EXAME - 2ª Chamada
3·Fevereiro·2003
Duração: 2:30 horas

Paradigmasda Programação I
LMCC + LESI

NOME:_____

CURSO: _____ NUM: _____

I

Duas versões do Crivo de Eratosthenes para determinar a sequência de números primos

```
sem_multiplos p = filter (\x -> x `mod` p /= 0)

crivo []     = []
crivo (x:xs) = x : (crivo (sem_multiplos x xs))

crivo' []     = []
crivo' (x:xs) = x : (sem_multiplos x (crivo' xs))

primos = crivo [2..]
```

1. Exemplifique o funcionamento das duas funções de crivo apresentando a sequência de expressões intermédias que resultam dos cálculos de `crivo [2..12]` e `crivo' [2..12]`.

2. Justifique porque é que `crivo` é computacionalmente mais eficiente que `crivo'`.

3. Usando a sequência infinita de primos calculada por `primos`, escreva uma função HASKELL que, dado qualquer inteiro positivo, construa a sua factorização por primos.

   NOTA: Se $x < 2$ a factorização de x é a lista vazia. Se $x \geq 2$ a factorização de x é a lista cujo primeiro elemento é o primeiro primo p que divide x e o resto é a factorização do quociente x/p.

NOME:_____

CURSO: _____ NUM: _____

II

No plano cartesiano, quadrados e rectângulos com os lados paralelos aos eixos podem ser univocamente determinados pelas coordenadas do vértice inferior esquerdo e pelos comprimentos dos lados. Assim, para representar estas figuras geométricas, definiu-se em Haskell o seguinte tipo de dados:

```
data Figura = Quadrado Ponto Lado
            | Rectangulo Ponto Lado Lado
type Ponto = (Float,Float)
type Lado = Float
```

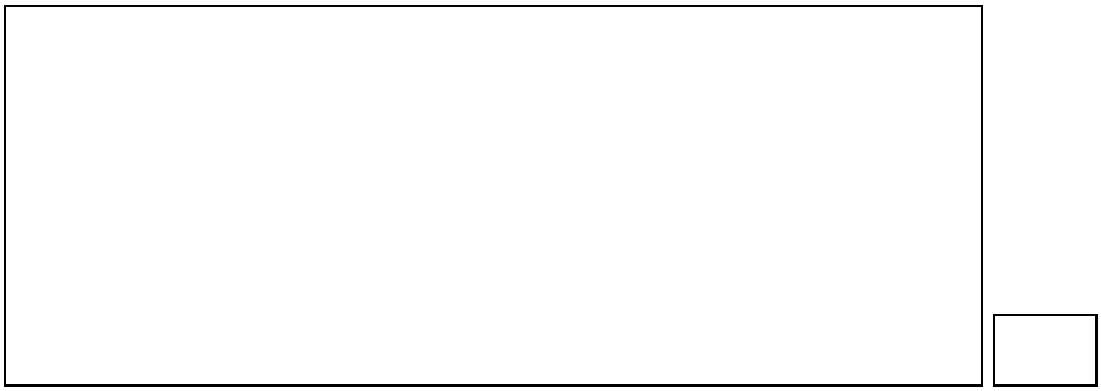
1. Defina a função `area :: Figura -> Float` que calcula a área da figura geométrica.

2. Defina `Figura` como instância da classe `Eq`. Note que um quadrado é um caso particular de um rectângulo.

3. Uma forma alternativa de representar quadrados e rectângulos no plano cartesiano é indicando os seus 4 vértices.

   (a) Defina um tipo de dados `FigVert` adequado para esta representação.

   (b) Defina a função `converte :: Figura -> FigVert` que faz a conversão de dados entre as duas representações.



NOME:_____

CURSO: _____ NUM: _____

III

Em anexo apresenta-se uma implementação simplificada do monade `ST` com as suas funcionalidades fundamentais.

1. As linhas 11-13 definem o tipo `(ST s)` como instância da classe `Monad`.

   (a) Para um monade genérico `m` diga qual o tipo dos dois construtores fundamentais da classe (`return` e `>>=`) e explique qual é a tarefa que se espera de cada um deles.

   (b) Explique como é que, nesta instância particular do tipo `(ST s)`, as funções `return` e `>>=` apresentadas cumprem esses objectivos.

2. Usando o construtor `up` (linhas 29-30) construa uma função

   ```
   raiz4 :: Integer -> Integer  -> Maybe Integer}
   ```

   que, dados inteiros x e m calcula (se existir) um inteiro y tal que

   $$(y^4 - x) \mod m = 0$$

   Sugestão: recorde a construção da função que calcula a raiz quadrada modular.

3. Apresente o texto completo de um programa HASKELL compilável que inclua a definição da função `raiz4` e que, na sua execução, leia um inteiro positivo do teclado, use-o como argumento de `raiz4` e escreva o resultado.

Módulo ST

```
1   module ST (module ST) where
2
3   import Monad
4
5   data ST s a =  ST { st :: s -> Maybe (a,s) }
6
7   instance Functor (ST s)
8       where
9       fmap f m = ST (\s -> do { (a,s') <-  st m s ; Just  (f a,s') })
10
11  instance Monad (ST s) where
12       return x = ST (\s -> Just (x,s))
13       m >>= f  = ST (\s -> do { (a,s') <-  st m s ;  st (f a) s' })
14
15  instance MonadPlus (ST s) where
16          mzero        = ST (\_ -> Nothing)
17          p `mplus` q  = ST (\s -> (st p s) `mplus` (st q s))
18
19  -- Funções no monade ST
20  on :: (ST s a) -> s -> Maybe a
21  on m s = do { (a,_) <- st m s ; Just a }
22
23  set :: (s -> Maybe a) -> ST s a
24  set w = ST (\s -> do { a <- w s ; Just (a,s) })
25
26  andthen :: (ST s a) -> (s -> a -> s) -> ST s a
27  andthen m f =  ST (\s -> do { (a,s') <- st m s ; Just (a, f s' a ) })
28
29  up :: (Enum a) => ST a a
30  up    = ST (\n -> Just (n,succ n))
31
32  -- Classe Range
33  class Range r  where
34      pick  :: ST (r a) a
35
36  instance Range [] where
37      pick = ST (\s -> case s of { [] -> Nothing ; (x:s') -> Just (x,s') })
38
39  -- Funções utilitárias em monades
40  till :: (Monad m) => (m a) -> (a -> Bool) -> (m a)
41  r `till` p = do { x <- r ;  if p x then return x else r `till` p }
42
43  while :: (MonadPlus m) => m a -> (a -> Bool) -> m a
44  r `while` p = do { x <- r ; if p x then return x else mzero }
45
46  collect :: (MonadPlus m) => (m a) ->  m [a]
47  collect r  = do { x <- r ; xs <- collect r ; return (x:xs) } `mplus` return []
48
49  loop :: (MonadPlus m) => (a -> m a) -> a -> m a
50  loop f x =  do { y <- f x ; loop f y }  `mplus` return x
```