COMP3258: Functional Programming

Assignment 2

Deadline: 23:59, Mar 31, 2025 (HKT)

---

1. You are allowed to import other modules. However for 3rd-party packages (i.e. `cabal install` packages), please ask the TA for permission first.
2. For absurd cases (don't be confused with base case), feel free to raise exception (e.g. using `undefined`) or give a default value for it.
3. Please submit a single zip file `A2_XXX.zip`, with `XXX` replaced by your UID, which includes the 4 Haskell files `Prop.hs`, `RList.hs`, `JSON.hs`, `DFA.hs`.
4. Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment, and explain it in a detailed manner. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.
5. The style and quality of the code will also be assessed. Check out the Haskell style guide for the recommended code style.
6. Feel free to define any auxiliary functions or data types if needed.

Part 1: Propositional Logic (25 pts)

In this exercise, we're going to implement parts of propositional logic, where propositions are joined with logical connectives. In our setting, the syntax is defined as follows:

```haskell
type Atom = String

data Prop
  = Var Atom
  | F
  | T
  | Not Prop
  | Prop :|: Prop
  | Prop :&: Prop
  | Prop :->: Prop
  | Prop :<->: Prop
  deriving (Show, Eq)
```

In this datatype definition:

- The propositions are represented as `Prop` type.
- We use `Var` to create propositional variables (e.g., `Var "P"`)
- Use `F` to build a logic false value (don't be confused with Haskell `False` value)
- Use `T` to build a logic true value;
- Use `Not` to build a negation (e.g., `Not (Var "P")`)
- Use `:|:` and `:&:` to construct disjunction and conjunction
- `:->:` is implication and `:<->:` is bi-implication ("if and only if")

To avoid confusion, the truth table of the last two constructs are provided below:

P	Q	$P \to Q$	$P \leftrightarrow Q$
T	T	T	T
T	F	F	F
F	T	T	F
F	F	T	T

Problem 1

Define an `eval` function, given an environment (contains a list of names associated with their truth values) and a proposition, and return a boolean to indicate its truth.

```haskell
type Env = [(Atom, Bool)]

eval :: Env -> Prop -> Bool
eval = undefined
```

Problem 2

We say that proposition A is:

- **Satisfiable**, if there is *some* truth assignment for the variables of A that satisfies A (make A be true)
- **Unsatisfiable**, if *no* truth assignment satisfies A
- **Valid** (or tautology), if *every* truth assignment satisfies A

Define three functions to detect whether a proposition is satisfiable, unsatisfiable or valid.

For the `satisfiable` function, if a proposition is satisfiable, you should also return **a *possible* truth assignment** (in any order), otherwise return `Nothing`.

```haskell
satisfiable :: Prop -> Maybe [(Atom, Bool)]
satisfiable = undefined

unsatisfiable :: Prop -> Bool
unsatisfiable = undefined

valid :: Prop -> Bool
valid = undefined
```

Expected Results

```haskell
satisfiable ((Var "P") :&: (Not (Var "Q"))) == Just [("Q",False),("P",True)]
satisfiable ((Var "P") :&: (Not (Var "P"))) == Nothing
unsatisfiable ((Var "P") :&: (Not (Var "P"))) == True
valid ((Var "P") :|: (Not (Var "P"))) == True
```

Problem 3

Conjunctive normal form (CNF) is a canonical normal form of a logical formula consisting of a **conjunction of clauses**, where a clause is a disjunction of literals (variables (i.e. `Var a`) and negated variables (i.e. `Not (Var a)`)).

In Haskell, we use **a list of lists** to represent CNF. For the following propositions, we have:

Propositions (in CNF)	*Haskell Representation (in clauses)*
$True \land P$	`[ [ Var "P" ] ]`[1]
$False \land False$	`[ [], [] ]`
$A \lor \neg B$	`[ [ Var "A", Not (Var "B") ] ]`
$(A \lor B) \land C \land D$	`[ [ Var "A", Var "B" ]` `, [ Var "C" ], [ Var "D" ] ]`
$(A \lor \neg B \lor \neg C) \land (\neg D \lor E)$	`[ [ Var "A", Not (Var "B"), Not (Var "C") ]` `, [ Not (Var "D"), Var "E" ] ]`

Your task is to implement a function `toCNF` that takes a `Prop` and return a `CNF` which is in CNF.

```haskell
type Clause = [Prop]
type CNF = [Clause]

toCNF :: Prop -> CNF
toCNF = undefined
```

You can use this link to check the examples of CNF.

Notes

1. CNF of a given proposition is **not** unique, your answer is accepted as long as it is in **CNF** and **logically equivalent** to the original proposition.
2. The value `T` (true) should be represented as `[]` in CNF (**no clauses**).
   - The reason is that for two CNF **formulas** `p, q :: CNF`, their conjunction $P \land Q$ is simply their concatenation `p ++ q`. Since `T` represents a formula that is always true, it contributes no clauses to the overall formula.
3. The value `F` (false) should be represented as `[[]]` in CNF (an **empty clause**).
   - This follows from the fact that, in CNF, the disjunction $P \lor Q$ of two **clauses** `p, q :: Clause` is also their concatenation: `p ++ q`. An empty clause within a CNF formula means no disjunction and is always unsatisfiable (false).

Part 2: Random Access List (25 pts)

Lists are used ubiquitously in our course. However, accessing any element in a list, even if you know its numeric index i, will always take $O(i)$ time. Here, we are going to design a data structure called *Random Access List* to improve the indexing time to $O(\log(i))$.

```haskell
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)
            deriving (Show, Eq)
```

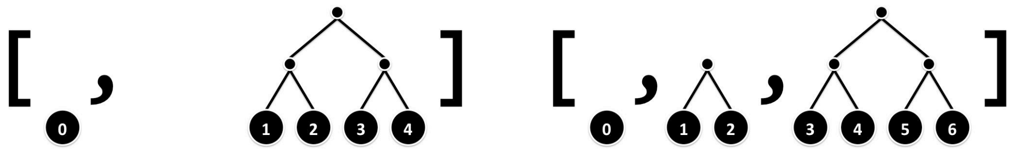
The data type `Tree a` represents a binary tree:
- A `Leaf` is a leaf node of the tree, with a payload data of the generic type `a`.
- A `Node` is an intermediate node, consisting of its size (the total number of leaves in all its sub-trees), and its two child trees.[2]

```haskell
newtype RList a = RList [Tree a] deriving (Show, Eq)
```

[1]Updated (March 17): fixed typo
[2]Updated (March 24): clarified

Internally, a random access list is represented as a list of **complete binary trees** whose sizes are in **strictly ascending order**, i.e. if a tree has n elements, the next tree in the list must have more than n elements (not the same size).

Two examples are:



Problem 4

To start, implement the helper function `size` that returns the size of a (well-formed) binary tree (total number of leaves). Then implement the function `len` that computes the length of the random access list (sum of the sizes of its trees).

```
size :: Tree a -> Int
size = undefined

len :: RList a -> Int
len = undefined
```

Expected Results

```
size (Node 2 (Leaf 1) (Leaf 2)) == 2
len (RList [Leaf 0, Node 2 (Leaf 1) (Leaf 2)]) == 3
```

Problem 5

The next step is to implement the `cons` operation (similar to `(:)` for lists), which adds a new element to the **head** of the random access list. We start by defining a helper function `insertTree` that inserts a new tree to an existing list of trees while maintaining the invariant (sizes of the trees in strictly ascending order).

After putting a new tree `t` at the front of the list `xs`:

- If two adjacent trees have the same sizes, we should merge them into a larger tree.
- If a larger tree is placed before a smaller tree, the invariant is violated. Raise an error in this case.

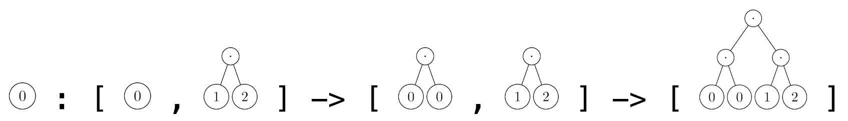


Figure 1: Example of `insertTree`

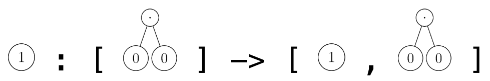


Figure 2: Example of `insertTree`

```
insertTree :: Tree a -> [Tree a] -> [Tree a]
insertTree = undefined
```

The `cons` operation for a random access list is simply inserting a new leaf to the list.

```
cons :: a -> RList a -> RList a
cons x (RList xs) = RList $ insertTree (Leaf x) xs
```

Now you can convert a normal list to a random access list using the following function:

```
fromList :: [a] -> RList a
fromList = foldr cons nil
```

Expected Results

```
insertTree (Leaf 0) [Leaf 1, Node 2 (Leaf 2) (Leaf 3)] ==
  [Node 4 (Node 2 (Leaf 0) (Leaf 1)) (Node 2 (Leaf 2) (Leaf 3))]
cons 0 (RList [Leaf 1]) == RList [(Node 2 (Leaf 0) (Leaf 1))]
```

Problem 6

Before implementing the uncons function, you will need a helper function that splitTree that splits a tree. For the returned tuple, the first element will be the **left-most** leaf of the input tree with the second element being a list of the remaining sub-trees (their sizes should also be in ascending order).

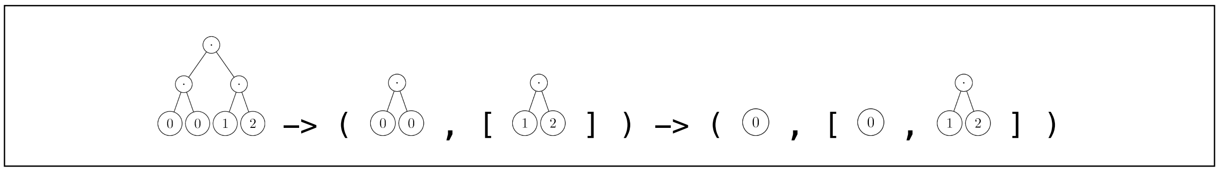


Figure 3: Example of splitTree

```
splitTree :: Tree a -> (a, [Tree a])
splitTree = undefined
```

Implement the uncons function using splitTree defined previously. This function would work as a combination of the head and tail function. That is, the first component of the returned tuple should be the head of the list with the second component being the tail of the list.

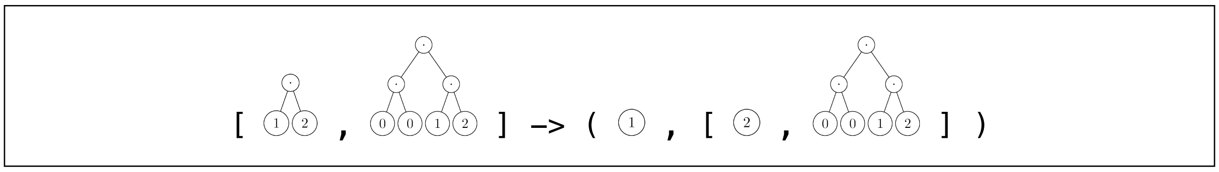


Figure 4: Example of uncons

```
uncons :: RList a -> (a, RList a)
uncons = undefined
```

Note and Hints

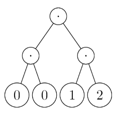
1. The splitTree function should "invert" the effect of the insertTree.
2. When implementing uncons, you should raise an exception if the input is an empty list.

Expected Results

```
splitTree (Node 4 (Node 2 (Leaf 0) (Leaf 1)) (Node 2 (Leaf 2) (Leaf 3))) ==
  (0,[Leaf 1,Node 2 (Leaf 2) (Leaf 3)])
uncons (RList [Node 2 (Leaf 1) (Leaf 2)]) == (1, RList [Leaf 2])
```

Problem 7

Before implementing the actual indexing function for the random access list, it would be a good idea to implement a lookup function for the binary tree. The function ixTree takes a tree and an index i, and returns the i-th leaf of the tree (counting from the left).

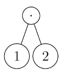
ixTree 

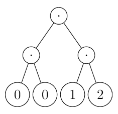
2 = 1

Figure 5: Example of `ixTree`

```
ixTree :: Tree a -> Int -> a
ixTree = undefined
```

Now implement the `ix` function using `ixTree`. The function should takes a random access list and an index, and returns the indexed element.

ix [

, 

] 4 = 1

Figure 6: Example of `ix`

```
ix :: RList a -> Int -> a
ix = undefined
```

Note

1. Raise an exception if the given index is too large.

Expected Results

```
t = (Node 4 (Node 2 (Leaf "a") (Leaf "b")) (Node 2 (Leaf "c") (Leaf "d")))
ixTree t 0 == "a"
ixTree t 2 == "c"
ix (fromList [0..4]) 0 == 0
ix (fromList [0..4]) 2 == 2
```

Part 3: JSON (25 pts)

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for configuration files and transmitting data in web applications. In this exercise we would consider a *simplified* version of JSON, represented by the following `JSON` data type:

```
data JSON
  = JNum Integer
  | JStr String
  | JArr [JSON]
  | JObj [(String, JSON)]
  deriving (Show, Eq)
```

In the `JSON` data type:
- `JNum` represents a numeric value[3]
- `JStr` represents a string value
- `JArr` represents an array of JSON values
- `JObj` represents an object, modeled as a list of key-value pairs[4]

[3]While real-world JSON also allows floating point numbers, we only consider `Integer` here.
[4]While the Haskell representation preserves the order of pairs in the list, JSON objects should be **treated as unordered**.

Here is an example of JSON and its Haskell representation:

JSON	*Haskell Representation*
`{ "language": "Haskell", "year": 1990, "extension names": [ ".hs", ".lhs" ] }`	`JObj [ ("language", JStr "Haskell"), ("year", JNum 1990), ("extension names", JArr [ JStr ".hs", JStr ".lhs" ]) ]`

Problem 8

The JSON RFC states that *The names within an object SHOULD be unique.* However, our current JSON representation allows for duplicated keys. Implement a function `verifyJSON` that returns `True` if a JSON is valid (i.e. **all objects in the JSON structure have unique keys**), and `False` otherwise.

```haskell
verifyJSON :: JSON -> Bool
verifyJSON json = undefined
```

Notes

1. Note that the same key may appear in *different* objects, but within each object, all keys must be unique.

Expected Results

```haskell
verifyJSON (JObj []) == True
verifyJSON (JArr [JObj [("name", JStr "TypeScript")]]) == True
verifyJSON (JObj [("name", JStr "Java"), ("name", JStr "C#")]) == False
```

Problem 9

To retrieve data from a JSON structure, we need a way to **index** into it. Only the following two types of JSON value can be indexed into:

- JSON object: can be indexed with a string key (`StrIx`) to look up the corresponding value.
- JSON array: can be indexed with an integer index (`NumIx`) to access the element at that corresponding position.

Your task is to implement the function `index` that takes a (valid) JSON value and a list of indexes, then recursively applies each index to retrieve the nested value. If any lookup fails (e.g., the key is missing or the index is out of bounds), the function should return `Nothing`.

```haskell
data Index = NumIx Int | StrIx String deriving (Show, Eq)

index :: JSON -> [Index] -> Maybe JSON
index json indexes = undefined
```

Expected Results

```haskell
json = JObj [
  ("languages", JArr [
    JObj [("name", JStr "JavaScript")],
    JObj [("name", JStr "Python")]
    ])
  ]
```

```
index json [StrIx "languages", NumIx 1, StrIx "name"] == Just (JStr "Python")
index json [StrIx "languages", NumIx 2] == Nothing
index json [StrIx "name"] == Nothing
index json [NumIx 0] == Nothing
```

Problem 10

JSON Schema, defined in JSON itself, provides a way to describe the structure, data types, and constraints of JSON data.

In this part, we will work on a *simplified* version of JSON Schemas[5]. The types of JSON Schemas and their semantics are defined below:

JSON Schema	*Haskell Representation*	*Explanation*
`{ "type": "XXX" }`	`JObj [ ("type", JStr "XXX") ]`	• Matches any JSON value with the corresponding type. • `XXX` can be `number`, `string`, `array` or `object`
`{ "type": "array", "items": /* schema */, }`	`JObj [ ("type", JStr "array"), ("items", {- schema -}) ]`	• Matches the array type of JSON value. • **All items** in the array should match the sub-schema described by the `items` field.
`{ "type": "object", "properties": { "key1":/* schema1 */, "key2":/* schema2 */ // other properties... } }`	`JObj [ ("type", JStr "object"), ("properties", JObj [ ("key1",{- schema1 -}), ("key2",{- schema2 -}) -- other properties... ]) ]`	• Matches the object type of JSON value. • The object **should have the *exact* keys** described in `"properties"`, and each field should match the corresponding sub-schema.

Note that:

1. Every JSON Schema is a JSON object.
2. A JSON Schema can contain sub-schemas (as shown in the array and object case).
3. The key-value pairs **should be treated as unordered**, and there shouldn't be any duplicated keys.

And here are some **valid** schemas:

#	*JSON Schema*	*Haskell Representation*
1	`{ "type": "object", "properties": {}, }`	`JObj [ ("type", JStr "object"), ("properties", JObj []) ]`

[5]The semantics of JSON Schema described here is *different* from the real-world ones. But for this assignment, please follow our definitions.

#	*JSON Schema*	*Haskell Representation*

2	`{`
	`  "items": { "type": "number" },`
	`  "type": "array"`
	`}`
		`JObj [`
		`  ("items",`
		`    JObj [("type", JStr "number")]),`
		`  ("type", JStr "array"),`
		`]`

3	`{`
	`  "type": "object",`
	`  "properties": {`
	`    "language": { "type": "string" },`
	`    "year" : { "type": "number" },`
	`    "extension names": {`
	`      "type": "array",`
	`      "items": { "type": "string" }`
	`    }`
	`  }`
	`}`
		`JObj [`
		`  ("type", JStr "object"),`
		`  ("properties", JObj [`
		`    ("language",`
		`      JObj [("type", JStr "string")]),`
		`    ("year",`
		`      JObj [("type", JStr "number")]),`
		`    ("extension names", JObj [`
		`      ("type", JStr "array"),`
		`      ("items",`
		`        JObj [("type", JStr "string")])`
		`    ])`
		`  ])`
		`]`

Implement the function `verifySchema` that takes a JSON Schema as an input. Return `True` if it is a **valid schema** (having the same format as described above), otherwise return `False`.

```
verifySchema :: JSON -> Bool
verifySchema schema = undefined
```

> ***Hint:*** *For Problem 10 and 11, you may find it easier to tackle if you can figure out a suitable data type (e.g.* `data Schema`*) to represent JSON Schemas.*

Problem 11

We say a JSON value is compliant to a schema, when it matches the descriptions in the schema. Consider validating the following JSON values using schema #3 from Problem 10:

JSON	*Haskell Representation*	*Result*
`[1, 2]`	`JArr[JNum 1, JNum 2]`	❌ `array` is not `object`

`{`
`  "language": "OCaml",`
`  "year": "1996"`
`}`
	`JObj [`
	`  ("language", JStr "OCaml"),`
	`  ("year", JStr "1996")`
	`]`
		❌ missing field `"extension names"`
		
		❌ type of field `"year"` should be `number` instead of `string`

`{`
`  "language": "C++",`
`  "year": 1985,`
`  "extension names": {`
`    "common": ".cpp"`
`  },`
`  "family": "C"`
`}`
	`JObj [`
	`  ("language", JStr "C++"),`
	`  ("year", JNum 1985),`
	`  ("extension names", JObj [`
	`    ("common", JStr ".cpp")`
	`  ]),`
	`  ("family", JStr "C")`
	`]`
		❌ type of `"extension names"` is not an `array`
		
		❌ unexpected redundant field `"family"`

JSON	*Haskell Representation*	*Result*
`{ "extension names": [ ".scala", ".sc" ], "year": 2004, "language": "Scala" }`	`JObj [ ("extension names", JArr [ JStr ".scala", JStr ".sc" ]), ("year", JNum 2004), ("language", JStr "Scala") ]`	✅ all properties described in schema matched

Your task is to implement a function `validate` that takes a JSON Schema and a JSON value as inputs. Return `Just True` if the json is compliant to the schema, otherwise return `Just False`. If either input is invalid, return `Nothing`.

```
validate :: JSON -> JSON -> Maybe Bool
validate schema json = undefined
```

Part 4: Finite State Automaton (25 pts)

This section is designed to familiarize you with Haskell's `Map` and `Set` containers. You can refer to Appendix A and B for their usages. We start by importing the following modules:

```
import Data.Map (Map)
import Data.Map qualified as M
import Data.Set (Set)
import Data.Set qualified as S
```

Formally, a deterministic finite automata (DFA) is defined as a 5-tuple of $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is **a set of finite states**.
- Σ is a set of characters representing the **input alphabet**.
- δ is the **transition function** $\delta : (Q \times \Sigma) \to Q$ that maps a **current state** and an **input character** to the **next state**.
- q_0 is an **initial state**, where $q_0 \in Q$.
- F is a set of **accepting states**, where $F \subset Q$.

In Haskell, we use the following data type to model a DFA:

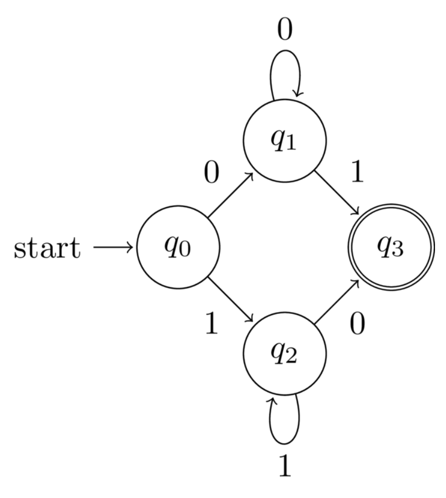
```
type State = String

data DFA = DFA {
    states :: Set State,                       -- set of states
    alphabet :: Set Char,                      -- set of input alphabet
    transition :: Map (State, Char) State,     -- transition function
    initialState :: State,                     -- initial state
    acceptingState :: Set State                -- set of accepting states
  } deriving (Show, Eq)
```

Notice that we use a `Map` from `(State, Char)` to `State` to model the transition function, instead of a normal Haskell function.

Here is a DFA represented both in diagram[6] and in Haskell:

[6]**Arrows** represent transitions (with the input character written on top), and the **start arrow** points at the initial state. The **circles** represent states, where a **double circle** indicates an accepting state.

DFA



Haskell Representation

```haskell
DFA
  { states = S.fromList ["q0", "q1", "q2", "q3"],
    alphabet = S.fromList "01",
    transition =
      M.fromList
        [ (("q0", '0'), "q1"),
          (("q0", '1'), "q2"),
          (("q1", '0'), "q1"),
          (("q1", '1'), "q3"),
          (("q2", '1'), "q2"),
          (("q2", '0'), "q3")
        ],
    initialState = "q0",
    acceptingState = S.fromList ["q3"]
  }
```

Problem 12

Define the function `dfa` (also called a "smart constructor") that takes:
- `q`: a list of states
- `sigma`: a list of `Char` as the input alphabet
- `delta`: a list of triples as transitions. For each triple `(s, a, s')`:
  - All states (i.e. `s`, `s'`) must be elements of `q`.
  - All characters (i.e. `a`) must be elements of `sigma`.
  - The input list (i.e. `delta`) should **not** contain multiple triples with the same `(s, a)` pair mapping to different `s'`.[7]
- `q0`: an initial state, which is an element of `q`
- `f`: a list of final states, which is a subset of `q`

Return the constructed `DFA` if the input satisfies the constraints above, otherwise return `Nothing`.

```haskell
dfa :: [State] -> [Char] -> [(State, Char, State)] -> State -> [State] -> Maybe DFA
dfa q sigma delta q0 f = undefined
```

Problem 13

With a DFA, we perform **transitions** based on a *current state* and an *input character*. At each step, the DFA reads one character from the input and moves to the next state according to its transition function.

Consider a DFA (the one shown at the beginning of this part) with an initial state q_0, processing the input 001. The DFA follows these steps:

- Starting at q_0, read the input 0, and move to q_1.
- At q_1, read the input 0, and move to (stay in) q_1.
- At q_1, read the input 1, and move to q_3.
- At q_3, as we have consumed all the inputs, the transition process is complete.

Since the transition process ends at an *accepting state*, the input string is **accepted**.

Define a function `step` that takes a (well-constructed) `DFA`, a current state, and an input character. It should perform a **single step** of transition base on its inputs, and return the next state. If such transition cannot be made, return `Nothing`.

[7]Otherwise, we don't know which `s'` to choose for that `(s, a)` when constructing the `Map`.

```haskell
step :: DFA -> State -> Char -> Maybe State
step = undefined
```

Implement the function `accept` that determines whether a (well-constructed) DFA accepts a given **input string** by simulating the DFA's behavior from its **initial state**:

- Start from the initial state.
- Process the input string character by character, using `step` to determine the next state.
- After consuming the whole input string, return `True` if the final state is an **accepting state**.
- If a transition fails or it ends at a non-accepting state, return `False`.

```haskell
accept :: DFA -> String -> Bool
accept = undefined
```

Expected Results

```haskell
-- we define example to be the DFA presented at the beginning of this part
step example "q0" '1' == Just "q2"
step example "q3" '1' == Nothing
accept example "00001" == True
accept example "0" == False
accept example "00010" == False
```

Problem 14

Given a well-constructed DFA, can you find an input string that leads us to the accepting state? Try to implement a function `search` that takes a well-constructed `DFA` as input, and return **a *possible* input string** that leads to an accepting state. Return `Nothing` if such input doesn't exist.

```haskell
search :: DFA -> Maybe String
search = undefined
```

Hints:

1. *If you are at state q_i, can you find the next state to move to?*
2. *Be aware of cycles. Consider the DFA given at the beginning of this part: if we start at q_1, consuming the letter 0 brings us back to q_1! If you're not careful, you might end up looping infinitely...*

Possible Result

Your answer is accepted as long as the path you returned leads to an accepting state (if it exists). Here is one possible search result:

```haskell
search example == Just "01"
```

Appendix A: Overview of Map

You can also refer to the Haskell's Map documentation on Hackage.

```haskell
-- imports
import Data.Map (Map)
import Data.Map qualified as M

-- empty map
M.empty :: Map k a

-- create a map from associated lists
M.fromList :: Ord k => [(k, a)] -> Map k a
-- >>> M.fromList [(5,"a"), (3,"b"), (5, "c")]
-- fromList [(3,"b"),(5,"c")]

-- insert a (new) key value pair
M.insert :: Ord k => k -> a -> Map k a -> Map k a
-- >>> M.insert "a" 1 M.empty
-- fromList [("a",1)]
-- >>> M.insert "a" 2 (M.fromList [("a", 1)])
-- fromList [("a",2)]

-- delete a key
M.delete :: Ord k => k -> Map k a -> Map k a
-- >>> M.delete 5 (M.fromList [(5,"a"), (3,"b")])
-- fromList [(3,"b")]

-- lookup a key
M.lookup :: Ord k => k -> Map k a -> Maybe a
-- >>> M.lookup 5 (M.fromList [(5,"a"), (3,"b")])
-- Just "a"
-- >>> M.lookup 7 (M.fromList [(5,"a"), (3,"b")])
-- Nothing

-- check if a key exists
M.member :: Ord k => k -> Map k a -> Bool
-- >>> member 5 (fromList [(5,'a'), (3,'b')])
-- True
-- >>> member 1 (fromList [(5,'a'), (3,'b')])
-- False

-- return all keys
M.keys :: Map k a -> [k]
-- >>> keys (fromList [(5,"a"), (3,"b")])
-- [3,5]

-- convert to associated list
M.toList :: Map k a -> [(k, a)]
-- >>> toList (fromList [(5,"a"), (3,"b")])
-- [(3,"b"), (5,"a")]
```

Appendix B: Overview of Set

You can also refer to the Haskell's Set documentation on Hackage.

```haskell
-- imports
import Data.Set (Set)
import Data.Set qualified as S

-- similar to the Map interface
S.empty :: Set a
S.fromList :: Ord a => [a] -> Set a
S.insert :: Ord a => a -> Set a -> Set a
S.delete :: Ord a => a -> Set a -> Set a
S.toList :: Set a -> [a]

-- create a singleton set
S.singleton :: a -> Set a
-- >>> S.singleton 1
-- fromList [1]
-- check if the element is in the set
S.member :: Ord a => a -> Set a -> Bool
-- >>> S.member 0 (S.fromList [1,2])
-- False

-- set union
S.union :: Ord a => Set a -> Set a -> Set a
-- >>> S.fromList [1,2] `S.union` S.fromList [2,3]
-- fromList [1,2,3]

-- set intersection
S.intersection :: Ord a => Set a -> Set a -> Set a
-- >>> S.fromList [1,2] `S.intersection` S.fromList [2,3]
-- fromList [2]

-- subset
S.isSubsetOf :: Ord a => Set a -> Set a -> Bool
-- >>> S.fromList [1,2] `S.isSubsetOf` S.fromList [1,2,3]
-- True
```