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).
- Conditional statements in the form of `if else` blocks.
- Curly brace syntax to improve readability.
- Outputting the tile using the `output` function.
- Type checking. . .
- Strongly or weakly typed. . .

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 little 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

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.

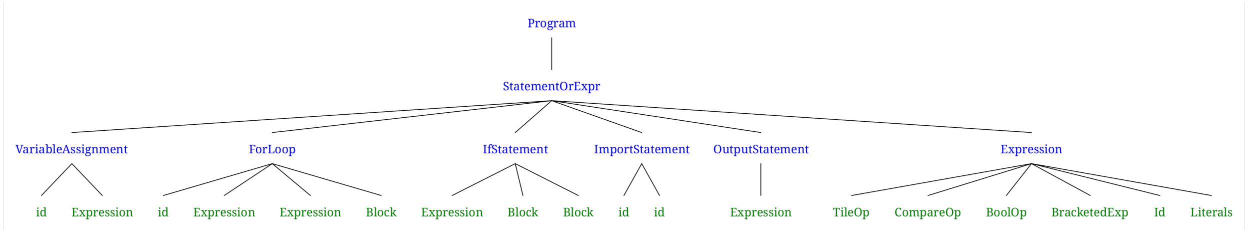

Figure 2: Graphical representation of the syntax tree. Please note that “StatementOrExpr” is misleading as a program cannot just be an expression. The name was kept like that to ensure compatibility.

3.4 Type checker

TypeEnv: A type environment, represented as an association list of variable names and their corresponding types.

VarType: A data type defining the types supported by the DSL: IntType, TileType, BoolType, and Undefined.

unparseType: A function to convert a VarType into a string representation.

verify: The main function to be called with a list of statements in the DSL. It returns a list of errors in the code after running the type checker.

verifyBlock: A function that takes a type environment and a list of statements, and returns an updated type environment while reporting any errors through a Writer monad.

getBinding, addBinding: Functions to retrieve and add variable bindings to the type environment, respectively.

verifyStmt: A function that pattern matches on each statement type and verifies the statement according to the typing rules of the DSL. It returns an updated type environment.

typeof: A function that computes the type of an expression in the DSL, returning a VarType wrapped in a Writer monad.

assertOperands, assertSingleOperand: Helper functions that check the types of operands for binary and unary operators, respectively, and report type errors if needed.