Triangle Manipulation Language TrML

XUECHEN FENG (XF2120)
QISHU CHEN (QC2166)
YU WAN (YW2506)
LIANHAO QU (LQ2140)
WANQIU ZHANG (WZ2241)

Dec 19, 2012

Contents

1 Introduction to TrML

TrML is a programming language that allows user express trigonometry concept, and construct/solve complex trigonometry puzzles. Trigonometry induction problem can be as simple as naïve substitution and subtraction, yet sometimes it could drive people crazy with its subtle logic and hidden clues. We believe most science students share the painful process of solving trigonometry induction problems in high math classes.

The language could be used for educational purpose for both geometric class and entry level programming class. This report provides a comprehensive description of the TrML language.

Motivation

Trigonometry induction problems can be as simple as substitution and subtraction, yet these seemingly elementary problems could become tangled and messy due to its subtle logic and hidden deductions. We believe most students share the painful process of solving trigonometry induction problems in higher level math classes.

Overview

Though there exist some calculators (GUI) that help with certain simple calculation problems, TrML is more comprehensive. What is more, as a simple programming language, TrML allow users to practice simple programming as well as understanding trigonometry from a logical perspective. The language could be used for educational purpose for both geometric and entry level programming class.

2 Language Tutorial

TrML was designed with the intention of simplifying and clarifying trigonometry for users. The following tutorial will act as a quick guide to users and will cover the basics of the language as well as give out helpful samples.

2.1 Basic

We will start by exploring the fundamental concepts of the TrML language.

2.1.1 Comments

Comments are notes programmer wrote in the code. They are ignored by the compiler.

The character @ introduces a comment, which terminate with the new-line character \n. Comments do not nest and they do not occur within string. In order to read code better, comments will be shown in blue in this document.

```
@ this is a comment and it will be ignored
```

2.1.2 Declarations

There are two data types in TrML: value and triangle. Value is a floating point number, and triangle is a triangle in 2D plane. Here is how to declare values:

```
@ declaring a value called 'i'
value i;
@ declaring a value called 'sum'
value sum;
```

Here is how to declare triangles:

```
@ declaring a triangle called 'ABC'
triangle ABC V [( , ),( , ),( , )];
@ declaring a triangle called 'DEF'
triangle DEF L [ , , ];
```

2.1.3 Assignment

The variable and expression must have the same type.

```
@assign 'i' the value of 4.0
value i 4.0;
```

```
@assign three vertex values to triangle ABC
triangle ABC V [(1.1, 2.2),(3.3, 4.4),(5.5, 6.6)];
@assign three side-length values to triangle DEF
triangle DEF L [4.2, 3.5, 3.6];
```

2.1.4 Expressions

An expression is some operation that can be executed and evaluated.

```
@Arithmetic expression
4.0*5.0
@Logical expression
(Tri_1.sideA == Tri_2.sideA) && (Tri_1.sideB == Tri_2.sideB)
```

2.1.5 Functions and Statements

```
@A while statement
while(i > 0.0){
        sum = sum + i;
        i = i - 1.0;
}
@An if statement
if(sin(i)==sin(1.0)){
    prints("True");
    result = sin(num);
    printv(result);
}
@Note that prints(for string-constant) and printv(for arithmetic-
expression) are built-in functions
```

2.2 Examples

Here is the "Hello World!" code:

```
initialize:

rules:

operations:

prints("Hello \nWorld!\n");
```

A more complicated and comprehensive example:

```
@ keyword “initialize:” starts triangle initialization phase
initialize:
@ initialize triangle with 2-D vertex location
triangle ABC V [(1.1, 2.2) , (3.3, 4.4) , (5.5, 6.6)];
@initialize triangle with line segment length
triangle DEF L [4.2, 3.5, 3.6];
value agl 10.0;

@ Keyword “rules:” starts rules construction phase
rules:
@ Explain regular triangle’s meaning in terms of line length.
@ This is a judgment rule
identical_triangle (triangle Tri_1, triangle Tri_2) (
((Tri_1.sideA == Tri_2.sideA ) && (Tri_1. sideB == Tri_2. sideB) &&
(Tri_1. sideC == Tri_2. sideC)) ||
((Tri_1.sideA == Tri_2.sideA ) && (Tri_1. sideB == Tri_2. sideC) &&
(Tri_1. sideC == Tri_2. sideB)) ||
((Tri_1. sideA == Tri_2. sideB) && (Tri_1. sideB == Tri_2. sideC) &&
(Tri_1. sideC == Tri_2. sideA)) ||
((Tri_1. sideA == Tri_2. sideB) && (Tri_1. sideB == Tri_2. sideA) &&
(Tri_1. sideC == Tri_2. sideC)) ||
```

```
((Tri_1. sideA == Tri_2. sideC) && (Tri_1. sideB == Tri_2. sideA)
&&(Tri_1. sideC == Tri_2. sideB)) ||
((Tri_1. sideA == Tri_2. sideC) && (Tri_1. sideB == Tri_2. sideB)
&&(Tri_1. sideC == Tri_2. sideA))) {true};
@ Explain right triangle's meaning in terms of angle
regular_triangle (triangle ABC) (ABC.sideA == ABC.sideB && ABC.sideB
== ABC.sideC) {true};
regular_triangle (triangle ABC) (ABC.angleA == 60.0 && ABC.angleB ==
60.0) {true};
@ Explain what means of two triangles be equal
equal_triangle (triangle ABC, triangle DEF) ( ABC.sideA == DEF.sideA
&& ABC.sideB == DEF.sibeB && ABC.sideC== DEF.sideC) {true};

@ Explain angleC in terms of sides
@ This is a calculation rule
angle_C (triangle ABC) (true) {arccos((ABC.sideA * ABC.sideA) +
(ABC.sideB * ABC.sideB) - (ABC.sideC* ABC.sideC) / 2.0 * ABC.sideA *
ABC.sideB)};

@ keyword "operations:" starts operation and calculation phase
operations:
if (identical_triangle (triangle ABC, triangle DEF)) {
prints ("ABC and DEF are identical");
}
if (regular_triangle(triangle ABC))
printv (triangle ABC.sideA);
prints ("is regular triangle");
while (agl < 180.0){
agl = agl + 5.0;
prints ("value agl's valus is: ");
printv ( agl );
}
```

2.3 Compilation

In order to compile a TrML file and pass it into the TrML compiler, run the make file as follows:

```
OBJS = scanner.cmo parser.cmo compiler.cmo

compiler : $(OBJS)
        ocamlc -o compiler $(OBJS)
scanner.ml : scanner.mll
        ocamllex scanner.mll
        ocamlc -c ast.ml
parser.ml parser.mli : parser.mly
        ocamlyacc parser.mly
%.cmo : %.ml
        ocamlc -c $<
%.cmi : %.mli
        ocamlc -c $<
TARFILES = Makefile ast.ml parser.mly scanner.mll compiler.ml
.PHONY : clean
clean :
        rm -rf *.cmo *.cmi compiler parser.mli parser.ml scanner.ml
# generated by ocamldep *.ml *.mli
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
ast.cmi:
parser.cmi: ast.cmi
```

2.4 Installation

Unzip the tar.gz archive and use the source files.

3 Language Reference Manual

3.1 Lexical Conventions

TrML has six kinds of tokens: keywords, identifiers, constants, string literals, operators, and separators. Whitespace such as blanks, tabs, and newlines are ignored except when they serve to separate tokens. Comments are also ignored.

3.1.1 Comments

The character @ introduces a comment, which terminate with the new-line character \n. Comments do not nest, and they do not occur within string.

3.1.2 Identifiers

An identifier is a series of alphabetical letters and digits; the first character must be alphabetic and it must have at least 3 characters in length.

3.1.3 Keywords

The following identifiers are res reserved for the use as keywords, and may not be used otherwise:

triangle	true	false	initialize
rules	operations	while	if
value	sin	cos	tan
arcsin	arccos	arctan	sqrt
prints	printv	angleA	angleB
angleC	sideA	sideB	sideC
V	L		

3.1.4 Constants

There are five constants in TrML:

- ***Value***

Value constant consists of an integer part, a decimal point and a fraction part. No exponential part is supported. All values should be larger than 0 or it will be considered as unknown.

- ***Triangle***

Triangle constant can be either Vertex-constant starts with letter V followed by three tuples of values in square brackets. i.e. V[(1.1, 2.2), (3.3, 4.4), (1.2, 4.2)] or Line-segment-constant starts with letter L followed by three values in square brackets. i.e. L[1.3, 4.2, 5.4] or both i.e. V[(1.1, 2.2), (3.3, 4.4), (1.2, 4.2)] L[1.3, 4.2, 5.4].

- ***Boolean***

The reserved Boolean constants are true and false.

- ***Sh-cat***

The reserved Sh-cat constant is character _ . As Schrödinger's cat means there is a field to be filled but the value in corresponding field is unknown.

- ***String***

String constant is a sequence of characters surrounded by double quotes, as in "…". A string's value is initialized with the given characters. Only alphabet and number are allowed in string.

3.2 Types

There are two data types in TrML:

Value	A floating point number
Triangle	A Triangle in 2D plane

3.3 Operators

TrML support most of the standard C Programming Language's arithmetic operations and inherit their standard operator.

Operators' Precedence and order of evaluation:

Operators	Associativity
* /	left to right
+ -	left to right
>= <= > <	left to right
= = ! = !!	left to right
&& ||	left to right
=	right to left

3.4 Expression and Operations

3.4.1 Expression

Expression consists of numerical expression and general expression

- ***Numerical Expression***

Numerical expressions include subtraction, addition, multiplication, division, and mathematical operations such as sin, cos, tan .etc.

- ***General Expression***

General expressions include Boolean expressions such as &&, ||, !!, and numerical expressions.

3.4.2 Operations

Operations is a block of operations. An operation consists of while operation, conditional operation, assignment and built-in function calls such as print value and print string.

3.5 Building Blocks

The program consists of three building blocks initialization, rules and operations

3.5.1 Initialization

Initialization declares all variables used in the program. A variable may be a value or a triangle. All variable declarations end with semicolon.

value-declaration:

value-identifier value-constant

triangle-declaration:

triangle-identifier vertex-constant

triangle-identifier line-segment-constant

triangle-identifier vertex-constant line-segment-constant

Noting that a triangle with empty information could be created by using sh-cat.

3.5.2 Rules

Trigonometry rules are defined in the rules section. Rule declaration starts with rule name followed by parameters, conditions as rules qualification and statements as operations.

rules:

identifier (parameter_ list) (expression) {statements}

Note that the there are two different rules in trigonometry: judgment rule and calculation rule. Both rules can be expressed using the same rule expression. If statements equal "true", then it is a judgment rule. Otherwise, it is a calculation rule.

3.5.3 Operations

Operations invoke rules from previous block and variables from initialization block. Operations consist of a series of statements, including conditional statements, while statement, rule calls, assignment statement and built-in functions calls.

4 Project Plan

4.1 Process

Initial planning and implementation process occurred during weekly group meetings. Members collaborated in person as well as on Google Drive. Everyone contributed to the high-level language design.

Members used various text editors during the development process. The compiler was written in O'Caml and the byte-code interpreter was written in Java. Tests cases were created early and team members all contributed to this process.

4.2 Roles and Responsibilities

Everyone contributed to writing the initial proposal, LRM, and the final document. There were much cross-contribution during the implementation stage. Here are the assigned responsibilities:

- Xuechen Feng
  language conception and team leader
  primary on scanner and compiler
- Qishu Chen
  primary on scanner, parser, compiler and report
- Lianhao Qu
  primary on byte code interpreter and test cases
- Yu Wan
  primary on reports, test cases and debugging
- Wanqiu Zhang
  primary on scanner, parser, compiler and report

4.3 Tools and Languages

For byte code generation, we use O'caml, ocamlyacc, ocamllex. For byte code interpreter, we use Java. Various text editors were used for writing code. Google Drive and Email were used for discussions and real-time collaboration. Google Drive was used for version control.

4.4 Project Log

Sep. 14 First meeting, discussing ideas

Sep. 21 Second meeting, finalizing proposal

Sep. 28 Discussing in detail: language features and scope

Oct. 5 Deciding Grammar

Oct. 26 Skeleton of parser written, finalizing LRM

Nov. 16 Initial tests written in advance

Nov. 30 Completing scanner

Dec. 14 Working on test cases and parser

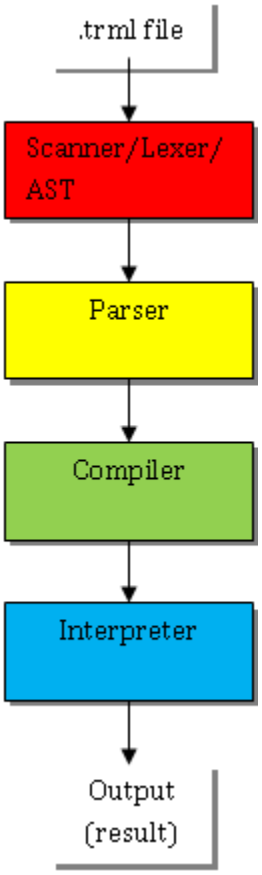
Dec. 17 Working on compiler and code generation

Dec. 18 Completed debugging, byte code interpreter and final report

5 Architectural Design

5.1 Block Diagram

TrML is implemented using a standard model for compilers. The input .trml file goes through our specific scanner, parser, compiler, interpreter, and the result of the query will be the output.



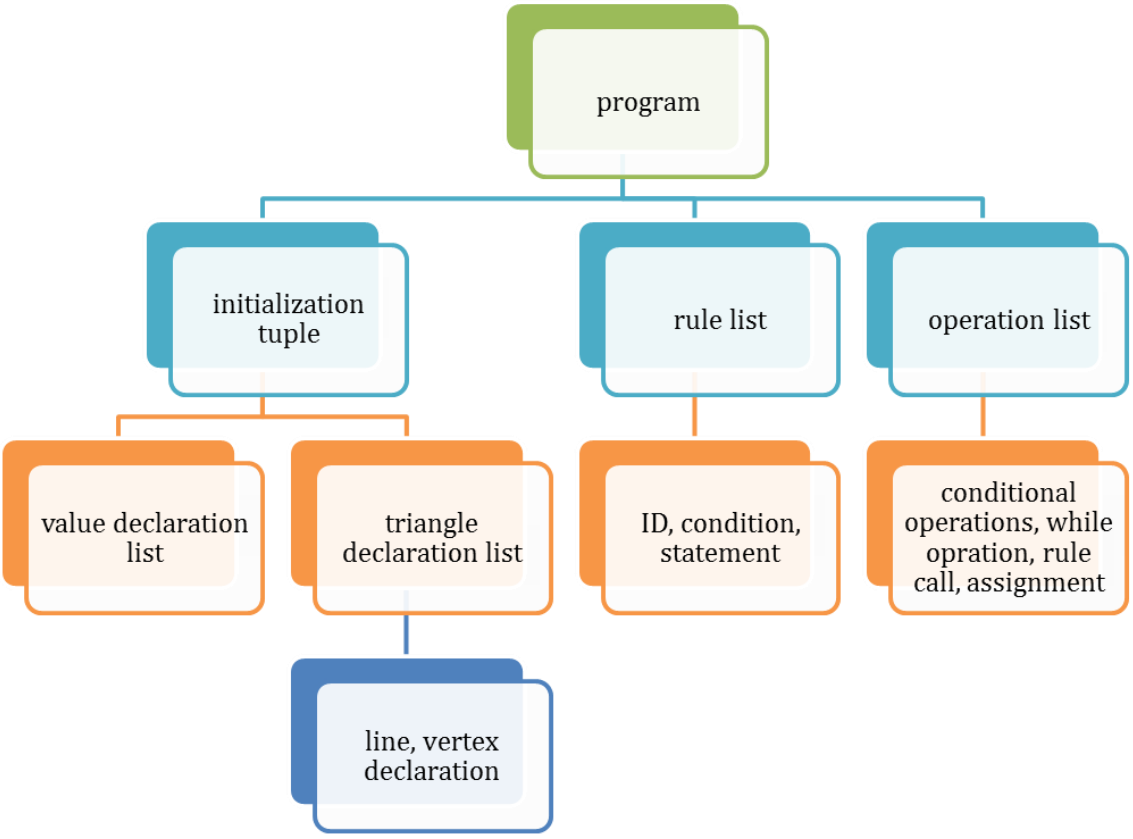
5.2 Components

5.2.1 Scanner

The scanner received the original code and transforms it into a stream of tokens. It breaks up the input stream using ocamllex.

5.2.2 Parser

TrML uses ocamlyacc to generate abstract syntax tree and test syntax of the original code.



5.2.3 Symbol Tables

Variable table: all the global variables are stored in variable table. It is a string map with variable name (key) and variable index (value).

Rule tables: all rules are stored in rule table. It is a string map with rule name (key) and rule entry point (value).

5.2.4 Abstract Syntax Tree

```
type op = And | Or | Equal | Neq | Less| Greater| Leq | Geq
type op2 = Plus | Minus | Times | Divide
type op3 = Sin | Cos| Tan| Arcs | Arcc| Arct |Sqrt
```

```
type argument ={
                atp: string;
                vid: string;
                tid: string;
}

type parameter = {
        ptp : string;
        pname : string;
}

type nexpr =
        Num of float
        | Bool of string
        | Val of string
        | Tri_ele of string * int
        | Binop2 of nexpr * op2 * nexpr
        | Monop of op3 * nexpr

type expr =
        Binop of expr * op * expr
        | Nexpr of nexpr
        | Rev of expr

type oexpr =
        Expr of expr
|   Rulec of string * parameter list

type operation =
        Block of operation list
        | If of oexpr * operation
        | While of oexpr * operation
        | Prints of string
        | Printv of oexpr
        | Assign of string * oexpr
```

```
type lines = {
        alen : float;
        blen : float;
        clen : float; }

type vertexes = {
        avtxx : float;
        avtxy : float;
        bvtxx : float;
        bvtxy : float;
        cvtxx : float;
        cvtxy : float; }

type tri_decl = {
        tname : string;
        tline : lines;
        tvertex : vertexes;}

type v_decl = {
        vname: string;
        value: float;
}

type rule_decl = {
        rname : string;
        paras : parameter list;
        cond : expr;
        body :  nexpr;
}

type decl = v_decl list * tri_decl list

type program = decl * rule_decl list *operation list
```

5.2.5 Compiler

The complier mainly functions to read the AST tree and generate byte-code. The structure of the complier is declared as follows:

- Global variable storage

Read all the variables in value declaration list and print the values at the top of the byte code file; at the same time store those variables' names and indexes in variable symbol table. And then read all the variables in triangle declaration list and print each element of each triangle at the top of the byte code file including triangle vertices coordinates and side lengths; at the same time, store those triangles' names and indexes in variable symbol table. Noting that for the variables with unknown fields we set the corresponding value to 0.

- Rule memory segment

Local variables are the copies of parameters in TrML. Each local variable is referred by the corresponding location of its name in parameter list. As argument, local variable is only valid within the rule's scope. When the rule is called in operations, global variables are pushed into stack. After the program changes state, arguments are popped out of the stack and stored inside the local variable table.

- Rules storage

Every rule has its own block of code. The name and the entry point of each rule are recorded in string map. Thus, when we use a rule, we can just it by its entry point. Both rule's arguments and the byte code is stored in the rules storage, the format is data field followed by the code chunk with a standard FP-SP alike layout. An entry point is calculated based on the length of the code segment and the length of the arguments.

- Byte-code layout

The byte-code consists of three portions.

Global variable portion: all the value is listed from the beginning of the file, values fields are listed before triangle fields. Each value field occupies one line and one memory space during run time. The interpreter reads this portion sequentially till it runs into separator "rul" and adds contents from each line as one individual data field.

The second part of the file is used to store customized rule as well as its arguments. The entry point is located between argument data field and the rule code chunk. The interpreter runs along the line till it hit separator "opt" all lines is stored inside the rule memory segment.

The third part stores operation sequence. Again the representation of the byte code segment is sequential; it is located after the rules segment. Noted during runtime, two program-counters (PC and RC) are maintained so that the code jumps between two states to keep tracks of two different activate locations for code segment. Global variable is available throughout the scope while argument is only visible inside the rule. Two system print calls are implemented and the string is stored outside of normal memory location.

5.2.6 Interpreter

lod -- load
load global variable from location [num], then push to the register stack

str -- store
pop the register stack and then store it into global variable with location [num]

pop -- pop
load local variable from location [num], then push to the register stack

psh -- push
pop the register stack and then store it into local variable with location [num]

bra -- branch
change the program/rule counter with absolute [num], then branch to the new location

bne -- branch not equal to zero
pop the register stack and check the number whether equals to zero. If no, branch, same as **bra**

beq -- branch equal to zero
pop the register stack and check the number whether equals to zero. If yes, branch, same as **bra**

not -- logic not
peek the register stack, if it is 0.0, change it to 1.0, otherwise, change it to 0.0(if it is a string, also change to 0.0)

add -- addition
pop two numbers from register stack, add them, then push the result back to the register stack.

sub -- subtraction
pop two numbers from register stack, subtract them, then push the result back to the register stack.

mul -- multiplication
pop two numbers from register stack, multiply them, then push the result back to the register stack.

div -- division
pop two numbers from register stack, divide them, then push the result back to the register stack.

addi --addition immediate
pop one number from register stack, add it with [num], then push the result back to the register stack

subi -- subtraction immediate
pop one number from register stack, subtract it with [num], then push the result back to the register stack

muli – multiplication immediate
pop one number from register stack, multiply it with [num], then push the result back to the register stack

divi -- division immediate
pop one number from register stack, divide it with [num], then push the result back to the register stack

sqt -- square root
pop one number from register stack, calculate the square root, then push the result back to the register stack.

sin -- sin
pop one number from register stack, calculate the sin of it, then push the result back to the register stack.

cos -- cos
pop one number from register stack, calculate the cos of it, then push the result back to the register stack.

tan -- tan
pop one number from register stack, calculate the tan of it, then push the result back to the register stack.

acs -- arcsin
pop one number from register stack, calculate the arcsin of it, then push the result back to the register stack.

acc -- arccos
pop one number from register stack, calculate the arccos of it, then push the result back to the register stack.

act -- arctan
pop one number from register stack, calculate the arctan of it, then push the result back to the register stack.

ptv -- print value
pop out the first item from register stack, then println this item.

pts -- print string
print out the string after pts command.

rc -- function call
store current program counter, switch to the rule counter, with absolute location of [num]

rt -- return
jump back to the stored program counter+1

hlt -- stop program
stop the program

6 Test Plan

6.1 Example Programs

Test Sample Program:

```
initialize:
triangle DEF L [1.0, 1.0, 1.0];
value agl 10.0;

rules:
angle_C (triangle ABC) (true) {arccos(((triangle ABC.sideA * triangle ABC.sideA) + (triangle ABC.sideB * triangle ABC.sideB) - (triangle ABC.sideC * triangle ABC.sideC)) / 2.0 * triangle ABC.sideA *triangle ABC.sideB)};

operations:
agl = rule angle_C (triangle DEF);
printv (value agl);
```

Output from compiler:

```
10.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
1.000000
1.000000
1.000000
rul
pop 0
pop 1
pop 2
pop 3
pop 4
pop 5
pop 6
pop 7
pop 8
pop 8
pop 7
pop 6
pop 5
pop 4
pop 3
pop 2
pop 1
pop 0
ldf 1.0
bne 2
rtn
psh 6
```

```
psh 6
mul
psh 7
psh 7
mul
add
psh 8
psh 8
mul
sub
ldf 2.
div
psh 6
mul
psh 7
mul
acc
rtn
opt
lod 1
lod 2
lod 3
lod 4
lod 5
lod 6
lod 7
lod 8
lod 9
rc 9
str 0
lod 0
ptv
hlt
```

Output result from interpreter (which is just 60 degree of that angle defined in our rule):

```
1.0471975511965979
```

6.2 Test Suites

Test cases were members' best effort to ensure that all TrML features were covered by at least one test case. These cases work quite simply. The testing strategy is "bottom-up": Test cases were written for the simplest functionality, then afterwards, the complexity of these test cases were increased while implementing more of the grammar.

7 Lessons Learned

7.1 Reflection

Finishing this project is not our only goal and definitely not our only gain. This project helps us to review knowledge learned in classes and put it into practice. Before the class, many of us have little understanding of functional language, but now through learning and coding in O'Caml, we become more familiar with this language and functional languages in general. In addition, we get knowledge of the structure of a compiler and its working strategies. And since we work in a group, we met excellent group members and we have become good friends. Corporations between group members are essential during this project and this enhanced our teamwork ability. Although we've been through lot difficulties, every time we overcome one, we can see our progress. Meanwhile, there're some lessons learned through this project. First, starting early is always a good idea. Everyone agree to the idea of starting early, however, actually carrying out this idea is much harder. It is fortunate that the group never had problem reaching any member. A possible improvement could be creating a strict deadline for the team as a whole and each of the team members. Another suggestion for future references is delegating tasks early on. A big challenge aside from timing our project was pacing. Developing a balanced schedule was quite necessary, and it will help the group as well as each group member

with time allocation. It was a good idea to finalize the group's Language Reference Manual before coding, because finalizing general ideas and concerns before implementing really help promote efficient implementation of the language. Last but not least, members will need to be able to work independently and the interface of each piece of code should be well decided. In this project we've met many problems when combing code of different group members.

7.2 Future development

- Type checking

In this project we can check whether a rule or global variable is existed, however, we didn't check the type of arguments when do function calls, thus in future development, we should check the type of each argument to see whether it match it in rule declaration and also map the length of argument list with the parameter list.

- Nested rule calls

So far in rule declaration, we cannot call another rule, so in future we want to implement nested rule calls for adding more functionalities and providing convenience to user.

- Exception handling

We didn't handle exceptions such as typo error and wrong declarations; we can do more exception handling to provide user convenience in debugging.

8 Appendix-Complete Code Reference

scanner.mll (mainly written by Qishu Chen and Wanqiu Zhang)

```
{open Parser}

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0' - '9']

rule token =
      parse [' ' '\t' '\r' '\n']              { token lexbuf }
      | "triangle"                            { TRIANGLE }
      | "value"                             { VALUE }
      | "true" as tvalue                      { BOOL(tvalue) }
      | "false" as fvalue                     { BOOL(fvalue) }
      | "initialize"                          { INITIALIZATION }
      | "rules"                             { RULES }
      | "operations"                          { OPERATIONS }
      | "rule"                              {RULE}
      | "while"                             { WHILE }
      | "if"                                  { IF }
      | '_'                                 { WILDCAT }
      | "&&"                                  { AND }
      | "=="                                  { EQ }
      | "||"                                  { OR }
      | '='                                 {ASSIGN}
      | "!!"                                  { NOT }
      | "sin"                                 { SIN }
      | "cos"                                 { COS }
      | "tan"                                 { TAN }
      | "arcsin"                            { ARCSIN }
      | "arccos"                            { ARCCOS }
      | "arctan"                            { ARCTAN }
      | "sqrt"                              { SQRT }
      | 'V'                                 { VERTEX }
      | 'L'                                 { LINE }
```

```
      | "prints"                          { PRINTS }
      | "printv"                          { PRINTV }
      | "Ax"                                    { AX }
      | "Ay"                                    { AY }
      | "Bx"                                    { BX }
      | "By"                                    { BY }
      | "Cx"                                    { CX }
      | "CY"                                    { CY }
      | "sideA"                         { SIDEA }
      | "sideB"                         { SIDEB }
      | "sideC"                         { SIDEC }
      | '('                             { LPAREN }
      | ')'                             { RPAREN }
      | '['                             { LBRACKET }
      | ']'                             { RBRACKET }
      | '{'                             { LBRACE }
      | '}'                             { RBRACE }
      | '.'                             { DOT }
      | ':'                             { COLON }
      | ';'                             { SEMI}
      | ','                             { COMMA}
      | '+'                             { PLUS }
      | '-'                             { MINUS }
      | '/'                             { DIVIDE }
      | '*'                             { TIMES }
      | "!="                                 { NEQ }
      | ">="                                 { GEQ}
      | '>'                             { GT }
        | "<="                               { LEQ}
      | '<'                             { LT }
      | '\"'[^'\"']* '\"' as astring
      { STRING(astring) }
      |  (letter | '_') (letter | digit | '_') (letter | '_' |
digit)* as id          { ID(id) }
      | digit+ '.' digit+  as value
      { NUM(float_of_string value) }
```

```
      | ("true" | "false") as bvalue
      { BOOL(bvalue) }
      | '@'                                    { comment lexbuf }
      | eof                                    { EOF }

and comment =
      parse '\n'                    { token lexbuf }
      | _                           { comment lexbuf }
```

ast.ml (mainly written by Qishu Chen and Wanqiu Zhang)

```
type op = And | Or | Equal | Neq | Less| Greater| Leq | Geq
type op2 = Plus | Minus | Times | Divide
type op3 = Sin | Cos| Tan| Arcs | Arcc| Arct |Sqrt

type argument ={
            atp: string;
            vid: string;
            tid: string;
}

type parameter = {
      ptp : string;
      pname : string;
}

type nexpr =
      Num of float
      | Bool of string
      | Val of string
      | Tri_ele of string * int
      | Binop2 of nexpr * op2 * nexpr
      | Monop of op3 * nexpr

type expr =
```

```
      Binop of expr * op * expr
      | Nexpr of nexpr
      | Rev of expr

type oexpr =
      Expr of expr
|   Rulec of string * parameter list

type operation =
      Block of operation list
      | If of expr * operation
      | While of expr * operation
      | Prints of string
      | Printv of expr
      | Assign of string * oexpr

type lines = {
      alen : float;
      blen : float;
      clen : float; }

type vertexes = {
      avtxx : float;
      avtxy : float;
      bvtxx : float;
      bvtxy : float;
      cvtxx : float;
      cvtxy : float; }

type tri_decl = {
      tname : string;
      tline : lines;
      tvertex : vertexes;}

type v_decl = {
      vname: string;
```

```
      value: float;
}

type rule_decl = {
      rname : string;
      paras : parameter list;
      cond : expr;
      body :      nexpr;
}

type decl = v_decl list * tri_decl list

type program = decl * rule_decl list *operation list
```

parser.mly (mainly written by Qishu Chen and Wanqiu Zhang)

```
%{ open Ast %}

%token      INITIALIZATION  RULES  OPERATIONS EOF
%token      RULE
%token      VALUE
%token      WHILE  IF WILDCAT  PRINTS  PRINTV
%token      TRIANGLE  VERTEX  LINE
%token      AX AY BX BY CX CY SIDEA  SIDEB  SIDEC
%token      ASSIGN SIN  COS  TAN  ARCSIN  ARCCOS  ARCTAN  SQRT
%token      LPAREN  RPAREN  LBRACKET  RBRACKET  LBRACE  RBRACE
%token      SEMI  COMMA COLON DOT
%token      TRUE  FALSE
%token      AND  OR  NOT EQ  NEQ  GT  LT GEQ LEQ
%token      PLUS  MINUS  DIVIDE  TIMES
%token      <string> ID
%token      <float> NUM
%token      <string> BOOL
%token      <string> STRING


%right      ASSIGN
%left AND  OR
%left EQ  NEQ NOT
%left LT GT GEQ LEQ
%left PLUS  MINUS
%left DIVIDE  TIMES

%start program
%type <Ast.program> program
```

```
%%
program:
      initialization rules operations    {$1, $2, $3}

/*INITIALIZE*/
initialization:
      INITIALIZATION COLON initial_declarator_list
                              {$3}

initial_declarator_list:
      /*NOTHING*/             { [ ] , [ ] }
      |     initial_declarator_list value_declarator
                              { ( (fst $1) @ [$2]), snd $1}
      |     initial_declarator_list triangle_declarator
                              { fst $1, ((snd $1) @ [$2])}

value_declarator:
      VALUE ID NUM SEMI       {{vname= $2; value= $3;}}
/* type 0 for value, 1 for triangle */
triangle_declarator:
      TRIANGLE ID line_status       vertex_status SEMI
                              { { tname = $2;  tline = $3; tvertex =
$4; } }

vertex_status:
      /*NOTHING*/             { { avtxx =0.0;
                                    avtxy =0.0;
                                    bvtxx =0.0;
                                    bvtxy =0.0;
                                  cvtxx =0.0;
                                    cvtxy =0.0; } }
      |     VERTEX  LBRACKET  LPAREN initial_value COMMA
initial_value RPAREN COMMA  LPAREN initial_value COMMA
            initial_value RPAREN COMMA LPAREN initial_value COMMA
initial_value RPAREN RBRACKET
                              { { avtxx =$4;
                                    avtxy =$6;
                                    bvtxx =$10;
                                    bvtxy =$12;
                                    cvtxx =$16;
                                    cvtxy =$18; } }

line_status:
      /*NOTHING*/             { { alen = 0.0;
                                    blen = 0.0;
                                    clen = 0.0; } }
      |      LINE LBRACKET initial_value COMMA initial_value COMMA
initial_value RBRACKET
                              { { alen = $3;
                                    blen = $5;
                                    clen = $7; } }

initial_value:
            NUM               { $1  }
```

```
      |     WILDCAT                 {   0.0      }



/*RULES*/

rules:
      RULES COLON rule_declarator_list {  List.rev $3     }

rule_declarator_list:
      /*NOTHING*/             { [ ]  }
      |rule_declarator_list rule_declarator
                              { $2::$1 }

rule_declarator:
      ID LPAREN para_opt RPAREN LPAREN expr RPAREN LBRACE
numerical_expr RBRACE SEMI
                              { { rname = $1;
                                  paras = $3;
                                  cond = $6;
                                  body = $9;}}

para_opt:
      /*NOTHING*/             { [ ] }
      | para_lst              { List.rev $1 }

para_lst:
            para_type ID                { [{ptp = $1; pname =
$2;}] }
      |     para_lst COMMA para_type ID  { {ptp = $3; pname =
$4;}::$1 }

para_type:
            TRIANGLE          { "triangle" }
      |     VALUE             { "value" }

expr:
      expr AND expr               {Binop($1, And, $3)     }
|     expr OR expr                {Binop($1, Or, $3)      }
|     NOT expr                    {Rev($2 )   }
|     expr EQ expr                {Binop($1, Equal, $3)   }
|     expr NEQ expr               {Binop($1, Neq, $3)     }
|     expr LT expr                {Binop($1, Less, $3)    }
|     expr GT expr                {Binop($1, Greater, $3)}
|     expr LEQ expr               {Binop($1, Leq, $3)     }
|     expr GEQ expr               {Binop($1, Geq, $3)     }
|     LBRACKET expr RBRACKET      {  $2 }
|     numerical_expr              {Nexpr($1)}

oexpr:
      expr                                                {Expr($1)}
|     RULE ID LPAREN para_opt RPAREN          {Rulec($2, $4)}

numerical_expr:
```

```
      NUM
      {Num($1)}
|     BOOL                                                {Bool($1)}
|     VALUE ID                                            {Val($2)}
|     TRIANGLE ID DOT element
      {Tri_ele($2, $4)}
|     numerical_expr MINUS numerical_expr
      {Binop2($1,Minus,$3)}
|     numerical_expr PLUS numerical_expr      {Binop2($1,Plus,$3)}
|     numerical_expr TIMES numerical_expr
      {Binop2($1,Times,$3)}
|     numerical_expr DIVIDE numerical_expr    {Binop2($1,Divide,$3)}
|     SIN LPAREN numerical_expr RPAREN        {Monop(Sin, $3)   }
|     COS LPAREN numerical_expr RPAREN        {Monop(Cos, $3)   }
|     TAN LPAREN numerical_expr RPAREN        {Monop(Tan, $3)   }
|     ARCSIN LPAREN numerical_expr RPAREN           {Monop(Arcs, $3)}
|     ARCCOS LPAREN numerical_expr RPAREN           {Monop(Arcc, $3)}
|     ARCTAN LPAREN numerical_expr RPAREN           {Monop(Arct, $3)}
|     SQRT LPAREN numerical_expr RPAREN       {Monop(Sqrt, $3)}
|   LPAREN numerical_expr RPAREN              {$2}


/*
arg_opt:
                  { [ ] }
      | arg_lst              { List.rev $1 }

arg_lst:
            arg
      {[$1] }
      |     arg_lst COMMA arg                        { $3::$1 }
arg:
        TRIANGLE ID               {{atp = "triangle"; vid = "na";
tid = $2}}
      | VALUE ID                  {{atp = "value";  vid = $2; tid =
"na"}}*/


element:
      AX                          { 0 }
|     AY                          { 1 }
|     BX                          { 2 }
|     BY                          { 3 }
|     CX                          { 4 }
|     CY                          { 5 }
|     SIDEA                 { 6 }
|     SIDEB                 { 7 }
|     SIDEC                 { 8 }


operations:
      OPERATIONS COLON operation_list         {List.rev $3}

operation_list:
```

```
       /*NOTHING*/                          { [ ] }

      |     operation_list operation        { $2 :: $1 }

operation:
      LBRACE operation_list RBRACE          { Block(List.rev $2) }
      | PRINTS LPAREN STRING RPAREN SEMI     { Prints($3) }
      | PRINTV LPAREN expr RPAREN SEMI       { Printv($3) }
      | IF LPAREN expr RPAREN LBRACE operation RBRACE
                                    { If($3, $6) }
      | WHILE LPAREN expr RPAREN LBRACE operation RBRACE
                                        { While($3, $6) }
      | ID ASSIGN oexpr SEMI          { Assign($1, $3) }
```

compiler.ml (mainly written by Xuechen Feng)

```
open Printf
open Ast
module StringMap = Map.Make(String)

let source = "example.dat"
let ir = "IR.txt"
let rmap = StringMap.empty
let ic = open_in source
let lexbuf = Lexing.from_channel ic
let ((values, triangles), rule_decl, operations) = Parser.program
Scanner.token lexbuf
let oc = open_out ir
let global_value = []



let rec enum stride n = function
        [] -> []
      | hd::tl -> (n, hd) :: enum stride (n+stride) tl


let string_map_vpairs map pairs =
      List.fold_left (fun m (i,n) -> fprintf oc "%f\n"
n.value ;StringMap.add n.vname i m) map pairs


let string_map_tpairs map pairs =
      List.fold_left (fun m (i,n) ->

            fprintf oc "%f\n" n.tvertex.avtxx ;
            fprintf oc "%f\n" n.tvertex.avtxy ;
            fprintf oc "%f\n" n.tvertex.bvtxx ;
            fprintf oc "%f\n" n.tvertex.bvtxy ;
            fprintf oc "%f\n" n.tvertex.cvtxx ;
```

```
            fprintf oc "%f\n" n.tvertex.cvtxy ;
            fprintf oc "%f\n" n.tline.alen ;
            fprintf oc "%f\n" n.tline.blen ;
            fprintf oc "%f\n" n.tline.clen ;
            StringMap.add n.tname i m) map pairs

let string_map_rpairs map pairs =
      List.fold_left (fun m (i,n) -> StringMap.add n i m) map pairs
let _ =
(*binding two maps for value and triangle*)
let vlist = enum 1 0 values in
let vmap = string_map_vpairs StringMap.empty vlist in
let tlist = enum 9 (List.length values) triangles in
let tmap = string_map_tpairs StringMap.empty tlist in
(*print rul for starting the rul definition*)
let _ = fprintf oc "rul\n" in



let translate rule offset  =
      let arg = rule.paras and condition = rule.cond and stmt =
rule.body in
      let rule_string_and_length =
            let rec load_argument counter = function
                  [] -> ([], counter)
                  | hd::tl -> if hd.ptp =  "value" then let rtn =
load_argument (counter+1) tl in ( ([ String.concat " " ["pop";
(string_of_int counter)]]@ (fst rtn)) ,(snd rtn))
                                                  else let rtnt =
load_argument (counter+9) tl in
                                                                  ((

      [String.concat " " ["pop"; string_of_int (counter)];

      String.concat " " ["pop"; string_of_int (counter+1)];

      String.concat " " ["pop"; string_of_int (counter+2)];

      String.concat " " ["pop"; string_of_int (counter+3)];

      String.concat " " ["pop"; string_of_int (counter+4)];

      String.concat " " ["pop"; string_of_int (counter+5)];

      String.concat " " ["pop"; string_of_int (counter+6)];

      String.concat " " ["pop"; string_of_int (counter+7)];

      String.concat " " ["pop"; string_of_int (counter+8)]]
                                                            @ (fst
rtnt)),(snd rtnt)) in load_argument 0 arg in
      let argument_string = fst rule_string_and_length in
      let arglength = snd rule_string_and_length in
      printf "the total argument fields for this rule is : %d \n"
```

```
arglength;
      (*get expr list*)
      let rec find_local_index aname counter = function
              [] -> counter
            | hd::tl -> if hd.pname = aname then counter else (if
hd.ptp = "value" then (find_local_index aname (counter+1) tl) else
(find_local_index aname (counter+9) tl)) in

      let rec nexpr = function
              Num f -> [String.concat " " ["ldf"; string_of_float f]]
            | Bool b -> if b = "true" then ["ldf 1.0"] else ["ldf
0.0"]
            | Val vid -> let index = (find_local_index vid 0 arg) in
      printf "the location for value %s is : %d \n" vid index;
                                    if index > arglength  then raise
(Failure ("undeclared variable" ^ vid)) else  [String.concat " "
["psh"; string_of_int index]]
            | Tri_ele (tid, field) -> let tindex = (find_local_index
tid 0 arg) in printf "%d st element of value %s is : %d \n" field
tid (tindex+field);
                                    if tindex > arglength then raise
(Failure ("undeclared variable" ^ string_of_int arglength)) else
[String.concat " " ["psh"; string_of_int (tindex+field)]]
            | Binop2 (e1, op, e2)  ->
                                (match op with
                                    | Minus -> nexpr e1 @ nexpr e2 @
["sub"]
                                    | Plus -> nexpr e1 @ nexpr e2 @
["add"]
                                    | Times -> nexpr e1 @ nexpr e2 @
["mul"]
                                    | Divide -> nexpr e1 @ nexpr e2 @
["div"])
            | Monop (op, npr) ->
                                (match op with
                                    | Sin -> nexpr npr @ ["sin"]
                                    | Cos -> nexpr npr @ ["cos"]
                                    | Tan -> nexpr npr @ ["tan"]
                                    | Arcs -> nexpr npr @ ["acs"]
                                    | Arcc -> nexpr npr @ ["acc"]
                                    | Arct -> nexpr npr @ ["act"]
                                    | Sqrt -> nexpr npr @ ["sqt"]) in
      let rec expr = function
            Binop (e1, op, e2)  ->
                  (match op with
                        | And -> expr e1 @ expr e2 @ ["and"]
                        | Or -> expr e1 @ expr e2 @ ["or"]
                        | Equal -> expr e1 @ expr e2 @ ["eql"]
                        | Neq -> expr e1 @ expr e2 @ ["neq"]
                        | Less -> expr e1 @ expr e2 @ ["les"]
                        | Greater -> expr e1 @ expr e2 @ ["gtr"]
                        | Leq -> expr e1 @ expr e2 @ ["leq"]
                        | Geq -> expr e1 @ expr e2 @ ["geq"])
                        | Rev ep -> expr ep @ ["not"]
```

```
                        | Nexpr nexp -> nexpr nexp                in


      let rule_string = argument_string @ List.rev argument_string
in
      let rule_string = rule_string @ (expr condition) in
      let rule_string = rule_string @ ["bne 2"] @ ["rtn"] in
      let rule_string = rule_string @ nexpr stmt in
      let rule_string = rule_string @ ["rtn"] in
      (fprintf oc "%s\n" (String.concat "\n" rule_string));
      printf "the total argument fields for this rule is : %d \n"
arglength;
      printf "the original offset for  this rule is : %d \n" offset;
      printf "the the length of rule_string is: %d \n" (List.length
rule_string);
      printf "the offset for next rule is  is  : %d \n" (offset +
(List.length rule_string));
      (arglength,((offset ) + (List.length rule_string)))  in



let rec entry_pair offset = function
        []->[]
      | hd::tl -> let rtn = (translate hd (offset)) in
                        let arglen = (fst rtn) and new_offset = (snd
rtn) in
            (*print_int entrypoint;print_newline(); print_int
arglen;print_newline(); print_int new_offset;print_newline();*)
            ((offset + arglen), hd.rname) :: entry_pair new_offset tl
in
let pairs = (entry_pair 0 rule_decl) in
let rmap = string_map_rpairs (StringMap.empty) pairs in
printf "the the entry point of angleC is : %d \n" (StringMap.find
"angle_C" rmap);



let rec translate_operations operationlst =
      let rec process_var = function
              [] -> []
            | hd :: tl ->  if hd.ptp = "value" then let  vindex =
(StringMap.find hd.pname vmap) in

      [String.concat " " ["lod"; string_of_int vindex]] @
(process_var tl)
                                                          else let
tindex = (StringMap.find hd.pname tmap) in

      [String.concat " " ["lod"; string_of_int (tindex)];

      String.concat " " ["lod"; string_of_int (tindex+1)];

      String.concat " " ["lod"; string_of_int (tindex+2)];
```

```
      String.concat " " ["lod"; string_of_int (tindex+3)];

      String.concat " " ["lod"; string_of_int (tindex+4)];

      String.concat " " ["lod"; string_of_int (tindex+5)];

      String.concat " " ["lod"; string_of_int (tindex+6)];

      String.concat " " ["lod"; string_of_int (tindex+7)];

      String.concat " " ["lod"; string_of_int (tindex+8)]
                                                              ] @
(process_var tl) in


      let rec procnexpr = function
            Num f -> print_float f; [String.concat " " ["lod";
string_of_float f]]
            | Bool b -> if b = "true" then ["ldf 1.0"] else ["ldf
0.0"]
            | Val vid -> [String.concat " " ["lod"; string_of_int
(StringMap.find vid vmap)]]

            | Tri_ele (ttid, ffield) ->[String.concat " " ["lod";
string_of_int ((StringMap.find ttid tmap) + ffield)]]

            | Binop2 (e1, op, e2)  ->
                  (match op with
                        | Minus -> procnexpr e1 @ procnexpr e2 @
["sub"]
                        | Plus -> procnexpr e1 @ procnexpr e2 @
["add"]
                        | Times -> procnexpr e1 @ procnexpr e2 @
["mul"]
                        | Divide -> procnexpr e1 @ procnexpr e2 @
["div"])
            | Monop (op, npr) ->
                  (match op with
                        | Sin -> procnexpr npr @ ["sin"]
                        | Cos -> procnexpr npr @ ["cos"]
                        | Tan -> procnexpr npr @ ["tan"]
                        | Arcs -> procnexpr npr @ ["acs"]
                        | Arcc -> procnexpr npr @ ["acc"]
                        | Arct -> procnexpr npr @ ["act"]
                        | Sqrt -> procnexpr npr @ ["sqt"])
            in

      let rec expr = function
            Binop (e1, op, e2)  ->
                  (match op with
                        | And -> expr e1 @ expr e2 @ ["and"]
                        | Or -> expr e1 @ expr e2 @ ["or"]
                        | Equal -> expr e1 @ expr e2 @ ["eql"]
                        | Neq -> expr e1 @ expr e2 @ ["neq"]
```

```
                        | Less -> expr e1 @ expr e2 @ ["les"]
                        | Greater -> expr e1 @ expr e2 @ ["gtr"]
                        | Leq -> expr e1 @ expr e2 @ ["leq"]
                        | Geq -> expr e1 @ expr e2 @ ["geq"])
                        | Rev ep -> expr ep @ ["not"]
                        | Nexpr np -> procnexpr np        in

      let oexpr = function
              Expr exp -> expr exp
            | Rulec (rid, parlst) -> printf "%s" rid; let rindex =
(StringMap.find rid rmap) in printf "a rull is called with id = %s
and index = %d\n" rid rindex; (process_var parlst) @ [String.concat
" " ["rc";string_of_int rindex]] in

      let rec operation = function
              Block oplst -> translate_operations oplst
            | Prints str -> [String.concat " " ["pts"; str]]
            | Printv temp -> (expr temp) @ ["ptv"]
            | If (oxper, aoperation) -> let aoperation' = operation
aoperation in
                                (expr oxper) @ [String.concat " "
["beq" ;string_of_int (1 + List.length aoperation')]] @ aoperation'
            | Assign (id, oxper) -> printf "a assignment is called
with id = %s \n" id; printf "a assignment is called with address of
id = %d \n" (StringMap.find id vmap);
                        (oexpr oxper) @ (try [String.concat " "
["str"; string_of_int (StringMap.find id vmap)]] with Not_found ->
raise (Failure ("Only value assignment allowed: undeclared variable"
^ id)))
            | While (oxper, aoperation) -> let aoperation' =
operation aoperation and oxper' = expr oxper in
                                              [String.concat " " ["bra" ;
string_of_int (1 + List.length aoperation')]] @ aoperation' @
oxper'@
                                              [String.concat " " ["bra" ;
string_of_int (0 - (List.length aoperation' + List.length oxper'))]]
in
      let rec enumopt = function
              [] -> []
            | hd::tl -> (operation hd) @ (enumopt tl) in
      enumopt operationlst in

let operation_string = ["opt"] @ ( translate_operations operations)
@ ["hlt"]in
(fprintf oc "%s\n" (String.concat "\n" operation_string)) ;


close_out oc;
print_newline(); flush stdout;
```

interpreter.java (mainly written by Lianhao Qu)

```
import java.util.ArrayList;
import java.util.Stack;
import java.io.*;

public class interpreter {
      static String error1 = "Segmentation Fault";
      static String error2 = " (core dumped)";

      public static void main(String[] args) {
            // Create a new, empty stack, used as register
        Stack register = new Stack();
            // Create a new, empty arraylist, used for storing global
variables
        ArrayList<String> globalVar = new ArrayList<String>();
        // Create a new, empty arraylist, used for storing local
variables
        //ArrayList<String> localVar = new ArrayList<String>();
        // Create a new, empty arraylist, used for storing converted
byte code for rule
        ArrayList<String[]> rulecode = new ArrayList();
        // Create a new, empty arraylist, used for storing converted
byte code for operation
        ArrayList<String[]> opcode = new ArrayList();

        //-------------------------------------------------------------
-------------
        //read the bytecode file, store corresponding data
        boolean rule = false;//flag to check whether to start
reading rule from bytecode
        boolean operation = false;//flag to check whether to start
reading operation from bytecode file

        try{
             // Open the bytecode file
```

```
            FileInputStream fstream = new
FileInputStream("src/IR.txt");
            // Get the object of DataInputStream
            DataInputStream in = new DataInputStream(fstream);
            BufferedReader br = new BufferedReader(new
InputStreamReader(in));
            String strLine;
            //Read File Line By Line
            while ((strLine = br.readLine()) != null)   {
                  if(strLine.equals("rul"))
                        rule = true;
                  if(strLine.equals("opt"))
                        operation = true;
                  if(rule == false && operation == false
&&  !strLine.isEmpty()){//assign the global variable into
arraylist:globalVar
                        globalVar.add(strLine);
                  }
                  else if(rule == true && operation == false
&& !strLine.isEmpty() && !strLine.equals("rul")){
                        String delims = "\\s+";
                        String[] tokens = strLine.split(delims);
                        rulecode.add(tokens);
                  }
                  else        if(operation        ==        true
&&  !strLine.equals("opt")  &&  !strLine.isEmpty()){//assign the
operation into arraylist:code
                        String delims = "\\s+";
                        String[] tokens = strLine.split(delims);
                        opcode.add(tokens);
                  }
            }
            //Close the input stream
            in.close();
        }catch (Exception e){//Catch exception if any
            System.err.println("Error: " + e.getMessage());
```

```
        }



        //test, print out what has stored in the rulecode and opcode
        /*System.out.println(rulecode.size());
        for(int i=0;i<rulecode.size();i++){
            for(int j=0;j<rulecode.get(i).length;j++){
                  System.out.println(rulecode.get(i)[j]);
            }

        }

        for(int i=0;i<opcode.size();i++){
            for(int j=0;j<opcode.get(i).length;j++){
                  System.out.println(opcode.get(i)[j]);
            }

        }*/

        //-------------------------------------------------------------
-------------
        //operation start
        int pc = 0; //init programming counter
        int rc = 0; //init rule counter(function counter)
        double temp = 0;//used for temporary store a num for math
operation, popped from register
        String x ="";//used for temporary store a string for math
operation, first popped from register
        String y ="";//used for temporary store a string for math
operation, second popped from register
        while(!opcode.get(pc)[0].equalsIgnoreCase("hlt")){
            switch(opcode.get(pc)[0]){
                  case "rc":
                        //if(!isInt(opcode.get(pc)[1])              ||
Integer.parseInt(opcode.get(pc)[1])>=globalVar.size()){
```

```
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        rc = Integer.parseInt(opcode.get(pc)[1]);
                        /*a huge while*/

      while(!rulecode.get(rc)[0].equalsIgnoreCase("rtn")){
                        switch(rulecode.get(rc)[0]){
                              case "lod":
                                    //if(!isInt(rulecode.get(rc)[1])
|| Integer.parseInt(rulecode.get(rc)[1])>=globalVar.size()){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

      register.push(  globalVar.get((int)Double.parseDouble(rulecode.
get(rc)[1])) );
                                    break;
                              case "str":
                                    //if(!isInt(rulecode.get(rc)[1])
|| Integer.parseInt(rulecode.get(rc)[1])>=globalVar.size()){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

      globalVar.set(Integer.parseInt(rulecode.get(rc)[1]),(String)
register.pop());
                                    break;
                              case "psh":
                                    //if(!isInt(rulecode.get(rc)[1])
|| Integer.parseInt(rulecode.get(rc)[1])>=rulecode.size()){
                                    //
      System.out.println(error1+error2);
                                    //    return;
```

```
                                    //}

      //System.out.println(rulecode.get(Integer.parseInt(rulecode.ge
t(rc)[1]))[0]);

      //System.out.println(Integer.parseInt(rulecode.get(rc)[1]));

      register.push(  rulecode.get(Integer.parseInt(rulecode.get(rc)[
1]))[0]);//original localval.get

      //System.out.println(register.peek());
                                    break;
                               case "pop"://push, into local variable
                                    //if(!isInt(rulecode.get(rc)[1])
|| Integer.parseInt(rulecode.get(rc)[1])>=rulecode.size()){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    String[]    tokens    =    {(String)
register.pop()};

      //System.out.println(rulecode.get(rc)[1]);

      rulecode.set(Integer.parseInt(rulecode.get(rc)[1]),tokens);//o
riginal localval.set

      //System.out.println(rulecode.get(Integer.parseInt(rulecode.ge
t(rc)[1]))[0]);
                                    break;
                               case "ldf"://load float point number/or
anything into register immediately

      register.push(rulecode.get(rc)[1]);

      //System.out.println(register.peek());
```

```
                                    break;
                              case "bra"://jump
                                    //if(!isInt(rulecode.get(rc)[1])){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    rc                                    +=
Integer.parseInt(rulecode.get(rc)[1])    -    1;//jump    to    relevant
position
                                    break;
                              case "bne":
                                    //if(!isInt(rulecode.get(rc)[1])){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    if(Double.parseDouble(x)!=0.0)
                                          rc                              +=
Integer.parseInt(rulecode.get(rc)[1])    -    1;//jump    to    relevant
position if not equal to zero
                                    //System.out.println(rc);
                                    break;
                              case "beq":
                                    //if(!isInt(rulecode.get(rc)[1])){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    x = (String) register.pop();
```

```
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    if(Double.parseDouble(x)==0.0)
                                          rc                          +=
Integer.parseInt(rulecode.get(rc)[1])   -   1;//jump   to   relevant
position if equal to zero
                                    break;
                              case "and":
                                    x = (String) register.pop();
                                    y = (String) register.pop();
                                    if(Double.parseDouble(x)==0.0
||Double.parseDouble(y)==0.0)
                                          register.push("0.0");
                                    else
                                          register.push("1.0");
                                    break;
                              case "or":
                                    x = (String) register.pop();
                                    y = (String) register.pop();
                                    if(Double.parseDouble(x)!=0.0
||Double.parseDouble(y)!=0.0)
                                          register.push("1.0");
                                    else
                                          register.push("0.0");
                                    break;
                              case "eql":
                                    x = (String) register.pop();
                                    y = (String) register.pop();

      if(Double.parseDouble(x)==Double.parseDouble(y))

                                          register.push("1.0");
                                    else
```

```
                                    register.push("0.0");
                              break;
                        case "neq":
                              x = (String) register.pop();
                              y = (String) register.pop();

    if(Double.parseDouble(x)!=Double.parseDouble(y))

                                    register.push("1.0");
                              else
                                    register.push("0.0");
                              break;
                        case "les":
                              x = (String) register.pop();
                              y = (String) register.pop();

    if(Double.parseDouble(y)<Double.parseDouble(x))

                                    register.push("1.0");
                              else
                                    register.push("0.0");
                              break;
                        case "gtr":
                              x = (String) register.pop();
                              y = (String) register.pop();

    if(Double.parseDouble(x)>Double.parseDouble(y))

                                    register.push("1.0");
                              else
                                    register.push("0.0");
                              break;
                        case "leq":
                        x = (String) register.pop();
                        y = (String) register.pop();
```

```
if(Double.parseDouble(x)<=Double.parseDouble(y))

                              register.push("1.0");
                        else
                              register.push("0.0");
                        break;
                        case "geq":
                        x = (String) register.pop();
                        y = (String) register.pop();

if(Double.parseDouble(x)>=Double.parseDouble(y))

                              register.push("1.0");
                        else
                              register.push("0.0");
                        break;
                        case "not":
                              x = (String) register.pop();
                              if(!isNumeric(x)){

System.out.println(error1+error2);
                                    return;
                              }
                              if(Double.parseDouble(x)==0.0)
                                    x = "1.0";
                              else
                                    x = "0.0";
                              register.push(x);
                              break;
                        case "add":
                              x = (String) register.pop();
                              if(!isNumeric(x)){

System.out.println(error1+error2);
                                    return;
                              }
```

```
                                    y = (String) register.pop();
                                    //if(!isNumeric(y)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp   =   Double.parseDouble(x)   +
Double.parseDouble(y);

      register.push(Double.toString(temp));
                                    break;
                              case "sub"://stack1 - stack2
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    y = (String) register.pop();
                                    //if(!isNumeric(y)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp   =   Double.parseDouble(y)   -
Double.parseDouble(x);

      register.push(Double.toString(temp));
                                    break;
                              case "mul":
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
```

```
                                    y = (String) register.pop();
                                    //if(!isNumeric(y)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp   =   Double.parseDouble(x)   *
Double.parseDouble(y);

      register.push(Double.toString(temp));
                                    break;
                              case "div"://stack1 / stack2
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    y = (String) register.pop();
                                    //if(!isNumeric(y)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp   =   Double.parseDouble(y)   /
Double.parseDouble(x);

      register.push(Double.toString(temp));
                                    break;
                              case "addi":

      //if(!isNumeric(rulecode.get(rc)[1])){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
```

```
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

                                    temp                                =
Double.parseDouble(rulecode.get(rc)[1]);
                                    temp += Double.parseDouble(x);

      register.push(Double.toString(temp));
                                    break;
                               case "subi":

      //if(!isNumeric(rulecode.get(rc)[1])){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

                                    temp                                =
Double.parseDouble(rulecode.get(rc)[1]);
                                    temp   =   Double.parseDouble(x)   -
temp;

      register.push(Double.toString(temp));
                                    break;
                               case "muli":
```

```
      //if(!isNumeric(rulecode.get(rc)[1])){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

                                    temp                                =
Double.parseDouble(rulecode.get(rc)[1]);
                                    temp *= Double.parseDouble(x);

      register.push(Double.toString(temp));
                                    break;
                               case "divi":

      //if(!isNumeric(rulecode.get(rc)[1])){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

                                    temp                                =
Double.parseDouble(rulecode.get(rc)[1]);
                                    temp   =   Double.parseDouble(x)   /
temp;
```

```
      register.push(Double.toString(temp));
                                    break;
                              case "sqt"://sqrt, range [0,infinite]
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp = Double.parseDouble(x);
                                    //if(temp<0){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp = Math.sqrt(temp);

      register.push(Double.toString(temp));
                                    break;
                              case "sin":
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

                                    temp                                    =
Math.sin(Double.parseDouble(x));

      register.push(Double.toString(temp));
                                    break;
                              case "cos":
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
```

```
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

                                    temp                                    =
Math.cos(Double.parseDouble(x));

      register.push(Double.toString(temp));
                                    break;
                              case "tan":
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

                                    temp                                    =
Math.tan(Double.parseDouble(x));

      register.push(Double.toString(temp));
                                    break;
                              case "acs"://arcsin, range [-1,1]
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp = Double.parseDouble(x);
                                    //if(temp<-1 || temp>1){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
```

```
                                    temp = Math.asin(temp);

      register.push(Double.toString(temp));
                                    break;
                              case "acc"://arccos, range [-1,1]
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp = Double.parseDouble(x);
                                    //if(temp<-1 || temp>1){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}
                                    temp = Math.acos(temp);

      register.push(Double.toString(temp));
                                    break;
                              case "act":
                                    x = (String) register.pop();
                                    //if(!isNumeric(x)){
                                    //
      System.out.println(error1+error2);
                                    //    return;
                                    //}

                                    temp                                    =
Math.atan(Double.parseDouble(x));

      register.push(Double.toString(temp));
                                    break;
                              case "ptv":
```

```
      System.out.println(register.pop());
                                    break;
                               case "pts":
                                    for(int
i=1;i<rulecode.get(rc).length;i++){

      System.out.print(rulecode.get(rc)[i]);
                                          System.out.print(" ");
                                    }
                                    System.out.println();
                                    break;
                               default:

      System.out.println(rulecode.get(rc)[0]);
                         }
                         rc++;
                    }
                         /*finish a huge while*/
                         break;
                  case "lod":
                         //if(!isInt(opcode.get(pc)[1])              ||
Integer.parseInt(opcode.get(pc)[1])>=globalVar.size()){
                         //    System.out.println(error1+error2);
                         //    return;
                         //}


      register.push(  globalVar.get((int)Double.parseDouble(opcode.ge
t(pc)[1])) );
                         break;
                  case "str":
                         //if(!isInt(opcode.get(pc)[1])              ||
Integer.parseInt(opcode.get(pc)[1])>=globalVar.size()){
                         //    System.out.println(error1+error2);
                         //    return;
                         //}
```

```
      globalVar.set(Integer.parseInt(opcode.get(pc)[1]),(String)
register.pop());
                        break;
                  case "psh":
                        //if(!isInt(opcode.get(pc)[1])              ||
Integer.parseInt(opcode.get(pc)[1])>=rulecode.size()){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}

      register.push(  rulecode.get(Integer.parseInt(opcode.get(pc)[1]
))[0] );
                        break;
                  case "pop"://push, into local variable
                        //if(!isInt(opcode.get(pc)[1])              ||
Integer.parseInt(opcode.get(pc)[1])>=rulecode.size()){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        String[] tokens = {(String) register.pop()};

      rulecode.set(Integer.parseInt(opcode.get(pc)[1]),tokens);
                        break;
                  case  "ldf"://load  float  point  number/or  anything
into register immediately
                        register.push(opcode.get(pc)[1]);
                        break;
                  case "bra"://jump
                        //if(!isInt(opcode.get(pc)[1])){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        pc  +=  Integer.parseInt(opcode.get(pc)[1])  -
1;//jump to relevant position
                        break;
```

```
                case "bne":
                      //if(!isInt(opcode.get(pc)[1])){
                      //    System.out.println(error1+error2);
                      //    return;
                      //}
                      x = (String) register.pop();
                      //if(!isNumeric(x)){
                      //    System.out.println(error1+error2);
                      //    return;
                      //}
                      if(Double.parseDouble(x)!=0.0)
                            pc                                          +=
Integer.parseInt(opcode.get(pc)[1]) - 1;//jump to relevant position
if not equal to zero
                      break;
                case "beq":
                      //if(!isInt(opcode.get(pc)[1])){
                      //    System.out.println(error1+error2);
                      //    return;
                      //}
                      x = (String) register.pop();
                      //if(!isNumeric(x)){
                      //    System.out.println(error1+error2);
                      //    return;
                      //}
                      if(Double.parseDouble(x)==0.0)
                            pc                                          +=
Integer.parseInt(opcode.get(pc)[1]) - 1;//jump to relevant position
if equal to zero
                      break;
                case "and":
                      x = (String) register.pop();
                      y = (String) register.pop();
                      if(Double.parseDouble(x)==0                       ||
Double.parseDouble(y)==0)
                            register.push("0.0");
```

```
                        else
                              register.push("1.0");
                        break;
                  case "or":
                        x = (String) register.pop();
                        y = (String) register.pop();
                        if(Double.parseDouble(x)!=0                    ||
Double.parseDouble(y)!=0)
                              register.push("1.0");
                        else
                              register.push("0.0");
                        break;
                  case "eql":
                        x = (String) register.pop();
                        y = (String) register.pop();

     if(Double.parseDouble(x)==Double.parseDouble(y))

                              register.push("1.0");
                        else
                              register.push("0.0");
                        break;
                  case "neq":
                        x = (String) register.pop();
                        y = (String) register.pop();

     if(Double.parseDouble(x)!=Double.parseDouble(y))

                              register.push("1.0");
                        else
                              register.push("0.0");
                        break;
                  case "les":
                        x = (String) register.pop();
                        y = (String) register.pop();
```

```
	if(Double.parseDouble(x)<Double.parseDouble(y))

				register.push("1.0");
			else
				register.push("0.0");
			break;
		case "gtr":
			x = (String) register.pop();
			y = (String) register.pop();

	if(Double.parseDouble(x)>Double.parseDouble(y))

				register.push("1.0");
			else
				register.push("0.0");
			break;
		case "leq":
		x = (String) register.pop();
		y = (String) register.pop();
		if(Double.parseDouble(x)<=Double.parseDouble(y))

			register.push("1.0");
		else
			register.push("0.0");
		break;
		case "geq":
		x = (String) register.pop();
		y = (String) register.pop();
		if(Double.parseDouble(x)>=Double.parseDouble(y))

			register.push("1.0");
		else
			register.push("0.0");
		break;
		case "not":
			x = (String) register.pop();
```

```
                        //if(!isNumeric(x)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        if(Double.parseDouble(x)==0.0)
                              x = "1.0";
                        else
                              x = "0.0";
                        register.push(x);
                        break;
                  case "add":
                        x = (String) register.pop();
                        //if(!isNumeric(x)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        y = (String) register.pop();
                        //if(!isNumeric(y)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        temp      =      Double.parseDouble(x)      +
Double.parseDouble(y);
                        register.push(Double.toString(temp));
                        break;
                  case "sub"://stack1 - stack2
                        x = (String) register.pop();
                        //if(!isNumeric(x)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        y = (String) register.pop();
                        //if(!isNumeric(y)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
```

```
                        temp      =      Double.parseDouble(y)      -
Double.parseDouble(x);
                        register.push(Double.toString(temp));
                        break;
                  case "mul":
                        x = (String) register.pop();
                        //if(!isNumeric(x)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        y = (String) register.pop();
                        //if(!isNumeric(y)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        temp      =      Double.parseDouble(x)      *
Double.parseDouble(y);
                        register.push(Double.toString(temp));
                        break;
                  case "div"://stack1 / stack2
                        x = (String) register.pop();
                        //if(!isNumeric(x)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        y = (String) register.pop();
                        //if(!isNumeric(y)){
                        //    System.out.println(error1+error2);
                        //    return;
                        //}
                        temp      =      Double.parseDouble(y)      /
Double.parseDouble(x);
                        register.push(Double.toString(temp));
                        break;
                  case "addi":
                        //if(!isNumeric(opcode.get(pc)[1])){
```

```
                    //    System.out.println(error1+error2);
                    //    return;
                    //}
                    x = (String) register.pop();
                    //if(!isNumeric(x)){
                    //    System.out.println(error1+error2);
                    //    return;
                    //}

                    temp = Double.parseDouble(opcode.get(pc)[1]);
                    temp += Double.parseDouble(x);
                    register.push(Double.toString(temp));
                    break;
                case "subi":
                    //if(!isNumeric(opcode.get(pc)[1])){
                    //    System.out.println(error1+error2);
                    //    return;
                    //}
                    x = (String) register.pop();
                    //if(!isNumeric(x)){
                    //    System.out.println(error1+error2);
                    //    return;
                    //}

                    temp = Double.parseDouble(opcode.get(pc)[1]);
                    temp = Double.parseDouble(x) - temp;
                    register.push(Double.toString(temp));
                    break;
                case "muli":
                    //if(!isNumeric(opcode.get(pc)[1])){
                    //    System.out.println(error1+error2);
                    //    return;
                    //}
                    x = (String) register.pop();
                    //if(!isNumeric(x)){
                    //    System.out.println(error1+error2);
```

```
                //    return;
                //}

                temp = Double.parseDouble(opcode.get(pc)[1]);
                temp *= Double.parseDouble(x);
                register.push(Double.toString(temp));
                break;
            case "divi":
                //if(!isNumeric(opcode.get(pc)[1])){
                //    System.out.println(error1+error2);
                //    return;
                //}
                x = (String) register.pop();
                //if(!isNumeric(x)){
                //    System.out.println(error1+error2);
                //    return;
                //}

                temp = Double.parseDouble(opcode.get(pc)[1]);
                temp = Double.parseDouble(x) / temp;
                register.push(Double.toString(temp));
                break;
            case "sqt"://sqrt, range [0,infinite]
                x = (String) register.pop();
                //if(!isNumeric(x)){
                //    System.out.println(error1+error2);
                //    return;
                //}
                temp = Double.parseDouble(x);
                //if(temp<0){
                //    System.out.println(error1+error2);
                //    return;
                //}
                temp = Math.sqrt(temp);
                register.push(Double.toString(temp));
                break;
```

```
                case "sin":
                      x = (String) register.pop();
                      //if(!isNumeric(x)){
                      //    System.out.println(error1+error2);
                      //    return;
                      //}

                      temp = Math.sin(Double.parseDouble(x));
                      register.push(Double.toString(temp));
                      break;
                case "cos":
                      x = (String) register.pop();
                      //if(!isNumeric(x)){
                      //    System.out.println(error1+error2);
                      //    return;
                      //}

                      temp = Math.cos(Double.parseDouble(x));
                      register.push(Double.toString(temp));
                      break;
                case "tan":
                      x = (String) register.pop();
                      //if(!isNumeric(x)){
                      //    System.out.println(error1+error2);
                      //    return;
                      //}

                      temp = Math.tan(Double.parseDouble(x));
                      register.push(Double.toString(temp));
                      break;
                case "acs"://arcsin, range [-1,1]
                      x = (String) register.pop();
                      //if(!isNumeric(x)){
                      //    System.out.println(error1+error2);
                      //    return;
                      //}
```

```
                temp = Double.parseDouble(x);
                //if(temp<-1 || temp>1){
                //    System.out.println(error1+error2);
                //    return;
                //}
                temp = Math.asin(temp);
                register.push(Double.toString(temp));
                break;
            case "acc"://arccos, range [-1,1]
                x = (String) register.pop();
                //if(!isNumeric(x)){
                //    System.out.println(error1+error2);
                //    return;
                //}
                temp = Double.parseDouble(x);
                //if(temp<-1 || temp>1){
                //    System.out.println(error1+error2);
                //    return;
                //}
                temp = Math.acos(temp);
                register.push(Double.toString(temp));
                break;
            case "act":
                x = (String) register.pop();
                //if(!isNumeric(x)){
                //    System.out.println(error1+error2);
                //    return;
                //}

                temp = Math.atan(Double.parseDouble(x));
                register.push(Double.toString(temp));
                break;
            case "ptv":
                System.out.println(register.pop());
                break;
            case "pts":
```

```
                        for(int i=1;i<opcode.get(pc).length;i++){
                              System.out.print(opcode.get(pc)[i]);
                              System.out.print(" ");
                        }
                        System.out.println();
                        break;
                  default:
                        System.out.println(opcode.get(pc)[0]);
            }
            pc++;
         }
      }

      //helper methods
      //check whether a String is a numeric type
      public static boolean isNumeric(String str){
            return str.matches("-?\\d+(\\.\\d+)?");  //match a number
with optional '-' and decimal.
      }
      public static boolean isInt(String str){
            try{
                  int d = Integer.parseInt(str);
            }
            catch(NumberFormatException nfe){
                return false;
            }
            return true;
      }
}
```