UNIVERSITY OF EDINBURGH

FACULTY OF SCIENCE AND ENGINEERING

COMPUTER SCIENCE THREE

FUNCTIONAL PROGRAMMING AND SPECIFICATION

Wednesday 29 May 2002

09:30 to 11:00

Examiners:

Prof. K. Turner (External)

Prof. C. Stirling (Chairman)

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.

All questions carry equal weight.

1. (a) One way to represent a list `[`a_1`,...,`a_n`]` is as a function taking an integer i, $1 \le i \le n$, to the value a_i and raising an exception for $i < 1$ or $i > n$. Such a representation could be based on the datatype declaration

```
datatype 'a flist = mkflist of int -> 'a
```

with the empty list and a test for the empty list defined as follows:

```
exception OutOfBounds
val fnil = mkflist(fn n => raise OutOfBounds)
fun fnull(mkflist f) = (f 1; false) handle OutOfBounds => true
```

Define the following functions for this representation of lists:

`fcons : 'a * 'a flist -> 'a flist`	like `::`
`fhd : 'a flist -> 'a`	first element of a list
`ftl : 'a flist -> 'a flist`	list of elements after the first
`fmap : ('a -> 'b) -> ('a flist -> 'b flist)`	like `map`

[*8 marks*]

Answer: The exception handling here is slightly fancier than required for a correct answer.

```
exception Empty
fun fcons(a,mkflist f) = mkflist(fn n => if n=1 then a else f(n-1))
fun fhd(mkflist f) = f 1 handle OutOfBounds => raise Empty
fun ftl(mkflist f) = if fnull(mkflist f) then raise Empty
                     else mkflist(fn n => f(n+1))
fun fmap g (mkflist f) = mkflist(g o f)
```

(b) Notice that representing lists in this fashion gives us infinite lists as well as finite lists. For example, the declaration:

```
val ones = mkflist(fn i => 1)
```

corresponds to the infinite list `[1,1,1,...]`.

Explain why the following definition of `append` does not work for infinite lists:

```
fun append(l,l') = if fnull l then l'
                   else fcons(fhd l,append(ftl l,l'))
```

(**Hint:** Consider what happens in the case of `append(ones,ones)`.) Give a definition of `append` that works for both finite and infinite lists. [*8 marks*]

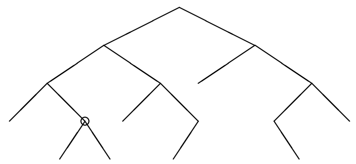
Answer: This definition of `append` doesn't work because SML uses call by value (eager evaluation) which means that it fails to terminate when the first argument is an infinite list. For example:

```
append(ones,ones)
  = fcons(fhd ones,append(ftl ones,ones))
  = fcons(1,fcons(fhd(ftl ones),append(ftl(ftl ones),ones)))
  = fcons(1,fcons(1,fcons(fhd(ftl(ftl ones)),append(ftl(ftl(ftl ones)),ones))))
  = ...
```

Here is a definition of `append` that does work:

```
fun append(l,l') =
    if fnull l then l'
    else mkflist(fn i => if i=1 then fhd l
                         else let val mkflist f = append(ftl l,l')
                              in f(i-1)
                              end)
```

(c) Using a similar idea, define labelled binary trees represented as functions mapping lists of boolean values to node labels, where for example the list `[true,true,false]` is the "address" of the circled node in the diagram below (`[]` is the "address" of the root; `[true,...]` means descend to the left subtree; `[false,...]` means descend to the right subtree).



Define the following constant and functions for this representation of trees:

`empty : 'a tree`	an empty tree
`mkleaf : 'a -> 'a tree`	a tree consisting of one labelled node
`mknode : 'a tree * 'a * 'a tree -> 'a tree`	a labelled node with left and right subtrees
`isempty : 'a tree -> bool`	is the tree empty?
`label : 'a tree -> 'a`	label of the root of a tree
`left : 'a tree -> 'a tree`	left subtree
`right : 'a tree -> 'a tree`	right subtree

[*9 marks*]

Answer: The exception handling here is slightly fancier than required for a correct answer.

```
datatype 'a ftree = mkftree of bool list -> 'a
exception OutOfBounds
exception Empty
val empty = mkftree(fn l => raise OutOfBounds)
fun mkleaf a =
    mkftree(fn [] => a | b::l => raise OutOfBounds)
fun mknode(mkftree f,a,mkftree f') =
    mkftree(fn [] => a | true::l => f l | false::l => f' l)
fun isempty(mkftree f) = (f []; false) handle OutOfBounds => true
fun label(mkftree f) = f [] handle OutOfBounds => raise Empty
fun left(mkftree f) = if isempty(mkftree f) then raise Empty
                      else mkftree(fn l => f(true::l))
fun right(mkftree f) = if isempty(mkftree f) then raise Empty
                       else mkftree(fn l => f(false::l))
```

2. Consider implementing integer arithmetic from scratch. Some operations are more basic than others. For instance:

   addition can be defined in terms of successor ($n \mapsto n+1$), negation and ≤ 0

   subtraction can be defined in terms of negation and addition

   multiplication can be defined in terms of addition, negation, zero, ≤ 0 and successor

   division can be defined in terms of zero, ≤ 0, negation, subtraction and successor

   (a) Design a system of SML modules with dependencies as outlined above that builds an implementation of integer arithmetic starting from a representation of integers with some basic constants and operations like `zero` and `succ`. Give signatures and structure/functor headings *but not structure/functor bodies.* Use separate modules for addition, subtraction, multiplication and division. All modules should have explicit (output) signatures, ascribed either transparently or opaquely. [*15 marks*]

   Answer: Module bodies are not required but I give them anyway; note that this doubles the length of the solution.

```
signature INT =
  sig
    type integer
    val zero : integer
    val succ : integer -> integer
    val neg : integer -> integer
    val leqzero : integer -> bool
  end

structure Int : INT =
  struct
    type integer = int
    val zero = 0
    fun succ(n:int)= n+1
    fun neg(n:int) = ~n
    fun leqzero(n:int) = n<=0
  end

signature ADDITION =
  sig
    structure X : INT
    val plus : X.integer * X.integer -> X.integer
  end

functor Addition(X:INT):ADDITION =
  struct
    structure X=X
    fun plus(n,m) =
      if X.leqzero n andalso X.leqzero(X.neg n) then m
```

```
      else if X.leqzero n then plus(X.succ n,X.neg(X.succ(X.neg m)))
           else plus(X.neg(X.succ(X.neg n)),X.succ m)
  end

signature SUBTRACTION =
  sig
    structure X : INT
    val minus : X.integer * X.integer -> X.integer
  end

functor Subtraction(structure X:INT
                    structure A:ADDITION
                    sharing type X.integer=A.X.integer):SUBTRACTION =
  struct
    structure X=X
    fun minus(n,m) = A.plus(n,X.neg m)
  end

signature MULTIPLICATION =
  sig
    structure X : INT
    val times : X.integer * X.integer -> X.integer
  end

functor Multiplication(structure X:INT
                       structure A:ADDITION
                       sharing type X.integer=A.X.integer):MULTIPLICATION =
  struct
    structure X=X
    fun times(n,m) =
      if X.leqzero n andalso X.leqzero(X.neg n) then X.zero
      else if X.leqzero n then X.neg(times(X.neg n,m))
           else A.plus(m,times(X.neg(X.succ(X.neg n)),m))
  end

signature DIVISION =
  sig
    structure X : INT
    val divide : X.integer * X.integer -> X.integer
  end

functor Division(structure X:INT
                 structure S:SUBTRACTION
                 sharing type X.integer=S.X.integer):DIVISION =
  struct
    structure X=X
    fun divide(n,m) =
      if X.leqzero n andalso X.leqzero(X.neg n) then X.zero
      else if X.leqzero n then X.neg(divide(X.neg n,m))
           else if X.leqzero m then X.neg(divide(n,X.neg m))
                else if not(X.leqzero(S.minus(m,n))) then X.zero
```

```
                     else X.succ(divide(S.minus(n,m),m))
    end
```

(b) Consider the simultaneous use of two different representations of integers. Revise your modules for addition and subtraction to allow an integer represented one way to be added to and subtracted from an integer represented another way. Include the module bodies using versions of the following:

```
fun plus(n,m) =
   if leqzero n andalso leqzero(neg n) then m
   else if leqzero n then plus(succ n,neg(succ(neg m)))
        else plus(neg(succ(neg n)),succ m)

fun minus(n,m) = plus(n,neg m)
```

[*10 marks*]

Answer:

```
signature ADDITION =
  sig
    structure X : INT and Y : INT
    val plus : X.integer * Y.integer -> Y.integer
  end

functor Addition(structure X:INT
                 structure Y:INT):ADDITION =
  struct
    structure X=X and Y=Y
    fun plus(n,m) =
      if X.leqzero n andalso X.leqzero(X.neg n) then m
      else if X.leqzero n then plus(X.succ n,Y.neg(Y.succ(Y.neg m)))
           else plus(X.neg(X.succ(X.neg n)),Y.succ m)
  end

signature SUBTRACTION =
  sig
    structure X : INT and Y : INT
    val minus : X.integer * Y.integer -> Y.integer
  end

functor Subtraction(structure X:INT
                    structure Y:INT
                    structure A:ADDITION
                    sharing type X.integer=A.X.integer
                    sharing type Y.integer=A.Y.integer):SUBTRACTION =
  struct
    structure X=X and Y=Y
    fun minus(n,m) = A.plus(n,Y.neg m)
  end
```

3. (a) Use Extended ML to specify a functor that implements polynomials with coefficients taken from a given type with appropriate addition and multiplication functions, as follows.

The argument to the functor should be a structure containing a type `t` with addition (commutative and associative, with identity `zero`) and multiplication (commutative and associative, with identity `one` and such that $v \times \texttt{zero} = \texttt{zero} = \texttt{zero} \times v$) where multiplication distributes over addition. The result should be a structure implementing polynomials with coefficients of type `t`, and with the following functions:

`eval`	takes a value v of type `t` and a polynomial, and produces the value of the polynomial when the variable takes value v
`plus`	adds two polynomials
`scalartimes`	multiplies a polynomial by a value of type `t`
`times`	multiplies two polynomials
`create`	turns a list of coefficients $[a_0, a_1, \ldots, a_n]$ into the polynomial $a_0 + a_1 x + \cdots + a_n x^n$.

The axioms specifying `plus`, `scalartimes` and `times` should require that, for all values v and a of type `t` and polynomials p and p',

$$\begin{aligned}\texttt{eval}(v,p) + \texttt{eval}(v,p') &= \texttt{eval}(v,\texttt{plus}(p,p'))\\ a \times \texttt{eval}(v,p) &= \texttt{eval}(v,\texttt{scalartimes}(a,p))\\ \texttt{eval}(v,p) \times \texttt{eval}(v,p') &= \texttt{eval}(v,\texttt{times}(p,p'))\end{aligned}$$

You need not write the axioms in the argument signature of your functor. [*7 marks*]

Answer: Axioms are not required in the argument signature but I include them anyway.

```
signature RING =
  sig
    type t
    val plus : t * t -> t
    axiom forall (a,b) => plus(a,b) == plus(b,a)
    axiom forall (a,b,c) => plus(a,plus(b,c)) == plus(plus(a,b),c)
    val zero : t
    axiom forall a => plus(a,zero) == a andalso plus(zero,a) == a
    val times : t * t -> t
    axiom forall (a,b) => times(a,b) == times(b,a)
    axiom forall (a,b,c) => times(a,times(b,c)) == times(times(a,b),c)
    val one : t
    axiom forall a => times(a,one) == a andalso times(one,a) == a
    axiom forall a => times(a,zero) == zero andalso times(zero,a) == zero
    axiom forall (a,b,c) => times(a,plus(b,c)) == plus(times(a,b),times(a,c))
    axiom forall (a,b,c) => times(plus(b,c),a) == plus(times(b,a),times(c,a))
end

signature POLY =
  sig
```

```
    structure R:RING
    type poly
    val create : R.t list -> poly
    val eval : R.t * poly -> R.t
    axiom forall v => eval(v,create []) == R.zero
    axiom forall (v,a,l) =>
              eval(v,create(a::l)) == R.plus(a,R.times(v,eval(v,create l)))
    val plus : poly * poly -> poly
    val scalartimes : R.t * poly -> poly
    val times : poly * poly -> poly
    axiom forall (v,p,p') => R.plus(eval(v,p),eval(v,p')) == eval(v,plus(p,p'))
    axiom forall (a,v,p) => R.times(a,eval(v,p)) == eval(v,scalartimes(a,p))
    axiom forall (v,p,p') => R.times(eval(v,p),eval(v,p')) == eval(v,times(p,p'))
end

functor Poly(R:RING):POLY = ?
```

(b) Implement your specification in (a), representing the polynomial $a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ as the list `[`a_0`,`a_1`,`a_2`,`$\ldots$`,`a_n`]`. Recall the algorithms for adding and multiplying polynomials you learned in school:

$$\begin{array}{cccc}
 & a_0 & + & a_1 x \\
+ & b_0 & + & b_1 x \\
\hline
 & (a_0 + b_0) & + & (a_1 + b_1)x
\end{array}$$

$$\begin{array}{cccccc}
 & a_0 & + & a_1 x & & \\
\times & b_0 & + & b_1 x & & \\
\hline
 & b_0 a_0 & + & b_0 a_1 x & & \\
+ & & & b_1 a_0 x & + & b_1 a_1 x^2 \\
\hline
 & b_0 a_0 & + & (b_0 a_1 + b_1 a_0)x & + & b_1 a_1 x^2
\end{array}$$

[*7 marks*]

Answer:

```
functor Poly(R:RING):POLY =
  struct
    structure R:RING = R
    type poly = R.t list
    fun create(l) = l
    fun eval(v,[]) = R.zero
      | eval(v,a::l) = R.plus(a,R.times(v,eval(v,create l)))
    fun plus(p,[]) = p
      | plus([],p') = p'
      | plus(a::p,a'::p') = R.plus(a,a')::plus(p,p')
    fun scalartimes(a,[]) = []
      | scalartimes(a,b::p) = R.times(a,b)::scalartimes(a,p)
    fun times([],p') = []
      | times(a::p,p') = plus(scalartimes(a,p'),R.zero::times(p,p'))
end
```

(c) Prove that your implementation in (b) satisfies the axiom in (a) specifying `times`, assuming that it satisfies the axioms specifying `plus` and `scalartimes`. *[11 marks]*

Answer:

```
Proving: R.times(eval(v,p),eval(v,p')) = eval(v,times(p,p')) by induction on p

Base case: p = [], then
R.times(eval(v,p),eval(v,p')) = R.times(R.zero,eval(v,p')) = R.zero
eval(v,times(p,p')) = eval(v,[]) = R.zero

Step: Assume R.times(eval(v,p),eval(v,p')) = eval(v,times(p,p'))
      Prove R.times(eval(v,a::p),eval(v,p')) = eval(v,times(a::p,p'))

R.times(eval(v,a::p),eval(v,p'))
    = R.times(R.plus(a,R.times(v,eval(v,p))),eval(v,p'))
    = R.plus(R.times(a,eval(v,p')),R.times(R.times(v,eval(v,p)),eval(v,p')))
    = R.plus(R.times(a,eval(v,p')),R.times(v,R.times(eval(v,p),eval(v,p'))))
    = R.plus(R.times(a,eval(v,p')),R.times(v,eval(v,times(p,p')))) [IH]

eval(v,times(a::p,p'))
    = eval(v,plus(scalartimes(a,p'),R.zero::times(p,p')))
    = R.plus(eval(v,scalartimes(a,p')),eval(v,R.zero::times(p,p')))
                                                        [axiom for plus]
    = R.plus(R.times(a,eval(v,p')),eval(v,R.zero::times(p,p')))
                                                        [axiom for scalartimes]
    = R.plus(R.times(a,eval(v,p')),R.plus(R.zero,R.times(v,eval(v,times(p,p')))))
    = R.plus(R.times(a,eval(v,p')),R.times(v,eval(v,times(p,p'))))
```