University of Southampton

Comp2212 – Coursework Report

Language features, syntax, and more

Mhz1g21 - Mathew Zone
Plw1g21 - Paul Winpenny
Th2g20 – Thomas Holwill
03/05/2023

Language Features

Control Structures

repeat n {}	Creates a block that will repeat all the statements contained in its brackets n times.	`X = 1;`
		`repeat 5 {`
		`  + X 1;`
		`};`

while x {}	Creates a black that will repeat all the statements contained in its brackets until the condition x becomes false.	`test = 1;`
		`while lt test 5 {`
		`  + test 1;`
		`};`

if x {} else {}	A conditional statement. If condition x evaluates to true then the first block of code will be executed, otherwise the code in the else block will be executed. If an else block is not required, then put “none;” in the else to tell the interpreter to skip it.	`X = 1;`
		`Y = 2;`
		`if gt y x {`
		`  print true;`
		`} else {`
		`  none;`
		`};`

none	Performs no operation, tells the interpreter to skip. Only used for when you do not need an else block in an if-else statement.	`none;`

Expressions and Statements

Assignment Statements

X = y	Assign a variable to a value. Variable can be assigned to integers, Booleans, and tiles. Assignments can also be conducted by assigning a variable to the output of an expression.	`X = y;` `X = 1;`

Expressions

Operations that return a value.

joinV x y	Joins two tiles vertically and returns the new tile. Tiles must have the same width to join vertically.	`t1 << tile1;`
		`t2 << tile2;`
		`t3 = joinV t1 t2;`

joinH x y	Joins two tiles horizontally and returns the new tile. Tiles must have the same height to join horizontally.	`t1 << tile1;`
		`t2 << tile2;`
		`t3 = joinH t1 t2;`

rotate n tile	Rotates a tile 90° clockwise n times and returns the new tile.	`t1 << tile1;` `t2 = rotate 1 t1;`
scale n tile	Scales a tile n times and returns the new tile.	`t1 << tile1;` `t2 = scale 2 t1;`
reflectX tile	Reflects a tile in the x-axis and returns the new tile.	`t1 << tile1;` `t2 = reflectX t1;`
reflectY tile	Reflects a tile in the y-axis and returns the new tile.	`t1 << tile1;` `t2 = reflectY t1;`

_ x	Creates a blank tile. X can be an integer to create a NxN tile or a reference tile to create a blank tile of the same size.	`blank = _ 2;`
subtile x y tile	Creates a subtile where the top left corner is at (x,y) of the input tile and returns the new tile.	`t1 << tile1;` `t2 = subtile 1 1 t1;`

gibb x y tile1 tile2	Pastes tile1 onto tile2 so that the top left of tile1 is at (x,y) in tile2 and returns the new tile.	`t1 << tile1;`
		`t2 << tile2;`
		`t3 = gibb 1 1 t1 t2;`

width tile	Returns the width of an input tile.	`t1 << tile1;` `w = width t1;`
height tile	Returns the height of an input tile.	`t1 << tile1;` `w = height t1;`
not x	Inverts a tile or Boolean value and returns the new tile/value.	`t1 << tile1;` `t2 = not t1;`

or x y	Performs a logical or on two tiles or two Boolean values.	`t1 << tile1;`
		`t2 << tile2;`
		`t3 = or t1 t2;`

and x y	Performs a logical and on two tiles or two Boolean values.	`t1 << tile1;`
		`t2 << tile2;`
		`t3 = and t1 t2;`

lt x y	Performs a less than comparison such that x < y and returns the Boolean value of this.	`X = 1;`
		`Y = 2;`
		`tf = lt X Y;`

gt x y	Performs a greater than comparison such that x > y and returns the Boolean value of this.	`X = 1;`
		`Y = 2;`
		`tf = gt X Y;`

eq x y	Performs an equality comparison such that x == y and returns the Boolean value of this.	`X = 1;`
		`Y = 2;`
		`tf = eq X Y;`

neq x y	Performs a not equal comparison such that x != y and returns the Boolean value of this.	`X = 1;`
		`Y = 2;`
		`tf = neq X Y;`

+ x y	Addition of two integers such that result = x + y.	`X = 1;`
		`Y = 2;`
		`Z = + X Y;`

- x y	Subtraction of two integers such that result = x – y.	`X = 1;`
		`Y = 2;`
		`Z = - X Y;`

IO Statements

X << filename	Imports a filename.tl and assigns it to variable x. tiles must be NxN when importing.	`t1 << tile1;`
X >> filename	Exports variable x to filename.tl. variable x must be a tile.	`t1 >> tile1;`
print x	Prints value/variable x to stdout. x can be an int, Boolean, tile, or a variable that reference those types.	`print t1;`

Syntax & Lexical Rules

Lexical rules:

The lexer is implemented through the "Tokens" file, which runs through a given program and converts strings into various tokens. This is done by using regular expressions (regExp), for example:

```
"gt"        { \p s -> TGreaterThan p}
```

This will identify when the greater than symbol (gt) is used.

```
$digit+         { \p s -> TNumber p (read s) }
```

This regExp looks for when at least one digit ([0-9]) is spotted.

Within the bracers, the "\p s" refer to the position of the string in the code (p) and the value of the string {s}, e.g.

gt 2 1 is lexed as: [TGreaterThan (AlexPn 0 1 1),TNumber (AlexPn 3 1 4) 2, TNumber (AlexPn 5 1 6) 1]

The positions of each regExp are included because of the 'posn' wrapper and have been included to aid in the debugging process, as any issue with lexing will highlight the exact location of the error.

Whitespace and Comments:

```
$white+          ;
```

Any instance of at least one whitespace (' ') will be converted into no token, I.e., Whitespace will be ignored.

```
":)".*             ;
```

Any instance of the comment symbol ":)" followed by any number of characters will be converted into no token (ignored).

The inclusion of commenting is beneficial for any developer as it allows for easy description of code, as well as ignoring code that might be used in later development.

TokenPosn:

Within the tokens file is the function "tokenPosn :: Token -> String", which will allow errors to display the line and column number of a token.

Grammar:

Once the tokens are lexed, they are then parsed into an abstract syntax tree (AST). This AST is derived by following the grammar outlined in the "grammar.y" file.

Terminals and Non-Terminals:

```
gt {TGreaterThan $$}
true {TBoolean (AlexPn x y z) $$}
```

These two terminal production rules are used when their respective symbols are identified.

```
ExpSeq: Exp ';' ExpSeq { ExpSeq $1 $3}  | Exp ';' { Exp $1}
```

This production rule allows for the inclusion of multi-lined programs, due to the recursive call of the rule. Each line must end in a ";"

```
Exp : repeat int '{' ExpSeq '}' { Repeat $2 $4}
```

One of the Exp production rules which allows for the "repeat" symbol to be followed by several expressions.

```
lt Exp Exp { LessThan $2 $3 }
```

The "less than" symbol must be followed by two expressions to allow a comparison to be made.

```
neq Exp Exp {NotEqual $2 $3}
```

The "not equal" symbol must be followed by two expressions to allow a comparison to be made.

*This is not the full set of symbols defined within the grammar

Syntax Sugar: We have employed a few pieces of syntax sugar to convenience a programmer. One of these is using acronyms for comparing two expressions, e.g. using "eq" and "neq" for "equal to" and "not equal to" respectively. Opting out of the mathematical operators will better convey the meaning of each symbol. Another piece of syntactic sugar used is the "none" symbol, as well as including the "else" symbol in the production rule for 'if' statements:

```
none {None}
```

```
if Exp '{' ExpSeq '}' else '{' ExpSeq '}' {If $2 $4 $8}
```

This will ensure that any user is able to understand how if statements are formatted, with "none" being a symbol to use if they would like to leave either the "if" or "else" empty.

Another piece of syntax sugar is that mathematical symbols (eq,neq) are placed in front of the values the operator is being applied to:

```
eq temp 2
```

is parsed as

```
IsEqual (Var "temp") (Int 2)
```

This was done to keep consistency for the user if they want to use the functions. Furthermore, each symbol is defined as left associative (%left 'eq'), which will aid the user as most symbols are evaluated from the left.

Runtime

Execution environment

Our runtime environment is a map. Using this structure we store our variables, which means the variables are not garbage collected as the program leaves loops where assignments are made.

```
--enviroment variables
data Enviroment = Enviroment { stack :: [Value], symbolTable :: Map String Value } deriving Show
```

The language is written to allow variable manipulation to be its core strength; for this reason, we did not see it necessary to add variable scoping to the language. If a variable was assigned previously, you could reference it anywhere after the assignment; additionally, this prevents an issue of trying to access variables assigned in control structures from the outside. Although a stack is present in the runtime environment, it is not used, the stack was added in an attempt to make the interpreter 'future proof' in the event that a certain feature is required and is needed to be added later that requires a stack.

The execution of the instructions are stored in our AST (abstract syntax tree) and are executed one instruction after another. If we take the instructions for joining two tiles vertically:

```
t1 << tile1;
t2 << tile2;
t3 = joinV t1 t2;
```

line 3 gets parsed as:

```
Exp (Equals "t3" (JoinV (Var "t1") (Var "t2"))
```

When the interpreter reaches this line, it will first be treated as an assignment operation, however when it starts to evaluate, we can see that we have another expression to evaluate first. Our runtime will see this, evaluate this section first (as soon as it sees it), collect the returned values, and add the assignment to the map, then return the updated environment for the next instruction to use. This is also true for Var "t1" and Var "t2" as these are also treated as expressions to indicate that we are referencing a variable.

Type checking

For our language, we decided to implement strict type checking within the runtime of the interpreter, and not within the grammar itself. In our interpreter we have three distinct types:

```
data Value = TileValue Tile
  | IntValue Int
  | BoolValue Bool deriving (Show,Eq)

type Tile = [[Bool]]
```

The interpreter has functionality for integers, Booleans, and tiles. These are the only types included and are strictly checked at the evaluation of an instruction.

For example, the joinV expressions parses into our Haskell AST as:

```
JoinV Exp Exp
```

Meaning that JoinV does not have to be immediately passed two tiles. However, the Exps must evaluate to TileValue or the input will not be accepted and a runtime error will the thrown. By implementing our type checking this way, we allow the writer to create complex multi-line instructions. An example of this could be in Haskell when you can write:

```
 take 2 (head (tail (replicate 2 ([1] ++ []))))
```

The key distinction here is that our interpreter checks the type of the Exp once it has evaluated it, not before, making it easy to perform a strict type check of the returned value against our 'data Value' structure.

Error messages

In our interpreter there are four different types of errors:

Argument errors – when the file given to the interpreter is not a .tsl file or no file is given as an argument.

Lexical errors – when the given file does not meet the lexical rules, the position of the incorrect character will be provided to the user in the error.

Parsing errors – when the given file does not meet the rules of our grammar and therefore cannot be translated to our AST, the position of the incorrect statement will be provided to the user in the error.

Runtime errors – runtime errors are thrown if something unexpected happens for example invalid types in an expression or incorrect use of certain control structures.