

1. [10] Define in Haskell a function `runs` that divides a list of `Int`s into its runs of odd and even integers. For example,

```
runs [3,1,6,2,4,7,3,5,8] ~~ [[3,1],[6,2,4],[7,3,5],[8]]
```

To receive full credit, your solution must be not only correct but clear and simple.

A solution:

```
> runs :: [Int] -> [[Int]]
> runs [] = []
> runs (x:xs) = (x : takeWhile p xs) : runs (dropWhile p xs)
>   where
>     p = if odd x then odd else even
```

2. [10] Define a Haskell expression `facs` of type `[Integer]` so that the value of `facs!!n` is $n!$. Use memoization to minimize the cost of computing each element.

```
facs :: [Integer]
facs = 1 : [ n * f | (n,f) <- zip [1..] facs ]
```

3. [15] Define a Haskell parser `assign :: Parser ([String],[String])` that recognizes GCN assignment commands, whose syntax is

Assign ::= Variable := Expression | Variable , Assign , Expression

Assume that the following functions are available for you to use:

- `var :: Parser String` -- recognizes Variables
- `expr :: Parser String` -- recognizes Expressions
- `literal :: String -> Parser String` -- `literal s` recognizes occurrences of `s`

together with all of the definitions in `ParseLib.hs`.

As its type indicates, your `assign` parser should deliver a pair of lists; the first list should contain the recognized assignment command's left-hand-side variables, and the second should contain its right-hand-side expressions. Both lists should contain their items in the order in which they appear in the statement. For example,

```
assign "a,b,c:=1,2,3" should deliver ([ "a", "b", "c" ], [ "1", "2", "3" ])
```

A solution:

```
> assign :: Parser ([String],[String])
> assign = do v <- var
>           literal "!="
>           e <- expr
>           return ([v],[e])
>
>           +++
>           do v <- var
>             char ','
>             (vs,es) <- assign
>             char ','
>             e <- expr
>             return (v:vs,es++[e])
```

4. [15] Given the Haskell functions

$$\begin{aligned} \text{map } f \ [] &= [] \\ \text{map } f \ (z:zs) &= f \ z : \text{map } f \ zs \\ [] ++ ys &= ys \\ (x:xs) ++ ys &= x : (xs ++ ys) \end{aligned}$$

prove formally, or disprove by showing a counterexample, that

$$\text{map } f \ (as ++ bs) = \text{map } f \ as ++ \text{map } f \ bs$$

for all functions f and finite lists as and bs for which the expressions are well-defined (i.e., which are type-compatible). If you give a proof, be sure to give reasons for all steps.

Proof:

Case $[]$: $\text{map } f \ ([] ++ bs) = \text{map } f \ [] ++ \text{map } f \ bs$

$$\begin{aligned} &\text{map } f \ ([] ++ bs) = \text{map } f \ [] ++ \text{map } f \ bs \\ &= \{ ++.0, \text{map}.0 \} \\ &\text{map } f \ bs = [] ++ \text{map } f \ bs \\ &= \{ ++.0 \} \\ &\text{map } f \ bs = \text{map } f \ bs \\ &= \{ \text{reflexivity of } (=) \} \\ &\text{true} \end{aligned}$$

Case $(a:as)$: $\text{map } f \ ((a:as) ++ bs) = \text{map } f \ (a:as) ++ \text{map } f \ bs$

$$\begin{aligned} &\text{map } f \ ((a:as) ++ bs) \\ &= \{ ++.1 \} \\ &\text{map } f \ (a : (as ++ bs)) \\ &= \{ \text{map}.1 \} \\ &f \ a : \text{map } f \ (as ++ bs) \\ &= \{ \text{IH} \} \\ &f \ a : (\text{map } f \ as ++ \text{map } f \ bs) \\ &= \{ ++.1 \} \\ &(f \ a : \text{map } f \ as) ++ \text{map } f \ bs \\ &= \{ \text{map}.1 \} \\ &\text{map } f \ (a:as) ++ \text{map } f \ bs \quad \S \end{aligned}$$

5. [10] Find formally the weakest precondition which guarantees that x exceeds 6 after the execution of

if $x > 3 \rightarrow x := y+2$ **fi** $x < 3 \rightarrow x := y*2$ **fi**

Answer:

$$\begin{aligned} &\text{wp}(\text{"if } x > 3 \rightarrow x := y+2 \text{ fi } x < 3 \rightarrow x := y*2 \text{ fi"} , x > 6) \\ &= \{ \text{def. of if...fi} \} \\ &(x > 3 \vee x < 3) \wedge (x > 3 \Rightarrow \text{wp}(\text{" } x := y+2 \text{"}, x > 6)) \wedge (x < 3 \Rightarrow \text{wp}(\text{" } x := y*2 \text{"}, x > 6)) \\ &= \{ \text{algebra, def of '}' \} \\ &(x \neq 3) \wedge (x > 3 \Rightarrow y+2 > 6) \wedge (x < 3 \Rightarrow y*2 > 6) \\ &= \{ \text{algebra} \} \\ &(x \neq 3) \wedge (x > 3 \Rightarrow y > 4) \wedge (x < 3 \Rightarrow y > 3) \\ &= \{ \text{algebra} \} \\ &(x > 3 \wedge y > 4) \vee (x < 3 \wedge y > 3) \end{aligned}$$

6. [10] How —and why— would a type checker respond to the statement

```
print( if roman then "XVII" else 17 );
```

assuming that type checking is...

- a. static?

A static type checker must be able to determine the type of every expression by analyzing its text. In this case, the type of the `if`-expression may be string or integer, depending on the value of `roman`. Hence this expression would be rejected.

- b. dynamic?

A dynamic type checker requires only that every function is applied to arguments of the appropriate type. Assuming that `print` is overloaded to handle both strings and integers, this statement would be accepted.

7. [15] Translate the following:

- a. from C into GCN:

```
if ( x > 0 ) y = y + 2;
```

```
if x > 0 → y := y+2 || x ≤ 0 → skip fi
```

- b. from GCN into C or Pascal

```
a, b, c := b+1, c+1, a+b
```

```
int tc = a + b;
```

```
a = b+1; b = c+1; c = tc;
```

- c. from C into GCN

```
switch( x )
```

```
{
```

```
  case 3: y = y+1;
```

```
  case 2: z = z+1;
```

```
  case 1: w = w+5;
```

```
}
```

```
if x = 3 → y := y+1; z := z+1; w := w+5
```

```
|| x = 2 → z := z+1; w := w+5
```

```
|| x = 1 → w := w+5
```

```
|| x < 1 ∨ x > 3 → skip
```

```
fi
```