EZ LANGUAGE

SER 502 – PROGRAMMING LANGUAGES AND PARADIGMS

SPRING 2018

TEAM 29

https://github.com/pratik8064/SER-502-Spring-2018-Team29

https://youtu.be/x1RiYHtVbXc

CAPTAIN AMERICA

RAAM PRASHANTH NS

IRON MAN

HARI SIDDARTH VK

THE HULK

MOHAN V YADAV

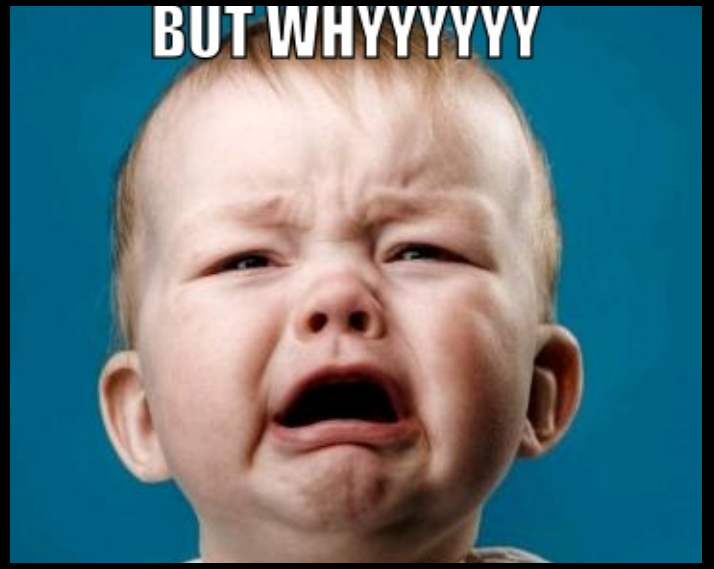
THOR

PRATIK SURYAWANSHI

THE AVENGERS

WHY “EZ”?

- “Why does this language has to be so complex?”



WHY "EZ"?

- "Why does this language has to be so complex?"
- We wanted to develop a simple and easily programmable language.
- Solution – EZ Language.

HIGHLIGHTS

- The EZ language supports the use of functions with multiple parameters and recursive calls.
- The scope of variables in different blocks is handled.
- We have handled basic exceptions.
- We have supported nested if's and nested loops.
- We have also tried to decrease the runtime of the code to an extent.
- We allow users to input values using read command and also write to the console.

GRAMMAR

```
grammar EZ;
program : statement_list;
statement_list : statement statement_list | ;
statement : decl_statement | assign_statement | read_statement | write_statement | if_statement | loop_statement | function_call_statement ';' | function_statement;
decl_statement : 'variable' identifier ';' ;
assign_statement : identifier '=' expression ';';
read_statement : 'read' identifier ';';
write_statement : 'write' expression ';' | 'write' '"'(.)*?'";';
if_statement : 'if' '(' cond_expression ')' 'then' '{' statement_list RPARA else_statement?;
else_statement : 'else' '{' statement_list RPARA;
loop_statement : 'repeat_when' '(' cond_expression ')' '{' statement_list RPARA;
function_statement : 'function' identifier '('(identifier | (identifier (',' identifier)*))?')' '{' statement_list (return_statement)? RPARA;
return_statement :'return' (expression)';';
function_call_statement : identifier '('(expression | (expression (',' expression)*))?')' ;
expression : exp1 '+' expression  #addition | exp1 '-' expression #subtraction | exp1 #expPrecedence;
exp1 : factor '*' exp1 #multiplication | factor '/' exp1 #division | factor '%' exp1 #mod | factor #fact ;
factor : identifier | number | function_call_statement ;
cond_expression : expression cond_operators expression | bool_val;
cond_operators :  '==' | '<' | '>' | '<=' | '>=' | '!=';
identifier :  lowerChar (letters | underscore | digit)* ;
letters : (lowerChar | upperChar)+;
underscore : '_';
bool_val : 'true' | 'false';
number : '-'? digit+;
digit : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
lowerChar : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' ;
upperChar : 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' ;
RPARA : '}';
Comment_statement : '/#' (.)*? '#/' -> skip;
WS: [ \t\r\n]+ -> skip ;
```

GRAMMAR

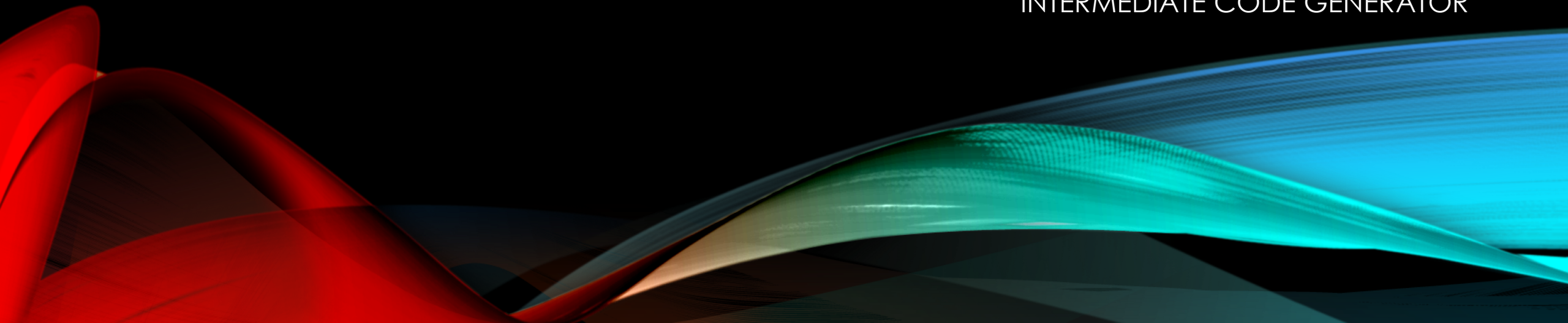
- The EZ Grammar supports the following:
  - Declaration statements – variable <variable_name>
  - Assignment statements - <variable_name> = <value> or <variable_name>
  - Read Statements – read <variable_name>
  - Write Statements – write <variable_name>
  - Conditional blocks - if (condition) then {…} else {…}
  - Iterative blocks – repeat_when (condition) {…}
  - Function blocks – function <function_name> (<parameter list>) {…}

COMPILER

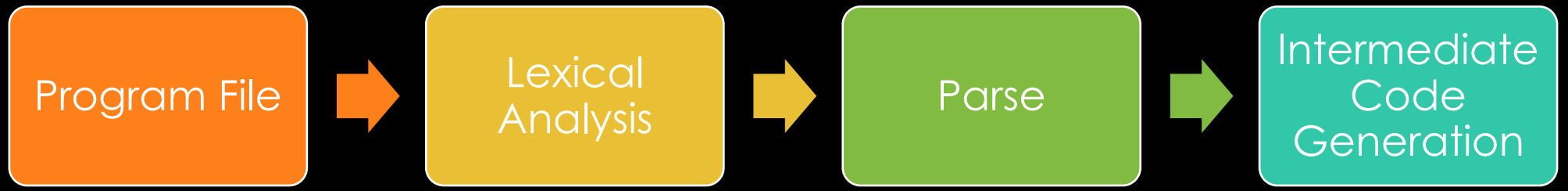
LEXICAL ANALYZER

PARSER

INTERMEDIATE CODE GENERATOR

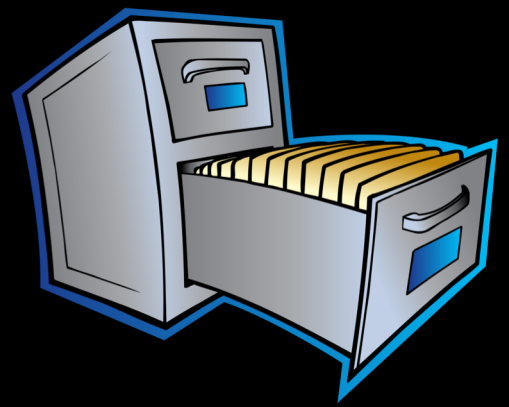


COMPILER FLOW



LEXICAL ANALYZER

- Input – Program file
- Output – Tokens
- The lexical analyzer breaks down the program into tokens which is sent as input to the parser.



PARSER

- Input – Tokens
- Output – Parse Tree
- The parser uses the tokens to generate the parse tree based on the grammar for further execution.

SAMPLE PARSE TREE

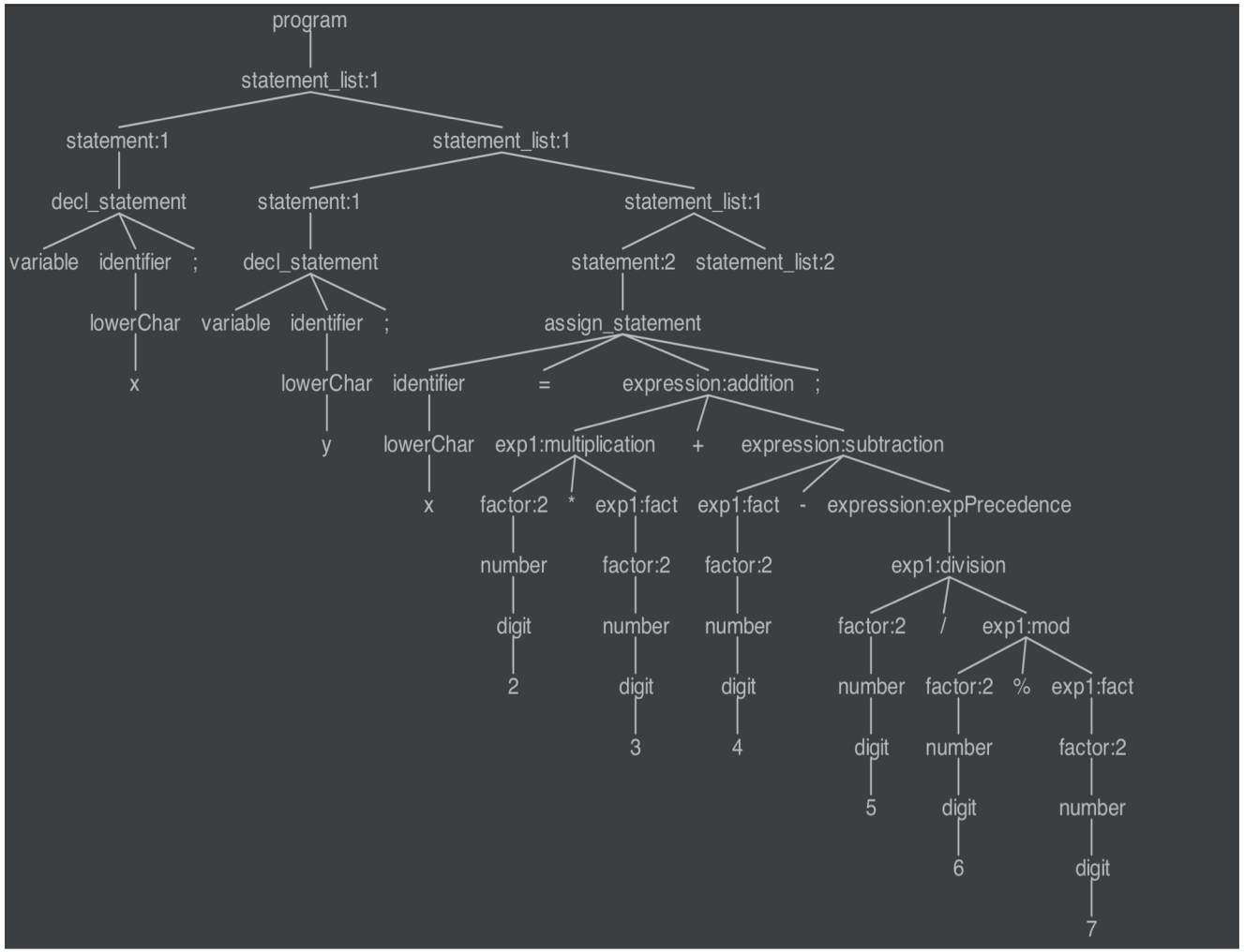
- Code:

  variable x;

  variable y;

  x = 2 * 3 + 4 - 5 / 6 % 7;

- Note: Precedence is handled.



INTERMEDIATE CODE GENERATOR

- Input – Parse Tree
- Output – Byte code
- The byte code of the program file is generated in this phase using the parse tree from the parser.

SAMPLE INTERMEDIATE CODE

- Code:

  variable x;
  variable y;
  x = 2 * 3 + 4 - 5 / 6 % 7;

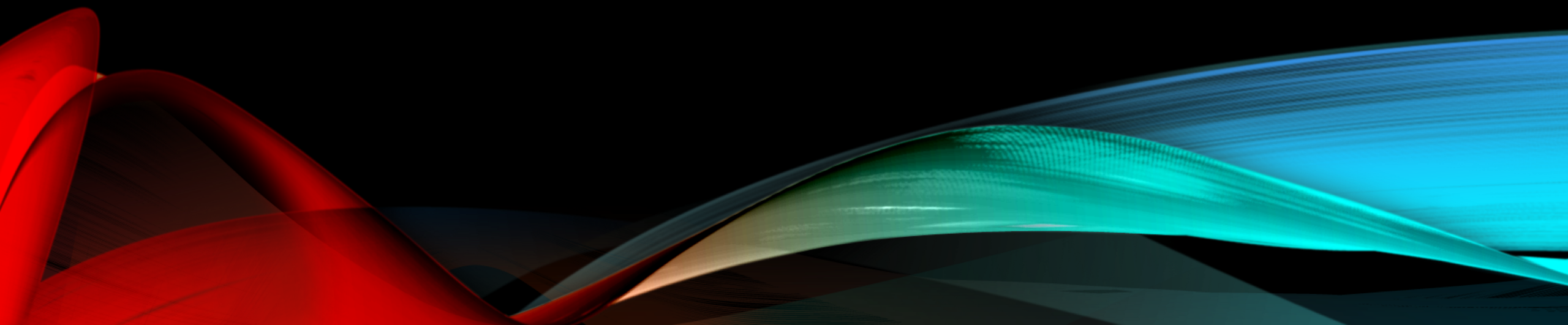
```
DECL x
DECL y
PUSH 2
PUSH 3
MUL
PUSH 4
PUSH 5
PUSH 6
PUSH 7
REM
DIV
SUB
ADD
MOV x
```

COMPILER ARCHITECTURE

```
EZLexer lexer = new EZLexer(charStream);
CommonTokenStream tokenStream = new CommonTokenStream(lexer);
parser = new EZParser(tokenStream);
ParseTreeWalker.DEFAULT.walk(EZIntermediateCodeGenarator.getInstance(), parser.program());
ArrayList<String> intermediateCode =  EZIntermediateCodeGenarator.getInstance().getintermediateCode();
writeIntermediateFile(filename, intermediateCode);
```

1. charStream contains the input program file.
2. It is then broken down into tokens and stored in tokenStream using the lexer object.
3. Then the parse tree is generated from the tokenStream.
4. Finally, the corresponding intermediate code is written to a file that is used in runtime.

CODE STRUCTURE



INTERMEDIATE CODE

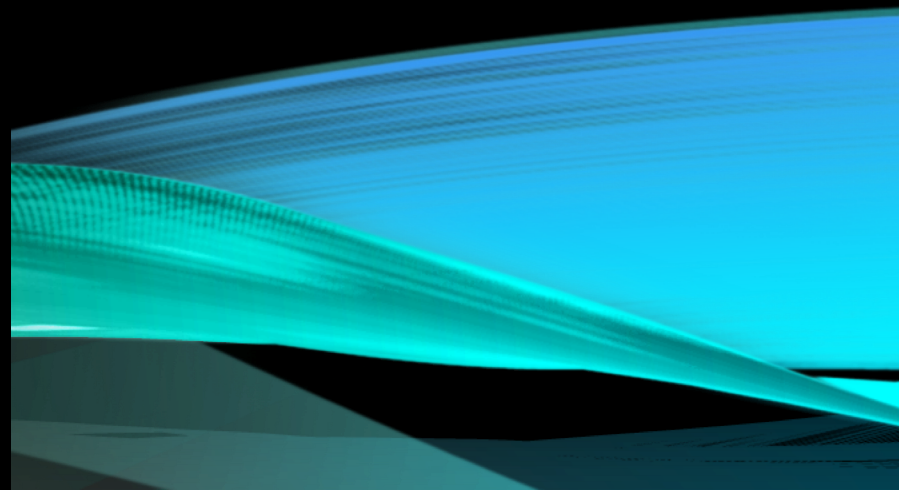
```
private static EZIntermediateCodeGenarator INSTANCE = null;
private static ArrayList<String> intermediateCode;
private static int nestCount = 1;
private static Stack<Integer> nestedStack = new Stack<Integer>();
private static Stack<String> function = new Stack<String>();
```

```
@Override public void enterIf_statement(EZParser.If_statementContext ctx) {
    intermediateCode.add(EZConstants.IF.trim() +"_" + nestCount);
    nestedStack.push(nestCount);
    nestCount++;
}

@Override public void exitIf_statement(EZParser.If_statementContext ctx) {
    intermediateCode.add(EZConstants.END_IF.trim() + "_" + nestedStack.pop());
}

@Override public void enterElse_statement(EZParser.Else_statementContext ctx) {
    int accm = nestedStack.pop();
    nestedStack.push(accm);
    intermediateCode.add(EZConstants.ELSE.trim() + "_" + accm);
}
```

```
static HashMap<String,Integer> variables = new HashMap<String,Integer>();
static Stack<Integer> intStack = new Stack<Integer>();
static Stack<Boolean> boolStack = new Stack<Boolean>();
static ArrayList<String> codeList = new ArrayList<String>();
static int iteratorIndex = 0;
static Stack<Integer> nestedStack = new Stack<Integer>();
static Stack<Boolean> isLoop = new Stack<Boolean>();
static Stack<Integer> stackTrace = new Stack<Integer>();
static Stack<Integer> scope = new Stack<Integer>();
static int scopeCount = 1;
```

```
switch(split[0] + " ") {
    case EZConstants.ADD :
        accm = intStack.pop() + intStack.pop();
        intStack.push(accm);
        break;
    case EZConstants.ASSIGN :
        if (scope.isEmpty()) {
            variables.put(split[1], intStack.pop());
        } else {
            variables.put(getScope() + split[1], intStack.pop());
        }
        break;
    case EZConstants.BOOL:
        boolStack.push(Boolean.parseBoolean(split[1]));
        break;
```

RUNTIME

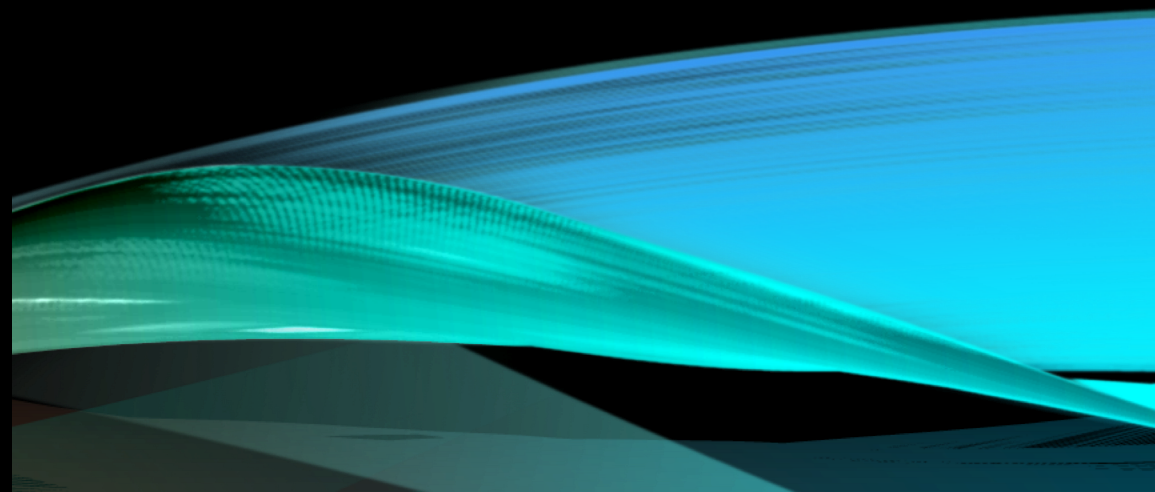


HOW DOES IT WORK?

```
/# Program to compute GCD  using recursion #/
variable x;
variable y;
variable z;
write "X";
read x;
write "Y";
read y;
z=gcd(x,y);
function gcd(x,y)
{
variable m;
if(y!=0)
then
{

m=gcd(y,x%y);
}
else
{
write "GCD";
write x;
}
return m;
}
```

SAMPLE CODE

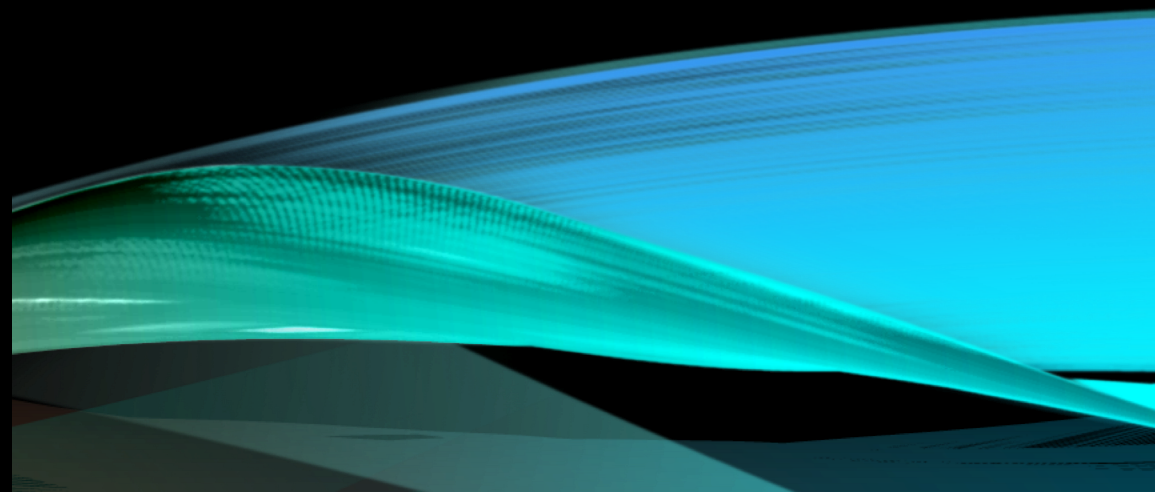
EZ Program to compute GCD



```
Haris-MacBook-Pro:SER-502-Spring-2018-Team29 harisiddarthvk$ java -jar compiler.jar resources/sample/gcd.ez
Haris-MacBook-Pro:SER-502-Spring-2018-Team29 harisiddarthvk$ cat resources/sample/gcd.ezi
DECL x
DECL y
DECL z
WRITE_STRING X
READ x
WRITE_STRING Y
READ y
LOAD x
LOAD y
FUNC CALL_gcd
MOV z
FUNC DECL_gcd
FUNC_PARAM #gcdx #gcdy
DECL #gcdm
IF_1
LOAD #gcdy
PUSH 0
NOT_EQUAL
COND_END
LOAD #gcdy
LOAD #gcdx
LOAD #gcdy
REM
FUNC CALL_gcd
MOV #gcdm
ELSE_1
WRITE_STRING GCD
LOAD #gcdx
WRITE
END IF_1
LOAD #gcdm
RETURN
FUNC END_gcd
```

SAMPLE CODE

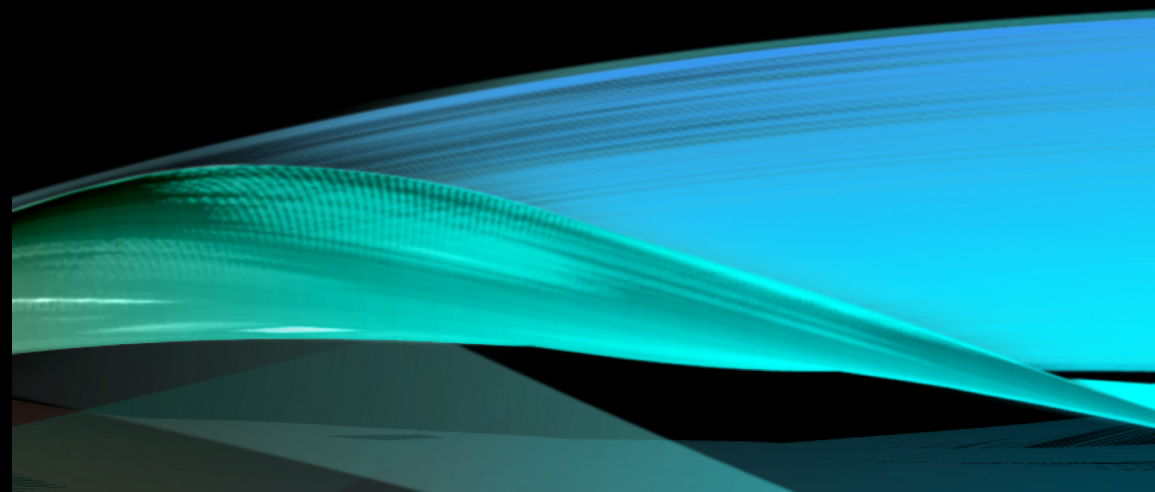
Intermediate code for GCD.ez



```
Haris-MacBook-Pro:SER-502-Spring-2018-Team29 harisiddarthvk$ java -jar runtime.jar resources/sample/gcd.ezi
X
10
Y
5
GCD
5
```

SAMPLE CODE

Runtime for GCD.ez



FUTURE SCOPE

- Future plan is to support:
  - Lists
  - Multiple datatypes
  - Data structures

THANK YOU!

No, we don’t have post credits.