'$.

† In $\mathcal{M}$, (PQ) stands for the application of function P to argument Q.
† In $\mathcal{M}'$, $\langle Q'\rangle P'$ stands for the application of function P' to argument Q' (note the reverse order).

† The syntax of the classical λ-calculus with de Bruijn indices is given by
$\Lambda ::= \mathbb{N} \mid (\lambda\Lambda) \mid (\Lambda\Lambda)$.

† We define free variables in the classical λ-calculus with de Bruijn indices as follows: $FV(n) = \{n\}$, $FV(AB) = FV(A) \cup FV(B)$ and $FV(\lambda A) = FV(A) \setminus \{1\}$.

† For $[x_1, \cdots, x_n]$ a list (not a set) of variables, we define $\omega_{[x_1,\cdots,x_n]} : \mathcal{M} \mapsto \Lambda$ inductively by:
$\omega_{[x_1,\cdots,x_n]}(v_i) = \min\{j : v_i \equiv x_j\}$
$\omega_{[x_1,\cdots,x_n]}(AB) = \omega_{[x_1,\cdots,x_n]}(A)\omega_{[x_1,\cdots,x_n]}(B)$
$\omega_{[x_1,\cdots,x_n]}(\lambda x.A) = \lambda\omega_{[x,x_1,\cdots,x_n]}(A)$
Hence $\omega_{[x,y,x,y,z]}(x) = 1$, $\omega_{[x,y,x,y,z]}(y) = 2$ and $\omega_{[x,y,x,y,z]}(z) = 5$.
Also $\omega_{[x,y,x,y,z]}(xyz) = 1\ 2\ 5$.
Also $\omega_{[x,y,x,y,z]}(\lambda xy.xz) = \lambda\lambda 2\ 7$.

Assume our variables are ordered as follows: $v_1, v_2, v_3, \cdots$.
We define $\omega : \mathcal{M} \mapsto \Lambda$ by $\omega(A) = \omega_{[v_1,\cdots,v_n]}(A)$ where $FV(A) \subseteq \{v_1, \cdots, v_n\}$.
So for example, if our variables are ordered as $x, y, z, x', y', z', \cdots$ then $\omega(\lambda xyx'.xzx') = \omega_{[x,y,z]}(\lambda xyx'.xzx') = \lambda\omega_{[x,x,y,z]}(\lambda yx'.xzx') = \lambda\lambda\omega_{[y,x,x,y,z]}(\lambda x'.xzx') = \lambda\lambda\lambda\omega_{[x',y,x,x,y,z]}(xzx') = \lambda\lambda\lambda 3\ 6\ 1$.

† The syntax of the λ-calculus in item notation is given by
$\Lambda' ::= \mathbb{N} \mid [\,]\Lambda' \mid \langle\Lambda'\rangle\Lambda'$.

† The syntax of combinatory logic is given by
$\mathcal{M}'' ::= \mathcal{V} \mid \mathrm{I}'' \mid \mathrm{K}'' \mid \mathrm{S}'' \mid (\mathcal{M}''\mathcal{M}'')$
We assume that application associates to the left in $\mathcal{M}''$. I.e., $P''Q''R''$ stands for $((P''Q'')R'')$.
We use the reduction rules:
(I'') $\underline{\mathrm{I}''v} =_c v$ (K'') $\underline{\mathrm{K}''v_1v_2} =_c v_1$ (S'') $\underline{\mathrm{S}''v_1v_2v_3} =_c v_1v_3(v_2v_3)$.

Note that these rules are from left to right (and not right to left) even though they are written with an = sign.

† We define free variables in combinatory logic as follows:
$FV''(v) = \{v\}$
$FV''(\mathrm{I}'') = FV''(\mathrm{K}'') = FV''(\mathrm{S}'') = \{\}$
$FV''(P''Q'') = FV''(P'') \cup FV''(Q'')$.

† Here is a possible translation function T from $\mathcal{M}'$ to $\mathcal{M}''$:
$T(v) = v$ $\quad T([v]P') = f(v, T(P'))$ $\quad T(\langle Q'\rangle P') = (T(P')T(Q'))$ 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 $v \notin FV(P'')$
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}

† Assume the following SML datatypes which implement $\mathcal{M}$, Λ, $\mathcal{M}'$, Λ' and $\mathcal{M}''$ respectively (here, if `e1` implements A_1' and `e2` implements A_2', then `IAPP(e1,e2)` implements $\langle A_1'\rangle A_2'$ which stands for the functionA_2' applied to argumentA_1'):

```
datatype LEXP =
   APP of LEXP * LEXP | LAM of string *  LEXP |  ID of string;

datatype BEXP =
   BAPP of BEXP * BEXP | BLAM of BEXP |  BID of int;

datatype IEXP =
   IAPP of IEXP * IEXP | ILAM of string *  IEXP |  IID of string;

datatype IBEXP =
   IBAPP of IBEXP * IBEXP | IBLAM of    IBEXP |  IBID of int;

datatype COM = CAPP of COM*COM | CID of string | CI | CK | CS;
```

Jordan Walker
Foundations Coursework 2017

† The Following Answers are my own work regarding the questions asked for the assignment "Foundations 1" There is a total of 13 answers each numerated to clearly show which answer is for which question.

1. Handwritten Translation Functions

$$U : \mathcal{M} \mapsto \mathcal{M}''$$

$$U(\lambda x.x) = I'' \qquad U(\lambda xy.x) = K'' \qquad U(\lambda xyz.xz(yz)) = S''$$

$$U(v) = v, \qquad U(\lambda v.P) = f(v, U(P)), \qquad U(PQ) = (U(P)U(Q))$$

1. $f(v, v) = I''$
2. $f(v, P) = K''P$ if $v \notin FV(P)$
3. $f(v, P, Q) = \begin{cases} P & \text{if } v \notin FV(P) \text{ and } P \equiv v \\ S''f(v, P)f(v, Q) & \text{otherwise.} \end{cases}$

$$V : \mathcal{M} \mapsto \mathcal{M}'$$

$$\mathrm{V(v) = v} \qquad \mathrm{V(\lambda v.P) = [v]V(P)} \qquad \mathrm{V(PQ) = \langle v(P)\rangle v(Q)}$$

$$\omega' : \mathcal{M}' \mapsto \Lambda'.$$

$$\omega'_{[x_1,\cdots,x_n]}(v_i) = \min\{j : v_i \equiv x_j\}$$
$$\omega'_{[x_1,\cdots,x_n]}(PQ) = \omega'_{[x_1,\cdots,x_n]}(P)\omega'_{[x_1,\cdots,x_n]}(Q)$$
$$\omega'_{[x_1,\cdots,x_n]}(\lambda x.P) = \lambda\omega'_{[x,x_1,\cdots,x_n]}(P)$$

2. Translation Tables

	I''	U	ω	ω'
x	x	x	1	1
y	y	y	2	2
z	z	z	3	3

	$\lambda x.x$
V	$[x]x$
U	I''
ω	$\lambda 1$
ω'	$[]1$

	$\lambda y.x$
V	$[y]x$
U	$(K''x)$
ω	$\lambda 2$
ω'	$[]2$

	$(\lambda x.x)(\lambda y.x)z$
V	$\langle z\rangle\langle[y]x\rangle[x]x$
U	$((I''(K''x))z)$
ω	$((\lambda 1\lambda 2)3)$
ω'	$\langle 3\rangle\langle[]2\rangle[]1$

	$(\lambda x.x)z$
V	$\langle z\rangle[x]x$
U	$(I''z)$
ω	$\lambda 13$
ω'	$\langle 3\rangle[]1$

	$((\lambda x.x)(\lambda y.x)z)((\lambda x.x)(\lambda y.x)z)$
V	$\langle\langle z\rangle\langle[y]x\rangle[x]x\rangle\langle z\rangle\langle[y]x\rangle[x]x$
U	$(((I''(K''x))z)((I''(K''x))z))$
ω	$(((\lambda 1\lambda 2)3)((\lambda 1\lambda 2)3))$
ω'	$\langle\langle 3\rangle\langle[]2\rangle[]1\rangle\langle 3\rangle\langle[]2\rangle[]1$

	$(\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$
V	$[x][y][z]\langle\langle z\rangle y\rangle\langle z\rangle x$
U	S''
ω	$(\lambda\lambda\lambda((31)(21)))$
ω'	$[][][]\langle\langle 1\rangle 2\rangle\langle 1\rangle 3$

	$(((\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))(\lambda x.x))(\lambda x.x))$
V	$\langle[x]x > \langle[x]x > [x][y][z]\langle\langle z > y > \langle z > x$
U	$((S''I'')I'')$
ω	$((\lambda\lambda\lambda((31)(21))\lambda 1)\lambda 1)$
ω'	$\langle[]1\rangle\langle[]1\rangle[][][]\langle\langle 1\rangle 2\rangle\langle 1\rangle 3$

	$(\lambda z.(z((\lambda x.x)z)))$
V	$[z]\langle\langle z\rangle[x]x\rangle z$
U	$((S''I'')I'')$
ω	$(\lambda(1(\lambda 11)))$
ω'	$[]\langle\langle 1\rangle[]1\rangle 1$

	$((\lambda z(z((\lambda x.x)z)))(((\lambda x.x)(\lambda y.x))z))$
V	$\langle\langle z\rangle\langle[y]x\rangle[x]x\rangle[z]\langle\langle z\rangle[x]x\rangle z$
U	$(((S''I'')I'')((I''(K''x))z))$
ω	$(\lambda(1(\lambda 11))((\lambda 1\lambda 2)3))$
ω'	$\langle\langle 3\rangle\langle[]2\rangle[]1\rangle[]\langle\langle 1\rangle[]1\rangle 1$

3. Implementing the terms in $\mathcal{M}'$, Λ, Λ', $\mathcal{M}''$

IEXP

```
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));
```

BEXP

```
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));
```

IBEXP

```
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));
```

COM

```
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));
```

4. Implementing Printing Functions for $\mathcal{M}'$, Λ, Λ', $\mathcal{M}''$

```
IEXP
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);

BEXP
fun printBEXP(BID v)=
      print (Int.toString v)
    | printBEXP(BLAM(e))=
      (print "\\";
      printBEXP e;
      print " ")
    | printBEXP(BAPP(e1,e2))=
      (print "(";
      printBEXP e1;
      print " ";
      printBEXP e2;
      print ")"
      );

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

```
      print ">";
      printIBEXP e2);

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

IEXP
> 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
```

BEXP

```
> printBEXP Bv1;
1val it = (): unit
> printBEXP Bv2;
2val it = (): unit
> printBEXP Bv3;
3val it = (): unit
> printBEXP Bt1;
\1 val it = (): unit
> printBEXP Bt2;
\2 val it = (): unit
> printBEXP Bt3;
((\1  \2 ) 3)val it = (): unit
> printBEXP Bt4;
(\1  3)val it = (): unit
> printBEXP Bt5;
(((\1  \2 ) 3) ((\1  \2 ) 3))val it = (): unit
> printBEXP Bt6;
\\\((3 1) (2 1))   val it = (): unit
> printBEXP Bt7;
((\\\((3 1) (2 1))    \1 ) \1 )val it = (): unit
> printBEXP Bt8;
\(1 (\1  1)) val it = (): unit
> printBEXP Bt9;
(\(1 (\1  1))  ((\1  \2 ) 3))val it = (): unit
```

IBEXP

```
> 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
```

COM

```
> printCOM Cvx;
xval it = (): unit
> printCOM Cvz;
zval it = (): unit
> printCOM Cvy;
yval it = (): unit
> printCOM Ct1;
I''val it = (): unit
> printCOM Ct2;
(K'' x)val it = (): unit
> printCOM Ct3;
((I'' (K'' x)) z)val it = (): unit
> printCOM Ct4;
(I'' z)val it = (): unit
> printCOM Ct5;
(((I'' (K'' x)) z) ((I'' (K'' x)) z))val it = (): unit
> printCOM Ct6;
S''val it = (): unit
> printCOM Ct7;
```

```
((S'' I'') I'')val it = (): unit
> printCOM Ct8;
((S'' I'') I'')val it = (): unit
> printCOM Ct9;
(((S'' I'') I'') ((I'' (K'' x)) z))val it = (): unit
```

5. Translation Functions for T, U and V

```
Needed for all Translations

fun cfree id1 (CID id2) = if (id1 = id2) then true else false |
    cfree id1 (CAPP(e1,e2)) = (cfree id1 e1) orelse (cfree id1 e2) |
    cfree id1 (CI) = false |
    cfree id1 (CK) = false |
    cfree id1 (CS) = false ;

fun cfunf id (CID id1) =
          if id = id1 then CI
          else CAPP(CK,(CID id1)) |
    cfunf id(CAPP(e1,e2))=
          if not(cfree id (CAPP(e1,e2)))
          then CAPP(CK,CAPP(e1,e2))
          else
          if((CID id) = e2) andalso not (cfree id e1)
          then e1
          else CAPP(CAPP(CS,(cfunf id e1)),(cfunf id e2)) |
    cfunf id (_) = raise runtime_error;


                         U : M->M''
fun toC (ID id) = (CID id)
|   toC (LAM(id,e)) = cfunf id (toC e)
|   toC(APP(e1,e2)) = CAPP(toC e1, toC e2);

                         T : M'->M''
fun tooC (IID id) = (CID id)
|   tooC (ILAM(id,e)) = cfunf id (tooC e)
|   tooC (IAPP(e1,e2)) = CAPP(tooC e2, tooC e1);

                         V : M->M'
fun Itran (ID id) = (IID id)
|   Itran (APP(e1,e2)) = (IAPP((Itran e2),(Itran e1)))
|   Itran (LAM(id,e)) = (ILAM (id,(Itran e)));
```

6. Testing of the functions

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


                                T : M'->M''
> printCOM (tooC Ivx);
xval it = (): unit
> printCOM (tooC Ivz);
zval it = (): unit
> printCOM (tooC Ivy);
yval it = (): unit
> printCOM (tooC It1);
I''val it = (): unit
> printCOM (tooC It2);
(K'' x)val it = (): unit
```

```
> printCOM (tooC It3);
((I'' (K'' x)) z)val it = (): unit
> printCOM (tooC It4);
(I'' z)val it = (): unit
> printCOM (tooC It5);
(((I'' (K'' x)) z) ((I'' (K'' x)) z))val it = (): unit
> printCOM (tooC It6);
S''val it = (): unit
> printCOM (tooC It7);
((S'' I'') I'')val it = (): unit
> printCOM (tooC It8);
((S'' I'') I'')val it = (): unit
> printCOM (tooC It9);
(((S'' I'') I'') ((I'' (K'' x)) z))val it = (): unit
```

V : M->M'

```
> printIEXP (Itran vx);
xval it = (): unit
> printIEXP (Itran vz);
zval it = (): unit
> printIEXP (Itran vy);
yval 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
```

7. Subterms of $\mathcal{M}''$

```
Subterms (ID) = {ID}
Subterms (S") = {S"}
| (k") = {K"}
| (P"1, P"2 = Subterms (P"1) U Subterms (P"2) U {P"1,P"2}

fun subterms(CID id)=[CID id]
|   subterms(CK)=[CK]
|   subterms(CI)=[CI]
|   subterms(CS)=[CS]
|   subterms(CAPP(e1,e2))=  [CAPP(e1,e2)] @ (subterms e1) @ (subterms e2);

> subterms (Ct1);
val it = [CI]: COM list

> subterms (Ct2);
val it = [CAPP (CK, CID "x"), CK, CID "x"]: COM list

> subterms (Ct3);
val it = [CAPP (CAPP (CI, CAPP (CK, CID "x")), CID "z"),
    CAPP (CI, CAPP (CK, CID "x")), CI, CAPP (CK, CID "x"), CK,
    CID "x",
    CID "z"]: COM list

> subterms (Ct4);
val it = [CAPP (CI, CID "z"), CI, CID "z"]: COM list

> subterms (Ct5);
val it = [CAPP (CAPP (CAPP (CI, CAPP (CK, CID "x")), CID "z"),
     CAPP (CAPP (CI, CAPP (CK, CID "x")), CID "z")),
     CAPP (CAPP (CI, CAPP (CK, CID "x")), CID "z"),
     CAPP (CI, CAPP (CK, CID "x")), CI, CAPP (CK, CID "x"), CK, CID "x",
     CID "z", CAPP (..., ...), CAPP (..., ...), ...]: COM list

> subterms (Ct6);
val it = [CS]: COM list
```

```
> subterms (Ct7);
val it = [CAPP (CAPP (CS, CI), CI), CAPP (CS, CI), CS, CI, CI]: COM list

> subterms (Ct8);
val it = [CAPP (CAPP (CS, CI), CI), CAPP (CS, CI), CS, CI, CI]: COM list

> subterms (Ct9);
val it = [CAPP (CAPP (CAPP (CS, CI), CI),
            CAPP (CAPP (CI, CAPP (CK, CID "x")), CID "z")),
            CAPP (CAPP (CS, CI), CI),
            CAPP (CS, CI), CS, CI, CI,
            CAPP (CAPP (..., ...), CID "z"), CAPP (CI, CAPP (..., ...)), CI,
            CAPP (..., ...), ...]: COM list
```

8. Implement Combinatory reduction rules and test the implementation

```
fun iscredex(CAPP(CI,_))=true
|   iscredex(CAPP(CAPP(CK,_),_))=true
|   iscredex(CAPP(CAPP(CAPP(CS,_),_),_))=true
|   iscredex(_)=false;

fun has_credex (CID id) = false
|   has_credex (CK)= false
|   has_credex (CI)= false
|   has_credex (CS)= false
|   has_credex (CAPP(e1,e2)) = if (iscredex  (CAPP(e1,e2))) then true else
((has_credex e1) orelse (has_credex e2));

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(_) = raise runtime_error;

fun one_creduce (CID id) = (CID id)
|   one_creduce (CAPP(e1,e2)) = if (iscredex (CAPP(e1,e2))) then (cred (CAPP(e1,e2)))
 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 (_) = raise runtime_error;

fun Creduce(CID id)= [(CID id)]
|   Creduce(CK)=[(CK)]
|   Creduce(CI)=[(CI)]
|   Creduce(CS)=[(CS)]
|   Creduce (CAPP(e1,e2)) = (let val l1 = if (iscredex (CAPP(e1,e2)))
 then  (Creduce (cred (CAPP(e1,e2)))) else
 if (has_credex e1) then (Creduce (CAPP(one_creduce e1, e2))) else
 if (has_credex e2) then  (Creduce (CAPP(e1, (one_creduce e2)))) else []
 in [CAPP(e1,e2)]@l1
 end);

fun printlistcreduce [] = ()
|   printlistcreduce (e::[]) = (printCOM e)
|   printlistcreduce (e::l) = (printCOM e; print "-->\n" ; (printlistcreduce l));
```

```
> printlistcreduce (Creduce Cvx);
xval it = (): unit
> printlistcreduce (Creduce Cvz);
zval it = (): unit
> printlistcreduce (Creduce Cvy);
yval it = (): unit
> printlistcreduce (Creduce Ct1);
I''val it = (): unit
> printlistcreduce (Creduce Ct2);
(K'' x)val it = (): unit
> printlistcreduce (Creduce Ct3);
((I'' (K'' x)) z)-->
((K'' x) z)-->
xval it = (): unit
> printlistcreduce (Creduce Ct4);
(I'' z)-->
zval it = (): unit
> printlistcreduce (Creduce 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
> printlistcreduce (Creduce Ct6);
S''val it = (): unit
> printlistcreduce (Creduce Ct7);
((S'' I'') I'')val it = (): unit
> printlistcreduce (Creduce Ct8);
((S'' I'') I'')val it = (): unit
> printlistcreduce (Creduce 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. Implement a counter to count the number of steps to Normal Form

```
fun printSteps(e)= print (Int.toString((length e)-1));

fun printcreduce e = (let val tmp =  (Creduce e)
in (printlistcreduce tmp; print "\nNumber Of Steps: ";printSteps tmp;print"\n") end);

> printcreduce Cvx;
x
Number Of Steps: 0
val it = (): unit

> printcreduce Cvz;
z
Number Of Steps: 0
val it = (): unit

> printcreduce Cvy;
y
Number Of Steps: 0
val it = (): unit

> printcreduce Ct1;
I''
Number Of Steps: 0
val it = (): unit

> printcreduce Ct2;
(K'' x)
Number Of Steps: 0
val it = (): unit

> printcreduce Ct3;
((I'' (K'' x)) z)-->
((K'' x) z)-->
x
Number Of Steps: 2
val it = (): unit
```

```
> printcreduce Ct4;
(I'' z)-->

Number Of Steps: 1
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)
Number Of Steps: 4
val it = (): unit

> printcreduce Ct6;
S''
Number Of Steps: 0
val it = (): unit

> printcreduce Ct7;
((S'' I'') I'')
Number Of Steps: 0
val it = (): unit

> printcreduce Ct8;
((S'' I'') I'')
Number Of Steps: 0
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)
Number Of Steps: 7
val it = (): unit
```

10. Implement eta reduction and test on examples of your own

```
fun printlistreduce [] = ()
|     printlistreduce (e::[]) = (printLEXP e)
|     printlistreduce (e::l) = (printLEXP e; print "-->\n" ; (printlistreduce l));


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);

fun is_eta_redex (LAM(id,(APP( e1, e2)))) =
    if free id e1 then false else (if (ID id) =  e2 then true else false)
|   is_eta_redex _ = false;


fun has_eta_redex (ID id) = false |
    has_eta_redex(APP(e1,e2)) =  (has_eta_redex e1) orelse (has_eta_redex e2) |
    has_eta_redex (LAM(id,e)) = if (is_eta_redex(LAM(id,e)))
    then true else (has_eta_redex e) ;

fun ered (LAM(id,(APP(e1,e2)))) = e1 |
    ered (_) = raise runtime_error;

fun one_eta_reduce (ID id) = (ID id)|
    one_eta_reduce (LAM(id,e)) = if (is_eta_redex (LAM(id,e))) then (ered (LAM(id,e)))
    else LAM(id, (one_eta_reduce e))|
    one_eta_reduce (APP(e1,e2)) =  if (has_eta_redex e1)
    then APP(one_eta_reduce e1, e2) else
    if (has_eta_redex e2) then APP(e1, (one_eta_reduce e2)) else (APP(e1,e2));

fun etareduce (ID id) =  [(ID id)] |
    etareduce (LAM(id,e)) = (let val l1 = if (is_eta_redex (LAM(id,e)))
    then  etareduce (ered (LAM(id,e))) else
 if (has_eta_redex e) then (etareduce (LAM(id, (one_eta_reduce e)))) else []
 in [(LAM(id,e))]@l1  end) |
    etareduce (APP(e1,e2)) = (let val l1 =  if (has_eta_redex e1)
    then (etareduce (APP(one_eta_reduce e1, e2)))
    else if (has_eta_redex e2) then  (etareduce (APP(e1, (one_eta_reduce e2))))
    else [] in [APP(e1,e2)]@l1
   end);
```

```
fun print_eta_reduce e = (let val tmp =  (etareduce e)
    in printlistreduce tmp end);

> print_eta_reduce Et1;
(\x.(y x))-->
yval it = (): unit

> print_eta_reduce Et2;
((\x.(y x)) (\x.(y x)))-->
(y (\x.(y x)))-->
(y y)val it = (): unit

> print_eta_reduce Et3;
(((\x.(y x)) (\x.(y x))) ((\x.(y x)) (\x.(y x))))-->
((y (\x.(y x))) ((\x.(y x)) (\x.(y x))))-->
((y y) ((\x.(y x)) (\x.(y x))))-->
((y y) (y (\x.(y x))))-->
((y y) (y y))val it = (): unit

> print_eta_reduce Et4;
(\x.((\x.(y x)) (\x.(y x))))-->
(\x.(y (\x.(y x))))-->
(\x.(y y))val it = (): unit

> print_eta_reduce Et5;
((\x.x) (\x.(y x)))-->
((\x.x) y)val it = (): unit

> print_eta_reduce Et6;
(\x.(x x))val it = (): unit

> print_eta_reduce Et7;
(\y.((\x.((\z.z) x)) y))-->
(\x.((\z.z) x))-->
(\z.z)val it = (): unit
```

11. Translate $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ into $\mathcal{M}'$, $\mathcal{M}''$, Λ and Λ'

    $\mathcal{M}'$

    $\langle[x]\langle x\rangle x\rangle[x]\langle x\rangle x$

    Λ

    $(\lambda 11)(\lambda 11)$

    Λ'

    $\langle[\ ]\langle 1\rangle 1\rangle[\ ]\langle 1\rangle 1$

    $\mathcal{M}''$

    $S''I''I''S''I''I''$

    Implementation;

```
val oIEXP = (ILAM("x",(IAPP(Ivx, Ivx))));
val oBEXP =(BLAM(BAPP(Bv1, Bv1)));
val oIBEXP = (IBLAM(IBAPP(IBv1, IBv1)));
val oCom =(CAPP(CAPP(CS, CI),(CI)));

val omegaIEXP = (IAPP(oIEXP, oIEXP));
val omegaBEXP =(BAPP(oBEXP, oBEXP));
val omegaIBEXP = (IBAPP (oIBEXP, oIBEXP));
val omegaCom = (CAPP(oCom, oCom));
```

12. running creduce on Omega

    Running creduce on Omega causes an infinite loop in the software you are working in. Although in most coding exercises this would be seen as an indequatie implementation this is the only correct way of showing omega reduction works.

13. Implementation of leftmost and rightmost reduction

```
Reductions  on T1 - T9
> printloreduce t1;
(\x.x)val it = (): unit
> printloreduce t2;
(\y.x)val it = (): unit
> printloreduce t3;
(((\x.x) (\y.x)) z)-->
((\y.x) z)-->
xval it = (): unit
> printloreduce t4;
((\x.x) z)-->
zval it = (): unit
> printloreduce t5;
((((\x.x) (\y.x)) z) (((\x.x) (\y.x)) z))-->
(((\y.x) z) (((\x.x) (\y.x)) z))-->
(x (((\x.x) (\y.x)) z))-->
(x ((\y.x) z))-->
(x x)val it = (): unit
> printloreduce t6;
(\x.(\y.(\z.((x z) (y z)))))val it = (): unit
> printloreduce t7;
(((\x.(\y.(\z.((x z) (y z))))) (\x.x)) (\x.x))-->
((\y.(\z.(((\x.x) z) (y z)))) (\x.x))-->
(\z.(((\x.x) z) ((\x.x) z)))-->
(\z.(z ((\x.x) z)))-->
(\z.(z z))val it = (): unit
> printloreduce t8;
(\z.(z ((\x.x) z)))-->
(\z.(z z))val it = (): unit
> printloreduce t9;
((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))-->
((((\x.x) (\y.x)) z) ((\x.x) (((\x.x) (\y.x)) z)))-->
(((\y.x) z) ((\x.x) (((\x.x) (\y.x)) z)))-->
(x ((\x.x) (((\x.x) (\y.x)) z)))-->
(x (((\x.x) (\y.x)) z))-->
(x ((\y.x) z))-->
(x x)val it = (): unit
```

```
> printrightreduce t1;
(\x.x)val it = (): unit
> printrightreduce t2;
(\y.x)val it = (): unit
> printrightreduce t3;
(((\x.x) (\y.x)) z)-->
((\y.x) z)-->
xval it = (): unit
> printrightreduce t4;
((\x.x) z)-->
zval it = (): unit
> printrightreduce t5;
((((\x.x) (\y.x)) z) (((\x.x) (\y.x)) z))-->
((((\x.x) (\y.x)) z) ((\y.x) z))-->
((((\x.x) (\y.x)) z) x)-->
(((\y.x) z) x)-->
(x x)val it = (): unit
> printrightreduce t6;
(\x.(\y.(\z.((x z) (y z)))))val it = (): unit
> printrightreduce t7;
(((\x.(\y.(\z.((x z) (y z))))) (\x.x)) (\x.x))-->
((\y.(\z.(((\x.x) z) (y z)))) (\x.x))-->
(\z.(((\x.x) z) ((\x.x) z)))-->
(\z.(((\x.x) z) z))-->
(\z.(z z))val it = (): unit
> printrightreduce t8;
(\z.(z ((\x.x) z)))-->
(\z.(z z))val it = (): unit
> printrightreduce t9;
((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))-->
((\z.(z ((\x.x) z))) ((\y.x) z))-->
((\z.(z ((\x.x) z))) x)-->
(x ((\x.x) x))-->
(x x)val it = (): unit
```

```
As seen in the example t9 right side reduction is much more efficient if
there is reduction that can take place on the right hand side otherwise it
will terminate in the same amount of steps as left reduction.
Unlike left reduction in t9 the right reduction clearly shows that right-most
reduction is much more efficient.
However using leftmost reduction will guarantee a reduction path if there
is one.

lr1 = val lr1 = (APP(t2,omegaLEXP));

printloreduce lr1;
((\y.x) ((\x.(x x)) (\x.(x x))))-->
xval it = (): unit

However using printrightreduce lr1 will cause an infinite loop because it trys to
reduce Omega first which ends in the infinite recursive loop.
```