Introduction to Computer Science

Author: Daria Shutina

Haskell cheat sheet

A function that takes a Number (Integer, Double, Rational) and returns a Number

```haskell
f :: Num a => a -> a
f x = x^2
```

Lists

```haskell
concat [[1..3],[4..7],[8..10]]
reverse [1..10]

take 5 [1..]  -- [1, 2, 3, 4, 5]
drop 5 [1..10]  -- [6, 7, 8, 9, 10]

elem 5 [1..10]  -- if `5` is in the list

[1..] !! 9  -- take the 9-th elem, `!!` operation is not safe


```

Strings

```haskell
head "foo"  -- 'f'
tail "foo"  -- "oo"
last "foo"  -- 'o'
init "foo"  -- "fo"
```

map, filter, zip, zipWith

```haskell
map (\x -> x * 2 + 1) [1..3] -- [3, 5, 7]
map ((+ 1) . (* 2)) [1..10]  -- [3, 5, 7]

filter even [1..4]  -- [1, 3]

zip [1..] "hi"  -- [ (1,'h'), (2,'i') ]
zipWith (\a b -> (show a) ++ [b]) [1..] "hi"  -- [ "1h", "2i" ]
```

foldl, foldr

```
foldl (-) 0 [1..3]  -- ((0 - 1) - 2) - 3 = -6
foldr (-) 0 [1..3]  -- 1 - (2 - (3 - 0)) = 2
```

quickSort

```
import Data.List

quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = quickSort smaller ++ [x] ++ quickSort larger
    where
        (smaller, larger) = partition (< x) xs
```

Datatypes

```
data EmployeeInfo = Employee String Int Double [String] deriving (Show)
--       type                    data constructor(s)
--    constructor          consists of components or fields of type

p = Employee "Joe Sample" 22 186.3 []
```

```
data Person = Person
    { name :: String
    , age :: Int
    }
    deriving (Show, Eq)

increaseAge :: Person -> Person
increaseAge person = person { age = age person + 1 }
```

Binary tree

```
data Tree a = Empty
            | Leaf a
            | Branch a (Tree a) (Tree a)
            deriving (Eq, Show)

toList :: Tree a -> [a]
toList Empty = []
toList (Leaf x) = [x]
```

```haskell
toList (Branch x l r) = toList l ++ [x] ++ toList r

map :: (a -> b) -> Tree a -> Tree b
map f Empty = Empty
map f (Leaf x) = Leaf (f x)
map f (Branch x l r) = Branch (f x) (map f l) (map f r)

foldr :: (a -> b -> b) -> b -> Tree a -> b
foldr _ z Empty = z
foldr f z (Leaf x) = f x z
foldr f z (Branch x l r) = foldr f (f x (foldr f z r)) l

foldl :: (a -> b -> b) -> b -> Tree a -> b
foldl _ z Empty = z
foldl f z (Leaf x) = f x z
foldl f z (Branch x l r) = foldr f (f x (foldl f z l)) r
```

Typeclasses

Typeclasses describe a set of types that have a common interface and behavior. A type belonging to a typeclass implements the functions and behavior defined by the typeclass.

Let us look at the expression:

```haskell
(Eq a) => a -> a -> Bool
```

Everything before the symbol `=>` is a *class constraint*.

Usage of a constraint in a function declaration

We want to check whether the element is in the given list. We put a constraint on a type `a` -- it should be an instance of `Eq` typeclass.

```haskell
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem y (x:xs) = y == x || elem y xs
```

Eq

```
class Eq a where
    (==) :: a -> a -> Bool
    (==) a b = not (a /= b) -- default implementation
    (/=) :: a -> a -> Bool
    (/=) a b = not (a == b) -- default implementation
```

`a` is an instance of typeclass `Eq`. It can overload operators `==` and `/=` of use a default version.

Creating an instance of Eq

```
-- custom type
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

instance Eq Weekday where
    Mon == Mon = True
    Tue == Tue = True
    Wed == Wed = True
    Thu == Thu = True
    Fri == Fri = True
    Sat == Sat = True
    Sun == Sun = True
    _ == _ = False
```

```
-- lists
-- we create an instance of a list and also put a constraint on a type `a`
instance Eq a => Eq [a] where
    [] == [] = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _ == _ = False
```

Ord

It is an extension of `Eq`, so `Ord` is a subclass of `Eq`.

Default implementation:

```
data Ordering = LT | EQ | GT

instance Eq Ordering where
    LT == LT = True
    EQ == EQ = True
    GT == GT = True
```

```
    _ == _ = False

class (Eq a) => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    compare :: a -> a -> Ordering
    max, min :: a -> a -> a

    compare x y | x == y = EQ
                | x <= y = LT
                | otherwise = GT

    x <= y = compare x y /= GT
    x < y = compare x y == LT
    x >= y = compare x y /= LT
    x > y = compare x y == GT

    max x y | x <= y = y
            | otherwise = x

    min x y | x <= y = x
            | otherwise = y
```

Functor

Part 1

Maze solving algorithms

We can imagine that a maze is a graph. The cells are vertexes of the graph. If you can get from one cell to another, then there is an edge between the corresponding vertexes.

More formally, for a maze M we have a tuple $M = (T, S, X)$ where

- $T = (V, E)$ is a graph with the vertexes V and edges E;
- $S \in V$ is the start node;
- $X \in V$ is the exit node.

We want to find a solution for the maze $\Leftrightarrow$ we want to build a spanning tree.

Kruskal's algorithm

We want to build an MST T in the connected graph G.

Initially, T is an empty graph and every vertex forms a subset of a size 1.

Edges of G are sorted in an increasing order. We consider an edge and add it to the answer, if it connects vertexes from different subsets. Then two subsets are merged into one subset.

```cpp
std::vector<int> parent(n, -1);
std::vector<int> rank(n, 0);  // ну типа высота дерева

int find_set(int v) {
    if (parent[v] == v) {
        return v;
    }
    return parent[v] = find_set(parent[v]);
}

void union_set(int a, int b) {
    int v_a = find_set(a);
    int v_b = find_set(b);
    if (v_a == v_b) {
        return;
    }
    if (rank[v_a] < rank[v_b]) {
        std::swap(v_a, v_b);
    }
    parent[v_b] = v_a;
    if (rank[v_a] == rank[v_b]) {
        ++rank[v_a];
    }
}

int kraskal(std::set<std::pair<int, std::pair<int, int>>> &Edges) {
    int cost = 0;
    for (const auto &edge : Edges) {
        int w = edge.first, u = edge.second.first, v = edge.second.second;
        if (find_set(u) != find_set(v)) {
            union_set(u, v);
            cost += w;
        }
    }
    return cost;
}
```

Total compexity is $O(\log n)$.

String search algorithms

We need a program to find a (relatively short) string in a (possibly long) text.

Problem formalization

Σ is an alphabet. $k = |\Sigma|$.

Σ^* is a set of all words that can be created out of Σ.

$$\Sigma^0 = \{\epsilon\}$$
$$\Sigma^1 = \Sigma$$
$$\Sigma^i = \{wv \;:\; w \in \Sigma^{i-1} \;\wedge\; v \in \Sigma\},\ i > 1$$
$$\Sigma = \bigcup_{i \geqslant 0} \Sigma^i$$

$t \in \Sigma^*$ is a text and $p \in \Sigma^*$ is a pattern.

$|t| = n,\ |p| = m,\ n > m.$

We need to find the first occurrence of p in t.

Naive String Search

Check <u>at each position</u> whether the pattern matches.

Lowercase characters indicate comparisons that were skipped.

Total complexity is $O(nm)$.

Example

Example: t = FINDANEEDLEINAHAYSTACK, p = NEEDLE

```
F I N D A N E E D L E I N A H A Y S T A C K

N e e d l e
  N e e d l e
    N E e d l e
      N e e d l e
        N e e d l e
          N E E D L E
```

Boyer-Moore algorithm

The idea is to compare the pattern right to left instead left to right. If there is a mismatch, try to move the pattern as much as possible to the right.

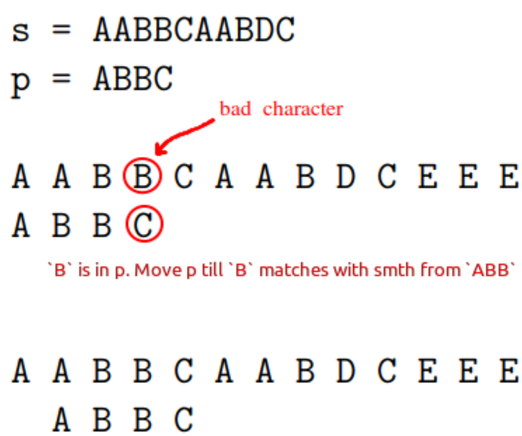
Bad character rule

$s :=$ `AABBCAABDCEEE`, $p :=$ `ABBC`.

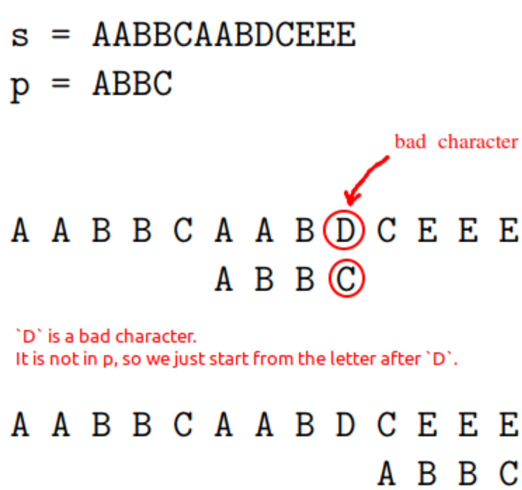
We begin to compare the first alignment from right to left. `B` from `s` and `C` from `p` mismatch, so `B` is called a *bad character*.

There are two cases of a bad character:

1. The bad character is in p. Then we move p till the bad character matches with some letter from p.



2. The bad character is not in p. Then we move p so that it starts from the position which has not been considered yet.



Implementation

For the bad character rule, we need a function that takes the bad character and returns the number of alignments that can be skipped. We can pre-compute all possible skips and store the skips in a two-dimensional table. The table can be seen as a function that maps the unmatched character and the position in the pattern to the number of alignments that can be skipped.

For example, $p :=$ `NEED`. The complete table is shown on the left of the picture. The behaviour is the same for letters which are not present in p, so we can make the table shorter, like on the right of the picture.

$$\begin{array}{cc} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \\ \vdots \\ N \\ \vdots \\ Z \end{array} & \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & - \\ 0 & - & - & 0 \\ \vdots & \vdots & \vdots & \vdots \\ - & 0 & 1 & 2 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 2 & 3 \end{pmatrix} \end{array} \qquad \begin{array}{cc} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} D \\ E \\ N \\ * \end{array} & \begin{pmatrix} 0 & 1 & 2 & - \\ 0 & - & - & 0 \\ - & 0 & 1 & 2 \\ 0 & 1 & 2 & 3 \end{pmatrix} \end{array}$$

Example using the table:

```
s = D D D D D D D
p = N E E D
    0 1 2 3


1)   D  D  D [D]  D  D  D
     N  E  E [D]

D:D --> matched


2)   D  D [D] D  D  D  D
     N  E [E] D

D:E --> table[D][2] = 2
    --> 2 alignments are skipped.


3)   D  D  D  D  D  D [D]
              N  E  E [D]


```

```
4) etc.
```

Landau notations

$$f(n) = O(g(n)) \; \Leftrightarrow \exists C \in R, n_0 \; : \; f(n) < C \cdot g(n), \forall n \geqslant n_0.$$

$$f(n) = \Omega(g(n)) \; \Leftrightarrow \; \exists C \in R, n_0 \; : \; f(n) > C \cdot g(n), \forall n \geqslant n_0.$$

$$f(n) = \Theta(g(n)) \; \Leftrightarrow \; \exists C_1, C_2 \in R, n_0 \;\; C_1 \cdot g(n) < f < C_2 \cdot g(n), \forall n \geqslant n_0.$$

Th. (Landau Set Ranking)

The commonly used Landau Sets establish a ranking such that

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^k) \subset O(I^n)$$

for all $k > 2$ and $I > 1$.

Th. (Landau Set Computation Rules)

1. $\left.\begin{array}{l} k \neq 0 \\ f = O(g) \end{array}\right\} \Rightarrow \; kf = O(g)$

2. $\left.\begin{array}{l} f_1 = O(g_1) \\ f_2 = O(g_2) \end{array}\right\} \Rightarrow \; f_1 + f_2 = O(\max(g_1, g_2))$

3. $\left.\begin{array}{l} f_1 = O(g_1) \\ f_2 = O(g_2) \end{array}\right\} \Rightarrow \; f_1 f_2 = O(g_1 g_2)$

(Non)determinism or randomness

A *deterministic algorithm* is an algorithm which, given a particular input, will always produce the same output, with the execution always passing through the same sequence of states.

A *nondeterministic algorithm* is an algorithm that can exhibit different behaviors on the same input.

A *randomized algorithm* is a (nondeterministic) algorithm that employs a degree of randomness as part of its logic. Random number generators often use algorithms to produce so called *pseudo random numbers* -- sequences of numbers that "look" random but that are not really random (since they are calculated using a deterministic algorithm).

Part 2

Relations

Definition

A relation $R \subseteq A \times A$ is called

- *reflexive* iff $\forall a \in A.(a, a) \in R$
- *irreflexive* iff $\forall a \in A.(a, a) \notin R$
- *symmetric* iff $\forall a, b \in A.(a, b) \in R \Rightarrow (b, a) \in R$
- *asymmetric* iff $\forall a, b \in A.(a, b) \in R \Rightarrow (b, a) \notin R$
- *antisymmetric* iff $\forall a, b \in A.((a, b) \in R \wedge (b, a) \in R) \Rightarrow a = b$
- *transitive* iff $\forall a, b, c \in A.((a, b) \in R \wedge (b, c) \in R) \Rightarrow (a, c) \in R$
- *connected* iff $\forall a, b \in A.(a, b) \in R \vee (b, a) \in R \vee a = b$
- *serial* iff $\forall a \in A.\exists b \in A.(a, b) \in R$

A relation R ⊆ A × A is called an *equivalence relation* on A if and only if R is reflexive, symmetric, and transitive.

A relation R ⊆ A × A is called a *partial order* on A if and only if R is reflexive, antisymmetric, and transitive on A.

A relation R ⊆ A × A is called a *strict partial order* on A if and only if it is irreflexive, asymmetric and transitive on A.

Functions

A relation $f \subseteq X \times Y$ is a *partial function* $\Leftrightarrow \forall x \in X$ there is at most one $y \in Y$ with $(x, y) \in f$
.

f is undefined at $x \in X$ if $(x, y) \notin f, \ \forall y \in Y$. In this case, we write $f(x) = \perp$.

A relation $f \subseteq X \times Y$ is a *total function* $\Leftrightarrow \forall x \in X$ there is exactly one $y \in Y$ such that $(x, y) \in f$.

If f is a total function, we write $f\ :\ X \rightarrow Y$.

Function Properties

Given a function $f\ :\ X \rightarrow Y$.

- injective $\Leftrightarrow \forall y \in Y$ is mapped to by at most one element from X:
  $\forall x_1, x_2 \in X\ :\ f(x_1) = f(x_2) \Rightarrow x_1 = x_2$
- surjective $\Leftrightarrow \forall y \in Y$ is mapped to by at least one element from X:
  $\forall y \in Y\ \exists x \in X\ :\ f(x) = y$
- bijective $\Leftrightarrow \forall y \in Y$ is mapped to by exactly one element from X. Thus, a function should be both injective and surjective.

Lambda Notation of Functions

$\lambda x \in N\ .\ E$, where N stands for natural numbers and E is a function body. Examples:

- $\lambda x \in N. x\ \Leftrightarrow\ f(x) = x\ \Leftrightarrow\ \{(x, y) \in N \times N\ :\ y = x\}$
- $\lambda x \in N. x^2\ \Leftrightarrow\ f(x) = x^2\ \Leftrightarrow\ \{(x, y) \in N \times N\ :\ y = x^2\}$
- $$\lambda(x, y) \in N \times N. x + y\ \Leftrightarrow\ f(x, y) = x + y\ \Leftrightarrow\ \{((x, y), z) \in (N \times N) \times N\ : z = x + y\}$$

Part 3

Integer Numbers

$(b-1)$-Complement Fixed Integer Numbers

We have a fixed number space with n digits and base b. It can represent b^n different numbers.

Positive numbers are represented in the usual way.

To represent negative numbers, we invert the absolute value $(a_n \dots a_0)_b$ into $(a'_n \dots a'_0)_b$ where $a'_i = (b-1) - a_i$.

Thus, we will have $+0$ and -0, so there are $b^n - 1$ different numbers.

Example:

$b = 2, n = 4 \ : \ 5_{10} = 0101_2, \ \ -5_{10} = 1010_2.$

b-Complement Fixed Integer Numbers

We have a fixed number space with n digits and a base b.

Positive numbers are represented in the usual way.

To represent negative numbers, we invert the absolute value $(a_n \dots a_0)_b$ into $(a'_n \dots a'_0)_b + 1$ where $a'_i = (b-1) - a_i$. To get a positive from a negative, we do the same.

Example:

$b = 2, n = 4 \ : \ 5_{10} = 0101_2, \ \ -5_{10} = 1011_2.$

- Positive numbers and 0 have the most significant bit set to 0
- Negative numbers have the most significant bit set to 1
- There is only a single representation for 0

Floating Point Numbers

b is the basis.

A floating point number with precision p is represented as $s \ \cdot \ \overline{d_0.d_1 \dots d_{p-1}} \ \cdot \ b^e = s \cdot (\sum_0^{p-1} d_i b^{-i}) \cdot b^e.$

p is precision.

e is the exponent.

$d_0, d_1, \dots, d_{p-1}$ are digits of mantissa. $d_i \in \{0, \dots, b-1\}$. d_0 is usually in range $\{1, \dots, b-1\}$, except when the number is zero.

$s \in \{-1, 1\}$ is the sign.

Single precision format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|   exponent    |         mantissa (23 bits)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Double precision format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| exponent (11 bits)  |                                       :
+-+-+-+-+-+-+-+-+-+-+-+-+         mantissa (52 bits)            :
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Quadruple precision format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|     exponent (15 bits)      |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+                               |
:                                      mantissa (112 bits)      |
:                                                               |
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Here, $s = 0$ if the number is positive, 1 otherwise.

Converting a fraction

```
String decToBin(int f) {
    string s;
    while (f > 0) {
        f *= 2;
        int integer_part = int(f);
        s += integer_part.to_string();
        f -= int(f);
    }
    return s;
}

int binToDec(String s) {
    int f = 0;
    while (!s.isEmpty()) {
        b = s.pop().toInt();
```

```
        f = (f + b) / 2;
    }
    return f;
}
```

Example

We have a number -114.625 with base 10.

First, convert $114_{10} \to 1110010_2$ in a usual way.

Then, convert the fraction $0.625 \to 0.101_2$ (using the algorithm above).

Thus, we have an absolute value $01110010.101_2 = 1.110010101 \cdot 2^6$.

The given number is negative, so the first digit is 1.

Exponent is equal to $6 + 127(exponent\ bias\ for\ single\ precision) = 133_{10} = 10000101_2$.

The fraction is equal to 110010101_2.

The result is $1\underbrace{10000101}_{exponent}\underbrace{11001010100000000000000}_{fraction}$.

Total length is 32 bits.

ASCII encoding

Traditionally, ASCII encodes 128 specified characters into seven-bit natural numbers. Extended ASCII encodes the 128 specified characters into eight-bit natural numbers.

Source for decoding: https://graphemica.com/