Question 1. Write a Haskell program

\[ \text{yesOrNo :: String \to IO Bool} \]

such that (yesOrNo str) prints the question str, reads the user’s reply from the keyboard, and returns True when the first letter of the reply was ‘y’ or ‘Y’, otherwise False is returned. For example (the text the user types is \[ \text{gray} \]):

\[ \text{Main> yesOrNo "Do you want toast?"} \]
Do you want toast?
Yes, thank you
True

Question 2. Background. Fractran is a Turing-complete programming language due to John Conway. A Fractran program is an ordered list of positive fractions together with an initial positive integer input \( n \). The program is run by updating the integer \( n \) as follows:

1. For the first fraction \( f \) in the list for which \( n \cdot f \) is an integer, replace \( n \) by \( n \cdot f \).
2. Repeat rule 1 until no fraction in the list produces an integer when multiplied by \( n \), then halt.

For example, the Fractran program

\[
\left[ \frac{455}{33}, \frac{11}{13}, \frac{1}{11}, \frac{3}{7}, \frac{11}{2}, \frac{1}{3} \right]
\]
takes any input of the form \( 2^a 3^b \) to the output \( 5^a b \). E.g.,

\[
2^3 3^2 = 36 \sim 36 - 11 \times 2 = 198 \sim 198 \times \frac{455}{33} = 2730 \sim \ldots \sim 5625 \times \frac{1}{3} = 1875 \sim 1875 \times \frac{1}{3} = 625 = 5^2 2^2.
\]
Your Problem. Represent fractions as pairs of Integers and Fractran programs as lists of pairs of Integers. So the example Fractran program above would be

\[
(\{455,33\}, (11,13), (1,11), (3,7), (11,2), (1,3)).
\]

Write a Haskell functions

\[ \text{run :: [(Integer,Integer)] \to Integer \to Integer} \]

such that \( \text{run} \ p \ n \) is the result of running Fractran program \( p \) on input \( n \). (You may assume each pair \( (a,b) \) in the program list has \( a,b > 0 \).) Recall that \( \text{quotRem} \ a \ b \) returns the pair \( (q,r) \) where \( q \) is the quotient of \( a \div b \) and \( r \) is the remainder of \( a \div b \). (Obvious Hint: \( a \cdot b \div c = (a \cdot b) \div c \).)

NOTE: For Questions 3–5, all that is required is that you give the numbers returned and/or printed. However, if you are wrong, showing your work may get you partial credit.

Question 3. Consider the expression:

\[
\text{let } a = 10
\]
\[ \text{in let } g = \lambda z.(a \ast z) \]
\[ \text{in } (g \ 3) + (\text{let } a = 100 \text{ in } (g \ 4)) \]

What is the expression’s value under call-by-value/lexical-scoping?
(a) lexical scoping?  (b) dynamic scoping?

Question 4. Assume all locations start out with contents 0.
Consider:

\[
\text{let } f = \lambda x.\{ \ell_0 := 1 + !\ell_0; \ \text{return} \ (1000 + x) \} \\
\text{in let } b = (f \ \ell_1) \\
\text{in } \{ \ell_1 := 10; \ell_2 := !\ell_2 + b; \ell_1 := 100; \ell_2 := !\ell_2 + b; \} \\
\text{print } (\ell_0, \ell_1, \ell_2) \}
\]

What is the expression’s value lexical-scoping and (a) call-by-value?  (b) call-by-name?

Question 5. Assume all locations start out with contents 0.
Consider:

\[
\text{let } t = 100 \\
\text{in let } h = \lambda u.\{ \ell := !\ell + t; \ \text{return} \ u \} \\
\text{in let } g = \lambda t.(t + (t + (h \ 1))) \\
\text{in let } z = (g \ (h \ 10)) \\
\text{in print } (z, !\ell) \\
\]

What does the above print under:
(a) call-by-value/lexical-scoping? (b) call-by-name/lexical-scoping? (c) call-by-value/dynamic-scoping?
### Practice Questions for Quiz 5

#### CIS 352: Programming Languages

**An answer for Question 1**

```haskell
yesOrNo question = do { putStrLn question
    ; (c:_)<-getLine
    ; return (c=='y'|c=='Y')
}
```

**An answer for Question 2**

```haskell
run n prg = step n prg
    where step n [] = n
          step n ((a,b):fs)
            | r==0 = step q prg
            | otherwise = step n fs
            where (q,r) = quotRem (n*a) b
```

**An answer for Question 3(a)**

#### ENVIROMENT EXPRESSION

| $\rho_0$ | $\emptyset$ |
| $\uparrow$ | $\uparrow$ |
| $\rho_1$ | $a \mapsto 10$ |

- $g$ binds $(\lambda z. a + z) \rho_1$
- $\rho_2: g \mapsto (\lambda z. a + z) \rho_1$
- $\rho_3: z \mapsto 3 \Rightarrow \rho_1$
- $\rho_4: a \mapsto 100 \Rightarrow \rho_2$
- $\rho_5: z \mapsto 4 \Rightarrow \rho_1$

So $((g3) + (let\ a = 100 in\ (g4)))\rho_2$ evaluates to $30 + 400 = 430$.

**An answer for Question 3(b)**

### ENVIRONMENT EXPRESSION

| $\rho_0$ | $\emptyset$ |
| $\uparrow$ | $\uparrow$ |
| $\rho_1$ | $a \mapsto 10$ |

- $g$ binds $(\lambda z. a + z) \rho_1$
- $\rho_2: g \mapsto (\lambda z. a + z) \rho_1$
- $\rho_3: z \mapsto 3 \Rightarrow \rho_2$
- $\rho_4: a \mapsto 100 \Rightarrow \rho_2$
- $\rho_5: z \mapsto 4 \Rightarrow \rho_1$

So $((g3) + (let\ a = 100 in\ (g4)))\rho_2$ evaluates to $30 + 400 = 430$.

**An answer for Question 4(a)**

First note that $\ell_0$ will count how many times $f$ gets called.
- In the call $(f \ell_1)$, $\ell_1 = 0$ and so we end up with $\ell_0 = 1$ and $1000 + 0 = 1000$ being returned.
- Then $b$ is bound to $1000$.
  - So since $b \equiv 1000$, the `{ $\ell_1$ : 10; ... }` section is equivalent to:
    ```haskell
    { $\ell_1$ : 10; $\ell_2$ : $\ell_2 + 1000$; $\ell_1$ : 100; $\ell_2$ : $\ell_2 + 1000$; print ($\ell_0$, $\ell_1$, $\ell_2$) }
    ```
  - We thus end up with $!\ell_0 = 1$, $!\ell_1 = 100$, and $!\ell_2 = 2000$ and $(1, 100, 2000)$ is printed.

**An answer for Question 4(b)**

- The `let` binds $b$ to the thunk $(f \ell_1)\rho_0$ where we don’t really care what is in $\rho_0$ since the expression has no variables.
  - So since $b \equiv (f \ell_1)\rho_0$, the `{ $\ell_1$ : 10; ... }` section is equivalent to:
    ```haskell
    { $\ell_1$ : 10; $\ell_2$ : $\ell_2 + (f \ell_1)$; $\ell_1$ : 100; $\ell_2$ : $\ell_2 + (f \ell_1)$; print ($\ell_0$, $\ell_1$, $\ell_2$) }
    ```
  - We thus end up with $!\ell_0 = 2$ (since $f$ is called twice), $!\ell_1 = 100$, and $!\ell_2 = 2110$ and $(2, 100, 2110)$ is printed.
An note about Question 5(a)

Under lexical scoping, the t in h always refers to the t declared in the top line which will have the value 100. So under lexical scoping, h will increase ℓ by 100 and return its argument. So we’ll short circuit the h-evaluations for the lexical-scoping cases.

**Part (a).**

<table>
<thead>
<tr>
<th>Environment</th>
<th>Expression</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>ρ₀:</td>
<td>/ 0</td>
<td>!ℓ = 0</td>
</tr>
<tr>
<td></td>
<td>let t = 100...</td>
<td></td>
</tr>
<tr>
<td>ρ₁:</td>
<td>t → 100</td>
<td>!ℓ = 0</td>
</tr>
<tr>
<td>ρ₂:</td>
<td>h → (λu. {...}) ρ₁</td>
<td>!ℓ = 0</td>
</tr>
<tr>
<td>ρ₃:</td>
<td>g → (λt. {...}) ρ₁</td>
<td>!ℓ = 0</td>
</tr>
</tbody>
</table>

Thus z gets bound to 21 and, since h is evaluated twice, ℓ’s value is 100+100=200. Therefore, **(21,200)** is printed.

An answer for Question 5(b)

This is largely a repeat of part (a), except that in the call (g(h 10)) ρ₃, t is bound to the thunk (h 10) ρ₃. So in computing t + t + (h 1), h is called three times. Hence we end up with **(21,300)** being printed.

An answer for Question 5(c)

This is largely a repeat of part (a), except that in the call (g(h 10)) ρ₃, t is bound to the thunk (h 10) ρ₃. So in computing t + t + (h 1), h is called three times. Hence we end up with **(21,300)** being printed.
Practice Questions for Quiz 5

CIS 352: Programming Languages

Call-by-value, lexical-scoping

\[ \rho \vdash (e_1, s) \Downarrow \text{V} \left\langle \lambda x. \hat{e}_1 \right\rangle \]

\[ \rho \vdash (e_2, s') \Downarrow \text{V} \left\langle v_2, s'' \right\rangle \]

\[ \hat{\rho} [x \mapsto v_2] \vdash (\hat{e}_1, s'') \Downarrow \text{V} \left\langle v, s''' \right\rangle \]

App:

\[ \rho \vdash ((e_1 e_2), s) \Downarrow \text{V} \left\langle v, s''' \right\rangle \]

Var:

\[ \rho \vdash \langle x, s \rangle \Downarrow \text{V} \left\langle v, s \right\rangle \quad (v = \text{lookup}(\rho, x)) \]

Fun:

\[ \rho \vdash \langle \lambda x.e, s \rangle \Downarrow \text{V} \left\langle (\lambda x.e) \rho \right\rangle \]

Call-by-value, dynamic-scoping

\[ \rho \vdash (e_1, s) \Downarrow \text{V} \left\langle \lambda x.e'_1 \right\rangle \]

\[ \rho \vdash (e_2, s') \Downarrow \text{V} \left\langle v_2, s'' \right\rangle \]

\[ \rho[x \mapsto v_2] \vdash (\hat{e}_1, s'') \Downarrow \text{V} \left\langle v, s''' \right\rangle \]

App:

\[ \rho \vdash ((e_1 e_2), s) \Downarrow \text{V} \left\langle v, s''' \right\rangle \]

Var:

\[ \rho \vdash \langle x, s \rangle \Downarrow \text{V} \left\langle v, s \right\rangle \quad (v = \text{lookup}(\rho, x)) \]

Fun:

\[ \rho \vdash \langle \lambda x.e, s \rangle \Downarrow \text{V} \left\langle (\lambda x.e) \rho \right\rangle \]

Call-by-name, lexical-scoping

\[ \rho \vdash (e_1, s) \Downarrow \text{N} \left\langle \lambda x. \hat{e}_1 \right\rangle \]

\[ \rho \vdash (e_2, s') \Downarrow \text{N} \left\langle v_2, s'' \right\rangle \]

\[ \hat{\rho} [x \mapsto v_2] \vdash (\hat{e}_1, s'') \Downarrow \text{N} \left\langle v, s''' \right\rangle \]

App:

\[ \rho \vdash ((e_1 e_2), s) \Downarrow \text{N} \left\langle v, s''' \right\rangle \]

Var:

\[ \rho \vdash \langle e, s \rangle \Downarrow \text{N} \left\langle v, s' \right\rangle \quad (\rho' = \text{lookup}(\rho, x)) \]

Fun:

\[ \rho \vdash \langle \lambda x.e, s \rangle \Downarrow \text{N} \left\langle (\lambda x.e) \rho \right\rangle \]

Call-by-name, dynamic-scoping

\[ \rho \vdash (e_1, s) \Downarrow \text{N} \langle \lambda x.e'_1 \rangle \]

\[ \rho \vdash (e_2, s') \Downarrow \text{N} \langle v_2, s'' \rangle \]

\[ \rho[x \mapsto v_2] \vdash (\hat{e}_1, s'') \Downarrow \text{N} \langle v, s''' \rangle \]

App:

\[ \rho \vdash ((e_1 e_2), s) \Downarrow \text{N} \langle v, s''' \rangle \]

Var:

\[ \rho \vdash \langle e, s \rangle \Downarrow \text{N} \langle v, s' \rangle \quad (e = \text{lookup}(\rho, x)) \]

Fun:

\[ \rho \vdash \langle \lambda x.e, s \rangle \Downarrow \text{N} \langle (\lambda x.e) \rho \rangle \]

Types of some Haskell built-ins

| Symbol (\(|\) | (\&\&) | (\<<\)>> | (\|\|) | (\\) |
|---|---|---|---|---|
| Union | :| (\text{Eq} \ a) => \[a\] -> \[a\] -> \[a\] | \text{Bool} -> \text{Bool} -> \text{Bool} | \text{a} -> \[a\] -> \[a\] | \text{[a]} -> \text{a} |
| Read | :| (\text{Eq} \ a) => \text{String} -> \text{a} | \text{a} -> \text{[a]} | \text{String} -> \text{IO} | \text{String} |
| PutChar | :| \text{Char} -> \text{IO} (\) | \text{Char} -> \text{IO} (\) | \text{String} -> \text{IO} (\) | \text{String} |
| PutStr | :| \text{String} -> \text{IO} (\) | \text{String} -> \text{IO} (\) | \text{String} -> \text{IO} (\) | \text{String} |
| Read | :| \text{Read} a => String -> \text{a} | \text{Read} a => String -> \text{a} | \text{Read} a => String -> \text{a} | \text{Read} a => String -> \text{a} |
| ZipWith | :| (\text{Eq} \ a) => (\text{b} -> \text{c}) -> (\text{a} -> (\text{b} -> \text{c})) | (\text{a} -> (\text{b} -> \text{c}) -> (\text{a} -> (\text{b} -> \text{c}))) | (\text{a} -> (\text{b} -> \text{c}) -> (\text{a} -> (\text{b} -> \text{c}))) | (\text{a} -> (\text{b} -> \text{c}) -> (\text{a} -> (\text{b} -> \text{c}))) |

---

Page 4 of 4

13 April 2017