Foundations1 assignment 2019
Coursework 1

Sam Fay-Hunt `sf52@hw.ac.uk`

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)

   - $U : \mathcal{M} \mapsto \mathcal{M}''$:
     $U(v) = v \qquad U(\lambda v.P) = f(v, U(P)) \qquad U(PQ) = (U(P)U(Q))$
     check the function T in the appendix.
   - $V : \mathcal{M} \mapsto \mathcal{M}'$:
     $V(v) = v \qquad V(\lambda v.A) = [v]V(A) \qquad V(AB) = \langle V(B)\rangle V(A).$
   - 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:
     $\omega'_{[x_1,\cdots,x_n]}(v_i) = \min\{j : v_i \equiv x_j\}$
     $\omega'_{[x_1,\cdots,x_n]}(\langle B\rangle A) = \langle \omega'_{[x_1,\cdots,x_n]}(B)\rangle \omega'_{[x_1,\cdots,x_n]}(A)$
     $\omega'_{[x_1,\cdots,x_n]}([x]A) = [] \omega'_{[x_1,\cdots,x_n]}(A)$
     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
     $\omega'([x][y][x']\langle x'\rangle\langle z\rangle x) = \omega'_{[x,y,z]}([x][y][x']\langle x'\rangle\langle z\rangle x) = []\omega_{[x,x,y,z]}([y][x']\langle x'\rangle\langle z\rangle x) = [][]\omega_{[y,x,x,y,z]}([x']\langle x'\rangle\langle z\rangle x) = [][][]\omega_{[x',y,x,x,y,z]}(\langle x'\rangle\langle z\rangle x) = [][][]\langle 1\rangle\langle 6\rangle 3.$

2. For each of the SML terms vx, vy, vz, t1, $\cdots$ t9 in `http://www.macs.hw.ac.uk/~fairouz/foundations-2019/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)

		V
vx	x	x
vy	y	y
vz	z	z
t_1	$\lambda x.x$	$[x]x$
t_2	$\lambda y.x$	$[y]x$
t_3	$(\lambda x.x)(\lambda y.x)z$	$\langle z\rangle\langle[y]x\rangle[x]x$
t_4	$(\lambda x.x)z$	$\langle z\rangle[x]x$
t_5	$(\lambda x.x)(\lambda y.x)z((\lambda x.x)(\lambda y.x)z)$	$\langle\langle z\rangle\langle[y]x\rangle[x]x\rangle\langle z\rangle\langle[y]x\rangle[x]x$
t_6	$\lambda xyz.xz(yz)$	$[x][y][z]\langle\langle z\rangle y\rangle\langle z\rangle x$
t_7	$(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x)$	$\langle[x]x\rangle\langle[x]x\rangle[x][y][z]\langle\langle z\rangle y\rangle\langle z\rangle x$
t_8	$\lambda z.z((\lambda x.x)z)$	$[z]\langle\langle z\rangle[x]x\rangle z$
t_9	$(\lambda z.z((\lambda x.x)z))((\lambda x.x)(\lambda y.x)z)$	$\langle\langle z\rangle\langle[y]x\rangle[x]x\rangle[z]\langle\langle z\rangle[x]x\rangle z$

	U	ω	ω'
vx	x	1	1
vy	y	2	2
vz	z	3	3
t_1	I''	$\lambda 1$	$[\,]1$
t_2	$K''x$	$\lambda 2$	$[\,]2$
t_3	$I''(K''x)z$	$(\lambda 1)(\lambda 2)3$	$\langle 3\rangle\langle[\,]2\rangle\,]1$
t_4	$I''z$	$(\lambda 1)3$	$\langle 3\rangle[\,]1$
t_5	$I''(K''x)z(I''(K''x)z)$	$(\lambda 1)(\lambda 2)3((\lambda 1)(\lambda 2)3)$	$\langle\langle 3\rangle\langle[\,]2\rangle[\,]1\rangle\langle 3\rangle\langle[\,]2\rangle[\,]1$
t_6	S''	$\lambda\lambda\lambda 31(21)$	$[][][]\langle\langle 1\rangle 2\rangle\langle 1\rangle 3$
t_7	$S''I''I''$	$(\lambda\lambda\lambda 31(21))(\lambda 1)\lambda 1$	$\langle[]1\rangle\langle[]1\rangle[][][]\langle\langle 1\rangle 2\rangle\langle 1\rangle 3$
t_8	$S''I''I''$	$\lambda 1((\lambda 1)1)$	$[]\langle\langle 1\rangle[]1\rangle 1$
t_9	$S''I''I''(I''(K''x)z)$	$(\lambda 1((\lambda 1)1))((\lambda 1)(\lambda 2)3)$	$\langle\langle 3\rangle\langle[\,]2\rangle\,]1\rangle[]\langle\langle 1\rangle[]1\rangle 1$

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-2019/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));
```

- ```
  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));
  ```

- ```
  val Bvx = (BID 1);
  val Bvy = (BID 2);
  val Bvz = (BID 3);
  val Bt1 = (BLAM(Bvx));
  val bt2 = (BLAM(bvy));
  val bt3 = (BAPP(BAPP(bt1,bt2),bvz));
  val bt4 = (BAPP(bt1,bvz));
  val bt5 = (BAPP(bt3,bt3));
  val bt6 = (BLAM(BLAM(BLAM(BAPP(BAPP(BID 3,BID 1),(BAPP(BID 2,BID 1)))))));
  val bt7 = (BAPP(BAPP(bt6,bt1),bt1));
  val bt8 = (BLAM(BAPP(BID 1,(BAPP(bt1,BID 1)))));
  val bt9 = (BAPP(bt8,bt3));
  ```

- ```
  val ibvx = (IBID 1);
  val ibvy = (IBID 2);
  val ibvz = (IBID 3);
  val ibt1 = (IBLAM(ibvx));
  val ibt2 = (IBLAM(ibvy));
  val ibt3 = (IBAPP(ibvz, IBAPP(ibt2,ibt1)));
  val ibt4 = (IBAPP(ibvz,ibt1));
  val ibt5 = (IBAPP(ibt3,ibt3));
  val ibt6 = (IBLAM(IBLAM(IBLAM(IBAPP(IBAPP(IBID 1, IBID 2),
                                  IBAPP(IBID 1, IBID 3))))));
  val ibt7 = (IBAPP(ibt1, IBAPP(ibt1, ibt6)));
  val ibt8 = (IBLAM(IBAPP((IBAPP(IBID 1, ibt1)), IBID 1)));
  val ibt9 = (IBAPP(ibt3,ibt8));
  ```

- ```
  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. 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
```

- ```
  (*Prints a term in item lambda calculus*)
  fun printILEXP (IID v) = (print v; print " ") |
      printILEXP (ILAM (v,e)) = (print "[";
                                 print v; print "]";      printILEXP e) |
      printILEXP (IAPP (e1,e2)) = (print "<"; printILEXP e1;
                                   print ">"; printILEXP e2);
  ```

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

  ```
  - printILEXP ivx;
  x val it = () : unit

  - printILEXP ivy;
  y val it = () : unit

  - printILEXP ivz;
  z val it = () : unit

  - printILEXP it1;
  [x]x val it = () : unit

  - printILEXP it2;
  [y]x val it = () : unit

  - printILEXP it3;
  <z ><[y]x >[x]x val it = () : unit

  - printILEXP it4;
  <z >[x]x val it = () : unit

  - printILEXP it5;
  <<z ><[y]x >[x]x ><z ><[y]x >[x]x val it = () : unit

  - printILEXP it6;
  [x][y][z]<<z >y ><z >x val it = () : unit
  ```
  ```

```
- printILEXP it7;
<[x]x ><[x]x >[x][y][z]<<z >y ><z >x val it = () : unit

- printILEXP it8;
[z]<<z >[x]x >z val it = () : unit

- printILEXP it9;
<<z ><[y]x >[x]x >[z]<<z >[x]x >z val it = () : unit
```

- ```
  (*Prints a term in classical lambda calculus with de Bruijn indices*)
  fun printBLEXP (BID v) = print (Int.toString v) |
      printBLEXP (BLAM (e)) = (print "(\\"; printBLEXP(e); print ")") |
      printBLEXP (BAPP (e1,e2)) = (print "("; printBLEXP (e1);
                                   print " "; printBLEXP (e2); print ")");
  ```

Printing these Λ terms yields:

```
- printBLEXP bvx;
1val it = () : unit

- printBLEXP bvy;
2val it = () : unit

- printBLEXP bvz;
3val it = () : unit

- printBLEXP bt1;
(\1)val it = () : unit

- printBLEXP bt2;
(\2)val it = () : unit

- printBLEXP bt3;
(((\1) (\2)) 3)val it = () : unit

- printBLEXP bt4;
((\1) 3)val it = () : unit
```
```

```
- printBLEXP bt5;
((((\1) (\2)) 3) (((\1) (\2)) 3))val it = () : unit

- printBLEXP bt6;
(\(\(\(\((3 1) (2 1)))))val it = () : unit

- printBLEXP bt7;
((((\(\(\(\((3 1) (2 1))))) (\1)) (\1))val it = () : unit

- printBLEXP bt8;
(\(1 ((\1) 1)))val it = () : unit

- printBLEXP bt9;
((\(1 ((\1) 1))) (((\1) (\2)) 3))val it = () : unit
```

- ```
  (*Prints a term in item lambda calculus with de Bruijn indices*)
  fun printIBLEXP (IBID v) = print (Int.toString v) |
      printIBLEXP (IBLAM (e)) = (print "[]"; printIBLEXP(e)) |
      printIBLEXP (IBAPP (e1, e2)) = (print "<"; printIBLEXP(e1); print ">";
                                      printIBLEXP(e2) );
  ```

Printing these Λ' terms yield:

```
- printIBLEXP ibvx;
1val it = () : unit

- printIBLEXP ibvy;
2val it = () : unit

- printIBLEXP ibvz;
3val it = () : unit

- printIBLEXP ibt1;
[]1val it = () : unit

- printIBLEXP ibt2;
[]2val it = () : unit

- printIBLEXP ibt3;
```
```

```
<3><[]2>[]1val it = () : unit

- printIBLEXP ibt4;
<3>[]1val it = () : unit

- printIBLEXP ibt5;
<<3><[]2>[]1><3><[]2>[]1val it = () : unit


- printIBLEXP ibt6;
[][][]<<1>2><1>3val it = () : unit

- printIBLEXP ibt7;
<[]1><[]1>[][][]<<1>2><1>3val it = () : unit

- printIBLEXP ibt8;
[]<<1>[]1>1val it = () : unit

- printIBLEXP ibt9;
<<3><[]2>[]1>[]<<1>[]1>1val it = () : unit
```

- ```
  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 ")");
  ```

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

```
- printCOM cvx;
xval it = () : unit

- printCOM cvy;
yval it = () : unit

- printCOM cvz;
zval 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. Implement in SML the translation functions T, U and V and give these implemented functions here. (2)

   - $V : \mathcal{M} \mapsto \mathcal{M}'$ is implemented as follows:

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

   - $U : \mathcal{M} \mapsto \mathcal{M}''$ is implemented as follows:

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


fun fFun (v1, v2) = if (v1 = v2) then (CI)
                    else if not(Cfree v1 v2) then (CAPP(CK, v2))
                    else if not(Cfree v1 (lhs(v2))) andalso
                                    (v1 = (rhs(v2))) then lhs(v2)
                    else (CAPP(CAPP(CS,
                              fFun(v1, (lhs(v2)))), fFun(v1, (rhs(v2)))));

fun Utran (ID id) = (CID id) |
    Utran (LAM(v,e)) = fFun(CID v, Utran(e)) |
    Utran (APP(e1,e2)) = (CAPP(Utran(e1), Utran(e2)));
```

- $T : \mathcal{M}' \mapsto \mathcal{M}''$ is implemented as follows (we use all the auxilliary functions defined for toC above):

```
(*translates from item notation to combinators*)
fun Ttran (IID id) = (CID id) |
    Ttran (ILAM(v,e)) = fFun(CID v, Ttran(e)) |
    Ttran (IAPP(e1, e2)) = (CAPP(Ttran(e2), Ttran(e1))) ;
```

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)

-
```
- printILEXP (Itran vx);
x val it = () : unit
```

```
- printILEXP (Itran vy);
y val it = () : unit

- printILEXP (Itran vz);
z val it = () : unit

- printILEXP (Itran t1);
[x]x val it = () : unit

- printILEXP (Itran t2);
[y]x val it = () : unit

- printILEXP (Itran t3);
<z ><[y]x >[x]x val it = () : unit

- printILEXP (Itran t4);
<z >[x]x val it = () : unit

- printILEXP (Itran t5);
<<z ><[y]x >[x]x ><z ><[y]x >[x]x val it = () : unit

- printILEXP (Itran t6);
[x][y][z]<<z >y ><z >x val it = () : unit

- printILEXP (Itran t7);
<[x]x ><[x]x >[x][y][z]<<z >y ><z >x val it = () : unit

- printILEXP (Itran t8);
[z]<<z >[x]x >z val it = () : unit

- printILEXP (Itran t9);
<<z ><[y]x >[x]x >[z]<<z >[x]x >z val it = () : unit
```

- ```
  - printCOM (Utran vx);
  xval it = () : unit

  - printCOM (Utran vy);
  yval it = () : unit
  ```
  ```

```
- printCOM (Utran vz);
zval it = () : unit

- printCOM (Utran t1);
I''val it = () : unit

- printCOM (Utran t2);
(K'' x)val it = () : unit

- printCOM (Utran t3);
((I'' (K'' x)) z)val it = () : unit

- printCOM (Utran t4);
(I'' z)val it = () : unit

- printCOM (Utran t5);
(((I'' (K'' x)) z) ((I'' (K'' x)) z))val it = () : unit

- printCOM (Utran t6);
S''val it = () : unit

- printCOM (Utran t7);
((S'' I'') I'')val it = () : unit

- printCOM (Utran t8);
((S'' I'') I'')val it = () : unit

- printCOM (Utran t9);
(((S'' I'') I'') ((I'' (K'' x)) z))val it = () : unit
```

- ```
  - printCOM(Ttran(ivx));
  xval it = () : unit

  - printCOM(Ttran(ivy));
  yval it = () : unit

  - printCOM(Ttran(ivz));
  zval it = () : unit
  ```
  ```

```
- printCOM(Ttran(it1));
I''val it = () : unit

- printCOM(Ttran(it2));
(K'' x)val it = () : unit

- printCOM(Ttran(it3));
((I'' (K'' x)) z)val it = () : unit

- printCOM(Ttran(it4));
(I'' z)val it = () : unit

- printCOM(Ttran(it5));
(((I'' (K'' x)) z) ((I'' (K'' x)) z))val it = () : unit

- printCOM(Ttran(it6));
S''val it = () : unit

- printCOM(Ttran(it7));
((S'' I'') I'')val it = () : unit

- printCOM(Ttran(it8));
((S'' I'') I'')val it = () : unit

- printCOM(Ttran(it9));
(((S'' I'') I'') ((I'' (K'' x)) z))val it = () : unit
```

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
```

(2)

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

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

  fun ClrDup [] = [] |
      ClrDup (h::t) = h :: ClrDup (List.filter (fn x => x <> h) t) ;

  fun setterms t = ClrDup(subterms(t));

  fun PrintCOMlist [] =  print "" |
      PrintCOMlist (h::t)= (printCOM h; print "\n"; PrintCOMlist t);

  fun printlistcomb t = PrintCOMlist(setterms(t));
  ```

- ```
  (* subterm list with duplication *)
  - 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 (CI,CAPP (CK,CID "x")),CID "z"),CAPP (CI,CAPP (CK,CID "x")),CI,
    CAPP (CK,CID "x"),...] : 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 (CI,CAPP (CK,CID "x")),CID "z"),CAPP (CI,CAPP (CK,CID "x")),CI,
   CAPP (CK,CID "x"),CK,CID "x",...] : COM list
```

- `(* subterm list without duplication (setterms) *)`

```
- setterms ct1;
val it = [CI] : COM list

- setterms ct2;
val it = [CAPP (CK,CID "x"),CK,CID "x"] : COM list

- setterms 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

- setterms ct4;
val it = [CAPP (CI,CID "z"),CI,CID "z"] : COM list
```

```
- setterms 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"] : COM list

- setterms ct6;
val it = [CS] : COM list

- setterms ct7;
val it = [CAPP (CAPP (CS,CI),CI),CAPP (CS,CI),CS,CI] : COM list

- setterms ct8;
val it = [CAPP (CAPP (CS,CI),CI),CAPP (CS,CI),CS,CI] : COM list

- setterms 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,
   CAPP (CAPP (CI,CAPP (CK,CID "x")),CID "z"),CAPP (CI,CAPP (CK,CID "x")),
   CAPP (CK,CID "x"),CK,CID "x",CID "z"] : COM list
```

- Printing the list nicely.

```
- printlistcomb(ct1);
I''
val it = () : unit

- printlistcomb(ct2);
(K'' x)
K''
x
val it = () : unit

- printlistcomb(ct3);
((I'' (K'' x)) z)
(I'' (K'' x))
I''
```

```
(K'' x)
K''
x
z
val it = () : unit

- printlistcomb(ct4);
(I'' z)
I''
z
val it = () : unit

- printlistcomb(ct5);
(((I'' (K'' x)) z) ((I'' (K'' x)) z))
((I'' (K'' x)) z)
(I'' (K'' x))
I''
(K'' x)
K''
x
z
val it = () : unit

- printlistcomb(ct6);
S''
val it = () : unit

- printlistcomb(ct7);
((S'' I'') I'')
(S'' I'')
S''
I''
val it = () : unit

- printlistcomb(ct8);
((S'' I'') I'')
(S'' I'')
S''
I''
```

```
val it = () : unit

- printlistcomb(ct9);
(((S'' I'') I'') ((I'' (K'' x)) z))
((S'' I'') I'')
(S'' I'')
S''
I''
((I'' (K'' x)) z)
(I'' (K'' x))
(K'' x)
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, $\cdots$ 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
```

(2)

The functions below are all adapted versions of those in data-files.sml.

```
(*Finds a c-redex*)
fun is_credex (CAPP(CI, _)) = true |
```

```
    is_credex (CAPP(CAPP(CK, _), _)) = true |
    is_credex (CAPP(CAPP(CAPP(CS, _), _), _)) = true |
    is_credex _ = false;

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

fun one_credex (CAPP(CI, e)) = e |
    one_credex (CAPP(CAPP(CK, e1), e2)) = e1 |
    one_credex (CAPP(CAPP(CAPP(CS, e1), e2), e3)) =
                                (CAPP(CAPP(e1, e3), CAPP(e2, e3))) ;

(* adds capp backward to list *)
fun caddbackapp [] e2 = []|
    caddbackapp (e1::l) e2 = (CAPP(e1,e2)):: (caddbackapp l e2);

(* adds capp forward to list *)
fun caddfrontapp e1 [] = []|
    caddfrontapp e1  (e2::l) = (CAPP(e1,e2)):: (caddfrontapp e1 l);

fun cmreduce (CID id) =  [(CID id)] |
    cmreduce (CI) = [(CI)] |
    cmreduce (CK) = [(CK)] |
    cmreduce (CS) = [(CS)] |
    cmreduce (CAPP(e1,e2)) = (let val l1 = (cmreduce e1)
               val l2 = (cmreduce e2)
               val l3 = (caddbackapp l1 e2)
               val l4 = (caddfrontapp (List.last l1) l2)
               val l5 = (List.last l4)
               val l6 =  if (is_credex l5) then (cmreduce (one_credex l5))
                         else [l5]
               in l3 @ l4 @ l6
               end);

 fun Printcredexlist [] =  print "" |
     Printcredexlist (h::nil) = (printCOM h; print "\n") |
     Printcredexlist (h::t)= (printCOM h; print "--> \n"; Printcredexlist t);
```

```
fun credex e = (Printcredexlist(ClrDup(cmreduce(e)));
                print (Int.toString(numCredex(e))); print " steps \n");
```

For tests:

- ```
  - credex ct1;
  I''
  0 steps
  val it = () : unit

  - credex ct2;
  (K'' x)
  0 steps
  val it = () : unit

  - credex ct3;
  ((I'' (K'' x)) z)-->
  ((K'' x) z)-->
  x
  2 steps
  val it = () : unit

  - credex ct4;
  (I'' z)-->
  z
  1 steps
  val it = () : unit

  - credex 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

  - credex ct6;
  S''
  ```
  ```

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

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

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

- credex ct9;
(((S'' I'') I'') ((I'' (K'' x)) z))-->
(((S'' I'') I'') ((K'' x) z))-->
(((S'' I'') I'') x)-->
((I'' x) (I'' x))-->
(x (I'' x))-->
(x x)
5 steps
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,

```
- credex ct1;
I''
0 steps
val it = () : unit
- credex ct2;
(K'' x)
0 steps
val it = () : unit
```

(1)

- `(* count the number of redex' *)`

```
fun numCredex l = List.length (ClrDup(cmreduce(l))) -1 ;

(* print the redex with arrows and a counter at the end *)
fun credex e = (Printcredexlist(ClrDup(cmreduce(e)));
                  print (Int.toString(numCredex(e))); print " steps \n");
```

.....

Tests are as follows:

```
- credex ct1;
I''
0 steps
val it = () : unit

- credex ct2;
(K'' x)
0 steps
val it = () : unit

- credex ct3;
((I'' (K'' x)) z)-->
((K'' x) z)-->
x
2 steps
val it = () : unit

- credex ct4;
(I'' z)-->
z
1 steps
val it = () : unit

- credex 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
```

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

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

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

- credex ct9;
(((S'' I'') I'') ((I'' (K'' x)) z))-->
(((S'' I'') I'') ((K'' x) z))-->
(((S'' I'') I'') x)-->
((I'' x) (I'' x))-->
(x (I'' x))-->
(x x)
5 steps
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 is_eredex (LAM(id, (APP(e1, e2)))) = (if not(free id e1) andalso (ID id = e2)
                                          then true else false) |
    is_eredex (_) = false;

fun has_eredex (ID id) = false |
    has_eredex (APP(e1, e2)) = ((has_eredex(e1)) orelse (has_eredex(e2))) |
    has_eredex (LAM(id, e)) = if (is_eredex(LAM(id, e))) then true else (has_eredex(e)

(* perform eta reduction *)
fun ered (LAM(id, (APP(e1, e2)))) = e1;
```

```
(* find the eta reduction in the term *)
fun one_ereduce (ID id) = (ID id) |
    one_ereduce (APP(e1, e2)) = (APP(one_ereduce(e1), one_ereduce(e2))) |
    one_ereduce (LAM(id, e)) = if (is_eredex(LAM(id, e))) then ered(LAM(id, e))
else if (has_eredex(e)) then one_ereduce(e)
else e;

(* return list of eta reductions of term *)
fun eloreduce (ID id) =  [(ID id)] |
    eloreduce (LAM(id,e)) = if (is_eredex(LAM(id,e))) then ([(LAM(id, e))]
                @(eloreduce (ered(LAM(id,e))))) else (addlam id (eloreduce e)) |
    eloreduce (APP(e1,e2)) = (let val l1 =
                 if (has_eredex e1) then (eloreduce (APP(one_ereduce e1, e2)))
                 else if (has_eredex e2) then
                    (eloreduce (APP(e1, (one_ereduce e2))))
                 else []
                 in [APP(e1,e2)]@l1
      end);

(* prints a list of redex' with arrows *)
fun Printlredexlist [] =  print "" |
    Printlredexlist (h::nil) = (printLEXP h; print "\n") |
    Printlredexlist (h::t)= (printLEXP h; print "--> \n"; Printlredexlist t);

(* prints eta reduction of arg0 *)
fun ereduce e = (Printlredexlist(eloreduce(e)));
```

Tests:

```
- printLEXP t10;
(\x.(y x))val it = () : unit
- ereduce t10;
(\x.(y x))-->
y
val it = () : unit


- printLEXP t11;
((\z.((\x.x) z)) (\x.(y x)))val it = () : unit
```

```
- ereduce t11;
((\z.((\x.x) z)) (\x.(y x)))-->
((\x.x) (\x.(y x)))-->
((\x.x) y)
val it = () : unit


- printLEXP t12;
(\x.((\x.(y x)) ((\z.((\x.x) z)) (\x.(y x)))))val it = () : unit
- ereduce t12;
(\x.((\x.(y x)) ((\z.((\x.x) z)) (\x.(y x)))))-->
(\x.(y ((\z.((\x.x) z)) (\x.(y x)))))-->
(\x.(y ((\x.x) y)))
val it = () : unit


- printLEXP t13;
(\x.((\x.((\x.(y x)) ((\z.((\x.x) z)) (\x.(y x))))) x))val it = () : unit
- ereduce t13;
(\x.((\x.((\x.(y x)) ((\z.((\x.x) z)) (\x.(y x))))) x))-->
(\x.((\x.(y x)) ((\z.((\x.x) z)) (\x.(y x)))))-->
(\x.(y ((\z.((\x.x) z)) (\x.(y x)))))-->
(\x.(y ((\x.x) y)))
val it = () : unit

- printLEXP t14;
((\y.((\x.((\x.((\x.(y x)) ((\z.((\x.x) z)) (\x.(y x))))) x)) y))
                        (\x.(\y.(\z.((x z) (y z))))))val it = () : unit
- ereduce t14;
((\y.((\x.((\x.((\x.(y x)) ((\z.((\x.x) z))
                    (\x.(y x))))) x)) y)) (\x.(\y.(\z.((x z) (y z))))))-->
(((\x.((\x.(y x)) ((\z.((\x.x) z)) (\x.(y x))))) y)
                     (\x.(\y.(\z.((x z) (y z))))))-->
(((y ((\x.x) y)) y) (\x.(\y.(\z.((x z) (y z))))))
val it = () : unit


- printLEXP t15;
(\x.((\x.x) x))val it = () : unit
- ereduce t15;
(\x.((\x.x) x))-->
```

```
(\x.x)
val it = () : unit


- printLEXP t16;
((\x.((\x.x) x)) (\x.((\x.x) x)))val it = () : unit
- ereduce t16;
((\x.((\x.x) x)) (\x.((\x.x) x)))-->
((\x.x) (\x.((\x.x) x)))-->
((\x.x) (\x.x))
val it = () : unit


- printLEXP t17;
(((\x.((\x.x) x)) (\x.((\x.x) x))) ((\x.((\x.x) x))
                        (\x.((\x.x) x))))val it = () : unit
- ereduce t17;
(((\x.((\x.x) x)) (\x.((\x.x) x))) ((\x.((\x.x) x)) (\x.((\x.x) x))))-->
(((\x.x) (\x.x)) ((\x.((\x.x) x)) (\x.((\x.x) x))))-->
(((\x.x) (\x.x)) ((\x.x) (\x.x)))
val it = () : unit

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

11. Give an implementation of leftmost reduction in $\mathcal{M}$ and test it on a number of rich examples. (1)

    - Leftmost is printloreduce in the data-files.sml (and it needs all the necessary functions in that file).

      ```
      (* use given function loreduce() from data files *)

      (* counts number of reduction steps *)
      fun numLredex l = List.length (ClrDup(loreduce(l))) -1 ;

      (* uses helper function from Question 10 to print reductions with
       arrows and counter *)
      ```

```
fun leftreduce e = (Printlredexlist(loreduce(e));
                    print (Int.toString(numLredex(e))); print " steps \n");
```

- Tests

```
- leftreduce vx;
x
0 steps
val it = () : unit

- leftreduce t1;
(\x.x)
0 steps
val it = () : unit

- leftreduce t3;
(((\x.x) (\y.x)) z)-->
((\y.x) z)-->
x
2 steps
val it = () : unit

- leftreduce 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)
4 steps
val it = () : unit

- leftreduce t6;
(\x.(\y.(\z.((x z) (y z)))))
0 steps
val it = () : unit

- leftreduce 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))
4 steps

- leftreduce 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)
6 steps

- leftreduce t17;
(((\x.((\x.x) x)) (\x.((\x.x) x))) ((\x.((\x.x) x)) (\x.((\x.x) x))))-->
(((\x.x) (\x.((\x.x) x))) ((\x.((\x.x) x)) (\x.((\x.x) x))))-->
((\x.((\x.x) x)) ((\x.((\x.x) x)) (\x.((\x.x) x))))-->
((\x.x) ((\x.((\x.x) x)) (\x.((\x.x) x))))-->
((\x.((\x.x) x)) (\x.((\x.x) x)))-->
((\x.x) (\x.((\x.x) x)))-->
(\x.((\x.x) x))-->
(\x.x)
7 steps
```

DATA SHEET

At `http://www.macs.hw.ac.uk/~fairouz/foundations-2019/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 anywhere 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:
  $\underline{(\lambda v.P)Q} \rightarrow_\beta P[v := Q]$.

- 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: $\underline{\langle Q'\rangle[v]}P' \rightarrow_{\beta'} [x := Q']P'$.

- 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:

$$\begin{array}{lcl} \omega_{[x_1,\cdots,x_n]}(v_i) & = & \min\{j : v_i \equiv x_j\} \\ \omega_{[x_1,\cdots,x_n]}(AB) & = & \omega_{[x_1,\cdots,x_n]}(A)\omega_{[x_1,\cdots,x_n]}(B) \\ \omega_{[x_1,\cdots,x_n]}(\lambda x.A) & = & \lambda\omega_{[x,x_1,\cdots,x_n]}(A) \end{array}$$

  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 and de Bruijn indices 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:
  $(\mathrm{I}'')\,\underline{\mathrm{I}''P''} =_c P''$ $\quad$ $(\mathrm{K}'')\,\underline{\mathrm{K}''P''Q''} =_c P''$ $\quad$ $(\mathrm{S}'')\,\underline{\mathrm{S}''P''Q''R''} =_c P''R''(Q''R'')$.
  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;
```