COMP2212 Programming Language Concept Coursework

Ziqian Qin (zq4g21@soton.ac.uk)
Zhengyang Zhu (zz22u21@soton.ac.uk)
Xiaopu Hu (Xh9n21@soton.ac.Uk)

Group members share equal contribution.

Syntax – scoping and lexical rules

Language features

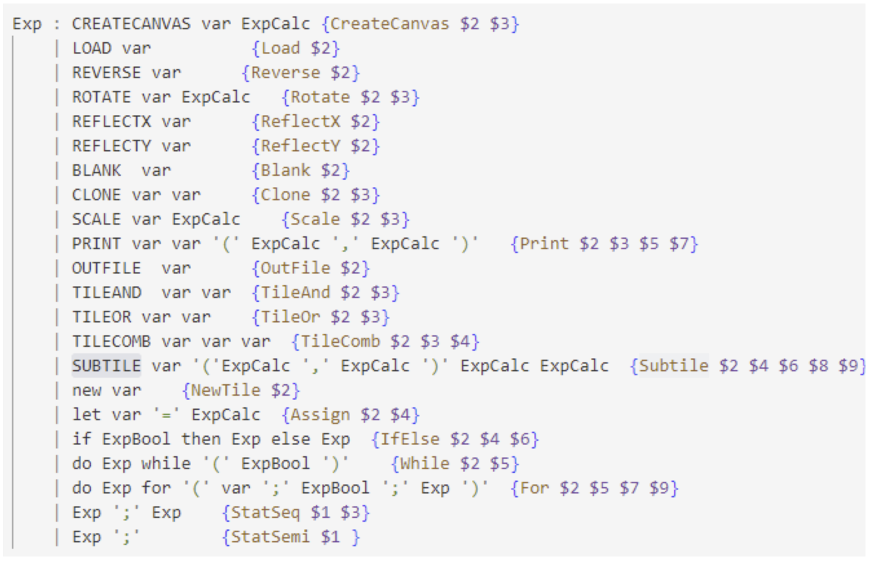
Our language is inspired by SQL in both syntax and functionality. We have built-in support for tile-level operations, one example is **TILEAND tile1 tile2**. It performs **AND** operation on 2 tiles and saves the result to variable tile1. Meanwhile, we also have features commonly available in procedural programming paradigms such as loops. We support both do-for loop and do-while loop. Such syntax allows us to eliminate the use of '{' and '}' symbols.

Parser and Grammar

The parser(**TileGrammar.y**) is developed by Happy tool into haskell. The grammar of the parser is divided into three parts: Exp, ExpCalc, and ExpBool. The Exp, the main grammar for statements, will call ExpCalc or ExpBool when it requires integer or boolean calculations.

Exp:

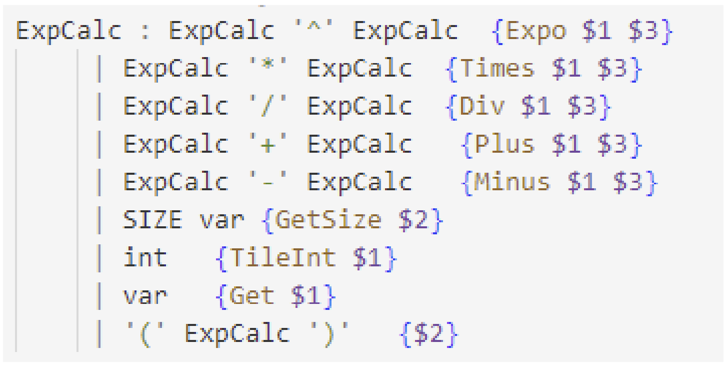
The Exp grammar is the main statement of the programming language, consisting of functional keywords. This part of the grammar is responsible for parsing the primary statements of the language. These statements are the building blocks of the programming language and include variable declarations like **Let**, function calls like **LOAD** and **PRINT**, and control structures like **If-Statement** and **For-Loop**. It also contains essential keywords being used in the evaluation function for determination of an expression that finished its evaluation process(**Cl**), and the termination status of a program, **Output**, similar to the return statement in vast-majority programming languages.



```
Exp : CREATECANVAS var ExpCalc {CreateCanvas $2 $3}
    | LOAD var          {Load $2}
    | REVERSE var      {Reverse $2}
    | ROTATE var ExpCalc   {Rotate $2 $3}
    | REFLECTX var      {ReflectX $2}
    | REFLECTY var      {ReflectY $2}
    | BLANK  var        {Blank $2}
    | CLONE var var     {Clone $2 $3}
    | SCALE var ExpCalc    {Scale $2 $3}
    | PRINT var var '(' ExpCalc ',' ExpCalc ')'   {Print $2 $3 $5 $7}
    | OUTFILE  var      {OutFile $2}
    | TILEAND  var var  {TileAnd $2 $3}
    | TILEOR var var    {TileOr $2 $3}
    | TILECOMB var var var  {TileComb $2 $3 $4}
    | SUBTILE var '('ExpCalc ',' ExpCalc ')' ExpCalc ExpCalc  {Subtile $2 $4 $6 $8 $9}
    | new var    {NewTile $2}
    | let var '=' ExpCalc  {Assign $2 $4}
    | if ExpBool then Exp else Exp  {IfElse $2 $4 $6}
    | do Exp while '(' ExpBool ')'    {While $2 $5}
    | do Exp for '(' var ';' ExpBool ';' Exp ')'  {For $2 $5 $7 $9}
    | Exp ';' Exp    {StatSeq $1 $3}
    | Exp ';'        {StatSemi $1 }
```

ExpCalc:

The ExpCalc grammar is designed for all integer calculations in the programming language. It is responsible for parsing and evaluating expressions that involve arithmetic operations like ***addition, subtraction, multiplication, and division***. This part of the grammar provides the means to perform mathematical calculations, and it allows variables in the calculation by calling the ***Get*** function to get the value of a variable.

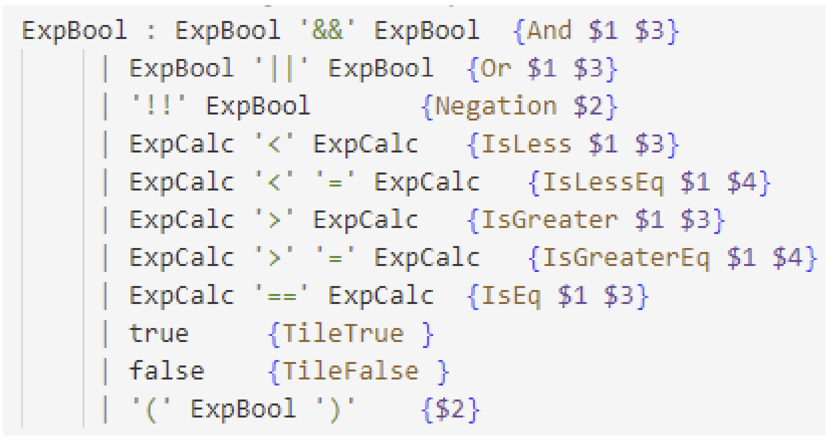


```
ExpCalc : ExpCalc '^' ExpCalc  {Expo $1 $3}
        | ExpCalc '*' ExpCalc  {Times $1 $3}
        | ExpCalc '/' ExpCalc  {Div $1 $3}
        | ExpCalc '+' ExpCalc   {Plus $1 $3}
        | ExpCalc '-' ExpCalc   {Minus $1 $3}
        | SIZE var {GetSize $2}
        | int   {TileInt $1}
        | var   {Get $1}
        | '(' ExpCalc ')'   {$2}
```

ExpBool:

The ExpBool grammar is designed for all boolean calculations in the programming language. It is responsible for parsing and evaluating expressions that involve logical operations like ***And, Or,*** and ***Not.*** This part of the grammar is essential in determining the flow of control in the program, enabling it to make decisions based on specific conditions.

The parser takes the input code and applies the rules of the grammar to parse it into a data structure that represents the program's syntax tree. The resulting data structure is then passed on to the **Eval.hs**, which interprets the program's logic and produces the output.



```
ExpBool : ExpBool '&&' ExpBool  {And $1 $3}
        | ExpBool '||' ExpBool  {Or $1 $3}
        | '!!' ExpBool       {Negation $2}
        | ExpCalc '<' ExpCalc   {IsLess $1 $3}
        | ExpCalc '<' '=' ExpCalc   {IsLessEq $1 $4}
        | ExpCalc '>' ExpCalc   {IsGreater $1 $3}
        | ExpCalc '>' '=' ExpCalc   {IsGreaterEq $1 $4}
        | ExpCalc '==' ExpCalc  {IsEq $1 $3}
        | true     {TileTrue }
        | false    {TileFalse }
        | '(' ExpBool ')'    {$2}
```

Execution Model

Our interpreter(Tsl.hs) would read an input file line by line. It first calls the parser to tokenize it then calls the evaluation function in **Eval.hs** to calculate and obtain the result.

We have implemented a CEK machine like mechanism when evaluating the program. Such mechanisms have three fields: the current expression being evaluated, the environment and the continuation. An entry function(**letsEval**) to the evaluation program would check whether the program is terminated by the following three conditions. Whether the expression remains the same if

we evaluate it further, whether it is intended to be a finished expression(**Cl**, **Output**, **OutFile**) and whether the continuation stack is empty. In the case that the program reaches the end, the evaluated result will be returned and parsed by function **parseOutput** for output. On the other hand, if the current CEK machine does not fully qualify the termination status, it will be evaluated further.
In our implementation of CEK, the environment is a pair of String and list of strings. The first element stores the name of the variable and the second element stores its content.

For instance, if a variable named *i* is assigned value 1, a corresponding environment representing such a relationship would be [("i", ["1"])]. On retrieval, only the latest binding gets returned. This mechanism allows simply adding the new binding pair at the start of the environment and shadows its old binding.

The continuation is a list of frames that keeps track of expressions waiting to be evaluated. It could be an unfinished loop, assignment waiting to be executed, printing a tile to the canvas or code after a ;. Once the current expression reaches a termination status, the frame on top of the continuation gets popped out and sent to be evaluated.

Type checking

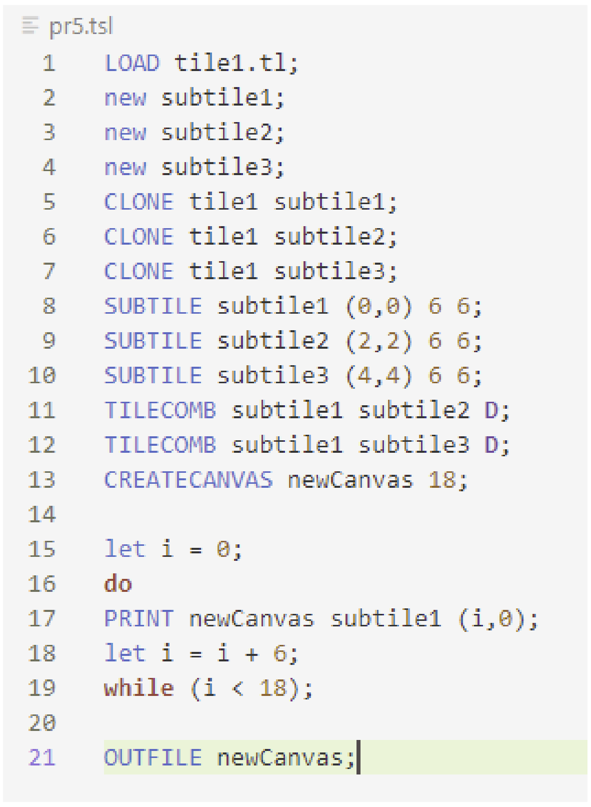
Our language is designed to be a strong and static type language.

A static type programming language is a language where the data types of variables and values are known at compile-time and the type-checking occurs at compile-time as well. We apply the type checking to the parser so that during the construction of the **Abstract Syntax Tree**, the parser can perform type checking by verifying that the types of the expressions and identifiers used in the program are consistent with the types expected by the language.

A strong type programming language is a language where the data types of variables and values are strictly enforced, and type-conversion must be explicit. The basic data type we have in the language is **Boolean, Integer** and **Tile**. It is defined strictly and stored separately in the environment. The interpreter will check whether the data type is correct before run time.

Programming Convenience

Syntax highlighting

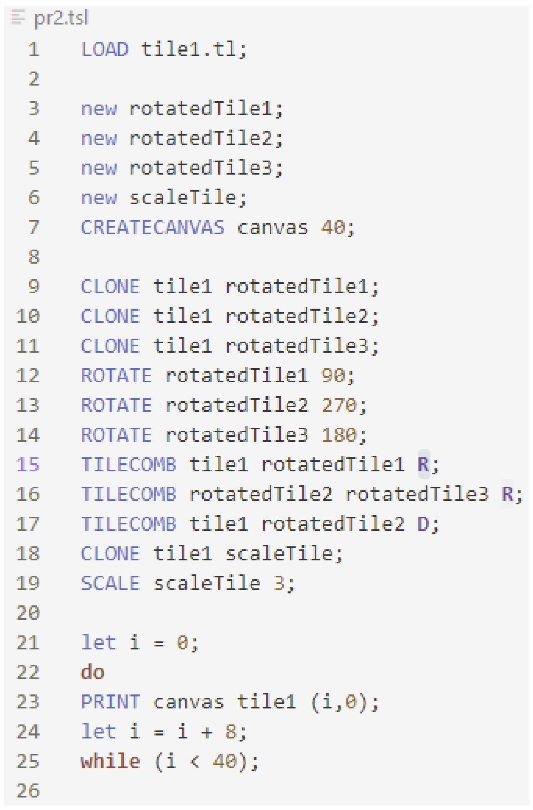


pr5.tsl

```
LOAD tile1.tl;
new subtile1;
new subtile2;
new subtile3;
CLONE tile1 subtile1;
CLONE tile1 subtile2;
CLONE tile1 subtile3;
SUBTILE subtile1 (0,0) 6 6;
SUBTILE subtile2 (2,2) 6 6;
SUBTILE subtile3 (4,4) 6 6;
TILECOMB subtile1 subtile2 D;
TILECOMB subtile1 subtile3 D;
CREATECANVAS newCanvas 18;

let i = 0;
do
PRINT newCanvas subtile1 (i,0);
let i = i + 6;
while (i < 18);

OUTFILE newCanvas;
```



pr2.tsl

```
LOAD tile1.tl;

new rotatedTile1;
new rotatedTile2;
new rotatedTile3;
new scaleTile;
CREATECANVAS canvas 40;

CLONE tile1 rotatedTile1;
CLONE tile1 rotatedTile2;
CLONE tile1 rotatedTile3;
ROTATE rotatedTile1 90;
ROTATE rotatedTile2 270;
ROTATE rotatedTile3 180;
TILECOMB tile1 rotatedTile1 R;
TILECOMB rotatedTile2 rotatedTile3 R;
TILECOMB tile1 rotatedTile2 D;
CLONE tile1 scaleTile;
SCALE scaleTile 3;

let i = 0;
do
PRINT canvas tile1 (i,0);
let i = i + 8;
while (i < 40);

```

We developed a Visual Studio Code extension that implements the syntax highlight feature. It uses Visual Studio Code's tokenization engine to categorize the code into different patterns. Then let the theme in Visual Studio Code assign a specific color to each category.
This extension is available at https://marketplace.visualstudio.com/items?itemName=paji.tsl-highlight

Informative error messages

Our interpreter employs 4 different error messages. First is the variable not found error. If the interpreter can't find a variable in the environment during execution, it would output the following message to stderr: "run time error. variable {var_name} not found" where {var_name} is the placeholder for the name of the variable that is not present in the environment. The second one is when using the ROTATE function with a parameter that is not divisible by 90. The following error message will be printed to stderr: "invalid degree". And the third error message is when calling the TILECOMB function with parameters other than specified L,R,U,D. The message "Invalid Direction" will be printed to stderr. Last error message is "Error: Unexpected line, Last line of program should be OutFile". This one is very self-explanatory. It occurs when the last statement of the program does not function OUTFILE. A scenario similar to missing return statements in Java.