Brent Yelle
CSCI 3210, Project 1
April 17, 2023

QUESTION 1

Haskell's first version—Haskell 1.0—was released in 1990. There is no single author or small group of authors of Haskell; the online report for Haskell 2010 lists no less than 45 "main" creators, as well as many more contributors. Officially, the group that has authored Haskell is called the "Haskell Committee," formerly the "Functional Programming Language Committee."

Haskell was *named* after Haskell Curry, a researcher who essentially founded the field of combinatory logic after working on in lambda calculus. He and other influential mathematicians like Alonzo Church and John Rosser are credited with laying the foundation for functional programming.

(Hudak, Hughes, Jones, & Wadler, 2007)

QUESTION 2

The most recent *official* version of Haskell is Haskell 2010, released in July 2010. However, the most widely used implementation, GHC (Glasgow Haskell Compiler), has expanded upon the language with several extensions, the most recent being GHC2021, released with GHC version 9.2.1 on October 21, 2021.

(Lipovača, 2011)

QUESTION 3

The birth of Haskell can be traced to September 1987 at a conference in Portland, OR that agreed that functional programming as a whole was being held back by the lack of a common standard and common language. As a result, a committee was formed to create such a standard & language. According to the Haskell 2010 report, the exact aims of the committee were to make a language that fulfilled all of the following goals:

1) It should be suitable for teaching, research, and applications, including building large systems.
2) It should be completely described via the publication of a formal syntax and semantics.
3) It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4) It should be based on ideas that enjoy a wide consensus.
5) It should reduce unnecessary diversity in functional programming languages.

(Hudak, Hughes, Jones, & Wadler, 2007)

QUESTION 4

In the 1980s, there were many researchers who wanted to work on design and implementation of pure functional languages that used lazy evaluation, but without any central "core" that everyone could agree on to use as a base. Haskell's purpose was to create such a core language to serve as the basis for functional programming research, following the guidelines laid out above.

(Hudak, Hughes, Jones, & Wadler, 2007)

QUESTION 5

As it was made as a way to bridge the gap across the many functional programming languages of the late 1980s, Haskell has had influences from several languages, the most significant being Miranda, a successor to KGP. As far as individual languages go, the Haskell 2010 report notes that these languages were "particularly influential" to Haskell: Miranda, Lisp, ISWIM, APL, FP, ML, Hope, Clean, Id, Gofer, and Sisal.

Miranda, a successor to earlier languages **SASL** and **KRC**, was the most direct influence on Haskell. In fact, Miranda's creator David Turner was first asked if

Miranda could be used as the basis of functional programming research (before Haskell was even conceived), but he declined, keeping Miranda proprietary. Haskell's features taken from Miranda include non-strictness, lazy evaluation, polymorphic typing, sum-of-products definitions for algebraic data types, and the optional use of "declaration style" (using |, where, otherwise, etc.) for defining functions in terms of multiple different expressions. Miranda also was the origin of Haskell's rule that types and type constructors had to start with uppercase letters, whereas variables and functions had to start with lowercase ones.

Lisp, particularly the dialects Daisy and Scheme, influenced Haskell both directly and inversely, that is, certain elements of them were adopted for Haskell while others were deliberately avoided. Adopted elements include the capability of creating higher-order functions (functions that return functions) and the optional use of "expression style" (let, if, then, else, etc.) for defining functions. Deliberately avoided features include Lisp's extreme use of parentheses, as well as Lisp being a strict, non-lazy, imperative language.

The language **ML** similarly influence Haskell directly and inversely. Direct influences include polymorphic typing, robust type inference, the use of higher-order functions, and eliminating the need for semicolons. Haskell deliberately avoided certain features of ML such as ML's being a strict, non-lazy language. However, a derivative of ML known as **Lazy ML** did implement non-strictness and lazy evaluation, and furthermore demonstrated that lazy functional programs can actually be compiled, not just interpreted—Haskell, too, has inherited this feature of being both compile-able and interpret-able.

FP and the derivative μFP influenced the functional programming world, and thus Haskell, by pioneering the use of higher-order functions in order to model physical hardware, particularly synchronous circuits, through lazy evaluation.

Parts of the influences from **ISWIM** include the avoiding of semicolons and the idea of sequential program control, having higher-order functions, and using extensive pattern matching for function definitions.

The language **Hope** was very influential in that it was one of the first languages to introduce introduced “algebraic data types”—composite types made from other types adjoined together—for functional programming. However, Hope’s algebraic types were distinct in that they separated “sum” and “product” algebraic types. Hope’s creator, Burstall, was also a pioneer of the idea that functional programs can be written as collections of equations that are matched by extensive pattern matching, from top to bottom over the program (matching the first available one), which was implemented in Hope. The derivative language Hope+ instead used “tightest match”-style pattern matching, which Haskell’s creators considered but decided against using.

The language **Clean** shares many similarities with Haskell and was designed in much the same environment as Haskell later was: it has higher-order functions, lazy evaluation, extensive list comprehension, referential transparency, and currying—the breaking down of two-parameter functions into functions that return functions of one variable; this is what allows Haskell to avoid needing parentheses around arguments.

Haskell’s concurrent and parallel programming structures called “MVar”s were largely based on similar “M-Structures” from the programming language **Id**.

Gofer—originally a Haskell implementation that eventually branched out into its own language—was the first language to include multi-parameter type classes and constructor classes, which were initially not included in Haskell, but were implemented in the 2010 update.

The Haskell 2010 report gives thanks to **Sisal** as an influence, though without any explanation. At the very least, shared features between Haskell and Sisal include being purely functional with strong typing. Sisal has *definitely* been a major influence on the Haskell derivative pH ("parallel Haskel"), which aims to make the language as parallelized/parallelizable as possible.

(Hudak, Hughes, Jones, & Wadler, 2007)
(Marlow, 2010)

QUESTION 6

I personally really enjoy the concept of currying, where a "function of multiple variables" is really a function that maps to a chain of functions. For instance, if we have the definition

```
f a b c = a * b + c
```

where **a**, **b**, and **c** are integers, then the type of **f** is **Int -> Int -> Int -> Int**, not **(Int, Int, Int) -> Int**. This allows us to partially plug in values to **f**, letting us make, say, a single-variable function **f 2 3**, which can be defined as a new function or applied to a final integer to get an integer result.
The reason I enjoy currying is that reminds me fondly of mathematical *functionals* (elements of algebraic dual spaces) and calculus of variations, which I did a large project on during my master's degree several years ago.

I also much prefer the fact that Haskell function calls are much clearer and less cumbersome to write than in Scheme, particularly in that Haskell doesn't require

parentheses around every function, and that Haskell allows you to make any prefix function into an infix function by surrounding it with backticks, like: **`elem 3 [1, 2, 3]`** becoming **`3 `elem` [1, 2, 3]`** (checking if 3 is an element of the list [1, 2, 3].

QUESTION 7

Although polymorphic types and typeclasses make manipulating values very convenient, they can make reading and debugging code very inconvenient, as I often need to look up the definition of a typeclass to see what types it actually contains, and the Haskell specification document actually contains a large, cumbersome graph of the typeclass hierarchy.

In addition, Haskell supports n-tuples up to $n = 15$, but it has very limited built-in tuple manipulation in contrast to its robust list manipulation.
For instance, there are only two built-in functions for accessing tuple elements: **`fst`** and **`snd`**, which grab the first and second elements, respectively, of a 2-tuple… and that's it. For all later elements, you need to write your own function like **`get_4th_of_6tuple (_, _, _, x, _, _) = x`**, and even then these are not easily extensible, as this definition above can then only grab the 4th element of a 6-tuple. Another frustrating issue with tuple manipulation is the inability to use functions like **`head`**, **`tail`**, **`init`**, or **`last`** with tuples—you must build these yourself as well. (Fortunately, there are some public packages that provide more robust tuple element access and manipulation, but I find it bizarre that these kinds of functions aren't supported in the definition of Haskell.)

QUESTION 8

Haskell uses only static typing, which acts as expected:

```
f :: Integer -> Integer
```

Here, the function **`f`** has a fixed type **`(Integer -> Integer)`**, meaning that it can only act on one **`Integer`** and can only ever return one **`Integer`**.

However, Haskell also supports *typeclasses*, which are abstract categories of classes unified by common operators. For example, all types that can use the equality operator **==** are contained in the **Eq** typeclass. Thus, we can fine a function like so:

```
g :: (Eq a) => a -> a
```

This reads as "**g** acts on a value of type **a** and returns a value of type **a**, and type **a** must be part of the **Eq** typeclass." Whatever "type **a**" is will be determined by the context. If a type *cannot* be determined by context, then an error will be thrown. For example, the operation **read** looks at a string and interprets it as a numeric value, but Haskell has no way of knowing what numeric type to interpret it as without context. So, if we declare a new variable **x** without declaring its type:

```
x = read "17"
```

This will throw an error. However, if we specify a type, we won't have problems. We can do this by declaring the type of **x** first:

```
x :: Integer
x = read "17"
```

Or by declaring it on the assignment line:

```
x = read "17" :: Integer
```

(Lipovača, 2011)

QUESTION 9

Haskell has the following basic data types:

- **Int** 32-bit signed integers
- **Integer** arbitrary-length integers (like Java's *BigInteger* class)
- **Float** 32-bit floating-point integers
- **Double** 64-bit floating-point integers
- **Bool** Boolean values (true/false)
- **Char** ASCII characters

There is a String type, but it is an alias of **`[Char]`** – a list of characters.
Compound data types include:

- Lists – contain many values, all of the same type
    - versatile for sorting,
- Tuples – contain many values, possibly of different types
    - useful as data containers

And of course, these compound data types can also themselves contain compound data types.

There are also functions, which are operators that act on a value and return either another value or another function. Functions that "act on multiple variables" are actually built as chains of functions, each successively acting on the next value in the parameter list—this is called *currying*. Functions' types are always notated as:

```
functionName :: type1 -> type2 -> … typeN
```

Where **`typeN`** is the return type and the previous **`type1`** through **`typeNminus1`** are all the types of the parameters.

In addition, Haskell has many polymorphic typeclasses, which are collections of classes that all have similar properties:

- **`Eq`** can use **`==`** and **`!=`**
- **`Ord`** can use **`>`**, **`<`**, **`>=`**, and **`<=`** (subset of **`Eq`**)
- **`Show`** can be converted to a printable string with **`show`**
- **`Read`** can be converted from a printable string with **`read`**
- **`Enum`** can use **`succ`** (successor) and **`pred`** (predecessor)
- **`Bounded`** have upper and lower bounds (**`Int`**, **`Float`**, **`Double`**)
- **`Num`** can use **`+`** and **`*`**
- **`Integral`** **`Int`** and **`Integer`**
- **`Floating`** **`Float`** and **`Double`**

…and Haskell functions can return objects that merely need to follow a typeclass, and it is up to the receiving function to narrow down which type to use it as.

Typeclasses can be used to make constraints on valid parameter types declarations, such as the following:

```
getBigger :: (Ord a) => a -> a -> a
getBigger x y
     | x > y        = x
     | otherwise    = y
```

Which will return the greater value between two arguments **x** and **y**, but will throw an error if it is given an argument for **x** or **y** that cannot use **>**.

Haskell has no (implicit) type coercion between types, meaning that one always has to explicitly make a type conversion. However, thanks to typeclasses, many of the basic functions actually return polymorphic types: for example, **read** always returns a **Read** type, which can then be narrowed down back to most numeric types like **Int** and **Float**:

For example:

```
-- explicitly declaring as an Int
myInt = 5 :: Int

-- converting to a string with show, then to a Read type
-- with read, then narrowing down to a Double.
myDouble = read (show myInt) :: Double
```

This will store the 64-bit floating-point value 5.0 in **myDouble**.

(Lipovača, 2011)

(Marlow, 2010)

(Hudak, Hughes, Jones, & Wadler, 2007)

QUESTION 10

Haskell does not support OOP-style classes at all, let alone having private or protected data; all data is therefore "public," to use C++ terminology.
The only way to make data values inaccessible would be to provide no function that is able to directly extract them from an object, or to create some kind of subroutine that prevents using data access without providing proper credentials.

QUESTION 11

As Haskell does not support OOP, it does not have inheritance in the traditional sense of the word. However, there is a type of inheritance that can be found in the declaration of implicit parameters of functions, which has been introduced in the implementation of GHC using the **?** operator:

```
grouper :: (?equiv :: a -> a -> Bool) => [a] -> [[a]]
grouper = {- insert algorithm that separates the input list
into equivalency classes based on ?equiv -}
```

This is a function that takes a list of **a**-types and returns a list of lists of **a**-types, and it involves the use of an implicit parameter **?equiv** that takes two **a**-types and returns a Boolean. When invoking **grouper**, one must qualify what Boolean operator the **?equiv** actually stands for using something like **where**:

```
makeListOfElems = grouper myList
     where ?equiv = (==)
```

…which will return a list where every former element is replaced by a singleton list containing that element.

However, you can also declare a new function in terms of **grouper** without qualifying the comparator:

```
getFirstEquivClass lst = head (grouper lst)
```

If one does this, then **getFirstEquivClass** will inherit the implicit parameter **?equiv** from **grouper**, and it will need to be qualified when invoking it:

```
thisFirst = getFirstEquivClass this List
     where ?equiv = isCoprime
```

…assuming we've defined the Boolean operator **isCoprime** beforehand.

(Jones & Reid, 1994-2004)

QUESTION 12

Matching a function call to a function definition is always done statically, as overloading functions to have different type(class) signatures is explicitly *forbidden*. Multiple definitions of a function can exist, and these are always treated as "or" declarations, with the first-listed option always being used. For example, given this trivial function to grab the first character of a string:

```
getHead :: [Char] -> Char      -- forward declaration
getHead []       = '\0'        -- option 1: empty list
getHead (hd:tl)  = hd          -- option 2: non-empty list
```

…then if we invoke **getHead []**, it matches to the first option and returns **'\0'**, never continuing on to the second option. This is a good feature because if we were to try and pattern-match **[]** to **hd:tl**—i.e., a list consisting of one element **hd** followed by the rest of the list **tl**—there would be an error, since there is no element in **[]** that could be matched to **hd**.

(Lipovača, 2011)
(Hudak, Hughes, Jones, & Wadler, 2007)

Bibliography

Arvind, Augustsson, L., Hicks, J., Nikhil, R., Jones, S. P., Stoy, J., & WIliams, J. (1993, September 1). *pH Manifesto - pH: a parallel Haskell*. Retrieved from Computation Structures Group - MIT - LCS: http://csg.csail.mit.edu/projects/languages/ph-manifesto.html

Hudak, P., Hughes, J., Jones, S. P., & Wadler, P. (2007). A History of Haskell: Being Lazy With Class. *Proceedings of the third ACM SIGPLAN conference on History of programming languages* (pp. 12-1:12-55). HOPL III.

Jones, M. P., & Reid, A. (1994-2004). *Chapter 6. Language extensions supported by Hugs and GHC - 6.5. Implicit Parameters*. Retrieved from The Hugs 98 User's Guide: https://www.haskell.org/hugs/pages/users_guide/implicit-parameters.html

Lipovača, M. (2011). *Learn You a Haskell for Great Good! : A Beginner's Guide.* San Francisco: No Starch Press.

Marlow, S. (. (2010). *Haskell 2010 Language Report.* Cambridge.