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, $1 \leq i \leq n$, 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*]

(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)
```

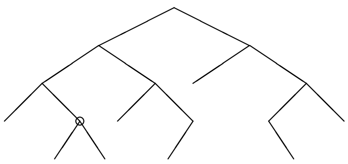
corresponds to the infinite list `[1,1,1,...]`.

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*]

(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:

QUESTION CONTINUES ON NEXT PAGE

`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*]

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*]

(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*]

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 $\texttt{[}a_0\texttt{,}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{]}$ into the polynomial $a_0 + a_1x + \cdots + a_nx^n$.

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*]

(b) Implement your specification in (a), representing the polynomial $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$ as the list $\texttt{[}a_0\texttt{,}a_1\texttt{,}a_2\texttt{,}\ldots\texttt{,}a_n\texttt{]}$. Recall the algorithms for adding and multiplying polynomials you learned in school:

$$\begin{array}{rccc}
 & a_0 & + & a_1x \\
+ & b_0 & + & b_1x \\
\hline
 & (a_0+b_0) & + & (a_1+b_1)x
\end{array}$$

$$\begin{array}{rccccc}
 & a_0 & + & a_1x & & \\
\times & b_0 & + & b_1x & & \\
\hline
 & b_0a_0 & + & b_0a_1x & & \\
+ & & & b_1a_0x & + & b_1a_1x^2 \\
\hline
 & b_0a_0 & + & (b_0a_1+b_1a_0)x & + & b_1a_1x^2
\end{array}$$

[*7 marks*]

(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*]