Programming Language Concepts Report

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

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.

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.
- 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…

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.

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>
```

Lexical scope

Type checking