German University in Cairo
Faculty of Media Engineering and Technology
Dr. Nada Sharaf

Concepts of Programming Languages, Spring 2021
CPU Cache: Haskell functions and Examples
Deadline: 25 June 2021

Representations

Cache

The cache is represented by list of items. The item is a type that is defined as follows:

```
data Item a = It Tag (Data a) Bool Int | NotPresent deriving (Show, Eq)
data Tag = T Int deriving (Show, Eq)
data Data a = D a deriving (Show, Eq)
```

- A single item is represented using the type `Item a` using two constructors `It` and `NotPresent`, where the `It` constructor is used to represent an entry to be placed/removed from the cache, while the `NotPresent` constructor is used when data can't be retrieved from the cache.
  - The `It` constructor carries the tag, data, validity and order of an entry in the cache.
  - The tag and data are represented using the types `Tag` and `Data` respectively
  - The third `Bool` argument in the `It` constructor represents the validity of the cache item, while the `Int` that follows represents its order.
  - The `NotPresent` constructor is used to denote when the data couldn't be retrieved from the cache.
- `Tag` is a type with constructor `T` that represents the tag of an address as an Int.
- `Data` is a type that is represented with the constructor `D` and carries data of type `a`.
- `ValidBit` is the `Bool` present in the `Item` type representing the validity of the item in the cache.
- `Order` is an `Int` representing the placement order of the item in the cache. The lower the number, the newer it is, where zero is the least number.

An example of a cache is shown below:

```
[(It (T 0000) (D "1") False 0), (It (T 0000) (D "b") True 0),
(It (T 0001) (D "0") False 3), (It (T 0001) (D "c") False 2)]
```

This example represents the cache:

Block Index	Tag	Valid	Data
000	0000	False	"1"
001	0000	True	"b"
010	0001	False	"0"
011	0001	False	"c"

Memory

The memory is represented as a list and the order inside it represent the address meaning the first element in the list is of address zero.
e.g. `["100000","100001","100010",`
`"100011","100100","100101","100110","100111"]`

Functions to be added

You are going to implement this system purely through Haskell, you can add as many funstions as you need to make sure the following functions work correctly. You **have to implement ALL of the following functions. Your implementation should be GENERIC meaning it accepts cache and memory of any size.**

The general section should be implemented by the whole team. On the other hand, the cache section should be divided by the team and each member should be responsible for the part chosen.

General

To be implemented by the whole team.

convertBinToDec

```
convertBinToDec :: Integral a => a -> a
```

The function `convertBinToDec bin` converts a binary number `bin` to its decimal equivalent .

```
Example:
> convertBinToDec 0101
5

> convertBinToDec 1101
13

> convertBinToDec 10
2
```

replaceIthItem

```
replaceIthItem :: t -> [t] -> Int -> [t]
```

The function `replaceIthItem item l index` replaces the element at `index` with `item` in list `l`.

```
Example:
> replaceIthItem 'a' ['1', '2', '3', '4'] 2
"12a4"

> replaceIthItem 5 [1, 2, 3, 4, 9, 8, 7, 6] 5
[1,2,3,4,9,5,7,6]
```

splitEvery

```
splitEvery :: Int -> [a] -> [[a]]
```

The function `splitEvery n l` splits every `n` consecutive elements in the list `l` grouping them in a list maintaining the order

```
Example:
> splitEvery 2 ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
["ab","cd","ef","gh"]

> splitEvery 4 [1, 2, 3, 4, 5, 6, 7, 8]
[[1,2,3,4],[5,6,7,8]]

> splitEvery 8 ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
["abcdefgh"]
```

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Nada Sharaf

logBase2

```
logBase2 :: Floating a => a -> a
```

The function `logBase2 num` computes log_2(`num`).

```
Example:
> logBase2 8
3.0

> logBase2 1
0.0

> logBase2 32
5.0
```

getNumBits

```
getNumBits  :: (Integral a, RealFloat a1) =>
                    a1 -> [Char] -> [c] -> a
```

Given the number of sets `numOfSets`, the cache mapping type `cacheType` and the cache `cache`, the function `getNumBits numOfSets cacheType cache` computs the number of bits required for the index. **Hint:** You might find the function fromIntegral useful.

```
Examples:
> getNumBits 1  "fullyAssoc" [(It (T 0000) (D "1") False 0), (It (T 0000)
(D "b") True 0), (It (T 0001) (D "0") False 3), (It (T 0001) (D "c") False 2)]

0

> getNumBits 2  "setAssoc" [(It (T 0000) (D "1") False 0), (It (T 0000)
(D "b") True 0), (It (T 0001) (D "0") False 3), (It (T 0001) (D "c") False 2)]

1

> getNumBits 4  "directMap" [(It (T 0000) (D "1") False 0), (It (T 0000)
(D "b") True 0), (It (T 0001) (D "0") False 3), (It (T 0001) (D "c") False 2)]

2
```

fillZeros

```
fillZeros :: [Char] -> Int -> [Char]
```

Given a string representing a number `s` and a number `n`, the function `fillZeros s n` adds `n` preceding zeros to the string.

```
Examples:
> fillZeros "100" 1
"0100"

> fillZeros "10" 4
"000010"

> fillZeros "0" 2
"000"
```

Cache

To be divided by the team members

Cache Mapping 1: Direct Mapping

getDataFromCache

```
getDataFromCache
  :: (Integral b, Eq a) =>
     [Char] -> [Item a] -> [Char] -> b -> Output a
```

The function `getDataFromCache stringAddress cache "directMap" bitsNum` returns an argument of type `Output`:

```
data Output a = Out (a, Int) | NoOutput deriving (Show, Eq)
```

, where the constructor `Out` consists of a tuple containing the data retrieved from the cache and the hops number performed respectively `Out (data, hopsNum)`. The `hopsNum` denotes the number of misses that occured before reaching a hit. In case the data is not present in the cache, the value constructor `NoOutput` is returned.

- `stringAddress` is a string of the binary number which represents the address of the data you are required to address.
- `cache` is the cache using the representation discussed previously .

- `BitsNum` The BitsNum is the number of bits the index needs.

```
Example:
> getDataFromCache "000001" [(It (T 0000) (D "10000") False 1),
(It (T 0000) (D "110000") True 0), (It (T 0001) (D "11100") False 3),
(It (T 0001) (D "11110") False 2)] "directMap" 2

Out ("110000",0)

> getDataFromCache "000010" [(It (T 0000) (D "10000") False 1),
(It (T 0000) (D "110000") True 0), (It (T 0001) (D "11100") False 3),
(It (T 0001) (D "11110") False 2)] "directMap" 2

NoOutput

> getDataFromCache "111101" [(It (T 0000) (D "10000") False 1),
(It (T 0000) (D "110000") True 0), (It (T 0001) (D "11100") False 3),
(It (T 0001) (D "11110") False 2)] "directMap" 2

NoOutput
```

convertAddress

```
convertAddress
  :: (Integral b1, Integral b2) => b1 -> b2 -> p -> (b1, b1)
```

The function `convertAddress binAddress bitsNum "directMap"` returns a tuple containing the tag and index `(tag, index)` in binary computed from the given address `binAddress` and `bitsNum` (which denotes the number of bits needed to represent the index) according to the cache type provided.

```
Example:
> convertAddress 1110 2 "directMap"
(11,10)

> convertAddress 11100 2 "directMap"
(111,0)

> convertAddress 000011 2 "directMap"
(0,11)
```

replaceInCache

```
replaceInCache
  :: Integral b =>
     Int -> Int -> [a] -> [Item a] -> [Char] -> b -> (a, [Item a])
```

The function `replaceInCache tag idx memory oldCache "directMap" bitsNum` updates the cache with the data retrieved from the memory by replacing an item according to direct mapping cache technique. The `tag` and `idx` are the memory address components used to access the cache. They are numbers provided in binary format. `oldCache` represents the cache before replacement. `bitsNum` represent the number of bits used to represent the index. The function returns a tuple containing the data retrieve from the memory and the updated cache respectively `(data, updatedCache)`.

Example:

```
> replaceInCache 0 10 ["100000","100001","100010", "100011",
"100100","100101","100110","100111"] [(It (T 0000) (D "10000") False 1),
(It (T 0000) (D "100001") True 0), (It (T 0001) (D "11100") False 3),
(It (T 0001) (D "11110") False 2)]  "directMap" 2

(
    "100010",
    [It (T 0) (D "10000") False 1,It (T 0) (D "100001") True 0,
    It (T 0) (D "100010") True 0,It (T 1) (D "11110") False 2]
)


> replaceInCache 0001 11 ["100000","100001","100010", "100011",
"100100","100101","100110","100111"] [(It (T 0000) (D "10000") False 1),
(It (T 0000) (D "100001") True 0), (It (T 0001) (D "11100") False 3),
(It (T 0001) (D "11110") False 2)] "directMap" 2

(
    "100111",
    [It (T 0) (D "10000") False 1,It (T 0) (D "100001") True 0,
    It (T 1) (D "11100") False 3,It (T 1) (D "100111") True 0]
)
```

getData

Note: This function is already implemented at the end of the description document. No need to re-implement it.

```
getData
  :: (Integral b, Eq a) =>
     [Char] -> [Item a] -> [a] -> [Char] -> b -> (Output a, [Item a])
```

The function `getData stringAddress oldCache memory cacheType bitsNum` returns a tuple containing the data retrieved from the cache and the final state of the cache respectively `(data, newCache)`. If the data wasn't present in the cache, it retrieves it from the memory, places it in the cache and returns it along with the updated cache in the output tuple. The data retrieval and replacement are done according to the direct Map cache technique. `BitsNum` is the number of bits the index needs.

```
Example:
> getData "000001" [It (T 0000) (D "10000") False 1,
It (T 0000) (D "100001") True 0, It (T 0001) (D "11100") False 3,
It (T 0001) (D "11110") False 2] ["100000","100001","100010","100011",
"100100","100101","10011","100111"] "directMap" 2

(
    "100001",
    [It (T 0) (D "10000") False 1,It (T 0) (D "100001") True 0,
    It (T 1) (D "11100") False 3,It (T 1) (D "11110") False 2]
)

> getData "000001" [It (T 0000) (D "10000") False 1,
It (T 0000) (D "100001") False 0, It (T 0001) (D "11100") False 3,
It (T 0001) (D "11110") False 2] ["100000","100101","100010","100011",
"100100","100101","10011","100111"] "directMap" 2

(
    "100101",
    [It (T 0) (D "10000") False 1,It (T 0) (D "100101") True 0,
    It (T 1) (D "11100") False 3,It (T 1) (D "11110") False 2]
)
```

runProgram

Note: This function is already implemented at the end of the description document. No need to re-implement it.

```
runProgram
  :: (RealFloat a1, Eq a2) =>
     [[Char]] -> [Item a2] -> [a2] -> [Char] -> a1 -> ([a2], [Item a2])
```

`runProgram adressList oldCache memory cacheType numOfSets` Given a list of addresses `adressList`, `runProgram` function returns a tuple containing a list of the data retrieved and the final state of the cache `(outputDataList, FinalCache)`. The `NumOfSets` is the same as the size of the cache. The runProgram should utilise direct mapping in this case.

```
Example:
> runProgram ["000011","000100","000011","001000"]
[It (T 0) (D "") False 1, It (T 0) (D "") False 0,
It (T 1) (D "") False 3, It (T 1) (D "") False 2]
["a", "b", "c", "d", "e", "f", "ab", "ac", "ad",
"ae", "af"] "directMap" 4

(
    ["d","e","d","ad"],
    [It (T 10) (D "ad") True 0,It (T 0) (D "") False 0,
    It (T 1) (D "") False 3,It (T 0) (D "d") True 0]
)
```

Cache Mapping 2: Fully Associative

getDataFromCache

```
getDataFromCache
  :: (Integral b, Eq a) =>
     [Char] -> [Item a] -> [Char] -> b -> Output a
```

The function `getDataFromCache stringAddress cache "fullyAssoc" bitsNum` returns an argument of type `Output`:

```
data Output a = Out (a, Int) | NoOutput deriving (Show, Eq)
```

, where the constructor `Out` consists of a tuple containing the data retrieved from the cache and the hops number performed respectively `Out (data, hopsNum)`. The `hopsNum` denotes the number of misses that occured before reaching a hit. In case the data is not present in the cache, the value constructor `NoOutput` is returned.

- `stringAddress` is a string of the binary number which represents the address of the data you are required to address.
- `cache` is the cache using the representation discussed previously .
- `BitsNum` The BitsNum is the number of bits the index needs.

Example:

```
> getDataFromCache "000001" [(It (T 000000) (D "10000") False 1),
(It (T 000001) (D "11000") True 0), (It (T 000100) (D "11100") False 3),
(It (T 000101) (D "11110") False 2)] "fullyAssoc" 0

Out ("11000",1)

> getDataFromCache "000011" [(It (T 000000) (D "10000") False 1),
(It (T 000001) (D "11000") True 0), (It (T 000100) (D "11100") False 3),
(It (T 000101) (D "11110") False 2)] "fullyAssoc" 0

NoOutput
```

convertAddress

```
convertAddress
  :: (Integral b1, Integral b2) => b1 -> b2 -> p -> (b1, b1)
```

The function `convertAddress binAddress bitsNum "fullyAssoc"` returns a tuple containing the tag and index `(tag, index)` in binary computed from the given address `binAddress` and `bitsNum` (which denotes the number of bits needed to represent the index) according to the cache type provided.

Example:

```
> convertAddress 001110 0 "fullyAssoc"
(1110,0)

> convertAddress 000011 0 "fullyAssoc"
(11,0)

> convertAddress 000001 0 "fullyAssoc"
(1,0)
```

replaceInCache

```
replaceInCache
  :: Integral b =>
     Int -> Int -> [a] -> [Item a] -> [Char] -> b -> (a, [Item a])
```

The function

```
replaceInCache tag idx memory oldCache "fullyAssoc" bitsNum
```

updates the cache with the data retrieved from the memory by replacing an item according to fully associative cache technique, where FIFo replacement policy is followed. The `tag` and `idx` are the memory address components used to access the cache. They are numbers provided in binary format. `oldCache` represents the cache before replacement. `bitsNum` represent the number of bits used to represent the index. The function returns a tuple containing the data retrieve from the memory and the updated cache respectively `(data, updatedCache)`

Note that: invalid data is considered as empty space and have the priority to replace over the order of replacement.

```
Example:
> replaceInCache 1 0 ["100000","100001","100010", "100011","100100",
"100101","100110","100111"] [(It (T 000000) (D "10000") False 1),
(It (T 000010) (D "100010") True 0), (It (T 000100) (D "11100") False 3),
(It (T 000101) (D "11110") False 2)]  "fullyAssoc" 1

(
    "100001",
    [It (T 1) (D "100001") True 0,It (T 10) (D "100010") True 1,
    It (T 100) (D "11100") False 3,It (T 101) (D "11110") False 2]
)

> replaceInCache 1 0 ["100000","100001","100010", "100011","100100",
"100101","100110","100111"] [(It (T 000000) (D "10000") True 1),
(It (T 000010) (D "100100") True 0), (It (T 000100) (D "100100") True 3),
(It (T 000101) (D "100101") True 2)] "fullyAssoc" 1

(
    "100001",
    [It (T 0) (D "10000") True 2,It (T 10) (D "100100") True 1,
    It (T 1) (D "100001") True 0,It (T 101) (D "100101") True 3]
)
```

```
> replaceInCache 1 0 ["100000","100001","100010", "100011","100100",
"100101","100110","100111"] [(It (T 000001) (D "100001") False 1),
(It (T 000010) (D "100010") True 0), (It (T 000100) (D "100100") True 3),
(It (T 000101) (D "100101") True 2)] "fullyAssoc" 1

(
    "100001",
    [It (T 1) (D "100001") True 0,It (T 10) (D "100010") True 1,
    It (T 100) (D "100100") True 4,It (T 101) (D "100101") True 3]
)
```

getData

Note: This function is already implemented at the end of the description document. No need to re-implement it.

```
getData
  :: (Eq t, Integral b) =>
     String -> [Item t] -> [t] -> [Char] -> b -> (t, [Item t])
```

The function `getData stringAddress oldCache memory cacheType bitsNum` returns a tuple containing the data retrieved from the cache and the final state of the cache respectively `(data, newCache)`. If the data wasn't present in the cache, it retrieves it from the memory, places it in the cache and returns it along with the updated cache in the output tuple. The data retrieval and replacement are done according to the fully associative cache technique. `BitsNum` is the number of bits the index needs.

`Examples:`

```
> getData "000001" [It (T 0) (D "10000") True 1,
It (T 1) (D "11000") True 0, It (T 10) (D "11100") False 3,
It (T 0) (D "11110") False 2] ["100000","100001","100010",
"100011","100100","100101","100110","100111"] "fullyAssoc" 0

(
    "11000",
    [It (T 0) (D "10000") True 1,It (T 1) (D "11000") True 0,
    It (T 10) (D "11100") False 3,It (T 0) (D "11110") False 2]
)
```

```
> getData "000001" [It (T 1) (D "100001") False 1,
It (T 10) (D "100010") True 0, It (T 100) (D "100100") True 3,
It (T 101) (D "100101") True 2] ["100000","100001","100010",
"100011","100100","100101","100110","100111"] "fullyAssoc" 0

(
    "100001",
    [It (T 1) (D "100001") True 0,It (T 10) (D "100010") True 1,
    It (T 100) (D "100100") True 4,It (T 101) (D "100101") True 3]
)
```

runProgram

Note: This function is already implemented at the end of the description document. No need to re-implement it.

```
runProgram
  :: (RealFloat a1, Eq a2) =>
     [[Char]] -> [Item a2] -> [a2] -> [Char] -> a1 -> ([a2], [Item a2])
```

`runProgram adressList oldCache memory cacheType numOfSets` Given a list of addresses `adressList`, `runProgram` function returns a tuple containing a list of the data retrieved and the final state of the cache `(outputDataList, FinalCache)`. The `NumOfSets` is 1 as the entire cache is considered a set. The runProgram should utilise fully associative cache in this case.

```
> runProgram ["000000","000001","000000","000011"]
[It (T 0) (D "10000") False 0, It (T 0) (D "11000") False 0,
It (T 1) (D "11100") False 3, It (T 1) (D "e") False 0]
["a", "b", "c", "d", "e", "f", "ab", "ac", "ad", "ae"] "fullyAssoc" 1

(
    ["a","b","a","d"],
    [It (T 0) (D "a") True 2,It (T 1) (D "b") True 1,
    It (T 11) (D "d") True 0,It (T 1) (D "e") False 0]
)
```

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Nada Sharaf

Cache Mapping 3: Set-associative

getDataFromCache

```
getDataFromCache
  :: (Integral b, Eq a) =>
     [Char] -> [Item a] -> [Char] -> b -> Output a
```

The function `getDataFromCache stringAddress cache "setAssoc" bitsNum` returns an argument of type `Output`:

```
data Output a = Out (a, Int) | NoOutput deriving (Show, Eq)
```

, where the constructor `Out` consists of a tuple containing the data retrieved from the cache and the hops number performed respectively `Out (data, hopsNum)`. The `hopsNum` denotes the number of misses that occured before reaching a hit. In case the data is not present in the cache, the value constructor `NoOutput` is returned.

- `stringAddress` is a string of the binary number which represents the address of the data you are required to address.
- `cache` is the cache using the representation discussed previously .
- `BitsNum` The BitsNum is the number of bits the index needs.

```
Examples:
Hit case
> getDataFromCache "000001" [(It (T 00000) (D "11100") False 3),
(It (T 00000) (D "11110") False 2), (It (T 0000) (D "10000") False 1),
(It (T 00000) (D "11000") True 0)] "setAssoc" 1

Out ("11000",1)

Miss case
> getDataFromCache "000001" [(It (T 00000) (D "10000") False 1),
(It (T 00000) (D "100000") True 0), (It (T 00010) (D "11100") False 3),
(It (T 00000) (D "11110") False 2)] "setAssoc" 1

NoOutput

Miss case
> getDataFromCache "000000" [(It (T 00000) (D "10000") False 1),
(It (T 00001) (D "11000") True 0), (It (T 00010) (D "11100") False 3),
```

```
(It (T 00000) (D "11110") True 0)] "setAssoc" 1

NoOutput

Hit case
> getDataFromCache "000000" [(It (T 00000) (D "10000") True 1),
(It (T 00001) (D "11000") True 0), (It (T 00010) (D "11100") False 3),
(It (T 00000) (D "11110") True 0)] "setAssoc" 1

Out ("10000",0)
```

convertAddress

```
convertAddress
  :: (Integral b1, Integral b2) => b1 -> b2 -> p -> (b1, b1)
```

The function `convertAddress binAddress bitsNum "setAssoc"` returns a tuple containing the tag and index `(tag, index)` in binary computed from the given address `binAddress` and `bitsNum` (which denotes the number of bits needed to represent the index) according to the cache type provided.

```
Example:
> convertAddress 001110 2 "setAssoc"
(11,10)

> convertAddress 000011 2 "setAssoc"
(0,11)

> convertAddress 001110 3 "setAssoc"
(1,110)

> convertAddress 001110 0 "setAssoc"
(1110,0)
```

replaceInCache

```
replaceInCache
  :: Integral b =>
     Int -> Int -> [a] -> [Item a] -> [Char] -> b -> (a, [Item a])
```

The function

```
replaceInCache tag idx memory oldCache "setAssoc" bitsNum
```

updates the cache with the data retrieved from the memory by replacing an item according to set associative cache technique, where FIFo replacement policy is followed. The `tag` and `idx` are the memory address components used to access the cache. They are numbers provided in binary format. `oldCache` represents the cache before replacement. `bitsNum` represent the number of bits used to represent the index. The function returns a tuple containing the data retrieve from the memory and the updated cache respectively `(data, updatedCache)` **Note that**: invalid data is considered as empty space and have the priority to replace over the order of replacement.

```
Examples:
> replaceInCache 0 1 ["100000","100001","100010", "100011","100100",
"100101","100110","100111"] [(It (T 00000) (D "10000") False 1),
(It (T 00000) (D "100000") True 0), (It (T 00010) (D "11100") False 3),
(It (T 00000) (D "11110") False 2)] "setAssoc" 1

(
    "100001",
    [It (T 0) (D "10000") False 1,It (T 0) (D "100000") True 0,
    It (T 0) (D "100001") True 0,It (T 0) (D "11110") False 2]
)

> replaceInCache 11 1 ["100000","100001","100010", "100011","100100",
"100101","100110","100111"] [(It (T 00000) (D "10000") False 1),
(It (T 00000) (D "100000") True 0), (It (T 00001) (D "100011") True 0),
(It (T 00000) (D "100001") True 1)] "setAssoc" 1

(
    "100111",
    [It (T 0) (D "10000") False 1,It (T 0) (D "100000") True 0,
    It (T 1) (D "100011") True 1,It (T 11) (D "100111") True 0]
)

> replaceInCache 0 0 ["100000","100001","100010", "100011","100100",
"100101","100110","100111"] [(It (T 00000) (D "10000") False 0),
(It (T 00000) (D "100000") True 0), (It (T 00010) (D "11100") False 3),
(It (T 00000) (D "11110") False 2)] "setAssoc" 1

(
    "100000",
    [It (T 0) (D "100000") True 0,It (T 0) (D "100000") True 1,
```

```
    It (T 10) (D "11100") False 3,It (T 0) (D "11110") False 2]
)
```

getData

Note: This function is already implemented at the end of the description document. No need to re-implement it.

```
getData
  :: (Eq t, Integral b) =>
     String -> [Item t] -> [t] -> [Char] -> b -> (t, [Item t])
```

The function `getData stringAddress oldCache memory cacheType bitsNum` returns a tuple containing the data retrieved from the cache and the final state of the cache respectively `(data, newCache)`. If the data wasn't present in the cache, it retrieves it from the memory, places it in the cache and returns it along with the updated cache in the output tuple. The data retrieval and replacement are done according to the set associative cache technique. `BitsNum` is the number of bits the index needs.

```
Examples:
> getData "000001" [It (T 0) (D "10000") False 1,
It (T 0) (D "100000") True 0, It (T 10) (D "11100") False 3,
It (T 0) (D "11110") False 2] ["100000","100001","100010",
"100011","100100","100101","100110","100111"] "setAssoc" 1

(
    "100001",
    [It (T 0) (D "10000") False 1,It (T 0) (D "100000") True 0,
    It (T 0) (D "100001") True 0,It (T 0) (D "11110") False 2]
)

> getData "000001" [It (T 0) (D "10000") False 1,
It (T 0) (D "11000") True 0, It (T 10) (D "11100") False 3,
It (T 0) (D "11110") True 0] ["100000","100001","100010",
"100011","100100","100101","100110","100111"] "setAssoc" 1

(
    "11110",
    [It (T 0) (D "10000") False 1,It (T 0) (D "11000") True 0,
    It (T 10) (D "11100") False 3,It (T 0) (D "11110") True 0]
)
```

```
> getData "000000" [It (T 0) (D "10000") False 1,
It (T 1) (D "11000") True 0, It (T 10) (D "11100") False 3,
It (T 0) (D "11110") True 0] ["100000","100001","100010",
"100011","100100","100101","100110","100111"] "setAssoc" 1

(
    "100000",
    [It (T 0) (D "100000") True 0,It (T 1) (D "11000") True 1,
    It (T 10) (D "11100") False 3,It (T 0) (D "11110") True 0]
)
```

runProgram

Note: This function is already implemented at the end of the description document. No need to re-implement it.

```
runProgram
  :: (RealFloat a1, Eq a2) =>
     [[Char]] -> [Item a2] -> [a2] -> [Char] -> a1 -> ([a2], [Item a2])
```

`runProgram adressList oldCache memory cacheType numOfSets` Given a list of addresses `adressList`, `runProgram` function returns a tuple containing a list of the data retrieved and the final state of the cache `(outputDataList, FinalCache)`. The `NumOfSets` specifies the number of set present in the cache. The runProgram should utilise set associative cache in this case.

```
> runProgram ["000011","000100","000011","001100"]
[It (T 0) (D "10000") False 0, It (T 0) (D "11000") False 0,
It (T 1) (D "11100") False 3, It (T 1) (D "e") False 0]
["a", "b", "c", "d", "e", "f", "ab", "ac", "ad", "ae",
"af", "a", "a", "a", "a", "aa", "a"] "setAssoc" 2

(
    ["d","e","d","a"],
    [It (T 10) (D "e") True 1,It (T 110) (D "a") True 0,
    It (T 1) (D "d") True 0,It (T 1) (D "e") False 0]
)
```

German University in Cairo
Faculty of Media Engineering and Technology
Dr. Nada Sharaf

Implemented functions

getData

```
getData stringAddress cache memory cacheType bitsNum
    | x == NoOutput = replaceInCache tag index memory cache cacheType bitsNum
    | otherwise = (getX x, cache)
    where
        x = getDataFromCache stringAddress cache cacheType bitsNum
        address = read stringAddress:: Int
        (tag, index) = convertAddress address bitsNum cacheType

getX (Out (d, _)) = d
```

runProgram

```
runProgram [] cache _ _ _ = ([], cache)
runProgram (addr: xs) cache memory cacheType numOfSets =
        ((d:prevData), finalCache)
  where
   bitsNum = round(logBase2 numOfSets)
   (d, updatedCache) = getData addr cache memory cacheType bitsNum
   (prevData, finalCache) = runProgram xs updatedCache memory cacheType numOfSets
```