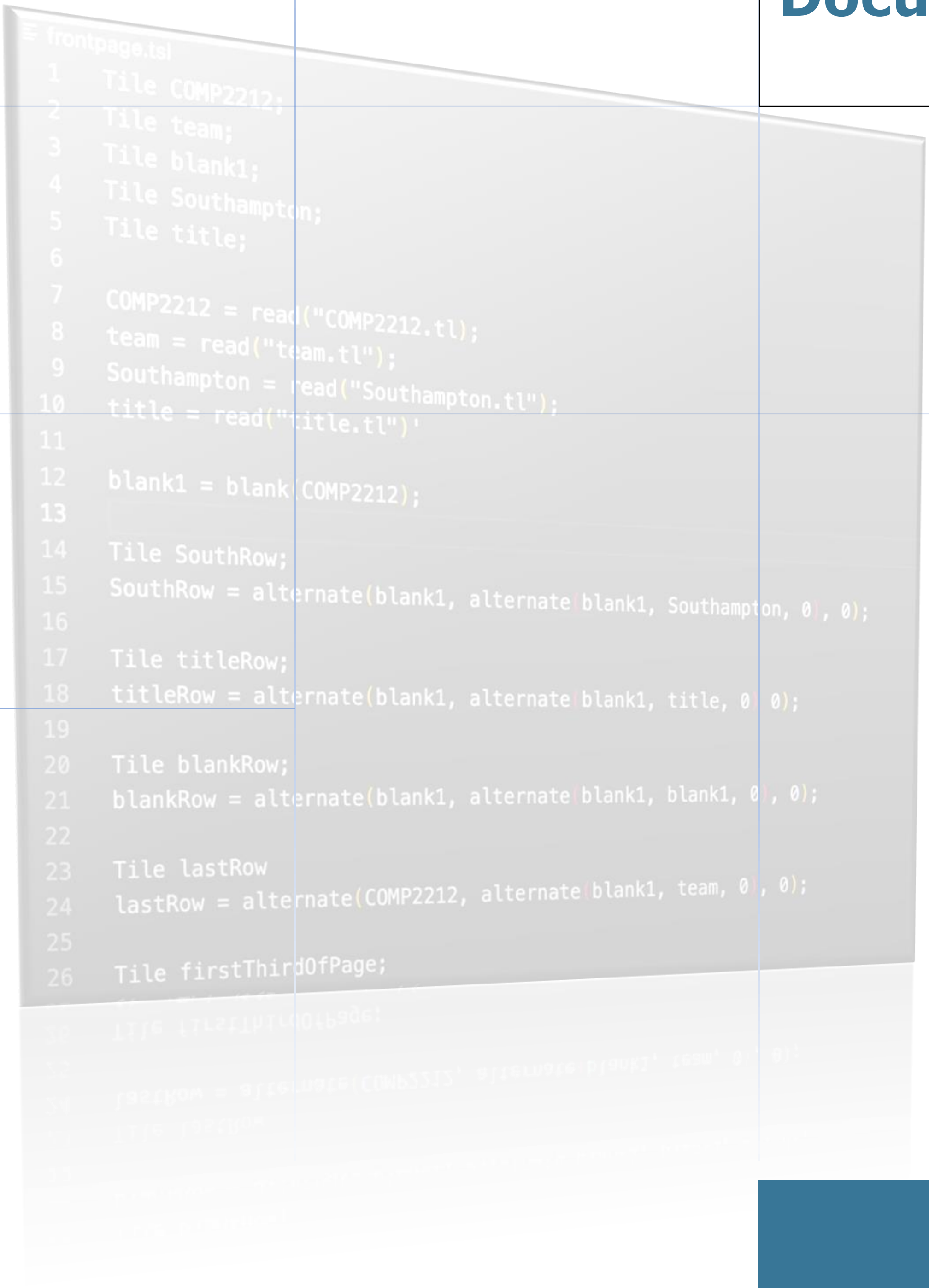
TSL Documentation


frontpage.tsl

```
Tile COMP2212;
Tile team;
Tile blank1;
Tile Southampton;
Tile title;

COMP2212 = read("COMP2212.tl);
team = read("team.tl");
Southampton = read("Southampton.tl");
title = read("title.tl")'

blank1 = blank(COMP2212);

Tile SouthRow;
SouthRow = alternate(blank1, alternate(blank1, Southampton, 0), 0);

Tile titleRow;
titleRow = alternate(blank1, alternate(blank1, title, 0), 0);

Tile blankRow;
blankRow = alternate(blank1, alternate(blank1, blank1, 0), 0);

Tile lastRow
lastRow = alternate(COMP2212, alternate(blank1, team, 0), 0);

Tile firstThirdOfPage;
```

Author:
Lam Giang

TABLE OF CONTENTS

1. TSL

TSL is a tiling language that has been created to address problems for domain specific problems that revolve around a tile.

Tiles have a set of attributes, the language was designed to consider them and render problem solving for tile specific problems, easier.

Here are the attributes of a tile:

- It is a language for tiles. A tile is of size N and has NxN cells.
- Tiles can be filled or empty
- Tiles consisted of strings, and in the implemented language (Haskell), would therefore be a list of lists of strings.

Other important considerations about tiles, namely how they are read:

- Input tiles would have the .tl extension and could be given in the source folder, the same as where the compiler would be found and run through an executable make file.

The domain specific language was inspired by problems that were given to solve by the university. Code used to solve these problems will be used to explain functions that were implemented to facilitate the domain,

2. The Compiler

Language implementation

The compiler's main components are the lexer, parser, and interpreter. The lexer was created using Alex, and the parser using Happy. Both are tools specific to the Haskell language to allow for an easy implementation of a lexer and parser when using Haskell as a language to create a compiler. The interpreter was written in Haskell.

The Lexer

The Tokens.x file contains a pre-amble with the tokens module followed by a "posn" wrapper declaration and macro declarations of digits and characters. The "posn" wrapper was chosen for error handling during the lexing process (Initialization state), in order to show the line and column where the error occurred. Finally the post-amble contains the following Haskell code to be able to output the error handling. The lexemes or what would end up as the words used in the language were carefully considered for readability and how they would be used when programming in TSL.

Apart from Tokenization, the Lexer provides actual functionality with the ability to comment code on a line-by-line basis, with "//" denotating the start of a comment.

Token naming convention was as such to be able to identify the token functionality, with language constructors written "TokenConstructor…" and functions named "TokenFunction…" to improve the readability of the file, this convention follows suite to the parser (Grammar.y file).

The Parser

The parser was created using Haskell's Happy tool, which in the compiler can be found under the Grammar.y file. Syntactic structure was given to the tokens from the lexer creating the parser's abstract syntax tree (AST).

The root of the AST is the "program". The program is a list of statements. A statement can either be a compound statement (block), a while statement or an if statement. The statements take expressions, the compound statements take a list of statements, the while statements take an expression and a statement and an if statement is takes an expression and a statement. The expressions that follow and how they help create the AST can be found in Grammar.y file.

To help picture how the AST is constructed, here is the BNF grammar for the parser for the TSL language:

```
<program> ::= <declarations> <statements>
<declaration> ::= "Tile" <identifier> ";" | "Int" <identifier> ";"
<statements> ::= (<statement>)*
<statement> ::= <assignment> | <while-loop> | <if-else> | <function-call>
<assignment> ::= <identifier> "=" <function-call> ";"
<while-loop> ::= "while" "(" <expression> ")" "{" <statements> "}"
<if-else> ::= "if" "(" <expression> ")" "{" <statements> "}" ("else" "{" <statements> "}")?
<function-call> ::= <function-name> "(" <argument-list> ")"
<argument-list> ::= (<expression> ("," <expression>)*)?
<expression> ::= <identifier> | <literal> | <binary-operation> | <unary-operation> |
<parenthesized-expression>
<literal> ::= <integer> | <string>
<binary-operation> ::= <expression> <binary-operator> <expression>
<binary-operator> ::= "+" | "-" | "*" | "/" | "&&" | "||" | "==" | "<" | ">" | "<=" | ">=" | "&" | "%"
| "|"
<unary-operation> ::= <unary-operator> <expression>
<unary-operator> ::= "neg" | "!"
<parenthesized-expression> ::= "(" <expression> ")"
<function-name> ::= "read" | "scale" | "blank" | "write" | "stack" | "alternate" | "rotate" | "reflectX"
| "reflectY" | "make_tile" | "getCol" | "getRow" | "neg" | "modify" | "access"
<identifier> ::= <letter> (<letter> | <digit>)*
<integer> ::= (<digit>)+
<string> ::= '"' <character>* '"'
<character> ::= <letter> | <digit> | <symbol>
<letter> ::= ("a" ... "z") | ("A" ... "Z")
<digit> ::= "0" ... "9"
<symbol> ::= "." | "," | ";" | ":" | "(" | ")"  | "{" | "}" | "|" | "&"
```

The Interpreter

When the interpreter executes it performs lexing and parsing of the programs source code (initialization state). When parsing the grammar file begins by building the abstract syntax tree. The root of the AST is the data type program. The hierarchy of the program will be a list of statements. From a statement, you can derive another statement, or a block or while loop. From a Statement you can also derive an expression, there are various types of expressions denoted by the Exp data type, as seen in the Grammar file.

In the main file, the function that is executed is the interpreter on an empty map which is applied to the AST. The map stores identifiers and then is recursively called which updates the identifiers in the map. This allows for an immutable state to act stateful. After execution the map will store the statements from the AST which the interpreter will pattern match and evaluate them, recursively updating the map.

The next state the interpreter goes in is the execution state. The interpreter has several functions, of which the main functions are the "interpreter" function, and the "evaluateExpression" function. The data types for these are the Map.Map String, as you pass in the item from what was initially the empty map, as well as the String which is the identifier, and then whatever statement was produced from the AST that you want interpreted. If the item that wanted to be interpreted was only a statement, such as the write(); function, it could be interpreted from the "interpreter" function alone. But if you had a statement that also had an expressions, such as "exp + exp", as this isn't a statement you need to evaluate it separately. This is where evaluateExpression was used.

'EvaluateExpression' also takes in a Map and a String, the same types as the "interpreter" function, as we are using the same map since we have a global scope and don't want to constrain it, so it should be able to access everything as well. It takes in a second argument to allow it to be recursive.

If an expression is a base case, since we only support two atomic expressions which do not need to be evaluated (being tiles/integers), the whole recursive procedure for evaluateExpression ends if we manage to get it down to integers or tiles.

There is no I/O in our language as this language is domain specific to tile configurations therefore there is no suspension state in the language. The termination state will occur when the program finishes executing or when an error/exception has caused it to prematurely exit.

Error handling

TSL supports error handling with error messages shown in the terminal. The interpreter produces output on standard output (stdout) and as requested, the error messages on standard error (stderr).

The supported errors include: variable declaration, errors in reading tiles, tile errors where the tile is not a 0 or 1, condition errors for "if" and "while" statements, variable initialisation.

The error handling functions by using the pattern matching integrated into the structure of the interpreter.

Type checking

There are two types in the TSL language: "Tiles" and "Int". The way the language is constructed, they must be initialised before being used. Initialisation checking has been implemented in the error handling, which functions as an inexpensive "Type checker". A full type checking system was not implemented due to the simplicity of the language. This will require the programmer to be responsible for the correct usage of types when using TSL.

3. Language features

A number of features are supported in TSL, which were used to address the domain specific problems.

End of statement

Statements are terminated using a semi-colon, this can be seen in the example below.

Initialisation of variables

Variables are required to be initialised before assigning a value. To the right, please see an example of the initialising of a tile "T1":

```
Tile t1;
t1 = 0;
```

The end of each statement can be seen by the usage of semicolons.

Assignment and re-assignment of variables

The language facilitates the assignment of variables using the "=" symbol, to be assigned, a variable must first be initialised. See example to the right:

```
Tile t1;
t1 = 0;
```

TSL also facilitates variable reassignment, to see an example see the code, right:
Here variable "t1" has it's value re-assigned from "0" to "1".

```
Tile t1;
t1 = 0;
t1 = 1;
```

Types

TSL supports two types "Tiles" and "Int". A "Tile" being a list of lists, or a NxN sized block. "Int" denotes an integer. Booleans are implicitly typed.

Scoping

There is global scoping that is featured as part of this language. Therefore, when programming using TSL, one must consider that variables are accessible throughout the program.

Comments

The language supports comments. To write a comment simply input "//" without the quotations and whatever one would like to comment after and it will not be read in the program.

While loops

While loops, as well as nested while loops are featured in the language. Here is an example of how to apply a while loop in TSL:

```
Int i;

i = 0;

while( i < 4){
print (i);
i = i + 1;
}
```

You implement the condition parenthesis after the while function, the overall statement contains another statement, within the while loop. This as one would expect would print i as an integer into the terminal, and one would see "0, 1, 2 , 3" on new lines.

Nested while loops are also featured in TSL.

If-else statements

If-else statements are used in TSL. It functions identically to the if-else statements in other imperative programming languages. For the syntax used in TSL, please see an if-else statement example to the right:

```
    if(col < 1) {
        new_tile = col_helper;
    } else {
        new_tile = alternate(new_tile,
col_helper, 0);
    }
```

Please note that in TSL, "else" blocks are not required.

Built in functions

There are numerous built-in functions in TSL that are domain specific in reducing the code necessary to solve problems. These will can be seen in the following section, 4.

Operations

Standard arithmetic operations on integers are supported in TSL. Booleans are implicitly supported as well as operations on Booleans "&&" (AND), "||" (OR), and Boolean negation "!". The modulo operator "%" is also supported on integers, with the similar syntax and functionality seen in other imperative programming languages.

4. Programming in TSL

The input tiles are required to be read. To use an input tile, it must first be initialised. Then, the programmer would want to ensure that the read() function is used to input a tile. Once the tile has been initialised and has been assigned, the built in functions below can be performed on the tile. Example on initialising a tile to the right:

```
Tile tl;
tl = read("tile1.tl");
```

Built-in functions

read()

The read function reads one of the .tl files found in the directory, and allows the user to assign it to a tile type so that it can be used in the program.

```
t1 = read("tile1.tl");
```

write()

This function writes a hardcode "output.tl" tile when called, or can be a string can be specified by the user for the output name. See both examples to the right:

```
write(res); //writes to
output.tl
write(res, "stringhere.tl");
//writes res to
"stringhere.tl"
```

stack()

This function works similarly to the alternate() function, except instead of horizontally, it has a vertical implementation as the name would imply. stack() takes in two arguments, the first item which will be stacked on the second item.
An example of stack would be (once the variables were initialised):

```
Tile res;
res = stack(t1,
t2);
print(res)
```

This would print out "t1" stacked on top of "t2", or if t1 = 0 and t2 = 1, it would print: 0 and 1 consecutively to the output.

alternate()

The alternate function allows a user to alternate between one tile, and another. Here is how it works with an example on the right. Alternate will take t1 and t2, and will implement a pattern that resembles: t1t2, t1t2, t1t2…x amount of times.
If the function was called alternate(t1, t2, 0); then there are 0 alternations, and it simply would implement the pattern: t1t2. alternate(t1, t2, 1); would implement the pattern t1t2t1t2.

```
Tile t1;
Tile t2;
alternate(t1, t2,
x);
```

rotate()

This functionperforms an operation that rotates a tile at a 90 degree angle in a clockwise manner. As it follows a clockwise operation. Performing rotate 4 times, would bring it back to it's original position.

```
Tile tr;
tr = rotate(tl);
//"tl" is the input tile
```

scale()

This allows the user to increase the size of a given tile. It takes in a tile and the integer used after gives it it's scaling factor. Example to the right:

```
t1 = scale(t1i, 50);
```

reflectX() and reflectY()

These functions reflect the tile along the x-axis and or the y-axis and can be used as seen in the code to the right:

```
tx = reflectX(t);
ty = reflectY(t);
```

make_tile()

This function allows the user to make a tile of specified dimensions that can crop the pixels from another time. See code to the right:

```
subtile1 =
make_tile(6,6);
```

getCol() and **getRow()**

These give an integer which is the amount of pixels on a tile in a row or column of the specified tile, see code to the right where "t3" is a tile.

```
x = getCol(t3);
y = getRow(t3);
```

blank()

This creates a blank tile of the dimensions of the tile input into the blank() function, example code to the right:

```
blank_tile = blank(t3);
//t3 is a tile
```

modify() this permits changing a pixel in a tile. Example code to the right:

```
modify(t3, xx, yy,
neg(val));
```

access() this allows access to pixels in a tile. Example code to the right:

```
val = access(t3, xx, yy);
```

neg, applied infront of a tile, this gives the negation of the pixels from the tile. This would turn a tile of "0"s into "1"s.

print() this function prints to the terminal to facilitate debugging.

5. Running TSL

To run the program, you have all the necessary files in the source folder:

- Tokens.x
- Grammar.y
- Interpreter.hs
- Tsl.hs
- The tiles in question, files with an extension .tl
- viewer.hs

1) First run the make file entering the following command in the terminal:
   `ghc Tsl Tsl.hs`
2) Then run the program in question, in this scenario the program "prog"
   `./Tsl prog.tsl`

This will print out the file to the source folder.