University of Southampton

Lucian Serbanescu (ljs1u21@soton.ac.uk)
Renata Kim (rk12g21@soton.ac.uk)
Ahmet Binatlioglu (acb2g21@soton.ac.uk

COMP2212 Coursework

1 Main language features

The developed language is named **"Hasktile"**. Haskile is a Haskell-based language configured to perform manipulations on tiles. With a syntax and a paradigm inspired from Java and Haskell, our language manipulates tiles time efficient. The created features provide user with the ability to read tiles from file and perform the desired modifications to them. Using "print();" function you can display the tile in Standard Output.

1.1 Lexical rules, Syntax and Scoping

Lexing and Parsing are implemented using **Alex** and **Happy**, and are stored inside "HasktileTokens.x" and "HasktileGrammar.y" files.
Some lexing details would be that variables have to start with a letter and can contain letters and numbers, but no other charaters. All the functions have predefined tokens and you have to call them with capital letters.
Regading syntax we allow a few main actions : declare a variable giving the type, change the value stored by a variable, call a function, print a variable or even storing the output of a function in a variable.
Variable declaration using the output of a function example :

<varType> <varName> = <functionCall>;

We also included **for loops**, but they work in a haslkell-like way. We have some functions such as SCALE(), REPEAT-HORZ() ,SUBTILES(), SUBLIST() which cover the entire loops functionalities reagarding tile creation. The main reason why we stick with a loop function is time-efficiency. We added **if statements** so we can check the input tiles, or do some unregular changes in tiles.
We tried to keep the AST as simple as possible passing only variables name of funtions and parameters:

```
Parsed as SEQ (DECLARE_LIST_STRING "t1" (READ_FILE "tile1.tl")) (SEQ (DECLARE_LI
ST_STRING "t2" (READ_FILE "tile2.tl")) (SEQ (DECLARE_LIST_STRING "v1" (ADD_HORZ
"t1" "t2")) (SEQ (DECLARE_LIST_STRING "v2" (ADD_HORZ "t2" "t1")) (SEQ (DECLARE_L
IST_STRING "line1" (REPEAT_HORZ (PARAMETER_STRING "v1") (PARAMETER_INT 32))) (SE
Q (DECLARE_LIST_STRING "line2" (REPEAT_HORZ (PARAMETER_STRING "v2") (PARAMETER_I
NT 32))) (SEQ (DECLARE_LIST_STRING "v" (ADD_VERT "line1" "line2")) (LAST_ACTION_
CALLED (PRINT (REPEAT_VERT (PARAMETER_STRING "v") (PARAMETER_INT 32)))))))))))
```

We would say that we have only one scope (or none at all). We do not provide declarations of new functions, neither classes, so the scope is only one. Even the if statements do not have scope.

1.2 Syntax sugar for programmer convenience

Our language allows adding functions as parameters to other functions.
For instance, we have a function "PRINT", which can take either a variable (e.g.: "PRINT(var1)), or a function (e.g.: "PRINT(ADD_VERT(tile1, tile2))".

1.3 Type checking

Our coding language performs dynamic type checking. The types of variables and values are determined at runtime during program execution, rather than being declared and checked at compile time. If we use 'DECLARE LIST STRING' (Called when initializing a new tile variable) as an example, a function is given as an argument to evaluate the value assigned to the variable. We call the function we are assigning to the variable and if the function returns anything with a type other than a List of Strings we throw an error.
When we are calling functions that have variables as arguments we check the types of these arguments by pattern matching. In the case of the function 'REPEAT HORZ', the first argument must be 'StrList' type (which represents a tile) and the second argument must be an integer value. Otherwise, an error is thrown:

```
Paramaters are not in the correct Type (StrList, Integer) 9:25
```

2 Interpreter

The interpreter is written in Haskell and stored inside **"Tsl.hs"** file. It also uses additional Haskell functions stored inside **"Functions.hs"** file, which acts as a library.
The interpreter is designed to take a file name as a single command line argument.
Once Tsl.hs reads and parses the file, the "evalProg" function evaluates the Abstract Syntax Tree by traversing it. It maintains a map that tracks all the variables stored in memory.
A list of tokens is also maintained to keep track of which line we are on in the original .tl file which is being processed. This list is stored so that if there happens to be an error in the source code we will be able to display to the user information about which line caused the error.
Finally, it prints the appropriate output to Standard Output, or an error message to Standard Error.
A large focus was set on making the interpreter efficient. As the result, the interpreter works fast and efficiently (most programs written in the language, including solutions to the 10 given problems, are executed within less than a second).

2.1 Functions library

During execution, the interpreter uses functions available inside the Hasktile library.
The library is stored in a Haskell file called **"Functions.hs"** and includes all functions that can be required for tile manipulations.
All functions included in the library are designed to serve as much efficiency as possible. For instance, the "scale" function takes a number as input, allowing to scale a tile 2, 3, 10, 100, basically any number of times. Features like this provide the user with a wider range of possible tile manipulations.
The library is made up of various sorts of functions. These include the ones required by the initially given 5 problems: add tiles horizontally/vertically, rotate, scale, reflect, negate, conjoin, split into subtiles.
Additionally, the library includes Haskell code that allows to work with lists (add to list, get an element by its index, remove an element from list), Boolean operations on tiles (disjunction, implication, XOR), a function for checking whether 2 tiles are equal, a function for concatenation of 2 strings and a function for printing a tile to Standard Output.

3 Design decisions

3.1 Error Handling

There are various types of errors displayed.

3.1.1 Parsing errors

In case of a parsing error, the user will get an informative message printed and the program will not be executed. The message will state that it is a "Parsing error" and the line and column number to identify which line of the user's code caused the problem.

```
Parse error at line:column 7:1
```

3.1.2 Type checking errors

In each case of function calls or declarations of variables we check the arguments by pattern matching. If they are of wrong type we will be throwing errors indicating what type they should be and the line of code the error is in. Example:

```
Paramaters are not in the correct Type (StrList, Integer) 9:25
```

3.1.3 Library function errors

Additionally, the Hasktile library functions stated in the previous section validate inputs to make sure that invalid parameters are caught. For instance, the "SCALE" function, which takes a tile and a number as inputs does not allow negative numbers to be inserted.
Whenever a user inputs invalid parameters to a function, an informative error message is displayed. For example, if a user wanted to run CONJUNCTION function on 2 sizes of unequal tiles, he would receive an error:

```
Tiles of unequal size passed as input for CONJUNCTION function.
```

3.1.4 Compile time errors

The language has been designed in a way that if a program takes more than 60 seconds to execute, an error message is displayed. The timeout was set to only 1 minute, because the interpreter is quite fast. Most programs written in the language (including solutions to all 10 given problems) are executed within less than a second.
The message says:

```
Compilation timed out!
```

3.2 Illegal inputs

Whenever a tile is read from file, the tile is validated by the interpreter.
If the inserted tile is valid (consists of only 0s and 1s, and the number of rows is equal to the number of columns), then the program continues executing. Otherwise, an appropriate error message is displayed:

```
Tile is not in valid format 4:19
```

3.3 Comments

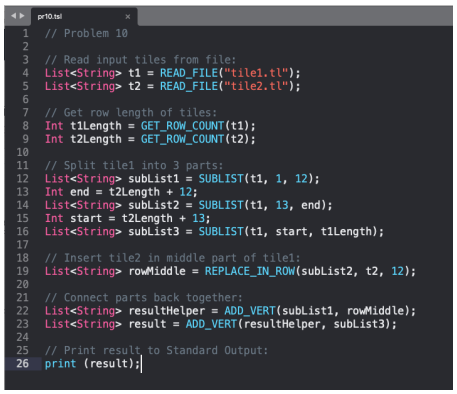
The HaskTile language allows for both Java-style and Haskell-style comment formats.
The inspiration for the formats was taken from Java and Haskell in order to achieve an intuitive user understanding.
Comment formats:

```
// This is a comment
```

```
-- This is also a comment
```

3.4 Syntax highlighting

Regarding the syntax highlighting, we approach only the Sublime code editor. We used YAML files which code are responsible for parsing and creating Sublime tokens. These tokens are later matched against a colour scheme file, where they receive a corresponding colour. What we did was: we copied the YAML file from GitHub (source: https://github.com/sublimehq/Packages/blob/master/Java/Java.sublime-syntax), and edited the file in a way that it matches our language syntax.



3.5 Type systems

Hasktile is a strongly typed language. Once a variable is defined with a specific data type, it can only store values of that type. This means that if you try to assign a value of a different type to that variable, the compiler will throw an error.

When a function is called the interpreter checks the types of variables that are being called. If the type of the variables is not consistent with what is being asked in the function, an error is thrown before the function is executed.

4 Cheat Sheet

HaskTile Cheat Sheet:

Comments

//	Java-style	// This is a Java-style comment.
--	Haskell-style	-- This is a Haskell-style comment.

Data types

String	Any sequence of characters (letters, numbers and symbols).
Int	Any whole number.
List<String>	Collection of String values. Typically used to represent a tile.
List<List<String>>	Collection of lists. Typically used to represent a list of subtiles.

Lists

Declaration	Creation of an empty list	List<String> list1 = [];
ADD_TO_LIST ([a], a): returns [a]	Function for adding a new element to list	List<String> added = ADD_TO_LIST(list1, str1);
REMOVE_FROM_LIST ([a], a): returns [a]	Function for removing an element from list	List<String> removed = REMOVE_FROM_LIST (list1, str1);
GET_BY_INDEX ([a], Int): returns a	Get an element from list by its index	String str = GET_BY_INDEX(list1, 2);
REVERSE_LIST ([a]): returns [a]	Reverse list	List<String> reversed = REVERSE_LIST(list1);

Comparison Operators

>	Greater than	if (num > 2) { print(tile1); };
<	Less than	if (num < 10) { print(tile1); };
>=	Greater than or equal to	if (num >= 2) { print(tile1); };
<=	Less than or equal to	if (num <= 10) { print(tile1); };

If statements

if (condition) { do_something; };	if (num > 3) { print(tile1); };
if (condition) { do_something; } else { do_something_else; **};**	if (num > 3) { print(tile1); } else { print(tile2); };

Mathematical Operators

+	Integer addition	Int num = 6 + 3;
-	Integer subtraction	Int num = 6 - 3;
*	Integer multiplication	Int num = 6 * 3;
/	Integer division	Int num = 6 / 3;

Strings

Declaration	Creation a String variable	String str = "010";
CONCAT_STRINGS (String, String): returns String	Function for concatenation of 2 String values	String conc = CONCAT_STRINGS (str1, sr2);

Functions for managing tiles		
READ_FILE (String): returns List<String>	Read a tile from file. The function validates the tile as well	List<String> tile1 = READ_FILE("tile1.tl);
GET_ROW_COUNT (List<String>): returns Int	Count the number of rows of a tile	Int count = GET_ROW_COUNT(tile1);
GET_COL_COUNT (List<String>): returns Int	Count the number of columns of a tile	Int count = GET_COL_COUNT(tile1);
ADD_HORZ (List<String>, List<String>): returns List<String>	Add 2 tiles horizontally	List<String> added = ADD_HORZ(tile1, tile2);
ADD_VERT (List<String>, List<String>): returns List<String>	Add 2 tiles vertically	List<String> added = ADD_VERT(tile1, tile2);
REPEAT_HORZ (List<String>, Int): returns List<String>	Repeat a tile N times horizontally	List<String> repeated = REPEAT_HORZ(tile1, N);
REPEAT_VERT (List<String>, Int): returns List<String>	Repeat a tile N times vertically	List<String> repeated = REPEAT_VERT(tile1, N);
ROTATE (List<String>): returns List<String>	Rotate a tile 90° clockwise	List<String> rotated = ROTATE (tile1);
SCALE (List<String>, Int): returns List<String>	Scale a tile by N times	List<String> scaled = SCALE (tile1, N);
REFLECT_VERT (List<String>): returns List<String>	Reflect a tile over y-axis	List<String> reflected = REFLECT_VERT (tile1);
REFLECT_HORZ (List<String>): returns List<String>	Reflect a tile over x-axis	List<String> reflected = REFLECT_HORZ (tile1);
MAKE_TILE (Char, N): returns List<String>	Create a tile of size NxN made filled with a certain character	List<String> newTile = MAKE_TILE (0, N);
NEGATION (List<String>): returns List<String>	Negate a tile using Boolean logic	List<String> negated = NEGATION (tile1);
CONJUNCTION (List<String>, List<String>): returns List<String>	Perform the AND Boolean operation on 2 tiles	List<String> and = CONJUNCTION (tile1, tile2);
DISJUNCTION (List<String>, List<String>): returns List<String>	Perform the OR Boolean operation on 2 tiles	List<String> and = DISJUNCTION (tile1, tile2);
IMPLICATION (List<String>, List<String>): returns List<String>	Perform the Implication Boolean operation on 2 tiles	List<String> and = IMPLICATION (tile1, tile2);
XOR (List<String>, List<String>): returns List<String>	Perform the XOR Boolean operation on 2 tiles	List<String> and = XOR (tile1, tile2);
MAKE_TRIANGLE_UP (List<String>, Int): returns List<String>	Make a triangle-shaped tile. The triangle will point upwards	List<String> triangle = MAKE_TRIANGLE_UP (tile1, N);
MAKE_TRIANGLE_DOWN(List<String>, Int): returns List<String>	Make a triangle-shaped tile. The triangle will point downwards	List<String> triangle = MAKE_TRIANGLE_DOWN (tile1, N);
SUBTILES (List<String>, Int, Int): returns List<List<String>>	Split a tile into N subtiles of size M	List<List<String>> subtiled = SUBTILES (tile1, M, N);

EQUAL_TILES (List<String>, List<String>): returns Bool	Check whether 2 tiles are equal	if (EQUAL_TILES(tile1, tile2) {
		  do_something;
		};

print (List<String>)	Print a tile to Standard output	print (tile1);