Foundations1 assignment 2017

Fairouz Kamareddine

October 19, 2017

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

Please note that all SML code was defined in separate structures corresponding to the notation they implement. These structures are named `Lambda, Item, Bruijn, ItemBruijn, Combinatorics`.
As such, various function and variable names are repeated across the structures where they are functionally similar, though different in implementation. Consequently code provided in this report may seem contradictory, as functions with the same name produce different output and handle different types.

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)

   The function $V : \mathcal{M} \mapsto \mathcal{M}'$ is defined thus:

   $$V(x) = x \quad | \quad V(\lambda x.y) = [x]V(y) \quad | \quad V(ab) = \langle V(b)\rangle V(a)$$

   The function $\omega' : \mathcal{M}' \mapsto \Lambda'$ is defined thus:

   $$\begin{array}{lll} \omega'(\mathcal{M}') & = \omega'_{[FV(\mathcal{M}')]}(\mathcal{M}') & \\ & & \\ \omega'_{[v_1,\ldots v_n]}(x) & = \min(i : x \equiv v_i \; and \; v_i \in [v_1, \ldots v_n]) & | \\ \omega'_{[v_1,\ldots v_n]}([x]y) & = [\,]\omega'_{[x,\, v_1,\ldots v_n]}(y) & | \\ \omega'_{[v_1,\ldots v_n]}(\langle b\rangle a) & = \langle \omega_{[v_1,\ldots v_n]}(b)\rangle \omega_{[v_1,\ldots v_n]}(a) & \end{array}$$

The function $U : \mathcal{M} \mapsto \mathcal{M}''$ is defined thus:

$$U(x) = x \quad | \quad U(\lambda x.y) = f(x, U(y)) \quad | \quad U(ab) = U(a)U(b)$$

where the function f is defined thus:

$$\begin{array}{lll} f(a,a) & = I'' & | \\ f(a,b) & = K''b \quad where \quad a \notin FV(b) & | \\ f(a,bc) & = b \quad if \quad a \notin FV(b) \quad and \quad a \equiv c & | \\ f(a,bc) & = S''(f(a,b))(f(a,c)) \quad if \quad a \in FV(b) \quad or \quad a \not\equiv c & \end{array}$$

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: (1)

   Table overleaf.

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

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));
```

The implementation in $\mathcal{M}'$:

```
val vx = IID "x";
val vy = IID "y";
val vz = IID "z";

val t1 = ILAM ("x", vx);
val t2 = ILAM ("y", vx);
val t3 = IAPP (vz, IAPP (t2, t1));
val t4 = IAPP (vz, t1);
val t5 = IAPP (t3, t3);
val t6 = ILAM("x", (ILAM ("y", (ILAM ("z", IAPP( IAPP (vz, vy), IAPP
↪  (vz, vx)))))));
val t7 = IAPP( t1, IAPP (t1, t6));
val t8 = ILAM("z", IAPP( IAPP (vz, t1), vz));
val t9 = IAPP (t3, t8);
```

The implementation in $\mathcal{M}''$:

```
val vx = CID "x";
val vy = CID "y";
val vz = CID "z";

val t1 = CI;
val t2 = CAPP (CK, vx);
val t3 = CAPP (CAPP (CI, CAPP (CK, vx)), vz);
val t4 = CAPP (CI, vz);
val t5 = CAPP (CAPP (CAPP (CI, CAPP (CK, vx)), vz), CAPP (CAPP (CI,
↪  CAPP (CK, vx)), vz));
val t6 = CS;
val t7 = CAPP (CAPP (CS, CI), CI);
val t8 = CAPP (CAPP (CS, CI), CI);
val t9 = CAPP (CAPP (CAPP (CS, CI), CI), CAPP (CAPP (CI, CAPP (CK,
↪  vz)), vz));
```

The implementation in Λ:

```
val vx = BID 1;
val vy = BID 1;
val vz = BID 1;

val t1 = BLAM (BID 1);
val t2 = BLAM (BID 2);
val t3 = BAPP (BAPP (t1, t2), BID 2);
val t4 = BAPP (BLAM (BID 1), BID 1);
val t5 = BAPP (BAPP (BAPP ( BLAM (BID 1), BLAM (BID 2)),BID 2), (BAPP
↪  (BAPP ( BLAM (BID 1), BLAM (BID 2)),BID 2)));
val t6 = BLAM (BLAM (BLAM ( BAPP (BAPP (BID 3, BID 2), BAPP (BID 2,
↪  BID 1)))));
val t7 = BAPP (BAPP ( t6, t1), t1);
val t8 = BLAM (BAPP (BID 1,BAPP( BLAM ( BID 1), BID 1)));
val t9 = BAPP (t8, t3);
```

The implementation in Λ':

```
val vx = IBID 1;
val vy = IBID 1;
val vz = IBID 1;

val t1 = IBLAM (IBID 1);
val t2 = IBLAM (IBID 2);
val t3 = IBAPP (IBAPP (t2, t1), IBID 2);
val t4 = IBAPP (IBID 1, IBLAM (IBID 1));
val t5 = IBAPP ((IBAPP (IBID 2,IBAPP (IBLAM (IBID 2), IBLAM (IBID
↪  1)))),IBAPP (IBID 2, IBAPP (IBLAM (IBID 2), IBLAM (IBID 1))));
val t6 = IBLAM (IBLAM (IBLAM ( IBAPP (IBAPP (IBID 1, IBID 2), IBAPP
↪  (IBID 2, IBID 3)))));
val t7 = IBAPP (t1, IBAPP ( t1, t6));
val t8 = IBLAM (IBAPP (IBAPP(IBID 1, IBLAM ( IBID 1)), IBID 1));
val t9 = IBAPP (t3, t8);
```

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
```

In $\mathcal{M}'$ notation:

```
fun printEXP (IID x) = print x
  | printEXP (ILAM (x, y)) = (
        print "[";
        print x;
        print "]";
        printEXP y
        )
  | printEXP (IAPP (a, b)) = (
        print "<";
        printEXP a;
        print ">";
        printEXP b
        );

vx:     x
vy:     y
vz:     z
t1:     [x]x
t2:     [y]x
t3:     <z><[y]x>[x]x
t4:     <z>[x]x
t5:     <<z><[y]x>[x]x><z><[y]x>[x]x
t6:     [x][y][z]<<z>y><z>x
t7:     <[x]x><[x]x>[x][y][z]<<z>y><z>x
t8:     [z]<<z>[x]x>z
t9:     <<z><[y]x>[x]x>[z]<<z>[x]x>z
```

In $\mathcal{M}''$ notation:

```
fun printEXP (CID v) = print v
  | printEXP (CI) = print "I''"
  | printEXP (CS) = print "S''"
  | printEXP (CK) = print "K''"
  | printEXP (CAPP (a, b)) = (
        print "(";
        printEXP a;
        printEXP b;
        print ")"
        );
```

```
vx:     x
vy:     y
vz:     z
t1:     I''
t2:     (K''x)
t3:     ((I''(K''x))z)
t4:     (I''z)
t5:     (((I''(K''x))z)((I''(K''x))z))
t6:     S''
t7:     ((S''I'')I'')
t8:     ((S''I'')I'')
t9:     (((S''I'')I'')((I''(K''z))z))
```

In Λ notation:

```
fun printEXP (BID x) = print (Int.toString x)
  | printEXP (BLAM x) = (
        print "(\206\187";
        printEXP x;
        print ")"
        )
  | printEXP (BAPP (a, b)) = (
        print "(";
        printEXP a;
        printEXP b;
        print ")"
        );
```

```
vx:     1
vy:     1
vz:     1
t1:     (λ1)
t2:     (λ2)
t3:     (((λ1)(λ2))2)
t4:     ((λ1)1)
t5:     ((((λ1)(λ2))2)(((λ1)(λ2))2))
t6:     (λ(λ(λ((32)(21)))))
t7:     (((λ(λ(λ((32)(21)))))(λ1))(λ1))
t8:     (λ(1((λ1)1)))
t9:     ((λ(1((λ1)1)))(((λ1)(λ2))2))
```

In Λ' notation:

```
fun printEXP (IBID x) = print (Int.toString x)
  | printEXP (IBLAM x) = (
        print "[]";
        printEXP x
        )
  | printEXP (IBAPP (a, b)) = (
        print "<";
        printEXP a;
        print ">";
        printEXP b
        );
```

```
vx:     1
vy:     1
vz:     1
t1:     []1
t2:     []2
t3:     <<[]2>[]1>2
t4:     <1>[]1
t5:     <<2><[]2>[]1><2><[]2>[]1
t6:     [][][]<<1>2><2>3
t7:     <[]1><[]1>[][][]<<1>2><2>3
t8:     []<<1>[]1>1
t9:     <<<[]2>[]1>2>[]<<1>[]1>1
```

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

The functions defined for converting from Item and Lambda notations make calls to this function *free* in the Combinatorics structure. Since they are referencing across structures, the function calls use the fully qualified name *Combinatorics.free*.

```
(* Note that SML will type this COM -> COM -> bool, even if every
 ↪  first argument was (CID id). Thus it whines about non exhaustive
 ↪  matches without the double wildcard clause *)
fun free (CID id) (CID x) = (id = x)
| free (CID id) (CAPP (a, b)) = (free (CID id) a) orelse (free (CID
 ↪  id) b)
| free _ _ = false;
```

$T : \mathcal{M}' \mapsto \mathcal{M}''$

```
(* Note that the ordering here is due to the inability to equality
↪  test during patern matching,
*  f (x, x) is erronous in SML. Also due to the fact that any if
↪  statement must have an else clause.
*  Also becuase ( CID x, CID y) and (CID x, COM y) are not mutually
↪  exclusive *)
fun f (CID x, CAPP (a, b)) =
        if (((CID x) = b) andalso (not (Combinatorics.free (CID x)
        ↪  a))) then
                a
        else
                CAPP( CAPP (CS, f ((CID x), a)), f ((CID x), b))
  | f (CID x, a) =
        if ((CID x) = a) then (* Clause 1 *)
                CI
        else (* clause 2 *)
                CAPP (CK, a);

fun toCombinatorics (IID x) = CID x
| toCombinatorics (IAPP (a, b)) = CAPP (toCombinatorics b,
↪  toCombinatorics a)
| toCombinatorics (ILAM (x, a)) = f (CID x, (toCombinatorics a));
```

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

```
(* Note that the ordering here is due to the inability to equality
↪  test during patern matching,
*  f (x, x) is erronous in SML. Also due to the fact that any if
↪  statement must have an else clause.
*  Also becuase ( CID x, CID y) and (CID x, COM y) are not mutually
↪  exclusive *)
fun f (CID x, CAPP (a, b)) =
        if (((CID x) = b) andalso (not (Combinatorics.free (CID x)
        ↪  a))) then
                a
        else
                CAPP( CAPP (CS, f ((CID x), a)), f ((CID x), b))
  | f (CID x, a) =
        if ((CID x) = a) then (* Clause 1 *)
                CI
        else (* clause 2 *)
                CAPP (CK, a);

fun toCombinatorics (ID x) = CID x
  | toCombinatorics (APP (a, b)) = CAPP (toCombinatorics a,
  ↪  toCombinatorics b)
  | toCombinatorics (LAM (x, a)) = f (CID x, (toCombinatorics a));
```

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

```
fun toItem (ID x) = IID x
  | toItem (LAM (x, y)) = ILAM (x, toItem y)
  | toItem (APP (a, b)) = IAPP (toItem b, toItem a);
```

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)

   The following output was generated by the script below, modified in each case to account for the different types being handled

```
val list = [vx, vy, vz, t1, t2, t3, t4, t5, t6, t7, t8, t9];

fun function (hd::[]) = (Combinatorics.printEXP(toCombinatorics hd);
 ↪  print"\n")
  | function (hd::tl) = (Combinatorics.printEXP(toCombinatorics hd);
   ↪  print"\n";function tl);

function list;
```

Converting from $\mathcal{M}$ to $\mathcal{M}'$

```
(* Item.printEXP (Lambda.toItem Lambda.t2); *)
x
y
z
[x]x
[y]x
<z><[y]x>[x]x
<z>[x]x
<<z><[y]x>[x]x><z><[y]x>[x]x
[x][y][z]<<z>y><z>x
<[x]x><[x]x>[x][y][z]<<z>y><z>x
[z]<<z>[x]x>z
<<z><[y]x>[x]x>[z]<<z>[x]x>z
```

Converting from $\mathcal{M}$ to $\mathcal{M}''$

```
(* Combinatorics.printEXP (Lambda.toCombinatorics Lambda.t2); *)
x
y
z
I''
(K''x)
((I''(K''x))z)
(I''z)
(((I''(K''x))z)((I''(K''x))z))
S''
((S''I'')I'')
((S''I'')I'')
(((S''I'')I'')((I''(K''x))z))
```

Converting from $\mathcal{M}'$ to $\mathcal{M}''$

```
(* Combinatorics.printEXP (Item.toCombinatorics Item.t2); *)
x
y
z
I''
(K''x)
((I''(K''x))z)
(I''z)
(((I''(K''x))z)((I''(K''x))z))
S''
((S''I'')I'')
((S''I'')I'')
(((S''I'')I'')((I''(K''x))z))
```

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)

The set of subterms in $\mathcal{M}''$ for any term is defined as follows:

$subterm''(v) = \{v\}$
$subterm''(\mathrm{I}'') = \{\mathrm{I}''\}$
$subterm''(\mathrm{K}'') = \{\mathrm{K}''\}$
$subterm''(\mathrm{S}'') = \{\mathrm{S}''\}$
$subterm''(AB) = \{AB\} \cup subterm(A) \cup subterm(B)$

```
fun subtermsList (CID v) = [CID v]
| subtermsList (CI) = [CI]
| subtermsList (CK) = [CK]
| subtermsList (CS) = [CS]
| subtermsList (CAPP (a, b)) = [CAPP (a, b)]@(subtermsList
↪  a)@(subtermsList b);

(* Remove duplicates from a list *)
fun setify [] = []
| setify (hd::tl) =
if (List.exists (fn x => hd = x) tl) then
setify tl
else
hd::setify tl;

fun subterms x = setify (subtermsList x);

> subterms vx;
val it = [CID "x"]: COM list
> subterms vy;
val it = [CID "y"]: COM list
> subterms vz;
```

```
val it = [CID "z"]: COM list
> subterms t1;
val it = [CI]: COM list
> subterms t2;
val it = [CAPP (CK, CID "x"), CK, CID "x"]: COM list
> subterms t3;
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 t4;
val it = [CAPP (CI, CID "z"), CI, CID "z"]: COM list
> subterms t5;
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"]: COM list
> subterms t6;
val it = [CS]: COM list
> subterms t7;
val it = [CAPP (CAPP (CS, CI), CI), CAPP (CS, CI), CS, CI]: COM list
> subterms t8;
val it = [CAPP (CAPP (CS, CI), CI), CAPP (CS, CI), CS, CI]: COM list
> subterms t9;
val it =
   [CAPP
     (CAPP (CAPP (CS, CI), CI),
      CAPP (CAPP (CI, CAPP (CK, CID "z")), CID "z")),
    CAPP (CAPP (CS, CI), CI), CAPP (CS, CI), CS,
    CAPP (CAPP (CI, CAPP (..., ...)), CID "z"),
    CAPP (CI, CAPP (CK, CID "z")), CI, CAPP (CK, CID "z"), CK, CID
     ↪  "z"]:
   COM list
```

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)

```
> printReduction (reduce vx);
x
val it = (): unit
> printReduction (reduce vy);
y
val it = (): unit
> printReduction (reduce vz);
z
val it = (): unit
> printReduction (reduce t1);
I''
val it = (): unit
> printReduction (reduce t2);
(K''x)
val it = (): unit
> printReduction (reduce t3);
((I''(K''x))z) -->
((K''x)z) -->
x
val it = (): unit
> printReduction (reduce t4);
```

```
(I''z) -->
z
val it = (): unit
> printReduction (reduce t5);
(((I''(K''x))z)((I''(K''x))z)) -->
(((K''x)z)((I''(K''x))z)) -->
(x((I''(K''x))z)) -->
(x((K''x)z)) -->
(xx)
val it = (): unit
> printReduction (reduce t6);
S''
val it = (): unit
> printReduction (reduce t7);
((S''I'')I'')
val it = (): unit
> printReduction (reduce t8);
((S''I'')I'')
val it = (): unit
> printReduction (reduce t9);
(((S''I'')I'')((I''(K''z))z)) -->
((I''((I''(K''z))z))(I''((I''(K''z))z))) -->
(((I''(K''z))z)(I''((I''(K''z))z))) -->
(((I''(K''z))z)((I''(K''z))z)) -->
(((K''z)z)((I''(K''z))z)) -->
(z((I''(K''z))z)) -->
(z((K''z)z)) -->
(zz)
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. 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)

```
> printReductionCount (reduce vx);
x
Steps: 0
val it = (): unit
> printReductionCount (reduce vy);
y
Steps: 0
val it = (): unit
> printReductionCount (reduce vz);
z
Steps: 0
val it = (): unit
> printReductionCount (reduce t1);
I''
Steps: 0
val it = (): unit
> printReductionCount (reduce t2);
(K''x)
Steps: 0
val it = (): unit
```

```
> printReductionCount (reduce t3);
((I''(K''x))z) -->
((K''x)z) -->
x
Steps: 2
val it = (): unit
> printReductionCount (reduce t4);
(I''z) -->
z
Steps: 1
val it = (): unit
> printReductionCount (reduce t5);
(((I''(K''x))z)((I''(K''x))z)) -->
(((K''x)z)((I''(K''x))z)) -->
(x((I''(K''x))z)) -->
(x((K''x)z)) -->
(xx)
Steps: 4
val it = (): unit
> printReductionCount (reduce t6);
S''
Steps: 0
val it = (): unit
> printReductionCount (reduce t7);
((S''I'')I'')
Steps: 0
val it = (): unit
> printReductionCount (reduce t8);
((S''I'')I'')
Steps: 0
val it = (): unit
> printReductionCount (reduce t9);
(((S''I'')I'')((I''(K''z))z)) -->
((I''((I''(K''z))z))(I''((I''(K''z))z))) -->
(((I''(K''z))z)(I''((I''(K''z))z))) -->
(((I''(K''z))z)((I''(K''z))z)) -->
(((K''z)z)((I''(K''z))z)) -->
(z((I''(K''z))z)) -->
(z((K''z)z)) -->
(zz)
Steps: 7
```

```
val it = (): unit
```

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)

```
fun appToList expression [] = []
  | appToList expression (hd::tl) = (APP (expression, hd))::(appToList
  ↪  expression tl);

fun appFromList [] expression = []
  | appFromList (hd::tl) expression = (APP (hd,
  ↪  expression))::(appFromList tl expression);

fun lamToList expression [] = []
  | lamToList expression (hd::tl) = (LAM (expression, hd))::lamToList
  ↪  expression tl;

fun isEtaRedex (LAM (v, (APP (a, ID b)))) = (v = b) andalso (not (free
↪  b a))
  | isEtaRedex _ = false;

fun hasEtaRedex (ID v) = false
  | hasEtaRedex (LAM (v, a)) = (isEtaRedex (LAM (v, a))) orelse
  ↪  (hasEtaRedex a)
  | hasEtaRedex (APP (a, b)) = (hasEtaRedex a) orelse (hasEtaRedex b);

fun etaRed (LAM (v, (APP (a, ID b)))) = a;
```

```
fun etaReduce (ID v) = [ID v]
  | etaReduce (LAM (v, a)) =
     if isEtaRedex (LAM (v, a)) then
        (LAM (v, a))::(etaReduce (etaRed(LAM (v, a))))
     else if hasEtaRedex a then
        (LAM (v, a))::(tl (lamToList v (etaReduce a))) (* tl prevents
        ↪ duplication of this state *)
    else
        [LAM (v, a)]
  | etaReduce (APP (a, b)) =
    if isEtaRedex a then
        (APP (a, b))::(etaReduce(APP((etaRed a), b)))
    else if (hasEtaRedex a) then (* no need to check a is not a redex
    ↪ due to order of statements *)
        let
            val abreduction = (appFromList (etaReduce a) b)
        in      (* Recusrions required to ensure b is reduced if
        ↪ applicable *)
            abreduction@(tl (etaReduce (List.last abreduction))) (* tl
            ↪ applied to remove duplicate of the eta normal of a
            ↪ applied to b *)
        end
    else if isEtaRedex b then
        (APP (a, b))::(etaReduce(APP(a, (etaRed b))))
    else if (hasEtaRedex b) then (* There can be no eta redexes in A
    ↪ at this point, and application cannot be eta reduced *)
        appToList a (etaReduce b) (* Thus no recursive call on AB
        ↪ neccessary, only on b *)
    else (*No redexes *)
        [APP (a, b)];
```

```
> val eta1 = LAM ("x", APP ( t1, ID "x"));
val eta1 = LAM ("x", APP (LAM ("x", ID "x"), ID "x")) : LEXP
> printEtaReduction(etaReduce eta1);
(λx.((λx.x) x)) -->η
(λx.x)
val it = () : unit

> val eta2 = LAM ("z", APP (eta1, ID "z"));
val eta2 =
   LAM ("z", APP (LAM ("x", APP (LAM ("x", ID "x"), ID "x")), ID "z"))
    ↪  : LEXP
> printEtaReduction (etaReduce eta2);
(λz.((λx.((λx.x) x)) z)) -->η
(λx.((λx.x) x)) -->η
(λx.x)
val it = () : unit

 > val eta3 = APP (eta2, eta1);
 val eta3 =
    APP
       (LAM ("z", APP (LAM ("x", APP (LAM ("x", ID "x"), ID "x")), ID
        ↪  "z")),
          LAM ("x", APP (LAM ("x", ID "x"), ID "x"))) : LEXP
 > printEtaReduction (etaReduce eta3);
 ((λz.((λx.((λx.x) x)) z)) (λx.((λx.x) x))) -->η
 ((λx.((λx.x) x)) (λx.((λx.x) x))) -->η
 ((λx.x) (λx.((λx.x) x))) -->η
 ((λx.x) (λx.x))
 val it = () : unit
```

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)

    The following were extracted from their respective structures, hence the repetition of variable name *omega*.

```
val omega = IAPP (ILAM ("x", IAPP (IID "x", IID "x")), ILAM ("x", IAPP
↪  (IID "x", IID "x")));
val omega = CAPP (CAPP (CAPP (CS, CI), CI), CAPP (CAPP (CS, CI), CI));
val omega = BAPP (BLAM (BAPP(BID 1, BID 1)), BLAM (BAPP(BID 1, BID
↪  1)));
val omega = IBAPP (IBLAM (IBAPP(IBID 1, IBID 1)), IBLAM (IBAPP(IBID 1,
↪  IBID 1)));
```

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

    The program rapidly consumes system memory with moderate CPU usage. Presumably if left to continue running the system would either crash, begin paging RAM to the hard drive, or the OS kills the process. Possibly some combination of those. The code fails to self-terminate, as the term cannot be fully reduced.

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)

```
(* appToList, appFromList, lamToList as in Q10 *)
fun isBetaRedex (APP(LAM(_,_),_)) = true
  | isBetaRedex _ = false;

fun hasBetaRedex (LAM(x, y)) = hasBetaRedex y
  | hasBetaRedex (APP(a, b)) = (isBetaRedex (APP(a, b))) orelse
  ↪  hasBetaRedex a orelse hasBetaRedex b (* remember orelse short
  ↪  circuits *)
  | hasBetaRedex _ = false;

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

```
fun leftmost (ID v) = [ID v]
  | leftmost (LAM (v, a)) =
        if hasBetaRedex a then
                (LAM (v, a))::(tl (lamToList v (leftmost a))) (* tl
                ↪ prevents duplication of this state *)
        else
                [LAM (v, a)]
  | leftmost (APP (a, b)) =
        if isBetaRedex (APP (a, b)) then
                (APP (a, b))::(leftmost(betaRed (APP(a, b))))
        else if isBetaRedex a then
                (APP (a, b))::(leftmost(APP((betaRed a), b)))
        else if (hasBetaRedex a) then (* no need to check a is not a
        ↪ redex due to order of statements *)
                let
                        val abreduction = (appFromList (leftmost a) b)
                in
                        abreduction@(tl (leftmost (List.last
                        ↪ abreduction))) (* tl applied to remove
                        ↪ duplicate of the beta normal of a applied
                        ↪ to b *)
                end
        else if isBetaRedex b then
                (APP (a, b))::(leftmost(APP(a, (betaRed b))))
        else if (hasBetaRedex b) then (* changing b cannot create a
        ↪ redex, else AB would be a redex *)
                appToList a (leftmost b) (* Thus no recursive call on
                ↪ AB neccessary *)
        else (*No redexes *)
                [APP (a, b)];
```

```
(* Rightmost reduction, as above but with the application redexes
↪  resolved in a different order *)
fun rightmost (ID v) = [ID v]
  | rightmost (LAM (v, a)) =
        if hasBetaRedex a then
               (LAM (v, a))::(tl (lamToList v (rightmost a))) (* tl
               ↪  prevents duplication of this state *)
       else
               [LAM (v, a)]
  | rightmost (APP (a, b)) =
         if ((not (isBetaRedex b)) andalso (hasBetaRedex b)) then
               let
                       val abreduction = (appToList  a (rightmost b))
               in
                       abreduction@(tl (rightmost (List.last
                       ↪  abreduction))) (* tl applied to remove
                       ↪  duplicate of a applied to the beta normal
                       ↪  form of b *)
               end
       else if isBetaRedex b then
               (APP (a, b))::(rightmost(APP(a, (betaRed b))))
       else if ((not (isBetaRedex a)) andalso (hasBetaRedex a)) then
               let
                       val abreduction = (appFromList (rightmost a)
                       ↪  b)
               in
                       abreduction@(tl (rightmost (List.last
                       ↪  abreduction))) (* tl applied to remove
                       ↪  duplicate of the beta normal of a applied
                       ↪  to b *)
               end
       else if isBetaRedex a then
               (APP (a, b))::(rightmost(APP((betaRed a), b)))
       else if isBetaRedex (APP (a, b)) then
               (APP (a, b))::(rightmost(betaRed (APP(a, b))))
       else (*No redexes *)
               [APP (a, b)];
```

```
> val beta1 = APP (t8, t1);
val beta1 =
   APP
      (LAM ("z", APP (ID "z", APP (LAM ("x", ID "x"), ID "z"))),
         LAM ("x", ID "x")) : LEXP
> printBetaReduction (leftmost beta1);
((λz.(z ((λx.x) z))) (λx.x)) -->β
((λx.x) ((λx.x) (λx.x))) -->β
((λx.x) (λx.x)) -->β
(λx.x)
val it = () : unit
> printBetaReduction (rightmost beta1);
((λz.(z ((λx.x) z))) (λx.x)) -->β
((λz.(z z)) (λx.x)) -->β
((λx.x) (λx.x)) -->β
(λx.x)
val it = () : unit


> val beta2 = APP (LAM ("x", ID "y"), omega);
val beta2 =
   APP
      (LAM ("x", ID "y"),
         APP
            (LAM ("x", APP (ID "x", ID "x")),
               LAM ("x", APP (ID "x", ID "x")))) : LEXP
> printBetaReduction (leftmost beta2);
((λx.y) ((λx.(x x)) (λx.(x x)))) -->β
y
val it = () : unit
> printBetaReduction (rightmost beta2);
Exception- Interrupt raised
(* Terminates only by user interrupt *);


> val beta3 = APP ( LAM ("z", ID "t"), beta1);
val beta3 =
   APP
      (LAM ("z", ID "t"),
         APP
            (LAM ("z", APP (ID "z", APP (LAM ("x", ID "x"), ID "z"))),
```

```
                LAM ("x", ID "x"))) : LEXP
> printBetaReduction (leftmost beta3);
((λz.t) ((λz.(z ((λx.x) z))) (λx.x))) -->β
t
val it = () : unit
> printBetaReduction (rightmost beta3);
((λz.t) ((λz.(z ((λx.x) z))) (λx.x))) -->β
((λz.t) ((λx.x) ((λx.x) (λx.x)))) -->β
((λz.t) ((λx.x) (λx.x))) -->β
((λz.t) (λx.x)) -->β
t
val it = () : unit
```

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
$\mathcal{M}'' ::= \mathcal{V} \mid \mathrm{I}'' \mid \mathrm{K}'' \mid \mathrm{K}'' \mid (\mathcal{M}''\mathcal{M}'')$
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'') $\underline{\mathrm{K}''v_1v_2} =_c v_1$ (K'') $\underline{\mathrm{K}''v_1v_2v_3} =_c v_1v_3(v_2v_3)$.

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:
$FV''(v) = \{v\}$
$FV''(\mathrm{I}'') = FV''(\mathrm{K}'') = FV''(\mathrm{K}'') = \{\}$
$FV''(P''Q'') = FV''(P'') \cup FV''(Q'')$.

† 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;
```