Foundations 1 (F29FA).

Daniyar Nazarbayev.

H00204990.

Task 1.

Common functions and variables.

listorder = [x, y, z, x’, y’, z’, x1, y1, z1, x2, y2, z2, ...]

First_N_Elements(n, l) - gives first n number of elements in list l.

find(n, l) – gives element of index n in the list l.

freeVariables(A) – gives free variables in term A.

BfreeVariables(A) – gives free variables in term A. For de Bruijn notation.

IBfreeVariables(A) – gives free variables in term A. For item de Bruijn notation.

BnumLum, IBnumLum – give lamba count for a term, for B and IB notations respectively.

$\omega_1: \Lambda \to M$

$\omega_1(A) = \omega'_1(n+1, \text{First\_N\_Elements}(n, \text{listorder}), A)$ where n is the largest free variable in A

$\omega'_1(n, l, m) = \text{find}(m, l)$
$\omega'_1(n, l, \lambda exp) = \lambda \text{find}(n, \text{listorder}).\omega'_1(n, \text{find}(n, \text{listorder})::l, exp)$
$\omega'_1(n, l, AB) = \omega'_1(n, l, A)\ \omega'_1(n + \text{number of } \lambda \text{ in } A, l, B)$

* I feel that (n + number of λ in A) is not necessary by the way, and you could just use n instead. It will give correct lambda output regardless. The only purpose it serves is by giving each term a unique identifier.

$\omega_2: M \to \Lambda'$

find_index(element, list) – gives the index of the element in list l.

$\omega_2(A) = \omega'_2(\text{First\_N\_Elements}(n, \text{listorder}), A)$ where n is the largest free variable in A

$\omega'_2(stack, m) = \text{find\_index}(m, stack)$
$\omega'_2(stack, \lambda id.exp) = []\omega'_2(id::stack, exp)$
$\omega'_2(stack, AB) = \langle\omega'_2(stack, B)\rangle\ \omega'_2(stack, A)$

ω_3: $\Lambda' \rightarrow M'$

$\omega_3(A) = \omega'_3(n+1, First\_N\_Elements(n, listorder), A)$ where n is the largest free variable in A

$\omega'_3(n, l, m) = find(m, l)$
$\omega'_3(n, l, \lambda exp) = [find(n, listorder)]\ \omega'_3(n, find(n, listorder)::l, exp)$
$\omega'_3(n, l, AB) = <\omega'_3(n, l, A) >\omega'_3(n + \text{number of } \lambda \text{ in } A, l, B)$

V_1: $M' \rightarrow M$

$V_1(m) = m$

$V_1([id]exp) = \lambda id.V_1(exp)$

$V_1(<A>B) = V_1(B)V_1(A)$

Task 2.

	.	ω
vx	x	1
vy	y	2
vz	z	3
t1	λx.x	λ1
t2	λy.x	λ2
t3	((λx.x)(λy.x))z	((λ1)(λ2))3
t4	(λx.x)z	(λ1)3
t5	(((λx.x)(λy.x))z)(((λx.x)(λy.x))z)	(((λ1)(λ2))3)(((λ1)(λ2))3)
t6	λx.λy.λz.xz(yz)	λλλ31(21)

t7	(((λxyz.xz(yz)) λx.x) λx.x)	(((λλλ31(21)) λ1) λ1)
t8	λz.z((λx.x)z)	λ1((λ1)1)
t9	(λz.z((λx.x)z))(((λx.x) (λy.x))z)	(λ1((λ1)1))(((λ1)(λ2))3)

ω_2
1
2
3
[]1
[]2
<3><[]2>[]1
<3>[]1
<<3><[]2>[]1><3><[]2>[]1
[][][]<<1>2><1>3
<[]1><[]1>[][][]<1<1>2><1>3
[]<<1>[]1>1
<<3><[]2>[]1>[]<1<1>[]1>1

ω_1	ω_3
x	x
y	y
z	z
λx.x	[x]x

λy.x	[y]x
((λx.x)(λy.x))z	<z><[y]x>[x]x
(λx.x)z	<z>[x]x
(((λx.x)(λy.x))z)(((λx.x)(λy.x))z)	<<z><[y]x>[x]x><z><[y]x>[x]x
λx.λy.λz.xz(yz)	[x][y][z]<<z>y><z>x
(((λxyz.xz(yz)) λx.x) λx.x)	<[x]x><[x]x>[x][y][z]<<z>y><z>x
λz.z((λx.x)z)	[z]<<z>[x]x>>z
(λz.z((λx.x)z))(((λx.x)(λy.x))z)	<<z><[y]x>[x]x>[z]<<z>[x]x>z

Task 3.

```
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(BID 2));
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(IBID 2));
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));
```

Part 4.

```
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);

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 ")");

fun printIBEXP (IBID v) =
    print(Int.toString v)
  | printIBEXP (IBLAM e) =
    (print "[]";
```

```
     printIBEXP e)
  | printIBEXP (IBAPP(e1,e2)) =
    (print "<";
     printIBEXP e1;
     print ">";
     printIBEXP e2);

- printBEXP Bvx;
1val it = () : unit

- printBEXP Bvy;
2val it = () : unit

- printBEXP Bvz;
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

- 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

- printIBEXP IBvx;
1val it = () : unit

- printIBEXP IBvy;
2val it = () : unit

- printIBEXP IBvz;
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
```

Task 5.

```
(* M *)
datatype LEXP = APP of LEXP * LEXP | LAM of string * LEXP | ID of string;
(* A - de Bruijn indices *)
datatype BEXP = BAPP of BEXP * BEXP | BLAM of BEXP | BID of int;
(* M' - item notation *)
datatype IEXP = IAPP of IEXP * IEXP | ILAM of string * IEXP | IID of string;
(* A' - de Bruijn indices item notation *)
datatype IBEXP = IBAPP of IBEXP * IBEXP | IBLAM of IBEXP | IBID of int;

(* Task 5 *)

val listorder = ["x","y","z", "x'", "y'", "z'", "x1", "y1", "z1", "x2", "y2", "z2"]

(* M -> M' *)
fun itran (ID id) = (IID id)
        | itran (LAM(id,e)) = ILAM(id,(itran e))
        | itran (APP(e1,e2))= IAPP((itran e2), (itran e1));

(* give the function the index of the element and the list, and it gives back the element *)
fun find (1, l1::l2) = l1
|       find (_, []) = raise Fail "no index like that"
|       find (num : int, l1::l2) = find(num-1, l2);

(*
        input - (1) the value of the element, (2) the list
        output is index of the element in that list
        inverse of find
*)
fun find_int (_ : string, []) = 0
```

```
|	find_int (ch : string, l1::l2) = if ch<>l1 then 1+find_int(ch, l2) else 1;

(*
	free variables function for de Bruijn notation
	usage - BfreeVars(term,0)
	note that you have to set the depth initially to 0
*)
fun BfreeVars ((BID id2), depth: int)       = if id2>depth then [id2-depth] else []
  | BfreeVars ((BAPP(e1,e2)), depth: int)   = BfreeVars(e1, depth) @ BfreeVars(e2, depth)
  | BfreeVars ((BLAM e1), depth: int) = (BfreeVars (e1, depth+1));

fun IBfreeVars ((IBID id2), depth: int)       = if id2>depth then [id2-depth] else []
  | IBfreeVars ((IBAPP(e1,e2)), depth: int)   = IBfreeVars(e1, depth) @ IBfreeVars(e2, depth)
  | IBfreeVars ((IBLAM e1), depth: int) = (IBfreeVars (e1, depth+1));

(*
	return the maximum element in an int list
	usage - requires setting max initially to 0
	ex: getMax(list,0)
*)
fun getMax (l1::l2, max: int) = if l1>max then getMax(l2, l1) else getMax(l2, max)
|	getMax ([], max : int) = max;

val test = BLAM(BAPP(BAPP(BID 1,BID 3),BLAM(BAPP(BLAM(BID 4),BID 1))))
val test2 = (BLAM(BAPP(BAPP(BID 1, BLAM(BAPP(BID 2, BID 1))),BID 3)));
val test3 = BLAM(BAPP(BID 1, BID 2));
val test4 = BLAM (BAPP ( BAPP (BID 1, BLAM (BAPP(BID 2, BID 1))),  BID 3));

(* gives the first n elements of the list *)
fun First_N_Elems (0, l1::l2) = []
|	First_N_Elems (num : int, l1::l2) = [l1] @ First_N_Elems(num-1, l2)
|	First_N_Elems (_, []) = [];

(*
	free variables function for normal lambda calculus syntax
	copied from the data-files.sml
*)
fun freeVars (ID id2)       = [id2]
  | freeVars (APP(e1,e2))   = freeVars e1 @ freeVars e2
  | freeVars (LAM(id2, e1)) = List.filter (fn x => not (x = id2)) (freeVars e1)

fun BnumberLam (BID m) = 0
|	BnumberLam (BAPP(e1,e2)) = (BnumberLam e1) + (BnumberLam e2)
|	BnumberLam (BLAM (e))= 1+ (BnumberLam e);

fun IBnumberLam (IBID m) = 0
|	IBnumberLam (IBAPP(e1,e2)) = (IBnumberLam e1) + (IBnumberLam e2)
|	IBnumberLam (IBLAM (e))= 1+ (IBnumberLam e);
```

```
(*
	translates the list from alphabetic to numeric, according to their order in listorder list
	* had to implement this because i couldn't get List.map to work.
*)
fun translate_list(l1::l2) = find_int(l1, listorder)::translate_list(l2)
|	translate_list([]) = [];
```

Translation functions start here ...

```
(* w: M -> A

usage:

val n = getMax(translate_list(freeVars(term)),0)

omega(First_N_Elems(n,listorder),term);

* replace term if you want to copy and paste
*)

fun omega (stack : string list, ID id) = BID(find_int(id,stack))
	| omega (stack : string list, LAM(id,e)) = BLAM(omega(id::stack, e))
	| omega (stack : string list, APP(e1,e2)) = BAPP((omega(stack, e1)),(omega(stack,e2)));


(* w1: A -> M

usage:

val n = getMax(BfreeVars(term, 0),0)

omega1(n+1,First_N_Elems(n,listorder),term)

* replace term if you want to copy and paste
*)

fun omega1 (n : int, l, (BID id)) = (ID(find(id,l)))
	| omega1 (n : int, l, (BLAM e)) = let val x=find(n,listorder) in LAM(x,omega1(n+1,x::l,e)) end
	| omega1 (n : int, l, (BAPP(e1,e2)))= APP(omega1(n,l,e1),omega1(n+BnumberLam(e1),l,e2));
(*
use "C:\\Users\\daniel_laptop\\Downloads\\foundations 1\\cw_foundations.sml";
val _ = (printIEXP It9); print "\n";

w2: M -> A'

usage:
```

val n = getMax(translate_list(freeVars(term)),0)

omega2(First_N_Elems(n,listorder),term);

* replace term if you want to copy and paste

*)

```
fun omega2 (stack : string list, ID id) = IBID(find_int(id, stack))
	| omega2 (stack : string list, LAM(id,e)) = IBLAM(omega2(id::stack, e))
	| omega2 (stack : string list, APP(e1,e2)) =  IBAPP((omega2(stack, e2)),(omega2(stack,e1)));
```

(* w3: A' -> M'

usage:

val n = getMax(BfreeVars(term, 0),0)

omega3(n+1,First_N_Elems(n,listorder),term)

* replace term if you want to copy and paste

```
*)
fun omega3 (n : int, l, (IBID id)) = (IID(find(id,l)))
	| omega3 (n : int, l, (IBLAM e)) = let val x=find(n,listorder) in ILAM(x,omega3(n+1,x::l,e)) end
	| omega3 (n : int, l, (IBAPP(e1,e2)))= IAPP(omega3 (n, l, e1), omega3 (n+IBnumberLam(e1), l, e2));

(* V1: M' -> M
	the most basic translation function
	just copies the data from one dataset to another.
*)
fun tran (IID id) = (ID id)
	| tran (ILAM(id,e)) = LAM(id,(tran e))
	| tran (IAPP(e1,e2))= APP((tran e2),(tran e1));
```

Task 6.

```
- printIEXP(omega3(getMax(IBfreeVars(IBvx, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBvx, 0),0),listorder),IBvx));
xval it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBvy, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBvy, 0),0),listorder),IBvy));
yval it = () : unit
```

```
- printIEXP(omega3(getMax(IBfreeVars(IBvz, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBvz,
0),0),listorder),IBvz));
zval it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt1, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt1,
0),0),listorder),IBt1));
[x]xval it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt2, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt2,
0),0),listorder),IBt2));
[y]xval it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt3, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt3,
0),0),listorder),IBt3));
<z><[x']x>[y']y'val it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt4, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt4,
0),0),listorder),IBt4));
<z>[x']x'val it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt5, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt5,
0),0),listorder),IBt5));
<<z><[x']x>[y']y'><z><[z']x>[x1]x1val it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt6, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt6,
0),0),listorder),IBt6));
[x][y][z]<<z>y><z>xval it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt7, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt7,
0),0),listorder),IBt7));
<[x]x><[y]y>[z][x'][y']<<y'>x'><y'>zval it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt8, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt8,
0),0),listorder),IBt8));
[x]<<x>[y]y>xval it = () : unit
- printIEXP(omega3(getMax(IBfreeVars(IBt9, 0),0)+1,First_N_Elems(getMax(IBfreeVars(IBt9,
0),0),listorder),IBt9));
<<z><[x']x>[y']y'>[z']<<z'>[x1]x1>z'val it = () : unit
-
- printLEXP(tran(Ivx));
xval it = () : unit
- printLEXP(tran(Ivy));
yval it = () : unit
- printLEXP(tran(Ivz));
zval it = () : unit
- printLEXP(tran(It1));
(\x.x)val it = () : unit
- printLEXP(tran(It2));
(\y.x)val it = () : unit
- printLEXP(tran(It3));
(((\x.x) (\y.x)) z)val it = () : unit
- printLEXP(tran(It4));
((\x.x) z)val it = () : unit
- printLEXP(tran(It5));
((((\x.x) (\y.x)) z) (((\x.x) (\y.x)) z))val it = () : unit
- printLEXP(tran(It6));
(\x.(\y.(\z.((x z) (y z)))))val it = () : unit
```

```
- printLEXP(tran(It7));
(((\x.(\y.(\z.((x z) (y z))))) (\x.x)) (\x.x))val it = () : unit
- printLEXP(tran(It8));
(\z.(z ((\x.x) z)))val it = () : unit
- printLEXP(tran(It9));
((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))val it = () : unit
-
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(vx)),0),listorder),vx));
1val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(vy)),0),listorder),vy));
2val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(vz)),0),listorder),vz));
3val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t1)),0),listorder),t1));
[]1val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t2)),0),listorder),t2));
[]2val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t3)),0),listorder),t3));
<3><[]2>[]1val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t4)),0),listorder),t4));
<3>[]1val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t5)),0),listorder),t5));
<<3><[]2>[]1><3><[]2>[]1val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t6)),0),listorder),t6));
[][][]<<1>2><1>3val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t7)),0),listorder),t7));
<[]1><[]1>[][][]<<1>2><1>3val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t8)),0),listorder),t8));
[]<<1>[]1>1val it = () : unit
- printIBEXP(omega2(First_N_Elems(getMax(translate_list(freeVars(t9)),0),listorder),t9));
<<3><[]2>[]1>[]<<1>[]1>1val it = () : unit
-
- printLEXP(omega1(getMax(BfreeVars(Bvx, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bvx,
0),0),listorder),Bvx));
xval it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bvy, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bvy,
0),0),listorder),Bvy));
yval it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bvz, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bvz,
0),0),listorder),Bvz));
zval it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt1, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt1,
0),0),listorder),Bt1));
(\x.x)val it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt2, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt2,
0),0),listorder),Bt2));
(\y.x)val it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt3, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt3,
0),0),listorder),Bt3));
```

```
(((\x'.x') (\y'.x)) z)val it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt4, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt4,
0),0),listorder),Bt4));
((\x'.x') z)val it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt5, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt5,
0),0),listorder),Bt5));
((((\x'.x') (\y'.x)) z) (((\z'.z') (\x1.x)) z))val it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt6, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt6,
0),0),listorder),Bt6));
(\x.(\y.(\z.((x z) (y z)))))val it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt7, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt7,
0),0),listorder),Bt7));
(((\x.(\y.(\z.((x z) (y z))))) (\x'.x')) (\y'.y'))val it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt8, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt8,
0),0),listorder),Bt8));
(\x.(x ((\y.y) x)))val it = () : unit
- printLEXP(omega1(getMax(BfreeVars(Bt9, 0),0)+1,First_N_Elems(getMax(BfreeVars(Bt9,
0),0),listorder),Bt9));
((\x'.(x' ((\y'.y') x'))) (((\z'.z') (\x1.x)) z))val it = () : unit
-
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(vx)),0),listorder),vx));
1val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(vy)),0),listorder),vy));
2val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(vz)),0),listorder),vz));
3val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t1)),0),listorder),t1));
(\1)val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t2)),0),listorder),t2));
(\2)val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t3)),0),listorder),t3));
(((\1) (\2)) 3)val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t4)),0),listorder),t4));
((\1) 3)val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t5)),0),listorder),t5));
((((\1) (\2)) 3) (((\1) (\2)) 3))val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t6)),0),listorder),t6));
(\(\(\((3 1) (2 1)))))val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t7)),0),listorder),t7));
(((\(\(\((3 1) (2 1))))) (\1)) (\1))val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t8)),0),listorder),t8));
(\(1 ((\1) 1)))val it = () : unit
- printBEXP(omega(First_N_Elems(getMax(translate_list(freeVars(t9)),0),listorder),t9));
((\(1 ((\1) 1))) (((\1) (\2)) 3))val it = () : unit
```

Task 7.

w = (λx.xx)(λ.xx)

Bw = (λ11)(λ11)

Iw = <[x]<x>x>[x]<x>x

Ibw = <[]<1>1>[]<1>1

SML implementations

```
val w = (APP(LAM("x",APP(ID "x",ID "x")),LAM("x",APP(ID "x",ID "x"))));

val Iw = (IAPP(ILAM("x",IAPP(IID "x",IID "x")),ILAM("x",IAPP(IID "x",IID "x"))));

val Bw = (BAPP(BLAM(BAPP(BID 1, BID 1)),BLAM(BAPP(BID 1, BID 1))));

val IBw = (IBAPP(IBLAM(IBAPP(IBID 1, IBID 1)),IBLAM(IBAPP(IBID 1, IBID 1))));
```

Task 8.

```
(*
Solution to task 8 and 9 - the counter

usage - count_reduce(rireduce(term)) or count_reduce(loreduce(term))
 *)

fun count_reduce [] = 0
|	count_reduce (e::[]) = 0
|	count_reduce (e::l) = 1+count_reduce(l);
```

Task 9.

```
(* task 9 duplicates solution *)
fun countprintnewrireduce (ID id) =  [(ID id)] |
   countprintnewrireduce (LAM(id,e)) = (addlam id (countprintnewrireduce e)) |
   countprintnewrireduce (APP(e1,e2)) = (
let
	val l1 = (countprintnewrireduce e2)
	val e3 = (List.last l1)
   val l2 = (addfrontapp e1 l1)
	val e4 = (APP(e1,e3))
   val l3 =  if (is_redex e4) then (countprintnewrireduce (red e4)) else if has_redex(e1)

	then (countprintnewrireduce (APP(one_rireduce e1, e3)))
```

```
        else []
    in l2 @ l3
    end);
```

- count_reduce(rireduce(vx)); val it = 0 : int - count_reduce(rireduce(vy)); val it = 0 : int - count_reduce(rireduce(vz)); val it = 0 : int - count_reduce(rireduce(t1)); val it = 0 : int - count_reduce(rireduce(t2)); val it = 0 : int - count_reduce(rireduce(t3)); val it = 2 : int - count_reduce(rireduce(t4)); val it = 1 : int - count_reduce(rireduce(t5)); val it = 5 : int - count_reduce(rireduce(t6)); **Interrupt** - count_reduce(rireduce(t7)); val it = 5 : int - count_reduce(rireduce(t8)); val it = 2 : int - count_reduce(rireduce(t9)); val it = 5 : int	- count_reduce(loreduce(vx)); val it = 0 : int - count_reduce(loreduce(vy)); val it = 0 : int - count_reduce(loreduce(vz)); val it = 0 : int - count_reduce(loreduce(t1)); val it = 0 : int - count_reduce(loreduce(t2)); val it = 0 : int - count_reduce(loreduce(t3)); val it = 2 : int - count_reduce(loreduce(t4)); val it = 1 : int - count_reduce(loreduce(t5)); val it = 4 : int - count_reduce(loreduce(t6)); val it = 0 : int - count_reduce(loreduce(t7)); val it = 4 : int - count_reduce(loreduce(t8)); val it = 1 : int - count_reduce(loreduce(t9)); val it = 6 : int	- count_reduce(countprintnewrireduce(vx)); val it = 0 : int - count_reduce(countprintnewrireduce(vy)); val it = 0 : int - count_reduce(countprintnewrireduce(vz)); val it = 0 : int - count_reduce(countprintnewrireduce(t1)); val it = 0 : int - count_reduce(countprintnewrireduce(t2)); val it = 0 : int - count_reduce(countprintnewrireduce(t3)); val it = 2 : int - count_reduce(countprintnewrireduce(t4)); val it = 1 : int - count_reduce(countprintnewrireduce(t5)); val it = 4 : int - count_reduce(countprintnewrireduce(t6)); val it = 0 : int - count_reduce(countprintnewrireduce(t7)); val it = 4 : int - count_reduce(countprintnewrireduce(t8)); val it = 1 : int - count_reduce(countprintnewrireduce(t9)); val it = 4 : int

Task 10.

For termination, I suppose something like “(λyz.x)(λx.xx)(λx.xx)” would work.

As for efficiency, I noticed that in the example given there were a lot of bound variables at the left side. With every subsequent left outermost reduction the term would get longer.

example: (λx.xx)(λz.z(λx.x)z)((λx.x)(λy.x))z

I suppose that if we were to even further increase the bound variable count, say like:
(λx.xxxx)((λx.x)x)

```
val x = APP(LAM("x",APP(APP(APP(ID "x",ID "x"),ID "x"),ID "x")), APP(LAM("x",ID "x"),ID
"x"));
- count_reduce(loreduce(x));
val it = 5 : int
```

```
- count_reduce(countprintnewrireduce(x));
val it = 2 : int
-
```