

Tentamen i Programmeringsteknik D, del A

Friday 18th January, 2002, 8:45-12:45.

Examiner: John Hughes.

Questions during the exam may be directed to Dennis Björklund, 772 5402.

Permitted aids:

English-Swedish or English-other language dictionary.

- Begin each question on a new sheet. Write your personal number on every sheet.
- You may lose marks for unnecessarily long, complicated, or unstructured solutions.
- Full marks are awarded for solutions which are elegant, efficient, and correct.
- You are free to use any Haskell standard functions, including those whose definitions are attached, unless the question specifically forbids you to do so.
- You may use the solution of an earlier part of a question to help solve a later part, even if you did not succeed in solving the earlier part.
- The exam consists of 3 questions, worth 15, 25 and 20 points. A total of 20 points is sufficient to pass.

1. In this question you will develop functions to read and write numbers in hexadecimal notation. Hexadecimal numbers are expressed using base 16, which means that the digits range in value from 0 to 15. The digits 11 through 15 are written A, B, C, D, E and F. Thus 243 ($= 15 \times 16 + 3$) is written as F3.

You will find useful the standard functions

```
ord :: Char -> Int
chr :: Int -> Char
```

which convert characters to and from their *ASCII codes*. The ASCII codes for the characters '0', '1', ... '9' are consecutive integers (48, 49, 50... *etc.*), and so are the ASCII codes for 'A', 'B', ... 'F'.

- (a) Define a function

```
digit :: Int -> Char
```

which converts an integer between 0 and 15 into the corresponding hexadecimal digit. For example, `digit 11 = 'B'`.

(2 p)

- (b) Define a function

```
digitVal :: Char -> Int
```

which converts a hexadecimal digit into the number it represents. For example, `digitVal 'B' = 11`.

(2 p)

- (c) Define a function

```
showHex :: Int -> String
```

which converts a 32-bit integer to an eight-digit hexadecimal number. Since each hex digit can represent four bits of information, any 32-bit number can be written with *at most* eight hex digits, and (for consistent layout) is often written using *exactly* eight digits. Thus 243 would be written as 00000F3 instead of just F3. You should define `showHex` so that the result is always eight digits long.

(4 p)

- (d) Define a function

```
readHex :: String -> Int
```

which converts any string of hex digits to the corresponding integer. Do *not* assume the string is any particular length. For example, `readHex "F3" = readHex "00000F3" = 243`.

(4 p)

- (e) Are the following properties true or false?

- i. `readHex (showHex n) == n`, where `n` is an integer.
- ii. `showHex (readHex s) == s`, where `s` is a string of hex digits.

Give a counterexample for any property you believe to be false.

(3 p)

2. In this question you will develop functions for manipulating *regions in the plane*, which might form part of a library for 2D graphics. We shall represent points by their $x - y$ coordinates

```
data Point = Pt Float Float
```

and regions by *functions* with the type

```
type Region = Point -> Bool
```

A region r represents a set of points in the plane; a point p is in this set if $r\ p$ returns True.

- (a) Define

```
inR :: Point -> Region -> Bool
```

such that $p\ \text{'inR'}\ r$ is True if p is in the region r . (2 p)

- (b) Define functions

```
box :: Point -> Point -> Region
```

```
circle :: Point -> Float -> Region
```

where

- $\text{box } p\ q$ returns the region of points in the rectangle whose lower left hand corner is at p , and whose upper right hand corner is at q ,
- $\text{circle } p\ r$ returns the region of points in the circle of radius r with centre at p .

Notice that these functions return *functions* as results! (4 p)

- (c) Define

```
unionR :: Region -> Region -> Region
```

```
intersectR :: Region -> Region -> Region
```

```
complementR :: Region -> Region
```

such that a point p is in $r\ \text{'unionR'}\ r'$ if it is in r *or* r' , it is in $r\ \text{'intersectR'}\ r'$ if it is in r *and* r' , and it is in $\text{complementR } r$ if it is *not* in r . (6 p)

- (d) Define

```
ring :: Point -> Float -> Float -> Region
```

such that $\text{ring } p\ r\ w$ constructs a ring-shaped region centred on p , where the inside edge of the ring is a circle of radius r , and the width (or thickness) of the ring is w . (3 p)

- (e) Region operations can be made more efficient by storing a *bounding box* with each region: the points outside the box are either all in, or all not in the region, which makes testing whether such a point is in the region fast. We represent boxes by their lower left and upper right corners, and define

```
data Box = Box Point Point
type BoxedRegion = (Box,Bool,Region)
```

A *boxed region* (box,b,r) contains a point p if p lies inside the box and is in region r, or lies outside the box and b is True.

- i. Define

```
inBR :: Point -> BoxedRegion -> Bool
```

to test whether a point lies in a boxed region.

(2 p)

- ii. Define

```
boxBR :: Point -> Point -> BoxedRegion
```

```
circleBR :: Point -> Float -> BoxedRegion
```

to construct boxed regions representing the same sets of points as the functions `box` and `circle` from part 2b.

(4 p)

- iii. Define

```
unionBR :: BoxedRegion -> BoxedRegion -> BoxedRegion
```

to compute the union of two boxed regions.

(4 p)

3. This question concerns a simple kind of ‘expert system’, which can classify something by asking questions about it. For example, the documentation provided with a computer often contains a ‘trouble-shooting guide’ to help the user diagnose problems, consisting of a series of questions and instructions as to how to proceed in each case. We will take as an example an expert system to identify an animal, which we know in advance is either an ant, a zebra, or an angel fish.

Our expert system will use a *decision table* such as the following, containing questions and instructions to follow in case the answer is “yes”, and in case it is “no”.

Does it live in water?

- if yes, it is an angel fish.
- if no, is it black?
 - if yes, it is an ant.
 - if no, it is a zebra.

We can represent such a decision table using the datatype

```
data Table = Decision String | Question String Table Table
```

where `Decision d` is an instruction to make the decision `d`, and `Question q y n` is an instruction to ask question `q` and follow the instructions in table `y` if the answer is yes, or table `n` if the answer is no.

- (a) How would the decision table in the example above be represented as an element of the `Table` datatype? (2 p)
- (b) Write a function

```
decide :: Table -> IO ()
```

which follows the instructions in a `Table`, asking the user the questions and using the user’s answers to decide which instructions to follow next. An example interaction with `hugs` using the table above might be

```
Main> decide table
Does it live in water? n
Is it black? y
It is an ant
```

Use

```
putStr :: String -> IO ()
```

to display questions to the user, and

```
getLine :: IO String
```

(which reads one line of input) to read the user's reply. (4 p)

- (c) To construct an animal identification decision table, we must know the answers to a number of possible questions about each animal. For example, the table above was constructed from the following facts:

```
facts :: Facts
facts =
  [("It is an ant",
    [("Is it smaller than a pen",True),
     ("Is it black",True),
     ("Does it live in water",False)]),
   ("It is a zebra",
    [("Is it smaller than a pen",False),
     ("Does it live in water",False),
     ("Is it striped",True),
     ("Is it black",False)]),
   ("It is an angel fish",
    [("Is it smaller than a pen",True),
     ("Is it striped",True),
     ("Is it black",False),
     ("Does it live in water",True)])]
```

where every possible conclusion is paired with a list of relevant questions and the appropriate answers for that case.

Give a suitable definition of the type Facts. (1 p)

- (d) Define a function

```
restrict :: String -> Bool -> Facts -> Facts
```

which, given a question, its answer, and a list of possible facts, returns a list of those facts which are not ruled out by the given answer. Note that some facts may not refer to the given question; in that case they cannot be ruled out by any answer. For example,

```
restrict "Is it striped" False facts ==
  [("It is an ant",
    [("Is it smaller than a pen",True),
     ("Is it black",True),
     ("Does it live in water",False)])]
```

The possibility that "It is an ant" is not ruled out here because we know nothing about whether or not ants are striped. You should remove any references to the given question from the facts in the result: for example,

```
restrict "Is it black" True facts ==
  [("It is an ant",
    [("Is it smaller than a pen",True),
     ("Does it live in water",False)])]
```

where the pair ("Is it black", True) has been removed, since this question has already been asked.

(5 p)

(e) Define a function

```
table :: Facts -> Table
```

which builds a decision table from a collection of facts. You should aim to reach a decision by asking as few questions as possible: note in the example table that at most two questions are needed to identify the animal, even though we know three or more facts about each one. A good strategy is to ask a question that, whether the answer is yes or no, minimises the number of possible final decisions. In this example, asking "Does it live in water?" first guarantees that, whatever the answer, there can be no more than two remaining possibilities. Asking "Is it striped?" on the other hand would not be a good choice, because if the answer is "yes" then no animal is ruled out. (Ants are not striped, but this fact is not among those we are using for classification).

(8 p)