

1. [15] Suppose that polynomials are represented in a Haskell program by values of the type `Polynomial`, which is defined by

```
type Polynomial = [Float]
```

in which the list elements are the coefficients of terms in descending order of the terms' exponents. For example, the polynomial

$$4.5 \cdot x^3 - 3 \cdot x^2 + 9$$

would be represented by

```
[4.5, -3, 0, 9]
```

An efficient method for evaluating polynomials is Horner's Rule, which reduces the number of multiplications by taking advantage of the equation

$$a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \dots + a_1 \cdot x + a_0 = (\dots((a_n \cdot x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + \dots + a_1) \cdot x + a_0$$

Define a Haskell function `evaluate :: Polynomial -> Float -> Float` which evaluates a polynomial at a given value of `x` using Horner's Rule. Use no explicit recursion; instead, use one or more higher-order Prelude functions.

```
> evaluate :: Polynomial -> Float -> Float
> evaluate poly x = foldl (\n a -> n*x + a) 0 poly
```

2. [15] Define a Haskell IO action which

1. reads a line from the keyboard
2. reads more lines until it reads a line identical to the one it read in step 1
3. delivers a list containing all of the lines read between the first and the last

For example, if the lines read were

```
stop
abilene
san antonio
austin
houston
temple
dallas
stop
```

the list delivered would be

```
["abilene","san antonio","austin","houston","temple","dallas"]
```

A solution:

```
> readLines :: IO [String]
> readLines = do end <- getLine
>               lines <- readUpTo end
>               return lines
>
>   where
>     readUpTo :: String -> IO [String]
>     readUpTo end = do line <- getLine
>                       if line == end
>                         then return []
>                       else do lines <- readUpTo end
>                               return (line:lines)
```

3. [20] Define in Haskell a parser `ifCommand` for Guarded-Command Notation selection commands.

The syntax for these commands is

$$\begin{aligned} \text{IfCommand} &::= \mathbf{if} \text{ GC } \{ [] \text{ GC } \} \mathbf{fi} \\ \text{GC} &::= \text{Guard} \rightarrow \text{Command} \end{aligned}$$

Assume that you're given two parsers

```
command :: Parser Command
guard   :: Parser Guard
```

Your parser's type should be

```
Parser [(Guard,Command)]
```

and you may ignore considerations of white space except that **if** is followed, and **fi** is preceded, by at least one space.

A solution:

```
> ifCommand :: Parser [(Guard,Command)]
> ifCommand = do string "if"
>               spaces
>               gc <- guardedCommand
>               gcs <- many ( do string "[]"; guardedCommand )
>               spaces
>               string "fi"
>               return (gc:gcs)

> guardedCommand :: Parser (Guard,Command)
> guardedCommand = do g <- guard
>                    string ">"
>                    c <- command
>                    return (g,c)

> string :: String -> Parser String -- included in ParseLib.hs
> string "" = return ""
> string (x:xs) = do char x; string xs; return (x:xs)

> spaces :: Parser String
> spaces = do char ' '; many (char ' ')
```

4. [10] Find the weakest precondition such that execution of

$$\mathbf{if} \ x < y \rightarrow a, b := b, a \ \parallel \ x < y \rightarrow \mathbf{skip} \ \mathbf{fi}$$

establishes $a \leq b$. You should show the steps of the formal derivation, and give the result in simplified form.

$$\begin{aligned} & \text{wp } \mathbf{if} \ x < y \rightarrow a, b := b, a \ \parallel \ x < y \rightarrow \mathbf{skip} \ \mathbf{fi} \ (a \leq b) \\ &= \quad \{ \text{def. } \mathbf{if} \} \\ & (x < y) \wedge (x < y \Rightarrow \text{wp } \mathbf{if} \ x < y \rightarrow a, b := b, a \ \parallel \ x < y \rightarrow \mathbf{skip} \ \mathbf{fi} \ (a \leq b)) \wedge (x < y \Rightarrow \text{wp } \mathbf{skip} \ (a \leq b)) \\ &= \quad \{ \text{def. } \mathbf{if}, \text{ def. } \mathbf{skip} \} \\ & (x < y) \wedge (x < y \Rightarrow b \leq a) \wedge (x < y \Rightarrow a \leq b) \\ &= \quad \{ \text{predicate calculus} \} \\ & (x < y) \wedge (b \leq a) \wedge (a \leq b) \\ &= \quad \{ \text{predicate calculus} \} \\ & (x < y) \wedge (a = b) \end{aligned}$$

5. [15] Define GCN commands equivalent to each of the following C statements:

- | | |
|------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>a. <code>if(x > y) u = x+y; v = x-y;</code></p> | <p>if $x > y \rightarrow u := x+y \parallel x \leq y \rightarrow$ skip fi;
$v := x-y$</p> |
| <p>b. <code>while(n > j)
 for(j = 0, k = 1; k < m; ++j, ++k)
 n--;</code></p> | <p>do $n > j \rightarrow$
 $j, k := 0, 1;$
 do $k < m \rightarrow n, j, k := n-1, j+1, k+1$ od
od</p> |
| <p>c. <code>switch(x) {
 case 1: x = x + a;
 case 2: x = x + b;
 case 3: x = x + c;
}</code></p> | <p>if $x = 1 \rightarrow x := x + a + b + c$
 $\parallel x = 2 \rightarrow x := x + b + c$
 $\parallel x = 3 \rightarrow x := x + c$
 $\parallel x \neq 1 \wedge x \neq 2 \wedge x \neq 3 \rightarrow$ skip
fi</p> |

6. [10] For each of the following expressions, determine (i) whether a static type check would detect an error, and (ii) whether executing the code would result in a dynamic type error. Assume that the code is Haskell, but with either static or dynamic type checking (actual Haskell, of course, has only static checking).

Indicate your answer by circling **one** of {**s**tatic, **d**ynamic, **b**oth, **n**either}, and give a brief explanation of your answer.

For all of the samples assume the following definitions:

`x = True, y = 2 :: Int, z = "True";`

- | | | |
|---------------------------------------------|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a. <code>x z</code> | <u>s</u> d b n | z is the wrong type for an argument of <code>()</code> , but because in this case it would not be needed, a dynamic typechecker would not discover the error. |
| b. <code>x && z</code> | s d <u>b</u> n | Again z is the wrong type for an argument of <code>()</code> , but this time it will be evaluated, and a dynamic typechecker would discover the error. |
| c. <code>x && y</code> | s d <u>b</u> n | Same reason as (b). |
| d. <code>if x then "y" else z</code> | s d b <u>n</u> | $x :: Bool$ and "y", $z :: String$, which satisfies the type requirements of if...then...else . |
| e. <code>if x then y else z</code> | <u>s</u> d b n | $x :: Bool$ but y and z have different types. This would be treated as an error by a static type-checker, but a dynamic typechecker would simply type the expression as <code>Int</code> . |