Lobster Programming Language User Guide:

***Overview***: Lobster is the name of the programming language that I have written. The language is syntactically similar to C++ and Java, and makes use of multi-threading to increase the speed of its programs. The language is an interpreted language, which is written in a custom bytecode that I have made. The Lobster Virtual Machine can read a file of this bytecode, and can produce the correct output of running the program.

***From Compilation to Running Program:***

To go from a user’s program written in Lobster (which has C++ like syntax) to a custom byte code file running on the Lobster virtual machine, there are several steps that need to occur. In order, the phases that a user’s program goes through are: preprocessing, lexical analysis, parsing, code generation, and running the code on the virtual machine.

***Preprocessing:*** The first phase of compilation is handled by the preprocessor, which removes lines which are comments (from // to the end of the line, and from /\* to \*/ for block comments). Furthermore, the preprocessor handles includes in files by combining all files appropriately into one big new file with the appropriate contents. For example, suppose we have the following 2 files:

1. Other.txt file

#include “Sample.txt”

//my comment

void main()

{

helloWorldFunc();

}

1. Sample.txt file

void helloWorldFunc()

{

print(“Hello World!\n”);

}

After the preprocessor is called on the file Other.txt, the resulting output will be sent to a new file which will look like this:

void helloWorldFunc()

{

print(“Hello World\n”);

}

void main()

{

helloWorldFunc():

}

Besides combining the files appropriately, the preprocessor directives #include “filename” are stripped from the resulting file as well. The preprocessor also creates a graph representation of the include files, and after every 100 includes encountered, the preprocessor will check to see if there is a cycle in the graph (ex. one file includes a second file, and the second file includes the first file). This would cause an endless loop, so if this happens, the program instead terminates with an error message.

The output of the preprocessor class is the combined output file, which is a temporary file of the format originalFileName + “.tempLobster” + randomStringOfNumbers. For example, after compiling “Other.txt,” the resulting output file could be stored in “Other.txt.tempLobster9458392.” This file will be deleted after compilation for the program is finished.

***Lexical Analysis:***

The lexical analyzer takes as input the file which is output by the preprocessor. The lexical analyzer class has variables which keep track of how far into the file it has read, so when a user calls the tokenize function, the next token is returned. the tokenize function takes as input a char[] to store the actual characters in the token and an int that represents the size of the character array. The tokenize function stores the characters of the token in this array, and then returns a tokenType, which is an enumerated type representing the type of a token (ex. FOR, INT\_LIT). The types of tokens in my language are:

**Literal Tokens**: INTEGER\_LITERAL | DOUBLE\_LITERAL | TRUE | FALSE | NULL | CHAR\_LITERAL | STRING\_LITERAL

**Assignment Token:** ASSIGNMENT\_OPERATOR

**Compound Assignment Tokens:** PLUS\_PLUS | MINUS\_MINUS | PLUS\_EQUALS | MINUS\_EQUALS | TIMES\_EQUALS | MOD\_EQUALS | DIVIDE\_EQUALS

**Binary Operator Tokens:** MOD | LOGICAL\_AND | LOGICAL\_OR | BITSHIFT\_LEFT | BITSHIFT\_RIGHT | EXPONENET | BITWISE\_AND | BITWISE\_OR | BITWISE\_XOR | BITWISE\_NOT | PLUS | MINUS | DIVIDE | MULTIPLY

**Boolean Operator Tokens:** NOT | LESS\_THAN | LESS\_THAN\_EQUALS | GREATER\_THAN\_EQUALS | GREATER\_THAN | EQUALS\_EQUALS | NOT\_EQUALS

**Unary Bitwise Token:** BITWISE\_NOT

**Identifier Token:** IDENTIFIER

**Type Tokens:** INT | BOOLEAN | CHAR | DOUBLE | VOID | STRING | QUEUE | LIST | ARRAY

**Print Token:** PRINT

**Read Input Tokens:** GET\_INT | GET\_CHAR | GET\_DOUBLE | GET\_WORD | GET\_LINE

**Input Test Token:** IS\_INPUT\_FAIL

**Object Related Tokens:** CLASS | NEW | THIS

**Control Tokens:** OPENING\_PARENTHASES | CLOSING\_PARENTHASES | OPENING\_BRACE | CLOSING\_BRACE | SEMI\_COLON | COMMA

**Access Token:** DOT\_OPERATOR

**Return Token:** RETURN

**Conditional Structure Tokens:** IF | ELSE | WHILE | FOR

The lexical analyzer can backtrack to the previous token using the backtrack function. After calling the backtrack function, the next call to tokenize will return the token last returned by the preceding call to tokenize.

As an example, consider the following line of code:

int Xan = 52.3 + Y;

Repeated calls to tokenize would yield the following results:

myAnalyzer.tokenize() -> INT “int”

myAnalyzer.tokenize() -> IDENTIFIER “Xan”

NOTE: If myAnalyzer.baktrack() is called, then calling myAnalyzer.tokenize() again here yields -> IDENTIFIER “Xan”

myAnalyzer.tokenize() -> ASSIGNMENT\_OPERATOR “=”

myAnalyzer.tokenize() -> DOUBLE\_LITERAL “52.3”

myAnalyzer.tokenize() -> PLUS “+”

myAnalyzer.tokenize() -> IDENTIFIER “Y”

myAnalyzer.tokenize() -> SEMI\_COLON “;”

The lexical analyzer is used by the parser class in order to break the program down into tokens and accurately parse the structure of the program.

***Parser:***

The parser reads through the file generated by the pre-processor token-by-token from repeated calls to the tokenize method of the lexical analyzer class.

The parser creates a symbol table of all the variables and classes. Additionally, the parser creates a parse tree representing all of the statements, functions and classes in the program. Besides this, the parser also combines like terms for literals (ex. 5 + 9.5 is simplified to 14.5), and performs type checking on each statement to make sure that each expression and function call contains type-compatible variables.

Binary operators in the language obey the same precedence order as they do in C++. The parser uses a bottom-up processing technique to determine which order to evaluate expressions in. For example, if the beginning of an expression starts off 5 + 3 \* 8 – 1 … this would be simplified to 5 + (3 \* 8) -1 … -> 5 + 24 – 1 … -> 29 – 1 … From here, further processing of the expression is needed to determine which order to combine the rest of the expression. For example, if the statement continues 29 – 1 – 2, then the expression would simplify to (29 – 1) – 2 -> 26. If the statement continues 29 – 1 \* 5, then the expression would simplify to 29 – (1 \* 5) -> 24.

This is all based on the fact that for any series of two binary expressions being combined, such as X + Y \* B, the expression is either (X + Y) \* B or X + (Y \* B). The fact that multiplication is higher precedence however means that X + (Y \* B) is the correct processing of the statement.

Below is a basic layout of the grammar for my language which the parser follows. Operator precedence is determined using the above guidelines when there is ambiguity as to what order to evaluate 2 binary expressions in. Additionally, certain expressions require certain types as operands (ex. for a Mod Expression, the format is Expression1 % Expression2, where both expressions are either ints or chars). The rules for what operands are allowed for each operator obey the same rules as Java, and there is no operator overloading in Lobster.

Note: The main function is where execution starts.

Also note that any declare-assign statements at the global level must evaluate to a constant literal expression (ex. int x = 5; could appear at the global level, but int x = y; would be syntactically invalid at the global level).

Start Symbol: **Program**

**Program** -> **Global\_Statement Program | Program Global\_Statement | Class\_Definition Program | Program Class\_Definition | Non-Member\_Function Program | Program Non-Member\_Function | Non-Member\_Function**

**primitiveType -> INT | BOOLEAN | CHAR | DOUBLE | STRING | Identifier**

**Type -> ARRAY < Type > | LIST < Type > | QUEUE < Type > | primitiveType**

**Global\_Statement -> Type Identifier SEMI\_COLON | Type Identifier ASSIGNMENT\_OPERATOR Expression SEMI\_COLON**

**FunctionReturnType -> Type | VOID**

**Class\_Definition -> CLASS\_TOKEN Identifier OPENING BRACE CLASS\_BODY CLOSING\_BRACE**

**CLASS\_BODY -> Member\_Function Class\_Body | Class\_Body Member\_Function | Member\_Variable\_Declaration Class\_Body | Class\_Body Member\_Variable\_Declaration | e**

**Member\_Variable\_Declaration -> Type Identifier SEMI\_COLON**

**parameter -> Type Identifier**

**parameterStructure -> parameter | parameter COMMA parameterStructure**

**ParameterList -> parameterStructure | e**

**Member\_Function -> FunctionReturnType Identifier OPENING\_PARENTHASES ParameterList CLOSING\_PARENTHASES OPENING\_BRACE functionBody CLOSING\_BRACE**

**Non\_Member\_Function -> FunctionReturnType Identifier OPENING\_PARENTHASES ParameterList CLOSING\_PARENTHASES OPENING\_BRACE functionBody CLOSING\_BRACE**

**functionBody -> Statement | Statement functionBody | functionBody Statement | conditionalBlock functionBody | functionBody conditionalBlock | e**

**Statement -> Declaration\_Statement | Declare\_Assign\_Statement | Assignment\_Statement | Compound\_Assignment\_Statement | Expression\_Statement | Return\_Statement | SEMI\_COLON**

**Declaration\_Statement -> Type Identifier SEMI\_COLON**

**Declare\_Assign\_Statement -> Type Identifier ASSIGNMENT\_OPERATOR Expression SEMI\_COLON**

**Assignment\_Statement -> Expression ASSIGNMENT\_OPERATOR Expression SEMI\_COLON**

**COMPOUND\_OPERATOR -> PLUS\_EQUALS | MINUS\_EQUALS | TIMES\_EQUALS | MOD\_EQUALS | DIVIDE\_EQUALS | POW\_EQUALS**

**Compound\_Assignment\_Statement -> Expresion PLUS\_PLUS | PLUS\_PLUS Expression | MINUS\_MINUS Expression | Expression MINUS\_MINUS | Expression COMPOUND\_OPERATOR Expression SEMI\_COLON**

**Expression\_Statement -> Expression SEMI\_COLON**

**Return\_Statement -> RETURN Expression SEMI\_COLON | RETURN SEMI\_COLON**

**Expression -> Binary\_Expression | Unary\_Expression | Function\_Call | Variable\_Access | Literal | Control\_Flow\_Jump**

**Binary\_Operator -> PLUS | MINUS | MULTIPLY | DIVIDE | MOD | POW | LOGICAL\_AND | LOGICAL\_OR | LOGICAL\_XOR | BITSHIFT\_LEFT | BITSHIFT\_RIGHT | BITWISE\_AND | BITWISE\_OR | BITWISE\_XOR | LESS\_THAN | LESS\_THAN\_EQUALS | GREATER\_THAN | GREATER\_THAN\_EQUALS | EQUALS\_EQUALS | NOT\_EQUALS**

**Binary\_Expression -> Expression Binary\_Operator Expression**

**Unary\_Operator -> PLUS | MINUS | NOT | BITWISE\_NOT**

**Unary\_Expression -> Unary\_Operator Expression**

**Function\_Call -> Non-Member\_Function\_Call | Member\_Function\_Call | Builtin\_Function\_Call | Constructor\_Call**

**argumentListStructure -> Expression | Expression COMMA argumentListStructure**

**argumentList -> argumentListStructure | e**

**Non-Member\_Function\_Call -> Identifier OPENING PARENTHASES argumentList CLOSING\_PARENTHASES**

**Member\_Function\_Call -> Identifier DOT\_OPERATOR Identifier OPENING\_PARENTHASES argumentList CLOSING\_PARENTHASES | THIS DOT\_OPERATOR Identifier OPENING\_PARENTHASES argumentList CLOSING\_PARENTHASES**

**Builtin\_Function\_Call -> Identifier DOT\_OPERATOR BuiltinTypeFunction OPENING\_PARENTHASES argumentList CLOSING\_PARENTHASES | PRINT OPENING\_PARENTHASES CLOSING\_PARENTHASES | builtinInputFunctions**

**BuiltinTypeFunction -> PUT | PUSH | POP | FRONT | GET | CONTAINS | SORT | FIND | AT**

**BuiltinInputFunctions -> GET\_INT() | GET\_DOUBLE() | GET\_BOOLEAN() | GET\_WORD() | GET\_LINE() | GET\_CHAR() | IS\_INPUT\_FAIL()**

**Constructor\_Call -> new TYPE ( argumentList )**

**Variable\_Access -> THIS | Identifier | Identifier DOT\_OPERATOR Identifier | THIS DOT\_OPERATOR Identifier**

**LITERAL -> INTEGER\_LITERAL | BOOLEAN\_LITERAL | DOUBLE\_LITERAL | CHAR\_LITERAL | STRING\_LITERAL | NULL**

**Controll\_Flow\_Jump -> CONTINUE | BREAK**

**conditionalBlock -> IF\_ELIF\_ELSE\_BLOCK | FOR\_BLOCK | WHILE\_BLOCK**

**blockBody -> { functionBody } | Statement | conditionalBlock**

**If\_Statement -> IF ( EXPRESSION ) blockBody**

**ELIF\_Statement -> ELSE If\_Statement**

**Else\_Statement -> ELSE blockBody**

**ELIF\_REPETITION -> ELIF\_STATEMENT | ELIF\_STATEMENT ELIF\_RREPITITION**

**IF\_ELIF\_ELSE\_BLOCK -> If\_Statement | If\_Statement ELIF\_REPETITION | If\_Statement Else\_Statement | If\_Statement ELIF\_REPETITION Else\_Statement**

**FOR\_BLOCK -> for ( Statement Expression SEMI\_COLON Statement ) blockBody**

**(note: the last statement in the for loop before the start of the block body has no semi-colon, but this detail is omitted from the grammar here for simplicity).**

**WHILE\_BLOCK -> While ( Expression ) blockBody**

As a simple example of how a program could be derived from this grammar, I will provide an example with the following short program:

void printNum(int x)

{

print(x);

}

void main()

{

for(int i = 0; i < 10; ++i)

{

printNum(i);

printNum(i + 10);

}

}

this would be derived as follows:

Program

| |

V V

Non-Member\_Function Non-Member\_Function

| | V V

void printNum ( parameterList ) { functionBody } void main () { functionBody }

| | |

V | |

int x V V

print(argumentList) ; for( Statement Expression ; Statement ) blockBody

| | | | |

V V V V |

X Declare\_Assign\_Statement Binary\_Expression ; Compound\_Assignment |

int i = 0; i < 10 i++ V

{ functionBody }

| |

V V

Statement functionBody

| |

V V

printNum( argumentList ) ; Statement

| |

V V

i printNum(argumentList)

|

V

Binary Expression

| | |

V V V

Expression Binary\_Operator Expression

| | |

V V V

variableAccess + INT\_LITERAL

| |

V V

i 10

When the parser is done, a complete parse tree and symbol table exists to describe the program, which is then sent to the code generator for further processing.

***Code Generator:***

***General Register Format:***

The Code Generator takes as input a complete parse tree and symbol table representing a user’s program, and outputs the bytecode representation of the Lobster code to run this program into a file with a name of the form inputFileName.Lobster. Every line of bytecode in Lobster is represented as a 15 byte sequence. Some instructions use less than 15 bytes to represent all of their values. However, in this case, what’s left of the 15 bytes is simply padded with 0s.

There are 256 registers numbered 0 – 255 which are used by a running Lobster program to store values and addresses. Registers can hold values of any type, including addresses of a variable, offsets into the code, integers, booleans, doubles, and chars. Registers 0 – 222 are general purpose registers which can hold temporary values. Registers 223 – 253 are used to represent the arguments to a function (with register 223 holding argument 0, register 224 holding argument 1 etc.). Register 254 is the RETURN\_REGISTER, which is where arguments are copied to in order to be returned by a function. Register 255 is the THIS\_REGISTER which is used in member functions to store the address of the variable that the function was called on. There is also a register which is separate from the other 256 called the PC register, which stores the offset into an array of chars representing Lobster instructions where the start of the next instruction to be executed is.

***Example Instruction:***

In order to see what an instruction looks like in the language, let’s consider the most complicated instruction in the language: assignments. Assignments for variables are handled via the assignment instruction, which has the format:

ASSIGNMENT\_OP Mem\_Reg var\_Type RIGHT\_ARG

The opcode for each instruction is a 1 byte number (since there are less than 256 instructions in the language). Mem\_Reg is a 1 byte number which represents a register between 0 – 255, with the value stored in Mem\_Reg being the address of the variable. var\_Type is a 1 byte number of the enumerated type Op\_Type which represents the type of the variable referred to by Mem\_Reg. RIGHT\_ARG is a combination of bytes which specify if the next argument is a literal, a register value, or an address of a variable stored in a register, and then where to find the next value.

For example, the following is a representation of the user code x = 5; where x is a variable of type int and register 36 contains the address of variable x in it:

ASSIGNMENT\_OP \*R36 INTEGER\_T LITERAL\_L 5

Many instructions in the language are relatively simple, however, like the unconditional jump instruction, which has the format:

JUMP\_TO lineNumber

***Code Anatomy:***

From here, we can most effectively understand how the code generator works by examining the basic anatomy of a Lobster program. Below is a representation of the basic components of a sample program:

1. GLOBAL INSTRUCTIONS (Optional)
2. JUMP TO MAIN INSTRUCTION
3. NON\_MAIN\_INSTRUCTIONS (Optional)
4. INSTRUCTIONS FOR MAIN FUNCTION
5. NON\_MAIN\_INSTRUCTIONS (Optional)
6. String Literals

Global instructions represent the code to execute all global statements by users that are outside of functions or classes, such as int x = 5; All of these instructions are put at the top of the Lobster code, which is done in the Code Generator’s first pass through the parse tree. After this, a jump to main instruction is placed which sets the PC to be at the start of the main function. After this, all non-global statements are processed in order from the top down. Member variable declarations in classes are temporarily skipped, although these are handled in the function for the constructor of the class. Member functions of classes (besides the constructor) are handled identically to non member functions, however (ignoring the use of values in the THIS\_REGISTER) . After the instructions for the main function, there may or may not be instructions for more functions (if any functions come after main in the file the user wrote). At the very bottom of the code, after all of the instructions are written, string literals are written at the bottom of the code. As a result of this, string literals in the program are represented as integer offsets into the array of chars representing the bytecode file.

***Variable Accesses and Classes:***

Variable declarations in the language can occur in 2 forms: Global declarations, and Stack declarations. Global Declarations take the following form:

GLOBAL\_DECLARE\_OP Unique\_Identifier\_Num

Where Unique\_Identifier\_Num is an int which is used to represent the variable in the program (as opposed to a string, which would take much longer to read through). Every identifier and literal in the program is given a unique identifier num by functions in the SymbolTable class, which assign values to identifiers, literals, and blocks starting from 0, and increasing by 1 each time a new definition is encountered (the Parser ensures that the symbol table is built correctly)

Stack\_Declarations are used inside of functions (the global declare op is used only in the global scope) to declare variables onto the stack, and take the form:

STACK\_DECLARE\_OP Unique\_Identifier\_Num

Objects of a user-defined class are represented as an array of long longs (since long longs are 8 bytes, they can hold any other primitive type or pointer). For example, if a user-defined class has 4 member variables, then the object will be represented as an array of 4 long longs. The instruction which actually creates a user-defined object on the heap with enough space to store the object’s member variables is inserted as the first instruction in the object’s constructor function, with the instruction:

ADD\_MEMBER\_VARS\_TO\_OBJ Number\_Of\_Vars

After this, the THIS register will have the address of the newly allocated object in it

To get the address of a variable, the 2 instructions:

MOVE\_GLOBAL\_ADDR\_TO\_REG Unique\_Identifier\_Num destRegNum

MOVE\_STACK\_ADDR\_TO\_REG Unique\_Identifier\_Num destRegNum

perform the function of copying the address of the variable referred to by unique identifier num into the register referred to by destRegNum.

Additionally, the instruction

GET\_MEMBER\_VAR\_ADDR\_OF\_OBJ destReg ObjReg offsetIntoObj

will get the address of the variable which is variable number offsetIntoObj of the object referred to by the address in ObjReg, and store this resulting address in destReg.

For a more concrete example, suppose that we have a class called Car, whose internal array of member variables contain int length, int width, and int height in that order.

Assume that register 52 holds the address of a Car object called myCar

Then, the following instruction would put the address of the variable myCar.height into register 35:

GET\_MEMBER\_VAR\_ADDR\_OF\_OBJ R35 \*R52 2

***Function Structure:***

Whenever a function call occurs, the instruction SAVE\_REGISTER regNum is used to save the values of each register currently in use onto the call stack. Additionally, the value PC + BYTES\_PER\_LINE is stored onto the stack. Then, an instruction of the format CALL \_FUNCTION location is executed, which sets PC equal to location, and pushes a new stack frame onto the stack for the new function.

In every function, the first lines of the function are of the form SET\_PARAMETER ArgReg Unique\_Identifier\_Num, which creates a variable on the stack identified by Unique\_Identifier\_Num and initializes it with the value stored in ArgReg. This is used to create parameter variables for functions, which are stored on the stack. For example, the instruction SET\_PARAMETER 223 18

would create a variable on the stack with an ID number of 18, which would be initialized with the value of register 223.

Whenever a return EXPRESSION statement is encountered, the value of EXPRESSION is first copied to RETURN\_REGISTER. Then, the instruction RETURN\_FROM\_FUNCTION is executed, which pops the top value off of the stack of PC values, sets PC equal to this value, pops the current stack function frame, and continues execution from the new value of PC.

In the calling function, after the line CALL\_FUNCTION location, the next lines of the program will be of the form RESTORE\_REGISTER regNum, where every register which was saved before calling the function will be restored from the stack here.

At the end of every void function is a RETURN\_FROM\_FUNCTION expression, regardless of whether or not return; was explicitly written there. In every non-void instruction, the special instruction OUT\_OF\_BOUNDS\_FAULT is placed after the last line of code in the function. If this line is hit during execution of the program by the virtual machine, then an exception occurs, because a function that was supposed to return a value did not return a value. As a result, the program will then terminate execution.

***Conditional Execution:***

In Lobster, If-else\_if-else branches are handled with the following logic scheme:

If statement structure:

1. Skip to 3
2. EXIT if-elif-else block (jump to 11)
3. BRANCH ON FALSE (Boolean Condition) to 6
4. Block Body
5. Jump to line 2

Else if statement structure:

1. BRANCH ON FALSE (Boolean Condition) to 9
2. Block body
3. Jump to 2

Else statement structure:

1. First line of else body
2. Last line of else body
3. Outside of Else

Everywhere in the above diagram where Jump is listed refers to an unconditional jump, implemented via the instruction JUMP\_TO codeLocation

The instruction BRANCH ON FALSE Expression codeLocation

is a special instruction which jumps to codeLocation if the Boolean value stored in Expression is false. Otherwise, the pc is set to the next instruction after this (PC += BYTES\_PER\_LINE), just like any other instruction.

Also of note is that section 2 of the above code is the line which all statements in the else if and else blocks jump to when they are true. From there, after the last line of code is processed by the code generator, section 2 will have its jump location set to section 11 (after the end of the if-else\_if-else block) of the above diagram. This is done purely to avoid having to go back through each individual block when the else or last else if is encountered to see where to jump to in each block. Instead, only one line in the first if statement has to be updated.

While loops in Lobster are implemented via the following format:

While Structure:

1. skip next instruction
2. JUMP to 6
3. BRANCH ON FALSE (boolean condition) 6
4. LOOP BODY
5. JUMP to 3
6. Outside of loop

Note that section 2 is used so that when a break instruction is encountered in the block, it can simply jump to section 2, which in turn jumps to the first line of code outside of the block.

For loops in Lobster are implemented via the following format:

For Structure:

1. Initial statement lines (initializer statement)
2. Skip next 2 instructions
3. CONTINUE instruction (jump to 7)
4. BREAK instruction (jump to 9)
5. BRANCH ON FALSE (Boolean condition) to 9
6. LOOP BODY
7. Post-action statement.
8. JUMP to 5
9. Outside of for loop

When the Code Generator is finished executing, the result is a complete file in Lobster bytecode. Additionally, for debugging purposes, I have included a version of the CodeGenerator that can output instructions in human-readable form if the option TRUE is used as an argument on the command line. As a complete example of this, consider the following program that a user could type in:

int factorial(int n)

{

if(n <= 0)

return 1;

return n \* factorial(n – 1);

}

int main()

{

print( factorial(8));

}

This would be converted to the following Lobster-bytecode program:

0. JUMP\_TO\_MAIN, CODE\_LOCATION: 33

1. SET\_PARAMETER, SOURCE\_REG: 223, IDENTIFIER\_NUM: 2

2. JUMP\_TO, CODE\_LINE\_NUMBER: 4

3. JUMP\_TO, CODE\_LINE\_NUMBER: 13

4. MOVE\_STACK\_ADDR\_TO\_REG, DEST\_REG: R9, IDENTIFIER\_NUM: 2

5. DEREFERENCE\_INTEGER, DEST\_REG: R120, SOURCE\_REG: \*R9

6. COPY\_INT\_LITERAL\_TO\_REGISTER, DEST\_REG: R103, 0

7. LESS\_THAN\_EQUALS\_OP, DESTINATION: R132, First Arg: R120, Second Arg: R103

8. BRANCH\_ON\_FALSE\_REG\_VAL, R132, CODE\_LOCATION: 13

9. COPY\_INT\_LITERAL\_TO\_REGISTER, DEST\_REG: R120, 1

10. COPY\_REGISTER\_TO\_REGISTER, DEST\_REG: R254, SOURCE\_REG: R120

11. RETURN\_FROM\_FUNCTION

12. JUMP\_TO, CODE\_LINE\_NUMBER: 3

13. DEREFERENCE\_INTEGER, DEST\_REG: R103, SOURCE\_REG: \*R9

14. SAVE\_REGISTER, R9

15. SAVE\_REGISTER, R103

16. SAVE\_REGISTER, R120

17. SAVE\_REGISTER, R132

18. MOVE\_STACK\_ADDR\_TO\_REG, DEST\_REG: R18, IDENTIFIER\_NUM: 2

19. DEREFERENCE\_INTEGER, DEST\_REG: R78, SOURCE\_REG: \*R18

20. COPY\_INT\_LITERAL\_TO\_REGISTER, DEST\_REG: R143, 1

21. MINUS\_OP, DESTINATION: R23, First Arg: R78, Second Arg: R143

22. COPY\_REGISTER\_TO\_REGISTER, DEST\_REG: R223, SOURCE\_REG: R23

23. CALL\_NON\_MEMBER\_FUNCTION, FUNCTION\_LOCATION: 1

24. RESTORE\_REGISTER, R9

25. RESTORE\_REGISTER, R103

26. RESTORE\_REGISTER, R120

27. RESTORE\_REGISTER, R132

28. COPY\_REGISTER\_TO\_REGISTER, DEST\_REG: R132, SOURCE\_REG: R254

29. MULTIPLY\_OP, DESTINATION: R120, First Arg: R103, Second Arg: R132

30. COPY\_REGISTER\_TO\_REGISTER, DEST\_REG: R254, SOURCE\_REG: R120

31. RETURN\_FROM\_FUNCTION

32. OUT\_OF\_BOUNDS\_FAULT

33. COPY\_INT\_LITERAL\_TO\_REGISTER, DEST\_REG: R48, 8

34. COPY\_REGISTER\_TO\_REGISTER, DEST\_REG: R223, SOURCE\_REG: R48

35. CALL\_NON\_MEMBER\_FUNCTION, FUNCTION\_LOCATION: 1

36. COPY\_REGISTER\_TO\_REGISTER, DEST\_REG: R175, SOURCE\_REG: R254

37. PRINT\_OP, Op\_Type: INTEGER\_T, SOURCE\_REG: 175

38. COPY\_INT\_LITERAL\_TO\_REGISTER, DEST\_REG: R175, 0

39. COPY\_REGISTER\_TO\_REGISTER, DEST\_REG: R254, SOURCE\_REG: R175

40. EXIT\_MAIN, RETURN\_TYPE: INTEGER\_T

***Lobster Virtual Machine:***

After the compilation of a lobster program has finished, the program can be run on the Lobster Virtual Machine by typing in a command of the format ./Lobster inputFileName.Lobster

The first thing the virtual machine does is read the file into an array of chars. After this, the file initializes the stack and global sections of memory. Both are implemented as objects of type tableOfStacks, where each contains a stack of tables. A new table is pushed onto the stack whenever a function is entered, and is popped from the stack when a function exits (the global section of memory contains just one table on the stack). The table maps the unique identifier integer of a variable (which is based on the numbers contained in the GLOBAL\_VARIABLE\_DECLARE and STACK\_VARIABLE\_DECLARE instructions) to an object of type Variable, which is the name of the class that represents all memory variables.

The Variable class contains a single long long called myVal as a member variable, which is used to hold data of any type via type conversions and casting. To store the address of a Variable in a register, the address of myVal is stored in the register. To see how this all works, consider the following pseudocode, which shows how the statement int x = 2; is handled. Assume that x maps to the unique identifier number 8:

Class Variable {

public:

Variable();

long long myVal;

};

STACK\_DECLARE\_OP 8

MOVE\_INT\_LITERAL\_TO\_REG R1 2

MOVE\_STACK\_ADDR\_TO\_REG R0 8

ASSIGNMENT\_OP R0 INTEGER\_T REGISTER\_VAL R1

The last 2 instructions are then implemented as follows, assuming that myVar is the Variable object corresponding to X:

\*((long long\*\*) &R0) = &(myVar.myVal);

\*((long long\*) R0) = R1;

In the last statement, the value contained in register R0 is treated as representing a pointer to a long long, and the value referenced by this address is then set equal to the value stored in register R1.

User-Defined types in the language are implemented via the class UserObject, which just contains a long long pointer. This variable gets set in the first line of the constructor to point to an array of long longs with as many entries as the number of member variables in the class. The address of this pointer is in turn stored in a register whenever an object’s address of a user defined type is copied to a register. Furthermore, the myVal variable of a Variable object will store this address when an object is assigned to a variable.

For the builtin types (Arrays, Lists, and Queues), myVal will be set equal to a pointer to the builtin type.

After the Lobster program sets the PC to 0, the program is then ready to start execution.

***Program Execution:***

The program reads through the first byte of the byte referenced by charArray[PC] to get the opCode, where charArray is the array of unsigned chars representing the bytecode of the program. A giant switch statement then determines what will happen next based on which opCode occurred. For any instruction that doesn’t cause a jump to occur, the last thing the virtual machine will do before reading the next line is set PC += BYTES\_PER\_LINE (Note: BYTES\_PER\_LINE is 15, since every instruction has 15 bytes per line, and an unsigned char is 1 byte in length)

Whenever a function call is encountered, the program pushes the value of PC + BYTES\_PER\_LINE onto a stack of functionReturnLocations, and then sets PC equal to the integer which comes after the opCode in the charArray (the function call instructions have the format FUNCTION\_CALL\_OP codeLocation).

Additionally, there is an array of 256 long longs, which represent the 256 registers the program has access to. Furthermore, there is an array of 256 stacks of long longs, which are used to store registers when functions are called and to restore values when the program returns from a function. More specifically, when the instruction SAVE\_REGISTER 13 is encountered, the value in register 13 is pushed onto the 13th stack in the array of stacks. When the instruction RESTORE\_REGISTER 13 is encountered after the function call, the value of register 13 is set equal to the value at the top of the 13th stack in the array of stacks, and then the 13th stack in the array of stacks is popped.

All allocation of memory in the program for both local variables, global variables, and memory for objects is internally allocated on the heap by the virtual machine (for clarity, I refer to local variables as being “stack variables” earlier in this document to distinguish them from variables declared in the global scope, which are kept in a separate table of stacks). I haven’t currently implemented a way to free memory that is no longer in use. However, I do push all memory allocated onto a list of type long long\*, and as a result, my program generates no memory leaks.

Output functions are handled via calls to cout, while input functions are handled via cin and getLine.

***Builtin Types:***

In the Lobster programming language, there are 3 builtin types: Arrays, Lists and Queues. These classes each contain several builtin functions to go with them, including at(int) to return the element at a certain index, get(element) to return the index of an element (starting from element 0), contains(element) which returns a Boolean indicating if the structure contains this value, and several other functions.

The array class also contains a sort function, which internally makes use of multi-threading to speed up processing. More specifically, the algorithm used is the following version of mergesort/timsort: First, break the array in half, and sort the first half of the array in one thread, and sort the second half of the array in the other thread. In the sort function, whenever the number of elements left to sort in a specific chunk is less than or equal to 60, then insertion sort is performed on that segment. Otherwise, the segment is broken in half, and the sort function is called on each half. After each of these halves are sorted, they are then combined into one sorted array using the merge process used in merge sort. After both threads finish, merge is called again to merge the two sorted subarrays, which results in the final array being fully sorted.

The Array, List and Queue classes each use locks to ensure that data races will not occur if multiple threads try to access the object at the same time. In Arrays and Lists, one big lock is used which is acquired and released for each builtin function. For queues, a Michael and Scott queue is used with 2 locks. By using this scheme, it is possible to enqueue and dequeue from the same queue simultaneously with no data races. This is achieved by initializing the queue to contain an empty dummy node, which acts to prevent the edge case where the array has only 1 element in it and an insertion and deletion occur simultaneously.

***Future Features:***

There were some features which I would have liked to have included in my language, but I didn’t have the time to do it. Among other things, I would like to have included switch statements, a builtin hash table type, and a garbage collector for my language. Additionally, I also wanted to include a pipeline structure for my language (technically, the code to create a pipeline in my language already exists and is functional, but I didn’t have time to make the code/statements to execute a pipeline). In a pipeline, multiple functions would work in parallel via multithreading, with the output of each function being sent to a queue. Then, the next function in the pipeline would pop this argument from the queue and process it, while sending its own output to the queue for the next function. For example, consider the following program:

int firstFunc(Array<int> myArray)

{

for(int I = 0; I < myArray.length(); ++I)

pipePush->(myArray.at(I) \* 15);

TERMINATE();

}

bool secondFunc(int a)

{

pipePush->(a == 10);

}

double thirdFunc(bool a)

{

if(a)

pipePush->(5.0);

else

pipePush(3.1415);

}

In a pipeline, whenever firstFunc->pipePush() was called, it would send its output along to function 2, which would be operating in parallel in a second thread. Function 2 would then receive this input and send its output to function 3 (while functions 1 and 2 continue to run) and so on. More specifically, in a pipeline, the pipePush statement sends a value to an internal queue and then continues execution from there. If the thread reaches the end of the function without hitting a TERMINATE() function call, then PC is set back to the top of the function, and the thread then goes to sleep until the thread ahead of it pushes a value onto its input queue, which wakes up the sleeping thread. When a function calls TERMINATE(), this stops the function from repeating and pushes a termination signal onto the queue. When a thread later on in the pipeline dequeues the TERMINATE signal, that function will also push a TERMINATE signal onto its output queue, and will then halt execution as well. In this manner, several lengthy calculations and computations that are mostly independent (with some values needed from other calculations at certain key points) could occur simultaneously, which would greatly speed up the execution of a program.

Another structure which I would have liked to have would be an event-dispatch structure, which would be similar to a pipeline, except that 1 function would send its output to the input queue for every other function/thread in the structure, which would all work in parallel. For a better picture of what this structure would look like, consider the diagram below:

secondFunction thirdFunction

**firstFunction**

Fourth Function Fifth Function

***Concluding Remarks:***

With everything said and done, I enjoyed writing the Lobster programming language. Now that I have more free time over summer break, I will (hopefully?) be able to do more work on my language, and establish more features for the language. I hope you have enjoyed reading this document, and I hope you enjoy trying Lobster for yourself as well!

***Link To Lobster Program Source Code:***

<https://github.com/SkylerBrivic/Lobster>