COMP2212 Programming Language Concepts Coursework - 'Tiling Toolkit'

Jiahua Lin Jianing Kong Danny Martinez Hibbert

April 2023

1 Introduction

The domain specific programming language we have designed for the tiling problems allows users to build and manipulate tile objects. A Tile type is considered to be a collection of discrete cells, where each cell may be either filled or empty, and the user can create larger tiles built from sets of small tiles. The language allows users to perform a variety of tile-specific operations on tiles, including rotating, reflecting, scaling, sub-tiles and more. Users can also import and manipulate with pre-existing tile patterns, define and use variables, and create complex logical expressions and conditions.

2 Syntax

<**Type**> **::= Int | Base** - type of a variable, is either a Base object or an Integer.
<**var**> **::= [a-z][a-zA-z]*** - variable identifiers, starting with a lower case alphabet and no special characters are allowed.
<**baseVar**> - refers to identifiers of Base Object variables.
<**fileName**> - refers to file names of files that contains a tile/base object. Files must be in the same directory as the interpreter. No suffixes required.
<**Operations**> - refers to operations that return Base Objects or Integers.
<**cells**> **::= ('0' | '1')*** - **'0'** represents a blank cell, **'1'** represents a filled cell.
<**numCom**> - refers to numerical comparisons, operations that return Boolean values.
<**numOp**> - refers to operations that return an Integer, identifier of an Integer variable or an Integer.
<**logicOp**> - refers to operations that return Boolean values (true/false).

2.1 Statements, Import, Iterations and Conditions

Function	Syntax	Description
Statements	<Stmts>; <Stmt>	<Stmts> refers to any operations, followed by ';'.
Statement	<Stmt>	<Stmt> refers to one single operation.
Import	import <fileName> as <var>	Given a <fileName>, read and store the content to variable <var> as a Base Object.
If-Then	if <logicOp> then {<Operations>} end	Execute given <Operations> if <logicOp> satisfied.
If-Then-Else	if <logicOp> then {<Operations>} else {<Operations>} end	Execute <Operations> within 'then' block if <logicOp> satisfied, otherwise execute <Operations> in 'else' block.
Repeat Loop	repeat <numOp> do <Operations> end	Execute given <Operations> for <numOp> of times.
While Loop	while <logicOp> do <operations> end	Execute given <Operations> while <logicOp> satisfied.

2.2 Variable Assignment

Function	Syntax	Description
Variable Assignment	<Type> <var> = <Operations>	Assigning a value to a variable. The return type of the operation should be matching to the indicated type.
Variable Re-assignment	<var> = <Operations>	Assigning an existing variable to a new value. The return type of the operation should be the same as the current type of the variable.

2.3 Base Object Operations - Return Base Objects

Function	Syntax	Description
Cells	cells : [<cells>]	Building a Base Object with filled/blank cells. Use vertical bar '|' to annotate changing of row.
Repetition	<baseVar> -* <numOp> <baseVar> +* <numOp>	Horizontally (-*) or vertically (+*) append a base object with itself for a given number of times.
Append	<baseVar> -& <baseVar> <baseVar> +& <baseVar>	Horizontally (-&) or vertically (+&) append 2 base objects.
Rotation	rotate <baseVar> by <degree> CW rotate <baseVar> by <degree> ACW	Rotate a base object by 90/180/270 degrees (<degree>) in clockwise (CW) or anti-clockwise (ACW) direction.
Reflection	reflect <baseVar> on row <numOp> reflect <baseVar> on col <numOp>	Reflect a base object on the nth row or column.
Expansion	expand <baseVar> by <numOp>	Expand a base object by a given number of times.
Subtile	take <row> <col> size <numOp> from <baseVar>	Extract a subtile from a larger base object. '<row> <col> ' indicates the top-left corner of the subtile and are integers. 'size ' indicates the size of the subtile.

2.4 Numerical Operations - Return Numerical Values

Function	Syntax	Description
Get Tile Length/Width	get length <baseVar> get width <baseVar>	Retrieve the length or width of a base object.
Arithmetic Operations	<numOp> + <numOp> <numOp> - <numOp> <numOp> * <numOp> <numOp> / <numOp>	Mathematical calculations, including addition (+), subtraction (-), multiplication (*) and division (/).

2.5 Logical Operations - Return Boolean Values

Function	Syntax	Description
Numerical Comparisons	(<numOp> < <numOp>) <numOp> > <numOp>) (<numOp> <= <numOp>) (<numOp> >= <numOp>) (<numOp> == <numOp>) (<numOp> != <numOp>)	Comparisons between numbers, including less than (<), larger than (>), less than or equal to (<=), larger than or equal to (>=), equal (==) and not equal (!=). All comparisons should be wrapped in parentheses.
Logical And/Or	<numCom> && <numCom> <numCom> || <numCom>	Logical conjunction operator (&&) and logical dis-junction operator (||). Numerical comparisons should be wrap -ped in parentheses.

3 Language Types

3.1 Type Rules

$$\frac{}{\Gamma \vdash n : Int}\ Int \qquad \frac{}{\Gamma \vdash b : Bool}\ Bool \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ Var \qquad \frac{}{\Gamma \vdash c : Char}\ Char$$

$$\frac{\Gamma \vdash E : Char \quad \Gamma \vdash ES : List}{\Gamma \vdash E :: ES \ : \ List}\ CellList \qquad \frac{\Gamma, x : T \quad \vdash E_1 : T}{\Gamma \vdash Base\ (x : T)\ =\ E_1\ :\ Base}\ Base$$

$$\frac{\Gamma, x : T \vdash E_1 : Base}{\Gamma \vdash import\ (x : T)\ as\ E_1\ :\ Base}\ Import \qquad \frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Base}{\Gamma \vdash rotate\ E_1\ E_2\ (CW \vee ACW)\ :\ Base}\ Rotate$$

$$\frac{\Gamma \vdash E_1 : Int}{\Gamma \vdash reflect\ (x : T)\ on\ (row \vee col)\ E_1\ :\ Base}\ Reflect$$

$$\frac{\Gamma \vdash E_1 : Int}{\Gamma \vdash expand\ (x : T)\ by\ E_1\ :\ Base}\ Expand \qquad \frac{\Gamma \vdash E_b : Bool \quad \Gamma \vdash E_1 : T \quad \Gamma \vdash E_2 : T}{\Gamma \vdash if\ E_b\ then\ E_1\ else\ E_2 : T}\ IfThen$$

$$\frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Int \quad E_3 : Int \quad E_4 : Base}{\Gamma \vdash E_1\ E_2\ size\ E_3\ from\ E_4 : Base}\ Take \qquad \frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Int}{\Gamma \vdash E_1 + E_2 : Int}\ Add$$

$$\frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Int}{\Gamma \vdash E_1 - E_2 : Int}\ Sub \qquad \frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Int}{\Gamma \vdash E_1 * E_2 : Int}\ Mul \qquad \frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Int}{\Gamma \vdash E_1 / E_2 : Int}\ div$$

$$\frac{\Gamma \vdash E_1 : Bool \quad \Gamma \vdash E_2 : Bool}{\Gamma \vdash E_{b1}\ \&\&\ E_{b2} : Bool}\ \&\& \qquad \frac{\Gamma \vdash E_1 : Bool \quad \Gamma \vdash E_2 : Bool}{\Gamma \vdash E_{b1}\ |||\ E_{b2} : Bool}\ |||$$

$$\frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Int}{\Gamma \vdash E_1\ <\ E_2 : Bool}\ LT \qquad \frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Int}{\Gamma \vdash E_1\ >\ E_2 : Bool}\ GT \qquad \frac{\Gamma \vdash E_1 : Int \quad \Gamma \vdash E_2 : Int}{\Gamma \vdash repeat\ E_1\ do}\ Repeat$$

3.2 Type system

The type relations for the language have been kept as simple as possible to focus strictly on performing operations on tiles. Tiles are represented as 1's and 0's where 1's are filled cells and 0's are empty cells. Operations are performed mainly through loops, transformations and appending. The language infers bools for any logical evaluation, for simplicity. 'Base' includes any objects made of a nested list of cells. 'Int' include any integers n st. $n \in \mathbb{Z}$. Bools are inferred by the interpreter for flow control.

4 Error Checking

4.1 Well typed and concrete system

Our language ensures that the code is executed correctly and prevents unexpected behaviours, which is mainly achieved through our type checker before run time. Only the correct operations can be performed on certain types and all assignments must be specified as a 'Base' or 'Int' object - re assignments must also match their original declarations. Our type checker runs during compile time before any code is evaluated to ensure a concrete type system. Any variables that are assigned to variables that haven't been declared will throw a "Variable not found error".

4.2 How the language manages errors

Error handling is implemented using the error directive in our Grammar file. The directive specifies the parseError function that will be called in case of a parsing error. The function raises an error message indicating the line and column where the error occurred, which is a useful feature for debugging. Other directives where also used as a form or error prevention, such as the nonassoc and left right directives to specify the precedence and associativity of the operators in the grammar. This ensures that the parser correctly groups and evaluates expressions.

Our Interpreter also implements error handling via the error function. It is used in multiple places to handle different kinds of errors. For example, it is used to report "Tile Rotate Error" when there is an error in the rotateCW or rotateACW functions. Similarly, it is used to report "Tile Reflect Error" when there is an error in the reflectR or reflectC functions and much more, such as Append, And Subtile operations.

4.3 Undefined Variable References

The type checker covers a wide range of language constructs, including basic types (Integers, Boolean), arithmetic operations, comparisons, logical operations, iteration statements, and conditional statements. The type checker also introduced a "no type" for undefined variable references.

4.4 If-then-else

The type checker also handles the issue of dangling else statements in if-then-else constructs. In particular, if the type of the condition is not a Boolean, the checker returns NoType. Furthermore, if the types of the two branches of the if-then-else construct do not match or have NoType, the checker returns NoType.

4.5 Handle Imports

The evaluator also recognises and handle any invalid input for reading file, including reading spaces, special symbols, letters and characters etc. One example of this is in the "getFromFile" function, which reads input from a file and only accepts 0s or 1s. If the function encounters an invalid character, it will throw an error message that indicates the invalid input.

5 Language Design

5.1 Model explanation

Our language follows a typical execution model for imperative languages, taken inspiration from popular languages such as Python and Java. The interpreter execute an Abstract Syntax Tree of the program in a Depth-First-Search like fashion after Tokenising and Lexing. The parse tree is made up of expressions and statements that are evaluated sequentially and literally in the order they are written in the program. The semi-colon separated expressions act as the nodes of the parse tree, and each node is evaluated one by one.

5.2 Flow control

The language has been designed with simplicity and ease of use in mind, but without sacrificing expressive power. The syntax and grammar have been carefully thought to strike a balance between these two goals, and the result is a language that is both easily understood and learnt by those who have experience with imperative programming languages, while still being capable of handling complex instructions.

In addition to standard sequential execution, our language supports control flow statements such as "for", "Repeat", "if-then-else", allowing more freedom and control of the evaluation order of any given expression.

In our language, every statement is terminated with a semicolon ";". This helps to clearly separate different statements and makes parsing easier. Additionally, we use the keyword "end" to indicate the end of loops and iterations.

5.3 Commenting and syntactic sugar

Our language allows the use of comments "//" to make the code more readable and help programmers explain their code. There are also a number of syntactic sugars we used for our language for the programmer's convenience. These include operations such as "-&", "+&", "-*", "+*" for direct tile manipulation, such as appending/adding horizontally or vertically. Common mathematical symbols are used for mathematical expressions and calculation, which are widely used and understood, helping to make the code more concise and easier to read.

6 Execution Model

The following big step semantics show the functionality of the underlying small step evaluation of expressions used by the interpreter.

6.1 Big step operational semantics

$$\frac{E_1 \Downarrow n \quad E_2 \Downarrow m \quad n+m=n'}{E_1+E_2 \Downarrow n'} \quad \frac{E_1 \Downarrow n \quad E_2 \Downarrow m \quad n-m=n'}{E_1-E_2 \Downarrow n'} \quad \frac{E_1 \Downarrow n \quad E_2 \Downarrow m \quad n/m=n'}{E_1/E_2 \Downarrow n'}$$

$$\frac{E_1 \Downarrow n \quad E_2 \Downarrow m \quad n*m=n'}{E_1*E_2 \Downarrow n'} \quad \frac{E_1 \Downarrow true \quad repeat\ E_1\ do\ E_2\ end; \Downarrow V}{repeat\ E_1\ do\ E_2\ end; \Downarrow V} \quad \frac{E_1 \Downarrow true \quad while\ E_1\ do\ E_2 \Downarrow V}{while\ E_1\ do\ E_2 \Downarrow V}$$

$$\frac{E_1 \Downarrow V \quad E_2[V/x] \Downarrow V'}{T\ x\ =\ E_1;\ E_2; \Downarrow V'} \quad \frac{E_1 \Downarrow t_1 \quad E_2 \Downarrow t_2 \quad t_1\ +\&\ t_2 = V}{E_1\ +\&\ E_2 \Downarrow V} \quad \frac{E_1 \Downarrow true \quad E_2 \Downarrow V}{if\ E_1\ then\ E_2\ else\ E_3 \Downarrow V} \quad \frac{E_1 \Downarrow n\ E_2 \Downarrow m\ n<m}{E_1\ <\ E_2 \Downarrow true}$$

$$\frac{E_1 \Downarrow n\ E_2 \Downarrow m\ n\not< m}{E_1\ <\ E_2 \Downarrow false} \quad \frac{E_1 \Downarrow n \quad E_2 \Downarrow V}{reflect\ E_1\ on\ (row \vee col)\ E_2 \Downarrow V} \quad \frac{E_1 \Downarrow n \quad E_2 \Downarrow V}{rotate\ E_1\ E_2\ (CW \vee ACW) \Downarrow V}$$

$$\frac{E_1 \Downarrow V \quad E_2 \Downarrow n}{expand\ E_1\ by\ E_2 \Downarrow V} \quad \frac{E_1 \Downarrow b_1 \quad E_2 \Downarrow b_2 \quad b_1\ and\ b_2\ =\ V}{(and \vee or)\ E_1\ E_2 \Downarrow V} \quad \frac{E_1 \Downarrow false}{repeat\ E_1\ do\ E_2\ end; \Downarrow V} \quad \frac{E_1 \Downarrow false}{while\ E_1\ do\ E_2 \Downarrow V}$$

$$\frac{E_1 \Downarrow false \quad E_3 \Downarrow V}{if\ E_1\ then\ E_2\ else\ E_3 \Downarrow V}$$