

University of Southampton
School of Electronics and Computer Science

PLC Comp2212 TileLang Coursework Report

Developer:

Chengkang Gu	cg1y21	ID:33071993
Shawn Yao	sy1g21	ID: 32535538
Zhaoxu He	zh2u21	ID: 33345503

21/4/2023

“This was a new challenge for us and this exercise gave us a whole new understanding of the language.”

Introduction:

This report presents our new programming language, 'TileLang,' which draws inspiration from Java and incorporates numerous enhancements and innovations to boost its flexibility, simplicity, and efficiency. Our primary objective is to offer users a concise and potent language for 'Tile' modules, streamlining the development process while enhancing the code's readability, maintainability, and stability. To accomplish this goal, we have conducted extensive research on Java's scope usage and incorporated various detection mechanisms to bolster the language's stability and maintainability. Furthermore, TileLang supports loops and a variety of methods to simplify the code and improve its readability. We have studied and referenced multiple languages, identifying their shortcomings to avoid making the same mistakes.

Design thinking:

Our new programming language features a Java-inspired execution model, in which semicolon-separated expressions serve as independent and indivisible units of work. These expressions execute sequentially according to the order in which they appear in the program. The order of expressions is controlled by keywords such as "for," "if," and "else," similar to Java. The interpreter employs a depth-first search (DFS) approach to execute the parse tree, with "Statement" expressions functioning as nodes within the tree. Starting with the leftmost innermost expression in the parse tree, the interpreter continues to evaluate the remaining expressions in the standard DFS order. To manage the state of variables and streams within the program, we have implemented a CEK-style model. In this model, the interpreter stores relevant information in a "State" data type. Whenever there is a change, the interpreter modifies this state and passes it to each new call of the expression evaluation method. This ensures that the program's state is accurately maintained and updated throughout the execution process.

The Lexical Definition:

A number of lexical rules have been laid down to make the language easier to write, understand and read, and to make it easier for developers to use. Similarly, we have taken many of the advantages of java to improve our lexical usability and convenience. We have set up identifiers, keywords, operators, separators, and literals. The identifier consists of a letter, a number and an underscore, where the letter can be upper or lower case and there is no limit to the length of the identifier but some keywords (such as *if*, *for* etc.) cannot be used as identifiers. Variable names are not allowed to start with a number. In addition we set up a lot of keywords, as well as operators and functions. These include the most basic operators such as +, -, *, /, % and so on, as well as the *if*, *else* and *for* keywords. In addition, we have specified that function calls and identifiers are not allowed to have the same name as the function. We also define symbols such as (), {}, [], ;, etc., in order to keep the language's scope and statements clear and to avoid confusion.

Our lexical analyser handles whitespace characters and comments efficiently, ensuring that they do not interfere with parsing.

```
$white+      ; ---space
"//".*       ; ---comment
```

For the sake of language flexibility and ease of reading files, we have designed tokens for the String type.

```
\".*\\"      { tok (\p s -> TokenString p s)}
```

The syntax and grammar:

-**Statement** : Represents a basic statement between a semicolon

-**Expr** : Represents all expressions, but cannot exist separately. They must exist inside a statement

-**Elemlist**: Used to handle list and matrix

-**Type**: All the basic type

Category	Syntax	Explanation
Statements and semicolons	s1 ; s2 s1;	All statements must be separated by a semicolon, and if two statements are linked, they are denoted by "sequence", i.e. "s1; s2;". If it is a closing statement, a semicolon should be used to end the statement, i.e. "semi".
Variable declaration and assignment	Type var var '=' Expr Type var '=' Expr	Our language allows for the separation of definition and assignment, making it easy to define when needed, similar to the advance declaration of java.
While loop	for '(' Expr ')' '{' Statement '}'	The for loop is slightly different from Java, but similar to the while loop: the parentheses computes the result of a BOOL . Operation and the curly brackets are the statements to be executed in the loop. For loop variables, users are asked to make changes within the loop. e.g.: for (foo<num) { foo+=1; }
If else Operation	if '(' Expr ')' '{' Statement '}' if '(' Expr ')' '{' Statement '}' else '{' Statement '}'	If else statements are defined in two further ways, one with else and one without. The main purpose is to make it easier for the user to be able to choose the right one to use. Parentheses are required to use expressions of type BOOL , which is checked in type check.
Arithmetic Operators	Expr '+' Expr Expr '-' Expr Expr '*' Expr Expr '%' Expr Expr '/' Expr	The mathematical calculation notation has essentially the same function as the calculation notation in java for calculations on int types. We have added some syntactic sugar to make it easier for developers to use. Expr '+=' Expr ;Expr '-=' Expr ;Expr '++'; Expr '--'

Boolean operators	Expr '==' Expr Expr Expr '!=' Expr Expr '!' Expr Expr ' ' Expr Expr '&&' Expr	Boolean operators have a similar function to java's operators and are mainly used for calculations on bool types.
Binary Comparison Operators	Expr '<' Expr Expr Expr '>' Expr Expr '<=' Expr Expr Expr '>=' Expr	Similar to the binary comparison operator in java, it is mainly used to compute on int types and produce BOOL results.
method	Expr.Method(Expr) Method(Expr)	We have designed different calls for different methods and some will have return values and some will not. Overall our methods have flexible scheduling and can adequately cater for the calculation of Tile modules.
List and Matrix	Expr ',' ElemList	The most flexible aspect of our language is that you can define your own Matrix and List, and you can define your own Tile, which makes it easy to deal with all the scenarios you need.

Evaluation:

The Eval module is responsible for evaluating the abstract syntax tree produced by the parser and producing a result.

1. Data Types

This module defines several data types that are used to represent values and environments. The TlValue data type defines the possible values that can be produced by the evaluation functions, including integers, booleans, tuples, and strings.

```
data TlValue = VInt Int
            | VBool Bool
            | VIntArrayList [Int]
            | VIntMatrix [[Int]]
            | VTuple TlValue TlValue
            | VString String
            deriving (Show, Eq)
```

The ValueEnvironment type is a list of (String, TlValue) pairs that represents the current state of the program's variables.

```
type ValueEnvironment = [ (String, TlValue) ]
```

TypeEnvironment, imported from TypeCheck represents the current type status of variables.

```
type TypeEnvironment = [ (String, TlType) ]
```

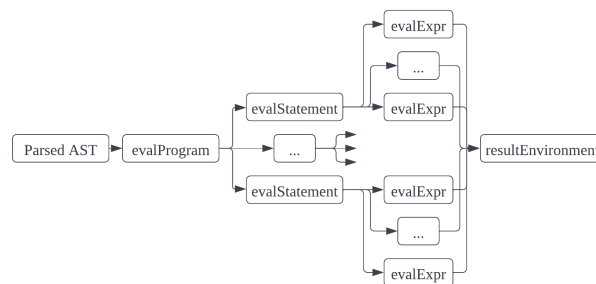
Notably, TlType is imported from Parser and constitutes of:

```
data TlType = TyBool | TyInt | TyString | TyIntMatrix | TyIntArrayList | TyTuple TlType TlType
            deriving (Show, Eq)
```

2. Functions

Several functions were used to evaluate expressions and manipulate the ValueEnvironment. The *lookupValue* and *updateValue* functions are used to look up and update variables in the ValueEnvironment, respectively.

The *evalProgram* function is a higher-level function that evaluates a list of statements representing a program in DFS order and returns the resulting environment of variable bindings. It takes a list of Statement as an argument and returns a ValueEnvironment. The function works by first initializing an empty environment for tracking variable values. It then applies the *evalStatement* function to each statement in the program, updating the environment along the way; *evalStatement* function will breakdown the statement to expressions and evaluates atomic syntax using the *evalExpr* function. Both function takes two environments as arguments - one for tracking variables that are in scope and another for tracking the values of those variables. Each call returns a new environment with any variable bindings created or updated by the statement. The resulting environment is returned as the final result.



3. Execution Model

The interpreter's execution model is driven by the environment, with the type and value environments serving as the runtime states. These environments are transformed throughout the program's execution to ensure type safety and proper handling of variable values. By adopting an environment-driven approach, the interpreter can efficiently evaluate expressions and operations while maintaining the integrity of variable states.

To evaluate expressions, the interpreter uses a recursive process that involves evaluating the sub-expressions and combining the results. The *evalExpr* function serves as the main entry point for atomic expression evaluation, and it is responsible for handling various types of expressions, such as integer literals, boolean literals, string literals, and variable references. On the other hand, *evalStatement* is responsible for breaking statements down into expressions.

During execution, the interpreter maintains a ValueEnvironment that represents the current state of the program's variables. The *lookupValue* and *updateValue* functions are used to look up and update variables in the ValueEnvironment, respectively. When a variable is referenced in an expression, the interpreter looks up its value in the ValueEnvironment. When a variable is assigned a new value, the interpreter updates its value in the ValueEnvironment; during this whole process the interpreter uses the type environment to ensure type safety and the value environment to retrieve the values of variables involved in the expression.

The interpreter also maintains a call stack that represents the current state of the program's function calls. When a function is called, a new frame is pushed onto the call stack that contains the function's arguments and local variables. When a function returns, its frame is popped from the call stack and its return value is passed back to the calling function.

Overall, the execution model for the interpreter is based on the evaluation of expressions and the manipulation of the ValueEnvironment and call stack. By recursively evaluating expressions and updating the ValueEnvironment and call stack, the interpreter is able to execute programs written in the TileLang.

Limitations:

Our language has certain limitations, such as the lack of user-defined function capabilities and Object-Oriented Programming (OOP) features. Additionally, the software has a limited set of data structures available.

Extension: error Check:

We have implemented a type detector in TileLang(TypeCheck.hs). This detector provides users with timely feedback on any incorrect type input, and it comes with a clear error report that can quickly pinpoint the location of the error message. Furthermore, we have included a syntax detector that identifies errors in the syntax of a position, incorrect input, and other related issues.

Extension: comment:

In TileLang, code can be annotated using the "//" symbol. These annotations are used to add comments to the code, and they are ignored by the interpreter during program execution. The purpose of annotations is to provide additional information to the readers of the code, such as explanations of what certain parts of the code do or why certain decisions were made during the programming process.

Extension: highlight:

As an additional feature, we have implemented syntax highlighting in TileLang. This feature visually distinguishes different elements of the code based on their syntactical role, making the code easier to read and understand. Syntax highlighting helps users quickly identify keywords, variables, comments, and other parts of the code, making it easier to spot errors and improve the overall coding experience.

Extension: Syntactic sugar:

TileLang supports a number of syntactic sugars that allow for more concise and expressive code, for example: +=, -=, ! etc...

Appendix:

TypeCheck:

```
Parsing : Int x = false;
Int t = 2;
Int y = 3;
```

```
Parsed as TlSequence (TlDeclareAssign TyInt "x" TlFalse) (TlSequence (TlDeclareAssign TyInt "t" (TlInt 2)) (TlSemi (TlDeclareAssign TyInt "y" (TlInt 3))))
```

```
Type Checking : TlSequence (TlDeclareAssign TyInt "x" TlFalse) (TlSequence (TlDeclareAssign TyInt "t" (TlInt 2)) (TlSemi (TlDeclareAssign TyInt "y" (TlInt 3))))
```

Wrong assignment

```
CallStack (from HasCallStack):
  error, called at ./TypeCheck.hs:270:23 in main:TypeCheck
```

```
Parsing : ArrayList<> = 1;
```

Parse error at line:column 1:10 Mistake unexpected token : '<'

```
CallStack (from HasCallStack):
  error, called at ./Parser.hs:3253:21 in main:Parser
```

```
Parsing : Matrix<Int>[][] a = [[1,2],[3,4]];
a = a.rotate(30);
```

cant rotate

```
CallStack (from HasCallStack):
  error, called at ./Eval.hs:217:14 in main:Eval
mac@yaoshiyangdeMacBook-Pro submission2 % ./Tsl test.tsl
Parsing : Matrix<Int>[][] a = [[1,2],[3,4]];
a = a.repeat(-1);
repeat num error
CallStack (from HasCallStack):
  error, called at ./Eval.hs:239:32 in main:Eval
```

Comment:

Parsing : //this is comment

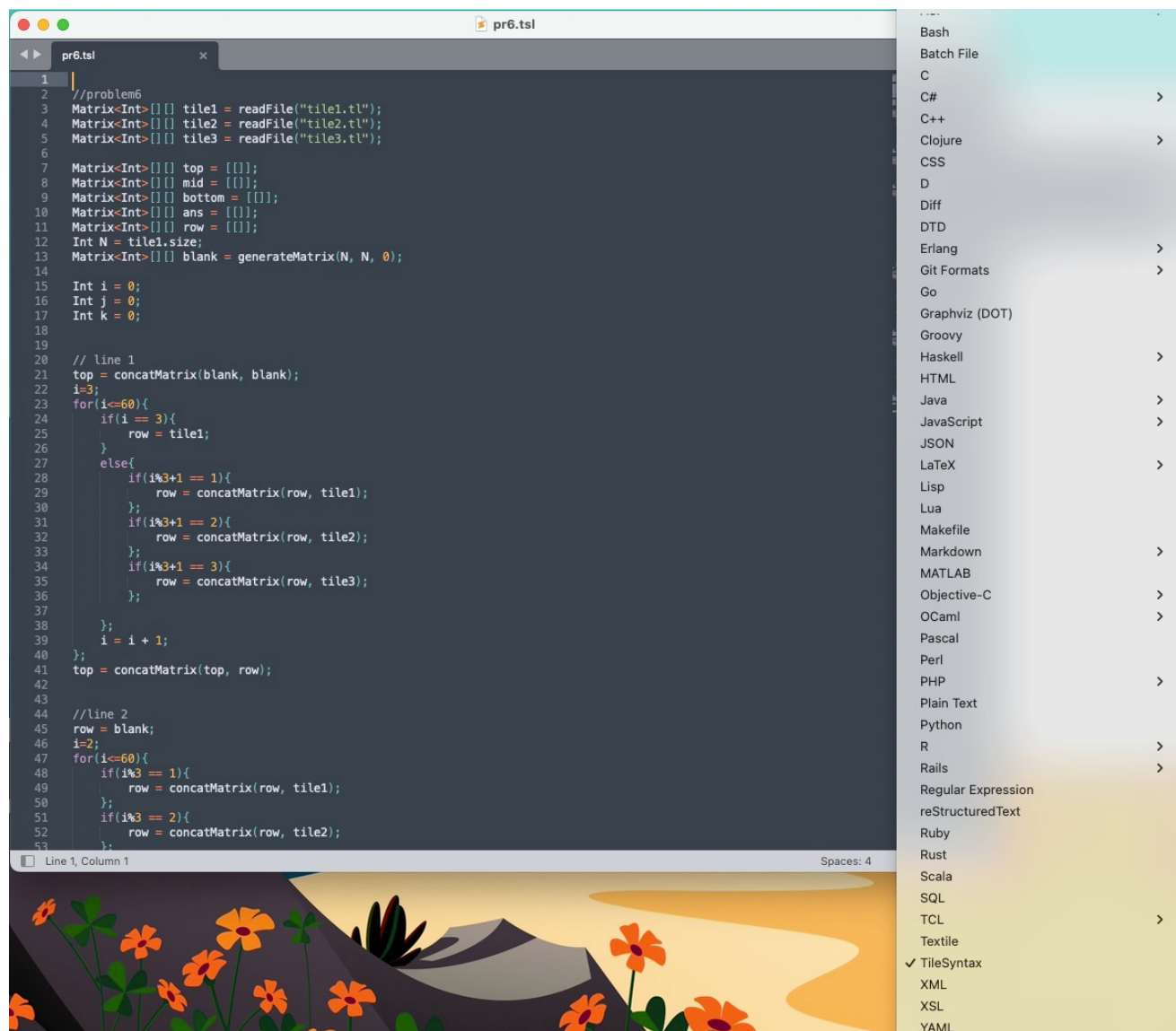
Int x = 1;

Parsed as TlSemi (TlDeclareAssign TyInt "x" (TlInt 1))

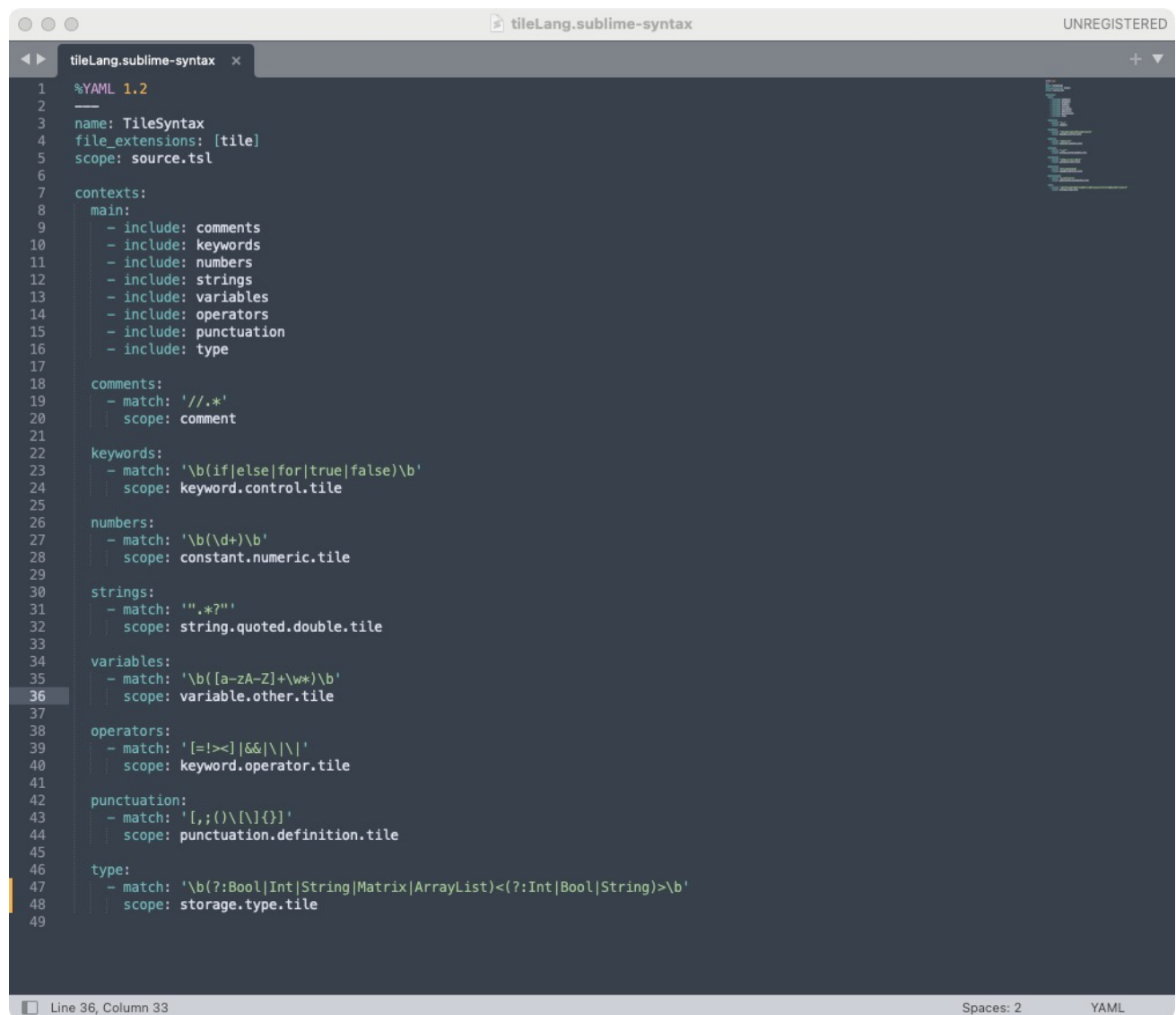
Type Checking : TlSemi (TlDeclareAssign TyInt "x" (TlInt 1))

Type Checking Passed with type no error

Syntax Highlighting:



Syntax Highlighting Configuration:



```
1 %YAML 1.2
2 ---
3 name: TileSyntax
4 file_extensions: [tile]
5 scope: source.tsl
6
7 contexts:
8   main:
9     - include: comments
10    - include: keywords
11    - include: numbers
12    - include: strings
13    - include: variables
14    - include: operators
15    - include: punctuation
16    - include: type
17
18   comments:
19     - match: '///.*'
20       scope: comment
21
22   keywords:
23     - match: '\b(if|else|for|true|false)\b'
24       scope: keyword.control.tile
25
26   numbers:
27     - match: '\b(\d+)\b'
28       scope: constant.numeric.tile
29
30   strings:
31     - match: '".*?"'
32       scope: string.quoted.double.tile
33
34   variables:
35     - match: '\b([a-zA-Z]+\w*)\b'
36     scope: variable.other.tile
37
38   operators:
39     - match: '[!=><]|&&|\||\|'
40       scope: keyword.operator.tile
41
42   punctuation:
43     - match: '[,;(){}{}]'
44       scope: punctuation.definition.tile
45
46   type:
47     - match: '\b(?:Bool|Int|String|Matrix|ArrayList)<(?:Int|Bool|String)>\b'
48     scope: storage.type.tile
49
```

Line 36, Column 33 Spaces: 2 YAML