COMP 2212 Programming Language Concepts Group Project

Jake Mortlock - jm15g21
Alberto Berni - ab3u21
Augustas Mirinas - am26g21

About the language

We determined that some of the most important concepts to the provided problem domain were:

- Functional tile building (Use of a function to determine the value of a tile based on the X-Y coordinate, for example problems like the Mandelbrot set utilise this)
- Extrapolating images (Repeating an image forever)
- Cropping images to a specific size
- Concatenating tiles together to form a larger compound tile (When designing this, we wanted to make the programming language look similar to an output image with rows and columns).

We designed the language by first writing the programs that would be able to theoretically solve the problems and then iteratively simplifying those programs until we had a more concise and simple language.

Return Value Rules

The final value at the end of a multi-line block is the value that is returned by that block. For the base program this means that the terminated value at the end of the program is the output of the program.
Our language design does not allow for IO-based side effects within regular execution (See scoping rules for more details), meaning that functions that return nothing are redundant. Adding an explicit keyword for returning would just add more complexity to the language which the programmer then has to deal with.

Functions and multi-line inline expressions return the value obtained at the end of execution.

Assignment returns the value that was assigned. For example, `a = 4` will return 4.

Scoping Rules

Our language scopes based on certain rules. The language enters a new scope when:

- Entering the body of a function.
- Entering the selector of a switch across expression.
- Entering the selector of a tile builder across expression.
- Entering an inline code block (`() => { ... }`).

When entering a new scope the state of the environment is stored. When exiting a scope block, the environment is restored to the state it was in before the code block was entered.
This means that you can access variables in the scope above, however any modifications to those variables remain local.

Functions

We designed the language under the idea that we wanted to have as little functions built into the AST as possible and instead use a standard library written in our language that could provide that functionality.

Our language was designed with the principle that all internal functions such as scale, mirror, etc. should be able to be written in our language. This allows for flexability, as when we encounter a function that is missing from the language we can easilly implement it. If we were going to release this as a full language, then these functions would be built into a standard library, however we could only submit a single file as our programs for this coursework, so we have written them to the top of the file instead.

Lines (Arrays)

Our language uses lines for arrays. Most operations across lines use deferred evaluation so are only evaluated when necessary. This means that there is no performance cost for making huge arrays and only sampling some specific values from them. This also allows us to create infinite arrays that have their values determined functionally from selectors.

Generation

Arrays can be generated from a range using the `..` syntax, for example `1..5` will generate a line containing the numbers from 1 to 5. Range is evaluated when needed, so there is no cost from creating huge ranges.

2D lines (which we will refer to as grids) can be created from files using the `in` operator. The `in` keyword takes an integer argument, and will create a grid from the file named `tilen.tl` where n is the number provided as the parameter.

Selectors

Lines can also be transformed using selector functions. These are evaluated on a need-to-know basis and work by taking the element requested from the source, passing it to the selector function and then returning the transformed value. Selectors can be applied across an input list.
There are 2 types of selectors in our language: switch across and tile builder across. The only difference between these 2 is that the tile builder across will concatenate 2D arrays of tiles, returning a 2D array whereas the switch across expression will just return a 4D array instead. In all other cases these are the same for convenience reasons.

These are used in the following manner:

```
{
        selector_function
} : (variable_name in source)
```

Where the selector function is the function applied to each element of source. An example of a selector that multiplies every element from a source array by 2 is as follows:

```
switch { else => x * 2 } : (x in source)
switch { x * 2 } : (x in source)
{ x * 2 } : (x in source) // All of these are valid and return the same output.
```

Indexing

Our language allows for getting the element at a specific index within a line. Indexing can be performed across repeating lines and regular lines, and the requested index may be either a number or a list of numbers.

```
example_list = [1..5] //All the numbers between 1 and 5
example_list[4] ~= 4
repeating_numbers = $(example_list) //The numbers between 1 and 5 repeating forever
repeating_numbers[6] ~= 1
```

If the source list is a repeating list, then the index will be determined by performing `index % length(source)`. When the value used as the index is a line, then a line will be returned.

```
example_list[1..3] ~= [1..3]  //Gets a line containing value 1, 2 and 3 from example_list
```

If the source list is multi-dimensional, then you can index across each dimension by using the comma. The indexer will perform the index with the first argument, take the result and then perform indexing with the second argument. Below are some examples:

```
grid_input = in(1) //Get an input, for the sake of example lets say this is a 5x5 image
grid_input[1, 1] // Top left pixel
grid_input[1..5, 1] // The row of pixels where y = 1
grid_input[1, 1..5] // The column of pixels where x = 1
grid_input[1..5, 1..5] // A grid cropped between (1, 1) and (5, 5)
```

This is extremely useful when cropping images to a specific size, if you want a 64×64 output then you can index the input with `[1..64, 1..64]`.

Repeating

The repeat operator, represented by the `$` token allows you to declare a line/block as infinitely repeating. This means that the provided line will repeat forever. This operator applies deeply to lists, meaning that 2D grids inputted into it will repeat on both the X and Y axis forever. Infinitely repeating sequences cannot be outputted, so need to be cropped using the index operator first.

Tile Builder

The tile builder pattern allows for creating more complex patterns within the context of the language. It supports concatenating tiles together as well as creating tiles from individual pixels. Each element in a row is separated with a comma, and every row is seperated with a semi-colon. The appearance of the elements inside the tile builder construct match the image that will be formed. For example:

```
grid = {
        0, 1;
        1, 0;
}
```

will create a 2×2 image with a 1 pixel in the top right and bottom left corner, with 0 pixels everywhere else.

When the expression fed into the tile builder is a tile itself, then the tile builder will concatenate the tiles together rather than forming 4-dimensional lines. For example:

```
combined_grid = {
        grid, grid;
        grid, grid;
}
```

will create a **2n**x**2m** grid by placing the input **nxm** grid in a 2×2 arrangement. In order for the tile builder to work properly, the height of every element in 1 row must be the same and the width of every element in 1 column must be the same. The widths and heights of different rows/columns do not need to be the same which allows you to place a larger image surrounded by some smaller ones.

Integrated Testing

Our language supports integrated testing using the assertion operator (`~=`). This will compare the left and right, similarly to how `==` operates, but will throw an AssertionFailure on failure which terminates the program and prints an error. When compared with our library for automatically finding and running tests, our language has a powerful method for creating tests to ensure language behaviours are working as expected.

Lexer

The lexer converts the patterns used by the language into tokens. Anything starting with `//` up to the end of the same line is a comment and will be ignored by the lexer. Our language uses new-lines as a break token as well as `;`. This means that semi-colons at the end of lines are allowed, but optional.

Variable names may consist of any characters (a-z, A-Z), numbers (0-9), underscores (_) and apostrophes (').

Anything surrounded by `#{...}` where `...` is any string will be treated as a raw abstract syntax tree written in a format readable by Haskell's `show` method. This is used in unit testing to ensure that the AST output of the parser is the same as what is expected, without actually evaluating the AST. This was used for testing the parse trees before the interpreter was implemented.

Parser

The parser uses a block like design, where a single rule can contain rules that have a lower precedence inside of them. This gave us a long, but highly readable parser which allowed us to do more complex behaviours.

One of the biggest difficulties with the parser was getting the expression `a().b()` to work correctly. This is because for this expression, the order of operator precedence depends on where it is. The `a()` should be executed first, followed by accessing the variable `b` from the result, followed by executing that whole expression. In the end, this expression was never actually used as our language is not object-oriented.

We use the `%prec` tokens in the parser to specify associativity. Precedence is determined by the height in the rule tree. The higher up a rule is in our tree of rules, the higher that expression will be in the outputted abstract syntax tree.

Evaluator

The evaluator uses a CEK machine model for interpretation. The states of the runtime consiste of:

- The expression currently being evaluated.
- The environment (A list of pairs that match up variable identifiers with an expression value).
- The kontinutation (A stack of frames that tell the CEK machine how to travel back to the root expression).

We perform pure and IO evaluation seperately. This is because input was one of the last things to be added into the interpreter and rewriting the entire thing to return an IO CEK instead of a regular CEK would have been too time consuming.
Instead the interpreter runs `io_eval_step` which attempts to perform an IO evaluation. If this fails, then it calls `eval_step` which attempts to perform a pure evaluation of the expression.

Typing Rules & Type Checking

Types

Our language consists of the following types (<T> represents a generic type, <T...> represents a 0 to many generic type):

- Numeric - Represents a type that stores any numerical value. There is no explicit distinction between integers and floats as a language for this domain does not require it. Since the domain is mathematical rather than engineering-focused, we don't need to make that distinction.
- Boolean - Represents a true or false value.
- Line<T> - Represents a finite set of values of type T.
- RepeatingLine<T> - Represents an infinitely repeating set of values of type T.
- Function<ParamTs..., ReturnT> - Represents a function that takes in 0 to many arguments of the types provided in paramTs (These do not have to be the same type) with a return type of ReturnT.
- AnyType - Represents any valid type.

When designing the language, we wanted to keep the typing as simple as possible and have as few types as possible. The initial language was designed around having only numbers, lines and repeating lines. Booleans were seperated from numbers after we found that strong static typing was preferred. If this preference was not present, then we likely would have stuck with the execution model that makes no differentiation between integers and booleans as it makes programming in the language more convenient, and the language has been designed to be simple so there are limited places where you would be greatly helped by stricter typing.

Type Inference

Variables in our language are strictly typed and their type is determined through type inference. The type of the variable will be the type of the first expression assigned to it.

Our type checker cannot currently infer the types of parameters to functions. Instead, all parameters to functions are given the AnyType type, which allows them to be used in all expressions without type checking. As the AnyType'd parameters go through the function body, their type gets restrained by the operations inside the function until the return type can be determined. Unfortunately, despite being restrained the type checker's current implementation cannot pass that restrained type to any callers of the function meaning function calls are unchecked.

Type Checker

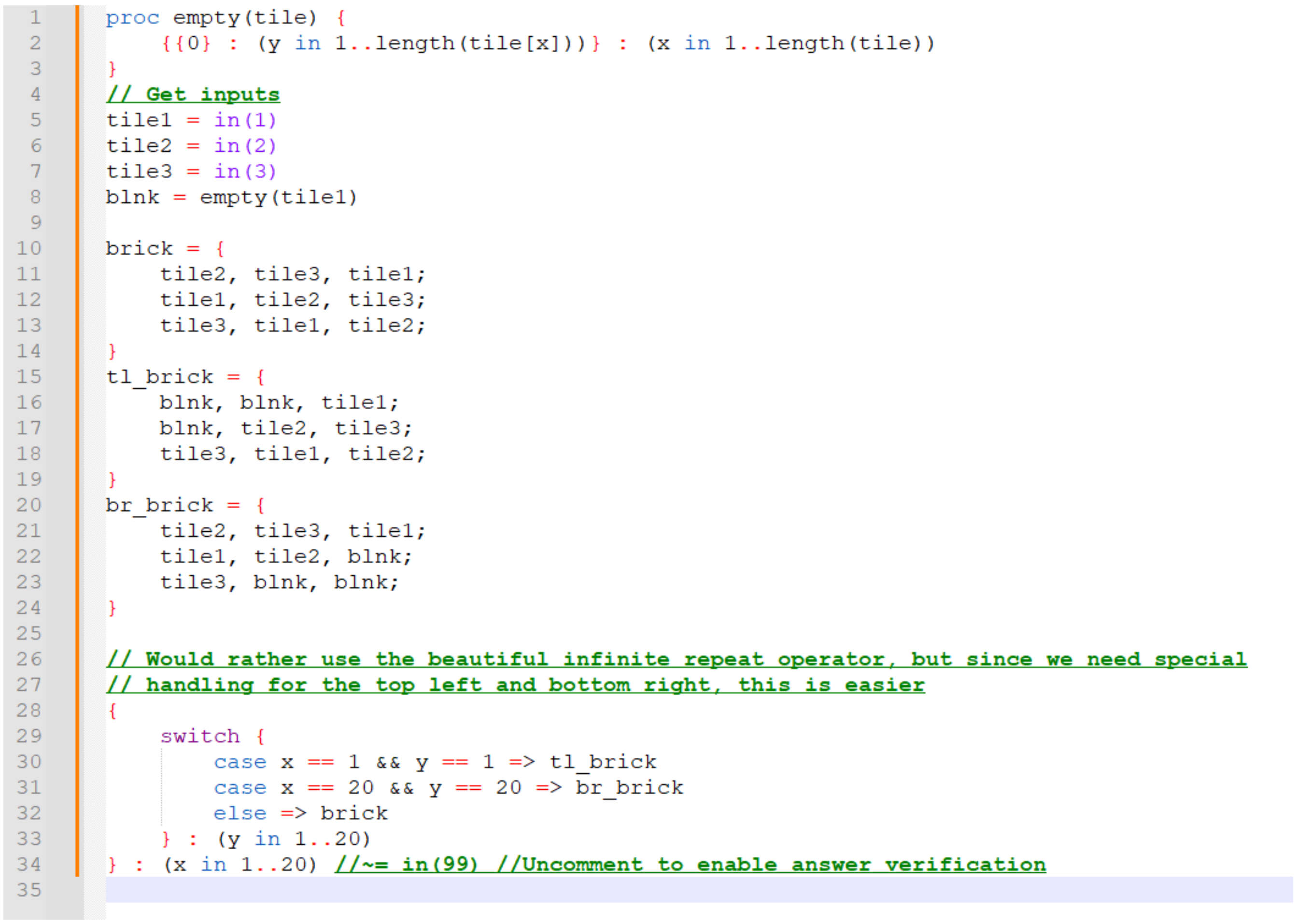
The type checker in our language only serves as a warning to the programmers and does not actually block execution of the program. Internally, the interpreter is written in such a manner that it can evaluate most expressions even if they are poorly typed.

There are some expressions that the type checker struggles to evaluate. Mainly regarding lines and the tile builder across expression. This is the reason we decided to make the type checker only act as a warning, as otherwise it would erronously block execution of programs that can execute.

Final Remarks

Syntax Highlighter

We have a very basic syntax highlighter made with notepad++ which provides the ability to highlight keywords in the language. Below is a screenshot of the syntax highlighter applied to one of our solutions.



```
proc empty(tile) {
    {{0} : (y in 1..length(tile[x]))} : (x in 1..length(tile))
}
// Get inputs
tile1 = in(1)
tile2 = in(2)
tile3 = in(3)
blnk = empty(tile1)

brick = {
    tile2, tile3, tile1;
    tile1, tile2, tile3;
    tile3, tile1, tile2;
}
tl_brick = {
    blnk, blnk, tile1;
    blnk, tile2, tile3;
    tile3, tile1, tile2;
}
br_brick = {
    tile2, tile3, tile1;
    tile1, tile2, blnk;
    tile3, blnk, blnk;
}

// Would rather use the beautiful infinite repeat operator, but since we need special
// handling for the top left and bottom right, this is easier
{
    switch {
        case x == 1 && y == 1 => tl_brick
        case x == 20 && y == 20 => br_brick
        else => brick
    } : (y in 1..20)
} : (x in 1..20) //~= in(99) //Uncomment to enable answer verification

```

Language Critisisms

One of the biggest issues with our language is the inability to set pixels, or regions of pixels, to a specific value. This would have been relatively easy to implement in our language, however was not needed for the initial problem set so was never completed. With hindsights, this is a feature that should be present in a language for manipulating images and not having it lead to some slightly more complex programs in the second problem-set, however the other features provided by the language lessened the impact of this.

There are some bugs in the interpreter which could have been caught with more time and better testing. The TileBuilderAcross sometimes returns its outputs inverted along the X axis, while the SwitchAcross doesn't. This issue is only minor and can be worked around by using the SwitchAcross syntax instead.