CHALMERS TEKNISKA HÖGSKOLA
Datavetenskap
Dennis Björklund

January 16th, 2004
Functional programming
INN040/TDA450

Exam
Functional Programming

Friday, January 16th, 2004, 8.45 - 12.45.
Examiner: Dennis Björklund, telephone office: 772 5402, cell: 073 636 51 79.

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 5 questions, worth 4, 6, 16, 14, and 20 points. For a Chalmers students the grade limits are: 3: 24p, 4: 36p, 5: 48p. For a Gothenburg University student they are: G: 28p, VG: 48p.

1. Describe the concepts of:

   (a) Higher order functions (1 p)

   (b) Lazy evaluation (1 p)

   (c) A polymorphic function (1 p)

   (d) A pure function (1 p)

2. Give the most general type for each of the following functions:

   (a) (2 p)
   ```
   fa []     _ = []
   fa (x:xs) p = [x,p] : fa xs p
   ```

   (b) `fb a b c d = if a then d b else d c` (2 p)

   (c) `fc x = fc x` (2 p)

3. We will work with the following type that represents binary trees in haskell:

```
data Tree a = Node a (Tree a) (Tree a)
            | Leaf
```

In the questions below when you should give the type of the functions, it should be the most general type[1].

(a) Write the tree

```
         6
        / \
       3   8
      / \
     1   5
```

as a value in the datatype above. (2 p)

(b) Give the type and define a function that calculates the number of nodes in a tree. (3 p)

(c) Give the type and define a function that calculates the maximum height of a tree. The maximum height of a tree is defined as the number of nodes in the longest path of the tree, from the root down to a leaf. For example, the tree above has a maximum height of 3. (2 p)

(d) For lists there is (as you know) a function:

```
map :: (a -> b) -> [a] -> [b]
```

that given a function `a->b` can be used to transform a list of `a` values into an another list with `b` values.
Define and give the type for the corresponding function for our trees. (3 p)

(e) A binary search tree is a tree like above with the added condition that for each node it holds that:

- all the nodes in the left subtree is smaller then or equal to the element in the node.
- all the nodes in the right subtree is greater then the element in the node.

If you look at the example tree above you see that it indeed is a binary search tree according to the above condition.
Binary search trees are good since you can fast find out if an element is in the tree or not (you don't need to visit every node to do that).
Define a search function that given a tree and a value returns True or False depending on if the value was in the tree or not. We assume here that the tree we get is a binary search tree so we can make an efficient implementation. Also give the type of the function. (3 p)

[1]This is the type you most likely expect it to have and the type that hugs would produce.

(f) Define and give the type of a function that inserts a value in a binary search tree. It should take the tree and the value as inputs and produce a new binary search tree as output. (3 p)

4. We want to make a small movie “database” so we define the following types:

```
type Name = String
type Info = String
type Title = String
type Story = String
type Role = String
type Rating = Double -- Better movies get higher points

data Gender = Male | Female
     deriving (Eq)

data Genre = Action | Comedy | Drama | Thriller | Romance
     deriving (Eq)

type Actor = (Name, Gender, Info)
type Movie = (Title, [Genre], Rating)

type Actors = [Actor]
type Movies = [Movie]
type Participants = [ (Title, Name, Role) ]
```

An example of a database with the above structure is:

```
gActors =
    [ ("Kevin Bacon", Male,
       "His father is Edmund Bacon, a famous Philidelphia City Planner."),
      ("Meryl Streep", Female,
       "Has a fear of helicopters.."),
      ("Tom Hanks", Male,
       "Gained, then lost 50+ lbs. for his role in Cast Away.")
    ]

gMovies =
    [ ("The River Wild", [Action,Thriller], 6.1),
      ("Apollo 13",      [Drama],           7.5)
    ]

gParticipants =
    [ ("The River Wild", "Kevin Bacon",    "Wade"),
      ("The River Wild", "Meryl Streep",   "Gail"),
      ("Apollo 13",      "Kevin Bacon",    "Jack Swigert"),
      ("Apollo 13",      "Tom Hanks",      "Jim Lovell")
    ]
```

(a) Define a function with the type:

```
betterThenOrEqual :: Movies -> Rating -> Movies
```

that takes a list of movies and a rating and returns all movies with at least that rating. (2 p)

(b) Define a function with the type:

```
hasGenre :: Movies -> Genre -> Movies
```

that takes a list of movies and a genre and returns all movies with that genre. (1 p)

(c) Use the above functions to write a new function:

```
find_good_comedies :: Movies -> Movies
```

that finds all comedies with a rating of at least 7.5. (1 p)

(d) Sometimes I remember two (or more) actors in a movie but don't remember the title of the movie (can be more then one). Define a function:

```
moviesWithThese :: Participants -> [Name] -> [Title]
```

(2 p)

(e) Define a function

```
movieInfo :: Movies -> Actors -> Participants
             -> Title -> IO ()
```

that presents a movie nicely on the screen. For example:

```
movieInfo gMovies gActors gParticipants "The River Wild"
```

would produce:

```
The River Wild
--------------
Rating: 6.1

Actors
------
Kevin Becon (m)........................Wade
Meryl Streep (f).......................Gail
```

You should produce the output exactly as above. Notice the padding to make the roles aligned under each other. The role column should start at column 40. The function should of course work for all databases and not just this example. You can assume that the actor fits within these 40 characters. Also notice the underlining under the title that is of different length depending on the title.

If the movie does not exist, you should print out some nice error message.

You probably need to make a couple of help function. Full points is (as always) only rewarded for nice solutions. (8 p)

5. This question concerns finding change. Suppose you have a collection of coins in your pocket, which we represent by a list of numbers such as `[1,1,5,5,5,10,10]`. Then you can pay 16kr by paying three 5kr coins and a 1kr, but you cannot pay 18kr at all. Suppose the person you are paying also has some coins, say `[1,1,1,5]`. Then you can pay 18kr by paying 20kr, and receiving 2kr in change. We will design functions to decide whether one person can pay another a specific amount, with or without giving change.

   (a) Define a function

   ```
   sublists :: [a] -> [[a]]
   ```

   which given a list, returns a list of all the ways of choosing some of the list elements. For example,

   ```
   sublists [1,2,3] ==
     [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
   ```

   (the order of this list doesn't matter here). (3 p)

   (b) Define a function

   ```
   amounts :: [Int] -> [Int]
   ```

   which given a list of the coins in your pocket, returns a list of all the amounts you can pay using those coins. (3 p)

   (c) Define a function

   ```
   withChange :: [Int] -> [Int] -> [Int]
   ```

   which, given a list of the coins in the first person's pocket, and a list of the coins in the second person's pocket, returns a list of all the amounts the first person can pay the second assuming that the second person is willing to provide change. The result should not contain negative numbers. (3 p)

   (d) The `sublists` function you wrote in part (*a*) is inefficient for this problem, because it ignores the fact that we may have many coins with the same value in our pockets. For example,

   ```
   sublists [1,1] ==
     [[],[1],[1],[1,1]]
   ```

   where the two sublists `[1]` represent choosing different coins, but with the same value. Since we do not care which coin we actually pay with, we do not need to consider these sublists separately.

   We can improve the efficiency of our program by working with *bags* instead, which we represent by lists of pairs of a coin value, and the number of coins of that value in the bag. For example, `[1,1,5,5,5,10,10]` corresponds to the bag `[(1,2),(5,3),(10,2)]`.

      i. Define a function

```
bag :: Eq a => [a] -> [(a,Int)]
```

which converts a list to a bag. (3 p)

ii. Define a function

```
list :: [(a,Int)] -> [a]
```

which converts a bag back into a list. (3 p)

iii. Define a function

```
subbags :: [(a,Int)] -> [[(a,Int)]]
```

which returns a list of every bag we can make by choosing some of the elements of the given bag. For example,

```
subbags [(1,2)] ==
  [[],[(1,1)],[(1,2)]]
```

(4 p)

How would you use these functions to improve your function `withChange`? (1 p)