

Tentamen i Programmeringsteknik D, del A

Saturday 27th October, 2001, 8:45-12:45.
Examiner: John Hughes, tel 070 756 3760.

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 10, 15, and 25 points. A total of 20 points is sufficient to pass.

1. One good way to sort a list of items is to divide the list into two roughly equal halves, sort each half, and then merge the resulting lists together. For example, sort `[4, 1, 3, 2]` by dividing it into its two halves, `[4, 1]` and `[3, 2]`, sorting each half to give `[1, 4]` and `[2, 3]`, and then merging these lists together to give `[1, 2, 3, 4]`.

(a) Define a function

```
merge :: Ord a => [a] -> [a] -> [a]
```

which merges two sorted lists to produce a sorted result. For example,

```
merge [1,3,4,5,7,9] [2,4,6,7] == [1,2,3,4,4,5,6,7,7,9]
```

 (5 p)

(b) Define a function

```
msort :: Ord a => [a] -> [a]
```

which sorts its argument using the method described above. For example,

```
msort [3,4,1,5,9] == [1,3,4,5,9]
```

 (5 p)

Do not use the standard functions `sort` or `insert` in this question.

2. In this question, you will write a program to generate a nonsense text *similar* to a given text. For example, given the chorus of “Mamma Mia”, your program might generate

```
Mamma Mia,  
Here we go again!  
My, my, how can I re we go again!  
My, my, how can I re we we we go again!  
My, my, my, how can I resist you?
```

The idea is to generate the text one character at a time, and use random numbers to decide which character to generate next. But in order to generate output similar to the original, we adopt the following rule:

Each triple of characters in the output must also occur in the input.

For example, if the original text is “Hello clouds”, then the generated text might be “Hello clo clo clo clo clouds”. After “cl”, the only possible character is “o”, because that is the only character that follows “cl” in the original text. But after “lo”, we can generate either “ ” or “u”, since both “lo ” and “lou” occur in the original.

(a) Define a function

```
triples :: [a] -> [(a,a,a)]
```

which returns a list of all triples of adjacent elements in a list. For example,

```
triples [1..5] = [(1,2,3),(2,3,4),(3,4,5)]
```

 (3 p)

(b) Define a function

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

such that `followers x y ts` finds a list of all the `z` such that `(x,y,z)` is a triple in the list `ts`. For example,

```
followers 'l' 'o' (triples "hello clouds") = " u"
```

since both `('l', 'o', ' ')` and `('l', 'o', 'u')` are elements of `triples "hello clouds"`. (2 p)

(c) Define a function

```
pick :: [a] -> Int -> a
```

such that `pick xs r` returns an element of `xs` depending on the random number `r`. You should ensure that

- `pick` returns an element of `xs` *no matter what the value of `r`* — do *not* assume `r` is between 0 and the length of `xs`!
 - If `r` is equally likely to be any integer of type `Int`, then `pick` is equally likely to choose any element of `xs`.
- (3 p)

(d) Define a function

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

so that, if

- `ts` is a list produced by `triples` from some original list `xs`,
- `x` and `y` are two elements of `xs` which do occur beside each other,
- `rs` is a (sufficiently long) list of random numbers,

then `generate x y ts rs` produces a random list as follows. Begin with `x` and `y`, then generate each successive element from the two elements before it by choosing some `z` so that those three elements form a triple in `ts`. Stop if there is no such `z`. Use the random numbers in `rs` to make the choices randomly, and make sure you don't use each random number to make more than one choice.

For example,

```
generate 'h' 'e' (triples "hello clouds") randomInts
```

might return

```
"hello clo clo clo clo clo clouds"
```

 (5 p)

(e) Define a function

```
nonsense :: Eq a => [a] -> [a]
```

so that `nonsense xs` generates random nonsense from an original list `xs` using the method described at the beginning of this question. You can assume that

```
randomInts :: [Int]
```

is a very long list of random numbers.

(2 p)

3. In this question, you will develop functions for storing *many versions* of a file in a compact way. The key idea is to store just the *differences* between versions, which hopefully will be small.

We shall represent the differences between two strings as a list of *editor commands* for a simple editor, which produce the new string from the previous version. We consider an editor with only three commands,

- *insert* a given character at the current editing position,
- *delete* the character at the current editing position,
- *copy* the character at the current editing position to the output, and move the editing position one character to the right.

Editing begins at the left hand end of the input, and continues until the editing position reaches the right hand end (i.e. the entire input has been consumed, either copied or deleted).

For example, the commands to produce “hello world” from “helo wosld” are

Output	Command	Input
“”		“helo wosld”
“h”	<i>copy</i>	“elo wosld”
“he”	<i>copy</i>	“lo wosld”
“hel”	<i>copy</i>	“o wosld”
“hell”	<i>insert 'l'</i>	“o wosld”
“hello”	<i>copy</i>	“ wosld”
“hello ”	<i>copy</i>	“wosld”
“hello w”	<i>copy</i>	“osld”
“hello wo”	<i>copy</i>	“sld”
“hello wor”	<i>insert 'r'</i>	“sld”
“hello wor”	<i>delete</i>	“ld”
“hello worl”	<i>copy</i>	“d”
“hello world”	<i>copy</i>	“”

Here each command is shown together with the output it produces and the input remaining after it is obeyed. Notice that editing continues until the input has been entirely consumed!

- (a) Define a new datatype to model an *editor command*:

data Command = ... (3 p)

(b) Define a function

```
edit :: [Command] -> String -> String
```

such that `edit cmds xs` returns the output produced by editing the string `xs` with the commands `cmds`. For example, if `cmds-example` represents the list of commands in the table above, then

```
edit cmds-example "helo wosld" == "hello world" (4 p)
```

(c) A list of editing commands is not a very compact way to represent a new version, because, with this simple editor, there must always be at least one command per input character — either a *copy*, or a *delete*. But, as we can see above, when the input and edited output are similar, then the editing commands contain long sequences of *copy* commands. Let us compress these sequences using *run length encoding*: we replace each element by its value and the number of repetitions. The editing commands in the example would thus be represented as “3× *copy*, 1× *insert 'l'*, 4× *copy*, 1× *insert 'r'*, 1× *delete*, 2× *copy*”.

Define functions

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

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

which `compress` and `uncompress` a list in this manner. For example,

```
compress "hello" == [(1,'h'), (1,'e'), (2,'l'), (1,'o')]
```

For any list `xs`, `uncompress (compress xs)` should be equal to `xs`. (8 p)

(d) Define a function

```
shorter :: [a] -> [a] -> [a]
```

such that `shorter xs ys` returns either `xs` or `ys`, whichever list is the shorter. (2 p)

(e) Now, define a function

```
shortestEdit :: String -> String -> [Command]
```

so that `shortestEdit xs ys` returns the *shortest list of editing commands which produces xs from ys*. For example,

```
shortestEdit "hello world" "helo wosld"
```

should produce the sequence of commands in the table. Of course, for any `xs` and `ys`, we expect that

```
edit ys (shortestEdit xs ys) == xs
```

Hint: When either input is the empty list, then `shortestEdit` is easy to define — there is only one possible list of commands. When both inputs start with the same character, it is also easy: just copy it and recurse. The tricky case is when `xs` and `ys` start with *different* characters — then we can either *insert* the first character of `xs`, or *delete* the first character of `ys`. Which should we choose? Whichever gives the shortest list of edits in the end...

(8 p)