Answers to exercises

Chapter 2

1)

```
i) λa.(a λb.(b a))

<function>
 <bound variable> - a
 <body> - (a λb.(b a))
  <application>
   <function exp> - <name> - a
   <argument exp> - λb.(b a)
    <function>
     <bound variable> - b
     <body> - (b a)
      <application>
       <function exp> - <name> - b
       <argument exp> - <name> - a

ii) λx.λy.λz.((z x) (z y))

<function>
 <bound variable> - x
 <body> λy.λz.((z x) (z y))
  <function>
   <bound variable> - y
   <body> - λz.((z x) (z y))
    <function>
     <bound variable> - z
      <body> - ((z x) (z y))
       <application>
        <function exp> - (z x)
         <application>
          <function exp> - <name> - z
          <argument exp> - <name> - x
        <function exp> - (z y)
         <application>
          <function exp> - <name> - z
          <argument exp> - <name> - y

iii) (λf.λg.(λh.(g h) f) λp.λq.p)

<application>
 <function exp>
  <function> - λf.λg.(λh.(g h) f)
   <bound variable> - f
   <body> - λg.(λh.(g h) f)
    <function>
     <bound variable> - g
     <body> - (λh.(g h) f)
      <application>
       <function exp> - λh.(g h)
        <function>
         <bound variable> - h
```

```
         <body> - (g h)
          <application>
           <function exp> - <name> - g
           <argument exp> - <name> - h
       <argument exp> - <name> f
 <argument exp> - λp.λq.p
  <function>
   <bound variable> - p
   <body> - λq.p
    <function>
     <bound variable> - q
     <body> - <name> - p

iv) λfee.λfi.λfo.λfum.(fum (fo (fi fee)))

<function>
 <bound variable> - fee
 <body> - λfi.λfo.λfum.(fum (fo (fi fee)))
  <function>
   <bound variable> - fi
   <body> - λfo.λfum.(fum (fo (fi fee)))
    <function>
     <bound variable> - fo
     <body> - λfum.(fum (fo (fi fee)))
      <function>
       <bound variable> - fum
       <body> - (fum (fo (fi fee)))
        <application>
         <function exp> - <name> - fum
         <argument exp> - (fo (fi fee))
          <application>
           <function exp> - <name> - fo
           <argument exp> - (fi fee)
            <application>
             <function exp> - <name> - fi
             <argument exp> - <name> - fee

v)   ((λp.(λq.p λx.(x p)) λi.λj.(j i)) λa.λb.(a (a b)))
<application>
 <function exp> - (λp.(λq.p λx.(x p)) λi.λj.(j i))
  <application>
   <function exp> - λp.(λq.p λx.(x p))
    <function>
     <bound variable> - p
     <body> - (λq.p λx.(x p))
      <application>
       <function exp> - λq.p
        <function>
         <bound variable> - q
         <body> - <name> - p
       <argument exp> - λx.(x p)
        <function>
         <bound variable> - x
         <body> - (x p)
          <application>
           <function exp> - <name> - x
           <argument exp> - <name p>
   <argument exp> - λi.λj.(j i)
```

```
     <function>
      <bound variable> - i
      <body> - λj.(j i)
       <function>
        <bound variable> - j
        <body> - (j i)
         <application>
          <function exp> - <name> - j
          <argument exp> - <name> - i
  <argument exp> - λa.λb.(a (a b))
   <function>
    <bound variable> - a
    <body> - λb.(a (a b))
     <function>
      <bound variable> - b
      <body>
       <application>
        <function exp> - <name> - a
        <argument exp> - (a b)
         <application>
          <function exp> - <name> - a
          <argument exp> - <name> - b
```

2)

```
i) ((λx.λy.(y x) λp.λq.p) λi.i) =>
   (λy.(y λp.λq.p) λi.i) =>
   (λi.i λp.λq.p) =>
   λp.λq.p

ii) (((λx.λy.λz.((x y) z) λf.λa.(f a)) λi.i) λj.j) =>
    ((λy.λz.((λf.λa.(f a) y) z) λi.i) λj.j) =>
    (λz.((λf.λa.(f a) λi.i) z) λj.j) =>
    ((λf.λa.(f a) λi.i) λj.j) =>
    (λa.(λi.i a) λj.j) =>
    (λi.i λj.j) =>
    λj.j

iii) (λh.((λa.λf.(f a) h) h) λf.(f f)) =>
     ((λa.λf.(f a) λf.(f f)) λf.(f f)) =>
     (λf.(f λf.(f f)) λf.(f f)) =>
     (λf.(f f) λf.(f f)) =>
     (λf.(f f) λf.(f f)) => ...

iv) ((λp.λq.(p q) (λx.x λa.λb.a)) λk.k) =>
    (λq.((λx.x λa.λb.a) q) λk.k) =>
    ((λx.x λa.λb.a) λk.k) =>
    (λa.λb.a λk.k) =>
    λb.λk.k

v)  (((λf.λg.λx.(f (g x)) λs.(s s)) λa.λb.b) λx.λy.x) =>
    ((λg.λx.(λs.(s s) (g x)) λa.λb.b) λx.λy.x) =>
    (λx.(λs.(s s) (λa.λb.b x)) λx.λy.x) =>
    (λs.(s s) (λa.λb.b λx.λy.x)) =>
    ((λa.λb.b λx.λy.x) (λa.λb.b λx.λy.x)) =>
    (λb.b (λa.λb.b λx.λy.x)) =>
    (λa.λb.b λx.λy.x) =>
```

```
    λb.b
```

3)

```
i)   a) (identity <argument>) => ... =>
        <argument>

     b) ((apply (apply identity)) <argument>) => ... =>
        ((apply identity) <argument>) => ... =>
        (identity <argument>) => ... =>
        <argument>

ii)  a) ((apply <function>) <argument>) => ... =>
        (<function> <argument>)

     b) ((λx.λy.(((make_pair x) y) identity) <function>) <argument>) =>
        (λy.(((make_pair <function>) y) identity) <argument>) =>
        (((make_pair <function>) <argument>) identity) => ... =>
        ((identity <function>) <argument>) => ... =>
        (<function> <argument>)

iii) a) (identity <argument>) => ... =>
        <argument>

     b) ((self_apply (self_apply select_second)) <argument>) => ... =>
        (((self_apply select_second) (self_apply select_second))
         <argument>) => ... =>
        (((select_second select_second) (select_second select_second))
         <argument>) => ... =>
        ((λsecond.second (select_second select_second) <argument>) => ... =>
        ((select_second select_second) <argument>) => ... =>
        (λsecond.second <argument>) =>
        <argument>
```

4)

```
def make_triplet = λfirst.
                    λsecond.
                     λthird.
                      λs.(((s first) second) third)

def triplet_first = λfirst.λsecond.λthird.first

def triplet_second = λfirst.λsecond.λthird.second

def triplet_third = λfirst.λsecond.λthird.third

make_triplet <item1> <item2> <item3> triplet_first ==
λfirst.
 λsecond.
  λthird.
   λs.(((s first) second) third) <item1> <item2> <item3> triplet_first => ... =>
(((triplet_first <item1>) <item2>) <item3>) ==
(((λfirst.λsecond.λthird.first <item1>) <item2>) <item3>) => ... =>
<item1>
```

```
make_triplet <item1> <item2> <item3> triplet_first ==
λfirst.
 λsecond.
  λthird.
   λs.(((s first) second) third) <item1> <item2> <item3> triplet_second => ... =>
(((triplet_second <item1>) <item2>) <item3>) ==
(((λfirst.λsecond.λthird.second <item1>) <item2>) <item3>) => ... =>
<item2>

make_triplet <item1> <item2> <item3> triplet_third ==
λfirst.
 λsecond.
  λthird.
   λs.(((s first) second) third) <item1> <item2> <item3> triplet_third => ... =>
(((triplet_third <item1>) <item2>) <item3>) ==
(((λfirst.λsecond.λthird.third <item1>) <item2>) <item3>) => ... =>
<item3>
```

5)

```
i)   λx.λy.(λx.y λy.x)
x bound at {x} in λx.λy.(λx.y λy.{x})
x free at {x} in λy.(λx.y λy.{x})
                 (λx.y λy.{x})
                 λy.{x}
                 {x}
y bound at {y} in λx.λy.(λx.{y} λy.x)
                  λy.(λx.{y} λy.x)
y free at {y} in (λx.{y} λy.x)
                 λx.{y}
                 {y}

ii)  λx.(x (λy.(λx.x y) x))
x bound at {x} in λx.({x} (λy.(λx.x y) {x}))
x free at {x} in ({x} (λy.(λx.x y) {x}))
              in {x}
              in λy.(λx.x y) {x})
              in {x}
x bound at {x} in λy.(λx.{x} y)
                  (λx.{x} y)
                  λx.{x}
x free at {x} in {x}
y bound at {y} in λx.(x (λy.(λx.x {y}) x))
                  (x (λy.(λx.x {y}) x))
                  (λy.(λx.x {y}) x)
                  λy.(λx.x {y})
y free at {y} in (λx.x {y})
                 {y}

iii) λa.(λb.a λb.(λa.a b))

a bound at {a} in λa.(λb.{a} λb.(λa.a b))
a free at {a} in (λb.{a} λb.(λa.a b))
                 λb.{a}
                 {a}
a bound at {a} in λa.(λb.a λb.(λa.{a} b))
                  (λb.a λb.(λa.{a} b))
```

```
                  λb.(λa.{a} b))
                  λa.{a}
a free at {a} in {a}
b bound at {b} in λa.(λb.a λb.(λa.a {b}))
                  (λb.a λb.(λa.a {b}))
                  λb.(λa.a {b})
b free at {b} in (λa.a {b})
                 {b}

iv)  (λfree.bound λbound.(λfree.free bound))

bound free at {bound} in (λfree.{bound} λbound.(λfree.free bound))
                         λfree.{bound}
                         {bound}
bound bound at {bound} in (λfree.bound λbound.(λfree.free {bound}))
                          λbound.(λfree.free {bound})
bound free at {bound} in  (λfree.free {bound})
                          {bound}
free bound at {free} in (λfree.bound λbound.(λfree.{free} bound))
                        λbound.(λfree.{free} bound)
                        (λfree.{free} bound)
                        λfree.{free}
free free at {free} in {free}

v)   λp.λq.(λr.(p (λq.(λp.(r q)))) (q p))

p bound at {p} in λp.λq.(λr.({p} (λq.(λp.(r q)))) (q {p}))
p free at {p} in λq.(λr.({p} (λq.(λp.(r q)))) (q {p}))
                 (λr.({p} (λq.(λp.(r q)))) (q {p}))
                 λr.({p} (λq.(λp.(r q))))
                 ({p} (λq.(λp.(r q))))
                 {p}
                 (q {p})
                 {p}
q bound at {q} in λp.λq.(λr.(p (λq.(λp.(r q)))) ({q} p))
                  λq.(λr.(p (λq.(λp.(r q)))) ({q} p))
q free at {q} in (λr.(p (λq.(λp.(r q)))) ({q} p))
                 ({q} p))
                 {q}
q bound at {q} in λp.λq.(λr.(p (λq.(λp.(r {q})))) (q p))
                  λq.(λr.(p (λq.(λp.(r {q})))) (q p))
                  (λr.(p (λq.(λp.(r {q})))) (q p))
                  λr.(p (λq.(λp.(r {q}))))
                  (p (λq.(λp.(r {q}))))
                  λq.(λp.(r {q})))
q free at {q} in (λp.(r {q}))
                 (r {q})
                 {q}
r bound at {r} in λp.λq.(λr.(p (λq.(λp.({r} q)))) (q p))
                  λq.(λr.(p (λq.(λp.({r} q)))) (q p))
                  (λr.(p (λq.(λp.({r} q)))) (q p))
                  λr.(p (λq.(λp.({r} q))))
r free at {r} in (p (λq.(λp.({r} q))))
                 λq.(λp.({r} q)))
                 (λp.({r} q))
                 ({r} q)
                 {r}
```

6)

```
i)   λx.λy.(λz.y λa.x)

ii)  λx.(x (λy.(λz.z y) x))

iii) λa.(λb.a λb.(λc.c b))

v)   λp.λq.(λr.(p (λq.(λs.(r q)))) (q p))
```

Chapter 3

1)

```
def implies = λx.λy.(x y true)

implies false false => ... => false false true => ... => true
implies false true => ... => false true true => ... => true
implies true false => ... => true false true => ... => false
implies true true => ... => true true true => ... => true
```

2)

```
def equiv = λx.λy.(x y (not y)

equiv false false => ... => false false (not false) => ... => true
equiv false true => ... => false true (not true) => ... => false
equiv true false => ... => true false (not false) => ... => false
equiv true true => ... => true true (not true) => ... => true
```

3)

```
i)   a) λx.λy.(and (not x) (not y)) false false => ... =>
        and (not false) (not false) => ... =>
        (not false) (not false) false) => ... =>
        true (not false) false => ... =>
        not false => ... => true

        λx.λy.(and (not x) (not y)) false true => ... =>
        and (not false) (not true) => ... =>
        (not false) (not true) false => ... =>
        true (not true) false => ... =>
        not true => ... => false

        λx.λy.(and (not x) (not y)) true false => ... =>
        and (not true) (not false) => ... =>
        (not true) (not false) false => ... =>
        false (not false) false => .. => false

        λx.λy.(and (not x) (not y)) true true => ... =>
        and (not true) (not true)
        (not true) (not true) false => ... =>
        false (not true) false => ... => false

     b) λx.λy.(not (or x y)) false false => ... =>
```

```
        not (or false false) => ... =>
        (or false false) false true => ... =>
        (false true false) false true => ... =>
        false false true => ... => true

        λx.λy.(not (or x y)) false true => ... =>
        not (or false true) => ... =>
        (or false true) false true => ... =>
        (false true true) false true => ... =>
        true false true => ... => false

        λx.λy.(not (or x y)) true false => ... =>
        not (or true false) => ... =>
        (or true false) false true => ... =>
        (true true false) false true => ... =>
        true false true => ... => false

        λx.λy.(not (or x y)) true true => ... =>
        not (or true true) => ... =>
        (or true true) false true => ... =>
        (true true true) false true => ... =>
        true false true => ... => false

ii)  a) - see 1) above

     b) λx.λy.(implies (not y) (not x)) false false => ... =>
        implies (not false) (not false) => ... =>
        (not false) (not false) true => ... =>
        true (not false) true => ... =>
        not false => ... => true

        λx.λy.(implies (not y) (not x)) false true => ... =>
        implies (not true) (not false) => ... =>
        (not true) (not false) true => ... =>
        false (not false) true => ... => true

        λx.λy.(implies (not y) (not x)) true false => ... =>
        implies (not false) (not true)
        (not false) (not true) true => ... =>
        true (not true) true => ... =>
        not true => ... => false

        λx.λy.(implies (not y) (not x)) true true => ... =>
        implies (not true) (not true) => ... =>
        (not true) (not true) true => ... =>
        false (not true) true => ... => true

iii) a) not false => ... => true
        not true => ... => false

     b) λx.(not (not (not x))) false => ... =>
        not (not (not false)) => ... =>
        (not (not false)) false true => ... =>
        ((not false) false true) false true => ... =>
        ((false false true) false true) false true => ... =>
        (true false true) false true => ... =>
        false false true => ... => true
```

```
        λx.(not (not (not x))) true => ... =>
        not (not (not true)) => ... =>
        ((not (not true)) false true) => ... =>
        ((not true) false true) false true => ... =>
        ((true false true) false true) false true => ... =>
        (false false true) false true => ... =>
        true false true => ... => false

iv)  a) - see 1) above

     b) λx.λy.(not (and x (not y))) false false => ... =>
        not (and false (not false)) => ... =>
        (and false (not false)) false true => ... =>
        (false (not false) false) false true) => ... =>
        false false true => ... => true

        λx.λy.(not (and x (not y))) false true => ... =>
        not (and false (not true)) => ... =>
        (and false (not true)) false true => ... =>
        (false (not true) false) false true => ... =>
        false false true => ... => true

        λx.λy.(not (and x (not y))) true false => ... =>
        not (and true (not false)) => ... =>
        (and true (not false)) false true => ... =>
        (true (not false) false) false true => ... =>
        (not false) false true => ... =>
        true false true => ... => false

        λx.λy.(not (and x (not y))) true true => ... =>
        not (and true (not true)) => ... =>
        (and true (not true)) false true => ... =>
        (true (not true) false) false true => ... =>
        (not true) false true => ... =>
        false false true => ... => true

v)   a) - see 2) above

     b) λx.λy.(and (implies x y) (implies y x)) false false => ... =>
        and (implies false false) (implies false false) => ... =>
        (implies false false) (implies false false) false => ... =>
        (false false true) (implies false false) false => ... =>
        true (implies false false) false => ... =>
        implies false false => ... =>
        false false true => ... => true

        λx.λy.(and (implies x y) (implies y x)) false true => ... =>
        and (implies false true) (implies true false)
        (implies false true) (implies true false) false => ... =>
        (false true true) (implies true false) false => ... =>
        true (implies true false) false => ... =>
        implies true false => ... =>
        true false false => ... => false

        λx.λy.(and (implies x y) (implies y x)) true false => ... =>
        and (implies true false) (implies false true)
        (implies true false) (implies false true) false => ... =>
        (true false true) (implies false true) false => ... =>
```

```
false (implies false true) false => ... => false

λx.λy.(and (implies x y) (implies y x)) true true => ... =>
and (implies true true) (implies true true) => ... =>
(implies true true) (implies true true) false => ... =>
(true true true) (implies true true) false => ... =>
true (implies true true) false => ... =>
implies true true => ... =>
true true true => ... => true
```

4)

```
λx.(succ (pred x)) λs.(s false <number>) =>
succ (pred λs.(s false <number>)
Simplifying: pred λs.(s false <number>) => ... =>
             <number>
so: succ <number> => ... =>
    λs.(s false <number>)

λx.(pred (succ x)) λs.(s false <number>) =>
pred (succ λs.(s false number>))
Simplifying: succ λs.(s false <number>) => ... =>
             λs.(s false λs.(s false <number>))
so: pred λs.(s false λs.(s false <number>)) => ... =>
    λs.(s false <number>)

λx.(succ (pred x)) zero =>
(succ (pred zero))
Simplifying: pred zero => ... => zero
so: succ zero ==
    one

λx.(pred (succ x)) zero =>
(pred (succ zero))
Simplifying: succ zero == one
so: pred one => ... =>
    zero
```

Chapter 4

1)

```
sum three => ... =>
recursive sum1 three => ... =>
sum1 (recursive sum1) three => ... =>
add three ((recursive sum1) (pred three)) -> ... ->
add three (sum1 (recursive sum1) two) -> ... ->
add three (add two ((recursive sum1) (pred two))) -> ... ->
add three (add two (sum1 (recursive sum1) one)) -> ... ->
add three (add two (add one ((recursive sum1) (pred one)))) -> ... ->
add three (add two (add one (sum1 (recursive sum1) zero))) -> ... ->
add three (add two (add one zero)) -> ... ->
six
```

2)

```
def prod1 f n =
 if equal n one
 then one
 else mult n (f (pred n))

def prod = recursive prod1

prod three
recursive prod1 three => ... =>
prod1 (recursive prod1) three => ... =>
mult three ((recursive prod1) (pred three)) -> ... ->
mult three (prod1 (recursive prod1) two) -> ... ->
mult three (mult two ((recursive prod1) (pred two))) -> ... ->
mult three (mult two (prod1 (recursive prod1) one)) -> ... ->
mult three (mult two one) -> ... ->
six
```

3)

```
def fun_sum1 f fun n =
 if iszero n
 then fun zero
 else add (fun n) (f fun (pred n))

def fun_sum = recursive fun_sum1

fun_sum double three => ... =>
recursive fun_sum1 double three => ... =>
fun_sum1 (recursive fun_sum1) double three => ... =>
add (double three)
    ((recursive fun_sum1) double (pred three)) -> ... ->
add (double three)
    (fun_sum1 (recursive fun_sum1) double two) -> ... ->
add (double three)
    (add (double two)
         ((recursive fun_sum1) double (pred two))) -> ... ->
add (double three)
    (add (double two)
         (fun_sum1 (recursive fun_sum1) double one)) -> ... ->
add (double three)
    (add (double two)
         (add (double one)
              ((recursive fun_sum1) double (pred one)))) -> ... ->
add (double three)
    (add (double two)
         (add (double one)
              (fun_sum1 (recursive fun_sum1) double zero))) -> ... ->
add (double three)
    (add (double two)
         (add (double one)
              (double zero))) -> ... ->
twelve
```

4)

```
def fun_sum_step1 f fun n s =
 if iszero n
```

```
 then fun n
 else add (fun n) (f fun (sub n s) s)

def fun_sum_step = recursive fun_sum_step1

i)
fun_sum_step double five two => ... =>
recursive fun_sum_step1 double five two => ... =>
fun_sum_step1 (recursive fun_sum_step1) double five two => ... =>
add (double five)
    ((recursive fun_sum_step1) double (sub five two) two) -> ... ->
add (double five)
    (fun_sum_step1 (recursive fun_sum_step1) double three two) -> ... ->
add (double five)
    (add (double three)
     ((recursive fun_sum_step1) double (sub three two) two)) -> ... ->
add (double five)
    (add (double three)
         (fun_sum_step1 (recursive fun_sum_step1) double one two)) -> ... ->
add (double five)
    (add (double three)
         (add (double one)
              ((recursive fun_sum_step1) double (sub one two) two))) -> ... ->
add (double five)
    (add (double three)
         (add (double one)
              (fun_sum_step1 (recursive fun_sum_step1) double zero two))) -> ... ->
add (double five)
    (add (double three)
         (add (double one)
              (double zero))) -> ... ->
eighteen

ii)
fun_sum_step double four two
recursive fun_sum_step1 double four two => ... =>
fun_sum_step1 (recursive fun_sum_step1) double four two => ... =>
add (double four)
    ((recursive fun_sum_step1) double (sub four two) two) -> ... ->
add (double four)
    (fun_sum_step1 (recursive fun_sum_step1) double two two) -> ... ->
add (double four)
    (add (double three)
     ((recursive fun_sum_step1) double (sub two two) two)) -> ... ->
add (double four)
    (add (double two)
         (fun_sum_step1 (recursive fun_sum_step1) double zero two)) -> ... ->
add (double four)
    (add (double two)
         (double zero))
twelve
```

5)

```
def less x y = greater y x

def less_or_equal x y = greater_or_equal y x
```

```
i)   less three two => ... =>
     greater two three => ... =>
     not (iszero (sub two three)) -> ... ->
     not (iszero zero) -> ... ->
     not true => ... => false

ii)  less two three => ... =>
     greater three two -> ... -> true - see 4.8.3

iii) less two two => ... =>
     greater two two => ... =>
     not (iszero (sub two two)) -> ... ->
     not (iszero zero) -> ... ->
     not true => ... => false

iv)  less_or_equal three two => ... =>
     greater_or_equal two three => ... =>
     iszero (sub three two) -> ... ->
     iszero one => ... => false

v)   less_or_equal two three => ... =>
     greater_or_equal three two => ... =>
     iszero (sub two three) -> ... ->
     iszero zero => ... => true

vi)  less_or_equal two two => ... =>
     greater_or_equal two two => ... =>
     iszero (sub two two) -> ... ->
     iszero zero => ... => true
```

6)

```
def mod x y =
 if iszero y
 then x
 else mod1 x y

rec mod1 x y =
 if less x y
 then x
 else mod1 (sub x y) y

i)   mod three two => ... =>
     mod1 three two
     mod1 (sub three two) two -> ... ->
     mod1 one two => ... => one

ii)  mod two three => ... =>
     mod1 two three => ... => two

iii) mod three zero => ... => three
```

Chapter 5

1)

```
i)   ISBOOL 3 => ... =>
     MAKE_BOOL (isbool 3) ==
     MAKE_BOOL (istype bool_type 3) -> ... ->
     MAKE_BOOL (equal (type 3) bool_type) -> ... ->
     MAKE_BOOL (equal numb_type bool_type) -> ... ->
     MAKE_BOOL false ==
     FALSE

ii)  ISNUMB FALSE => ... =>
     MAKE_BOOL (isnumb FALSE) ==
     MAKE_BOOL (istype numb_type FALSE) -> ... ->
     MAKE_BOOL (equal (type FALSE) numb_type) -> ... ->
     MAKE_BOOL (equal bool_type numb_type) -> ... ->
     MAKE_BOOL false ==
     FALSE

iii) NOT 1 => ... =>
     if isbool 1
     then MAKE_BOOL (not (value 1))
     else BOOL_ERROR -> ... ->
     if equal (type 1) bool_type
     then ...
     else ... -> ... ->
     if equal numb_type bool_type
     then ...
     else ... -> ... ->
     if false
     then ...
     else BOOL_ERROR -> ... ->
     BOOL_ERROR

iv)  TRUE AND 2 => ... =>
     if and (isbool TRUE) (isbool 2)
     then MAKE_BOOL (and (value TRUE) (value 2))
     else BOOL_ERROR -> ... ->
     if and (istype bool_type TRUE) (istype bool_type 2)
     then ...
     else ... -> ... ->
     if and (equal (type TRUE) bool_type) (equal (type 2) bool_type)
     then ...
     else ... -> ... ->
     if and (equal bool_type bool_type) (equal numb_type bool_type)
     then ...
     else ... -> ... ->
     if and true false
     then ...
     else ... -> ... ->
     if false
     then ...
     else BOOL_ERROR -> ... ->
     BOOL_ERROR

v)   2 + TRUE => ... =>
     if and (isnumb 2) (isnumb TRUE)
     then MAKE_NUMB (add (value 2) (value TRUE))
     else NUMB_ERROR -> ... ->
     if and (istype numb_type 2) (istype numbtype TRUE)
     then ...
```

```
else ... -> ... ->
if and (equal (type 2) numb_type) (equal (type TRUE) numb_type)
then ...
else ... -> ... ->
if and (equal numb_type numb_type) (equal numb_type numb_type)
then ...
else ... -> ... ->
if and true false
then ...
else NUMB_ERROR -> ... ->
NUMB_ERROR
```

2) i)

```
def issigned N = istype signed_type N

def ISSIGNED N = MAKE_BOOL (issigned N)

def sign = value (select_first (value N))

def SIGN N =
 if issigned N
 then select_first (value N)
 else SIGN_ERROR

def sign_value N = value (select_second (value N))

def VALUE N =
 if issigned N
 then select_second (value N)
 else SIGN_ERROR

def sign_iszero N = iszero (sign_value N)
```

ii)

```
def SIGN_ISZERO N =
 if issigned N
 then MAKE_BOOL (sign_iszero N)
 else SIGN_ERROR

def SIGN_SUCC N =
 IF SIGN_ISZERO N
 THEN +1
 ELSE
  IF SIGN N
  THEN MAKE_SIGNED POS (MAKE_NUMB (succ (sign_value N)))
  ELSE MAKE_SIGNED NEG (MAKE_NUMB (pred (sign_value N)))

def SIGN_PRED N =
 IF SIGN_ISZERO N
 THEN -1
 ELSE
  IF SIGN N
  THEN MAKE_SIGNED POS (MAKE_NUMB (pred (sign_value N)))
  ELSE MAKE_SIGNED NEG (MAKE_NUMB (succ (sign_value N)))
```

iii)

```
def SIGN_+ X Y =
 if and (issigned X) (issigned Y)
 then
  if iszero (sign_value X)
  then Y
  else
   if sign_iszero (sign_value Y)
   then X
   else
    if and (sign X) (sign Y)
    then MAKE_SIGNED POS (MAKE_NUMB (add (sign_value X) (sign_value Y)))
    else
     if and (not (sign X)) (not (sign Y))
     then MAKE_SIGNED NEG (MAKE_NUMB (add (sign_value X) (sign_value Y)))
     else
      if not (sign X)
      then
       if greater (sign_value X) (sign_value Y)
       then MAKE_SIGNED NEG (MAKE_NUMB (sub (sign_value X) (sign_value Y)))
       else MAKE_SIGNED POS (MAKE_NUMB (sub (sign_value Y) (sign_value X)))
      else
       if GREATER (sign_value Y) (sign_value X)
       then MAKE_SIGNED NEG (sub (sign_value Y) (sign_value X))
       else MAKE_SIGNED POS (sub (sign_value X) (sign_value Y))
  else SIGN_ERROR
```

Chapter 6

1)

```
def ATOMCONS A L =
 if isnil L
 then [A]
 else
  if equal (type A) (type (HEAD L))
  then CONS A L
  else LIST_ERROR
```

2)

```
i) rec STARTS [] L = TRUE
    or STARTS L [] = FALSE
    or STARTS (H1::T1) (H2::T2) =
     IF CHAR_EQUALS H1 H2
     THEN STARTS T1 T2
     ELSE FALSE

ii) rec CONTAINS L [] = FALSE
     or CONTAINS L1 L2 =
      IF STARTS L1 L2
      THEN TRUE
      ELSE CONTAINS L1 (TAIL L2)

iii) rec COUNT L [] = 0
      or COUNT L1 L2 =
```

```
       IF STARTS L1 L2
       THEN 1 + (COUNT L1 (TAIL L2))
       ELSE COUNT L1 (TAIL L2)

iv) rec REMOVE [] L = L
     or REMOVE (H1::T1) (H2::T2) = REMOVE T1 T2

v) rec DELETE L [] = []
    or DELETE L1 L2 =
     IF STARTS L1 L2
     THEN REMOVE L1 L2
     ELSE (HEAD L2)::(DELETE L1 (TAIL L2))

vi) rec INSERT L1 L2 [] = []
    or INSERT L1 L2 L3 =
     IF STARTS L2 L3
     THEN APPEND L2 (APPEND L1 (REMOVE L2 L3))
     ELSE (HEAD L3)::(INSERT L1 L2 (TAIL L3))

vii) rec REPLACE L1 L2 [] = []
     or REPLACE L1 L2 L3 =
      IF STARTS L2 L3
      THEN APPEND L1 (REMOVE L2 L3)
      ELSE (HEAD L3)::(REPLACE L1 L2 (TAIL L3))
```

3)

```
i) rec MERGE L [] = L
    or MERGE [] L = L
    or MERGE (H1::T1) (H2::T2) =
     IF LESS H1 H2
     THEN H1::(MERGE T1 (H2::T2))
     ELSE H2::(MERGE (H1::T1) T2)

ii) rec LMERGE [] = []
     or LMERGE (H::T) = MERGE H (LMERGE T)
```

Chapter 7

1)

```
i) def TOO_SECS [H,M,S] = (60 * ((60 * H) + M)) + S

   def MOD X Y = X - ((X / Y) * Y)

   def FROM_SECS S =
    let SECS = MOD S 60
    in
     let MINS = (MOD S 3600) / 60
     in
      let HOURS = S / 3600
      in [HOURS,MINS,SECS]

ii) def TICK [H,M,S] =
     let S = S + 1
     in
      IF LESS S 60
```

```
      THEN [H,M,S]
      ELSE
       let M = M + 1
       in
        IF LESS M 60
        THEN [H,M,0]
        ELSE
         let H = H + 1
         in
          IF LESS H 24
          THEN [H,0,0]
          ELSE [0,0,0]
```

```
iii) def TLESS [TR1,[H1,M1,S1]] [TR2,[H2,M2,S2]] =
      IF LESS H1 H2
      THEN TRUE
      ELSE
       IF EQUAL H1 H2
       THEN
        IF LESS M1 M2
        THEN TRUE
        ELSE
         IF EQUAL M1 M2
         THEN
          IF LESS S1 S2
          THEN TRUE
          ELSE FALSE
         ELSE FALSE
       ELSE FALSE

   rec TINSERT T [] = [T]
    or TINSERT T (T1::R) =
      IF TLESS T T1
      THEN T::T1::R
      ELSE T1::(TINSERT T R)

   rec TSORT [] = []
    or TSORT (H::T) = TINSERT H (TSORT T)
```

2)

```
i) rec TCOMP TEMPTY TEMPTY = TRUE
    or TCOMP TEMPTY T = FALSE
    or TCOMP T TEMPTY = FALSE
    or TCOMP [V1,L1,R1] [V2,L2,R2] =
     IF EQUAL V1 V2
     THEN AND (TCOMP L1 L2) (TCOMP R1 R2)
     ELSE FALSE

ii) rec TFIND TEMPTY T = TRUE
     or TFIND T TEMPTY = FALSE
     or TFIND T1 T2 =
      IF TCOMP T1 T2
      THEN TRUE
      ELSE
       IF LESS (ITEM T1) (ITEM T2)
       THEN TFIND T1 (LEFT T2)
       ELSE TFIND T1 (RIGHT T2)
```

```
iii) rec DTRAVERSE TEMPTY = []
      or DTRAVERSE [V,L,R] = APPEND (DTRAVERSE R) (V::(DTRAVERSE L))
```

3)

```
rec EVAL [E1,OP,E2] =
 let R1 = EVAL E1
 in
  let R2 = EVAL E2
  in
   IF STRING_EQUAL OP "+"
   THEN R1 + R2
   ELSE
    IF STRING_EQUAL OP "-"
    THEN R1 - R2
    ELSE
     IF STRING_EQUAL OP "*"
     THEN R1 * R2
     ELSE R1 / R2
 or EVAL N = N
```

Chapter 8

1)

```
i)
Normal order
λs.(s s) (λf.λa.(f a) λx.x λy.y) =>
(λf.λa.(f a) λx.x λy.y) (λf.λa.(f a) λx.x λy.y) =>
(λa.(λx.x a) λy.y) (λf.λa.(f a) λx.x λy.y) =>
(λx.x λy.y) (λf.λa.(f a) λx.x λy.y) =>
λy.y (λf.λa.(f a) λx.x λy.y) =>
λf.λa.(f a) λx.x λy.y =>
λa.(λx.x a) λy.y =>
λx.x λy.y =>
λy.y

8 reductions
λf.λa.(f a) λx.x λy.y reduced twice

Applicative order
λs.(s s) (λf.λa.(f a) λx.x λy.y) ->
λs.(s s) (λa.(λx.x a) λy.y) ->
λs.(s s) (λx.x λy.y) ->
λs.(s s) λy.y ->
λy.y λy.y ->
λy.y

5 reductions
λf.λa.(f a) λx.x λy.y reduced once

Lazy
λs.(s s) (λf.λa.(f a) λx.x λy.y)_1 =>
(λf.λa.(f a) λx.x λy.y)_1 (λf.λa.(f a) λx.x λy.y)_1 =>
(λa.(λx.x a) λy.y)_2 (λa.(λx.x a) λy.y)_2 =>
(λx.x λy.y)_3 (λx.x λy.y)_3 =>
λy.y λy.y =>
```

```
λy.y

5 reductions
λf.λa.(f a) λx.x λy.y reduced once

ii)
Normal order
λx.λy.x λx.x (λs.(s s) λs.(s s)) =>
λy.λx.x (λs.(s s) λs.(s s)) =>
λx.x

2 reductions
λs.(s s) λs.(s s) not reduced

Applicative order
λx.λy.x λx.x (λs.(s s) λs.(s s)) ->
λx.λy.x λx.x (λs.(s s) λs.(s s)) -> ...

Non-terminating - 1 reduction/cycle
λs.(s s) λs.(s s) reduced every cycle

Lazy
λx.λy.x λx.x (λs.(s s) λs.(s s)) =>
λy.λx.x (λs.(s s) λs.(s s)) =>
λx.x

2 reductions - as normal order

iii)
Normal order
λa.(a a) (λf.λs.(f (s s)) λx.x) =>
(λf.λs.(f (s s)) λx.x) (λf.λs.(f (s s)) λx.x) =>
λs.(λx.x (s s)) (λf.λs.(f (s s)) λx.x) =>
λx.x ((λf.λs.(f (s s)) λx.x) (λf.λs.(f (s s)) λx.x)) =>
(λf.λs.(f (s s)) λx.x) (λf.λs.(f (s s)) λx.x) => ...

Non-terminating - 3 reductions/cycle
λf.λs.(f (s s)) λx.x reduced every cycle

Applicative order
λa.(a a) (λf.λs.(f (s s)) λx.x) ->
λa.(a a) λs.(λx.x (s s)) ->
λs.(λx.x (s s)) λs.(λx.x (s s)) ->
λx.x (λs.(λx.x (s s)) λs.(λx.x (s s))) ->
λs.(λx.x (s s)) λs.(λx.x (s s)) -> ...

Non-terminating - 2 reductions/cycle
λf.λs.(f (s s)) λx.x reduced before non-terminating cycle

Lazy
λa.(a a) (λf.λs.(f (s s)) λx.x)_1 =>
(λf.λs.(f (s s)) λx.x)_1 (λf.λs.(f (s s)) λx.x)_1 =>
λs.(λx.x (s s)) λs.(λx.x (s s)) =>
λx.x (λs.(λx.x (s s)) λs.(λx.x (s s))) =>
λs.(λx.x (s s)) λs.(λx.x (s s)) => ...

Non-terminating - 2 reductions/cycle
λf.λs.(f (s s)) λx.x reduced before non-terminating cycle
```

Chapter 9

1)

```
i) fun cube (y:int) = y*y*y;

ii) fun implies (x:bool) (y:bool) = (not x) orelse y;

iii) fun smallest (a:int) (b:int) (c:int) =
      if a< b
      then
       if a<c
       then a
       else c
      else
       if b<c
       then b
       else c;

iv) fun desc_join (s1:string) (s2:string) =
     if s1<s2
     then s1^s2
     else s2^s1;

v) fun shorter (s1:string) (s2:string) =
    if (size s1) < (size s2)
    then s1
    else s2;
```

2)

```
i) fun sum 0 = 0 |
       sum (n:int) = n+(sum (n-1));

ii) fun nsum (m:int) (n:int) =
     if m>n
     then 0
     else m+(nsum (m+1) n);

iii) fun repeat (s:string) 0 = "" |
         repeat (s:string) (n:int) = s^(repeat s (n-1));
```

3)

```
i) fun ncount [] = 0 |
       ncount ((h::t):int list) =
        if h < 0
        then 1+(ncount t)
        else ncount t;

ii) fun scount (s:string) [] = 0 |
        scount (s:string) ((h::t):string list) =
         if h = s
         then 1+(scount s t)
         else scount s t;

iii) fun gconstr (v:int) [] = [] |
```

```
         gconstr (v:int) ((h::t):int list) =
          if h > v
          then h::(gconstr v t)
          else gconstr v t;

iv) fun smerge [] (s2:string list) = s2 |
        smerge (s1:string list) [] = s1 |
        smerge ((h1::t1):string list) ((h2::t2):string list) =
         if h1 < h2
         then h1::(smerge t1 (h2::t2))
         else h2::(smerge (h1::t1) t2);

v) fun slmerge [] = [] |
       slmerge ((h::t):(string list) list) = smerge h (slmerge t);

vi)
a) type stock = string * int * int;
   fun item (s:string,n:int,r:int) = s;
   fun numb (s:string,n:int,r:int) = n;
   fun reord (s:string,n:int,r:int) = r;

   fun getmore [] = [] |
       getmore ((h::t):stock list) =
        if (numb h) < (reord h)
        then h::(getmore t)
        else getmore t;

b) type upd = string * int;
   fun uitem (s:string,n:int) = s;
   fun unumb (s:string,n:int) = n;

   fun update1 [] (u:upd) = [] |
       update1 ((h::t):stock list) (u:upd) =
        if (item h) = (uitem u)
        then (item h,(numb h)+(unumb u),reord h)::t
        else h::(update1 t u);

   fun update (r:stock list) [] = r |
       update (r:stock list) ((h::t):upd list) = update (update1 r h) t;
```

4)

```
i) fun left1 0 (s:string list) = "" |
       left1 (n:int) [] = "" |
       left1 (n:int) ((h::t):string list) = h^(left1 (n-1) t);

   fun left (n:int) (s:string) = left1 n (explode s);

ii) fun drop 0 (s:string list) = s |
        drop (n:int) [] = [] |
        drop (n:int) ((h::t):string list) = drop (n-1) t;

    fun right (n:int) (s:string) = implode (drop ((size s)-n) (explode s));

iii) fun middle (n:int) (l:int) (s:string) = left1 l (drop (n-1) (explode s));

iv) fun starts [] (s2:string list) = true |
        starts (s1:string list) [] = false |
```

```
        starts ((h1::t1):string list) ((h2::t2):string list) =
         if h1=h2
         then starts t1 t2
         else false;

    fun find1 [] (s2:string list) = 1 |
        find1 (s1:string list) [] = 1 |
        find1 (s1:string list) (s2:string list) =
         if starts s1 s2
         then 1
         else 1+(find1 s1 (tl s2));

    fun find (s1:string) (s2:string) =
     let val pos = find1 (explode s1) (explode s2)
     in
      if pos > (size s2)
      then 0
      else pos
     end;
```

5)

```
fun east Queens_Street = Bishopbriggs |
    east Bishopbriggs = Lenzie |
    east Lenzie = Croy |
    east Croy = Polmont |
    east Polmont = Falkirk_High |
    east Falkirk_High = Linlithgow |
    east Linlithgow = Haymarket |
    east Haymarket = Waverly |
    east Waverly = Waverly;

fun west Queens_Street = Queens_Street |
    west Bishopbriggs = Queens_Street |
    west Lenzie = Bishopbriggs |
    west Croy = Lenzie |
    west Polmont = Croy |
    west Falkirk_High = Polmont |
    west Linlithgow = Falkirk_High |
    west Haymarket = Linlithgow |
    west Waverly = Haymarket;
```

6)

```
fun eval (numb(i:int)) = i |
    eval (add(e1:exp,e2:exp)) = (eval e1)+(eval e2) |
    eval (diff(e1:exp,e2:exp)) = (eval e1)-(eval e2) |
    eval (mult(e1:exp,e2:exp)) = (eval e1)*(eval e2) |
    eval (quot(e1:exp,e2:exp)) = (eval e1) div (eval e2);
```

Chapter 10

1)

```
i) (defun nsum (n)
    (if (eq 0 n)
```

```
        0
        (+ n (nsum (- n 1)))))

ii) (defun nprod (n)
     (if (eq 1 n)
         1
         (* n (nprod (- n 1)))))

iii) (defun napply (fun n)
      (if (eq 0 n)
          (funcall fun 0)
          (+ (funcall fun n) (napply fun (- n 1)))))

iv) (defun nstepapply (fun n s)
     (if (<= n 0)
         (funcall fun 0)
         (+ (funcall fun n) (nstepapply fun (- n s) s))))
```

2)

```
i) (defun lstarts (l1 l2)
    (cond ((null l1) t)
          ((null l2) nil)
          ((eq (car l1) (car l2)) (lstarts (cdr l1) (cdr l2)))
          (t nil)))

ii) (defun lcontains (l1 l2)
     (cond ((null l2) nil)
           ((lstarts l1 l2) t)
           (t (lcontains l1 (cdr l2)))))

iii) (defun lcount (l1 l2)
      (cond ((null l2) 0)
            ((lstarts l1 l2) (+ 1 (lcount l1 (cdr l2))))
            (t (lcount l1 (cdr l2)))))

iv) (defun lremove (l1 l2)
     (if (null l1)
         l2
         (lremove (cdr l1) (cdr l2))))

v) (defun ldelete (l1 l2)
    (cond ((null l2) nil)
          ((lstarts l1 l2) (lremove l1 l2))
          (t (cons (car l2) (ldelete l1 (cdr l2))))))

vi) (defun linsert (l1 l2 l3)
     (cond ((null l3) nil)
           ((lstarts l2 l3) (append l2 (append l1 (lremove l2 l3))))
           (t (cons (car l3) (ilnsert l1 l2 (cdr l3))))))

vii) (defun lreplace (l1 l2 l3)
      (cond ((null l3) nil)
            ((lstarts l1 l3) (append l2 (lremove l1 l3)))
            (t (cons (car l3) (lreplace l1 l2 (cdr l3))))))
```

3)

```
i) (defun merge (l1 l2)
    (cond ((null l1) l2)
          ((null l2) l1)
          ((< (car l1) (car l2)) (cons (car l1) (merge (cdr l1) l2)))
          (t (cons (car l2) (merge l1 (cdr l2))))))

ii) (defun lmerge (l)
     (if (null l)
         nil
         (merge (car l) (lmerge (cdr l)))))
```

4)

```
i) (defun hours (hms) (car hms))

   (defun mins (hms) (car (cdr hms)))

   (defun secs (hms) (car (cdr (cdr hms))))

   (defun too_secs (hms)
    (+ (* 60 (+ (* 60 (hours hms)) (mins hms))) (secs hms)))

   (defun from_secs (s)
    (list (truncate s 3600)
          (truncate (rem s 3600) 60)
          (rem s 60)))

ii) (defun tick (hms)
     (let ((h (hours hms))
           (m (mins hms))
           (s (secs hms)))
           (let ((s1 (+ s 1)))
                (if (< s1 60)
                    (list h m s1)
                    (let ((m1 (+ m 1)))
                         (if (< m1 60)
                             (list h m1 0)
                             (let ((h1 (+ h 1)))
                                  (if (< h1 24)
                                      (list h1 0 0)
                                      (list 0 0 0)))))))))

iii) (defun hms (trans) (car (cdr trans)))

     (defun tless (tr1 tr2)
      (let ((t1 (hms tr1))
            (t2 (hms tr2)))
            (let ((h1 (hours t1))
                  (m1 (mins t1))
                  (s1 (secs t1))
                  (h2 (hours t2))
                  (m2 (mins t2))
                  (s2 (secs t2)))
                 (if (< h1 h2)
                     t
                     (if (= h1 h2)
                         (if (< m1 m2)
                             t
```

```
                              (if (= m1 m2)
                                  (if (< s1 s2)
                                       t
                                       nil)
                                  nil))
                           nil)))))

        (defun tinsert (tr l)
         (cond ((null l) (cons tr nil))
               ((tless tr (car l)) (cons tr l))
               (t (cons (car l) (tinsert tr (cdr l))))))

        (defun tsort (l)
         (if (null l)
             l
             (tinsert (car l) (tsort (cdr l)))))
```

5)

```
    i) (defun tcomp (t1 t2)
        (cond ((and (null t1) (null t2)) t)
              ((or (null t1) (null t2)) nil)
              ((= (item t1) (item t2)) (and (tcomp (left t1) (left t2))
                                            (tcomp (right t1) (right t2))))
              (t nil)))

    ii) (defun tfind (t1 t2)
         (cond ((null t1) t)
               ((null t2) nil)
               ((tcomp t1 t2) t)
               ((< (item t1) (item t2)) (tfind t1 (left t2)))
               (t (tfind t1 (right t2)))))

    iii) (defun dtraverse (tree)
          (if (null tree)
              nil
              (append (dtraverse (right tree))
                      (cons (item tree) (dtraverse (left tree))))))
```