Please, Let him Cook
Domain-specific programming language coursework

COMP2212 Programming Language Concepts

Karim Elsayed
email kwsa1g20@soton.ac.uk

Lukas Kakogiannos
email lk1u20@soton.ac.uk

Jiechen Li
email jl14u21@soton.ac.uk

May 2, 2023
v1.0

1 Introduction

The aim of this coursework is to design and implement a unique domain-specific programming language for specifying tiling patterns. When designing our language, inspiration was taken from NumPy's array manipulation routines and from Java's syntax; the language includes predefined functions that allow the user to manipulate tiles in any way they desire. Additionally, all variables have global scope, thus, removing the need for multiple environments. Furthermore, the use of lambda calculus seemed impractical for tile manipulation, therefore it was not included in the language. In order to implement our language, we have developed a lexer, a parser, a type-checker, and an interpreter.

2 Design

2.1 Lexer

When designing the lexer, after studying and examining both the 'basic' and the 'posn' wrappers, we opted for the 'posn' wrapper since it gives us the capability of producing more meaningful errors. Moreover, taking inspiration from Haskell, the '−−' token with any characters that follow it was added to the lexer, allowing it to be used for commenting. Additionally, when deciding what types would be included in the language, we settled on the inclusion of two primitive types 'Int' and 'Bool', and two complex types 'Tile' and 'TileGroup'.

To make the grammar stricter, and to enforce type safety, we've opted for using a naming convention in variables. Normal variables can be anything, including numbers, booleans, and tiles. However, only the variables that are wrapped in square brackets can be assigned with the 'TileGroup' type (e.g. '[' var ']').

Furthermore, we decided to add simple math operations, such as addition, subtraction, multiplication, and division. We also added boolean operations, such as equality (cannot be used for tiles), less than, "and", "or", and "not". Inspired by Java syntax, we incorporated commas, semicolons, parentheses, square and curly brackets, conditional "if-else" statements, and a for-loop.

Finally, taking inspiration from NumPy's array manipulation routines, we included a few built-in functions in the language, such as: "import", "print", "create", "blank", "flipy", "flipx", "rotate", "subtile", "conj", "neg", "supersize", "stack", "join", "numCol", and "numRow".

2.2 Parser

The parser is where we match those tokens together into an AST (Abstract Structure Tree) so we can give them meaning later in the interpreter. When designing our AST, we decided to make the program always end with the 'print' function. We also decided that if-statements would always have to include an 'else' clause and that they take strictly a single command. Additionally, we added that any line written inside a for-loop should start with '#', to facilitate parsing. We also set some specified bindings for the following operators: '*' and '/' bind more tightly than everything else, '+' and '-' come next in order of how tightly they bind, followed by '||' and '&&', these are all left associative. Following these, are '<' and '==' and '=', which are non associative. Lastly, 'for' has the least binding and is right-associative. Moreover, we have defined two data types, one called 'Exp' which has all the expressions, used in the type checker and in the interpreter, and a data type called 'Type' to be used only in the type-checker.

2.3 Type Checker

Just like most programming languages, our language also utilises a notion of type for our programs. Specifically, our language uses "Strong Static" typing. Strong because all types are inferred in their usage in functions, and static because the types are checked during compilation-before the program executes. The type checker will prevent most run-time errors that might have proven to be costly otherwise. We drew inspiration from Haskell's powerful type inference system to allow our programming language to determine the type of variables based on the value they hold at compilation.

With this approach, we have made variable declaration more flexible and streamlined, while also enhancing the clarity and readability of code. It is important to note that all of our type rules culminate in the use of a unified data type known as "Type". This encompassing data type includes a variety of types such as 'TGType', 'TileType', 'RowType', 'IntType', 'BoolType', and 'BlockType'. Additionally, some types such as 'FunctionType', 'PairType', 'HashType', 'HashList', and 'SemiList' can take other types as input, adding a high degree of flexibility to our programming language.

2.4 Interpreter

Firstly, when designing the interpreter, we had to decide which operational semantics approach to take, after a thorough examination of many different approaches, we decided to go for structural operational semantics as opposed to reduction semantics or Big-step semantics. The reasoning behind this choice is that structural operational semantics has a higher expressive power and offers a thorough description of a program's internal state as it executes, allowing for more fine-grained analysis and reasoning about program behavior. It also provides a more

modular and composable approach, allowing for the definition of small, reusable transition rules that can be combined to define the behavior of larger programs.

Moreover, we decided to implement a CEK machine in the interpreter, which involved setting up an environment that stores all the variables and their values, frames that store the "rest" of the program, a 'Kontinuation' FIFO (First-In-First-Out) stack that stores all the frames, and lastly, configurations that store the expression being currently calculated, the current environment, and the current 'Kontinuation' stack. Lastly, instead of using closures, we decided to use the "print" function to end the program instead.

3 Implementation

3.1 Lexer

In order to implement the lexer, we created the tokens in a file called Tokens.x and generated the corresponding Haskell file by using the Alex command. We have included basic tokens for simple math operations such as '+', '-', '*', '=', '<', and '/', representing addition, subtraction, multiplication, equality, less than, and division, respectively. Additionally, we have included the following tokens: '(', ')', '[', ']', '{', '}', ',', and ';'.

For conditional statements, we have tokens for an "if-else-then" statement. We have also defined tokens for the aforementioned built-in functions that the lexer will tokenise according to their actual names. We have also included a token for the "for" loop and three Boolean operators, "||", "&&", as well as "not", representing "and", "or", and "not", respectively.

Lastly, we have included a 'token_posn' function for error printing that takes any token and prints its position in the code and the token itself. For example, for "TokenOPBracket", it would print "line:column -> (", which indicates that a parsing error was encountered just before that token. Below is an example of what the function looks like when pattern matching with the tokens "TokenDigit" and "TokenMinus". These tokens were all included in a single 'TokenType' data type, used in the parser.

```
1 token_posn (TokenDigit (AlexPn a l c) d) = show(l) ++ ":" ++ show(c) ++ " -> " ++ "digit " ++
      show d
2 token_posn (TokenMinus (AlexPn a l c) ) = show(l) ++ ":" ++ show(c) ++ " -> " ++ "-"
```

Listing 1: token_posn function example

3.2 Parser

There were a few important decisions taken during this stage of the implementation, when deciding the structure of programs written in our language, we also decided to include imports at the beginning of the programs for tiles. Below is the entry point to the AST and its ending.

```
Start : import var ';' Start    {Imports $2 $4}
      | End                     {$1}

End : Ent ';' End                 {Semis $1 $3}
    | print '(' Func ')' ';'      {Print $3}
```

Listing 2: Entry point to the AST and its ending

where 'Ent' represents the rest of the functions and operations of the language.

Furthermore, when creating a complex type such as a 'Tile', we decided to use recursive calls in the AST.

```
Row : '[' Block ']'              {Row $2}
    | '[' Block ']' ',' Row      {Rows $2 $5}

Block : int                {Block $1}
      | int ',' Block      {Blocks $1 $3}
```

Listing 3: Tile creation in the Parser

Lastly, due to the way we defined our 'token_posn' function in the lexer, our 'parseError' function gives the precise position of the token that caused the error.

```
parseError :: [TokenType] -> a
parseError rem | null rem = error "Missing input"
               | otherwise = error ("Parse error on: " ++ (token_posn $ head rem))
parseError _ = error "Parse error"
```

Listing 4: parseError function

3.3 Type Checker

To implement the type checker, we began by defining a set of handy helper functions - namely, **lookup2**, **bind**, and **unparsetype**. **lookup2** allows us to search for a variable's type within an environment, while **bind** adds a binding for a variable and its corresponding type to the environment. The **unparsetype** function simply converts types into strings, enabling them to be printed out more effectively in error messages.

With these helper functions in place, we then set about implementing the main type-checking function, called 'typeOf'. This function takes an expression and a type environment, and returns a pair comprising of a type and environment for later use in the interpreter if required. Additionally, we included an extract helper function, which extracts the type out of the pair.

It is worth noting that for each expression, the type checker ensures that all relevant type rules are satisfied. If any of these rules are not met, a comprehensive error message is generated, indicating the expression and the types that are expected as inputs. A very important rule we

made was also not allowing blocks to accept anything other than 0's and 1's which makes up the tiles. Overall, by leveraging these powerful functions, we were able to create a robust and reliable type-checking system.

```
typeOf xs (Block x)   | (x == 0 || x == 1) = (BlockType, xs)
                      | otherwise =  error "Block only accepts int 1 and int 0."
```

Listing 5: typeOf function example

3.4 Interpreter

In order to implement structural operational semantics, we created a function called 'eval1', which performs a small step evaluation once, either pushing a frame into the stack and returning a sub-expression to be evaluated in the next iteration, popping a frame off the stack and pushing another frame, popping a frame off the stack and returning a sub-expression to be calculated, or returning a calculated value. Depending on the configuration to be evaluated, the function pushes different frames into the stack and chooses whether to access or update the environment, if needed. It is important to highlight that several helper functions were coded to be called whenever a calculated value is to be returned by 'eval1'. Those functions are most of the built-in functions we decided to include, such as "join", "stack", "supersize", "subtile", for example, and also for the "for-loop".

Furthermore, in order to evaluate an entire program, 'eval1' had to be called repeatedly in a loop until a 'print' function is read by the parser. Consequently, we added an evalLoop function (**Listing 6**) which evaluates each expression up to the semicolon separately, passing on the current environment into the next iteration of the loop, and goes on until it finds a 'print' function.

```
evalLoop :: Exp -> Environment -> String
evalLoop e env = evalLoop' (e,env,[])
  where evalLoop' :: Configuration -> String
        evalLoop' (ex1,env2,k) = case decideSemi (ex1,env2,k) of
                                    (Print ex1, env2, k) -> printTile ex1 env2
                                    (ex1, env2, (SemisHole ex2):k) -> evalLoop' (ex2,env2,k)
        decideSemi :: Configuration -> Configuration
        decideSemi w@(Print ex1, env2, k) = w
        decideSemi (Semis ex1 ex2, env2, k) = semiEval (ex1, env2, (SemisHole ex2):k)
        semiEval :: Configuration -> Configuration
        semiEval cnf@(ex1, env3, (SemisHole ex2):k3) | isValue ex1 = cnf
        semiEval cnf = semiEval $ eval1 cnf
```

Listing 6: evalLoop function

where 'printTile' is a function that prints strictly tiles of size N x N, 'Semis ex1 ex2' is the data type that stores on 'ex1' the line to be interpreted next, where 'ex2' the rest of the program, and 'SemisHole' is a frame that stores 'ex2'.