Algebra of Programming
Practical Assessment
LEI — 2022/23

Department of Informatics
University of Minho

December 2022

Group nr.	99 (fill in)
a11111	Name1 (fill in)
a22222	Name2 (fill in)
a33333	Name3 (fill in)
a44444	Name4 (fill in, if applicable)

Preamble

Algebra of Programming's main objective is to teach computer programming as a scientific discipline. For that one starts from a repertoire of combinators that form an algebra of programming (set of universal laws and their corollaries) and use these combinators to build programs compositionally, that is, adding existing programs.

In the pedagogical sequence of the study plans of the courses that have this discipline, we chose to apply this method to programming in Haskell (without prejudice to its application to other functional languages). Thus, the present practical assessment places the students facing concrete problems that should be implemented in Haskell. There is yet another objective: teach how to document programs, to validate them and to produce quality technical-scientific texts.

Before addressing the problems proposed in the assessment, the groups should read carefully the attachment A where you will find the instructions concerning the software to be installed, etc.

Problem 1

Suppose a numerical sequence similar to the Fibonacci sequence such that each term subsequent to the first three corresponds to the sum of the three previous terms, subject to the coefficients a, b and c:

$$
\begin{array}{l}
f\ a\ b\ c\ 0 = 0 \\
f\ a\ b\ c\ 1 = 1 \\
f\ a\ b\ c\ 2 = 1 \\
f\ a\ b\ c\ (n+3) = a * f\ a\ b\ c\ (n+2) + b * f\ a\ b\ c\ (n+1) + c * f\ a\ b\ c\ n
\end{array}
$$

So, for example, $f\ 1\ 1\ 1$ will result in the following:

$$1, 1, 2, 4, 7, 13, 24, 44, 81, 149, \ldots$$

$f\ 1\ 2\ 3$ will generate the sequence:

$$1, 1, 3, 8, 17, 42, 100, 235, 561, 1331, \ldots$$

etc.

The given definition of f is very inefficient, having an exponential execution time degradation. One intends to optimize the given function by converting it to a for loop. Using the mutual recursion law, compute *loop* and *initial* in

$$fbl\ a\ b\ c = wrap \cdot \mathsf{for}\ (loop\ a\ b\ c)\ initial$$

in order for f and *fbl* to be (mathematically) the same function. To do so, you can use the rule of thumb explained in the appendix B.

Valuation: present performance tests that show how much faster is *fbl* when compared to f.

Problem 2

It is intended to classify the syllabus contents of all UCs (Curricular Units) taught at the Department of Informatics according to the ACM Computing Classification System. The taxonomy listing of that system is available in the file `Cp2223data`, starting with

```
acm_ccs = ["CCS",
           "    General and reference",
           "        Document types",
           "            Surveys and overviews",
           "            Reference works",
           "            General conference proceedings",
           "            Biographies",
           "            General literature",
           "            Computing standards, RFCs and guidelines",
           "        Cross-computing tools and techniques",
```

(first 10 items) etc., etc.[1]

It is intended to represent the same information in the form of an expression tree, using for this the library Exp which is part of the discipline's pedagogical material and which is included in the project zip, as it is more convenient for students.

1. Start by defining the function that converts the text given in *acm_ccs* (a list of strings) into such a tree as an Expanamorphism:

   $$tax :: [String] \to Exp\ String\ String$$
   $$tax = [\![(gene)]\!]_{\mathsf{Exp}}$$

   That is, define the *gene* of the anamorphism, taking into account the following diagram[2]:

   $$\begin{tikzcd}
   Exp\ S\ S & & S + S \times (Exp\ S\ S)^* \arrow[ll, "\mathsf{in}_{\mathsf{Exp}}"'] \\
   S^* \arrow[u, "tax"] \arrow[r, "out"] \arrow[rr, bend right, "gene"'] & S + S \times S^* \arrow[r, "\cdots"] & S + S \times (S^*)^* \arrow[u, "id + id \times tax^*"']
   \end{tikzcd}$$

   To do this, keep in mind that each level of the hierarchy is, in *acm_ccs*, marked by indentation of 4 additional spaces — as shown in the fragment above.

   Figure 1 shows the graphical representation of the Exp type tree that represents the *acm_ccs* shown above.

[1] Information retrieved from the site ACM CCS by selecting Flat View.
[2] S abbreviates *String*.

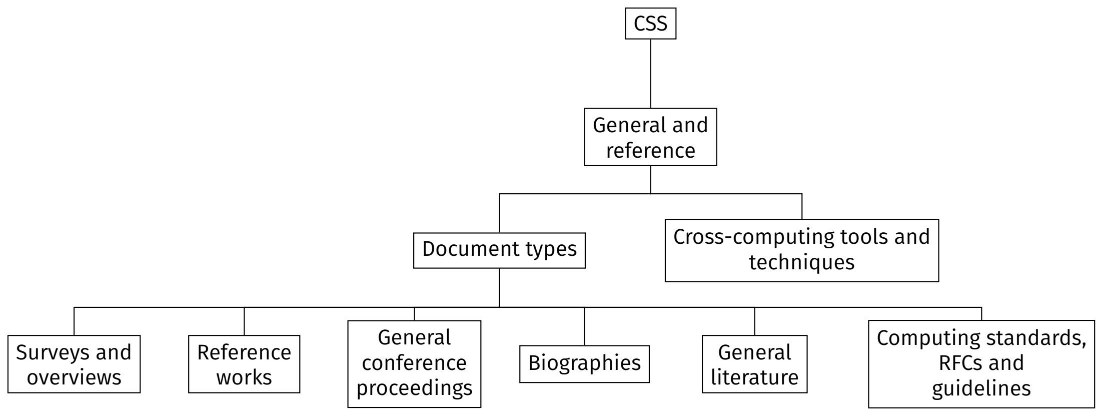

CSS
General and reference
Document types
Cross-computing tools and techniques
Surveys and overviews
Reference works
General conference proceedings
Biographies
General literature
Computing standards, RFCs and guidelines

Figure 1: Fragment of acm_ccs represented in the form of an Exptree.

2. Next we want all the paths of the tree that is generated by tax, since the classification of an UC can be done at any level (i.e. path root descendant `"CCS"` down to a sublevel or sheet). [3]

   We therefore need the composition of tax with a post-processing function $post$,

   $$tudo :: [String] \to [[String]]$$
   $$tudo = post \cdot tax$$

   to get the effect shown in the 1 table.

CCS			
CCS	General and reference		
CCS	General and reference	Document types	
CCS	General and reference	Document types	Surveys and overviews
CCS	General and reference	Document types	Reference works
CCS	General and reference	Document types	General conference proceedings
CCS	General and reference	Document types	Biographies
CCS	General and reference	Document types	General literature
CCS	General and reference	Cross-computing tools and techniques	

Table 1: ACM taxonomy closed by prefixes (first 10 items).

   Define the function $post :: Exp\ String\ String \to [[String]]$ in the cheapest way you can find.

Hint: Inspect the provided libraries for helper functions that you can reuse to make your solution simpler. Don't forget that, for the same result, in this discipline "wins" whoever writes less code!

Suggestion: For intermediate testing purposes, do not use the entirety of acm_ccs, which has 2114 lines! Use, for example, $take\ 10\ acm\_ccs$, as shown above.

Problem 3

The Sierpinski carpet is a fractal geometric figure in which a square is recursively subdivided into sub-squares. The classic construction of the Sierpinski carpet is as follows: assuming a square of side l, this one is subdivided into 9 equal squares of side $l/3$, removing the central square. This step is then repeated successively for each of the remaining 8 sub-squares (Fig. 2).

NB: In the example depicted in fig. 2, assuming the aforementioned classic construction, the squares are white and the background is green.

The complexity of this algorithm, depending on the number of squares to draw, for a depth n, is 8^n (exponential). However, if we assume that the squares to be drawn are the ones in green, the com-

[3]For a concrete UC classification example, please see section **ACM Classification** on Algebra of Programmingpublic page.

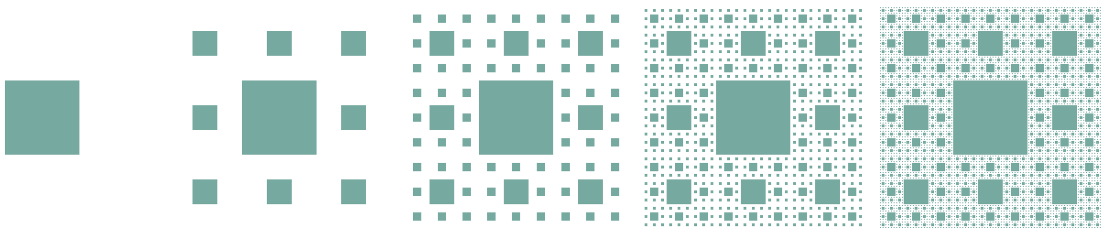


Figure 2: Construção do tapete de Sierpinski com profundidade 5.

plexity is reduced to $\sum_{i=0}^{n-1} 8^i$, obtaining a gain of $\sum_{i=1}^{n} \frac{100}{8^i}\%$. For example, for $n = 5$, the gain is 14.28%. The objective of this problem is the implementation of the algorithm through this optimization.

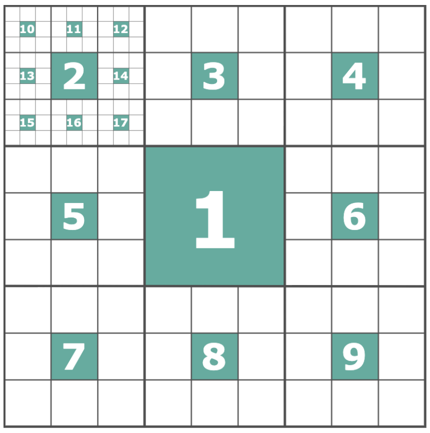


Figure 3: Sierpinski carpet with depth of 2 and numbered squares.

Thus, let each square be geometrically described by the coordinates of its lower left vertex and the length of its side:

$$\textbf{type}\ \mathit{Square} = (\mathit{Point}, \mathit{Side})$$
$$\textbf{type}\ \mathit{Side} = \mathit{Double}$$
$$\textbf{type}\ \mathit{Point} = (\mathit{Double}, \mathit{Double})$$

The recursive structure supporting the construction of Sierpinski carpets will be a Rose Tree, in which each level of the tree will store squares of equal size. For example, the construction of fig. 3 may [4] correspond to the tree in figure 4.

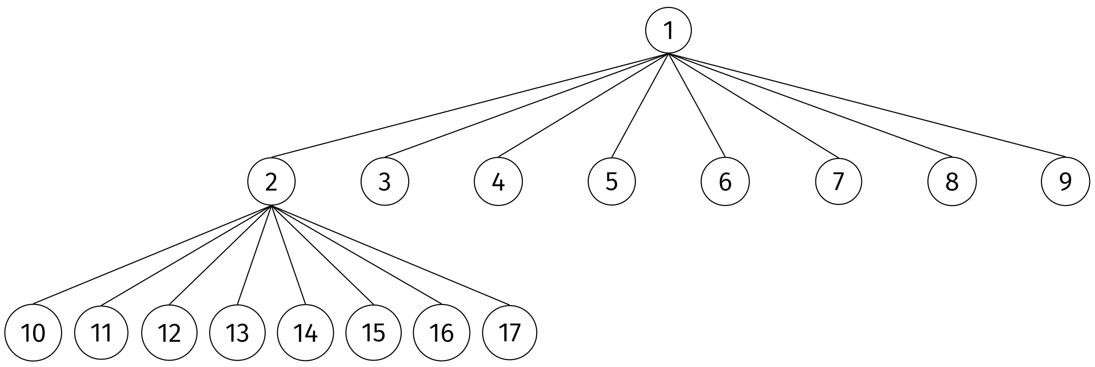


Figure 4: Possible support tree for building fig. 3.

Since the carpet itself is also a square, the objective will be, from the carpets's information (its lower left vertex coordinates and its side length), to draw the central square, subdivide the carpet into the 8 remaining sub-carpets, and recursively draw the square on these 8 sub-carpets. In this way, each

[4]The order of children is not relevant.

carpet determines its square and its 8 sub-carpets. In the example above, the carpet containing square 1 determines that square itself and determines the sub-carpets containing squares 2 to 9.

Therefore, in a first phase, given the carpet information, the support tree is built with all the squares to be drawn, for a given depth.

$$squares :: (Square, Int) \to Rose\ Square$$

NB: In the program, the depth starts at 0 and not at 1.

Once the tree with all the squares to be drawn has been generated, it is necessary to extract the squares into a list, which is processed by the function $drawSq$, available in the attachment D.

$$rose2List :: Rose\ a \to [a]$$

Thus, the construction of Sierpinski carpets is given by a hylomorphism of Rose Trees:

$$sierpinski :: (Square, Int) \to [Square]$$
$$sierpinski = [\![gr2l, gsq]\!]_{R}$$

Work to do:

1. Define the genes of the $sierpinski$ hylomorphism.
2. Run

$$
\begin{array}{l}
sierp4 = drawSq\ (sierpinski\ (((0,0),32),3)) \\
constructSierp5 = \mathbf{do}\ drawSq\ (sierpinski\ (((0,0),32),0)) \\
\quad await \\
\quad drawSq\ (sierpinski\ (((0,0),32),1)) \\
\quad await \\
\quad drawSq\ (sierpinski\ (((0,0),32),2)) \\
\quad await \\
\quad drawSq\ (sierpinski\ (((0,0),32),3)) \\
\quad await \\
\quad drawSq\ (sierpinski\ (((0,0),32),4)) \\
\quad await
\end{array}
$$

3. Define the function that presents the construction of the Sierpinski carpet as presented in $construcaoSierp5$, but for a depth $n \in \mathbb{N}$ received as a parameter.

$$
\begin{array}{l}
constructSierp :: Int \to \mathsf{IO}\ [()] \\
constructSierp = present \cdot carpets
\end{array}
$$

Hint: the function $constructSierp$ will be a lists hylomorphism, whose anamorphism $carpets :: Int \to [[Square]]$ constructs, receiving as a parameter the depth n, the list with all the carpets of depth $1..n$, and the catamorphism $present :: [[Square]] \to \mathsf{IO}\ [()]$ goes through the list drawing the carpets and waiting 1 second of interval.

Problem 4

This year, the 22nd Football World Cup, organized by the International Football Federation (FIFA), will take place, in Qatar, with the opening match on November 20th.

A bookmaker wishes to calculate, based on an approximation of the teams' rankings[5], the probability of each team winning the competition. For this, the director of the bookmaker hired the Department of Informatics of the University of Minho, which assigned the project to a team involving students and teachers of the Algebra of Programming course.

[5]The rankings obtained here have been scaled and rounded.

To solve this problem in a simple way, it will be approached in two phases:

1. academic version without probabilities, in which the winner of every game is known;
2. realistic version with probabilities using the Dist monad (probabilistic distributions) which is described in the appendix C.

The simpler first version should help build the second.

Problem Description

Once qualification is guaranteed (which has already occurred), the championship consists of two consecutive phases:

1. group stage;
2. knockout phase (or "mata-mata", as it is usually said in Brazil).

For the group stage, a draw of the 32 teams is made (which has already taken place for this competition) which places them in 8 groups, 4 selections in each group. Thus, each group is a list of selections.

The groups for this year's championship are:

```
type Team = String
type Group = [Team]
groups :: [Group]
groups = [["Qatar", "Ecuador", "Senegal", "Netherlands"],
  ["England", "Iran", "USA", "Wales"],
  ["Argentina", "Saudi Arabia", "Mexico", "Poland"],
  ["France", "Denmark", "Tunisia", "Australia"],
  ["Spain", "Germany", "Japan", "Costa Rica"],
  ["Belgium", "Canada", "Morocco", "Croatia"],
  ["Brazil", "Serbia", "Switzerland", "Cameroon"],
  ["Portugal", "Ghana", "Uruguay", "Korea Republic"]]
```

Thus, groups !! 0 matches group A, groups !! 1 to group B, and so on. In this phase, each selection of each group will face (once) the others of its group.

The two teams that score the most in each group go on to the "knockout", obtaining points, for each game of the group stage, as follows:

- win — 3 points;
- draw — 1 point;
- defeat — 0 points.

As mentioned, the final position in the group will determine whether a selection advances to the "knockout" and, if it does, what possible games will be ahead, since the disposition of the selections is defined from the beginning for this last phase, as can be seen in figure 5.

Thus, it is necessary to calculate the group winners under a probabilistic distribution. Once the distributions of the winners have been calculated, it is necessary to place them on the sheets of an LTree in order to make a match with the figure 5, thus entering the final phase of the competition, the long awaited "mata-mata". To advance in this final phase of the competition (i.e. climb the tree), you must win, whoever loses is automatically eliminated ("mata-mata"). When a team wins a game, it climbs the tree, when it loses, it stays on the way. This means that the winning selection is the one that wins all "knockout" games.

Proposed architecture

The team's compositional vision allowed it to realize from the outset that the Problem could be divided, regardless of the version, probabilistic or not, into two independent parts — the group stage and the

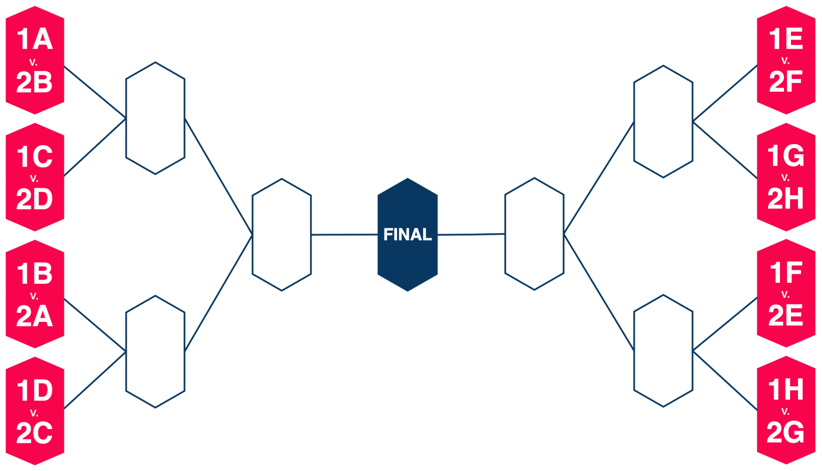


Figure 5: The “mata-mata”

“knockout”. Thus, two sub-teams could work in parallel, as long as the compositionality of the parts was guaranteed. It was decided that the students would develop the part of the group phase and the teachers the “match-and-match” part.

Non-probabilistic version

The final (non-probabilistic) result is given by the following function:

$$\begin{aligned}&winner :: Team\\&winner = wcup\ groups\\&wcup = knockoutStage \cdot groupStage\end{aligned}$$

The sub-team of teachers have already delivered their part:

$$knockoutStage = (\!|\,[id, koCriteria]\,|\!)$$

Now consider the proposal of the team leader of the students’ sub-team for the development of the group phase:

Let’s divide the process into 3 parts:

- generate the games,
- simulate the games,
- prepare the “matchback” by generating the game tree for that phase (fig. 5).

So:

$$\begin{aligned}&groupStage :: [Group] \to \mathsf{LTree}\ Team\\&groupStage = initKnockoutStage \cdot simulateGroupStage \cdot genGroupStageMatches\end{aligned}$$

Let’s start by defining the function $genGroupStageMatches$ which generates the group stage games:

$$\begin{aligned}&genGroupStageMatches :: [Group] \to [[Match]]\\&genGroupStageMatches = \mathsf{map}\ \ generateMatches\end{aligned}$$

where

$$\mathbf{type}\ Match = (Team, Team)$$

Now, we know that we were given the function

$$gsCriteria :: Match \to Maybe\ Team$$

which, based on a certain criterion, calculates the result of a game, returning $Nothing$ in case of a tie, or the winning team (under the constructor $Just$). So we need to define the function

$$\begin{array}{l} simulateGroupStage :: [[Match]] \to [[Team]] \\ simulateGroupStage = \mathsf{map}\ \ (groupWinners\ gsCriteria) \end{array}$$

which simulates the group stage and results in the list of winners, using the $groupWinners$ function:

$$groupWinners\ criteria = best\ 2 \cdot consolidate \cdot ({\gg\!=} matchResult\ criteria)$$

Only the definition of the $matchResult$ function is missing here.
Finally, we will have the function $initKnockoutStage$ which will produce the LTree that the "match-back" sub-team needs, with the proper positions. This will be the composition of two functions:

$$initKnockoutStage = [\![\, glt \,]\!] \cdot arrangement$$

Work to do:

1. Define an alternative to the generic function $consolidate$ let it be a list catamorphism:

   $$\begin{array}{l} consolidate' :: (Eq\ a, Num\ b) \Rightarrow [(a,b)] \to [(a,b)] \\ consolidate' = (\!|\, cgene \,|\!) \end{array}$$

2. Define function $matchResult :: (Match \to Maybe\ Team) \to Match \to [(Team, Int)]$ which calculates the points of the teams in a given game.
3. Define generic function $pairup :: Eq\ b \Rightarrow [b] \to [(b,b)]$ on what $generateMatches$ is based.
4. Define the glt gene.

Probabilistic version

In this more realistic version, $gsCriteria :: Match \to (Maybe\ Team)$ gives way to

$$pgsCriteria :: Match \to \mathsf{Dist}\ (Maybe\ Team)$$

which gives, for each game, the probability that each team will win or there will be a tie. For example, there is 50% of odds for Portugal to draw with England,

```
pgsCriteria("Portugal","England")
        Nothing  50.0%
 Just "England"  26.7%
Just "Portugal"  23.3%
```

etc.

What is Dist? It is the monad that deals with probability distributions and that is described in attachment C, page 11 and so on. What is there to do? Here's what your team leader says:

What needs to be done in this version is, first of all, to assess the impact from $gsCriteria$ turn monadic (on Dist) in overall version architecture previous. This impact must be reduced to a minimum by writing as little code as possible. as possible!

Everyone remembered something they had learned in theoretical classes about "monadification" of the code: you have to reuse the code from the previous version, monadifying it.

To distinguish the two versions it was decided to affix the 'p' prefix to identify a function that became monadic.

The sub-team of teachers meanwhile made the monadification of its part:

$$\begin{array}{l}
pwinner :: \mathrm{Dist}\ Team \\
pwinner = pwcup\ groups \\
pwcup = pknockoutStage \bullet pgroupStage
\end{array}$$

And he also delivered the probabilistic version of the "match-the-match":

$$\begin{array}{l}
pknockoutStage = mcataLTree'\ [return, pkoCriteria] \\
mcataLTree'\ g = k\ \textbf{where} \\
\quad k\ (Leaf\ a) = g1\ a \\
\quad k\ (Fork\ (x,y)) = mmbin\ g2\ (k\ x, k\ y) \\
\quad g1 = g \cdot i_1 \\
\quad g2 = g \cdot i_2
\end{array}$$

The sub-team of students has also already advanced work,

$$pgroupStage = pinitKnockoutStage \bullet psimulateGroupStage \cdot genGroupStageMatches$$

but still missing *pinitKnockoutStage* and *pgroupWinners*, is used in *psimulateGroupStage*, which is given in annex.

Work to do:

- Define functions that are not yet implemented in this version.
- **Valuation**: experiment with other team "ranking" criteria.

Important: (a) additional code will have to be placed in the attachment **??**, obligatorily; (b) all the code that is given cannot be changed.

Attachments

A How to carry out this assignment

In order to fulfill the aims of the assignment in an integrated way, we will use to a so-called programming technique "literary" [2], whose basic principle is the following:

> A program and its documentation must match.

In other words, the source code and documentation of a program must be in the same file.

The `cp2223t.pdf` file you are reading is already an example of literary programming: was generated from the source text `cp2223t.lhs`[6] which you will find in Pedagogical Material of this discipline by unzipping the file `cp2223t.zip` and running:

```
$ lhs2TeX cp2223t.lhs > cp2223t.tex
$ pdflatex cp2223t
```

where `lhs2tex` is a preprocessor that does "pretty printing" of Haskell code in LaTeX and that you should already install using the cabal utility available at haskell.org.

On the other hand, the same file `cp2223t.lhs` is executable and contains the basic "kit", written in Haskell, to get the job done. Just run

```
$ ghci cp2223t.lhs
```

Open the `cp2223t.lhs` file in your preferred text editor and check that it is: all the text that is inside the environment

[6]The suffix 'lhs' means literate Haskell.

```
\begin{code}
...
\end{code}
```

is selected by GHCi to be executed.

A.1 How to do the work

This theoretical-practical work must be carried out by groups of 3 (or 4) students. The details of the assessment (dates for submission of the report and its defense oral) are those published on the discipline page on the internet.

A participatory approach of group members is recommended in all work exercises, so that able to respond to any question posed in the oral defense of the report.

What, then, is the report referred to in the previous paragraph? It is editing the text being read, filling in the attachment E with the answers. The report must also contain the identification of the members of the working group, in the respective place on the cover page.

To generate the full PDF of the report, you must still run the following commands, that update the bibliography (with BibTEX) and the index (with `makeindex`),

```
$ bibtex cp2223t.aux
$ makeindex cp2223t.idx
```

and recompile the text as indicated above.

In the attachment D, some Haskell code for the presented Problems. This annex should be consulted and analyzed as necessary.

A.2 How to express calculations and diagrams in LaTeX/lhs2tex

As a first example, study the source text of this work to obtain the effect:[7]

$$
\begin{array}{ll}
& id = \langle f, g \rangle \\
\equiv & \qquad \{ \text{ universal property } \} \\
& \left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right. \\
\equiv & \qquad \{ \text{ identity } \} \\
& \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \\
\Box &
\end{array}
$$

Diagrams can be produced using the package LATEX xymatrix, for example:

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "(\!|g|\!)"'] & 1 + \mathbb{N}_0 \arrow[l, "\text{in}"'] \arrow[d, "id + (\!|g|\!)"] \\
B & 1 + B \arrow[l, "g"]
\end{tikzcd}$$

[7]Examples taken from [3].

B Rule of thumb for mutual recursion in $\mathbb{N}_0$

In this course we studied how to do dynamic programming by calculus, resorting to the law of mutual recursion.[8]

For the case of functions over natural numbers ($\mathbb{N}_0$, with functor $\mathsf{F}\ X = 1 + X$) it is easy to derive from the law that was studied once pocket rule that you can teach programmers who haven't studied Program Calculation. This rule is presented below, taking as example the calculation of the for-cycle that implements the Fibonacci function, remember the system:

$$\begin{array}{l} \mathit{fib}\ 0 = 1 \\ \mathit{fib}\ (n+1) = f\ n \end{array}$$

$$\begin{array}{l} f\ 0 = 1 \\ f\ (n+1) = \mathit{fib}\ n + f\ n \end{array}$$

You will get it right away

$$\begin{array}{l} \mathit{fib}' = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\ \quad \mathit{loop}\ (\mathit{fib}, f) = (f, \mathit{fib} + f) \\ \quad \mathit{init} = (1, 1) \end{array}$$

using the following rules:

- The body of the loop loop will have as many arguments as the number of functions mutually recursive.
- For the variables, choose the names of the functions, in order as you see fit.[9]
- The respective expressions are searched for the results, removing the variable n.
- In init the results of the base cases of the functions are collected, in the same order.

One more example, involving quadratic polynomials $ax^2 + bx + c$ in $\mathbb{N}_0$. Following the method studied in the classes[10], from $f\ x = ax^2 + bx + c$ two mutually recursive functions are derived:

$$\begin{array}{l} f\ 0 = c \\ f\ (n+1) = f\ n + k\ n \end{array}$$

$$\begin{array}{l} k\ 0 = a + b \\ k\ (n+1) = k\ n + 2\ a \end{array}$$

Following the above rule, the following implementation is immediately calculated in Haskell:

$$\begin{array}{l} f'\ a\ b\ c = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\ \quad \mathit{loop}\ (f, k) = (f + k, k + 2 * a) \\ \quad \mathit{init} = (c, a + b) \end{array}$$

C The monad of probability distributions

Monads are functors with additional properties that allow us to obtain special effects in programming. For example, the Probability library offers a monad to approach Probability Problems. In this library, the concept of statistical distribution is captured by the type

$$\mathbf{newtype}\ \mathsf{Dist}\ a = D\ \{\mathit{unD} :: [(a, \mathit{ProbRep})]\} \qquad (1)$$

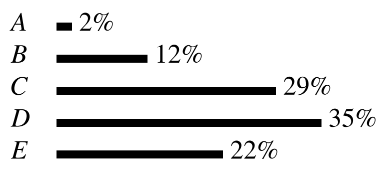
where $\mathit{ProbRep}$ is a real of 0 to 1, equivalent to a scale from 0 to 100%.

[8] Law (3.95) in [3], page 112.

[9] Obviously other symbols can be used, but in a first reading it is useful to use such names.

[10] Section 3.17 of [3] and topic Mutual recursion in the videos to support the theoretical classes.

Each pair (a,p) on a distribution $d :: \text{Dist}\ a$ indicates that the probability of a is p, the property that all probabilities of d add up 100%. For example, the following distribution of grades by tier A to E,



will be represented by the distribution

$$\begin{array}{l} d_1 :: \text{Dist}\ \mathit{Char} \\ d_1 = D\ [(\texttt{'A'},0.02),(\texttt{'B'},0.12),(\texttt{'C'},0.29),(\texttt{'D'},0.35),(\texttt{'E'},0.22)] \end{array}$$

which GHCi will show thus:

```
'D'  35.0%
'C'  29.0%
'E'  22.0%
'B'  12.0%
'A'   2.0%
```

It is possible to define distribution generators, for example uniform distributions,

$$d_2 = \mathit{uniform}\ (\mathit{words}\ \texttt{"Uma frase de cinco palavras"})$$

that is

```
     "Uma"  20.0%
   "cinco"  20.0%
      "de"  20.0%
   "frase"  20.0%
"palavras"  20.0%
```

a normal distribution, eg.

$$d_3 = \mathit{normal}\ [10\,..\,20]$$

etc.[11] Dist forms a **monad** whose unit is $\mathit{return}\ a = D\ [(a,1)]$ and whose composition by Kleisli is (simplifying the notation)

$$(f \bullet g)\ a = [(y, q * p) \mid (x,p) \leftarrow g\ a, (y,q) \leftarrow f\ x]$$

where $g : A \to \text{Dist}\ B$ e $f : B \to \text{Dist}\ C$ are **monadic** functions that represent probabilistic computations.

This monad is suitable for solving probability and statistics problems using functional programming, in an elegant way and as a particular case of monadic programming.

D Given Code

Problem 1

Some tests to validate the solution found:

$$\mathit{test}\ a\ b\ c = \text{map}\ \ (\mathit{fbl}\ a\ b\ c)\ x \equiv \text{map}\ \ (f\ a\ b\ c)\ x\ \textbf{where}\ x = [1\,..\,20]$$

$$\begin{array}{l} \mathit{test1} = \mathit{test}\ 1\ 2\ 3 \\ \mathit{test2} = \mathit{test}\ (-2)\ 1\ 5 \end{array}$$

[11]For more details see the source code for Probability, which is an adaptation of PHP library ("Probabilistic Functional Programming"). For those who want to know more it is recommended to read the article [1].

Problem 2

Check: the Exp type tree generated by

$$acm\_tree = tax\ acm\_ccs$$

must check the following properties:

- $expDepth\ acm\_tree \equiv 7$ (tree depth);
- $length\ (expOps\ acm\_tree) \equiv 432$ (number of tree nodes);
- $length\ (expLeaves\ acm\_tree) \equiv 1682$ (number of leaves in the tree).[12]

The end result

$$acm\_xls = post\ acm\_tree$$

must not be smaller than the total number of nodes and leaves in the tree.

Problem 3

Function for viewing in SVG:

$$drawSq\ x = picd''\ [Svg.scale\ 0.44\ (0,0)\ (x \gg= sq2svg)]$$
$$sq2svg\ (p,l) = (color\ \texttt{"\#67AB9F"} \cdot polyg)\ [p, p\ .\!+ (0,l), p\ .\!+ (l,l), p\ .\!+ (l,0)]$$

For timing purposes:

$$await = threadDelay\ 1000000$$

Problem 4

Rankings:

```
rankings = [
  ("Argentina",4.8),
  ("Australia",4.0),
  ("Belgium",5.0),
  ("Brazil",5.0),
  ("Cameroon",4.0),
  ("Canada",4.0),
  ("Costa Rica",4.1),
  ("Croatia",4.4),
  ("Denmark",4.5),
  ("Ecuador",4.0),
  ("England",4.7),
  ("France",4.8),
  ("Germany",4.5),
  ("Ghana",3.8),
  ("Iran",4.2),
  ("Japan",4.2),
  ("Korea Republic",4.2),
  ("Mexico",4.5),
  ("Morocco",4.2),
  ("Netherlands",4.6),
  ("Poland",4.2),
```

[12] That is, the total number of nodes and leaves is 2114, the number of lines of the given text.

```
    ("Portugal",4.6),
    ("Qatar",3.9),
    ("Saudi Arabia",3.9),
    ("Senegal",4.3),
    ("Serbia",4.2),
    ("Spain",4.7),
    ("Switzerland",4.4),
    ("Tunisia",4.1),
    ("USA",4.4),
    ("Uruguay",4.5),
    ("Wales",4.3)]
```

Generation of group stage matches:

```
generateMatches = pairup
```

Preparation of the “mata-mata” tree:

```
arrangement = (≫=swapTeams) · chunksOf 4 where
  swapTeams [[a1,a2],[b1,b2],[c1,c2],[d1,d2]] = [a1,b2,c1,d2,b1,a2,d1,c2]
```

Proposed function to obtain the ranking of each team:

```
rank x = 4 ∗∗ (pap rankings x − 3.8)
```

Criteria for non-probabilistic simulation of group stage matches:

```
gsCriteria = s · ⟨id × id, rank × rank⟩ where
  s ((s1,s2),(r1,r2)) = let d = r1 − r2 in
    if d > 0.5 then Just s1
      else if d < −0.5 then Just s2
        else Nothing
```

Criteria for non-probabilistic simulation of knockout stage matches:

```
koCriteria = s · ⟨id × id, rank × rank⟩ where
  s ((s1,s2),(r1,r2)) = let d = r1 − r2 in
    if d ≡ 0 then s1
      else if d > 0 then s1 else s2
```

Criteria for probabilistic simulation of group stage matches:

```
pgsCriteria = s · ⟨id × id, rank × rank⟩ where
  s ((s1,s2),(r1,r2)) =
    if abs (r1 − r2) > 0.5 then fmap Just (pkoCriteria (s1,s2)) else f (s1,s2)
  f = D · ((Nothing,0.5):) · map (Just × (/2)) · unD · pkoCriteria
```

Criteria for probabilistic simulation of knockout stage matches:

```
pkoCriteria (e1,e2) = D [(e1,1 − r2 / (r1 + r2)),(e2,1 − r1 / (r1 + r2))] where
  r1 = rank e1
  r2 = rank e2
```

Probabilistic version of group stage simulation:[13]

```
psimulateGroupStage = trim · map (pgroupWinners pgsCriteria)
trim = top 5 · sequence · map (filterP · norm) where
  filterP (D x) = D [(a,p) | (a,p) ← x, p > 0.0001]
  top n = vec2Dist · take n · reverse · presort π2 · unD
  vec2Dist x = D [(a,n / t) | (a,n) ← x] where t = sum (map π2 x)
```

[13] Distributions are “trimmed” to reduce simulation time.

More efficient version of $pwinner$ given in the main text, to shorten the time for each simulation:

$$
\begin{aligned}
&pwinner :: \mathsf{Dist}\ Team\\
&pwinner = mbin\, f\ x \gg\!= pknockoutStage\ \mathbf{where}\\
&\quad f\ (x,y) = initKnockoutStage\ (x \mathbin{+\!\!+} y)\\
&\quad x = \langle g \cdot take\ 4, g \cdot drop\ 4\rangle\ groups\\
&\quad g = psimulateGroupStage \cdot genGroupStageMatches
\end{aligned}
$$

Auxiliaries:

$$
\begin{aligned}
&best\ n = \mathsf{map}\ \ \pi_1 \cdot take\ n \cdot reverse \cdot presort\ \pi_2\\
&consolidate :: (Num\ d, Eq\ d, Eq\ b) \Rightarrow [(b,d)] \rightarrow [(b,d)]\\
&consolidate = \mathsf{map}\ \ (id \times sum) \cdot collect\\
&collect :: (Eq\ a, Eq\ b) \Rightarrow [(a,b)] \rightarrow [(a,[b])]\\
&collect\ x = nub\ [k \mapsto [d' \mid (k',d') \leftarrow x, k' \equiv k] \mid (k,d) \leftarrow x]
\end{aligned}
$$

Monadic binary function f:

$$
\begin{aligned}
&mmbin :: Monad\ m \Rightarrow ((a,b) \rightarrow m\ c) \rightarrow (m\ a, m\ b) \rightarrow m\ c\\
&mmbin\, f\ (a,b) = \mathbf{do}\ \{x \leftarrow a; y \leftarrow b; f\ (x,y)\}
\end{aligned}
$$

Monadification of a binary function f:

$$
\begin{aligned}
&mbin :: Monad\ m \Rightarrow ((a,b) \rightarrow c) \rightarrow (m\ a, m\ b) \rightarrow m\ c\\
&mbin = mmbin \cdot (return \cdot)
\end{aligned}
$$

Other functions that may be useful:

$$
\begin{aligned}
&(f\ `is`\ v)\ x = (f\ x) \equiv v\\
&rcons\ (x,a) = x \mathbin{+\!\!+} [a]
\end{aligned}
$$

E Students' Solutions

Students should put their solutions to the exercises in this annex. proposed, according to the supplied "layout". They can't change the names or types of the given functions, but can be add text, diagrams and/or other auxiliary functions that are necessary.

We value writing little code that corresponds to solutions simple and elegant.

Problem 1

Helper functions required:

$$
\begin{aligned}
&loop = \bot\\
&initial = \bot\\
&wrap = \pi_2
\end{aligned}
$$

Problem 2

tax gene:

$$gene = \bot$$

Post processing function:

$$post = \bot$$

Problem 3

$$
\begin{array}{l}
\mathit{squares} = [\![\mathit{gsq}]\!]_{\mathsf{R}} \\
\mathit{gsq} = \bot \\
\mathit{rose2List} = (\!| \mathit{gr2l} |\!)_{\mathsf{R}} \\
\mathit{gr2l} = \bot \\
\mathit{carpets} = \bot \\
\mathit{present} = \bot
\end{array}
$$

Problem 4

Non-probabilistic version

Gene of $\mathit{consolidate}'$:

$$\mathit{cgene} = \bot$$

Generation of group stage games:

$$
\begin{array}{l}
\mathit{pairup} = \bot \\
\mathit{matchResult} = \bot \\
\mathit{glt} = \bot
\end{array}
$$

Probabilistic version

$$
\begin{array}{l}
\mathit{pinitKnockoutStage} = \bot \\
\mathit{pgroupWinners} :: (\mathit{Match} \rightarrow \mathsf{Dist}\ (\mathit{Maybe\ Team})) \rightarrow [\mathit{Match}] \rightarrow \mathsf{Dist}\ [\mathit{Team}] \\
\mathit{pgroupWinners} = \bot \\
\mathit{pmatchResult} = \bot
\end{array}
$$

Index

References

[1] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. J. Funct. Program., 16:21–34, January 2006.

[2] D.E. Knuth. Literate Programming. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[3] J.N. Oliveira. Program Design by Calculation, 2022. Textbook in preparation, 310 pages. Informatics Department, University of Minho. Current version: Sept. 2022.