Quiz 3; Tuesday, January 27; 5 minutes; 5 points

[Solutions follow on next page]

1. Does the Java expression **`x + y == z`** have a side-effect? If so, what is it?

2. Write a function named **`add`** that can add two integers. You may use an explicit type specification, like **`::Integer`**, if you wish.

3. What is the type of the **`add`** function you just wrote? Use parentheses to show associativity.

4. Write a function **`eq3`** that returns **`True`** if and only its three arguments are equal. Assume the **`==`** and **`&&`** operators work just like they do in Java. **Don't worry about types**.

```
> eq3 5 5 5
True
```

5. The **`toLower`** function in the **`Data.Char`** module works like this:

```
> toLower 'A'
'a'
```

   What's the type of **`toLower`**?

6. Here are the types of the functions **`not`** and **`isLower`**:

```
not :: Bool -> Bool
isLower :: Char -> Bool
```

   What's the **type** of the result of **`not isLower 'x'`** ?

7. The function **`f x y z = x + y + z::Int`** has type **`Int -> Int -> Int -> Int`**. What are the types of the following expressions?

```
f 10 20

f 10 20 30
```

8. Add parentheses to the following expression to show the order in which operations would be done.

```
f  3  p  +  5  *  x  y
```

9. Name one general, discipline-independent characteristic of a paradigm, as defined by Kuhn.

10. Name one characteristic of the functional programming paradigm.

11. (Extra credit) Write a version of **`eq3`** from question 4 above that uses one or more guards.

12. (Extra credit) Haskell has both **`Int`** and **`Integer`** types. Why?

13. (Extra credit) What does REPL stand for?

Quiz 3 Solutions
Tuesday, January 27, 2015; 5 minutes (*extended 30 seconds*); 5 points
DRAFT; in progress

1. *Does the Java expression* **`x + y == z`** *have a side-effect? If so, what is it?*

   It has no side-effects.

   A few students wrote out a complete sentence, like the above or maybe "No, it does not have a side effect." That can chew up a lot of time. Strive for short answers. "No" is enough.

2. *Write a function named* **`add`** *that can add two integers. You may use an explicit type specification, like* **`::Integer`**, *if you wish.*

   ```
   add x y = x + y :: Int
   ```

3. *What is the type of the* **`add`** *function you just wrote? Use parentheses to show associativity.*

   ```
   Int -> (Int -> Int)
   ```

   Several students said just **`Int`**, or **`Integer`**. That's the type of the value produced by **`add`** but the type of a function includes the type of both inputs and output. A simple rule: The type of a function is what's reported by **`:type`**, *after the* **`::`**. Example:

   ```
   > :type add
   add :: Int -> Int -> Int
   ```

   The output of **`:type`** is a little chatty. We pronounce that **`::`** as "has type", so we'd literally read that output as "**`add`** has type **`Int -> Int -> Int`**". It's important to recognize that "**`add ::`**" is not part of the type of **`add`**; the type is simply "**`Int -> Int -> Int`**".

   Some said **`Num -> Num -> Num`**, and that's wrong; **`Num`** is a type class, not a type. Type classes appear in types only as part of a *class constraint*, like in **`Num a => a -> a`** and **`(Real a, Fractional b) => a -> b`**. In our limited Haskell world, if a type includes a class constraint, it will appear at the beginning of the type.

   `Int` is an instance of the type class **`Num`**. That's evidenced in the output of both **`:show Int`** and **`:show Num`** by the line **`instance Num Int`**.

4. *Write a function* **`eq3`** *that returns* **`True`** *if and only if its three arguments are equal. Assume the* **`==`** *and* **`&&`** *operators work just like they do in Java.* ***Don't worry about types***.

   ```
   > eq3 5 5 5
   True
   ```

   Solution:

   ```
   eq3 x y z = x == y && y == z
   ```

   Several students imagined an **`eq`** function, perhaps extending the idea of the **`add`** and **`add3`** examples. There's no **`eq`** function in the Prelude but I didn't deduct for it.

   Several students did this:

```
eq3 x y z = x == y == z
```

I was mostly interested in seeing a mostly function definition of some sort, so I didn't deduct for the above. Take a minute and work out the type of the above function. Note that to actually try it, you'll need to add some parentheses, like **`(x == y) == z`**. (See if you can figure out why that is, too.)

I was surprised by the number of students who didn't take advantage of the transitivity of equality and had something like this:

```
eq3 x y z = x == y && y == z && x == z
```

> Comparative moment: Here's a case where equality in JavaScript is not transitive:
>
> ```
> > [empty, zero, zerochar]
> [ '', 0, '0' ]
>
> > empty == zero
> true
>
> > zero == zerochar
> true
>
> > empty == zerochar
> false
> ```
>
> For something similar (and more) in PHP, see **`http://phpsadness.com/sad/52`**.

5. *The **`toLower`** function in the **`Data.Char`** module works like this:*

   ```
   > toLower 'A'
   'a'
   ```

   *What's the type of **`toLower`**?*

   ```
   Char -> Char
   ```

   Like some students said just **`Int`** for the type of **`add`** above, some said just **`Char`** for this one. **Remember: The type of a function comprises both the type of the inputs and the type of the value produced.**

6. *Here are the types of the functions **`not`** and **`isLower`**:*

   ```
   not :: Bool -> Bool
   isLower :: Char -> Bool
   ```

   *What's the type of the result of **`not isLower 'x'`** ?*

   According to my tally only three students got this one right but it's very close to the **`signum negate 2`** example on slide 43. Remember that function application, expressed with juxtaposition—two values beside each other with no intervening symbols—is left associative. Thus,

   ```
   not isLower 'x'
   ```

   means

   ```
   (not isLower) 'x'
   ```

   and is an error!

   You might think I should have added "Or, if the expression produces an error, state why." but I believe questions like this are fair game. Likewise, I might ask a 127A student, "What's the output of

`System.out.println(x y z)`? **Recognizing when something is wrong is an important part of mastering a subject**. Another example: "In what year did man first walk on Mars?"

7. *The function* ***`f x y z = x + y + z::Int`*** *has type* ***`Int -> Int -> Int -> Int`***. *What are the types of the following expressions?*

   `f 10 20`
   `Int -> Int`
   I counted "**`<function>`**" as correct on this quiz but that's <u>not a type</u>; it's a simple representation of a kind of value that Haskell considers to be unprintable. (Do you agree function values are unprintable?) **`<function>`** won't get counted as correct for a type in the future.

   `f 10 20 30`
   `Int`

   If you don't understand those answers, take another look at the **`ex-partialapps.html`** set of exercises and/or come and see me.

   Here's a simple approach that's usually right for questions like this: for each supplied argument, scratch off the leftmost "***`TYPE ->`***" in the function's type. The types are simple in this case—all just **`Int`**—but the types can be arbitrarily complicated.

8. *Add parentheses to the following expression to show the order in which operations would be done.*

   `f 3 p + 5 * x y`

   I was truly disappointed with the dismal results on this one! A full answer is below but anybody who recognized that **`f 3 p`** and **`x y`** **<u>are function calls(!)</u>** got full credit.

   `(((f 3) p) + (5 * (x y)))`

   <u>Maybe think of juxtaposition—two values side by side—as a highest precedence "invisible" binary operator</u>.

`let` expressions

If you search for "let it be" in LYAH you'll find where it talks about ***`let`*** *expressions*. I don't use or cover them in the slides because they're similar to **`where`** clauses and I think that can be a source of confusion early on. However, I did use a **`let`** expression to test the expressions above. Here's what I did:

```
> let {f x y = x + y; p = 97; x = negate; y = 1} in f 3 p + 5 * x  y
95

> let {f x y = x + y; p = 97; x = negate; y = 1} in (((f  3) p)+(5*(x  y)))
95
```

As you can see, inside the braces I bind names to several values and then use those bindings in the expression that follows **`in`**. The GNU Readline facilities (slide 42) let me edit and redo that one-liner.

Instead of writing the addition function **`f`** on the spot I could have simply bound **`f`** to the addition operator, like this:

```
> let {f = (+); p = 97; x = negate; y = 1} in f  3  p  +  5  *  x  y
95
```

9. *Name one general, discipline-independent characteristic of a paradigm, as defined by Kuhn.*

See slides 3-4. A good, short answer to have at your fingertips is, "a vocabulary". For example, "partial application" and "currying" are part of the vocabulary of the paradigm of functional programming.

There was a fair amount of confusion between Kuhn's definition of "paradigm", which can apply to any area of study, and elements of programming paradigms, evidenced by answers like "syntax", "expressions", "object-oriented", "procedural", and "modules".

10. *Name one characteristic of the functional programming paradigm.*

See slides 23-24. Of all those listed I'd say that "functions are values" is the one absolute must.

11. *(Extra credit) Write a version of* **eq3** *from question 4 above that uses one or more guards.*

```
eq3 x y z
    | x == y && y == z = True
    | otherwise = False
```

There were lots of interesting manglings on this one but it was graded very liberally.

12. (Extra credit) Haskell has both **Int** and **Integer** types. Why?

Values of type **Int** are "word"-sized integers. They are space-efficient and operations on them are fast. Values of type **Integer** can hold arbitrarily large (or small) values.

Haskell uses the **Int** type to great advantage wrt. performance but I consider the mix of types to be a wart. Icon, for example, supports arbitrary precision integers but the implementation switches between a word-sized representation for small values and a data structure for large values as needed. The programmer only sees one type: **integer**. Ralph Griswold felt that languages should be simple and consistent, and that the burden of making that so should fall on the implementors of the language.

Python, like Icon but unlike Haskell, supports arbitrary precision integers and only has one integer type: **int**. I'm not familiar with the implementation of Python and don't know whether Python switches between representations as needed, like Icon.

Many people are surprised to learn that Haskell doesn't provide any sort of overflow checking on Int values. Note the difference between **Int** and **Integer** values produced below.

```
> 29408329043284028*8204230420424823
241272707750773653786886810627044
it :: Integer

> 29408329043284028*8204230420424823::Int
8300605119622015972   -- WAT?
it :: Int
```

Googling for "haskell int vs integer" turns up lots of good discussion about the two types.

Chapter 18 in H10 has the official word on the **Int** type. You'll see there are **Int8**, **Int16**, **Int32**, and **Int64** types, too. (Remember, "H10" is my abbreviation for the Haskell 2010 Language Report.)

It's important to understand that choosing whether to provide a single integer type or multiple integer types is simply a language design decision; there's no right or wrong answer. A single type conserves the mental footprint of the user but multiple types offer speed and space benefits that can be dramatic, even pivotal, in some cases.

I was pleasantly surprised to see how many students got this one right. Maybe I said the right thing in class about **`Int`** vs. **`Integer`** and/or it connected well to existing knowledge.

Another comparative tidbit: JavaScript has only a single numeric type: **`number`**.

13. *(Extra credit) What does REPL stand for?*

Real-eval-print loop. Almost everybody got this one.