Language Influences

- **c#**
    - The use of semicolons to mark the end of a statement came from c#. The symbols are used as delimiters to split expressions so each can be passed through the lexer/parser separately giving separate abstract syntax trees. The symbols && || ^ % ! are taken from c# too.
- **Ocr pseudocode**
    - The if/then/else/endif construct was influenced by Ocr's pseudocode guide; I found the use of endif helped with program readability and it also solved the dangling else problem since both else and no else are supported.
- **Python**
    - The use of # for marking comments came from python. The symbol does not appear elsewhere in our language, making it easy to implement as a unique symbol in the lexer/parser. I also liked the simplicity of having a single symbol designate comments rather than multi-symbol solutions such as // or <!--. Additionally, use of colons inside if/for blocks was inspired by the use of colons in place of then/do in python - colons here instead of semicolons ensure if/for statements are treated as blocks rather than multiple separate expressions.

Language Syntax and Lexical Rules

Types: The language has three distinct types.

- Tint - A 64 bit signed integer. Can be positive or negative.
- Tstring - A string of characters, made from alphabetical letters only. Used primarily for debugging.
- Ttile - represents a tile. (In the language this is a 2D list of Ints).

Terminators: The ';' character is used to mark the end of a statement, unless the statement is within a conditional or a loop, in which case the ':' character is used. This is for ease of parsing as if/for statements and their contents are treated as single expressions so that a single abstract syntax tree is generated.

Assignment: The assignment operator '=' is used to assign values to a variable; and the let statement can be used to assign local variables.

```
#[type] [variable] = [expression];
Tstring var = Hello World;

#Let [variable] = [expression] in [expression]
Tint y = let x = 5 in (x + x*2);
```

Relational Operators: Equal, Less than, Greater than, Greater than or equal and Less than or equal are supported. Return a boolean expression. Type can be Tint or Ttile.
Further symbols for And, Or, Xor and Not can manipulate boolean expressions.

```
[type] exp1 == exp2 #equal
[type] exp1 < exp2 #less than
[type] exp1 > exp2 #greater than
[type] exp1 >= exp2 #greater than or equal to
[type] exp1 <= exp2 #less than or equal to
!([type] exp1 == exp2) #not
([type] exp1 > exp2) && ([type] exp3 > exp4) #and
([type] exp1 < exp2 || ([type] exp3 < exp4) #or
([type] exp1 >= exp2) ^ ([type] exp3 <= exp4) #xor
```

Conditionals: The ‘if’, ‘then’, ‘else’, and ‘endif’ keywords are used to create conditional statements. Else may be omitted, and the indentation is optional. Conditionals can be nested. An endif that is nested needs the ‘:’ terminator.

```
#if [boolean expression] then
if Tint x >= 5 then
      Tint x = 10:
else
      Tint x = 0:
endif;
```

Iteration: The ‘for’, ‘do’ and ‘endfor’ keywords are used to create loops. These can be nested, and nested loops must end in ‘:’ instead of ‘;’. Indentation is optional. For loops have a maximum iteration count of 9999.

```
#for [initialisation] : [boolean exp] : [expression] do
for Tint var = 0 : Tint var > 5 : Tint var = var + 1 do
      #do something here
      Tint var2 = var2 * var2:
endfor;
```

Output: The outputTile function writes a tile to stdout. The function does not return anything. Only applicable to Ttiles. There is also an output function for the Tstring type, for debugging.

```
outputTile t;
outputString Hello World;
```

Mathematical Operators: Addition, subtraction, multiplication and integer division are supported. The subtraction operator also creates negative numbers. Only applicable to integers. Most are associative e.g (5 * 3 * 10) is allowed.

```
Tint var = exp1 + exp2;
Tint var = exp1 - exp2;
```

```
Tint var = - exp1;
Tint var = exp1 * exp2;
Tint var = exp1 / exp2;
Tint var = exp1 % exp2;
```

Tile Functions:

Creating new tiles:

```
Ttile t = inputTile str.str; #file name must be .tl format.
Ttile t = blank x y; #creates tile of all unfilled squares to the specified dimensions
Ttile t = full x y; #creates tile of all filled squares to the specified dimensions
```

Transforming tiles:

```
Ttile t1 = rotateR t2; #rotates a tile 90 degrees clockwise
Ttile t1 = rotateL t2; #rotates a tile 90 degrees anti-clockwise
Ttile t1 = flipX t2; #flips a tile horizontally
Ttile t1 = flipY t2; #flips a tile vertically
Ttile t1 = scaleUp t2 i; #scales a tile up by the factor i
Ttile t1 = scaleDown t2 i; #scales a tile down by the factor i
```

Combining/Extracting tiles:

```
Ttile t1 = place t2 on t3 x y; #places t2 on t3, with the top-left the replaced part of t3 aligning with the top-left of t2, and x and y describing the coordinate of the top-left of the replaced section of t3
Ttile t1 = subtile t2 x y i j; #returns a section of t2 - x and y define the top-left of the part to be returned, i and j define the dimensions of the section to return
```

Boolean operations on tiles:

```
Ttile t1 = t2 and t3; #performs boolean AND on each square of t2 and t3
Ttile t1 = t2 or t3; #performs boolean OR on each square of t2 and t3
Ttile t1 = t2 xor t3; #performs boolean XOR on each square of t2 and t3
Ttile t1 = not t2; #performs boolean NOT on each square of t2
```

Integer values from tiles:

```
Tint x = getWidth t; #returns the width of tile t
Tint y = getHeight t; #returns the height of tile t
```

Scoping Rules

- The language has one scope; the global scope, meaning all variables are stored in a single stack for ease of evaluation. The one exception to this is **let** statements, where variables do not get added to the main stack and cannot be accessed outside the scope of that statement.

Syntactic Sugar

- Instead of writing **(exp1 < exp2) || (exp1 = exp2)** the language provides the syntax **exp1 <= exp2** that performs the same operation. **>=** does the same thing but for more than operations.
- Similarly, the language provides the **xor** operation for tiles, to remove the need for multiple **and** / **or** operations that would do the same thing.
- The language also provides the **!** operator to negate a boolean expression/value. This is provided so that bulky **if/else** blocks are not needed to perform the same task.
- The language supports easy negatives by allowing the **-** operator to be used with one expression on the right side (**- exp**). This removes the need for having to write **0 - exp** every time a negative is required.

Type Checking

- This language has **Strong Static** typing; a type is required before every assignment and relational operator expression (**manifestly typed**).

Error Messages and Warnings

- Errors and warnings in this language are made to be clear and distinct, stating exactly what went wrong and why. If the lexer encounters an invalid token then “**lexical error**” is output. If the parser encounters a parse error, the message “**Parse error: [token]**” where [token] is the token that caused the error. If the lexer and parser find no errors but the AST evaluation does, then a descriptive error message is output, unique to the situation, stating the type of error and what caused it.
- Because this language should only ever output tiles, if the string output option is used for debugging, the warning "**WARNING: String output for debugging only**" is output. If the script contains no output at all then the warning "**WARNING: No output**" is output to alert the programmer.
- There are 3 types of errors: stack, function, and type errors.

Stack (2 errors)

- If a for loop reaches the maximum iteration count of 9999 iterations, the error "**Stack error (for): Maximum iteration count reached (9999)**" is output.

- If the interpreter cannot find a variable's value in the stack, then this error is output: "**Stack error: Variable ' [variablename] ' doesn't exist, or the program is badly typed**" where [variablename] is the name of the variable.

Type (5 errors)

- If some operators are used with invalid types, clear and detailed error messages are output to alert the programmer, such as: "**Type error: type Tstring can't be used with operator '*'**" and "**Type error: type Tstring can't be used with function 'GetHeight'**"

Function (10 errors)

- There are also function errors for tile functions that are output when invalid parameters are passed, such as:
    - "**Function error (blank): Integer parameters must be positive**"
    - "**Function error (subtile): Invalid values passed**"
    - "**Function error (and|or|xor|not): Cannot use boolean operators on tiles with different dimensions**"
    - "**Function error (scaleDown): tile cannot be scaled down by provided value, ensure value is a factor of the tile height**"

Interpreter Execution Model

- First the code is split into individual statements using ; as a delimiter, then each statement is fed into alex to tokenize it, then fed into happy to parse it, ending up with an abstract syntax tree. The interpreter then makes extensive use of pattern matching to evaluate expressions into values.
- The interpreter uses a Params data type that it passess between most functions - it consists of 3 separate stacks for the 3 data types (Tint, Tstring, Ttile), a tile to be output, and a string to be output (for debugging). This makes assignment simple as variable names and their values are simply added to the appropriate stack which is then returned. The state of the Params variable (stacks and output variables) defines the runtime state.
- The interpreter stores tiles internally as a 2D array of integers (as a data type called Tile). Stacks are represented internally as a list of pairs: **[(variablename, value)]**.
- After the interpreter has evaluated the last expression, the output tile stored in Params is converted to a string and sent to stdout if its value has been changed by the outputTile function. The same process happens for the output string.