PLC Group Project - Tile Language and Interpreter

Tom Evans - tee1g21@soton.ac.uk
Lia Jeffries - lvj1g20@soton.ac.uk
William Percy - wip1g21@soton.ac.uk

May 2023

Contents

1 Language Description and Explanation

This section describes and explains our language structure and behaviour, and the process of how we designed it.

1.1 Design Process

After reading the project specification we decided to consider the problems our language would have to solve, before designing any language syntax. We read each set problem and designed a solution for each one in pseudo-code. This allowed us to get an understanding of functions that would be vital for our language. After we had the list of functions and constructs that our language must include (such as a place function and for loop), we decided that we would try to make our language's syntax resemble our pseudo-code. We decided that because our language will be used for a relatively simple task (tile pattern creation), it was suitable to have a language with quite simple and intuitive syntax & constructs. For example, we decided that variable types would not have to be declared with the variable name and that the place function would automatically place a tile in the next correct position. However, to aid parsing and prevent ambiguity, we decided that expressions must end in an ';' and certain blocks of code would be surrounded by '{' and '}' where necessary (i.e. for loops, repeats, if statements, build blocks, etc.).

1.2 Our Language

This is our Grammar:

```
%nonassoc if
%nonassoc else
%nonassoc '<' '>'
%nonassoc COMP
%left AND OR
%right NOT
%right '='
%left '+' '-'
%left '*' '/' '%'
%left NEG
%left ','
%left ';'
%left UVAR
%%
Init   : input Inps ';' Exp                                  {Input $2 $4}
       | Exp                                                 {TExp $1}
Exp    : place Tile                                          {Place $2}
       | newLine                                             {NewLine}
       | repeat '(' Num ')' '{' Exp '}'                      {Repeat $3 $6}
       | for '(' var ',' Num ',' Num ')' '{' Exp '}'         {For $3 $5 $7 $10}
       | if  '(' Bool ')' '{' Exp '}' else '{' Exp '}'       {IfElse $3 $6 $10}
       | if  '(' Bool ')' '{' Exp '}'                        {If $3 $6}
       | var '=' Vars                                        {Assign $1 $3}
       | Exp ';' Exp                                         {Seq $1 $3}
       | Exp ';'                                             {$1}
Tile   : rotate '(' Num ',' Tile ')'                         {Rotate $3 $5}
       | scale '(' Num ',' Tile ')'                          {Scale $3 $5}
       | reflectX '(' Tile ')'                               {ReflectX $3}
       | reflectY '(' Tile ')'                               {ReflectY $3}
       | conjugate '(' Tile ',' Tile ')'                     {Conjugate $3 $5}
       | negate '(' Tile ')'                                 {Negate $3}
       | subtile '(' Tile ',' Num ',' Num ',' Num ')'        {Subtile $3 $5 $7 $9}
       | fill '(' Colour ',' Num ')'                         {Fill $3 $5}
       | build '{' Exp '}'                                   {Build $3 }
       | var                                                 {TileVar $1}
       | '(' Tile ')'                                        {$2}
Num    : size '(' Tile ')'                                   {Size $3}
       | Num '+' Num                                         {Plus $1 $3}
       | Num '-' Num                                         {Subtract $1 $3}
       | Num '*' Num                                         {Multiply $1 $3}
       | Num '/' Num                                         {IntDiv $1 $3}
       | Num '%' Num                                         {Mod $1 $3}
       | '-' Num %prec NEG                                   {Negative $2}
       | int                                                 {Int $1}
       | var                                                 {NumVar $1}
       | '(' Num ')'                                         {$2}
Bool   : true                                                {TTrue}
       | false                                               {TFalse}
       | Bool AND Bool                                       {And $1 $3}
       | Bool OR Bool                                        {Or $1 $3}
       | NOT Bool                                            {Not $2}
       | Num '<' Num                                         {LessThan $1 $3}
       | Num '>' Num                                         {GreaterThan $1 $3}
       | Num '<' '=' Num %prec COMP                          {LessEqThan $1 $4}
       | Num '>' '=' Num %prec COMP                          {GreaterEqThan $1 $4}
       | var                                                 {BoolVar $1}
       | Vars '=' '=' Vars %prec COMP                        {EqualTo $1 $4}
       | '(' Bool ')'                                        {$2}
Vars   : Num                                                 {Num $1}
       | Bool                                                {Bool $1}
       | Tile                                                {Tile $1}
       | '~' var                                             {UVar $2}
Inps   : var                                                 {F $1}
       | var ',' Inps                                        {X $1 $3}
Colour : white                                               {White}
       | black                                               {Black}
```

As can be seen in the grammar, inputted files must be defined at the start of the program, followed by a main expression. These expressions contain different types that are defined below. This is to make sure only the right data types can be called in specific functions. For example, the condition in an if statement must reduce to a Boolean.

There are five different "Types" for storing information in the program environment that are dictated by the grammar to remove the need to code type checking within our Haskell interpreter files and also to make the user experience of the language easier as defining variable type is not required (also a syntax sugar).

- **Nums** contain any expression that can be reduced to an Int. They can undergo mathematical operations (i.e. addition, subtraction, multiplication, integer division, etc.).
- **Tiles** contain any expression that can be reduced to a tile (String). Tiles are used in many functions such as, scaling, subtitling, conjugation (AND) and negation (NOT), etc.
- **Bools** contain any expression that can be reduced to a Boolean. This type can have all standard Boolean operations applied to it, including AND, OR, NOT and comparisons (such as greater than or equal to).
- **UVAR** are unassigned variables, created in the grammar to prevent redundant rules and ambiguity. When assigning the value of a variable to another variable or using the '==' operation on 2 variables, the type of these variables is not clear when parsing, so they are set as unassigned. Then, the interpreter looks up these variables in the environment and defines their type.
- **Colours** simply store colours 'black' or 'white'. Having this as a separate type means that fill commands will only be parsed correctly with these colour names and it would be easy to add new colours in the future.

Within the grammar, if a function returns a type, this function is treated as that type (similarly to Haskell), allowing functions with acceptable return types to be used as parameters of other functions. For example, "reflectX(reflectY(tile))" or "scale(size(tile), tile2)". Exp's do not return a type as they each have unique behaviours such as place or newLine.

We added associativity to certain operations within our grammar to reduce shift and reduction errors:

- if, else, and all comparison operations are non-associative
- AND, OR, +, -, *, /, % (Modulo), NEG, ',', ';' and UVARs are all left associative
- NOT and = are both right-associative

1.3 Constructs & Functions

- **Input** takes in one or more tiles within the same directory our program is run from. The name of the tile file (without the extension) will then be the variable name for that tile. For example, "Input tile1;" will look for and read tile1.tl into a tile variable "tile1" in our program environment. All inputs must be done on 1 line (as seen in the grammar) however many input filenames can be listed in the command (i.e. input t1, t2, t3, t4;).
- **Place** adds a Tile into the programs grid in the next available position, this being the top-most left-most available line (usually end of the previous line unless has been closed using newLine).
- **NewLine** when called makes it not possible to place tiles in the lines of the previously placed object, this effectively moves the current position to be on the line under where the last tile was placed.
- **Repeat** takes a Num as an argument and repeats the statements in the block for that integer amount of times. There is no accessible index in this function.
- **For** acts the same as a repeat in that it executes statements from a start-to-finish index, but it stores the current count in a variable accessible within the block, allowing it to be manipulated or used in calculations.
- The **if/else** constructs (If-Else and If) compare two conditions to ascertain a true or false result, and conditionally execute a block. In an if statement, the else is optional and not required.
- **Rotate** takes a tile and rotates it by x degrees clockwise, where x is a multiple of 90. The rotate function will output an error caught by the main module if a non-multiple of 90 is submitted as an argument.
- **Scale** takes a Num (x) and a tile and scales the tile up by x times to then be returned. x has to be an integer, and the tile doesn't have to be a square
- **ReflectX** and **ReflectY** both take in a tile, ReflectX reflects the tile through the x-axis while ReflectY reflects the tile through the y-axis.
- **Conjugate** takes in two equally sized tiles and performs an AND operation on each of the two tiles elements, this gives us a single new tile which is of the same size as the two inputted.
- **Negate** takes in a single tile and performs a NOT operation on each of the tiles elements returning a new tile of the same size as the inputted ones.
- **Subtile** takes a given tile and from the inputted location extracts a square section of the total tile of the given size. If the size inputted is greater than the size of the tile then the full tile is returned (for example subtitling at size 5 with 3x3 tile will return the same 3x3 tile).
- **Fill** takes a Colour and a size as a Num and generates an NxN sized tile. The white colour uses all zeroes and the black colour uses all ones.
- **Build** is a function which is used to create larger, more complex tiles using smaller tiles and applying operations to them from our language. Builds have blocks of codes containing expressions and the grid which is created from this block is returned as a tile variable.

1.4 Scope

We have implemented scope for variables within our language, this is mainly for blocks of code inside build blocks. All variables declared inside build blocks are local to that build and cannot be accessed outside of it. However, variables declared outside of the build can be accessed within the build block, but changing them here will not affect their value outside of this space. Variables not inside any build block are considered global and also, the indexes of for loops are local to that for loop block.

1.5 Syntax Sugars

Our language includes many syntax sugars to improve its readability and to make it more intuitive (easier to code in). These include:

- **( )** are used in constructs and functions when arguments are required (these inputs are types like Tiles or Numbers), or to separate expressions to state the order of execution (BIDMAS). They also make code more readable.
- **{ }** are used to denote blocks of code (like inside for loops or for places inside builds) and to clearly separate them from others visually.
- **Commas** are used to separate arguments to a construct or function and are also used to list inputs to our language.
- **Semi-Colons** are used to define the end of an expression (exp). This allows the user to understand exactly when a block of code ends and another begins (and aids parsing massively).
- Our language also **ignores white space** entirely and thus is non-strict on indentation. This allows the user to decide how their programs are formatted while still applying the correct grammar.
- **Function nesting** is also part of out language. This allows functions to be used within other functions, making code more concise, readable and powerful.
- **Comments** allow code in programs to be explained and thus made easier to understand for those who didn't write it, making code more readable.
- **Implied typing** is used so that variable types do not have to be defined when the variable is declared, this improves the language's ease of use and simplicity.

This is an example program written in our language (pr7) containing all the syntax sugars listed.

```
-- Gets and stores the input tiles.
input tile1;
-- Generates the padding used in many of the generated tiles rows.
rowPadding = build {
    repeat ((size(tile1) * 100)) {
        place fill(white, 1);
    };
};
-- Generates the padding used in many of the generated tiles columns.
columnPadding = build {
    repeat ((size(tile1) * 100)) {
        place fill(white, 1);
        newLine;
    };
};
-- Builds the final tile of the program which will be placed in the grid.
tileQ = build {
    -- Initialises the accumulator with a 100x scale.
    tileAcc = build { place scale(100, tile1) };
    -- Loop which generates and XOR's all the other scale tiles.
    for(i, 1, 49){
        -- Calculates the current scaling of the tile, then scales it.
        N = (100 - (2 * i));
        tileN = scale(N, tile1);
        -- Generates the current padding based on the size of the scaled tile.
        currentColumnPadding = subtile(columnPadding, 0, 0, size(tileN));
        -- Builds a new tile with the scaled version of the inputed tile so it can be XORed.
        tileNext = build {
            -- Adds upper padding to the tile.
            repeat (2 * i * size(tile1)) {
                place rowPadding;
                newLine;
            };
            -- Adds the left padding to the tile.
            repeat (2 * i * size(tile1)) {
                place currentColumnPadding;
            };
            -- Adds the newly scaled tile to the grid
            place tileN;
        };
        -- Performs an XOR operation on the accumulator and newly generated tiles.
        tileNAND = negate(conjugate(tileAcc,tileNext));
        tileAccNAND = negate(conjugate(tileAcc,tileNAND));
        tileNextNAND = negate(conjugate(tileNext,tileNAND));
        tileXOR = negate(conjugate(tileAccNAND,tileNextNAND));
        -- Updates accumulator to be equivalent to the new tile.
        tileAcc = ~ tileXOR;
    };
    -- Places the final generated tile into the builds grid.
    place tileAcc;
};
-- Places the final tiles into the main grid as described in the spec.
place tileQ;
place (reflectY(tileQ));
newLine;
place (reflectX(tileQ));
place (reflectY(reflectX (tileQ)));
```

1.6 Type Checking

Type checking exists in our language's grammar. The first way it does this is to make sure constructs and functions can only accept specific types due to the way they are defined in the grammar, this means that if wrong types are used it will cause a parse error and thus removes the need for our interpreter to have inbuilt type checking unless in one instance. That instance is when unassigned variables are parsed, because it's a variable the interpreter would be able to look up said variable and map it to its correct type and value.

2 Evaluation

This section explains the process of taking a raw program in our program, lexing and parsing it, and then interpreting the parsed program to stdout.

2.1 Lexer

We used Alex to lex files written in our code. We wrote a Tokens.x file which defined our tokens in the correct format for an Alex file, this can then be compiled in the terminal using Alex to create a Haskell file to be imported into our interpreter. Our lexer uses the 'posn' wrapper that's built into Alex, this means that each occurrence of a token is paired with its position in the file being lexed. We did this to aid debugging massively when writing the parser and interpreter, as well as to be used as part of certain error messages in the interpreter. These are some of our tokens:

```
-- digits
    $digit = 0-9

    -- alpha chars
    $alpha = [a-zA-Z]

    tokens :-
      $white+ ;
      "--".* ;
      input { \p s -> TokenInput p }
      place { \p s -> TokenPlace p }
      newLine { \p s -> TokenNewLine p }
      repeat { \p s -> TokenRepeat p }
      for { \p s -> TokenFor p }
      if { \p s -> TokenIf p }
      else { \p s -> TokenElse p }
      rotate { \p s -> TokenRotate p }
      scale { \p s -> TokenScale p }
      size { \p s -> TokenSize p }
      reflectX { \p s -> TokenReflectX p }
      ...
```

As seen above, white space is removed, '–' indicates comments and is therefore removed as well. We create variables digit and alpha to help writing the regex for some tokens (TokenVar, TokenInt, etc). The lexer also has a Haskell data type to store tokens:

```
data Token =
      TokenInput AlexPosn |
      TokenPlace AlexPosn |
      TokenNewLine AlexPosn |
      TokenRepeat AlexPosn |
      TokenFor AlexPosn |
      TokenIf AlexPosn |
      TokenElse AlexPosn |
      TokenRotate AlexPosn
      ...
```

And a toknPosn function which returns the position of a token given as a parameter:

```
tokenPosn (TokenInput (AlexPn _ x y)) = show x ++ ":" ++ show y
tokenPosn (TokenPlace (AlexPn _ x y)) = show x ++ ":" ++ show y
tokenPosn (TokenNewLine (AlexPn _ x y)) = show x ++ ":" ++ show y
tokenPosn (TokenRepeat (AlexPn _ x y)) = show x ++ ":" ++ show y
tokenPosn (TokenFor (AlexPn _ x y)) = show x ++ ":" ++ show y
tokenPosn (TokenIf (AlexPn _ x y)) = show x ++ ":" ++ show y
tokenPosn (TokenElse (AlexPn _ x y)) = show x ++ ":" ++ show y
...
```

2.2 Parser

We used Happy to parse files of our program. We wrote a Grammar.y file, which defined the grammar of our language. This was also compiled in the terminal using Happy to create a Haskell file used in the interpreter. The parser imports the token file so that the grammar can be created from them. The BNF Grammar, associativity of the language and Haskell data type - to store syntax trees of parsed programs - are all defined inside the parser file (these are explained in later sections). The parser also has a parseError function, which returns the following error message after a parse failure:

```
error ("Parse error at line:column " ++ (tokenPosn t) ++ " - " ++ (show t))
```

2.3 Interpreter

Our interpreter source code is written in the file Tsl.hs and can be used to interpret a program once it is compiled into an executable; it takes one parameter, which is the program in our language, to be parsed.

The main module will run when the interpreter is executed in the terminal. Firstly, it will try to read the program file using getArgs. If the argument has an incorrect file extension (not '.tsl'), an error stating "Invalid File Type" will be thrown. If not, the sourceText will be read into a variable. The main function then uses the Happy function 'parseCalc' to parse the text into an abstract syntax tree of our language. If the program fails to parse, another error message will be thrown stating "Invalid Input File:" + the specific parse Error. After this, we decided that all programs must start with file inputs, so the interpreter tries to retrieve the input filenames. If this fails, an error message: 'No Input Tiles' is thrown. If successful, the contents of the files corresponding to these filenames are then read into a list of strings and added to the initial environment of the interpreter - alongside an empty 'grid' string and an empty string 'previousTile'. The main function responsible for interpretation: "interpret :: TExp -> Env -> Env" is then called to obtain the final environment after reducing the syntax tree. The grid variable is then extracted from this environment, ready to be displayed. Before it is displayed, all end markers 'x' (part of newLine functionality) and new-line chars '\n' have to be stripped from the grid; it is then output as stdout using putStr. Also, all error messages are dealt with inside of the main function and are all of type stderr; all error messages thrown inside the interpret functions will be caught by the function noParse, and a suitable error message is thrown: 'Invalid Input File' + caught error message.

After our main module, the environment data Type 'Env' is defined. In our interpreter, an Environment is a list of variables in the form of tuples, with each tuple containing a String (variable name) and a Var: Var = Grids String | Tiles String | Nums Int |

Bools Bool | Helper String. Var contains all the type names variables can be as well as the Haskell type in which these variables will be stored as in the interpreter, meaning that all variables inside of environments will be of these types. We then define functions which allow us to manipulate environments. The function lookupVar is used to find and return variables; it returns Maybe Var as the variable being searched may not exist. If the variable does not exist, an error message "Variable is Empty!" will be caught by the main module. Another function is updateVar, this allows variables to be added and edited in the Env, if the variable given to the function is not in the Env it will be added. If it is, its value will be updated. We also wrote a deleteVar function to remove variables from the environment.

The 'interpret' function is responsible for reducing a TExp Haskell type (corresponding to exp in the grammar) to an Environment with a final grid variable ready to be displayed. The function takes an Environment ('Env') as a parameter and also returns an Environment, as the latest version of the program environment needs to be parsed and returned through recursive calls of this interpret function. However, several interpret functions are required as (due to type checking built within our grammar) the Syntax tree will not always be of type TExp; therefore, 'interpretTile :: TTile -> Env -> String', 'interpretNum :: TNum -> Env -> Int' and 'interpretBool :: TBool -> Env -> Bool' are all required to certain types their reduced values (using operations that deal with that type). All interpret functions work by pattern-matching the current node of the syntax tree and calling helper functions to reduce that node. For example, "interpret (Repeat n exp) env = iRepeat (interpretNum n env) 1 exp env" uses the recursive helper function iRepeat to simulate a repeat loop within our language.

Below are explanations of how key aspects of our language are programmed within the interpreter, some but not all use specific helper functions:

- **Variable Assignment** is done using pattern matching within our interpret function:

```
interpret (Assign name (Num tNum)) env = updateVar env name (Nums (interpretNum tNum env))
interpret (Assign name (Tile tTile)) env = updateVar env name (Tiles (interpretTile tTile env))
interpret (Assign name (Bool tBool)) env = updateVar env name (Bools (interpretBool tBool env))
interpret (Assign leftVar (UVar rightVar)) env = updateVar env leftVar var
    where var = extractValue (lookupVar env rightVar)
```

  As can be seen, the name of the new variable created is added to the Env with the value of the reduced expression sent to the interpret function. The type of the variable is also defined in the syntax tree (also seen above) unless it is an unassigned variable. If an unassigned variable (name) is being assigned to a new variable, it means that the value of that unassigned variable can be looked up in the Env and then set as a new variable with the correct name.
- **If/IfElse** functionality does not require functions within the interpreter. The condition of the statement is reduced to a Boolean value using interpretBool, and guards are then used to interpret the TExp if the condition is true. If there is an else statement, the else TExp will be interpreted if the condition reduces to false.
- **iRepeat** is a recursive function which deals with repeat loops within our language which do not have an accessible index value. The function takes an Env and TExp and interprets the TExp 'n' many times, updating the Env each time.
- **iFor** is extremely similar to iRepeat, however, the index variable is added to the environment when the function is called (and removed after making it local to the for). Function iFor also has a start and end index, unlike iRepeat.
- **iPlace** takes a tile variable and an environment grid (populated or not) and places the tile on the top-left-most free spot. If a line has an end marker (added by the NewLine function) then the Place function will look at the line below to place the tile. If all populated lines are full, place starts on a new line. Once the start location for the tile has been found, the place function appends each line of the tile to the respective line of the grid.
- **iNewLine** takes the size of the last placed tile and appends an end-marker 'x' to that many lines in the current environment, which tells the place function that those lines are full and no further tiles can be placed to the right.
- **iRotate** takes a tile and rotates it by x degrees, where x is a multiple of 90. The rotate function makes use of Haskell's transpose and reverse functions to reorganise the lines in the tile.
- **iScale** takes a Num (x) and a tile and enlarges the tile by a scale factor of x. This is achieved by duplicating each character in the tile x times ignoring newline characters, and then duplicating each line in the tile x times.
- **iReflectX** and **iReflectY** call the same function in the interpreter. ReflectX reverses the order of the lines in the tile string using Haskell's reverse function. ReflectY reverses the order of the characters in each line of the tile using 'map' with the reverse function.
- **iConjugate** is an iterative function that goes through every element in two tiles, comparing them with an AND check, this generates a tile which is the AND of both inputs. As the function is an AND, both tiles must be the same size.
- **iNegate** is an iterative function that flips every element of a tile to the opposite colour (inverses 1s and 0s), performing a NOT operation on the tile.
- **iSubtile** uses mapping and an iterative function to strip unneeded sections from the inputted tile and then combine the leftover parts of the tile back together to give us a valid tile. This tile will be, at most, a size of NxN (where N is the given size by the user). If N is greater than the size of the tile, the full tile is returned.
- **iFill** takes a Colour and a size as a Num and generates an N by N sized tile using replicate. The white colour uses all 0s, and the black colour uses all 1s.
- **iBuild** takes a build block of code as a TExp and a copy of the current environment with the current 'grid' reset to an empty string. The main interpret function is then called on this expression creating a new grid based on the operations within the build block. The function then returns this grid as a tile variable to be used in the rest of the program.

There are also more helper functions used throughout the interpreter to complete small but vital tasks, and make the code easier to manage.

We believe that our interpreter resembles and shares principles with a **CEK** finite state machine. With the **C**ontrol being the current node of the tree being pattern matched; the **E**nvironment being our literal environment which stores variables and is updated during each state of our machine; and the **K**ontinuation being the stack of expressions built up using recursion of our interpret functions.