# Tile Language Manual

## Introduction

This interpreter enables tile manipulation. The language itself is procedural, declarative and imperative. The language supports all types of basic programming constructs: sequencing, branching and looping. For sequencing statements, each statement on a newline must be delimited with a semi-colon. Branching allows for a predicate; if the predicate evaluates to true then the first branch is executed; else the end branch is evaluated. Looping is managed by a for loop which allows for creation of larger tiles using grid coordinates. Tiles are represented as strings within the language composed of only '1's or '0's, with the final tile output being a single string separated by new lines.

## Syntax

The syntax of the language is composed of expressions, which themselves are composed of built-in functions, comparison operators, arithmetic operators, the 3 basic programming constructs, variable initialization and primitive data types. Apart from the primitive data types, all parts of an expression act as higher order functions, in that they can be built up of constituent expressions to create an overall Abstract Syntax Tree (AST).

Curly braces represent control and instructs the interpreter to execute whatever expression is contained within. Parentheses are used to assign precedence to expressions. Colons are used to specify the type of a declared variable. Semi-colons delimit sequences.

Comments are prefixed with double forward slashes.

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

In the following sections, `<Expr>` refers to a syntactically correct expression in the Tile Language (TL).

### Primitive data types

The TL contains 3 primitive data types:

- Boolean – represented as `true` or `false` in the syntax
- Integer – represented as `int` in the syntax, where a natural number, n, declared in an expression is tokenized as `int n`
- String – represented as `str` in the syntax, where a string surrounded by double quotes, s, declared in an expression is tokenized as `str s`

### Functions

Functions in the language may manipulate tiles (with the tiles in the format of a string), Booleans or positive integers.

`COPY <Expr> <Expr>` - takes in 2 expressions, the first of type `Int` and the second of type `String`, and copies the tile string in the x-direction by the amount set by the integer.

`CONCAT <Expr> <Expr>` - takes in 2 expressions, the first of type `String` and the second of type `String`, and attaches the tile strings together in the x-direction.

`ENLARGE <Expr> <Expr>` - takes in 2 expressions, the first of type `Int` and the second of type `String`, and enlarges the tile string by the scale set by the integer.

`ROTATE_R <Expr> <Expr> -` takes in 2 expressions, the first of type `Int` and the second of type `String`, and rotates the tile string clockwise by valid increments of 90 degrees. The integer may only be 90, 180, 270 and 360.

`STACK <Expr> <Expr> -` takes in 2 expressions, the first of type `String` and the second of type `String`, and attaches the tile strings together in the y-direction.

`EMPTY <Expr> -` takes in an expression of type `String` and returns a blank tile (containing zeros only) of the same dimensions as the tile string.

`REFLECT_Y <Expr> -` takes in an expression of type `String` and reflects the tile string in the y-axis.

`REFLECT_X <Expr> -` takes in an expression of type `String` and reflects the tile string in the x-axis.

`NEGATE <Expr>` - takes in an expression of type `String` and swaps the '1's for '0's and vice versa, in the tile string.

`SHRINK <Expr> <Expr>` - takes in 2 expressions, the first of type `Int` and the second of type `String` and shrinks the tile string by the scale set by the integer.

`CONJUNCT <Expr> <Expr> -` takes in 2 expressions, the first of type `String` and the second of type `String` and performs the conjunction of the 2 input tile strings.

`GET_SUBTILE ( <Expr>, <Expr>) <Expr> <Expr> -` takes in 4 expressions; the first 3 expressions of type `Int` and the 4th of type `String`. The first two integers refer to the row and column indices of the input tile string from where the sub-tile should begin. The 3rd integer specifies the width and length of the sub-tile to extract from the tile string (the input string).

`LOOP <Expr> { <Expr> } -` takes in 2 expressions, the first of type `Int` and the second of type `String,` and stacks the tile string n times in the vertical direction.

`LENGTH var <Expr> -` takes in a variable of either 'H' or 'W' (standing for height and width, respectively) and expression of type `String,` and returns the calculated height or width of the tile string, as specified, as an integer.

`COMPARE <Expr> <Expr> -` takes in 2 expressions, the first of type `String` and the second of type `String,` and compares the two tile strings, returning `true` if they are the same, or `false` otherwise.

Most functions are non-associative and do not bind tighter than any other function.

## Variable initialization

Variables are initialized using a LET statement in the syntax:

```
LET (var : <TLType> = <Expr>) IN { <Expr> }
```

The variable and its assigned expression may be of arbitrary type. The second expression within the curly braces is also of arbitrary type and represents the scope in which the variable acts. This enables the language to achieve local scope. Global scope can be emulated by wrapping the entire code in a LET statement.

## Programming constructs

### Sequence

Sequencing of expressions occurs via two methods in the language. Semi-colons are used to delimit expressions of type `String,` which results in the concatenation of the string outputs. Additionally, LET statements, where variable initialization takes place, can be chained if several variables need to be implemented in a code block. This helps to reduce code redundancy.

### Branching

Branching occurs via an IF-THEN-ELSE statement in the syntax:

```
IF ( <Expr> ) THEN { <Expr> } ELSE { <Expr> }
```

The first expression is of type Bool, which is evaluated and then decides whether to execute the expression in the THEN or ELSE block, which are both of arbitrary types.

### Looping

Looping occurs via a FOR loop statement in the syntax:

```
FOR (var, var : <TLType> = <Expr>, <Expr>, var) DO { <Expr> }
```

The primary purpose of the for loop is to build up a tile, using row and column indices. The first variable is either "ROW" or "COL", which indicates whether to create a row or a column in the for loop. The second variable is the character used to instantiate and initialize the index, which is then stored alongside the first expression of type `Int.` The second expression is of type Bool and is the loop predicate based on the value of the index. The third variable is either “ASC” or “DESC”, which indicates to the evaluator whether to increment or decrement the index. The predicate is evaluated and if true, the expression (of type `String`) in the `DO` block is executed. Afterwards, the index is incremented or decremented accordingly, and the predicate is evaluated again. If “ROW” the `DO` block output is concatenated along the horizontal direction and if “COL”, along the vertical direction. To build a 2D tile, a “COL” loop can be nested within a “ROW” loop, which builds the tile row by row.

### Comparison operators

The comparison operators are demonstrated in Figure 1. They can be split into two sets: one consisting of mathematics symbols and the other consisting of Boolean logical operators.

The mathematics symbols include greater than, less than, equals, not equals, greater than or equal to and less than or equal to. They take in expressions on both sides of the operator of type `Int` only and return type Bool. The operators are non-associative.

The Boolean logical operators include ‘and’, ‘not’ and ‘or’, as words and as symbolic representations: &&, ! and ||, respectively. They take in expressions of type Bool only and return type Bool. The operators are non-associative.

### Arithmetic operators

The arithmetic operators are demonstrated in Figure 1. These include addition, subtraction, multiplication, modulus, division and exponent.  They take in expressions on either side of type `Int` only and return type `Int`. Addition and subtraction bind tighter than multiplication and division to avoid shift/reduce conflicts and to follow the correct order of BIDMAS.

## Inspiration

The commands are written with capital letters, which is inspired by older programming languages such as BASIC. This makes it easier to read and separates the expression from the command. The FOR loop was inspired by the C programming languages, consisting of variable declaration, variable initialization, predicate and increment/decrement.

## Execution Model

The interpreter possesses the capacity to execute the fundamental stages of any interpreter.

### File reading

When the interpreter is executed, it first reads in the code file as a string and then the tile files are read in as strings and stored in a dictionary with their file name. The tile strings are cleaned to remove trailing whitespace. The interpreter then replaces instances of the tile file name in the source code as “tilex.tl”, with the actual tile string. Resultantly, tiles are declared in the code in the “tilex.tl” format.

### Lexing

The second stage of execution pertains to lexical analysis. The modified source code is then scanned and certain sequences of characters are identified as either keywords, literals, variables and operators. Alex was used to generate the lexer for the language. The Alex wrapper used was the posn wrapper which returns the position of each token in the source. This allows the parser to return the position of the syntactical error in the source, in turn allowing the programmer to amend the typo. Code prefixed with “//” are comments and are ignored by the lexer. String lexemes are prefixed and suffixed with double quotes and can contain any number of characters, numbers or special characters. Var lexemes are akin to their string lexemes except for double quotes. Integers can be built through either singular digit or multiple digits; however negative numbers are not supported by the lexer.

### Syntax analysis

The third stage of execution involves syntax analysis. After the tokens are generated from the source code, the parser ensures that the source code conforms to the grammar. The parser is generated by happy. The grammar of the Tile Language is described in the Syntax section and is shown in full in Figure 1. The nonterminal symbol `<Expr>` consist of the functions, arithmetic operators, boolean operators and terminal symbols. `<TLType>` constitutes the types that are formed from the terminal symbols. The language is strongly typed, with enforced type declarations for variables as seen in the grammar. This constructs a corresponding AST for the source according to the grammar. Associations have been specified regarding certain operators. This is to avoid shift/reduce conflicts where the resulting state machine is no longer a Deterministic Finite Automaton (DFA) but instead a Non-Deterministic Finite

Automaton (NFA). Most operators are non-associative, meaning they do not bind higher or lower than any other non-associative operators.

```
<Expr> ::= <Expr> ; <Expr>
        | COPY <Expr> <Expr>
        | CONCAT <Expr> <Expr>
        | ENLARGE <Expr> <Expr>
        | ROTATE_R <Expr> <Expr>
        | STACK <Expr> <Expr>
        | EMPTY <Expr>
        | REFLECT_Y <Expr>
        | REFLECT_X <Expr>
        | NEGATE <Expr>
        | SHRINK <Expr> <Expr>
        | CONJUNCT <Expr> <Expr>
        | LOOP <Expr> { <Expr> }
        | FOR (var, var : <TLType> = <Expr>, <Expr>, var) DO { <Expr> }
        | IF ( <Expr> ) THEN { <Expr> } ELSE { <Expr> }
        | LET (var : <TLType> = <Expr>) IN { <Expr> }
        | GET_SUBTILE ( <Expr>, <Expr>) <Expr> <Expr>
        | LENGTH var <Expr>
        | COMPARE <Expr> <Expr>
        | <Expr> == <Expr>
        | <Expr> /= <Expr>
        | <Expr> <= <Expr>
        | <Expr> >= <Expr>
        | <Expr> && <Expr>
        | <Expr> AND <Expr>
        | READ var
        | <Expr> || <Expr>
        | <Expr> OR <Expr>
        | ! <Expr>
        | NOT <Expr>
        | <Expr> < <Expr>
        | <Expr> > <Expr>
        | <Expr> + <Expr>
        | <Expr> - <Expr>
        | <Expr> / <Expr>
        | <Expr> * <Expr>
        | <Expr> ^ <Expr>
        | <Expr> % <Expr>
        | ( <Expr> )
        | true
        | false
        | str
        | var
        | int

TLType ::= Bool
        | String
        | Int
        | ListString
```

*Figure 1 Grammar for Tile Language*

### Type checking
After the AST has been produced, the interpreter must perform semantic analysis to ensure the types of the AST match. As mentioned, the language is strongly and statically typed, with types declared upon initialization of variables before runtime and types are checked at compile time. For each expression in the AST, the base type is determined and stored in the type environment. The AST is then pattern-matched against the type rules. The type checker fails when types do not match and thus the source is semantically incorrect. A dedicated error message is printed to stderr in this event.

### CEK machine
Finally, after the source code has been proven to be semantically correct, the evaluator will begin to perform reductions on the AST. The evaluator was implemented using a Closures, Environments and Kontinuations (CEK) machine. This was chosen because of the increased performance gained from the use of these 3 constructs.

The expression is passed to the evaluator, alongside an empty environment and empty continuation. The environment contains the current variables and their bindings which have been evaluated so far. The continuation is a stack of frames that represents the current queued evaluation sequence. When a sub-expression is evaluated, control is returned to the continuation, which determines the next stage of evaluation in the AST. Each expression is pattern matched in the evaluator and a one-step reduction is performed, with a frame pushed to the continuation. When the expression has terminated, the frame for the current evaluation step is popped off. If the continuation is empty, this signals the termination of evaluation, and the base case is returned by the evaluator. In the language, a base case is of type String, Bool or Int. Otherwise, if the top of the stack contains a base case, the beta-reduction is performed or if it contains another expression, this expression is further evaluated.

The expression is recursed on by the evaluation loop, which calls the one-step evaluator, and continues until the expression can no longer be reduced.

### Tile output
The resultant expression composed of base cases is then unparsed to produce an output string. This string is further formatted and then printed to stdout using putStrLn.

## Error handling
Syntactical errors are specified by the column and row of the source code to allow easier debugging. If a certain stage of the interpreter fails, the standard error out returns the stage of interpretation that failed, along with the specific error e.g. index out of range, incorrect argument etc. Additionally, type errors are thrown if the code does not conform to type rules, during type checking. This allows programmers to revaluate their code and refactor it according to what is failing.