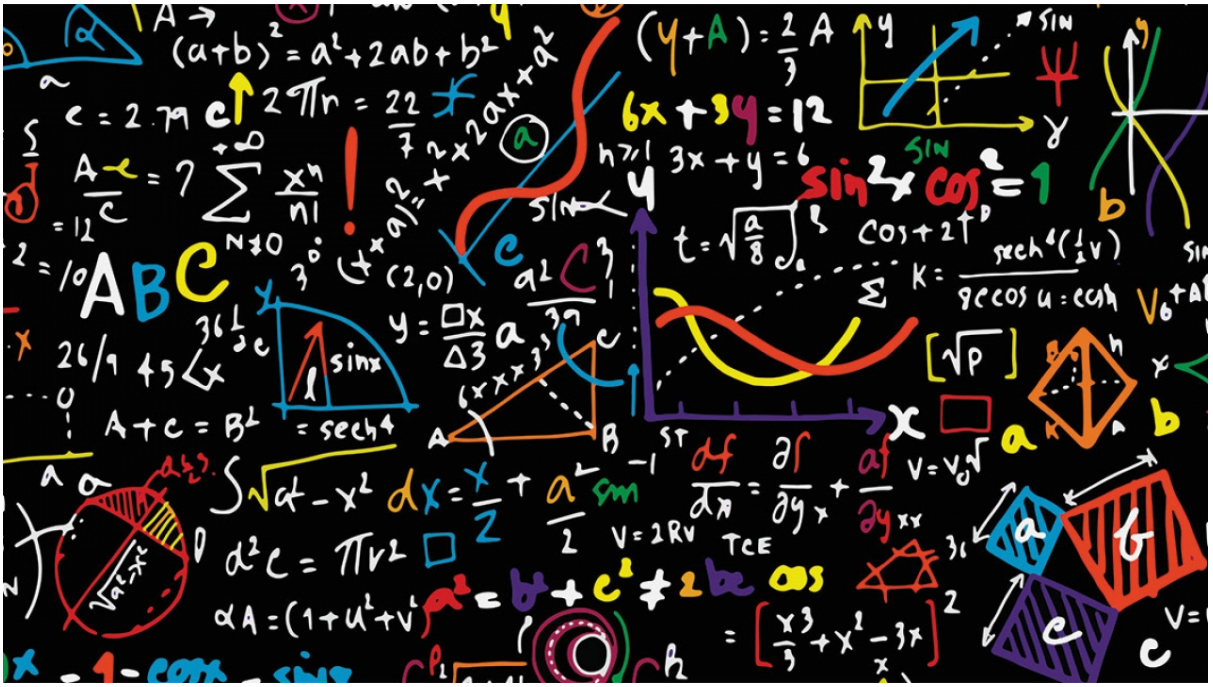
BISTAT



June 4, 2023

José Daniel Hernández Ríos A00825596

Summary of Contents

Project description **3**
Motivation and Scope 3
Language Requirements and Test Cases 3
Development Process 4
Commit History 4
Key Takeaways 9
Language Description **9**
Language Name 9
Language Features 9
Error Types 10
Compilation Errors 10
Halting Errors 10
Non Halting Errors 11
Runtime Errors 11
Compilation **11**
Development Tools 11
Lexical Analysis 12
Tokens 12
Syntactical Analysis 12
Intermediate Code Generation and Semantical Analysis 14
Operation codes and virtual addresses 14
Syntax diagram and intermediate code 16
Semantical Considerations 34
Memory Management 36
Execution **41**
Development Tools 41
Memory Management 41
Functionality Tests **45**
Basic Functionality 45
Domain Specific Functionality 54
User Manual **57**
Quick Reference 57
Demo 58

Project description

Motivation and Scope

The purpose of the compiler design class is for us to understand the ins and outs of a compiler's functioning and development. To this avail, the final project consists of developing a programming language of our own by defining a grammar, generating a parser and lexer with a lexer/parser generation tool, and subsequently traversing our language's parse tree in order to generate intermediate code that can then be executed by our own virtual machine.

Language Requirements and Test Cases

Bistat supports the following features:

- Basic arithmetic and logical operations on primitive data types: +, -, *, /, %, ==, !=, >, <, <=, >=, not()
    - Tested in exps.bs and basic.bs
- Control statements: if, while, for (only for arrays), which can be nested.
    - Tested in exps.bs, for.bs and basic.bs
- User function definition. Functions can be void but can't receive arrays/matrices as parameters nor can they return them (only atomic types). However, functions can define local arrays or matrices.
    - Tested in funcs.bs, factorial.bs and fibonacci.bs
- Arrays/matrices definition, as well as indexing
    - Tested in test_special_functions.bs and matmul.bs
- I/O functionality: print primitive types, arrays and matrices to the console as well as reading primitive types, arrays and matrices from console.
    - Tested in io.bs
- Mathematical functions that support primitive types as well as matrices/arrays: sqrt, sin, cos, tan, abs, floor, ceil, not
    - Tested in test_special_functions.bs
- Statistical aggregation functions that support arrays and matrices: sum, prod, min, max, avg, sMode, median
    - Tested in test_special_functions.bs
- Plotting matrices and arrays to the terminal
    - Tested in test_special_functions.bs

Memory management requirements

- Calculate the memory size of each type used for global, temporary and constant memory.
- Calculate the memory size of each type used for temporary and local memory in each function.
- Support function recursivity through the stack segment, allocating a new segment when a function is called and freeing it when a function's execution ends.

Development Process

The incremental changes in this project were made according to the project submissions outlined in Student Canvas, starting by the grammar creation and ending with intermediate code generation for arrays and domain specific functionality.

Commit History

```
Sun Jun 4 21:52:26 2023 -0600
    use indexing in tests and comment all source files

Sun Jun 4 18:19:24 2023 -0600
    Support indexing; refactor code to generate code for non
aggregate functions

Sun Jun 4 11:15:14 2023 -0600
    add more tests

Sun Jun 4 06:16:23 2023 -0600
    deref addresses in logical operations

Sun Jun 4 03:46:59 2023 -0600
    fix semantic cube

Sun Jun 4 00:16:11 2023 -0600
    avoid calculating function resource amount twice

Sun Jun 4 00:05:27 2023 -0600
    Throw error when redefining a function

Sat Jun 3 18:50:42 2023 -0600
    folder and file rename

Sat Jun 3 18:08:30 2023 -0600
    minor grammar changes

Sat Jun 3 05:42:32 2023 -0600
    plot

Sat Jun 3 04:15:11 2023 -0600
    rename execution folder
```

```
Sat Jun 3 04:14:00 2023 -0600
    generate binary file with the same name as input file

Sat Jun 3 03:58:42 2023 -0600
    handle aggregation stats functions

Fri Jun 2 20:16:23 2023 -0600
    handle non aggregation statistics functions

Thu Jun 1 14:17:39 2023 -0600
    add printQuads flag when compiling

Thu Jun 1 13:53:40 2023 -0600
    implement read in vm

Wed May 31 19:14:26 2023 -0600
    support assigning onw matrix to another matrix

Wed May 31 18:30:59 2023 -0600
    handle for loop

Wed May 31 00:47:48 2023 -0600
    add refsum and refmul for ref operations

Wed May 31 00:16:35 2023 -0600
    avoid double derefing

Wed May 31 00:03:38 2023 -0600
    fix issue with listAccess inside of listAsign

Tue May 30 22:44:02 2023 -0600
    WIP handle listAssign with listAccess as arg

Tue May 30 20:27:24 2023 -0600
    handle function execution in vm

Tue May 30 03:34:06 2023 -0600
    avoid creating double refs in array operations

Tue May 30 02:31:39 2023 -0600
```

```
    print print args from left to right and support printing arrays
and matrices

Mon May 29 13:48:29 2023 -0600
    fix quad generation for listAccess and listAssign

Mon May 29 02:25:51 2023 -0600
    handle quad generation for listAssign and listAccess

Sun May 28 16:09:03 2023 -0600
    array indexing

Sat May 27 17:43:04 2023 -0600
    don't allow arrays as function parameters or function return
types

Tue May 23 03:47:10 2023 -0600
    handle print

Tue May 23 03:31:00 2023 -0600
    fake bottom for function call and indexing

Tue May 23 03:06:23 2023 -0600
    handle logic

Tue May 23 02:36:39 2023 -0600
    handle basic arithmetic

Tue May 23 01:59:58 2023 -0600
    handle assign

Sun May 21 02:26:03 2023 -0600
    create execution context for running object code

Sun May 21 01:01:16 2023 -0600
    Support encoding ObjCode in .gob

Sat May 20 23:06:28 2023 -0600
    handle array indexing

Sat May 20 18:08:51 2023 -0600
```

```
    handle array declaration

Tue May 16 16:14:28 2023 -0600
    Handle return stmt

Tue May 16 15:40:26 2023 -0600
    Add End quad and store functions as variables in var table

Sun May 14 18:31:06 2023 -0600
    quad generation for functions

Wed May 10 18:16:59 2023 -0600
    generate nquads for for loops

Wed May 10 17:26:35 2023 -0600
    reorganize code

Wed May 10 17:08:04 2023 -0600
    quad generation for while loop

Wed May 10 16:48:55 2023 -0600
    quad generatino for conditionals

Wed May 3 20:04:10 2023 -0600
    generate quads for special functions

Wed May 3 19:33:40 2023 -0600
    generate quads for assignment

Wed May 3 19:01:06 2023 -0600
    validate types when generating quads

Tue May 2 20:04:06 2023 -0600
    basic quad generation for + - * / %; type checking missing

Mon May 1 20:21:50 2023 -0600
    create methods for pO and pOper manipulation

Mon May 1 19:46:06 2023 -0600
```

```
    create vm and address manager for vm; add logic for getting next
available address in variable declaration, function definition and
param declaration

Sun Apr 30 00:59:32 2023 -0600
    Create consTable and vm for memory management

Thu Apr 27 14:48:36 2023 -0600
    modularize code

Sun Apr 23 23:54:06 2023 -0600
    Create funcDir and varTable

Sun Apr 23 23:11:40 2023 -0600
    Create RType struct to store type information

Sun Apr 23 22:18:54 2023 -0600
    Define operator enums and semantic cube. Change grammar to fix
detection errors

Sun Apr 16 22:13:06 2023 -0600
    Add readme

Sun Apr 16 21:04:55 2023 -0600
    Add more listeners to test rules

Sun Apr 16 20:20:16 2023 -0600
    Use camel case in rules

Sun Apr 16 20:15:00 2023 -0600
    Add rule listeners to test they're being called correctly

Sun Apr 16 19:16:46 2023 -0600
    version inicial del grammar en antlr

Wed Apr 12 18:11:35 2023 -0600
    Create main program to walk antlr parse tree

Wed Apr 12 17:50:12 2023 -0600
    Init go module with grammar in text format
```

Key Takeaways

This project was a great opportunity to pick up a language I wasn't familiar with, so I chose Go in order to strengthen my skills, and that was one of the things I enjoyed the most about development, since this language has a very flat learning curve while still being powerful and performant and providing a clear mental image about data flow with its static type checking system. I really enjoyed this class because I always wondered how programming languages worked and, like most things in computer science, it seems like magic until you learn how it works, so that was a very gratifying process for me but it didn't come without challenges. There was a somewhat steep learning curve when it came to becoming familiar with syntax diagrams and intermediate code generation, as well as knowing how to manage stacks that hold information like jumps, operators or operands. However, the more time I invested in this project, the easier it became in the end, so I feel like I ended up really understanding how this process works. I also struggled with understanding how virtual memory addresses worked in the virtual machine, particularly in the stack segment, but I cleared my doubts during class. The last topic I struggled with at first were references, since they seemed unnecessary at first but the more I thought about them, the more they made sense. However, I often found myself stumbling when writing quadruples with references and during execution I found numerous bugs that I had no idea how to fix, but I got the hang of it in the end and all of the arrays/matrices tests ended up working, so that was also very satisfying. All in all, this was inarguably the school project I've enjoyed the most in the whole major, and I'm somewhat sad that it's come to an end, but I'll remember it fondly.

Language Description

Language Name

Beast (my nickname) + stats = Bistat.

Language Features

Bistat is a statistical and scientific computing language and as such defines mathematical and aggregation functions such as trigonometric operations and average and median functions to name a few, besides supporting basic features like arithmetic operations, list and matrix manipulation, user defined functions, as well as input/output interaction. As most compiled programming languages, Bistat is statically typed and supports the following primitive types: *int*, *float*, *bool* and *string*.

Error Types

Compilation Errors

Halting Errors

These errors will stop the compilation process

- Variable not found
    - “Variable {var} not found in scope”
- Non boolean expression used in conditional, while or for statement
    - “Expression type must be a boolean”
- The memory usage has been exceeded for a particular type
    - “Out of memory”
- For loop used in an atomic expression or matrix
    - “For loops can only be used in arrays”
- Control variable in for loop has a different type than the array type
    - “Control type must be the same type as expression variable in for loop”
- Void parameter declared in function
    - “Can’t use void parameters”
- Array parameter declared in function
    - "Can't use arrays as parameters in function definition"
- Non atomic type in function return statement
    - "Can't return an array or matrix in a function"
- Incorrect return type in function
    - "Incorrect return type for function {funcName}, expected {type1}, found {type2}”
- Calling an undefined function
    - “Function {funcName} was not defined”
- Incorrect number of arguments in function call
    - "Number of arguments in function call to {funcName} doesn't match the number of parameters with which it was defined"
- Incorrect type of argument in function call
    - "Type mismatch between argument and parameter in argument #{numArg} in function call to {funcName}”
- Declaring a void variable
    - “Can’t declare void variables”
- Redeclaring a variable
    - “Variable {varName} already exists”
- Assigning a matrix to a non matrix variable
    - “Variable {varName} isn’t a matrix”
- Incorrect number of rows in matrix assignment
    - “Incorrect number of rows in assignment to {matrixName}, expected {num1}, got {num2}”
- Incorrect number of columns in matrix assignment
    - “Incorrect number of columns in assignment to {matrixName}, expected {num1}, got {num2}”
- Assigning an array to a non array variable

    - “Variable {varName} isn’t an array”
- Incorrect number of elements in array assignment
    - “Incorrect number of elements in assignment to {arrayName}, expected {num1}, got {num2}”
- Type mismatch in assignment
    - “Cannot assign to {varName} because of type mismatch: expected {type1}, got {type2}”
- Using a special function with a non supported type
    - “{funcName} can only be called on {supportedType}”
- Using an array/matrix special function with an atomic type
    - “{funcName} can only be called on arrays or matrices”
- Performing an arithmetic or logical operation on unsupported types:
    - “Cannot perform {operation} on {type1} and {type2}”

Non Halting Errors

Due to the nature of ANTLR, the tool used for lexer/parser generators, lexing and syntax errors are not available to the rule listener and will thus not cause it to halt, and the compiler can crash in some cases. When this happens, the errors will appear above the compiler error message.
This can happen in the following cases

- When an array is declared with no dimensions (var int arr[];).
- When a void user defined function or special function is used in an expression (1 + print(3);).

Runtime Errors

These errors will halt program execution

- Out of bounds array/matrix access
    - “Access {index} out of bounds for element with dimension {dimension}”
- Stack length reaches 1000001 causing an overflow
    - “Stack overflow”

Compilation

Development Tools

This project was developed on MacOS *Big Sur (11.1)*. The compiler and virtual machine were developed in Go version *1.20.1 darwin/amd64*. ANTLR4 was used to generate the lexer and parser. The following external Go packages were used: *antlr4* for the parser and *exp* for array manipulation.

Lexical Analysis

Tokens

BOOL_CONS: “true” | “false”
NUMBER: [0-9]
STRING_CONS: “\”” [^”]* “\””
INT_CONS: NUMBER+
FLOAT_CONS: NUMBER+\.NUMBER+
Alpha: [a-zA-Z]
ID: (‘_’ | Alpha)+ (Alpha | NUMBER | ‘_’)+
COMMENT: #[^#]+#
OP_SEC: “+” | “-”
OP_FIRST: “/” | “%” | “*”
LOGIC_OPERATOR: “==” | “!=” | “&&” | “||” | “>” | “<” | “>=” | “<=”
CARDINALITY: “[“INT_CONS”]”(“[“INT_CONS”]”)?

No tokens were defined on single characters (ex.: LEFT_PAREN: “(“) since this is not required by ANTLR.

Syntactical Analysis

The grammar used for this language is outlined below

```
program ::= 'Program' id ';' var_declaration* func_def* main
main ::= 'main' '()' '{' statement+ '}'
func_def ::= 'func' id '(' param_declaration* ')' ':' atomic_type
var_declaration* '{' statement* '}'
param_declaration ::= 'var' atomic_type id ';'
var_declaration ::= 'var' var_type id ';'
indexing ::= id '{' expression '}' ('{' expression '}')?
assignment ::= (indexing | id) '=' (var_cons | list_assignment |
matrix_assignment)
matrix_assignment ::= '[' list_assignment (',' list_assignment)*
','? ']'
list_assignment ::= '[' var_cons (',' var_cons)* ']'
comment ::= '#' [^#]* '#'
statement ::= ((assignment | special_function | function_call |
return_stmt) ';') | conditional | for | while | comment
return_stmt ::= 'return' expression
special_function ::= read | print | plot | sum | min | max | prod |
avg | mode | median | sin | cos | tan | sqrt | floor | ceil | abs |
not
read ::= 'read' '(' id (',' id)* ')'
```

```
plot ::= 'plot' '(' expression ')'
sum ::= 'sum' '(' expression ')'
min ::= 'min' '(' expression ')'
max ::= 'max' '(' expression ')'
prod ::= 'prod' '(' expression ')'
avg ::= 'avg' '(' expression ')'
mode ::= 'sMode' '(' expression ')'
median ::= 'median' '(' expression ')'
sin ::= 'sin' '(' expression ')'
tan ::= 'tan' '(' expression ')'
cos ::= 'cos' '(' expression ')'
sqrt ::= 'sqrt' '(' expression ')'
floor ::= 'floor' '(' expression ')'
ceil ::= 'ceil' '(' expression ')'
abs ::= 'abs' '(' expression ')'
not ::= 'not' '(' expression ')'
for ::= 'for' '(' id 'in' expression ')' '{' statement+ '}'
while ::= 'while' '(' expression ')' '{' statement+ '}'
conditional ::= ('if' '(' expression ')' '{' statement+ '}') ('else
if' '(' expression ')' '{' statement+ '}')* ('else' '{' statement+
'}')?
print ::= 'print' '(' (expression) ( ',' (expression) )* ')'
expression ::= exp (('<' | '>' | '<=' | '>=' | '==' | '!=' | '&&' |
'||') exp)?
exp ::= term (('+' | '-') term)?*
function_call ::= (id '(' (expression (',' expression)*)? ')')
term ::= factor (('*' | '/' | '%') factor)*
factor ::= '-'? (('('expression')') | indexing | special_function |
function_call | var_cons)
atomic_type ::= ('int' | 'float' | 'string' | 'bool' | 'void')
var_type ::= ('int' | 'float' | 'string' | 'bool') cardinality?
cardinality ::= '[' int_cons ']' | '[' int_cons ']' '[' int_cons ']'
var_cons ::= string_cons | int_cons | float_cons | id | bool_cons
string_cons ::= '"' [^"]* '"'
float_cons ::= [0-9]+'.'[0-9]+
int_cons ::= [0-9]+
bool_cons ::= 'true' | 'false'
id ::= ('_'| [a-zA-Z] )+[0-9a-zA-Z]*
```

Intermediate Code Generation and Semantical Analysis

Operation codes and virtual addresses

Operation codes were defined as an enum *Op* (which is an int) that can take on the following values:

```
const (
        Sum Op = iota // 0
        Subtraction // 1
        Multiplication // 2
        Division // 3
        Modulus // 4
        And // 5
        Or // 6
        Gt // 7
        Lt // 8
        Ge // 9
        Le // 10
        Eq // 11
        Ne // 12
        Assign // 13
        InputRead // 14
        Print // 15
        PrintN // 16
        ListSum // 17
        Min // 18
        Max // 19
        Prod // 20
        Avg // 21
        SMode // 22
        Median // 23
        Plot // 24
        Sin // 25
        Cos // 26
        Tan // 27
        Sqrt // 28
        Floor // 29
        Ceil // 30
        Abs // 31
        Not // 32
        UnaryMinus // 33
        Goto // 34
        GotoT // 35
        GotoF // 36
        GoSub // 37
        Era // 38
        Param // 39
        EndFunc // 40
```

```
        End // 41
        Return // 42
        RefSum // 43
        RefMul // 44
        UndefinedOp // 45
        Verify // 46
        Other // 47
)
```

The virtual addresses are defined below:

```
const GLOBAL_INT_START = 0
const GLOBAL_FLOAT_START = 10000
const GLOBAL_STRING_START = 20000
const GLOBAL_BOOL_START = 30000
const GLOBAL_BOOL_END = 39999
const LOCAL_INT_START = 40000
const LOCAL_FLOAT_START = 50000
const LOCAL_STRING_START = 60000
const LOCAL_BOOL_START = 70000
const TEMP_INT_START = 80000
const TEMP_FLOAT_START = 90000
const TEMP_STRING_START = 100000
const TEMP_BOOL_START = 110000
const CONST_INT_START = 120000
const CONST_FLOAT_START = 130000
const CONST_STRING_START = 140000
const CONST_BOOL_START = 150000
const CONST_BOOL_END = 150003
const GLOBAL_REF_START = 150003
const LOCAL_REF_START = 160003
const LOCAL_REF_END = 170003
```

Each (start, end) pair is managed by the struct *AddressManager* defined as follows

```
type AddressManager struct {
 first, curr, last int
}
```

This struct has the following methods

- GetNext(): increments curr and returns the value before the increment
- GetSize(): returns curr - first
- Reset(): sets curr to first

There is an address manager for each primitive type and memory type (constant, local, global or temporary) as well as local reference and global reference manager. These address managers can be found in the struct *VM*, which is a *virtual* virtual machine, or a virtual address manager. This struct also holds the quadruples that will be used during execution

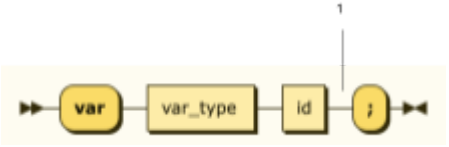
```
func NewVM() VM {
 return VM{
   quads:                  make([]Quad, 0),
```

```
   globalIntAddressMgr:    NewAddressManager(GLOBAL_INT_START,
GLOBAL_FLOAT_START-1),
   globalFloatAddressMgr:  NewAddressManager(GLOBAL_FLOAT_START,
GLOBAL_BOOL_START-1),
   globalStringAddressMgr: NewAddressManager(GLOBAL_STRING_START,
GLOBAL_BOOL_START-1),
   globalBoolAddressMgr:   NewAddressManager(GLOBAL_BOOL_START, LOCAL_INT_START-1),
   localIntAddressMgr:     NewAddressManager(LOCAL_INT_START, LOCAL_FLOAT_START-1),
   localFloatAddressMgr:   NewAddressManager(LOCAL_FLOAT_START,
LOCAL_STRING_START-1),
   localStringAddressMgr:  NewAddressManager(LOCAL_STRING_START,
LOCAL_BOOL_START-1),
   localBoolAddressMgr:    NewAddressManager(LOCAL_BOOL_START, TEMP_INT_START-1),
   tempIntAddressMgr:      NewAddressManager(TEMP_INT_START, TEMP_FLOAT_START-1),
   tempFloatAddressMgr:    NewAddressManager(TEMP_FLOAT_START,
TEMP_STRING_START-1),
   tempStringAddressMgr:   NewAddressManager(TEMP_STRING_START, TEMP_BOOL_START-1),
   tempBoolAddressMgr:     NewAddressManager(TEMP_BOOL_START, CONST_INT_START-1),
   constIntAddressMgr:     NewAddressManager(CONST_INT_START, CONST_FLOAT_START-1),
   constFloatAddressMgr:   NewAddressManager(CONST_FLOAT_START,
CONST_STRING_START-1),
   constStringAddressMgr:  NewAddressManager(CONST_STRING_START,
CONST_BOOL_START-1),
   constBoolAddressMgr:    NewAddressManager(CONST_BOOL_START, CONST_BOOL_END),
   globalRefAddressMgr:    NewAddressManager(GLOBAL_REF_START, LOCAL_REF_START-1),
   localRefAddressMgr:     NewAddressManager(LOCAL_REF_START, LOCAL_REF_END-1),
 }
}
```

Syntax diagram and intermediate code

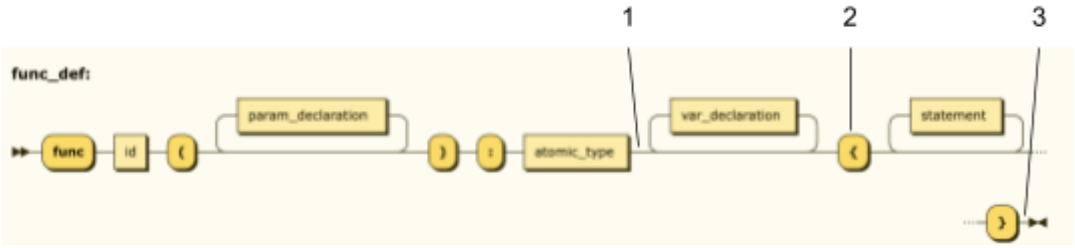
1. funcData = newFuncData(void)
   funcData.idx = 0
   addScope(“main”)
   funcDir.add(“main”, funcData)
   pushQuad(Goto, _, _, _)

// This first quad will contain the quad number of the main function body’s beginning.



```
1.  currScope = getCurrentScope()
    if var_type.primitive_type == Void {
            Error("Can't declare void variables")
    } else {
            addrMgr = getAddressManager(var_type.primitive_type)
            resolvedType = resolveType(var_type, addrMgr)
            if getVarInScope(currScope, id) != null {
                    Error("variable already declared")
            } else {
                    AddVarToScope(currScope, id, resolvedType)
                    AddVarToAddrTable(resolvedType.address, id)
            }
    }
```

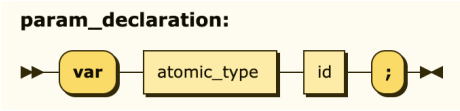
// Here, the resolveType function receives a var_type (which has a primitive type and a cardinality in text form) and an address manager and returns a struct containing a type in enum form and attributes for first and second dimensions as well as an address. The getAddressManager function returns either a local or global address manager of the same type as the variable's atomic type

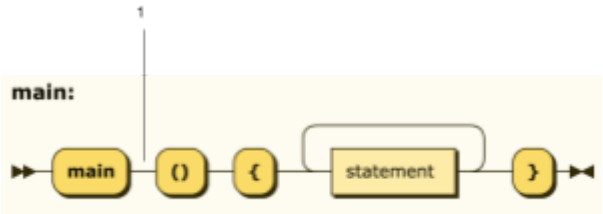



```
1.  if funcDir[id] != null {
            Error("Function already declared")
    } else {
            funcType = getAtomicTypeFromString(atomic_type)
            addrMgr = getAddressManager(funcType)
            resolvedType = NewResolvedType(funcType)
            resolvedType.address = addrMgr.getNext()
            AddVarToScope("main", id, resolved)
            AddScope(id)
            funcData = ResolvedTypeToFuncData(resolvedType)
            funcData.idx = functions.length
            // add function name to function names array
            functions.add(id)
            //  add function data to function directory
            funcDir[id] = funcData
            params = []
            for param in param_declaration {
                    type = param.type_primitive
                    if type == Void {
                            Error("Can't use void parameters")
                            return
                    }
                    resolved = NewResolvedType(param.type_primitive)
                    if resolved.firstDim > 0 {
                            Error("Can't use arrays as parameters")
                            return
                    }
                    addrMgr = getAddressManager(param.type_primitive)
                    resolved.address = addrMgr.getNext()
                    AddVarToScope(getCurrentScope(), param.id, resolved)
                    params.push(resolved)
            }
```

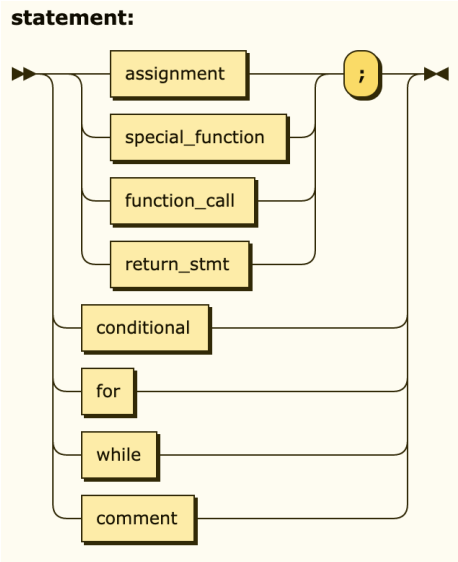
```
            params[id] = params
            funcDir[id].params = params.length

            functions.push(id)
    }
    // function and parameter data is stored and virtual addresses are assigned
```

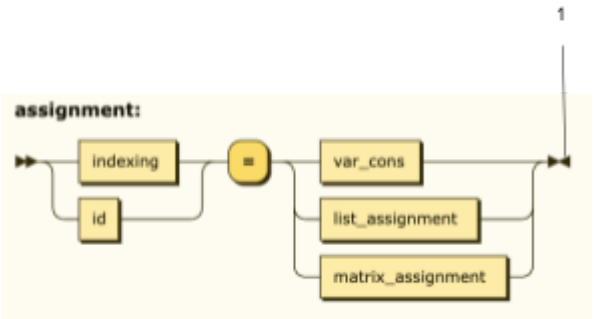
2. funcName = getCurrentScope()
   funcData = funcDir[funcName]
   funcData.localInts = localIntAddrMgr.getSize()
   funcData.localFloats = localFloatAddrMgr.GetSize()
   funcData.localBools = localBoolAddrMgr.GetSize()
   funcData.LocalStrings = localStringAddrMgr.GetSize()
   funcData.start = quads.length
   // store local resource usage and quad start of function body
3. funcName = getCurrentScope()
   funcData = funcDir[funcName]
   funcData.tempInts = tempIntAddrMgr.getSize()
   funcData.tempFloats = tempFloatAddrMgr.GetSize()
   funcData.tempBools = tempBoolAddrMgr.GetSize()
   funcData.tempStrings = tempStringAddrMgr.GetSize()
   tempIntAddrMgr.reset()
   tempFloatAddrMgr.reset()
   tempBoolAddrMgr.reset()
   tempStringAddrMgr.reset()
   localIntAddrMgr.reset()
   localFloatAddrMgr.reset()
   localBoolAddrMgr.reset()
   localStringAddrMgr.reset()
   quads.push(EndFunc, _, _, _)
   RemoveFunctionFromVarTable(funcName)
   PopCurrentScope()
   // virtual addresses are reset, function eliminated from var table and scope popped





1. funcData = funcDir[“main”]
   funcData.start = quads.length
   quads[0].destination = quads.length

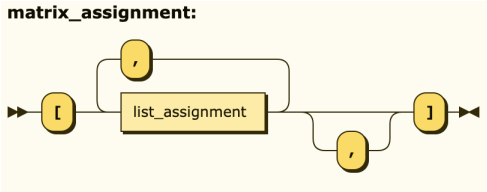


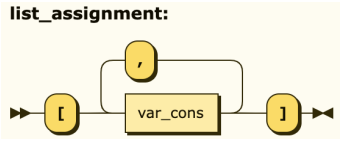


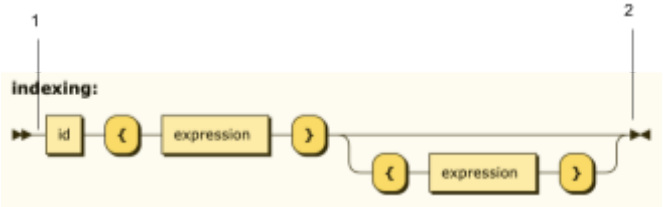
1.
```
varName = “”
leftVal = NewResolvedType(bool)
if id != null {
        leftVal = GetResolvedTypeFromVarName(id)
        if leftVal == null {
                Error(“variable not defined”)
                return
        }
} else {
        varName = indexing.id
        size = 1
        if list_assignment != null {
                size = list_assignment.expressions.length
        } else if matrix_assignment != null {
                Error(“Cannot assign a matrix with an indexing expression on the left side”)
                return
        }
        leftVal = pO[pO.length - size - 1]
}
if list_assignment != null || matriix_assignment != null {
        size = 0
        if list_assignment != null {
//list
                if leftVal.firstDim <= 0 {
                        Error(“Variable is not an array”)
                        return
                }
                if leftVal.firstDim != list_assignment.expressions.length {
                        Error(“Incorrect number of elements in assignment to array”)
```

```
                        return
                }
                size = leftVal.firstDim
        } else {
        //matrix
                if leftVal.secondDim <= 0 {
                        Error("Variable is not a matrix")
                        return
                }
                if leftVal.firstDim != matrix_assignment.list_assignments.length {
                        Error("Incorrect number of rows in matrix assignment")
                        return
                }
                for row in matrix_assignment.list_assignments {
                        cols = row.expressions.length
                        if cols != leftVal.secondDim {
                                Error("Incorrect number of columns in matrix assignment")
                                return
                        }
                }
                size = leftVal.firstDim * leftVal.secondDim
        }
        startAddr = leftVal.address
        addMgr = getRefAddrMgr(leftVal.address)
        for (i = size; i > 0; i--) {
                elem = pO.pop()
                refAddr = addrMgr.next()
                quads.push(RefSum, startAddr, i - 1, refAddr)
                quads.push(Assign, refAddr, _, elem.address)
        }

} else {
        rightVal = pO.pop()
        if leftVal.type != rightVal.type {
                Error("type mismatch during assignment")
                return
        }
        if rightVal.firstDim > 0 {
                if leftVal.firstDim != rightVal.firstDim {
                        Error("Dimension mismatch during assignment")
                        return
                }
                size = rightVal.firstDim
                if leftVal.secondDim > 0 {
                        if leftVal.secondDim != rightVal.secondDim {
                                Error("Dimension mismatch during assignment")
                                return
                        }
                        size *= rightVal.secondDim
                }
                lAddrMgr = GetRefAddressManager(leftVal.address)
                rAddrMgr = GetRefAddressManager(rightVal.address)
                for (i = 0; i < size; i ++) {
                        rightAddr = rAddrMgr.GetNext()
                        leftAddr = lAddrMgr.GetNext()
                        quads.push(RefSum, leftVal.address, i, leftAddr)
                        quads.push(RefSum, rightVal.address, i, rightAddr)
                        quads.push(Assign, leftAddr, _, rightAddr)
```

```
                }
        } else {
                quads.push(Assign, leftVal.address, _, rightVal.address)
        }
}
if indexing != null {
        pO.pop()
}
```



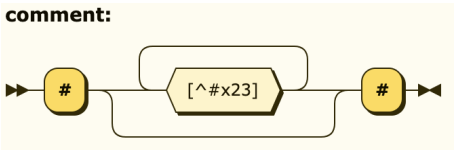


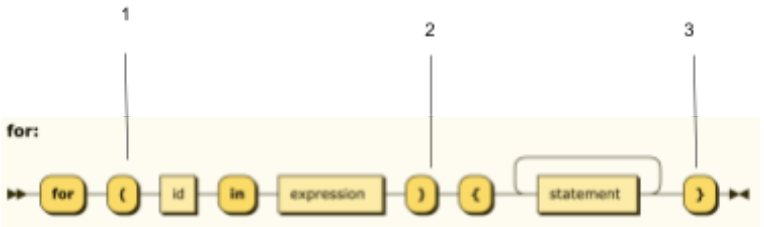


1. pOper.push(Other) // fake bottom
2. pOper.pop() // fake bottom

```
arr = GetResolvedTypeFromVarName(id)
if arr == null {
        Error("Variable not defined")
        return
}
if arr.firstDim == 0 {
        Error("Variable in indexing is not an array")
        return
}
levels = expressions.length
if levels == 1 {
        idx = pO.pop()
        if idx.type != int {
                Error("Expressions in indexing must be of type int")
                return
        }
        quads.push(Verify, idx.address, arr.address, arr.firstDim)
        addrMgr = getRefAddrMgr(arr.address)
        indexed = NewResolvedType(arr.type)
        refAddr = addrMgr.next()
        if arr.secondDim != 0 {
                quads.push(RefMul, arr.secondDim, idx.address, refAddr)
                endAddr = adrMgr.next()
                quads.push(RefSum, arr.address, refAddr, endAddr)
                indexed.firstDim = arr.secondDim
                indexed.address = endAddr
        } else {
                quads.push(RefMul, arr.secondDim, idx.address, refAddr)
                indexed.address = refAddr
```

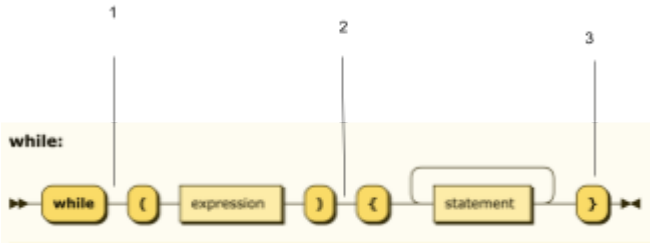
```
            }
            pO.push(indexed)
    } else {
            if arr.secondDim <- 0 {
                    Error("Variable isn't a matrix")
                    return
            }
            secondIdx = pO.pop()
            firstIdx = pO.pop()
            if firstIdx.type != int || secondIdx.type != int {
                    Error("Expression in indexing must be of type int")
                    return
            }
            quads.push(Verify, firstIdx.address, arr.address, arrFirstDim)
            quads.push(Verify, secondIdx.address, arr.address, secondDim)

            addrMgr = GetRefAddrMgr(arr.address)
            addr = addrMgr.next()
            quads.push(RefMul, arr.secondDim, firstIdx.address, addr)
            secondAddr = addrMgr.next()
            quads.push(RefSum, addr, secondIdx.address, secondAddr)
            endAddr = addrMgr.next()
            quads.push(RefSum, arr.address, secondAddr, endAddr)
            indexed = NewResolvedType(arr.type)
            indexed.address = endAddr
            pO.push(indexed)
    }
```



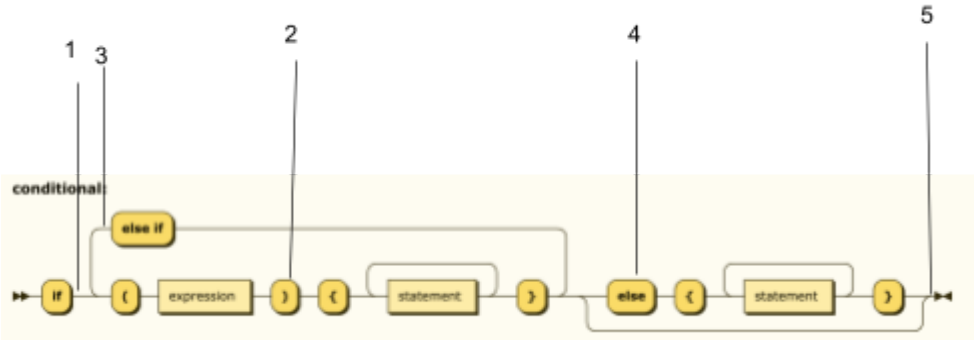


1. zeroResolvedType = constTable["0"] // value of control variable at start
   controlVariable = NewResolvedType(int)
   addr = tempIntAddrMgr.getNext()
   controlVariable.address = addr
   quads.push(Assign, controlVariable.address, _, zeroResolvedType.address) // start control variable at 0
   pO.push(controlVariable)
2. arrayElem = getResolvedTypeFromVarName(id)
   if arrayElem is null {
           Error("Control variable not defined in for loop")
           return
   }
   array = pO.pop()
   controlVar = pO.top()
   if o.FirstDim == 0 || o.SecondDim > 0 {
           Error("For loops can only be used in arrays")

```
            return
        }
        if o.type != controlVar.type {
            Error("Control variable must be the same as array variable")
            return
        }
        boolAddr = tempBoolAddrMgr.GetNext()
        jumps.push(quads.length)
        quads.push(Lt, controlVar.address, o.FirstDim, boolAddr)
        jumps.push(quads.length)
        quads.push(GotoF, boolAddr, _, _)
        refMgr = getRefAddrMgr(o.address)
        refAddr = refMgr.GetNext()
        quads.push(RefSum, controlVar.Address, o.address, refAddr)
        quads.push(Assign, arrayElem.address, _, refAddr)
3.  controlVar = pO.pop()
    oneResolvedType = constTable["1"] // to increment control variable
    quads.Push(Sum, controlVar.address, oneResolvedType.address, controlVar.address)
    end = jumps.pop()
    ret = jumps.pop()
    quads.push(Goto, _, _, ret)
    quads[end].destination = quads.length
```

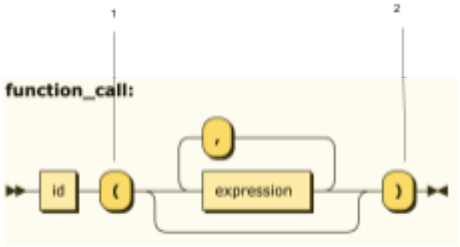


1. jumps.push(quads.length)
2. o = pO.pop()
   ```
   if o.type != boolean {
       Error("Expression type must be boolean")
       return
   }
   currQuad = quads.length
   quads.push(GotoF, o.address, _, _)
   jumps.push(currQuad)
   ```
3. end = jumps.pop()
   ```
   ret = jumps.pop()
   quads.push(Goto, _, _, ret)
   quads[end].destination = quads.length
   ```



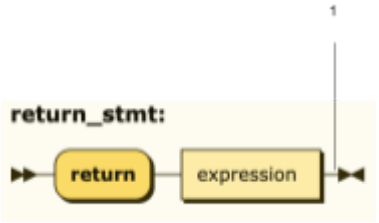
1. PushCondJumps() // add an array to condJumps matrix

2. o = pO.pop()
   if o.type != boolean {
   Error(“Expression type must be boolean”)
   return
   }
   currQuad = quads.length
   quads.push(GotoF, o.address, _, _)
   condJumpsPush(currQuad) // push to current array in condJumps
3. quads.push(Goto, _, _, _)
   jump = condJumps.current.pop()
   currQuad = quads.length
   quads[jump].destination = currQuad
   condJumps.current.push(currQuad - 1)
4. quads.push(Goto, _, _, _)
   jump = condJumps.current.pop()
   currQuad = quads.length
   quads[jump] = destination = currQuad
   condJumps.current.push(currQuad - 1)
5. currQuad = quads.length
   while !condJumps.current.empty() {
   jump = condJumps.current.pop()
   quads[jump].destination = currQuad
   }
   PopCondJumps() // pop array from matrix

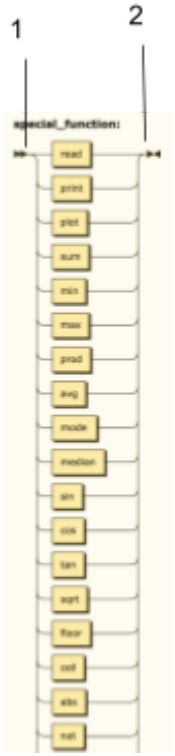


1. pOper.push(Other) // fake bottom
2. pOper.pop() // fake bottom
   funcData = funcDir[id]
   if funcData == null {
   Error(“Func data was not defined”)
   return
   }
   quads.push(Era, funcData.idx) // the index of the function in the functions array
   // validate param number
   if expressions.length != funcData.params {
   Error(“Incorrect number of params in function call”)
   return
   }
   // pass params in reverse because of stack
   for (i = funcData.params; i != 0; i--) {
   arg = pO.pop()
   param = paramTable[id][i - 1]
   if param.type != arg.type {
   Error(“incorrect argument type in function call”)
   return
   }
   quads.push(Param, param.address, -1, arg.address)
   }
   quads.push(GoSub, funcData.idx, -1, -1)
   if funcData.type != void {
   funcVar = getVarInScope(“main”, id)

```
        tempAddrMgr = getAddrMgr(funcData.type)
        temp = NewResolvedType(funcData.type)
        temp.address = tempAddrMgr.GetNext()
        // store return value in temp address
        quads.push(Assign, temp.address, _, funcVar.address)
        pO.push(temp)
}
```



1. o = pO.pop()
   ```
   if o.firstDim > 0 {
           Error("can't return array in function")
           return
   }
   funcVar = getVarInScope("main", getCurrentScope())
   if funcVar.type != o.type {
           Error("invalid type in function return statement")
           return
   }
   quads.push(Return, funcVar.address, _, o.Address)
   ```

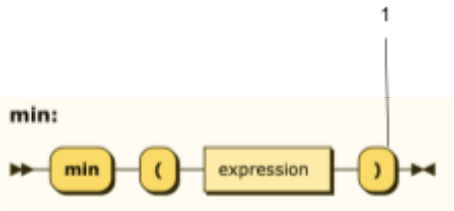


1. pOper.push(Other) // fake bottom

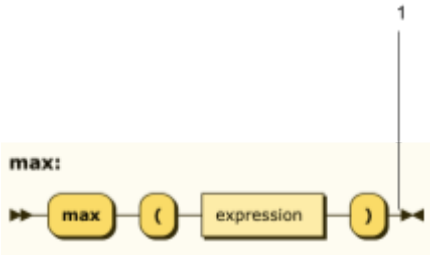
2. pOper.pop() // fake bottom

1. GenerateQuadsForAggregateFunction(ListSum, pO.top().type)

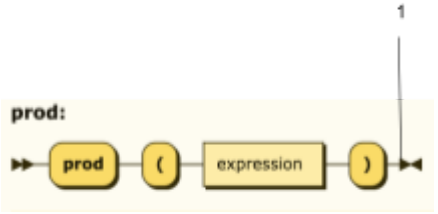
```
// GenerateQuadsForAggregateFunction definition
GenerateQuadsForAggregateFunction(OpCode, resultType) {
        o = pO.pop()
        if o.type != Float && o.Type != Int {
                Error("{opCode} can only be used on float and int expressions")
        }
        if o.firstDim == 0 {
                Error("{OpCode} can only be used on arrays")
        }
        quad = NewQuad(OpCode, o.address, o.FirstDim)
        result = NewResolvedType(resultType)
        if o.SecondDim != 0 {
                quad.Aux = o.SecondDim
                result.firstDim = o.FirstDim
                addrMgr = getTempAddrMgr(resultType)
                for (i = 0; i < o.FirstDim; i ++) {
                        addr = addrMgr.GetNext()
                        if i == 0 {
                                quad.Destination = addr
                                result.address = addr
                        }
                }
        } else {
                addrMgr = getTempAddrMgr(resultType)
                addr = addrMgr.GetNext()
                quad.destination = addr
                result.address = addr
        }
        quads.push(quad)
        pO.push(result)
}
```



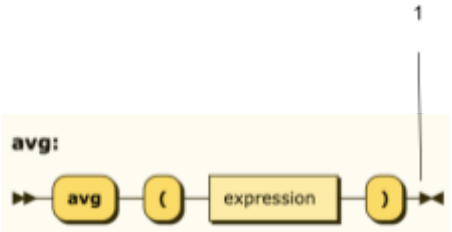
1. GenerateQuadsForAggregateFunction(Min, pO.top().type)



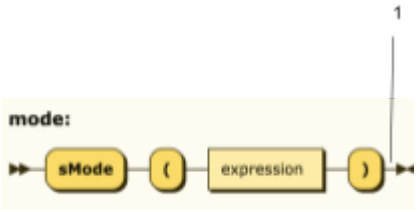
1. GenerateQuadsForAggregateFunction(Max, pO.top().type)



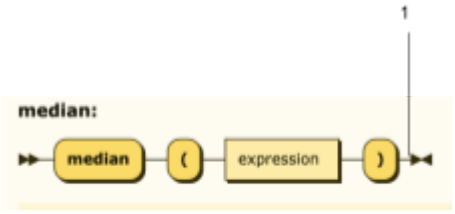
1. GenerateQuadsForAggregateFunction(Prod, pO.top().type)



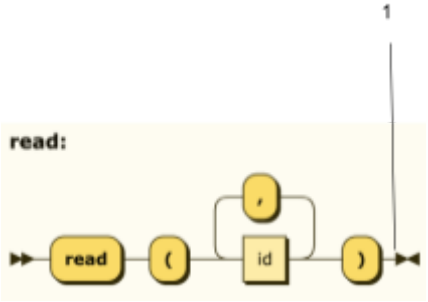
1. GenerateQuadsForAggregateFunction(Avg, float)



1. GenerateQuadsForAggregateFunction(SMode, pO.top().type)

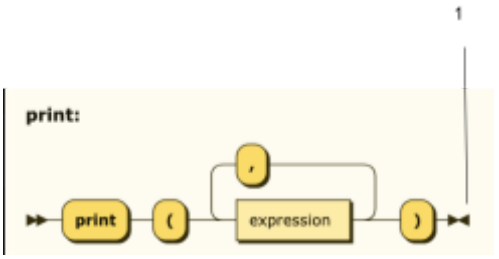


1. GenerateQuadsForAggregateFunction(Median, float)

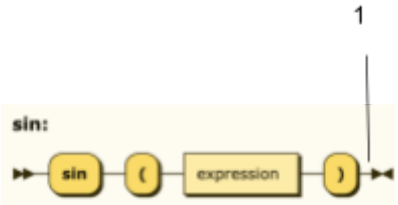


1. for id in ids {
   o = pO.pop()
   if o.firstDim > 0 {

```
                size = o.FirstDim
                if o.secondDim > 0 {
                        size *= o.SecondDim
                }
                addrMgr = getRefAddrMgr(o.address)
                for (i = 0; i < size; i++) {
                        addr = addMrg.getNext()
                        quads.push(RefSum, o.address, i, addr)
                        quads.push(InputRead, addr, _, _)
                }
        } else {
                quads.push(InputRead, addr, _, _)
        }
}
```



1.
```
toPrint = []
for expression in expressions {
        o = pO.pop()
        toPrint.insertAtStart(o) // reverse expressions because of stack
}
ws = consTable[“ “].address
for idx, element in toPrint {
        if elem.firstDim > 0 {
                lBrack = consTable[“[“]
                rBrack = consTable[“]”]
                size = elem.firstDim
                if elem.secondDim > 0 {
                        size *= elem.secondDim
                        quads.push(print, lbrack, _, _)
                }
                addrMgr = GetRefAddrMgr(elem.address)
                for (i = 0; i < size; i++) {
                        if i != 0 {
                                quads.print(ws, _, _, _)
                        }
                        if i == 0 || (elem.secondDim > 0 && i % elem.SecondDim == 0) {
                                quads.push(Print, lBrack, _, _)
                        }
                        addr = addrMgr.getNext()
                        quads.push(refSum, elem.address, i, addr)
                        quads.push(Print, addr, _, _)
                        if i == 0 || (elem.secondDim > 0 && (i+1) % elem.SecondDim == 0) {
                                quads.push(Print, lBrack, _, _)
                        }
                }
                quads.push(print, rbrack, _, _)
        } else {
                quads.push(print, elem.address, _, _)
        }
        quads.push(printN, _, _, _) // printN prints a newline
}
```

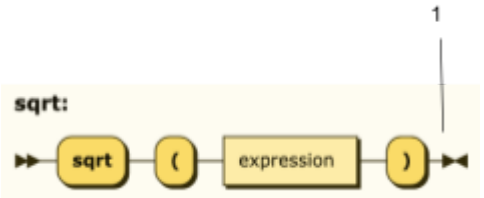


1. GenerateQuadsForNonAggregateFunction(Sin, float)

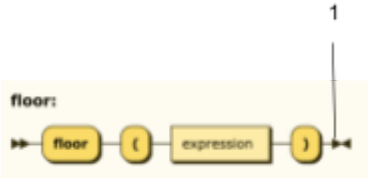
```
// GenerateQuadsForNonAggregateFunction definition
GenerateQuadsForNonAggregateFunction(opCode, resultType) {
        o = pO.pop()
        if opCode == Not && o.type != Bool {
                Error("Not can only be called on bool expressions")
                return
        }
        if (opCode == Ceil || opCode == Floor) && o.type != float {
                Error("{opCode} can only be called on float expressions")
                return
        }
        if (opCode == Sin || opCode == Tan || opCode == Cos || opCode == Abs || opCode == Sqrt) && o.type !=
float && o.type != int {
                Error("{opCode} can only be called on float or int expressions")
                return
        }
        result = NewResolvedType(resultType)
        if o.FirstDim > 0 {
                size = o.firstDim
                if o.secondDim > 0 {
                        size *= o.secondDim
                }
                addrMgr = GetRefAddrMgr(o.address)
                tempAddrMgr = GetTempAddrMgr(resultType)
                result.firstDim = o.firstDim
                result.secondDim = o.secondDim
                for (i = 0; i < size; i++) {
                        refAddr = addrMgr.next()
                        tempAddr = tempAddrMgr.next()
                        if i == 0 {
                                result.address = tempAddr
                        }
                        quads.push(RefSum, o.address, i, refAddr)
                        quads.push(opCode, refAddr, _, tempAddr)
                }
        } else {
                addr = GetTempAddrMgr(resultType).next()
                result.address = addr
                quads.push(opCode, o.Address, _, addr)
        }
        pO.push(result)
}
```

1. GenerateQuadsForNonAggregateFunction(Tan, float)

1. GenerateQuadsForNonAggregateFunction(Cos, float)



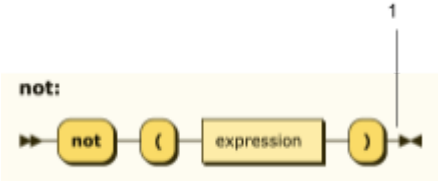
1. GenerateQuadsForNonAggregateFunction(Sqrt, pO.top().type)



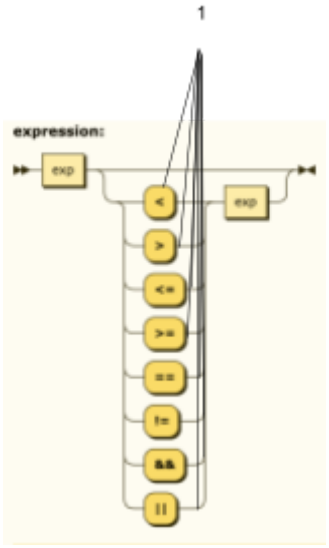
1. GenerateQuadsForNonAggregateFunction(Floor, int)

1. GenerateQuadsForNonAggregateFunction(Ceil, int)

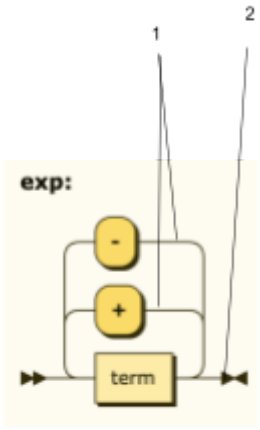
1. GenerateQuadsForNonAggregateFunction(Abs, pO.top().type)



1. GenerateQuadsForNonAggregateFunction(Not, bool)



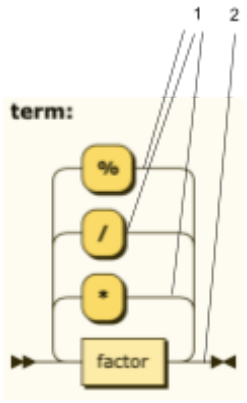
1. POper.push(oper)



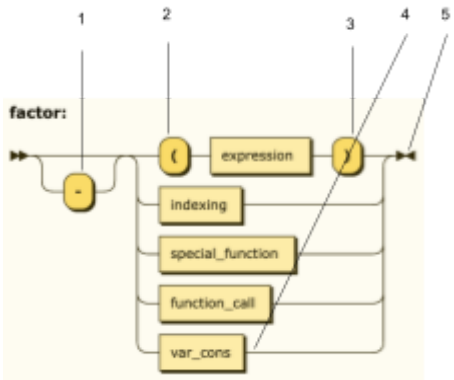
1. POper.push(oper)
2. if !pOper.empty && (pOper.top == '>' || pOper.top == '>=' || pOper.top == '==' || pOper.top == '!=' || pOper.top == '>=' || pOper.top == '<=' || pOper.top == '||' || pOper.top == '&&') {
   generateQuadsForExpression()
   }

```
// generateQuadsForExpression definition
generateQuadsForExpression() {
        oper = pOper.pop()
        o1 = pO.pop()
        if oper == UnaryMinus {
                if i1.type != int && o2.type != float {
                        Error("Can't use unaryMinus in type {o1.type}")
                        return
                }
                addr = getTempAddressMgr(o1.type).getNext()
                quads.push(UnaryMinus, o1.address, _, addr)
                resolvedType = NewResolvedType(o1.type)
                resolvedType.address = addr
                pO.push(resolvedType)
                return
        }
        o2 = pO.pop()
        resultType = semantcCubeLookup(o1.type, o2.type, oper)
        if resultType == Undefined {
                Error("unsupported operation")
```

```
                return
        }
        addr = GetAddressMgr(resultType).getNext()
        quads.push(oper, o1.address, o2.address, addr)
        resolvedType =  NewResolvedType(resultType)
        resolvedType.address = address
        pO.push(resovledType)
}
```

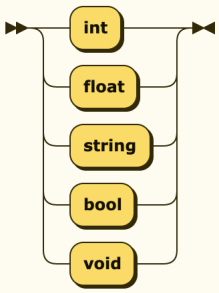


1. POper.push(operator)
2. if !POper.empty && (pOper.top == + || pOper.top == -) {
   generateQuadsForExpression()
   }

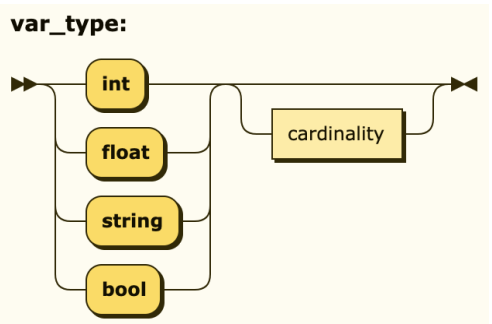


1. POper.push(UnaryMinus)
2. POper.push(Other) // fake bottom
3. POper.pop() // fake bottom
4. if isVariable(var_cons) {
   resolvedType = getResolvedType(var_cons.id)
   if resolvedType == null {
   Error(“Variable not defined”)
   return
   }
   pO.push(resolvedType)
   } else {
   resolvedType = NewResolvedType(var_cons.type)
   pO.push(resolvedType)
   }
5. if !pOper.empty && (pOper.top == ‘/’ || pOper.top == ‘%’ || pOper.top == ‘*’ || pOper.top == unaryMinus) {
   generateQuadsForExpression()
   }

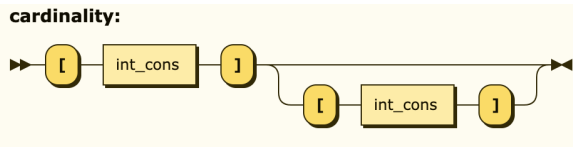
atomic_type:



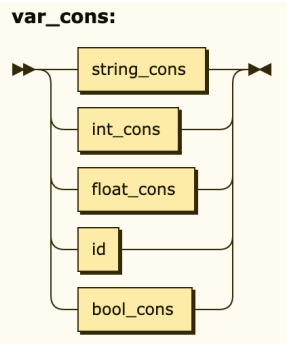
var_type:



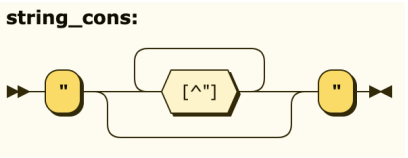
cardinality:



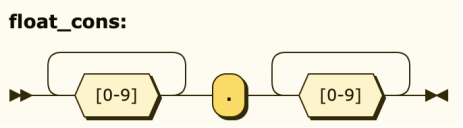
var_cons:



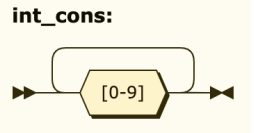
string_cons:

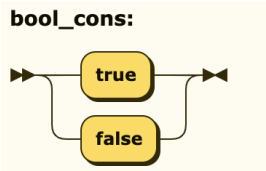


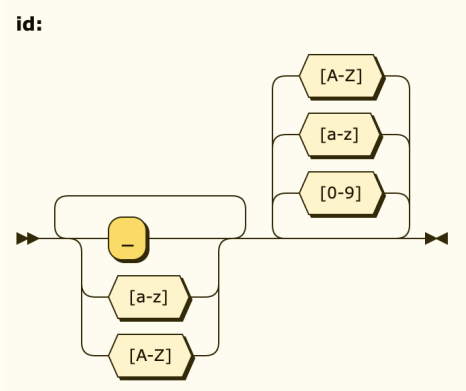
float_cons:



int_cons:







Semantical Considerations

The resulting types when using operators on different types are listed below:

Type 1	Type 2	Operator	Resulting type
int	int	Sum	int
int	float	Sum	float
float	int	Sum	float
float	float	Sum	float
int	int	Subtraction	int
int	float	Subtraction	float
float	int	Subtraction	float
float	float	Subtraction	float
int	int	Multiplication	int
int	float	Multiplication	float
float	int	Multiplication	float
float	float	Multiplication	float
int	int	Division	int

int	float	Division	float
float	int	Division	float
float	float	Division	float
int	int	Modulus	int
int	_	UnaryMinus	int
float	_	UnaryMinus	float
int	int	Eq	Bool
int	int	Ne	Bool
int	int	Gt	Bool
int	int	Lt	Bool
int	int	Ge	Bool
int	int	Le	Bool
float	float	Eq	Bool
float	float	Ne	Bool
float	float	Gt	Bool
float	float	Lt	Bool
float	float	Ge	Bool
float	float	Le	Bool
Bool	Bool	And	Bool
Bool	Bool	Or	Bool
Bool	Bool	Eq	Bool
Bool	Bool	Ne	Bool

Any type pair not listed in the table above will have *Undefined* as its result type, causing a semantic error.

Likewise, the special functions defined in the language can be used only with the following types:

- Sum: int or float matrices or arrays. Result type: same as input type.
- Prod: int or float matrices or arrays. Result type: same as input type.
- Min: int or float matrices or arrays. Result type: same as input type.
- Max: int or float matrices or arrays. Result type: same as input type.
- Avg: int or float matrices or arrays. Result type: float.

- SMode: int or float matrices or arrays. Result type: same as input type.
- Median: int or float matrices or arrays. Result type: float.
- Plot: int or float matrices or arrays. Result type: void.
- Sin: int or float atomic types, matrices or arrays. Result type: same as input type.
- Cos: int or float atomic types, matrices or arrays. Result type: same as input type.
- Tan: int or float atomic types, matrices or arrays. Result type: same as input type.
- Sqrt: int or float atomic types, matrices or arrays. Result type: same as input type.
- Abs: int or float atomic types, matrices or arrays. Result type: same as input type.
- Floor: float atomic types, matrices or arrays. Result type: int.
- Ceil: float atomic types, matrices or arrays. Result type: int.
- Not: boolean atomic types, matrices or arrays. Result type: boolean.
- read: any atomic type, matrix or array. Result type: void.
- print: any atomic type, matrix or array. Result type: void.

Calling any of the functions with unsupported types will cause a semantic error.

It is worth noting that aggregation functions (such as *avg*, *median*, *sum*, etc.) will return a flattened output (an atomic type case in case of receiving an array and an array in case of receiving a matrix).

Memory Management

The main data structure used for compilation is called *PCtx* (short for program context) and contains the following structures: *scopes*, *VM*, *functions*, *funcDir*, *paramTable*, *addrTable*, *varTable*, *consTable*, *semanticErrors*, *pO*, *pOper*, *jumps*, *condJumps*, *printQuads*, *fileName*.

Each attribute in *PCtx* as well as their subattributes are described below.

Scopes

This is a string stack used in order to determine the current program scope. This is useful for determining whether to use a local or global address manager as well as whether to look up a variable outside of the current scope (in case of being inside a function)

main	func1	func2

A scope is pushed when entering a function definition and popped when exiting a function's body.

VM

This struct, which is actually a *virtual* virtual machine, deals with managing the state of all virtual addresses as well as containing the quadruples that will be used during execution

globalIntAddressMgr, localIntAddressMgr,	AddressManager

tempIntAddressMgr, constIntAddressMgr, globalFloatAddressMgr, localFloatAddressMgr, tempFloatAddressMgr, constFloatAddressMgr, globalStringAddressMgr, localStringAddressMgr, tempStringAddressMgr, constStringAddressMgr, globalBoolAddressMgr, localBoolAddressMgr, tempBoolAddressMgr, constBoolAddressMgr, globalRefAddressMgr, localRefAddressMgr	
quads	[]Quad

AddressManager
This struct manages virtual addresses by incrementing the current virtual address as well as resetting it to its base value. It can also calculate the size to figure out how much space to allocate during execution

first	5000
last	6000
curr	5003
size	3

Quad
A quad is a collection of five int values: an *Op* (operation code), op1, op2, destination (which are usually all addresses, and an aux space (only used in special function quads)

Op	Sum // enum with value 0
op1	5000
op2	5001
destination	5002

Functions
This is a string array containing the names of all defined functions in a given program. The need for this structure arose because certain quads such as Era and GoSub actually expect a function name which will be then used as a key to look up the function directory. So instead of adding a string attribute to the *Quad* structure, which would allocate a string to all quads (which don't actually need it), the quad for Era and GoSub stores an int that is

represents the index of the function name in this functions array, and that function name is then used to look up the function directory. This array is also needed during execution.

func1	func2	func3	main

FuncDir
This is a hashmap with a key string (representing a function name) and a *FuncData* value, which holds relevant information about the function. An example *FuncData is shown below*
FuncData

pType	primitive type enum (int) that can be float, string, int or bool
Address	the address where the return value will be stored (int)
Params	The number of params defined in the function (int)
LocalRefs	Number of local references used (int)
LocalInts	Number of local ints defined (int)
LocalFloats	Number of local floats defined (int)
LocalBools	Number of local bools defined (int)
LocalStrings	Number of local strings defined (int)
TempRefs	Number of temp references used (int)
LocalInts	Number of temp ints defined (int)
TempFloats	Number of temp floats defined (int)
TempBools	Number of temp bools defined (int)
TempStrings	Number of temp strings defined (int)
FuncStart	The quad where the function body starts (int)
Idx	The function's index in the *functions* array (int)

FuncDir is also used during execution for resource allocation.

ParamTable
This is a hashmap which maps a string (function name) to an an array which holds the types of each parameter.
Example ParamTable value

Int	Float	Int

AddrTable
This structure is solely used for debugging purposes but has no effect on compilation or execution whatsoever. It is a hashmap mapping an int (virtual address) to a string (variable name). This is very useful during development as it allows us to see what variable name is stored in each address when printing quaduples. An example entry is shown below.

5003	result

VarTable
This structure is a hasmap mapping a string (function name) to a hashmap that maps a string (variable name) to an RType (short for resolvedType) structure. A new entry is added to the VarTable when entering a function and is deleted when exiting a function's body. A new entry to a VarTable entry is added when defining or looking up a variable. An example entry is shown below. The left column represents a function name, the left column in the right column represents a variable name and the right column in the right column represents an RType.

main	<table><tr><td>x</td><td>RType</td></tr></table>

Example RType
0 in firstDim indicates an atomic type

address	5001
firstDim	0
secondDim	0
pType	Int // enum with value 0

ConsTable
This is a hashmap mapping a string's representation of a value of any type to a virtual address. This is used during execution to load the initial constant values.

“true”	7001
“3.14”	6001
“100”	5001

SemanticErrors
This is a string array that holds any semantic errors that may arise during compilation.

pO
This is the operand stack and is an RType array.

pType	Float	pType	Float	pType	Bool
Address	6005	Address	6006	Address	7005
FirstDim	3	FirstDim	0	FirstDim	3
Secondim	0	Secondim	0	Secondim	0

pOper
This is the operator stack and is an int array.

0 // enum is Sum	1 // enum is subtraction	0	1

jumps
This is the jump stack and is an int array

3	10	15

condJumps
This is the conditional jumps matrix and is an int matrix. This structure was used to support nested if statements, because since this language supports else if statements, when an else statement is encountered, there is no way of knowing which of the jumps in the jump stack actually below to the current level of nesting, so this matrix effectively isolates jumps to their respective level of nesting. An array is added to the matrix when entering a conditional statement and is removed when exiting it.

PrintQuads
This is a boolean attribute that serves as a flag. When compiling, the user can choose whether to print the quads at the end of the compilation process, so this value determines if the quads should be printer

FIleName
This attribute is a string containing the name of the source program that was compiled. It's used to generate an intermediate code file with the same name but with ".gob" extension (this is the extension used for binary go files).

PCtx Definition

```
type PCtx struct {
 scopes          []string
 functions       []string
 funcDir         map[string]FuncData
 paramTable      map[string][]RType
 addrTable       map[int]string
 varTable        map[string]map[string]RType
 consTable       map[string]RType
 semanticErrors  []string
 pO              []RType
 pOper           []int
 vm              VM
 jumps           []int
 condJumps       [][]int
 printQuads      bool
 Filename        string
}
```

Execution

Development Tools

The virtual machine was developed in Go version *1.20.1 darwin/amd64*. The following external Go packages were used: *asciigraph* for plotting.

Memory Management

ObjCode

Upon finishing compilation, an *ObjCode* struct is generated. This structure contains all of the necessary information for execution. This structure is stored in .gob format (binary Go) and this is the file that can then be executed. This structure stores the function directory, constant table, functions array, quadruples, as well as the size needed for each global, constant and temporary type. The definition of the struct is provided below.

```
type ObjCode struct {
 FuncDir         map[string]FuncData
 ConsTable       map[string]RType
 Functions       []string
 Quads           []Quad
 ConstIntSize    int
 ConstFloatSize  int
 ConstBoolSize   int
```

```
 ConstStringSize int
 TempIntSize     int
 TempFloatSize   int
 TempBoolSize    int
 TempStringSize  int
 IntSize         int
 FloatSize       int
 BoolSize        int
 StringSize      int
 GlobalRefSize   int
}
```

ECtx
The file storing the ObjCode structure is then read by the execution program and creates the virtual machine, which is stored in the *ECtx* structure (short for execution context). This structure contains the following attributes: IP, quads, FuncDir, ConsTable, ConstMemory, TempMemory, GobalMemory, StackSegment, Jumps, Functions, Errors. Each attribute and its subattributes are described below.

IP
This is an int containing the index of the current quad (short for instruction pointer).

Quads
An array of quadruples with the same definition as in compilation

FuncDir
Same structure used in compilation

ConsTable
Same structure used in compilation

Functions
Same structure used in compilation

Errors
String array used to store execution errors.

Jumps
An array of ints used to store the quad number the execution context has to return to after a function call.

ConstMemory
A MemorySegment structure to store constant memory. The definition of MemorySegment is provided below.

TempMemory

A MemorySegment structure to store temporary memory. The definition of MemorySegment is provided below.

GlobalMemory
A MemorySegment structure to store global memory. The definition of MemorySegment is provided below.

MemorySegment
This structure manages memory by storing an array for each primitive type supported by Bistat, as well as an additional array for refs. The size of each array is passed as a constructor argument, as well as the base virtual address for each memory type. Access to a specific virtual address is performed by subtracting the base virtual address from a particular virtual address. The constructor for this structure is provided below.

```
func NewMemorySegment(fSize int, iSize int, sSize int, bSize int, refSize int,
baseFloatAddr int, baseIntAddr int, baseStringAddr int, baseBoolAddr int,
baseRefAddr int) *MemorySegment {
 return &MemorySegment{
   floats:         make([]float64, fSize),
   ints:           make([]int64, iSize),
   strings:        make([]string, sSize),
   bools:          make([]bool, bSize),
   refs:           make([]int, refSize),
   baseFloatAddr:  baseFloatAddr,
   baseIntAddr:    baseIntAddr,
   baseBoolAddr:   baseBoolAddr,
   baseStringAddr: baseStringAddr,
   baseRefAddr:    baseRefAddr,
 }
}
```

floats	3.14 | 5.4 | 13.4
ints	23 | 99
strings	“ovejota” | “hello world”
bools	true

refs	<table><tr><td>5002</td><td>7002</td></tr></table>
baseFloatAddr	6000
baseIntAddr	5000
baseStringAddr	7000
baseRefAddr	13000
baseBoolAddr	4000

StackSegment (attribute)
The StackSegment attribute is an array of StackSegment structs. A StackSegment struct is pushed when encountering an Era instruction and popped when encountering a Return or EndFunc instruction.

StackSegment (struct)
The StackSegment struct consists of two MemorySegment structs: one for temporary memory and another for local memory.

localMemory	MemorySegment (defined above)
tempMemory	MemorySegment (defined above)

The definition and constructor of the ECtx structure is provided below

```
type ECtx struct {
 IP           int
 Quads        []src.Quad
 FuncDir      map[string]src.FuncData
 ConsTable    map[string]src.RType
 ConstMemory  *MemorySegment
 TempMemory   *MemorySegment
 GlobalMemory *MemorySegment
 StackSegment []StackSegment
 Jumps        []int
 Functions    []string
 Errors       []string
}

func NewEctx(objCode src.ObjCode) ECtx {
 return ECtx{
   IP:           0,
```

```
    Quads:         objCode.Quads,
    FuncDir:       objCode.FuncDir,
    ConsTable:     objCode.ConsTable,
    ConstMemory:   NewMemorySegment(objCode.ConstFloatSize, objCode.ConstIntSize,
objCode.ConstStringSize, objCode.ConstBoolSize, 0, src.CONST_FLOAT_START,
src.CONST_INT_START, src.CONST_STRING_START, src.CONST_BOOL_START, 0),
    TempMemory:    NewMemorySegment(objCode.TempFloatSize, objCode.TempIntSize,
objCode.TempStringSize, objCode.TempBoolSize, 0, src.TEMP_FLOAT_START,
src.TEMP_INT_START, src.TEMP_STRING_START, src.TEMP_BOOL_START, 0),
    GlobalMemory: NewMemorySegment(objCode.FloatSize, objCode.IntSize,
objCode.StringSize, objCode.BoolSize, objCode.GlobalRefSize,
src.GLOBAL_FLOAT_START, src.GLOBAL_INT_START, src.GLOBAL_STRING_START,
src.GLOBAL_BOOL_START, src.GLOBAL_REF_START),
    StackSegment: make([]StackSegment, 0),
    Errors:        make([]string, 0),
    Functions:     objCode.Functions,
  }
}
```

Functionality Tests

Basic Functionality

Iterative and Recursive Factorial
The following Bistat program defines functions for calculating a number's factorial in both recursive and iterative fashion.

```
Program Factorial;

var int x;

func recursive_fac (var int x;): int {
  if (x == 1) {
    return x;
  }
  return x * recursive_fac(x - 1);
}

func iterative_fac(var int x;): int
  var int product;
  var int i;
{
  product = 1;
  i = 1;
  while (i <= x) {
    product = product * i;
    i = i + 1;
  }
  return product;
}

main() {
  x = recursive_fac(7);
  print(x);
  x = iterative_fac(7);
  print(x);
}
```

This outputs the following quads

0 Goto -1 #-1 -1 #-1 23 #23
1 Eq x #40000 1 #120000 110000 #110000
2 GotoF 110000 #110000 x #0 4 #4
3 Return recursive_fac #1 -1 #-1 x #40000
4 Subtraction x #40000 1 #120000 80000 #80000
5 Era recursive_fac #1 -1 #-1 -1 #-1
6 Param x #40000 -1 #-1 80000 #80000
7 GoSub recursive_fac #1 -1 #-1 -1 #-1
8 Assign 80001 #80001 -1 #-1 recursive_fac #1
9 Multiplication x #40000 80001 #80001 80002 #80002
10 Return recursive_fac #1 -1 #-1 80002 #80002
11 EndFunc -1 #-1 -1 #-1 -1 #-1
12 Assign product #40001 -1 #-1 1 #120000
13 Assign i #40002 -1 #-1 1 #120000
14 Le i #40002 x #40000 110000 #110000
15 GotoF 110000 #110000 -1 #-1 21 #21
16 Multiplication product #40001 i #40002 80000 #80000
17 Assign product #40001 -1 #-1 80000 #80000
18 Sum i #40002 1 #120000 80001 #80001
19 Assign i #40002 -1 #-1 80001 #80001
20 Goto -1 #-1 -1 #-1 14 #14
21 Return iterative_fac #2 -1 #-1 product #40001
22 EndFunc -1 #-1 -1 #-1 -1 #-1
23 Era recursive_fac #1 -1 #-1 -1 #-1
24 Param x #40000 -1 #-1 7 #120001
25 GoSub recursive_fac #1 -1 #-1 -1 #-1
26 Assign 80000 #80000 -1 #-1 recursive_fac #1

```
27 Assign x #0 -1 #-1 80000 #80000
28 Print x #0 -1 #-1 -1 #-1
29 PrintN -1 #-1 -1 #-1 -1 #-1
30 Era iterative_fac #2 -1 #-1 -1 #-1
31 Param x #40000 -1 #-1 7 #120001
32 GoSub iterative_fac #2 -1 #-1 -1 #-1
33 Assign 80001 #80001 -1 #-1 iterative_fac #2
34 Assign x #0 -1 #-1 80001 #80001
35 Print x #0 -1 #-1 -1 #-1
36 PrintN -1 #-1 -1 #-1 -1 #-1
37 End -1 #-1 -1 #-1 -1 #-1
```

Upon executing the binary file we get the following output

```
→  bistat git:(master) go run execute_bistat/*.go factorial.bs.gob
5040
5040
```

Iterative and Recursive Fibonacci

This program calculates the fibonacci number both recursively and iteratively

```
1   Program Fibonacci;
2
3   var int x;
4   var int[5] nums;
5
6   func recursive_fib(var int n;): int {
7     if (n <= 1) {
8       return n;
9     }
10    return recursive_fib(n - 1) + recursive_fib(n - 2);
11  }
12
13  func iterative_fib(var int n;): int
14  var int pprev;
15  var int prev;
16  var int curr;
17  var int i;
18  {
19    prev = 0;
20    curr = 1;
21    i = 1;
22    while (i < n) {
23      pprev = prev;
24      prev = curr;
25      curr = pprev + prev;
26      i = i + 1;
27    }
28    return curr;
29  }
30
31  main() {
32    x = recursive_fib(9);
33    print(x);
34    x = iterative_fib(9);
35    print(x);
36  }
```

This outputs the following quads

```
0 Goto -1 #-1 -1 #-1 31 #31
```

```
1 Le n #40000 1 #120000 110000 #110000
2 GotoF 110000 #110000 x #0 4 #4
3 Return recursive_fib #6 -1 #-1 n #40000
4 Subtraction n #40000 1 #120000 80000 #80000
5 Era nums #1 -1 #-1 -1 #-1
6 Param n #40000 -1 #-1 80000 #80000
7 GoSub nums #1 -1 #-1 -1 #-1
8 Assign 80001 #80001 -1 #-1 recursive_fib #6
9 Subtraction n #40000 2 #120001 80002 #80002
10 Era nums #1 -1 #-1 -1 #-1
11 Param n #40000 -1 #-1 80002 #80002
12 GoSub nums #1 -1 #-1 -1 #-1
13 Assign 80003 #80003 -1 #-1 recursive_fib #6
14 Sum 80001 #80001 80003 #80003 80004 #80004
15 Return recursive_fib #6 -1 #-1 80004 #80004
16 EndFunc -1 #-1 -1 #-1 -1 #-1
17 Assign prev #40002 -1 #-1 0 #120002
18 Assign curr #40003 -1 #-1 1 #120000
19 Assign i #40004 -1 #-1 1 #120000
20 Lt i #40004 n #40000 110000 #110000
21 GotoF 110000 #110000 -1 #-1 29 #29
22 Assign pprev #40001 -1 #-1 prev #40002
23 Assign prev #40002 -1 #-1 curr #40003
24 Sum pprev #40001 prev #40002 80000 #80000
25 Assign curr #40003 -1 #-1 80000 #80000
26 Sum i #40004 1 #120000 80001 #80001
27 Assign i #40004 -1 #-1 80001 #80001
28 Goto -1 #-1 -1 #-1 20 #20
29 Return iterative_fib #7 -1 #-1 curr #40003
30 EndFunc -1 #-1 -1 #-1 -1 #-1
31 Era nums #1 -1 #-1 -1 #-1
32 Param n #40000 -1 #-1 9 #120003
33 GoSub nums #1 -1 #-1 -1 #-1
34 Assign 80000 #80000 -1 #-1 recursive_fib #6
35 Assign x #0 -1 #-1 80000 #80000
36 Print x #0 -1 #-1 -1 #-1
37 PrintN -1 #-1 -1 #-1 -1 #-1
38 Era 2 #2 -1 #-1 -1 #-1
39 Param n #40000 -1 #-1 9 #120003
40 GoSub 2 #2 -1 #-1 -1 #-1
41 Assign 80001 #80001 -1 #-1 iterative_fib #7
42 Assign x #0 -1 #-1 80001 #80001
43 Print x #0 -1 #-1 -1 #-1
44 PrintN -1 #-1 -1 #-1 -1 #-1
45 End -1 #-1 -1 #-1 -1 #-1
```

And outputs the following upon execution

```
● → bistat git:(master) ✗ go run execute_bistat/*.go fibonacci.bs.gob
34
34
```

Sort and Find

The following program defines a function for sorting an array and another function for returning the index that stores an element in an array, or -1 if the element is not found.

```
1   Program SortAndFind;
2
3   var int x;
4   var int[5] nums;
5
6
7   func bubbleSort(): void
8     var int i;
9     var int j;
10    var int tmp;
11    var int[4] arr;
12   {
13    i = 0;
14    j = 0;
15    arr = [4, 6, 1, 2];
16    print(arr);
17    while (i < 3) {
18      j = 0;
19      while (j < 3 - i) {
20        if (arr{j} > arr{j+1}) {
21          tmp = arr{j};
22          arr{j} = arr{j+1};
23          arr{j+1} = tmp;
24        }
25        j = j + 1;
26      }
27      i = i + 1;
28    }
29    print(arr);
30  }
31
32  func find(var int num;): int
33    var int i;
34   {
35    i = 0;
36    while (i < 5) {
37      if (nums{i} == num) {
38        return i;
39      }
40      i = i + 1;
41    }
42    return -1;
43  }
44
45  main() {
46    bubbleSort();
47    nums = [6, 87, 35, 9, 3];
48    x = find(11);
49    print(x);
50    x = find(87);
51    print(x);
52  }
```

This outputs the following quads

0 Goto -1 #-1 -1 #-1 86 #86
1 Assign num #40000 -1 #-1 0 #120000
2 Assign i #40001 -1 #-1 0 #120000
3 RefSum 40003 #120005 3 #120006 160003 #160003
4 Assign 160003 #160003 -1 #-1 2 #120004
5 RefSum 40003 #120005 2 #120004 160004 #160004
6 Assign 160004 #160004 -1 #-1 1 #120003

```
7 RefSum 40003 #120005 1 #120003 160005 #160005
8 Assign 160005 #160005 -1 #-1 6 #120002
9 RefSum 40003 #120005 0 #120000 160006 #160006
10 Assign 160006 #160006 -1 #-1 4 #120001
11 Print [ #140001 -1 #-1 -1 #-1
12 RefSum 40003 #120005 0 #120000 160007 #160007
13 Print 160007 #160007 -1 #-1 -1 #-1
14 Print   #140000 -1 #-1 -1 #-1
15 RefSum 40003 #120005 1 #120003 160008 #160008
16 Print 160008 #160008 -1 #-1 -1 #-1
17 Print   #140000 -1 #-1 -1 #-1
18 RefSum 40003 #120005 2 #120004 160009 #160009
19 Print 160009 #160009 -1 #-1 -1 #-1
20 Print   #140000 -1 #-1 -1 #-1
21 RefSum 40003 #120005 3 #120006 160010 #160010
22 Print 160010 #160010 -1 #-1 -1 #-1
23 Print ] #140002 -1 #-1 -1 #-1
24 PrintN -1 #-1 -1 #-1 -1 #-1
25 Lt num #40000 3 #120006 110000 #110000
26 GotoF 110000 #110000 -1 #-1 57 #57
27 Assign i #40001 -1 #-1 0 #120000
28 Subtraction 3 #120006 num #40000 80000 #80000
29 Lt i #40001 80000 #80000 110001 #110001
30 GotoF 110001 #110001 -1 #-1 54 #54
31 Verify i #40001 arr #40003 4 #4
32 RefSum 40003 #120005 i #40001 160011 #160011
33 Sum i #40001 1 #120003 80001 #80001
34 Verify 80001 #80001 arr #40003 4 #4
35 RefSum 40003 #120005 80001 #80001 160012 #160012
36 Gt 160011 #160011 160012 #160012 110002 #110002
37 GotoF 110002 #110002 x #0 51 #51
38 Verify i #40001 arr #40003 4 #4
39 RefSum 40003 #120005 i #40001 160013 #160013
40 Assign tmp #40002 -1 #-1 160013 #160013
41 Sum i #40001 1 #120003 80002 #80002
42 Verify 80002 #80002 arr #40003 4 #4
43 RefSum 40003 #120005 80002 #80002 160014 #160014
44 Verify i #40001 arr #40003 4 #4
45 RefSum 40003 #120005 i #40001 160015 #160015
46 Assign 160015 #160015 -1 #-1 160014 #160014
47 Sum i #40001 1 #120003 80003 #80003
48 Verify 80003 #80003 arr #40003 4 #4
49 RefSum 40003 #120005 80003 #80003 160016 #160016
50 Assign 160016 #160016 -1 #-1 tmp #40002
51 Sum i #40001 1 #120003 80004 #80004
52 Assign i #40001 -1 #-1 80004 #80004
53 Goto -1 #-1 -1 #-1 28 #28
54 Sum num #40000 1 #120003 80005 #80005
55 Assign num #40000 -1 #-1 80005 #80005
56 Goto -1 #-1 -1 #-1 25 #25
57 Print [ #140001 -1 #-1 -1 #-1
58 RefSum 40003 #120005 0 #120000 160017 #160017
59 Print 160017 #160017 -1 #-1 -1 #-1
60 Print   #140000 -1 #-1 -1 #-1
61 RefSum 40003 #120005 1 #120003 160018 #160018
62 Print 160018 #160018 -1 #-1 -1 #-1
63 Print   #140000 -1 #-1 -1 #-1
64 RefSum 40003 #120005 2 #120004 160019 #160019
```

```
65 Print 160019 #160019 -1 #-1 -1 #-1
66 Print   #140000 -1 #-1 -1 #-1
67 RefSum 40003 #120005 3 #120006 160020 #160020
68 Print 160020 #160020 -1 #-1 -1 #-1
69 Print ] #140002 -1 #-1 -1 #-1
70 PrintN -1 #-1 -1 #-1 -1 #-1
71 EndFunc -1 #-1 -1 #-1 -1 #-1
72 Assign i #40001 -1 #-1 0 #120000
73 Lt i #40001 5 #120007 110000 #110000
74 GotoF 110000 #110000 -1 #-1 83 #83
75 Verify i #40001 nums #1 5 #5
76 RefSum 1 #120003 i #40001 160003 #160003
77 Eq 160003 #160003 num #40000 110001 #110001
78 GotoF 110001 #110001 x #0 80 #80
79 Return find #6 -1 #-1 i #40001
80 Sum i #40001 1 #120003 80000 #80000
81 Assign i #40001 -1 #-1 80000 #80000
82 Goto -1 #-1 -1 #-1 73 #73
83 UnaryMinus 1 #120003 -1 #-1 80001 #80001
84 Return find #6 -1 #-1 80001 #80001
85 EndFunc -1 #-1 -1 #-1 -1 #-1
86 Era nums #1 -1 #-1 -1 #-1
87 GoSub nums #1 -1 #-1 -1 #-1
88 RefSum 1 #120003 4 #120001 150003 #150003
89 Assign 150003 #150003 -1 #-1 3 #120006
90 RefSum 1 #120003 3 #120006 150004 #150004
91 Assign 150004 #150004 -1 #-1 9 #120010
92 RefSum 1 #120003 2 #120004 150005 #150005
93 Assign 150005 #150005 -1 #-1 35 #120009
94 RefSum 1 #120003 1 #120003 150006 #150006
95 Assign 150006 #150006 -1 #-1 87 #120008
96 RefSum 1 #120003 0 #120000 150007 #150007
97 Assign 150007 #150007 -1 #-1 6 #120002
98 Era 2 #2 -1 #-1 -1 #-1
99 Param num #40000 -1 #-1 11 #120011
100 GoSub 2 #2 -1 #-1 -1 #-1
101 Assign 80000 #80000 -1 #-1 find #6
102 Assign x #0 -1 #-1 80000 #80000
103 Print x #0 -1 #-1 -1 #-1
104 PrintN -1 #-1 -1 #-1 -1 #-1
105 Era 2 #2 -1 #-1 -1 #-1
106 Param num #40000 -1 #-1 87 #120008
107 GoSub 2 #2 -1 #-1 -1 #-1
108 Assign 80001 #80001 -1 #-1 find #6
109 Assign x #0 -1 #-1 80001 #80001
110 Print x #0 -1 #-1 -1 #-1
111 PrintN -1 #-1 -1 #-1 -1 #-1
112 End -1 #-1 -1 #-1 -1 #-1
```

And produces the following output in execution

```
● ➜  bistat git:(master) ✗ go run execute_bistat/*.go sortAndFind.bs.gob
  [4 6 1 2]
  [1 2 4 6]
  -1
  1
```

Matrix Multiplication

```
1   Program MatMul;
2
3   func matmul(): void
4     var int[3][3] x;
5     var int[3][4] y;
6     var int[3][4] z;
7     var int i;
8     var int j;
9     var int k;
10  {
11    i = 0;
12    j = 0;
13    k = 0;
14    x = [
15      [12, 7, 3],
16      [4, 5, 6],
17      [7, 8, 9]
18    ];
19    y = [
20      [5, 8, 1, 2],
21      [6, 7, 3, 0],
22      [4, 5, 9, 1]
23    ];
24    while (i < 3) {
25      j = 0;
26      while (j < 4) {
27        k = 0;
28        while (k < 3) {
29          z{i}{j} = z{i}{j} + x{i}{k} * y{k}{j};
30          k = k + 1;
31        }
32        j = j + 1;
33      }
34      i = i + 1;
35    }
36    print(z);
37  }
38
39  main() {
40    matmul();
41  }
```

Quads

0 Goto -1 #-1 -1 #-1 86 #86
1 Assign num #40000 -1 #-1 0 #120000
2 Assign i #40001 -1 #-1 0 #120000
3 RefSum 40003 #120005 3 #120006 160003 #160003
4 Assign 160003 #160003 -1 #-1 2 #120004
5 RefSum 40003 #120005 2 #120004 160004 #160004
6 Assign 160004 #160004 -1 #-1 1 #120003
7 RefSum 40003 #120005 1 #120003 160005 #160005
8 Assign 160005 #160005 -1 #-1 6 #120002
9 RefSum 40003 #120005 0 #120000 160006 #160006
10 Assign 160006 #160006 -1 #-1 4 #120001
11 Print [#140001 -1 #-1 -1 #-1
12 RefSum 40003 #120005 0 #120000 160007 #160007
13 Print 160007 #160007 -1 #-1 -1 #-1
14 Print #140000 -1 #-1 -1 #-1
15 RefSum 40003 #120005 1 #120003 160008 #160008
16 Print 160008 #160008 -1 #-1 -1 #-1
17 Print #140000 -1 #-1 -1 #-1
18 RefSum 40003 #120005 2 #120004 160009 #160009
19 Print 160009 #160009 -1 #-1 -1 #-1
20 Print #140000 -1 #-1 -1 #-1
21 RefSum 40003 #120005 3 #120006 160010 #160010

```
22 Print 160010 #160010 -1 #-1 -1 #-1
23 Print ] #140002 -1 #-1 -1 #-1
24 PrintN -1 #-1 -1 #-1 -1 #-1
25 Lt num #40000 3 #120006 110000 #110000
26 GotoF 110000 #110000 -1 #-1 57 #57
27 Assign i #40001 -1 #-1 0 #120000
28 Subtraction 3 #120006 num #40000 80000 #80000
29 Lt i #40001 80000 #80000 110001 #110001
30 GotoF 110001 #110001 -1 #-1 54 #54
31 Verify i #40001 arr #40003 4 #4
32 RefSum 40003 #120005 i #40001 160011 #160011
33 Sum i #40001 1 #120003 80001 #80001
34 Verify 80001 #80001 arr #40003 4 #4
35 RefSum 40003 #120005 80001 #80001 160012 #160012
36 Gt 160011 #160011 160012 #160012 110002 #110002
37 GotoF 110002 #110002 x #0 51 #51
38 Verify i #40001 arr #40003 4 #4
39 RefSum 40003 #120005 i #40001 160013 #160013
40 Assign tmp #40002 -1 #-1 160013 #160013
41 Sum i #40001 1 #120003 80002 #80002
42 Verify 80002 #80002 arr #40003 4 #4
43 RefSum 40003 #120005 80002 #80002 160014 #160014
44 Verify i #40001 arr #40003 4 #4
45 RefSum 40003 #120005 i #40001 160015 #160015
46 Assign 160015 #160015 -1 #-1 160014 #160014
47 Sum i #40001 1 #120003 80003 #80003
48 Verify 80003 #80003 arr #40003 4 #4
49 RefSum 40003 #120005 80003 #80003 160016 #160016
50 Assign 160016 #160016 -1 #-1 tmp #40002
51 Sum i #40001 1 #120003 80004 #80004
52 Assign i #40001 -1 #-1 80004 #80004
53 Goto -1 #-1 -1 #-1 28 #28
54 Sum num #40000 1 #120003 80005 #80005
55 Assign num #40000 -1 #-1 80005 #80005
56 Goto -1 #-1 -1 #-1 25 #25
57 Print [ #140001 -1 #-1 -1 #-1
58 RefSum 40003 #120005 0 #120000 160017 #160017
59 Print 160017 #160017 -1 #-1 -1 #-1
60 Print   #140000 -1 #-1 -1 #-1
61 RefSum 40003 #120005 1 #120003 160018 #160018
62 Print 160018 #160018 -1 #-1 -1 #-1
63 Print   #140000 -1 #-1 -1 #-1
64 RefSum 40003 #120005 2 #120004 160019 #160019
65 Print 160019 #160019 -1 #-1 -1 #-1
66 Print   #140000 -1 #-1 -1 #-1
67 RefSum 40003 #120005 3 #120006 160020 #160020
68 Print 160020 #160020 -1 #-1 -1 #-1
69 Print ] #140002 -1 #-1 -1 #-1
70 PrintN -1 #-1 -1 #-1 -1 #-1
71 EndFunc -1 #-1 -1 #-1 -1 #-1
72 Assign i #40001 -1 #-1 0 #120000
73 Lt i #40001 5 #120007 110000 #110000
74 GotoF 110000 #110000 -1 #-1 83 #83
75 Verify i #40001 nums #1 5 #5
76 RefSum 1 #120003 i #40001 160003 #160003
77 Eq 160003 #160003 num #40000 110001 #110001
78 GotoF 110001 #110001 x #0 80 #80
79 Return find #6 -1 #-1 i #40001
```

```
80 Sum i #40001 1 #120003 80000 #80000
81 Assign i #40001 -1 #-1 80000 #80000
82 Goto -1 #-1 -1 #-1 73 #73
83 UnaryMinus 1 #120003 -1 #-1 80001 #80001
84 Return find #6 -1 #-1 80001 #80001
85 EndFunc -1 #-1 -1 #-1 -1 #-1
86 Era nums #1 -1 #-1 -1 #-1
87 GoSub nums #1 -1 #-1 -1 #-1
88 RefSum 1 #120003 4 #120001 150003 #150003
89 Assign 150003 #150003 -1 #-1 3 #120006
90 RefSum 1 #120003 3 #120006 150004 #150004
91 Assign 150004 #150004 -1 #-1 9 #120010
92 RefSum 1 #120003 2 #120004 150005 #150005
93 Assign 150005 #150005 -1 #-1 35 #120009
94 RefSum 1 #120003 1 #120003 150006 #150006
95 Assign 150006 #150006 -1 #-1 87 #120008
96 RefSum 1 #120003 0 #120000 150007 #150007
97 Assign 150007 #150007 -1 #-1 6 #120002
98 Era 2 #2 -1 #-1 -1 #-1
99 Param num #40000 -1 #-1 11 #120011
100 GoSub 2 #2 -1 #-1 -1 #-1
101 Assign 80000 #80000 -1 #-1 find #6
102 Assign x #0 -1 #-1 80000 #80000
103 Print x #0 -1 #-1 -1 #-1
104 PrintN -1 #-1 -1 #-1 -1 #-1
105 Era 2 #2 -1 #-1 -1 #-1
106 Param num #40000 -1 #-1 87 #120008
107 GoSub 2 #2 -1 #-1 -1 #-1
108 Assign 80001 #80001 -1 #-1 find #6
109 Assign x #0 -1 #-1 80001 #80001
110 Print x #0 -1 #-1 -1 #-1
111 PrintN -1 #-1 -1 #-1 -1 #-1
112 End -1 #-1 -1 #-1 -1 #-1
```

Output

```
● → bistat git:(master) ✗ go run execute_bistat/*.go matmul.bs.gob
  [[114 160 60 27] [74 97 73 14] [119 157 112 23]]
```

Domain Specific Functionality

The following program tests some statistical functions as well as plotting.

```
Program Hello;

func special_functions1(): void
  var bool[4] bools;
  var int[4] nums;
  var float myf;
  var int x;
  {
    bools = [true, false, true, false];
    print(bools);
    print(not(bools));
    print(bools);
    nums = [-3, 4, -10, 777];
```

```
  print(nums);
  print(abs(nums));
  myf = 5.5;
  print(myf);
  x = floor(myf);
  print(x);
  myf = 5.7;
  print(myf);
  print(ceil(myf));
  x = 81;
  print(x);
  x = sqrt(x);
  print(x);
  print(cos(0.1));
  print(cos(-10.6));
  print(-2 + 2);
  print((2+2)>4);
}

func special_functions2(): void
  var int[4] nums;
  var int[4][3] mat;
  var int[2][3] mt;
  var int[4] ls;
  var float[5] fs;
  var float[2][3] fm;
  var float f;
  var int x;
  {
    nums = [67, 53, 64, 0];
    print(nums);
    x = sum(nums);
    print(x);
    x = prod(nums);
    print(x);
    x = max(nums);
    print(x);
    x = min(nums);
    print(x);
    mat = [
      [3, 5, 0],
      [5, 5, 6],
      [4, 7, 9],
      [8, 8, 1]
    ];
    mt = [
      [3, 5, 0],
      [5, 5, 6]
    ];
    plot(mt);
    print(avg(mat));
    print(sMode(mat));
    print("median");
    print(median(mat{0}));
    ls = sum(mat);
    print(ls);
    fs = [5.6, 6.7, 3.4, 9.9, 2.1];
    print(max(fs));
```

```
    f = avg(fs);
    print(f);
    fm = [
      [94.5, 3.14, 123.54],
      [4.5, 234.6, 34.5]
    ];
    print(prod(fm));
    print("hello");
    print(fm{1});
    print(avg(fm{1}));
    print(fm{0});
    print(avg(fm{0}));
    print(floor(fm));
    print(floor(fm{0}));
    print(avg(fm));
    plot(mat);
    plot(fs);
  }

main () {
  special_functions1();
  special_functions2();
}
```

Output

```
→  bistat git:(master) ✗ go run execute_bistat/*.go test_special_functions.bs.gob
[true false true false]
[false true false true]
[true false true false]
[-3 4 -10 777]
[3 4 10 777]
5.5
5
5.7
6
81
9
0.9950041652780257
-0.3853381907718297
0
false
[67 53 64 0]
184
0
67
0
 6.00 ┤  ╭╮
 5.00 ┤ ╭╯│
 4.00 ┤ │ │
 3.00 ┤╭╯ │
 2.00 ┤│  │
 1.00 ┤│  │
 0.00 ┤╰  ╰
[2.6666666666666665 5.333333333333333 6.666666666666667 5.666666666666667]
[3 5 4 8]
median
3
[8 16 20 17]
9.9
5.540000000000001
[36658.02420000001 36421.65]
hello
[4.5 234.6 34.5]
91.2
[94.5 3.14 123.54]
73.72666666666667
[[94 3 123] [4 234 34]]
[94 3 123]
[73.72666666666667 91.2]
 9.00 ┤  ╭
 8.00 ┤ ╭╯
 7.00 ┤ │
 6.00 ┤ │
 5.00 ┤ │
 4.00 ┤ │
 3.00 ┤ ╰╮
 2.00 ┤  │
 1.00 ┤  │
 0.00 ┤  ╰
 9.90 ┤   ╭╮
 8.79 ┤   ││
 7.67 ┤   ││
 6.56 ┤  ╭╯│
 5.44 ┤ ╭╯ │
 4.33 ┤ │  │
 3.21 ┤ │  │
 2.10 ┤ ╰  ╰
```

User Manual

Quick Reference

Running your first Bistat program

- Install Go (this project uses version 1.20.1)
- Clone repo
- In your terminal, go to the repository location
- Create a Bistat program inside the repository location
- Inside the repository location, run go get ./.. to install any required packages
- Inside the repository location run the following command
    - *go run compile_bistat.go <filename_with_extension>*
    - If you want the quads to be printed, append the command with 1
- Inside the repository location, run
    - *go run execute_bistat/*.go <filename_with_extension.gob>*
    - For example, if your source file is hello.bs, run the command with hello.bs.gob as the file name

Snippets

Declaring an array or matrix

```
var int[5] array;
var float[2][3] matrix;
```

Declaring a function

```
func myFunc(): void
        var int funcVar;
{
        funcVar = 1;
        print(funcVar);
}
```

Calculating an average

```
Program AvgDemo;

var int nums[5];
var float numsAvg;

main() {
        nums = [5, 3, 6, 3, 7];
        numsAvg = avg(nums);
        print(numsAvg);
}
```

Demo

Stats functionality
Input/Output