Programming Language Concepts Report

Sauhaard Poudel (sp11g19), Brian Le (. . .)

1 Introduction to "TiLang"

TiLang is a domain specific language that has been developed in order to solve problems around *tiles*.

The language was developed in order to have a robust and intuitive syntax for efficient rotation, manipulation, and patterning of smaller input tiles to create larger tiles.

A *tile* is a square collection of discrete cells, each of which is either coloured or uncoloured (represented by 1's and 0's, respectively). The size of a tile is always square, with a tile of size N consisting of N x N cells. With its focus on tiling, TiLang provides a powerful tool for solving complex tiling problems quickly and effectively.

2 High Level Overview of our Language

Our language was designed to be as intuitive and simple to understand as possible. It is in the style of an imperative language but it borrows some helpful features that you would see in a functional language (such as Haskell.)

- Tiles can be imported from an external source or they can be created internally.
  - For example: `let tile = [x]`, where x is a list of row definitions, or by using `import "tile_name" as tile_name` to import a tile definition file and assign it a variable in the program.
- There are a number of operations you can perform on tiles, such as *rotation* (`~`), *scaling* (`**`), *reflecting horizontally* (`<>`). This list is not exhaustive.
- Standard mathematical operators are included, as well as comparison operators (such as `<` for *less than*) and boolean operators.
- Iteration in the form of `for` loops. Ranging across a set of values uses the `..` syntax (similar to Haskell).
  - The range function generates a list of integers between the starting and ending values (inclusive).
- Conditional statements in the form of `if else` blocks.
- Curly brace syntax to improve readability.
- Outputting the tile using the `output` function.
- It is a dynamically typed language, where the variable is only assigned at run time. This allows for greater flexibility.
- It is a strongly typed language because there is strict enforcement of the type system once the variable is assigned a datatype.

3 Implementation of the Language

We used **Alex** to produce our lexer generator, **Happy** to generate our parser, and **Haskell** to write our interpreter. We made an effort to use as few external tools as we could, so we stuck to the `Base` library in Haskell.

3.1 Formal Representation of the Language in Backus-Naur form

```
<Program> ::= <StatementOrExpr> <Program> | epsilon
<StatementOrExpr> ::= <VariableAssignment> | <ForLoop> | <IfStatement>
        | <ImportStatement> | <OutputStatement>
<OutputStatement> ::= "output" <Expression>
<ImportStatement> ::= "import" '"' <id> '"' "as" <id>
<VariableAssignment> ::= "let" <id> "=" <Expression> | <id> "=" <Expression>
<ForLoop> ::= "for" <id> "in" <Expression> ".." <Expression> <Block>
<IfStatement> ::= "if" <Expression> <Block>
        | "if" <Expression> <Block> "else" <Block>
<Block> ::= "{" <Program> "}"
<Expression> ::= <Expression> "&&" <Expression>
        | <Expression> "||" <Expression> | <Expression> "==" <Expression>
        | <Expression> "!=" <Expression> | <Expression> ">" <Expression>
        | <Expression> "<" <Expression> | <Expression> ">=" <Expression>
        | <Expression> "<=" <Expression> | <Expression> "+" <Expression>
        | <Expression> "-" <Expression> | <Expression> "*" <Expression>
        | <Expression> "/" <Expression> | <Expression> "%" <Expression>
        | <Expression> "++" <Expression> | <Expression> "::" <Expression>
        | <Expression> "~" <Expression> | <Expression> "**" <Expression>
        | <Expression> "&" <Expression> | <Expression> "|" <Expression>
        | <Expression> "@" "(" <Expression> "," <Expression> "," <Expression> ")"
        | "?" <Expression> | "<>" <Expression> | "^^" <Expression>
        | "#" <Expression> | "!" <Expression> | "(" <Expression> ")" | <id>
        | <int> | "true" | "false" | <TileDefinition>
<TileDefinition> ::= "[" <RowDefinitions> "]"
<RowDefinitions> ::= <RowDefinitions> <RowDefinition> | epsilon
<RowDefinition> ::= "[" <Ints> "]"
<Ints> ::= <Ints> <int> | <int>
```

3.2 Lexical generator (Alex)

We used **Alex** to produce our lexical analyser, and opted to use the *basic wrapper*. The basic wrapper is an interface that provides a simple API for scanning input text.

We split up our tokens (a sequence of characters that represent something) into 4 categories:

1. **Keyword:** This includes things such as `let`, `if`, and `import`.
2. **Operator:** These are symbols that are used to represent an *action* or *operation* performed on one or more variables or values.
   (a) Some example operators including `~` (rotate a tile), `==` (testing for equality) and `+` (addition).
3. **Literal:** These are pieces of data that appear directly in our programs that are not represented by a variable or expression, such as `true` and `false`.
4. **Identifier:** This is a string of characters that represent the name associated with specific components of the program (perhaps a variable).

3.3 Parser

`Happy` was our tool of choice in order to generate our parser. The structure of the language as laid out in Backus-Naur form guided us in the design of the `Happy` file.

3.3.1 Syntax Tree in Bracketed Notation

```
[Program [StatementOrExpr [VariableAssignment [id] [Expression] ]
[ForLoop [id] [Expression] [Expression] [Block] ] [IfStatement [Expression]
[Block] [Block] ] [ImportStatement [id] [id] ] [OutputStatement [Expression] ]
[Expression [TileOp] [CompareOp] [BoolOp] [BracketedExp] [Id] [Literals] ]] ]
```

Figure 1: Please note that the indentations and line breaks were made due to to page size constraints. Also, this syntax tree is a general structure of the language, and is therefore a simplified version.

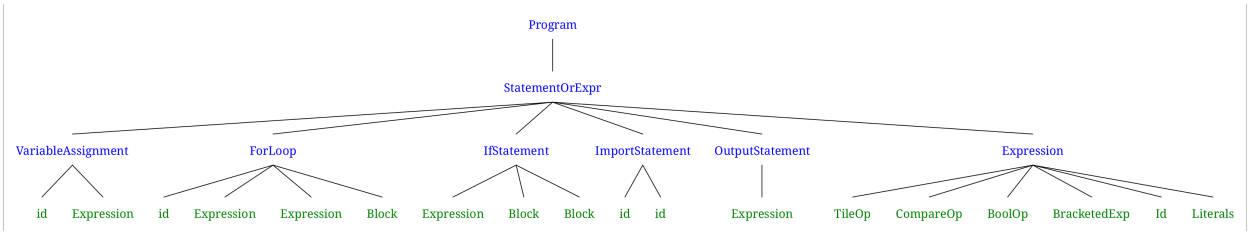

Figure 2: Graphical representation of the syntax tree. Please note that "StatementOrExpr" is misleading as a program cannot just be an expression. The name was kept like that to ensure compatibility.

3.4 Type Checker

The type system for **TiLang** revolves around three main types: `Int`, `Tile`, and `Booleans`. The type system ensures that expressions and statements conform to these types and that operations are performed only on compatible types. The language defines a set of operators with specific type requirements for their operands, which ensures the type safety and correctness of the DSL code.

`Int` represents integer values and is used with arithmetic operators such as addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulo (`%`). Comparison operators, like greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`), also expect integer operands and produce a boolean result.

`Tile` denotes tile patterns and is used with operators designed to manipulate and combine tiles. Tile-specific operators include horizontal join (`++`), vertical join (`::`), rotation (`~`), scaling (`**`), horizontal reflection (`<>`), vertical reflection (`^^`), blanking (`#`), tile-wise AND (`&`), tile-wise OR (`|`), tile-wise NOT (`?`), and snipping (`@`). These operators expect tile operands and return a tile result, except for the tile-wise logical operators (`&`, `|`, `?`), which return a boolean result.

`Booleans` represent boolean values and is used with logical operators such as AND (`&&`), OR (`||`), and NOT (`!`). Equality (`==`) and inequality (`!=`) operators are polymorphic and allow for comparison of values of the same type, but they return a boolean result.

The type checker ensures that all variables are declared before use, and their types are consistent throughout the code. It also enforces that loop indices, conditions in `if` statements, and other constructs adhere to the appropriate types. These rules ensure that a program written in our language will run in a predictable way.

3.5 Interpreter

Scoping in the interpreter is managed through the `Environment` data structure, which is a tuple of nested `Scopes` (lists of variable bindings) and a list of output strings. `Scopes` are organised as a stack, with the innermost scope at the top of the stack. `Scopes` are used to store variable bindings, and they ensure proper handling of variable visibility and lifetimes. When entering a new scope (such as a `for` loop or an `if` statement), a new, empty scope is pushed onto the stack of `Scopes`. When exiting a scope, the `restore` function is used to remove the topmost scope from the stack, which effectively discards all bindings at that scope.

Binding refers to the process of associating an identifier with a value (in the environment). The interpreter uses the `bind` function to search for a matching variable name up the scope chain and update its value. If the

variable is not found in the current scope, the function continues searching in the outer scopes. If a new variable is declared, the `bindCurrentScope` function is used to add a new binding to the current (innermost) scope. This allows for proper handling of variable assignments and declarations, ensuring that the appropriate scope is updated.

The interpreter makes use of informative error messages to provide the user with helpful feedback when an error occurs during execution. These error messages are generated using Haskell's error function, which produces a runtime error with a custom message. For example, in the `eval` function, there are various error messages for invalid operations on tiles, such as "`Cannot horizontally join tiles of different heights`" and "`Scale factor must be a positive integer`". These messages give the user clear information about what went wrong during the execution and how to fix the issue.

The execution model of the language is based on the transformation of the `environment` during runtime. The `environment` is comprised of `Scopes` (variable bindings) and a list of output strings. During the execution of the program, the interpreter traverses the AST, evaluates expressions, manipulates the environment, and generates output strings representing the tile patterns. The `execute` function processes statements (or code blocks), while the `executeStmt` function processes individual statements. The `eval` function evaluates expressions and returns the corresponding values.

As the interpreter processes the AST, it updates the environment by adding or updating variable bindings, pushing new scopes when entering nested constructs, and restoring previous scopes when exiting them. The output strings are generated by evaluating expressions and concatenating their string representations. At the end of the execution, the output strings represent the final result of the program (the output tile), which is then sent to standard output.