COMP2212 Programming Language Concepts Coursework

Dr Julian Rathke

Semester 2 2022/23

Introduction

In this coursework you are required to *design and implement* a domain specific programming language for specifying tiling patterns. For the purposes of this assignment we consider a *tile* to be a square of discrete cells. A tile of size N consists of $N \times N$ cells. A cell may either be filled or empty. You will need to create a language of your own design that allows users to create larger tiles built from given sets of small tiles.

You are welcome to research any existing languages to inspire the design of your own language. If you want, you could stick closely to the syntax of an existing language. Of course, you are also more than welcome to go your own way and be as original and creative as you want: it is *YOUR* programming language, and your design decisions.

You are required to (1) invent an appropriate syntax and (2) write an interpreter (possibly using `Alex` and `Happy` for lexing and parsing). Your overall goal is :

> To design and implement a programming language for specifying large tiling patterns.

For the five example problems listed below, you will be required to produce a program, in your language, that solves each of the problems. These five programs, together with the Haskell sources for your interpreter, are the required deliverables for the first submission.

Please keep a record of all the sources you consult that influence your design, and include them in your programming language manual. The manual will be required for the second submission, along with programs for five additional unseen problems, which we will make public *after* the deadline for the first submission. You should anticipate that the additional problems will be comprised of variations and combinations of the first five problems. You will find more procedural details at the end of this document.

The specification is deliberately loose. If we haven't specified something, it is a design decision for you to make: e.g. how to handle syntax errors, illegal inputs, whether to allow comments, support for syntax highlighting, compile time warnings, any type systems etc. A significant part (50%) of the mark will be awarded for these qualitative aspects, where we will also take into account the design of the syntax and reward particularly creative and clean solutions with additional marks. The remaining 50% of the mark will be awarded for correctly solving a number of problems using your programming language. The correctness of your solution will be checked with a number of automated tests.

The Input and Output Format

For each problem we will declare the name of one or more input files in a simple text-based tile format. The input file name will correspond to a file in the current working directory with an extension ".tl". For example, if the problem input file is called "foo" and your interpreter is executed in directory "C:/Home/Users/jr/TSL/" then the input file will be found at "C:/Home/Users/jr/TSL/foo.tl".

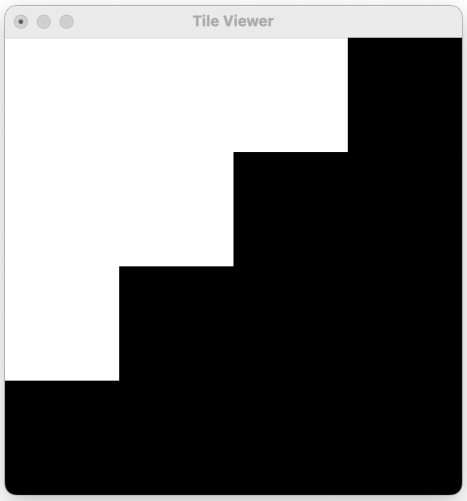
The input files will be in a simple (text-based) tile format as follows :

- each row of the tile will be given on a separate line of the file (separated by a newline character)
- each row will consist of a string of the characters '0' and '1' only where '1' represents a filled cell and '0' represents an empty cell.
- the length of the string in each row will be the same for every row.
- there will be the same number of rows as the length of each row.

For example, the input file

```
0001
0011
0111
1111
```

represents a tile that could be rendered as



You may assume in the stated problems that all input files will be well-formed in this simple format. A simple Haskell tile viewer program will be made available for you to check your outputs visually.

For each stated problem, the output should be in the same simple format. The output should not contain any other messages or formatting information and **the output should always be printed to Standard Out**. The output required will be specified as best we can and a visual representation of an example output will be provided in each case. In addition to this, an actual example output file in the simple tile format will also be provided for you to check your own output against.

Problems

For every input name `foo` used in a problem specification, you may assume that we will place a tile file called `foo.tl` in the same directory where we execute your interpreter. The file will always be compatible with the tile format given above. You may assume that we will not require you to perform any additional operations on the tile values other than those indicated by the problems given below and combinations of these.

Problem 1 - Checker Board

Assume two input tiles `tile1` and `tile2` of the same size. Output a single tile that comprises an alternating sequence of the two input tiles repeated in each row such that for the first row, reading from left to right `tile1` appears first, followed by `tile2`. For the second row, reading from left to right `tile2` appears first, followed by `tile1`. The output should contain 64 rows of alternating input tiles, each row consisting of 64 input tiles.

For example, if the two input tiles are both 1x1 tiles, the first consisting of a single filled cell the other a single empty cell then the output could be rendered as

Note that, it may be wise to consider implementing this using a form of iteration as the unseen problems may ask for much larger outputs.

Problem 2 - Rotation and Scaling

Assume one input tile `tile1`. Output a single tile that comprises a pattern built as follows:

Step 1 a "base" square is built by taking the input tile and constructing a square of itself and rotations of itself as follows : the top left cell of the square must be the input tile, the top right cell of the square must be the input tile rotated through 90 degress (around the centre of the input tile). The bottom left cell of the square must be the input tile rotated through 270 degrees and finally, the bottom right cell of the square must be the input tile rotated though 180 degrees.

For example, if the input tile is

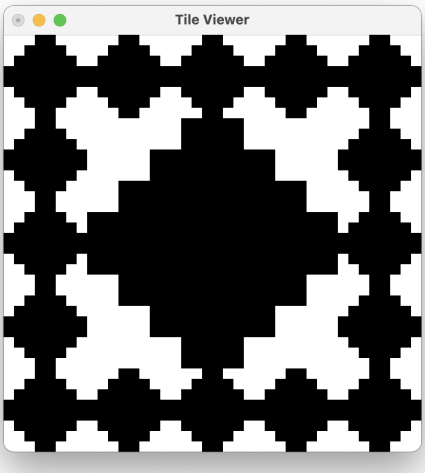
```
0001
0011
0111
1111
```

then the "base" square would be

```
00011000
00111100
01111110
11111111
11111111
01111110
00111100
00011000
```

Having built the base square the output required from Problem 2 is a single copy of the base square scaled to three times its original size surrounded by multiple copies of the base square.

For example, given the input tile as above the output could be rendered as



Problem 3 - Reflections and Blanks

Assume one input tile `tile1`. Output a single tile that is 60 times the size of the input tile that comprises a pattern built as follows:

Step 1 is to build a "base" object of 3x2 lots of the input tile and a constant blank tile. The base should be laid out as per the diagram below.

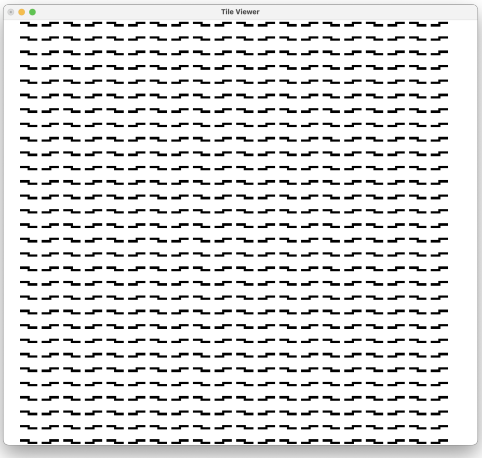
where Tile1 refers to the input tile, Tile1-Y refers to the input tile reflected across the middle vertical axis of the tile, Tile1-X refers to the input tile reflected across the middle horizontal axis of the tile and Tile-X-Y refers to the input tile reflected across both the horizontal and vertical axes. The tile named Blank refers to a constant empty tile of the appropriate size - this blank tile is not given as an input tile.

Having built the "base" object, the output required from Problem 3 is a copy of the base followed horizontally to the right by a copy of the whole base object reflected across the X-axis all repeated 10 times. This entire "row" of base objects is then repeated vertically downwards 30 times.

For example, given the input tile

```
000
000
011
```

the output tile could be rendered as



Problem 4 - Boolean Ops and Predicates

Assume three input tiles `tile1`, `tile2` and `tile3` all of the same size. In this problem we will be making use of boolean operations on tiles, by considering each cell as a boolean with '1' representing True and '0' representing False. We define the conjunction ("and") of two tiles as the tile resulting from the cell-wise conjunction of the two tiles. Similarly, we define the negation of a tile as the cell-wise negation of the tile. For example, the negation of

```
000
000
011
```

is

```
111
111
100
```

The output tile required in Problem 4 needs to be 50 times the size of the input tiles. Considering the input tiles as being unit size, then the output can be considered as having 50 rows and 50 columns, both numbered from 0 to 49.

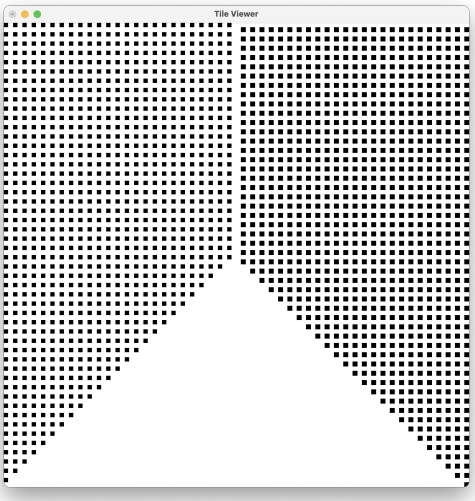
The output tile can be described as follows:

- for entries at (col, row) where $row + col < 50$ and $col < 25$ there should be the tile obtained as the conjunction of `tile1` and the negation of `tile3`.
- for entries at (col, row) where $row \leq col$ and $col \geq 25$ there should be the tile obtained as the conjunction of `tile2` and the negation of `tile3`.
- for all other entries there should be the blank tile of appropriate size.

For example, if the input tiles are

```
11    ,    00     and    01
00         11            10
```

respectively, then the output tile could be rendered as



Problem 5 - Subtiles

Assume one input file `tile1`. In this problem we are going to create subtiles of the input tile. We can specify a subtile by giving a (col,row) number of the input tile to be the top-left corner of the subtile. We then also specify how large a subtile to take. Of course, the subtile should be contained entirely within the given input tile.

For this problem, assume that the input size is a tile of size 10. Create three subtiles `subtile1`, `subtile2` and `subtile3` each of size 6 with top-left corners at (0,0), (2,2) and (4,4) of the input tile respectively.

Output a single tile comprising a row of three copies of `subtile1`, then a row of three copies of `subtile2` and finally a row of three copies of `subtile3`. The overall output tile should be size 18.

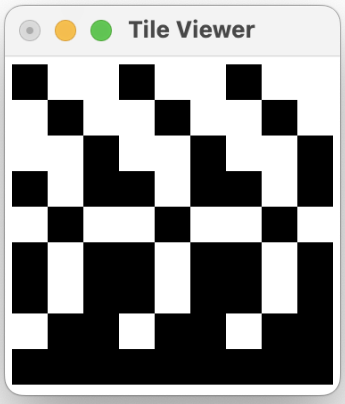
For example, if the input tile is

```
1100000011
1100000011
0011001111
0011001111
0000110011
0000110011
0011001111
0011001111
1111111111
1111111111
```

Then the three subtiles are

```
110000   ,   110011   and   110011
110000   ,   110011   and   110011
001100   ,   001100   and   001111
001100   ,   001100   and   001111
000011   ,   110011   and   111111
000011   ,   110011   and   111111
```

respectively. The overall output could be rendered as



First submission - due Thursday April 27th 4pm

You will be required to submit a zip file containing:

- the sources for the interpreter for your language, written in Haskell
- five programs, written in **YOUR** language, that solve the five problems specified above. The programs should be in files named `pr1.tsl`, `pr2.tsl`, `pr3.tsl`, `pr4.tsl`, `pr5.tsl`.

We will compile your interpreter using the command `ghc Tsl.hs` so you will need to include a file in your zip file called `Tsl.hs` that contains a main function along with any other Haskell source files required for compilation. Prior to submission, you are required to make sure that your interpreter compiles **on a Unix machine with a standard installation of GHC (Version 8.4.3) or earlier**: if your code does not compile then you will be awarded 0 marks for the functionality testing component of the assessment.

You can use Alex and Happy for lexing and parsing but make sure that you include the generated Haskell source files obtained by running the `alex` and `happy` commands as well as the Alex and Happy source files. Alternatively you can use any other Haskell compiler construction tools such as parser combinator libraries. You are welcome to use any other Haskell libraries, as long as this is clearly acknowledged and the external code is bundled with your submission, so that it can compile on a vanilla Haskell installation.

Interpreter spec. Your interpreter is to take a file name (the program in your language) as a single command line argument. The interpreter should produce output on standard output (`stdout`) and error messages on standard error (`stderr`). For each problem, we will test whether your code performs correctly by using a number of tests. We only care about correctness and performance will not be assessed (within reason - our marking scripts will timeout after a generous period of time). You can assume that for the tests we will use correctly formatted input. For example, when assessing your solution for Problem 1 we will run

```
./Tsl pr1.tsl
```

in a directory where we also provide our own versions of `tile1.tl` and `tile2.tl`. We will then compare the contents of `stdout` against our expected outputs. Whitespace and formatting is important and the output must adhere to the simple tile format and contain no other text.

Second submission - due Friday May 5th 4pm

Shortly after the first deadline we will release a further five problems. Although they will be different from Problems 1-5, you can assume that they will be closely related, and follow the same input/output conventions. You will be required to submit two separate files.

First, you will need to submit a zip file containing programs (`pr6.tsl`, `pr7.tsl`, `pr8.tsl`, `pr9.tsl`, `pr10.tsl`) written in your language that solve the additional problems. We will run our tests on your solutions **to all ten problems** and award marks for solving the additional problems correctly.

Second, you will be required to submit a 5 page report on your language **in pdf format** that explains the main language features, its syntax, including any scoping and lexical rules as well as additional features such as syntax sugar for programmer convenience, type checking, informative error messages, etc. In addition, the report should explain the execution model for the interpreter, e.g. what the states of the runtime are comprised of and how they are transformed during execution. Languages that support strong static typing and type safety with a formal specification are preferred. This report, together with the five programs will be evaluated qualitatively and your marks will be awarded for the elegance and flexibility of your solution and the clarity of the report.

Please note: **there is only a short period between the first and second submission**. I strongly advise preparing the report well in advance throughout the development of the coursework.

As you know, the coursework is to be done in groups of three. Only one submission per group is required. I don't need to know who is in which group at the first submission but as as part of the **second** submission we will require a declaration of who is in your group and how marks are to be distributed amongst the members of your group. You will receive all feedback and your marks by Friday May 26th. **Please ensure that it is the same group member that submits for both the first and second submissions**.

Marks. This coursework counts for 40% of the total assessment for COMP2212. There are a total of 40 marks available. These are distributed between the two submissions as follows:

You will receive the test results of Submission One prior to the second deadline but no marks will be awarded until after Submission Two.

After Submission Two we will award up to 20 marks for the qualitative aspects of your solution, as described in your programming language report. We will also award up to 20 marks for your solutions to the ten problems. For each problem there will be 2 marks available for functional correctness only.

You have the option of resubmitting the interpreter after receiving your testing results from Submission One. This will however incur a 50% penalty on the marks for functional correctness of all ten problems. Therefore, if you decide to resubmit your interpreter in the second submission the maximum possible total coursework mark is capped at 30 marks (20 for the report plus 10 for functional correctness).

Any late submission to either component will be treated as a late submission overall and will be subject to the standard university penalty of 10% per working day late.