Introduction

Beginner Level

Core Language Fundamentals

Syntax and Basic Expressions

The candidate must understand fundamental syntax rules and scoping in Haskell. Specifically, the candidate should be able to

- *identify* which variable binding is in scope at a given point in the code
- *explain* how variable shadowing works when the same name is used in multiple scopes
- *distinguish* between variable bindings in function parameters, `case` expressions, and lambda expressions
- *identify* syntax errors in basic expressions (such as incorrect layout)
- *explain* how the layout rule affects the parsing of `do` and `case` expressions
- *recognize* guard expressions and understand their syntax in function definitions and `case` expressions
- *identify* and use various literal forms (integer: `42`, `0xFF`; floating-point: `3.14`, `1e-5`; character: `'a'`, `'\n'`; string: `"hello"`; list literals: `[1,2,3]`, `"abc"`)
- *construct* and *pattern match* on record types using record syntax, including field access and updates (e.g., `Person { name = "Alice", age = 30 }`, `Person { name = n }`, `person { age = 31 }`)
- *distinguish* between infix and prefix operator usage, including operator sections and backtick notation for infix function application (e.g., `x + y` vs `(+) x y`, `(+ 1)`, `(1 +)`, `x `div` y`)
- *identify* lexical rules for variable names (must start with lowercase letter or underscore, e.g., `x`, `_temp`, `myVar`), constructor names (must start with uppercase letter, e.g., `Just`, `Nothing`, `Person`), infix constructors (must start with `:`, e.g., `:`, `:|`), and infix operators (must start with a symbol character, e.g., `+`, `==`, `++`, `<$>`)

Sample Question

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell code and answer the questions below.

```haskell
data Person = Person { fullName :: String, age :: Int }

appendName newName p
    | age p < 18 = Nothing
    | null fullName = Just $ p { fullName =  newName }
    | otherwise = p { fullName = Just $ (fullName p) ++ " " ++
newName }
```

Questions:

1. What is the result of calling `appendName "Johnson" (Person "Alice" 25)` ?
2. In which scenario the `fullName` of the existing `Person` exchanged with the `newName` in its entirety?

References

1. Haskell 2010 Language Report. §3.17 Layout
2. Haskell 2010 Language Report. §4.4.3 Function and Pattern Bindings
3. Haskell 2010 Language Report. §2.4 Lexical Structure
4. Haskell Wikibook Variables and functions

Types and Type Inference

The candidate must understand Haskell's type system and be able to perform manual type inference for simple expressions and definitions. Specifically, the

candidate should be able to

- *infer* the most general type of simple function definitions without explicit type signatures
- *infer* the types of local variables, subexpressions, and intermediate values in simple code snippets
- *identify* type errors in simple erroneous programs, including mismatched types, functions applied to too many or too few arguments, and type class constraint violations
- *distiguish* between concrete types (e.g. `Int`, `Maybe Bool`) and type variables (e.g. `a`, `sometype`)
- *identify* constraints in type signatures and explain their meaning (e.g. `(Num a, Eq b) => ...`) and *explain* how such constraints affect the possible instantiations of type variables
- *recognize* basic type constructors and their structure, including lists (`[a]`), `Maybe a`, `Either a b`, tuples of various arities, and function types (`a -> b`)
- *explain* how type variables represent polymorphism in type signatures
- *apply* type inference rules for function application, including understanding how argument types constrain the function type
- *determine* whether a given expression type-checks and, if so, what its inferred type is
- *reason* about the relationship between concrete types and polymorphic types, including how polymorphic types can be instantiated to specific types

> **Note**
>
> This section focuses on basic type inference without type classes. Type class constraints and their role in type inference are covered in the Typeclasses section.

Sample Question for Types and Type Inference

> **Note**

> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell snippet and answer the questions below.

```haskell
x :: Int
x = 42

f a b = x + a + b

apply x f = f x x

sample1 = f 10 20
sample2 = apply 5 (\a b -> a + b)
sample3 = apply "hello" (\a b -> length a)
sample4 = apply True
```

Questions:

1. What is the most general type of `f`?
2. Explain how the type of `x` constrains the types of `a` and `b` in the definition of `f`.
3. What is the most general type of `apply`?
4. Explain how the expression `f x x` in the definition of `apply` constrains the relationship between the types of `x` and `f`.
5. Does the expression `sample1` type-check? If so, what is its type? If not, explain why.
6. Does the expression `sample2` type-check? If so, what is its type? If not, explain why.
7. Does the expression `sample3` type-check? If so, what is its type? If not, explain why.
8. Does the expression `sample4` type-check? If so, what is its type? If not, explain why.

References for Types and Type Inference

1. Learn You a Haskell - Types and Typeclasses
2. Haskell Wikibooks - Type basics

3. Real World Haskell - Types and Functions
4. Haskell for All - Type Inference for Plain Data

Pattern Matching

> ![NOTE] At this level we do not require advanced pattern matching techniques, including lazy patterns and pattern synonyms.

The candidate must understand pattern matching in function definitions and `case` expressions, and be able to use it to write idiomatic Haskell code. Specifically, the candidate should be able to

- *match* on data constructors, including nullary constructors, constructors with arguments, and nested patterns
- *match* on list patterns, including the empty list (`[]`) and cons patterns (`(x:xs)` or `(x:y:zs)`)
- *match* on tuple patterns, extracting individual components
- *recognize* the relationship between pattern matching on literals (characters, integers) and explicit equality tests
- *transform* non-idiomatic code using `if` - `then` - `else` chains combined with partial functions (such as `if isJust x then g (fromJust x) else h`) into idiomatic pattern matching
- *identify* when pattern matching is exhaustive and when it is not, understanding the implications of incomplete patterns
- *use* wildcard patterns (`_`) and `as` -patterns (e.g., `xs@(x:y:zs)`) appropriately
- *explain* the nuances of pattern matching in function definitions, `case` expressions, `let` bindings, anonymous functions (e.g. `\(x:xs) -> ...`), `where` clauses, `do` -blocks, and list comprehensions with respect to **exhaustiveness** of matching

Sample Question for Pattern Matching

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell snippet and answer the questions below.

```haskell
data ParseIntResult = Parsed Int | Failed

process :: ParseIntResult -> Int
process x = if isParsed x then getInt x + 1 else 0
  where
    isParsed (Parsed _) = True
    isParsed Failed = False
    getInt (Parsed n) = n

safeHead :: [Int] -> Int
safeHead xs = if null xs then 0 else head xs

countPairs :: [(Int, Int)] -> Int
countPairs pairs = length (filter (\(x, y) -> x == y) pairs)

sumPair :: (Int, Int) -> Int
sumPair p = fst p + snd p

getFirstTwo :: [Int] -> Int
getFirstTwo xs = let (a:b:_) = xs in a + b
```

Questions:

1. Rewrite the `process` function using pattern matching instead of `if-then-else` and helper functions.
2. Rewrite the `safeHead` function using pattern matching instead of `null` and `head`.
3. Rewrite the `countPairs` function using pattern matching and recursion instead of `filter` and `length`.
4. Rewrite the `sumPair` function using pattern matching instead of `fst` and `snd`.

5. Rewrite the `getFirstTwo` function using pattern matching in a `where` clause instead of a `let` binding.
6. What happens if `getFirstTwo` is called with a list that has fewer than two elements? How could you make the function safer?
7. Which of the rewritten functions are exhaustive (handle all possible cases)? Explain.

References for Pattern Matching

1. Haskell 2010 Language Report. §3.17 Pattern Matching
2. Haskell 2010 Language Report. §4.2.1 Algebraic Datatype Declarations
3. GHC User Guide. §5.2 Warnings and sanity checking (`-Wincomplete-patterns`)

Guards and Conditionals

- syntax of guards (and `if` - `then` - `else`), both in declarations and in `case` expressions
- ability to convert between guards and `if` - `then` - `else`
- implications of `if` - `then` - `else` being an expression-level and guard being a declaration-level construct

Let and where bindings

The candidate must understand the syntax and semantics of `let` and `where` bindings. Specifically, the candidate should be able to

- *use* the syntax of `let` and `where` , following the layout rule
- *determine* the result of expressions involving `let` and `where` bindings
- *distinguish* between `let` bindings and `where` bindings in terms of their scope and usage contexts

- *expand* `let` and `where` with multiple bindings into nested `let` and `where` and vice versa
- *transform* code from using `let` to using `where` and vice versa
- *explain* the implications of `let` being an expression-level construct and `where` being a declaration-level construct
- *identify* syntax errors in `let` expressions (such as missing `in` keywords or incorrect layout)
- *explain* how the layout rule affects the parsing of `let` and `where` expressions
- *inline* `let` and `where` bindings, and *abstract* common definitions into `let` and `where`
- *distinguish* between `let` in expressions and `let` in `do` blocks

Sample Question for Let and Where Bindings

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell code and answer the questions below.

```haskell
x = 42

processValue val =
  let x = 10
      result = case val of
        Nothing -> x
        Just y  -> let z = y + 5
                   in x * 2
  in result + x
  where
    x = 100
```

Questions:

1. What is the result of calling `processValue Nothing`?
2. What is the result of calling `processValue (Just 3)`?

3. Which binding of `x` is used in the expression `result + x` at the end of the `let` expression?
4. In the `case` expression, when the pattern `Just y` matches, which binding of `x` is used in `x * 2`?

References for Let and Where Bindings

1. Haskell 2010 Language Report. §3.12 Let Expressions
2. Haskell 2010 Language Report. §4.4.3 Function and Pattern Bindings
3. HaskellWiki. Let vs where

Modules and imports

The candidate must be able to understand how to manage importing and exporting functions, types and constructors from modules. This includes:

- *identify* which values have been imported
- *identify* if imported values are imported with a namespace
- *explain* the syntax used to manage importing and exporting values from modules
- *understand* how to remove values from scope using `hiding`.

Lists and list comprehensions

- understanding the range syntax for lists, both with a specified end such as `[1 .. 5]` and without such as `[1 ..]`, and also for other types in `Enum` than just numbers
- being able to understand the meaning of list comprehension syntax, and being able to convert between list comprehensions and applications of `map` / `filter` / `concat` for simple examples.
- being able to apply filters to values in the list comprehension using the filter syntax

Tuples and records

> **Note**
>
> We are not expecting the candidate to be familiar with GHC extensions for records, including the common extensions such as `RecordWildCards`, `NamedFieldPuns`, `DuplicateRecordFields`, or `OverloadedRecordDot`.

The candidate must understand tuples (fixed-size product types) and records (algebraic data types with named fields), and be able to read and write small programs using them. Specifically, the candidate should be able to

- *construct* tuple values of various arities (pairs, triples, etc.) and *pattern match* on them to extract components
- *infer* the types of simple tuple expressions (including nested tuples), and explain how tuple arity affects the tuple type (e.g. `(a, b)` vs `(a, b, c)` or `((a, b), c)` vs `(a, (b, c))`)
- *distinguish* tuples from lists, including key differences:
  - tuples can contain values of different types, lists cannot
  - tuples have distinct types for each arity, lists have one type for their elements regardless of how many elements they contain
  - in contrast to lists, we cannot easily operate on all elements of a tuple (due to their different types) using operations such as `map`, `filter` and folds
- *replace* the use of `fst` and `snd` (and similar functions) with pattern matching
- *refactor* code from using tuples to using custom `data` types (e.g. `(Date, Text)` vs. `data Holiday = MkHoliday { date :: Date, name :: Text }`)
- *define* small record types using `data` declarations with named fields, and *construct* and *deconstruct* (using pattern matching) record values
- *use* record field accessors (e.g. `firstName :: Employee -> String`) and record pattern matching (e.g. `Employee{ salary = salary }`) to read fields
- *use* record update syntax (e.g. `e { salary = salary e + 1 }`) to produce updated values

- *refactor* from positional constuctors (in patterns and expressions) to record syntax (e.g. from `Point 1 2` to `Point { x = 1, y = 2 }`)
- *recognize* common pitfalls such as confusing tuple syntax with list syntax, or accidentally reordering fields when using positional constructors instead of named fields

Sample Questions for Tuples and Records

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell snippet and answer the questions below.

```haskell
-- Part A: Tuples
scnd :: (a, b, c) -> b
scnd = undefined

thrd :: (a, b, c) -> c
thrd = undefined

swapPair (x, y) = (y, x)

-- Part B: Records
data Member = Member
  { firstName :: String
  , lastName  :: String
  , points    :: Int
  , tier      :: String
  } deriving (Show, Eq)

starterTier :: String -> String -> Member
starterTier fn ln = Member fn ln 50000 "Starter"

addBonus :: Int -> Member -> Member
addBonus pct m = Member (firstName m) (lastName m) newPoints (tier
m)
  where
    newPoints = points m + (points m * pct `div` 100)

label :: Member -> String
label (Member fn ln _ t) =
  ln ++ ", " ++ fn ++ " (" ++ t ++ ")"
```

Questions:

1. Fill in implementations for `scnd` and `thrd`
2. What is the type of the expression `(scnd (True, "x", 10), thrd (True, "x", 10))`? Assume literal `10` has type `Int`.
3. Explain why `snd (1,2,3)` is a type error, while `scnd (1,2,3)` type-checks.
4. Write an alternative definition of `addBonus` that uses pattern matching on the record rather than field accessors, and uses record update syntax to return the updated member.
5. Function `label` uses positional `Member` constructor for pattern matching. Replace it with record pattern matching and explain how this can benefit the code, considering possible future changes to the `Member` data type.

6. `starterTier` uses the positional `Member` constructor. Rewrite it using named fields (record construction syntax) and explain one benefit of doing so.

References for Tuples and Records

1. Haskell 2010 Language Report. §3.15 Tuples
2. Haskell 2010 Language Report. §4.2 User-Defined Datatypes (records and field labels)
3. Learn You a Haskell. Tuples
4. Learn You a Haskell. Record syntax
5. Learn You a Haskell. Syntax in Functions (pattern matching on tuples)

Case Expressions

The candidate must understand the syntax and semantics of `case` expressions. Specifically, the candidate should be able to

- *identify* when a `case` expression is exhaustive and when it is not
- *explain* the consequences of non-exhaustive `case` expressions (warnings and runtime behavior)
- *rewrite* pattern matching in function definitions to use `case` expressions and vice versa
- *identify* redundant patterns in `case` expressions and understand which pattern will match first
- *use* guards within `case` expression alternatives
- *rewrite* `if-then-else` expressions using `case` expressions on `Bool` values
- *explain* the differences between pattern matching in `case` expressions and pattern matching in function definitions in terms of where they can be used

Sample Question on Case Expressions

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell code and answer the questions below.

```haskell
data Priority = Low | Medium | High
data TaskStatus = Pending Int | InProgress String | Completed |
Failed String

data Task = Task { taskName :: String, priority :: Priority, status
:: TaskStatus }

processTask task = case task of
    Task name High (Pending days)
        | days > 10 -> "Critical: " ++ name ++ " pending for " ++
show days ++ " days"
        | days > 7 -> "Urgent: " ++ name ++ " pending for " ++ show
days ++ " days"
    Task name p (InProgress worker)
        | p == High -> if worker == ""
                       then "High priority task with no assigned
worker"
                       else "High priority task assigned to " ++
worker
    Task name Medium (Completed) -> "Completed: " ++ name
    Task name Low (Failed reason)
        | reason == "" -> "Task " ++ name ++ " failed with no
reason"
        | length reason > 20 -> "Task " ++ name ++ " failed: " ++
take 20 reason ++ "..."
        | otherwise -> "Task " ++ name ++ " failed: " ++ reason
```

Questions:

1. The condition `p == High` is giving a type error, due to a missing `Eq Priority` instance. Suggest a fix that does not require adding a type class instance, i.e. change the check in a way that does not require an `Eq` instance.

2. Is the outer `case` expression in `processTask` exhaustive? If not, identify which cases are not covered.
3. What is the result of calling `processTask (Task "Fix bug" High (Pending 10))` ?
4. What is the result of calling `processTask (Task "Write docs" High (InProgress "Alice"))` ?
5. What is the result of calling `processTask (Task "Test feature" Low (Failed "Connection timeout error occurred"))` ?
6. Explain how the guards in the `Task name High (Pending days)` pattern work. What happens if `days` is `5` ?
7. Rewrite the `if-then-else` expression in the `InProgress` branch using a `case` expression on `worker` . Is there a more idiomatic appoarch?
8. Rewrite the function to use pattern matching in the function definition instead of `case` expressions. Which approach is more appropriate here and why?

References for Case Expressions

1. Haskell 2010 Language Report. §3.13 Case Expressions
2. Haskell 2010 Language Report. §4.4.3 Function and Pattern Bindings
3. GHC User Guide. §5.2 Warnings and sanity checking (`-Wincomplete-patterns`)

Recursion

The candidate must understand recursive function definitions and be able to work with recursion over various data structures. Specifically, the candidate should be able to

- *complete* the definitions of simple recursive functions (e.g. on lists, but also other algebraic data types) according to a given specification
- *identify* base cases and recursive cases in recursive function definitions
- *implement* recursive functions on integers (e.g., factorial, Fibonacci)

- *implement* recursive functions on algebraic data types (e.g., lists, trees, custom recursive types)
- *implement* a recursive function with an accumulator parameter pattern to work with local state in a purely functional way
- *apply equational reasoning* to trace the execution of simple recursive functions to determine their output
- *recognize* when a recursive function can be expressed more idiomatically using higher-order functions like `map` or `filter`

Sample Questions on Recursion

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell snippets and answer the questions below.

Question 1: Fill in the holes `_a` and `_b` to complete the recursive definition of `lengths` to compute length (number of elements) for every nested list:

```haskell
lengths :: [[a]] -> [Int]
lengths []       = _a
lengths (xs:xss) = _b
```

Is there a way to implement `lengths` without explicit recursion?

Question 2: Consider the following datatype and function:

```haskell
data RoseTree = Leaf Int | Branch [RoseTree]

sumTree :: RoseTree -> Int
sumTree (Leaf x)       = x
sumTree (Branch trees) = sum (map sumTree trees)
```

Questions:

1. What is the value of `sumTree (Branch [Leaf 1, Branch [Leaf 2, Leaf 3], Leaf 4])`?
2. Rewrite `sumTree` using explicit recursion with an accumulator parameter instead of using `map` and `sum`.

References for Recursion

1. Learn You a Haskell for Great Good!. Recursion
2. HaskellWiki. Performance/Accumulating parameter
3. Well-Typed. Higher-Order Functions (see sections on Fast Reverse and Strict Accumulators)

Higher-Order Functions

The candidate must understand higher-order functions and be able to typecheck, reduce, and refactor expressions that use them. Specifically, the candidate should be able to

- *recall* the type signatures of `map`, `filter`, `zipWith`, `foldr` and `foldl'`
- *infer* the types of concrete applications of folds, `map`, and other higher-order functions
- *typecheck* expressions that pass functions as arguments (e.g. involving `flip`, partial application, or lambda-expressions)
- *evaluate* expressions that apply higher-order functions (e.g. `foo f x = f (f x)` applied to specific arguments)
- *rewrite* recursive functions as applications of `foldr`, `foldl`, `foldl'`, or `map`
- *use* functions such as `flip`, `curry`, and `uncurry` to adapt functional arguments of higher-order functions

Sample Question for Higher-Order Functions

> **Note**

> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following definitions:

```haskell
-- feed a value through two functions in sequence
seqApply :: a -> (a -> b) -> (b -> c) -> c
seqApply x f g = g (f x)

processList :: [Int] -> [Int]
processList [] = []
processList (x : xs) = if even x then (x + 1) : processList xs else
processList xs
```

Questions:

1. What is the value of `seqApply 3 (*2) show`?
2. Which of the following expressions typecheck: `seqApply 0 (+1) (*2)`, `seqApply "ab" length (==2)`, `seqApply [1,2,3] tail head`? For each that typechecks, give the type or the result.
3. Consider `iter :: (a -> a) -> Int -> (a -> a)` defined by `iter f 0 = id` and `iter f n = f . iter f (n-1)`. What is the value of `iter (*2) 3 1`?
4. Rewrite the `processList` as an application of `foldr` or `foldl'` (perform a "mechanical" rewriting, without taking performance into account), without explicit pattern matching on the argument.

References for Higher-Order Functions

1. Haskell 2010 Language Report. §3.3 Curried Applications and Lambda Abstractions
2. Learn You a Haskell. Higher order functions
3. Well-Typed. Higher-Order Functions

Lazy evaluation and non-strict semantics

Type declarations and type aliases

Type System Mastery

Algebraic data types (ADTs)

The candidate must understand how to define and work with algebraic data types. Specifically, the candidate should be able to

- *define* datatypes using `data` declarations to model domain concepts precisely
- *distinguish* between type constructor identifier (e.g. `Maybe`) and value constructor identifiers of an algebraic datatype (e.g. `Nothing` and `Just`)
- *recall* that type constructor identifiers and value constructor identifiers occupy distinct namespaces (e.g. `data V2 = V2 Double Double` introduces both type `V2` and value constructor `V2`)
- *write down* types of value constructors of a given algebraic datatype (e.g. `Left :: a -> Either a b`)
- *provide examples* of values for a given sum type (disjoint unions, e.g., `Either a b`, `Maybe a`, or custom types with multiple constructors)
- *provide examples* of values for a given product type (tuples, records, custom data types with one constructor)
- *define* sum types to represent alternatives or choices in domain modeling
- *reason* about the "algebra" of datatypes, including counting the number of possible values of a given type (e.g. `Maybe Bool`)
- *convert* between two isomorphic types (such as `(Year, Month, Day)` and `data Date = MkDate Year Month Day`, or `Either String Int` and a custom sum type):
  - implement a pair of functions, converting values of the two types to each other
  - argue (informally or semi-formally) that the pair of functions are inverses of each other
- *refactor* code from using generic sum types (like `Either`) to using domain-specific sum types

- *implement* (potentially recursive) functions to process or generate terms of algebraic datatypes

Sample Questions on Algebraic Data Types

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Question 1: How many (total) values are there of the following types?

```
Either Bool Bool
Bool -> Bool
data Status = Active | Inactive | Pending
data Error = ConnectionError | UnknownError
Either Error Status
(Error, Status)
```

Question 2: In a given system, a user can authenticate using a password or a biometric token. This choice is currently represented as a sum type using `Either`:

```
type AuthMethod = Either Password BiometricToken

authenticate :: AuthMethod -> User -> AuthResult
authenticate (Left  pw) u = checkPassword pw u
authenticate (Right bt) u = checkBiometric bt u
```

Perform a refactoring so that instead of `Either`, a domain-specific sum type is being used. Also add a third option for session-based authentication that does not need any additional data to process. Assume a function `checkSession :: User -> AuthResult` exists. Modify `AuthMethod` and `authenticate` as needed.

Question 3: A system currently represents product information using a combination of `Either` (a sum type) and tuples (product types):

```
type ProductInfo1 = (String, Either Double Int)
```

The tuple contains `(name, price)` where `price` is either a monetary amount (`Double`) or a quantity-based price (`Int`).

Refactor this to use a domain-specific data type that combines both sum and product types. Define a type `ProductInfo2` that:

- Uses a single constructor with named fields for the name and pricing method
- Uses a sum type with constructors for monetary and quantity-based pricing (instead of `Either`)
- Maintains the same information content

Then implement conversion functions:

```
productInfo1ToProductInfo2 :: ProductInfo1 -> ProductInfo2
productInfo2ToProductInfo1 :: ProductInfo2 -> ProductInfo1
```

References

1. Haskell 2010 Language Report. §4.2 User-Defined Datatypes
2. School of Haskell. Algebraic Data Types
3. Learn You a Haskell. Making Our Own Types and Typeclasses

Parametric polymorphism

The candidate must understand parametric polymorphism and be able to reason about polymorphic types. Specifically, the candidate should be able to

- *reason* about the number of different possible implementations of a massively polymorphic type such as `a -> a` or `(a, a) -> (a, a)`
- *apply* principles of parametricity to reason about what functions of certain types can or cannot do (e.g. can it be total? can a function with type `(a -> b) -> [a] -> [b]` mix input list elements into the output?)
- *infer* the most general type of expressions involving polymorphic functions (e.g. `\ x -> const x x`)

- *unify* type variables when combining polymorphic functions (e.g., in lists or conditionals, such as "what is the most general type of `\ x -> [] : x`?")
- *distinguish* between parametric polymorphism and type-class polymorphism
- *explain* how type class constraints affect the set of possible implementations (e.g. `a -> a -> Bool` vs `Eq a => a -> a -> Bool`)

Sample Questions on Parametric Polymorphism

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Question 1: How many different total functions are there of type `Maybe a -> Maybe a`? Give their implementations.

Question 2: If you have

```
f1 :: a -> b -> [a]
f2 :: String -> c
```

What is the (most general / inferred) type of `[f1, f2]`? If you think it is a type error, explain why.

Question 3: What is the type of `map . map`?

Question 4: What can you say about a function of type

```
mystery :: Maybe a -> a
```

without knowing its implementation?

Question 5: If you have two functions:

```
f1 ::          ... some type involving the type variable a ...
f2 :: Eq a => ... the same type involving the type variable a ...
```

Does every implementation of `f1` also typecheck at the type of `f2`? Does every implementation of `f2` also typecheck at the type of `f1`? Or neither?

References for Parametric Polymorphism

1. Well-Typed. Part 4: Parametric Polymorphism and Overloading
2. Wadler, Philip. Theorems for free! In Proceedings of the fourth international conference on Functional programming languages and computer architecture, 1989.

> ⚠ **Warning**
>
> In Wadler's "Theorems for free!" section \S 1 Introduction provides valuable intuition. The formal treatment in the remainder of the paper is not necessary to master the skills at the beginner level.

Type Classes

> ⓘ **Note**
>
> This section focuses on commonly used type classes. In particular, the following standard type classes are **not** required at the beginner level: `Ix`, `Real`, `RealFrac`, `Typeable`, `Data`, `Generic`, `Category`, and `Arrow`.

The candidate must understand Haskell's type class system and be able to work with type class constraints, instances, and standard type classes. Specifically, the candidate should be able to

- *infer* the most general type (including the constraints on type variables) of a simple function that uses common type class methods
- *identify* missing, redundant (implied by others), and unused constraints in type signatures of valid Haskell functions, explaining why constraints are needed or not needed

- *implement* simple functions according to a specification and a type signature that involves class constraints
- *implement* instances of standard type classes for given data types, understanding the required methods and expected properties (laws)
- *verify* informally whether a given type class instance upholds the laws (expected properties) of that type class
- *recognize* when one constraint implies another (e.g., `Ord a` implies `Eq a`, `Monad m` implies `Applicative m` and `Functor m`)
- *explain* the purpose and *list* key methods of standard type classes, including
    - `Eq`, `Ord`, `Num`, `Read`, `Show`, `Bounded`, `Enum`
    - `Functor`, `Foldable`, `Applicative`, `Monad`
    - `Semigroup`, `Monoid`
    - `Integral`, `Fractional`, `Floating`
    - `IsString`
- *explain* and *give examples* of functions with a given generic type signature (e.g., `sum :: (Foldable t, Num a) => t a -> a`, `length :: Foldable t => t a -> Int`), demonstrating understanding of standard library type signatures
- *recognize* common patterns where type classes are used, such as
    - using `Eq` and `Ord` for comparisons and sorting
    - using `Monoid` for combining values
    - using `Functor`, `Applicative`, and `Monad` for effectful computations
    - using `Foldable` and `Traversable` for container operations
- *explain* typeclass-related compilation errors and *suggest* appropriate fixes (e.g., `No instance for (Ord a)` when using `>` on a type that only has `Eq`, fix by adding `Ord` constraint or using `==` instead)
- *recognize* and *explain* design anti-patterns involving type classes, such as

- using existential types with type classes (e.g., “Why not have a list of `Show` able values?”, see Existentials and the heterogenous list fallacy by Chris Done)
- adding type class constraints to type parameters in data type declarations (e.g. see deprecated GHC extension for Data type contexts)

Sample Question on Type Classes

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell snippet and answer the questions below.

```
data AssocList k v = AssocList [(k, v)]

instance (Eq k, Eq v) => Eq (AssocList k v) where
  AssocList xs == AssocList ys =
    any (\k -> k `elem` map fst ys) (map fst xs)

filterKeys :: (k -> Bool) -> [AssocList k v] -> [AssocList k v]
filterKeys p = fmap (\(AssocList pairs) -> AssocList (filter (p .
fst) pairs))
```

Questions:

1. What is the most general type of `filterKeys` (if we omit the explicit type signature)? Include all necessary constraints.
2. Does the `Eq` instance for `AssocList` uphold the `Eq` laws? If not, explain which law is violated and provide a counterexample.
3. The most general type of `filterKeys` function includes a `Functor f` constraint. Explain what the `Functor` constraint allows this function to do, and give an example of a concrete type (other than lists) that could be used for `f`.

References

1. Learn You a Haskell. Types and Typeclasses
2. Real World Haskell. Using Typeclasses
3. Haskell Wikibooks. Classes and types
4. Learn You a Haskell. Making Our Own Types and Typeclasses

Kind System Basics

The candidate must understand the kind system and be able to reason about kinds of type constructors and type class parameters. Specifically, the candidate should be able to

- *determine* the kind of a type constructor defined via `data` declarations, including types that are parameterised by arguments of higher kinds
- *infer* the kind of a type class parameter from the method signatures in the class definition
- *identify* kind errors in type signatures and explain why they occur
- *suggest* fixes for ill-kinded type signatures
- *recognize* when a type constructor has the correct kind to be used as an instance of a type class (e.g., `Functor`)
- *distinguish* between concrete types (kind `Type` or `*`) and type constructors of various arities

The following are **not** expected:

- `DataKinds` extension
- `PolyKinds` extension (phantom type parameters are expected to be defaulted to `Type`)
- `ConstraintKinds` extension (the only concrete kind used is `Type`)

Sample Question

> **Note**

> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell declarations and answer the questions below.

```haskell
data Wrapper a = Wrapper a
data Box f a = Box (f a)
data Nested f g a = Nested (f (g a))

class Processor p where
  process :: p Int -> p String

class Mapper m where
  mapIt :: (a -> b) -> m a -> m b
```

Questions:

1. What are the kinds of `Wrapper`, `Box`, and `Nested`?
2. What is the kind of the type parameter `p` in the `Processor` type class?
3. What is the kind of the type parameter `m` in the `Mapper` type class?
4. Is it possible to define `instance Functor Box`? If yes, specify what to write instead of `...` in `instance Functor ... where`. If not, explain why.
5. Consider the type signature `h :: Box (Either String) -> Box (Maybe String)`. Is this type signature well-kinded? If not, explain the kind error and suggest a fix.

References

1. Serokell. Kinds and Higher-Kinded Types in Haskell
2. Learn You a Haskell. Making Our Own Types and Typeclasses

User-Defined Wrapper Types (`newtype` vs `data` vs `type`)

The candidate must know the difference between the keywords `newtype`, `data`, and `type`. Specifically, the candidate should be able to

- *recall* when a `data` declaration can be turned into a `newtype` declaration without compiler errors
- *specify* limitations of `type` synonyms, such as
    - no recursive `type` definitions
    - no partial application of `type` synonyms (by default, without `LiberalTypeSynonyms` or other extensions)
- *list* some standard uses for `newtype`, such as
    - restricting type class instances (e.g. for identifiers like `UserId`),
    - separating incompatible types (e.g. `Miles` vs `Kilometers`),
    - overloading type class instances (e.g. with wrappers from `Data.Monoid`),
    - annotating data with type-level information (e.g. `Data.Tagged`),
    - abstracting away common patterns (e.g. standard monads/effects from `mtl`), etc.
- *expand* manually some types involving `type` synonyms (e.g. `FilePath → String → [Char]`)
- *explain* the difference between operational semantics of `newtype` and `data`
- *refactor* the code from using `type` synonyms to using `newtype` instead, updating the type declaration and correctly identifying all necessary adjustments to the code that uses valus of the type synonym in a given snippet
- *identify* situations in the code when changing `newtype` to `data` affects totality of a function

- *convert* between two simple isomorphic types (such as `(Bool, Char)` and `data Letter = Letter { isUpper :: Bool, symbol :: Char }`):
    - implement a pair of functions, converting values of the two types to each other
    - arguing (informally or semi-formally) that the pair of functions are inverses of each other

Sample Question for ADTs

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell snippet and answer the questions below.

```haskell
data Name = Name [String]

name :: String -> Name
name rawName
  | validName = Name nameParts
  | otherwise = error "invalid name"
  where
    validName = (length rawName > 0) && (all isValidPart nameParts)
    isValidPart part = all isAlpha part
    nameParts = words rawName

anonymize :: Name -> String
anonymize (Name _) = "John Doe"

main :: IO ()
main = do
  input <- getLine
  putStrLn (anonymize (name input))
```

Questions:

1. What can identifier `Name` refer to in this program?

2. Will this program work correctly (without crash/run-time error) on any user input?
3. Is it possible to replace `data` with `newtype` without introducing compiler errors? If yes, does it affect the behaviour of the program on some user inputs?
4. Is it possible to replace `data` with `type` (possibly updating some uses of `Name` constructor) without introducing compiler errors? If yes, does it affect the behaviour of the program on some user inputs?
5. Argue, which of `type`, `newtype`, or `data` is most appropriate in this code snippet.

References for ADTs

1. Haskell 2010 Language Report. §4.2 User-Defined Datatypes
2. GHC User Guide. §6.6.5 Generalised derived instances for newtypes
3. GHC User Guide. §5.2 Warnings and sanity checking (`-Wincomplete-patterns`)

Generalized Algebraic Data Type Notation (GADTs) for Regular Algebraic Data Types

---

> ![NOTE] This section focuses on GADT-style syntax for regular algebraic data types. Advanced GADT features such as existential types, heterogeneous lists, and deriving for GADTs are not expected at this level.

The candidate must understand GADT-style syntax for data type declarations and be able to convert between regular ADT syntax and GADT-style syntax. Specifically, the candidate should be able to

- *convert* a regular algebraic data type declaration to GADT-style syntax by providing explicit type signatures for each constructor

- *convert* a GADT-style data type declaration to regular ADT syntax when the GADT-style syntax is merely syntactic sugar (i.e., when it does not use type refinement)
- *identify* when a GADT-style declaration is equivalent to a regular ADT declaration versus when it uses actual GADT features (type refinement, existentials)
- *recognize* the syntax differences between regular ADT declarations and GADT-style declarations, including the use of `where` keyword and explicit constructor type signatures

Sample Question for GADT Notation

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell snippet and answer the questions below.

```haskell
{-# LANGUAGE GADTs #-}

data Tree a where
  Leaf  :: a -> Tree a
  Node  :: [Tree a] -> Tree a

size :: Tree a -> Int
size (Leaf _)   = 1
size (Node ts)  = sum (map size ts)
```

Questions:

1. Rewrite the `Tree` data type declaration using regular ADT syntax (without GADT-style syntax).
2. Does rewriting `Tree` to use regular ADT syntax affect the implementation or type signature of the `size` function? Explain.
3. What are the syntactic differences between the GADT-style declaration and the regular ADT declaration for this data type?

References for GADTs

1. GHC User Guide. §9.4.7.1. Generalised Algebraic Data Types (GADTs)
2. GHC User Guide. §9.4.7.2. GADT Syntax

Existential types

Rank-N types

Scoped type variables

Functional dependencies

Functional Programming Concepts

Pure functions

Immutability

Function composition and pipelines

Currying and partial application

Point-free style

Folding and unfolding

Functors, Applicatives, Monads

Monoid, Semigroup

Category

Traversables and Foldables

Lenses and optics (e.g. lens library)

Standard Libraries & Prelude

Prelude essentials

Data.List, Data.Maybe, Data.Either

- non-empty lists as in `Data.List.NonEmpty` [controversial]

Data.Map, Data.Set, Data.Sequence

Data.Text, Data.ByteString

Control.Monad, Control.Applicative

System.IO, Text.Read

base library

IO & Effects

The `IO` Monad

The candidate must understand Haskell's `IO` type and be able to reason about effectful computations. Specifically, the candidate should be able to

- *distinguish* pure values from `IO` actions (types like `String` vs `IO String`)
- *infer* the types of simple expressions involving `IO` (including `getLine`, `putStrLn`, `pure`, `(>>=)`, `(>>)`)
- *identify* basic type errors such as when values of type `IO a` are used where plain `a` is expected
- *explain* how `>>=` sequences actions by passing the produced value to the next computation, and how `>>` sequences actions while discarding the first result
- *recognize* that `IO` actions are values and can appear inside other expressions (including data structures) without executing their effects immediately
- *predict* which side effects happen in small programs, including cases where a `IO` actions are just used as values, but not executed
- *use* (higher-order) functions for combining and sequencing `IO` side effects (including `mapM`, `mapM_`, `sequence`, `sequence_`)
- *implement* (higher-order) functions for combining and sequencing `IO` effects, given their type signature (i.e. `maybeIO :: Maybe (IO a) -> IO (Maybe a)`, `unfoldM :: (a -> Maybe (IO a)) -> a -> IO [a]`)

Sample Questions for The IO Monad

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell program and answer the questions below.

```haskell
main :: IO ()
main =
  print $
    snd
      ( getLine >>= \s -> putStrLn "SIDE" >> putStrLn s
      , length [getLine, getLine, pure "x"]
      )
```

Questions:

1. What is the inferred type of `length [getLine, getLine, pure "x"]`?
2. Does running this program print `SIDE`? Explain briefly.
3. How many times does the program ask the user for input (how many `getLine` calls actually occur)? Explain briefly.
4. What value is printed by `main` when it runs? (Answer with the concrete integer.)
5. Consider this alternative program:

```haskell
main2 :: IO ()
main2 =
  getLine >>= \s -> putStrLn "SIDE" >> putStrLn s >> lengthIO
[getLine, getLine, pure "x"] >>= print
  where
    lengthIO [] = pure 0
    lengthIO (program:programs) = program >> lengthIO programs >>=
\n -> pure (n + 1)
```

How many times does it ask the user for input, and what is printed? (You may assume the user provides any input values you like.)

References for The IO Monad

1. Haskell 2010 Language Report. §7 Basic Input/Output
2. Real World Haskell. Chapter 7: I/O
3. Learn You a Haskell. Input and Output

do notation

The candidate must understand `do` notation and be able to use it to write and type simple programs. Specifically, the candidate should be able to

- *recognize* the grammar of `do` blocks, including indentation/layout requirements
- *explain* the role of `<-` binding (`x <- action`) versus sequencing without binding (e.g. `do { action1; action2 }`)
- *explain* the syntax and meaning of `let` bindings inside a `do` block, including that `let` does not execute effects
- *infer* the types of variables bound using `<-` or `let` (e.g. from `x <- getLine` , infer `x :: String` ; from `let x = getLine` infer `x :: IO String`)
- *infer* the overall type of a `do` block from the types of its statements
- *translate* small examples between `do` notation and explicit uses of `(>>=)` / `(>>)`
- *explain* that `return` is not a control flow operator and cannot exit the `do` block abruptly
- *recall* that `return` and `pure` are equivalent for `IO` (or, generally, any `Monad m`)

Sample Questions for do notation

> **Note**
>
> This sample combines some of the skills above, together with some knowledge of other topics.

Consider the following Haskell snippet and answer the questions below.

```haskell
program = do
  putStrLn "Enter a word:"
  w <- getLine
  let n = length w
  m <- return (length w, "Hello")
  putStrLn ("Length is " ++ show n)
  return (n, w)
```

Questions:

1. What is the inferred type of `w`?
2. What is the inferred type of `n`?
3. What is the inferred type of `m`?
4. Will running `program` ever print `Length is ...`?
5. What is the inferred type of `program`?
6. Rewrite `program` using only `(>>=)`, `(>>)` and `pure` (no `do` notation). Use `\` lambdas or (regular) `let` for bindings.
7. If we replace the last line `return (n, w)` with `putStrLn (show n)`, does the code type-check? If it does not, explain the type mismatch in terms of the required final `IO a` result.

References for do notation

1. Haskell 2010 Language Report. §3.14 Do Expressions
2. Learn You a Haskell. A Fistful of Monads

Reading from / writing to files

Handling command line arguments

Working with environment variables

Logging and debugging tools

Exceptions and error handling (Either, Maybe, Exception, MonadError)

Concurrency (forkIO, STM, MVar, TVar, Async)

State, Reader, Writer

Transformers

ST

About Intermediate Level

Tooling & Ecosystem

GHC (Glasgow Haskell Compiler)

Cabal (build tools)

GHCi (REPL)

Hoogle (documentation search)

Haddock (documentation generator)

HLint (linter)

Profiling and benchmarking (criterion, weigh)

Testing: QuickCheck, Hspec, Tasty

Dependency Management & Project Structure

Project scaffolding

Hackage and Stackage

Pinning dependencies

Package version bounds and PVP

Real-World Development

Core Patterns

- MTL (Monad Transformer Library)
- ReaderT pattern (ReaderT r IO)

Web Development

- WAI and Warp (web server stack)
- Aeson (JSON parsing/encoding)

Databases

- postgresql-simple

Advanced Concepts

Category theory basics

Fixpoints and recursion schemes

Final tagless style

Type-safe domain modeling

Data encoding/decoding strategies

Testing & Quality

Property-based testing (QuickCheck)

Unit testing (Hspec, Tasty)

Integration tests

Mocking and dependency injection using monads

Test coverage tools

Deployment & DevOps

Building executables and libraries

Deploying Haskell applications

CI/CD integration (GitHub Actions, GitLab, etc.)

Performance tuning

Memory profiling

Soft Skills & Extras

Reading GHC error messages

Navigating and contributing to open-source Haskell projects

Reading academic papers on functional programming

Participating in the Haskell community (Discourse, Reddit, Matrix, GitHub)

Writing maintainable, idiomatic Haskell

Contributors

Here is a list of the contributors who have helped in some way to form this syllabus (listed alphabetically):

- Alexandre Garcia de Oliveira @romefeller
- Andres Löh @kosmikus
- Brian McKenna @puffnfresh
- Doug Beardsley @mightybyte
- Jose Calderon @jmct
- Nikolai Kudasov @fizruk
- Tobi Oloke @tobz619
- Vaibhav Sagar @vaibhavsagar

If you feel you're missing from this list, feel free to add yourself in a PR.