Aufgabe 4: Nandu

Team-ID: 00776

Team: BWINF0

Bearbeiter/-innen dieser Aufgabe:
Tobias Maurer

17. November 2023

Inhaltsverzeichnis

Ausführen von Haskell

Man kann Haskell mit Hilfe von ghci interpretiert ausführen lassen, aber wie man auch an der Ordnerstruktur entnehmen kann, gibt es ein kompiliertes Programm, was man anstelle dessen verwenden kann.

Lösungsidee

Die Aufgabe gibt keinen Zustand der Lichter an, daher wird für jede mögliche Kombination der Lichtzustände (es wird von entweder “An” oder “Aus” ausgegangen). Lichter kann man als Wahrheitswerte repräsentieren: True für “An” und False für “Aus”. Wenn man nun eine Liste entsprechend der Länge, wie beschrieben in der Datei, erstellt und diese auffüllt mit den Wahrheitswerten hat man die Repräsentation vom Start. Lichter können auch True annehmen, ansonsten wird einfach mit False aufgefüllt. Die nächste Ebene erstellt man, durch das Überlegen, wo die Bausteine nun kommen. Dann schaut man in der Startebene, was die Eingaben an den Baustein sind. Entsprechend der Ausgaben ergänzt man seine nächste Ebene – die Ausgabeliste an Wahrheitswerten für das Licht. Indem man diesen Prozess, dass man sozusagen die Bausteinbeschreibung auf die jetzige Ausgabe anwendet, um eine neue Ausgabe an Lichtwerten zu erhalten. Indem man diese neue Ausgabe nun wieder als Eingabe verwendet und es auf die „Bausteinbeschreibung“ anwendet, werden die Lichter ebenenweise berechnet. Am Ende muss man nur noch an den Positionen der Lichtern schauen, ob dort ein True (Ausgabe „An“) oder False (Ausgabe „Aus“) Wert in der Ebene mit Wahrheitswerten ist.

Umsetzung

Wie bereits bei der Lösungsidee erwähnt, werden alle möglichen Kombinationen der gegeben Lichtquellen berechnet sowie deren Ausgaben bei den Lampen. Es gibt $2^{(Anzahl\ Lichtquellen)}$ Möglichkeiten, in welchen Zuständen die Lampen stehen können. Man kann durch hoch zählen in Einerschritten von 0 bis zur Anzahl der Möglichkeiten - 1 in der Bitdarstellung dieser Zahlen die Zustände der Lichtquellen darstellen. Das erste Bit repräsentiert die erste Lichtquelle, das Zweite die zweite Lichtquelle usw.

Als nächstes müssen die Lampen – unsere Ausgaben – berechnet werden. Lichter können als Wahrheitswerte dargestellt werden, da sie exakt zwei Zustände haben können. Ebenen entsprechen einem Zustand der Lichtausstrahlungen, welche während der Berechnung erforderlich sind. Als erstes muss die Startebene kreiert werden. In meiner Implementation werden die Informationen der ersten Zeile ignoriert, da diese in Haskell nicht notwendig sind, um Speicher zu allokieren oder Ähnliches. Dies kann alles mit Listenlängen und deren Inhalt gemacht werden. Für jede Berechnung der Ebene wird die entsprechende Zeile in eine Liste an Zeichen zerlegt: „X“, „B“, „R“, „r“, „W“, „Q“ (bei „Q1“, „Q2“ …) und „L“ (bei „L1“, „L2“, …). Die erste Ebene ist ein wenig besonders. Für jedes „X“ wird der Lichtausgang in dieser Spalte auf „Aus“ (False) gesetzt. Bei Lichtquellen, markiert durch „Q“, hängt das vom Zustand, angegeben durch das entsprechende Bit der Zahl ab. Wenn das Bit der entsprechenden Stelle auf 1 steht, wird die Lichtquelle auf „An“ (True) gesetzt, ansonsten „Aus“ (False). Danach wird die nächste Zeile der Datei in die entsprechende Liste an Zeichen übersetzt. Jedes „X“ wird weiterhin einfach durch einen ausgeschalteten Lichtausgang ersetzt (False). Wenn allerdings ein Baustein vorliegt, also „B“, „W“, „R“ oder „r“, wird (bei meiner Implementierung) ein Tuple erstellt, mit dem Zeichen aus der Liste, welches davor kam. Sofern dies einen validen Baustein ergibt, wird der Lichtausgang von einer Spalte davor und der aktuellen Spalte genommen als Eingabe in eine Funktion, welche die Logik des entsprechenden Bausteins abbildet. Die beiden Rückgaben (linke Seite, rechte Seite) entsprechen den entsprechenden Positionen und werden dort in die Ausgangsebene eingetragen. Sofern kein valider Baustein herauskommt, wird nichts vom Programm unternommen, da bei einer validen Datei (davon wird ausgegangen) in der nächsten Iteration ein valider Baustein entstehen wird. Diese neue Ausgangsebene, welche aus den „X“ und Bausteinen entsteht, wird nun als Eingabeebene für die nächste Zeile der Datei verwendet und wieder zerlegt in die einzelnen Zeichen usw. Das Programm erkennt, dass das Ende erreicht wird sobald ein „L“ in der Liste vorkommt. Deren implementierte Logik ist es, einfach den Lichtausgang der Ebene zu übernehmen. Somit sind sie nur „An“ (True), wenn der entsprechende Lichtausgang des Sensor darüber Licht an dieser Stelle ausgibt.

Danach werden diese Ergebnisse in ein lesbares Format für Menschen konvertiert.

Beispiele

nandu1.txt

Q1: Off

Q2: Off

L1: On

L2: On

Q1: On

Q2: Off

L1: Off

L2: Off

Q1: Off

Q2: On

L1: Off

L2: Off

Q1: On

Q2: On

L1: Off

L2: Off

---

nandu2.txt

Q1: Off

Q2: Off

L1: Off

L2: On

Q1: On

Q2: Off

L1: On

L2: Off

Q1: Off

Q2: On

L1: On

L2: Off

Q1: On

Q2: On

L1: On

L2: Off

---

nandu3.txtQ1: Off

Q2: Off

Q3: Off

L1: On

L2: Off

L3: Off

L4: On

Q1: On

Q2: Off

Q3: Off

L1: Off

L2: On

L3: Off

L4: On

Q1: Off

Q2: On

Q3: Off

L1: On

L2: Off

L3: On

L4: On

Q1: On

Q2: On

Q3: Off

L1: Off

L2: On

L3: On

L4: On

Q1: Off

Q2: Off

Q3: On

L1: On

L2: Off

L3: Off

L4: Off

Q1: On

Q2: Off

Q3: On

L1: Off

L2: On

L3: Off

L4: Off

Q1: Off

Q2: On

Q3: On

L1: On

L2: Off

L3: On

L4: Off

Q1: On

Q2: On

Q3: On

L1: Off

L2: On

L3: On

L4: Off

Ausgaben für nandu4.txt und nandu5.txt befinden sich im Ordner “results” in dem Aufgabenordner aufgrund deren Länge.

Freiwillige Webversion

Im Ordner der Aufgabe befindet sich ein Ordner „webVersion“. Dieser enthält eine „index.html“-Datei. Damit kann man die Website im Browser öffnen. In der „README.md“ bzw. „README.html“ stehen die Funktionalitäten sowie die Bedienung dieser Version. Die Version benutzt ausschließlich HTML, CSS und JS und keine externen Abhängigkeiten wie JQuery. Somit kann diese auch Offline verwendet werden.

Quellcode

(_:start:field): Ignoriert die erste Zeile, schreibt die zweite Zeile als String in “start” und die restlichen Zeilen als Liste von Strings in “field” hinein.

```haskell
-- steps to solve the problem for the given file
solve :: [String] -> ([Bool], [([Bool], [Bool])])
solve [] = error "Invalid input file?"   -- file needs at least 3 lines anyways
solve [_] = solve []
solve (_:start:field) = let  -- disregard of the first line in the file since it is not needed
    (lights, lightCount) = lightInformation start [] 0  -- obtaining bitmap of light/nolight
    in (
        lights,
        map (\x -> (buildInitialLight lights x 0, runBuilding field (buildInitialLight lights x 0))) [0..2 `intpow` lightCount - 1]  --calculating each option of the sates
    )
```

Parsen der ersten Zeile: Wo befinden sich die Lichter und wie viele gibt es

```haskell
-- mapping of where a light source is
lightInformation :: String -> [Bool] -> Int -> ([Bool], Int)
lightInformation [] lights count = (lights, count)  -- should never happen though
lightInformation (c:lightLine) lights count
    | null lightLine && c == ' ' = (lights, count)
    | null lightLine = (lights ++ [c == 'Q'], if c == 'Q' then count + 1 else count)
    | c == 'Q' = lightInformation lightLine (lights ++ [True]) (count + 1)
    | c == 'X' = lightInformation lightLine (lights ++ [False]) count
    | otherwise = lightInformation lightLine lights count
```

```haskell
-- the first layer with the lights matching the "light number" (light states encoded in bits)
-- uses bitwise and:
buildInitialLight :: [Bool] -> Int -> Int -> [Bool]
buildInitialLight lightPositions lightCode count
    | null lightPositions = []
    | otherwise =
      let pos = head lightPositions
      in (pos && (.&.) lightCode (2 `intpow` count) == 2 `intpow` count) : buildInitialLight (tail lightPositions) lightCode (if pos then count + 1 else count)

-- own power function to satisfy types in buildInitialLight
intpow :: (Num t1, Num t2, Eq t1, Eq t2) => t2 -> t1 -> t2
intpow a b
    | b == 0 = 1
    | a == 0 = 0
    | otherwise = a * intpow a (b-1)
```

```haskell
-- calculate the output for the given input lights and input bricks
runBuilding :: [String] -> [Bool] -> [Bool]
runBuilding f light
    | null f || head f == "" = light
    | otherwise = let
        cur = head f
        nexts = tail f
        in
            runBuilding nexts (translateBlock (tokenize cur) light [] 0 False)

-- tokenizing a line into list of Char:
-- "X  X" -> ['X', 'X']
tokenize :: String -> [Char]
tokenize = filter (\x -> x /= ' ' && not (isDigit x))
```

Die Berechnung der Ebenen:

```haskell
-- converting current light states with the described bricks to the new state of the lights
translateBlock :: [Char] -> [Bool] -> [Bool] -> Int -> Bool -> [Bool]
translateBlock tokenizedRow lights newLights index hadBrick
    | index >= length tokenizedRow = newLights
    | not hadBrick && brick tokenizedRow index == ('W', 'W') = let
        brickResult = whiteBrick (lightStateSegment lights index)
        in
            fst brickResult : snd brickResult : translateBlock tokenizedRow lights newLights (index + 1) True
    | not hadBrick && brick tokenizedRow index == ('B', 'B') = let
        brickResult = blueBrick (lightStateSegment lights index)
        in
            fst brickResult : snd brickResult : translateBlock tokenizedRow lights newLights (index + 1) True
    | not hadBrick && brick tokenizedRow index == ('R', 'r') = let
        brickResult = redBrickLSensor (lightStateSegment lights index)
        in
            fst brickResult : snd brickResult : translateBlock tokenizedRow lights newLights (index + 1) True
    | not hadBrick && brick tokenizedRow index == ('r', 'R') = let
        brickResult = redBrickRSensor (lightStateSegment lights index)
        in
            fst brickResult : snd brickResult : translateBlock tokenizedRow lights newLights (index + 1) True
    | tokenizedRow !! index == 'X' = False : translateBlock tokenizedRow lights newLights (index + 1) False
    | tokenizedRow !! index == 'L' = (lights !! index) : translateBlock tokenizedRow lights newLights (index + 1) False
    | otherwise = translateBlock tokenizedRow lights newLights (index + 1) False
```

Andere Hilfsfunktionen und die Logik der Bausteine

```haskell
-- create a tuple of two elements in a list
listTuple :: Show a => [a] -> Int -> (a, a)
listTuple lst lastIndex = (lst !! (lastIndex - 1), lst !! lastIndex)

-- convert at the index-1 and index position into a tuple to potentially get a valid brick
brick :: [Char] -> Int -> (Char, Char)
brick lst lastIndex = if lastIndex == 0 || lastIndex >= length lst
    then ('N', 'N')
    else listTuple lst lastIndex

-- same as listTuple, but with more type safety
lightStateSegment :: [Bool] -> Int -> (Bool, Bool)
lightStateSegment = listTuple -- only called if brick succeeded -> lastIndex will always be >= 1

-- functionality of the bricks
-- white
whiteBrick :: (Bool, Bool) -> (Bool, Bool)
whiteBrick (left, right) = let res = not left && not right in (res, res)

-- blue
blueBrick :: (Bool, Bool) -> (Bool, Bool)
blueBrick (left, right) = (left, right)

-- red: R r
redBrickLSensor :: (Bool, Bool) -> (Bool, Bool)
redBrickLSensor (left, _) = let res = not left in (res, res)

-- red: r R
redBrickRSensor :: (Bool, Bool) -> (Bool, Bool)
redBrickRSensor (_, right) = let res = not right in (res, res)
```