Programming Language Concepts Report

Sauhaard Poudel (sp11g19), Brian Le (dal1g21), Mitali Chavan (mc5g20)

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 collection of discrete cells, each of which is either coloured or uncoloured (represented by 1's and 0's, respectively). Whilst most use cases will involve square tiles, the language also supports rectangular tiles. 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 dynamically.
  - For example: `let tile = x`, where x is a list of row definitions, or an expression evaluating to a tile.
- 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 operator 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.
- Statically typed, meaning programs written in TiLang will be checked for types before interpreting.
- Strongly typed, meaning there is strict enforcement of the type system once the variable is assigned a datatype, which helps prevent a class of runtime exceptions.

3 Implementation of the Language

We used **Alex** to produce our lexer, **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>
```

Figure 1: Please note: Despite the naming for rule `StatementOrExpr`, it only consists of rules that define statements, not expressions.

3.2 Lexical analysis (Alex)

We used **Alex** to produce our lexer, 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 include `~` (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] ] ] ]
```

Figure 2: Please note that the indentations and line breaks were made due to page size constraints. Also, this syntax tree is a general structure of the language, and is therefore a simplified version.

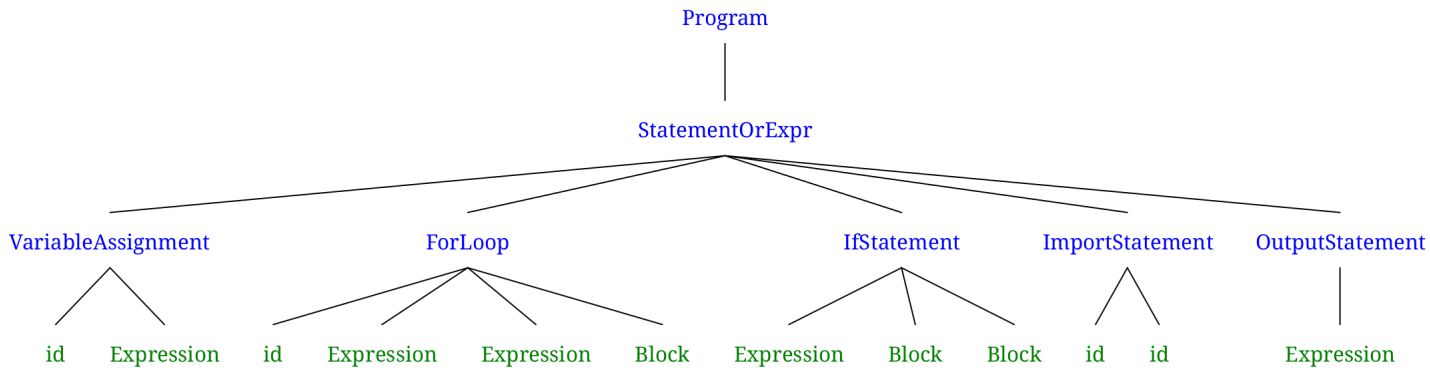

Figure 3: Graphical representation of the syntax tree. Please note that "StatementOrExpr" is misleading as a program cannot just be an expression. The name was kept to ensure compatibility.

3.4 Type Checker

The type system for **TiLang** revolves around three main types: `Int`, `Tile`, and `Bool`. 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 (`&`, `|`, `?`), and certain transformation operators (`~`, `**`, `@`), which return a boolean result and accept integer operands on the RHS, respectively.

`Bool` 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.

Whilst being a statically and strongly typed language, TiLang only supports implicit type inference. Once a variable is assigned, TiLang automatically infers the most appropriate type for the variable and enforces that it is consistent throughout the program.

In addition, TiLang type system makes sure that all variables are declared before use, and that no shadowing happens (when a variable declared in a innermost scope overwrites one declared in outer scopes). It also verifies 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

The interpreter manages runtime state through the `Environment` data structure, which keeps a stack of =Scope=(s), which are lists of variable bindings, and a list of output strings. Its execution model is based on transformation of this data structure, feeding a resulted environment after executing a statement to the next one, optionally modifying scopes in between.

The list of output strings is appended to when an `OutputStatement` is executed. Once all statements in the program are executed, these strings are printed to `stdout`.

Whenever a block is defined, either via a `IfStatement` or a `ForLoop`, the interpreter creates an empty `Scope` and push it onto the stack of scopes. A variable declaration is immediately add to this new scope, whilst variable assignments and lookups move down the scope stack to find the appropriate bindings. After exiting a block, the Interpreter immediately pops the newly created scope off the stack to make sure scoped variables are no longer accessible.

The interpreter resolves imports statically. Before execution, it scans the AST for `ImportStatement` and build a list of filenames to be read. It then reads and parses file contents and feed the parsed tile values to the execution functions. Tile values are represented using Haskell lists under the hood for maximum simplicity, flexibility and performance.

Moreover, TiLang attempts to validates operands of certain operators to minimise runtime errors. For example, the horizontal and vertical join operators (`++`, `::`) are checked to ensure the left and right operands are of the appropriate dimensions before joining. The detected errors are informative to help programmers pinpoint exactly what went wrong.