CSC488 Assignment 4

Team 2

Tuesday March 17, 2015

Eric Chen | g3chencm
Francesco Gramano | g2graman
Eric Snyder | g4snyder
Winston Yeung | g2yeungw

Table of Contents

Overview

For code generation we will be taking advantage of the visitor pattern. This pattern was previously used for semantic analysis, and will continue to work well for code generation. We will be defining a visitor class, CodeGenVisitor, which will be used to traverse the AST. This visitor will generate the appropriate code for each statement.

With influence from section 11.4 in the textbook, another type of visitor, LHSVisitor will be used to evaluate the locations of variable references. This visitor will be used to generate code to put references of variables at the top of the stack.

1. Storage

a. Variables in the main program

The main program is treated as a major scope, and will always be the first scope on the symbol table. Variables for the main scope will then thus be the first placed onto the stack so the main scope will begin at **startMSP**. All variables are then stored with an offset from the beginning of the current scope. Each variable in each scope knows what its offset is into the scope and the scope knows where it begins in memory so the position of every variable in every scope can be calculated quickly. The offset of each variable is stored in the symbol table when the variable is first defined.

The calculation for the actual offset of a variable in the main program scope would be

LOCATION = start + (offset * wordsize)

When a variable x is first defined, 0 is pushed to the stack to reserve memory for it.

```
PUSH 0
```

Assuming a variable x has already been defined (and thus exists on the symbol table in the current scope) to assign the variable to an integer X the operations would be as such:

```
ADDR LL OF (Where LL is the lexical level and OF the offset of the
variable into the symbol table)
PUSH X
STORE
```

And then to assign a variable x = x + 5 (which will both load and store a variable x)

```
ADDR LL OF (address of x is now on stack)
ADDR LL OF
LOAD  (address of x is on stack again, load will pop it and push its value to top)
```

`PUSH 5` (the value of x and 5 are now on top of the stack)
`ADD` (x + 5)
`STORE`

b. Variables in procedures and functions

Each time a new major scope is opened, a new symbol table is also created. As long as each symbol table knows where its address space begins in memory and each variable inside of it knows its own offset, its simple to calculate the location of variables within each scope.

The exact order that variables are stored is detailed in section a. of Functions and Procedures. The actual storage of variables remains largely the same, their symbol table entry will have the offset of the variable set, and because each variable can already return the symbol table it is part of, the store and load examples above will be the same with the LL used obviously being larger than the 0 of the main scope. When the scope ends the variables will no longer need to be stored in memory and that space can be written over much like what happens for minor scopes.

c. Variables in minor scopes

Our implementation doesn't create new lexical levels for minor scopes, so all of them will be created within the current major scope. Two minor scopes created one after another will occupy overlapping spaces in memory as variables inside of the minor scope can't exist at the same time, with the first scope being destroyed before the second is set up (and thus being written to a similar space in continuous memory). For more details, refer to the part a of the function section.

d. Integer and boolean constants

Integer and boolean constants can be represented by simply pushing the value without adding it to the symbol table.

For an integer constant:	PUSH V (where V is the constant)
For a boolean constant	PUSH MACHINE_TRUE (assuming the value was true)
	PUSH MACHINE_FALSE (assuming the value was false)

e. Text constants

Text constants are stored similarly to integers and booleans. Strings are used for printing, so any string constant in code can be replaced with the following:

PUSH C (where C is the ASCII code of the first character)
PRINTC

The number of times this is repeated depends on the length of the string. The end of each print will be followed by a push printc for an ASCII null terminator.

2. Expressions

a. Constant Values

Boolean Constants

During the parsing phase of an expression, when a Boolean literal is encountered, the appropriate literal (MACHINE_TRUE or MACHINE_FALSE) is pushed onto the stack. The literal is later popped from the stack for evaluation of the expression in which it was contained. If necessary, a placeholder can be put in the place of the Boolean constant, signifying that the appropriate literal should be popped from the stack.

During evaluation of that expression, when the literal (or placeholder for the literal) is reached, the according Boolean literal is simply popped off the stack with the instruction named POP

Examples

- During parsing ‘boolean b <= true’ becomes
  `PUSH MACHINE_TRUE`
- When the literal gets accessed it is simply popped with POP

Integer Constants

With a fixed range, Integer constants will therefore have a fixed, presumably of which the PUSH instruction could accommodate to push it to the top of the stack. Similar to Boolean constants, when the code generator reaches the Integer literal for an expression using an Integer literal, the appropriate literal is pushed onto the stack (during the parsing phase of the expression) to later be popped from the stack for the expression from which it is contained. If it should be necessary, a placeholder can be put in the place of the Integer constant, signifying that the appropriate literal should be popped from the stack.

During evaluation of that expression, when the literal (or placeholder for the literal) is reached, the according Integer literal is simply popped off the stack with the instruction named POP

Example

- During parsing ‘integer i <= 5’ becomes
  `PUSH 5`
- When the literal gets accessed it is simply popped with POP

Text Constants

Our lives are simplified since the only occurrences of text constants is in a **put** statement and the size of a text constant is not allowed to be static (since the **put** statement requires a text literal, which inherently has a fixed length). That said, whenever the code generator reaches a Text constant for an expression using a Text literal, the appropriate literal is pushed onto the stack on a per-character basis

b. Accessing Scalar Values

Since we can only have scalar integer and Boolean variables, we will only bother with scalar variables that refer to integer and boolean literals.

Since code generation would come after semantic analysis, where we would have already built the simply table in accordance to a traversal of the AST, and in order for semantic analysis of referencing a scalar variable to pass, the variable would have previously had to been assigned a value so we should already have that variable's information in the symbol table (such as its lexical level and offset).

To access the scalar variable, it's lexical level and offset will be looked up in the symbol table during code generation. Then, its address can be pushed onto the stack using the ADDR command, followed by a load. Therefore, the value of the variable will be at the top of the stack for further evaluation.

Example

For example, if we have the variable 'x' located in lexical level 1 and offset 10, then the following would be emitted:

```
ADDR 1 10
LOAD
```

After these commands, the value of x will be at the top of the stack.

c. Accessing Array Elements

Two-dimensional arrays

Before being able to explain how array elements will be accessed, we must first explain how array elements would be stored. If we had a two dimensional array A initialized with dimension 1 having size X (not to say that the variable's bound had been set by a variable, but that X represented A's size, whatever that would be) and dimension 2 having size Y. Assuming the index bounds to our array are strictly non-negative (since its type equivalence with the more general case will be explained a little below), if we were to reference A with A[x, y] with 0 <= x < X and 0 <= y < y then this scenario can be reduced to accessing the (x*X + y)th element of an array of a single dimension of size (X * Y). With this strategy we should only have to store the address of the array and compute its proper offsets into its one-dimensional contiguous analog with respect to both subscripts for accessing an element in array A over each dimension.

Therefore, upon declaration of a two dimensional array 'integer A[10, 10]', for example, its one-dimensional contiguous analog would be an array of one dimension but 10 * 10 = 100 elements, where if i and j were literals 'A[i][j]' would refer to element i + j of its analog.

N.B.: ‘integer A[10, 10]’ is structurally equivalent (but not equivalent in indices) with ‘integer A[i..j, k..l]’ where i,j,k,l are literals, i < j and k < l, and j – i = 10 = l – k, for the case of a two dimensional array with general bounds.

When it comes to declaring array A, then, for storage and accessing purposes, its one-dimensional analog is used where A[i, j] = A’[i*X+j] where A’ is the one-dimensional analog, and X is the size of the dimension referenced by i.

Along with allocating the appropriate amount of space in memory (with appropriate arithmetic to pointers msp and msl) and right address being stored in memory, it would be useful, should the language be built to allow the size of an array to be declared with an integer, to store the total size of the array as well to make sure of no out-of-bounds array references at runtime. However, this is the hypothetical extensions and need not be included since arrays will be declared with literal-sized dimensions. Since booleans and integers are each one word long then for both types of arrays we would use the scheme of the previous bullet to calculate the proper index of the two-dimensional array’s one-dimensional analog, add that to the address of the Array’s 0th element, and LOAD that address from memory to correctly access an element of integer and boolean two-dimensional arrays.

Space for the array will be allocated at the top of stack. During code generation, the beginning of the array will be stored in the symbol table. The size of the two dimensions will be pushed to the stack, followed by a MULT. Then a DUPN will be used to move the stack pointer up.

To access the array, what we would do is emit the base address of the array using the ADDR command, using the lexical level and offset (to the beginning of the array). Then we push the first index onto the stack. If the lower bound is not 0, then it is followed by the lower bound of the first dimension and then we emit a SUB. Then the size of the second dimension is pushed on to the stack, and then we emit a MULT. After this, we emit the second index. Again, if the lower bound of the second dimension is not 0, then it is emitted, followed by a SUB. A final ADD is emitted to get the final offset in the array. Finally, we emit another ADD to get the final address. So for instance if we had the array B[1..10, 1..10] the following code would be emitted:

```
PUSH 0
PUSH 10
PUSH 10
MULT
DUPN
```

Say the offset of B starts at 10 in lexical level 3. To access the B[2, 3] the following would be emitted:

```
ADDR 3 10  % base address of B
PUSH 2 % first index
PUSH 1 % lower bound of table row
SUB
PUSH 10 % number of columns
MULT
```

```
PUSH 3 % second dimension
PUSH 1 % lower bound of second dimension
SUB
ADD
ADD
LOAD
```

This is because array B starts at 10, and the item we are accessing is the 23rd element. So, the offset will be 33.

Then, if we were to allocate space for another variable after this array, it would start at offset 110.

One-dimensional arrays

Integer and boolean arrays of one dimension are declared with a size with respect to a single literal, and therefore when it comes it accessing an element in such an array, its offset is simply that single literal which we'd add to the address of the 0th element to access an arbitrary element in a one-dimensional integer or boolean array.

Similar to the two dimensional array, one dimensional arrays will be stored on the stack as well. This will be done by using the DUPN command to move the stack pointer up to save space. The offset for the beginning of the array in the lexical level will be stored in the symbol table during code generation. The access will be exactly the same as the two dimensional array. We emit the base address of the array to the top of stack using ADDR. Then we push the offset to the top of stack and then emit an ADD to get the final address.

For example, if we wanted to allocate space for the array A[10], then the following would be emitted:

```
PUSH 0
PUSH 10
MULT
DUPN
```

Now, say that A is at lexical level 0, beginning at offset 4. To access A[6] the following would be emitted:

```
ADDR 0 4
PUSH 10
ADD
LOAD
```

d. Arithmetic operators +, - , *, /

Since procedures for accessing array elements, scalar variables, and literal constants have been described, we can assume that each of these subexpressions can be treated as literals (i.e., values that we can plug into the arithmetic operators)

Deriving a value from arithmetical expressions must entail a recursive procedure by which left-associativity gets respected and evaluation occurs from the deepest level (of parentheses) to the outermost.

Since multiplication/division have higher precedence than addition/subtraction, if the two different sorts of arithmetical expressions are found at the same parenthetical depth for the derivation, then multiplication/division get evaluated first from left-to-right. Only after multiplication/divisions have been dealt with on a level can addition/subtraction expressions be evaluated.

In the case that we must deal with an expression involving occurrences of '*' or '/', assuming that the previous conditions have been met (to respect associativity and precedence) and since we would have the operands of such expressions, the literals which the operands evaluate to get pushed onto the top of the stack from right-to-left so that the right operand of any of the arithmetical operators is pushed just before that of the left. That said, after pushing both values, we would simply invoke the proper arithmetic instruction: ADD for addition, SUB for subtraction, MUL for multiplication and DIV for division. These instructions push the result to the top of the stack, so there is nothing more we needed to do after the arithmetic to be able to recursively evaluate an expression.

Note that unary '-' has the highest precedence, so upon encountering a negative literal or a negative applied to a scalar variable on a level, it must be evaluated before arithmetical operators, we simply invoke NEG and the value at the top of the stack at the time is set to its negative.

e. Comparison operators '<', '<=', '=', '!=', '>=', '>'

The first thing to note is that out of the above operators, the only instructions we are provided with which would directly map to the above operators are EQ and LT which map to **'='** and **'<'** respectively

The rest of the operators, then, broken down into expressions containing familiar operators. We can represent **'a != b'** with the sequence of instructions:

```
PUSH(b)
PUSH(a)
EQ
NOT
```

(where NOT is an instruction we will simulate in part f)

Similarly, **'a >= b'** can be represented with the sequence of instructions:

```
PUSH(b)
PUSH(a)
LT
NOT
```

‘a > b’ can be represented as the sequence:

```
PUSH (a)
PUSH (b)
LT
```

Lastly, **‘a <= b’** would be represented by the sequence:

```
PUSH(b)
PUSH(a)
LT
PUSH(b)
PUSH(a)
EQ
OR
```

Note that for the above operations, if a or b were variable references the appropriate loads would be emitted instead.

f. Boolean operators ‘&’, ‘|’, ‘!’

The first thing to consider is whether or not to add support for short-circuiting of the Boolean operators ‘&’ and ‘|’. However, since during semantic analysis we would have already checked that both operands be of type Boolean, assuming semantic analysis passed this makes it impossible for expressions like ‘true & (1 / 0)’. Furthermore, since both operands will be validated to have type Boolean then whether or not short-circuiting is supported, evaluation of Boolean expressions joined by these operators will not incur side-effects. Short-circuiting, therefore, results in no difference of evaluation of Boolean expressions, so it will be omitted for their implementations.

Next, since operators ‘&’ and ‘|’ are left-associative, when it comes to evaluate their operands (before being able to give expressions containing them their final result), the left operand must be evaluated first, until the leftmost branch of that AST subtree is consumed. The left-most expression gets evaluated and the parent gets evaluated, and the process continues recursively.

Similar to expressions involving the arithmetical operators, after evaluation of both operands in an expression, each operand gets pushed onto the stack right-to-left and the according instruction is invoked which will pop those operands and push their result after application of the operator. For the ‘|’ operator, this is straight-forward: the OR instruction.

However, since we lack a Boolean negation machine instruction (which is different from the NEG instruction since MACHINE_FALSE != -- MACHINE_TRUE), the ‘&’ and ‘!’ operators must be emitted indirectly. Noting that negation can be represented as (where **val** is a boolean value) 1 – val = result where result is the resulting negation which is to be pushed atop the stack, we can therefore simulate negation with the sequence of instructions

```
… sequence of instructions to retrieve val ...
PUSH(1)
SUB
```

For instance, if val was a boolean literal, like MACHINE_FALSE then we would push 0. If it was a variable then we would need to emit code to get the variable's value. If it was another expression, that expression would be evaluated.

Now that we can simulate negation, we can use it to represent '&' by means of De Morgan's Laws, noting that a & b = not(not(a) | not (b)).

Since we can represent both '|' and '!', we can now use the above form to represent an expression containing '&'. We can then simulate an '&' expression with the following sequence of instructions (where a and b are Boolean constants, and NOT is the Boolean negation operator which is described a couple of bullets above):

```
PUSH(b)
NOT
PUSH(a)
NOT
OR
NOT
```

Now all three Boolean operators are representable. The only other thing to note is that when encountering a Boolean expression, since '!' has a higher precedence than '&' or '|', unless operator precedence is modified with parentheses then the operand with a '!' applied is computed first.

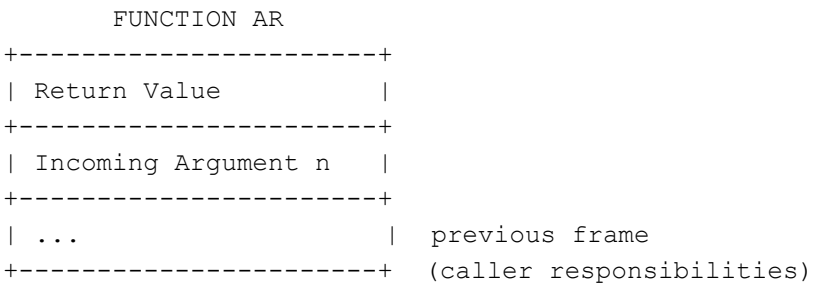
g. Describe how you will implement anonymous functions

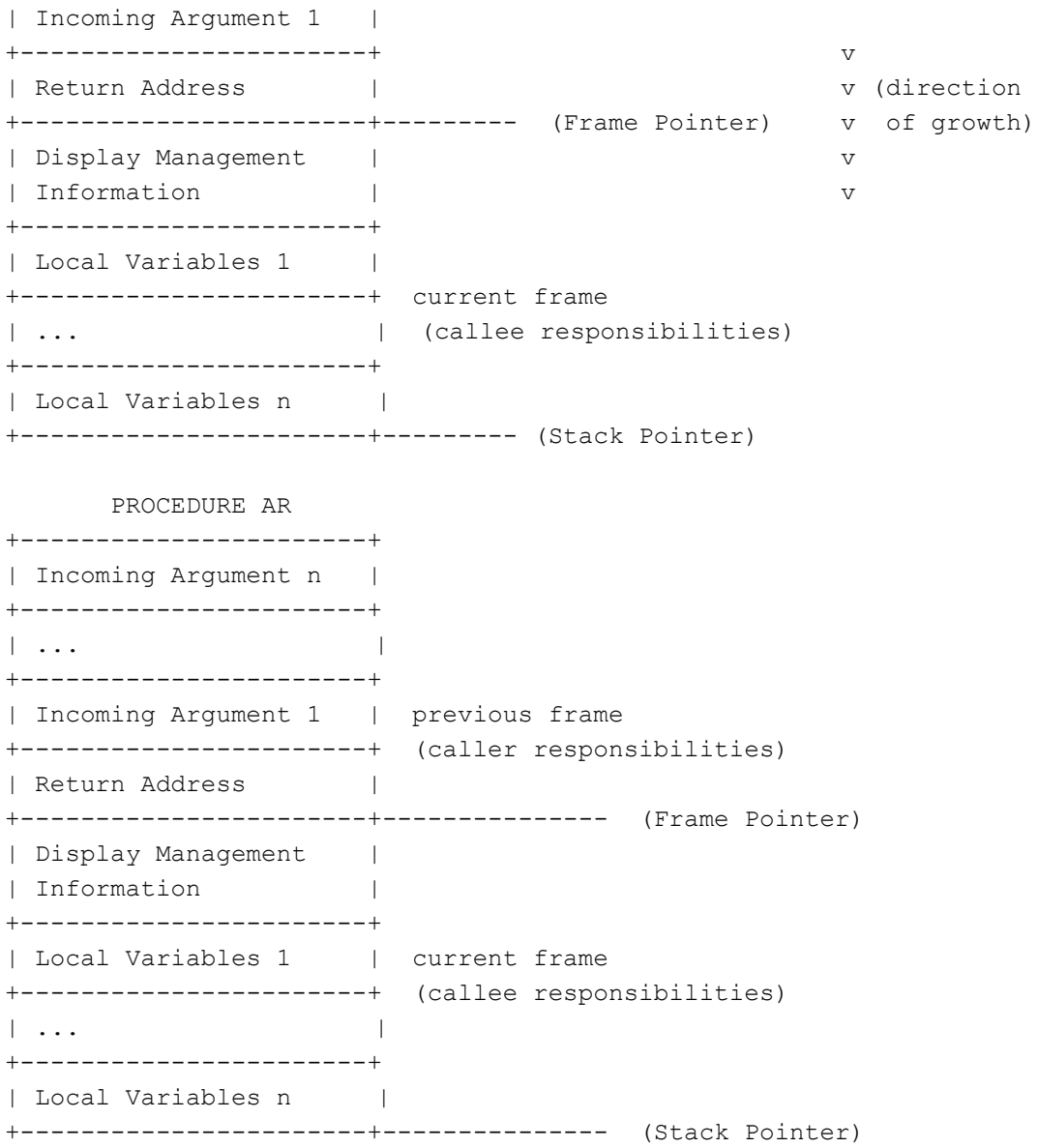
Since anonymous functions cannot be referenced later in a piece of code, they may be evaluated in-place. This means generated the appropriate code for the statement portion on the left side of the '**yields'** then subsequently the appropriate code for the expression portion on the right side of the **'yields'** with the result of the expression being pushed atop the stack as the result of the function.

3. Functions and Procedures

a. Activation Record for Functions and Procedures

Note: The direction of growth is downwards in these diagrams.





b. Procedure and Function Enter Code

The calling sequences for function or procedure setup that build the activation record are the combination of responsibilities from the caller and the callee.

In caller responsibilities, values such as return value, incoming arguments and return address need to be pushed to the stack before call instructions. The routine that carries out such process is described below.

First we must create a space for the return value. A return value needs to be always at the bottom of the stack frame so the caller is able to make use of it when the call instructions and its execution are done.

Next, spaces for arguments need to be allocated and pushed to the stack in reverse order. That is, the first argument would be closer to the top of the stack and the last argument would be closer to the bottom of the stack. This is done to improve the accessibility of these values from the display register. Details about parameter passing is discussed in part d.

Then we store the return address at the location of the current program counter (current address of the instruction being executed) so the callee can navigate back to the caller when it has finished its execution and has been taken down. This is implemented by keeping track of the address of the next instructions upon making the function call and writing the instructions into the memory.

At this point the caller responsibilities are done and the callee responsibilities commence.

The first in the callee responsibilities is to mark the beginning of the a new frame. Therefore we convert the old stack pointer into a new frame pointer and we push the callee register on the stack as well.

Then a display management information is stored here, and an explanation of this can be found in part g.

Finally, we allocate spaces for local variables. Unlike the arguments, local variables are stored in the same order that they are declared in. This is also done to improve the accessibility for the display register. In case of minor scopes, local variables are pushed in the same manner. However when the minor scope has ended, all variables declared within the scope will be popped from the stack to make more space for future allocations. That being said, two minor scopes cannot coexist on the stack.

The entrance code can be found below…

```
-----------------------------------------------------------------------
/* Function Entrance (caller responsibilities) */

PUSH 0                // return value

PUSH 'argument n'     // push arguments in reverse order
...
PUSH 'argument 1'

PUSH 'current program counter'  // return address

PUSH 'callee address'           // push callee address
```

```
BR                              // jump to callee

/* Function Entrance (callee responsibilities) */

ADDR LL 0       // save display register

PUSHMT
SETD LL         // set display register as new AR

PUSH 'local variable 1'    // push local variables onto the stack
...
PUSH 'local variable n'
```

Procedure entrance code is very much similar to the function entrance code but without the return value.

```
/* Procedure Entrance (caller responsibilities) */

PUSH 'argument n'     // push arguments in reverse order
...
PUSH 'argument 1'

PUSH 'current program counter'  // push return address

PUSH 'callee address'           // push callee address
BR                              // jump to callee

/* Procedure Entrance (callee responsibilities) */

ADDR LL 0       // save display register

PUSHMT
SETD LL              // set display register value pointing to start
of AR

PUSH 'local variable 1'    // push local variables onto the stack
...
PUSH 'local variable n'
```

---

c. Procedure and Function Exit Code

The procedure and function exit code will be very identical to the entrance code but in a reverse order. Therefore we need to first deal with the epilogue from callee responsibilities and work its way up to the caller responsibilities.

First, since the local variables are at the top of the stack, they will be deallocated from the stack but it is important to pop them reversely. In this case, local variable n will be popped first and local variable will be popped last. However, if we know the number of variable stored on the stack, then we can use POPN to pop all the local variables or other values at once. This can be done by obtaining a global variable during the code generation to keep track of the total arguments.

After local variables, display management information will be deallocated.

Then the next is the return address. Here we want to pop the return address and branch back to the caller.

Finally we deallocate all the arguments on the stack by using the same method as the local variables.

Keep in mind that return value will remain on the stack so the caller will have access to this value.

the exit code can be found below…

```
/* Function Exit (callee responsibilities) */

PUSH n          // n = # of local variables

POPN            // pop first n entries on the stack

SETD LL         // restore display register

BR              // pop return address and go there

/* Function Exit (caller responsibilities) */

PUSH m          // m = # of arguments

POPN            // pop first m entries on the stack
```

The procedure exit will be identical to the function exit.

d. Parameter passing

As described in part b, all arguments will be handled and pushed on to the stack by the caller. And this is done in a reverse order so the display register can faster navigate the desired variable according to the offset. More detailed description about display register can be found in part g.

According to the source language definition, an argument is an expression so parameters need to be first evaluated before they are pushed onto the stack. Therefore we need to generate code that evaluates the expressions. The activation record design allows all arguments to be evaluated and pushed to the stack before storing the return address. This way, possible errors in the argument expression will be catched before pushing more values onto the stack. More detailed description about how expressions are evaluated can be found in the expression section.

e. Function call and function value return.

The setup and takedown of a function call are shown in part b and c. Please refer to those parts for the code implementation.

According to our activation record template, a function's return value is at the very bottom of the stack and therefore it will the very last element to be on top in a function call. Therefore the implementation of a function value return is simply just leaving the the return value on the stack and restore the caller register.

f. Procedure call

The setup and takedown of a procedure are shown in part b and c. Please refer to those parts for the implementation.

g. Display management strategy

Display registers are the pointers to the start of each called stack frame and they are updated automatically as procedures entered or exited. We use display registers to navigate local variables and arguments at different lexical levels and this is extremely useful when dealing with nested functions and procedures, which our source language supports. Display registers are placed before local variables and after incoming arguments to allow faster search for such variable but knowing the offsets.

For example, a function f(argument 1, argument 2) at a lexical level of 2 is called with an activation record shown below.

```
+-----------------------+
| Argument 2            |  D[2,-3]
+-----------------------+
| Argument 1            |  D[2,-2]
+-----------------------+
| Return Address        |  D[2,-1]
+-----------------------+
| Display Management    |  D[2,0]
| Information           |
+-----------------------+
| Local Variable 1      |  D[2,1]
+-----------------------+
| Local Variable 2      |  D[2,2]
+-----------------------+
| Local Variable 3      |  D[2,3]
+-----------------------+
```

Because the return address is always store above the display management information, D[n,1] is always the value of display register. Therefore if we were to look for argument 1 from function f, then we simply called D[2,2] because we know function f is at lexical level of 2 and the first argument of f will be store right after its display register.

Since local variables are always stored right after the display management information, if we want to find the first local variable declared in the function, we can do it by calling D[2,-1]. And the same logic applies to the rest of the local variables.

4. Statements

a. Assignment Statement

In order to generate code for the assignment statement, code for the left and right side will be generated separately. This design is drawn heavily from section 11.3.5 (page 429) of the textbook.

First, a different type of visitor, LHSVisitor is used to evaluate the left hand side of the assignment statement (the variable that a value is being assigned to). This is because different actions will need to be done to get the address of the variable being assigned (it can ADular local variable, function parameter or even array element). After this step, the address for the variable will be at the top of the stack.

Next, the right hand side expression is evaluated, and it's result will be at the top of the stack.

Finally, we emit a STORE command to store the value.

The following is example pseudocode:

```
procedure visit(AssignStmt a)
      a.getLHS().accept(new LHSVisitor(this))
      a.getRHS().accept(this)
      emit STORE
end
```

The following is an example for the statement `a <= 5 + 1`

```
ADDR 2 10 % Assume a is at lexical level 2 and offset 10
PUSH 5
PUSH 1
ADD
STORE
```

b. If Statements

1. Emit the code which evaluates the expression. The final result of the expression will be at the top of a stack.
2. Emit a branch and save the address for this branch.
3. Emit the code for the rest of the statement.
    a. If there is an else branch then emit the unconditional branch, and save the address of the line where this beings.
4. Fill in the address saved in 2 now that the conditional is over.
    a. If there is an else branch then emit the statements for the else branch
    b. Fill in the address saved in 3a.

The following is an example without an else branch:

```
      … code for expression …
1     PUSH X % placeholder for branch statement to end
2     BF
      … code for statements …
10    … out of if statement now …
```

At line 10 the if statement is complete. So, we go back to line 1 and fill in with the address. The final result is:

```
      … code for expression …
1     PUSH 10
2     BF
      … code for statements …
10    … out of if statement now …
```

The following is an example of an if statement with an else branch:

```
     … code for expression …
1    PUSH X % placeholder for branch to else
2    BF
     … code for statements if condition held …
10   PUSH X % placeholder for branch to end of else
11   BR
     … else statements …
20   … else is complete …
```

Now that the if-then-else conditional is complete, we can go back and patch in the branches on line 10 and 1. Line 1 will jump to line 12, where the else branch begins, and line 10 will be filled in with line 20 where the else branch is complete.

```
     … code for expression …
1    PUSH 12
2    BF
     … code for statements if condition held …
10   PUSH 20
11   BR
     … else statements …
20   … else is complete …
```

c. Loop Statements

While Loop

1. Start with the ‘expression’ code and then save the address of the first line for backward branch later.
2. Emit `BF` and then save this address.
3. Emit while loop body, if there are any exits or ‘exit when’ statements, save the lines where the address for the branches are pushed.
4. Emit a branch back to 1.
5. Patch up addresses in 1, 2, 3.

The following is an example of a while loop

```
1    … expression code … %save address where this begins
     …
10   PUSH X % placeholder for end of loop
11   BF
     … While loop body statements …
20   Push 1
21   BR
22   … Loop ends here …
```

So now that the while loop has been written, we can go back and patch line 10 with the address of line 22. This is because if the conditional evaluates to false, then the program counter will jump to line 22, therefore circumventing the loop statements.

```
1     … expression code …
      …
10    PUSH 22
11    BF
      … While loop body statements …
20    Push 1
21    BR
22    … Loop ends here …
```

Loop

1. Emit the statement inside the loop, save address of the first line for backward branch later.
2. Emit a branch back to the address saved in 1.
3. Go back and patch all ‘exit’ statements or ‘exit when’ statements.

The following is an example of a loop statement

```
1     … loop body statements …
      …
10    PUSH 1
11    BR
12    … loop ends here …
```

In this case, if there were any ‘exit’ or ‘exit when’ statements, they would be patched with 12, because this is where the loop has just ended.

d. Exit and Exit When Statements

Exit Statements

For the regular exit statements, we do the following

```
1     PUSH X
2     BR
```

Where X will later be patched at the address where the code for the loop has just ended. In other words, the next statement after the loop.

Exit When Statements

For exit when statements, the expression will be evaluated as normal, and then a negation will be emitted. Following the negation a PUSH for the address of where the loop has just ended will be emitted. During generation, the address of this statement will need to be saved. Then a branch false, BF, is emitted.

```
1     … emit code for expression …
2     … emit boolean negation …
3     PUSH X
4     BF
```

As aforementioned, X will be patched with where the loop has just ended. Just like in the normal exit statement.

e. Return Statements

When a return statement is encountered, the teardown code for a procedure or function is emitted, depending on whether it is a procedure or function, respectively. Please refer to section c in the Functions and Procedure segment of this document for the return code.

f. Get and Put Statements

Get Statements

According to the specification, only integers can be read from standard input in this language. So, for each input, the LHSVisitor will be used to emit the code to retrieve the address of the variable (just like in **assignment statement**), followed by a READI and then STORE.

Example Machine Code:

```
1     … code to get address/reference for variable …
      …
5     READI
6     STORE
```

Example Pseudocode:

```
procedure visit(AssignStmt a)
      a.getLHS().accept(new LHSVisitor(this))
      emit READI
      emit STORE
end
```

Put Statements

There are three types of outputs for a Put statement: integer expresison, text and skips. The following are three different templates that will be used depending on the type of output.

If the output is an integer, then code will be emitted for evaluating the expression, followed by a PRINTI.
Example:

```
1     … code for evaluating expression …
      …
5     PRINTI
```

If the output is text, then for each character, code will be emitted where the ASCII code for the character is pushed onto the stack followed by PRINTC. The text constant will be evaluated from left to right and for each character, the following will be emitted:

```
1     PUSH A
2     PRINTC
```

Where A is the ASCII code for the character. The reasoning for reading it from right to left is because the stack operates on a last in first out basis.

If the output is a ‘skip’ then the ASCII code for the newline character will be pushed followed by an emitted PRINTC. 10 is the ASCII code for a new line.

```
1     PUSH 10
2     PRINTC
```

Other

a. main program initialization and termination

When the program is first initialized a few things need to be done by the interface for the pseudo-machine. Machine.SetPC must be called and passed the address of the first instruction generated by the machine. Machine.setMLP must be set to the address of the word immediately after the last word allocated on the stack. Machine.setMSP must be set to the first address after the last line of program code. As per the recommendation in the code generation rules handout we will assign these after the majority of the code generation of done.

Machine.setMSP can be set pretty easily after **HALT** is generated for termination by setting it to the address after that which **HALT** is inserted at. **Machine.setMLP** must be calculated as the program code is being generated to determine the maximum extent of the runtime stack. **Machine.SetPC** on the other hand can be set before code generation begins to the address where the first line is intended to go.

Machine.SetPC can start at address 0.
Machine.setMSP is set to the final address value of the **HALT**.
Machine.setMLP is set to **Machine.memorySize - 1**.

A **HALT** instruction will be added to the end of the generated code before the initialization variables above are set, this line will terminate the machine. **HALT** will be inserted after the **end** that closes the program's main scope.

Sample Programs

A4-01.488

```
% 1-2
PUSHMT
SETD 0

% 1 - 3
% integer i addr 0 1
%              j addr 0 2
%              k addr 0 3
%              l addr 0 4
%              m addr 0 5

PUSH 0
PUSH 5
DUPN % we push 5 values onto the stack to reserve space for i,j,k,l,m

% 1 - 4
ADDR 0 3     % k
ADDR 0 1     % i
LOAD
PUSH 3
ADDR         % i + 3
ADDR 0 2     % j
LOAD
ADDR 0 3     %k
LOAD
MUL          % j * k
SUB          % (i+3) - j*k
ADDR 0 3     % k
LOAD
ADDR 0 4     % l
LOAD
DIV          % k/l
ADD          %(i+3) - j*k + k/l
STORE

% 1 - 5
% boolean    p addr 0 6
%                  q addr 0 7
%                  r addr 0 8
```

```
%                   s addr 0 9
%                   t addr 0 10
PUSH 0
PUSH 5
DUPN

% 1 - 6
ADDR 0 8

ADDR 0 6
PUSH 1
STORE

% 1 - 7
ADDR 0 7
PUSH 0
STORE

% 1 - 8
ADDR 0 7
LOAD
PUSH 1
SUB   % ! q

ADDR 0 6
LOAD
ADDR 0 7
LOAD
OR          % ( p | q )

OR          % ! q | ( p | q )

% evaluated AND remember we use deMorgan's
ADDR 0 9
LOAD  % s
PUSH 1
SUB
ADDR 0 6 % p
LOAD
PUSH 1
SUB         % ! p
PUSH 1
SUB
OR
PUSH 1
SUB         % (s & ! p)

OR % ! q | ( p | q ) | (s & ! p)
```

```
STORE

% 1 - 9
ADDR 0 6     % p

ADDR 0 1
LOAD
ADDR 0 2
LOAD
LT                % ( i < j)

ADDR 0 3     % k
LOAD
ADDR 0 4     % l
LOAD
LT
ADDR 0 3     % k
LOAD
ADDR 0 4     % l
LOAD
EQ
OR                % ( k <= l )

OR                % ( i < j    ) | ( k <= l )

ADDR 0 2
LOAD
ADDR 0 4
LOAD
EQ                % ( j = l )

OR                % ( i < j    ) | ( k <= l ) | ( j = l )

STORE

% 1 - 10
ADDR 0 9

ADDR 0 3
LOAD
ADDR 0  5
LOAD
EQ
PUSH 1
SUB          % (k !- m)
PUSH 1
SUB
```

```
ADDR 0 2
LOAD
ADDR 0 3
LOAD
LT
PUSH 1
SUB                % ( j >= k )
PUSH 1
SUB

OR
PUSH 1
SUB                % ( k ! = m ) & ( j >= k )

PUSH 1
SUB

ADDR 0 8
LOAD
ADDR 0 9
LOAD
EQ
PUSH 1
SUB                % ! ( r = s )

PUSH 1
SUB

OR
PUSH 1
SUB          % ( k ! = m ) & ( j >= k ) & ! ( r = s )

STORE
```

% 1 - 11

```
ADDR 0 7

ADDR 0 8
LOAD
ADDR 0 9
LOAD
EQ                 % r = s

ADDR 0 9
LOAD
PUSH 1
SUB                % ! s
```

```
ADDR 0 8
LOAD
EQ                % r
PUSH 1
SUB               % ( !s != r )

OR                % ( r = s    ) | ( !s != r )

STORE

% 1-12
PUSH 0
PUSH 7
DUPN         % A[7]

PUSH 0
PUSH 150
PUSH 10
MULT
DUPN         % B[-100..50, -40..-30]

% 1- 13
PUSH 0
PUSH 4
DUPN         % C[-7 .. -3]

PUSH 50
PUSH 20
MULT
DUPN         %  D[ 50, 20 ]

% 1-14
ADDR 0 18    % base address of B
ADDR 0 1
LOAD         % i
PUSH 1
ADD
PUSH -100
SUB
PUSH 10
MULT         % got offset of the outer dimension
ADDR 0 2
LOAD         %j
PUSH 100
SUB
PUSH -40
SUB               % offset of the innter dimension
ADD          % offset within the entire array
```

```
ADD          % final address of B[i+1, j-100 ]

ADDR 0 11    % base address of A
ADDR 0 2
LOAD         % j
PUSH 2
SUB               % j - 2
ADD          % final address of A[j-2]
LOAD
STORE

% 1-15
ADDR 0 1518
PUSH -4
PUSH -7
SUB
ADD          % got the address of C[-4]

ADDR 0 1523
ADDR 0 1
LOAD
PUSH 20
ADD          % i + 20
PUSH 20
MULT
ADDR 0 3
LOAD
PUSH 7
SUB               % k - 7
ADD          % got offset within array
ADD          % got final offset
LOAD
STORE
HALT
```

A4-02.488

```
PUSHMT
SETD 0

PUSH 0
PUSH 3
DUPN % we push 3 zeros onto the stack as default values for a,b,c

PUSH 0
PUSH 3
DUPN % likewise for p, q, r
```

```
ADDR 0 5 % begin if
LOAD % push value of p on top of stack
ADDR 0 4
LOAD % push value of p on top of stack
OR
PUSH 22 % push address to end of if
BF % branch to end of if, if condition false
ADDR 0 1
PUSH 3
STORE % assign 3 to a at LL 0, end if

ADDR 0 6 %begin if
LOAD % push value of q on top of the stack
PUSH 1
SUB % negate q
PUSH 1
SUB % negate q again for demorgan's
ADDR 0 6
LOAD % push value of q on top of the stack
PUSH 1
SUB % negate q
PUSH 1
SUB % negate q again for demorgan's
OR
NOT % A
PUSH 44 % push address to (start of) else
BF % branch to else, if condition false
ADDR 0 2
PUSH 2
STORE % assign 2 to b at LL 0
PUSH 47 % branch to end of else
BR
ADDR 0 2 % begin else
PUSH 0
STORE % assign 0 to b at LL 0
% end else, end if

PUSH 7 % begin while
ADDR 0 3
LOAD % push value of c on top of the stack
LT
PUSH 60 % push label for end of while
BF % branch to end while
ADDR 0 3 % begin else
PUSH 8
STORE % assign 8 to c at LL 0
PUSH 49
BR % Branch back to begin while label
```

```
%end while

ADDR 0 1 % begin loop
PUSH 3
STORE % assign 3 to a at LL 0
PUSH 70 % push label for end loop
BR %branch to end loop
ADDR 0 2
PUSH 7
STORE % assign 7 to b at LL 0
PUSH 62 % push label for begin loop
BR % branch to begin loop
%end loop

ADDR 0 4 %begin while
LOAD % push value of p on top of the stack
PUSH 1
SUB % negate p
PUSH 1
SUB % negate p again for demorgan's
ADDR 0 5
LOAD % push value of q on top of the stack
ADDR 0 6
LOAD % push value of r on top of the stack
OR
PUSH 1
SUB % negate and operand for demorgan's
OR
PUSH 1
SUB % complete de morgan's
PUSH 98 % push label for end while
BF
PUSH 10
ADDR 0 2
LOAD % push value of b on top of the stack
EQ
PUSH 1
SUB
PUSH 1
SUB % evaluate the negation of the negation of b = 10
PUSH 98 % push label for end
BF
PUSH 74
BR % Branch back to begin while label
%end while

PUSH 86 % Push "V"
PRINTC
```

```
PUSH 97 % Push "a"
PRINTC
PUSH 108 % Push "l"
PRINTC
PUSH 117 % Push "u"
PRINTC
PUSH 101 % Push "e"
PRINTC
PUSH 32 % Push space
PRINTC
PUSH 105 % Push "i"
PRINTC
PUSH 115 % Push "s"
PUSH 32 % Push space
PRINTC
ADDR 0 2
LOAD % load b
ADDR 0 1
LOAD % load a
DIV
PRINTI % print a/b
PUSH 32 % Push space
PRINTC
PUSH 111 % Push "o"
PRINTC
PUSH 114 % Push "r"
PRINTC
PUSH 32 % Push space
PRINTC
ADDR 0 3
LOAD % load c
NEG
ADDR 0 2
LOAD
MUL
PRINTI % print b * -c
PUSH 10 %PUSH newline for skip
PRINTC

ADDR 0 1 % get address for a
READI
STORE % assign get result to a
ADDR 0 2 % get address for b
READI
STORE % assign get result to b
ADDR 0 3 % get address for c
READI
STORE % assign get result to c
```

```
PUSH 0
PUSH 3
DUPN % we push 3 zeros onto the stack as default values for m, n, c
ADDR 1 1 % prepare address of m for assignment
ADDR 0 2
LOAD % load b from LL 0
PUSH 7
SUB
ADDR 1 3
LOAD % load c from LL 1
ADD
STORE % store result to m

PUSH 0
PUSH 3
DUPN % we push 3 zeros onto the stack as default values for p, q, r
ADDR 2 1 % prepare address of p for assignment
ADDR 2 1 % prepare address of p for assignment
ADDR 0 1
LOAD % load a from LL 0
STORE % save p as a
ADDR 0 2
LOAD % load b from LL 0
PUSH
ADDR 2 3
LOAD % load r from LL 2
SUB
STORE % store result of anonymous function to p

ADDR 0 5 %begin while
LOAD % load q from LL 0
ADDR 0 4
LOAD % load p from LL 0
OR
PUSH 1
SUB % negate expression for while
PUSH 254 % push label for end while
BF
ADDR 0 6
LOAD % load p from LL 0
PUSH 1
SUB % negate p for demorgan's
ADDR 0 4
LOAD % load r from LL 0
PUSH 1
SUB % negate r for de morgan's
OR
```

```
PUSH 1
SUB % complete demorgan's for exit when
PUSH 1
SUB % negate expression for exit when
PUSH 254 % push label for end while
BF
PUSH 0 % begin loop
PUSH 2
DUPN % we push 2 zeros onto the stack as default values for w, x
ADDR 2 1 % begin if
LOAD % load w from LL 0
ADDR 0 1
LOAD % load a from LL 0
LT % check a < w
PUSH 1
SUB % negate for w <= a
PUSH 245 % push label for end if
BF
PUSH 251 % push label for end loop
BR
PUSH 0
PUSH 2
DUPN % we push 2 zeros onto the stack as default values for t, u
ADDR 2 3 % prepare address of t for assignment
PUSH 0
ADDR 3 1 % prepare address of m for assignment
ADDR 2 3
LOAD % load t from LL 2
ADDR 2 1
LOAD % load w from LL 2
LT
STORE % store result to m
PUSH 245 % begin if, push label for end if
BF
ADDR 2 3 % prepare address of t for assignment
ADDR 0 3
LOAD % load c from LL 0
ADDR 2 3
LOAD % load t from LL 2
ADD
STORE % store result to t
%end if
ADDR 2 3
LOAD % load t from LL 2 for yield
STORE % store result of anonymous function to t
PUSH 224
BR
%end loop
```

```
PUSH 186 % push label for begin while
BR
%end while

HALT
```

A4-03.488

```
%3-2
ADDR 0 0                % save D[0]
PUSHMT
SETD 0

%3-3
PUSH 0
PUSH 4
DUPN

%3-4
PUSH false
PUSH 4
DUPN

%3-5
PUSH PROC_P_END         % skip function/procedure setup
BR

(PROC_P address)------------------------------------------------------

%3-7
ADDR 1 0                % save D[1]
PUSHMT
SETD 1

PUSH 0
PUSH 2
DUPN                    % e & f

%3-8
ADDR 0 5
LOAD                    % load p
PUSH IF_P_END           % if p is false, go to end of if
BF
PUSH IF_P_END           % no content, go directly to end of if
BR
% (IF_P_END address)
```

```
%3-9
ADDR 0 1                % a
ADDR 1 1                % e
STORE                   % e <= a

%3-11
PUSH 2
POPN                    % deallocate e & f

SETD 1                        % restore D[1]


(PROC_P_END address)-----------------------------------------------------

%3-12
PUSH FUNC_F_END               % skip function/procedure setup
BR

(FUNC_F address)---------------------------------------------------------

%3-14
ADDR 1 0                % save D[1]
PUSHMT
SETD 1

ADDR 1 -1               % n
LOAD
PUSH IF_F_ELSE
BF                            % if n is false, go to then
ADDR 0 2                % b
LOAD
ADDR 1 -2               % m
LOAD
ADD                     % m + b
STORE
PUSH IF_F_END
BR

%3-15
% (IF_F_ELSE address)
ADDR 1 -2               % m
LOAD
ADDR 0 3                % c
LOAD
SUB                     % c - m
STORE
PUSH IF_F_END
BR
```

```
% (IF_F_END address)

SETD 1                          % restore D[1]

(FUNC_F_END address)-----------------------------------------------------

%3-17
PUSH PROC_Q_END          % skip function/procedure setup
BR

(PROC_Q address)---------------------------------------------------------

%3-19
ADDR 1 0           % save D[1]
PUSHMT
SETD 1

PUSH 0
PUSH 3
DUPN               % t, u, v

%3-20
PUSH FUNC_G_END          % skip function/procedure setup
BR

(FUNC_G address)---------------------------------------------------------

%3-22
ADDR 2 0           % save D[2]
PUSHMT
SETD 2

PUSH 0
PUSH 2
DUPN               %  w, x

% 3-24
PUSH 0
PUSH 2
DUPN               % b, d
ADDR 0 3           % c
ADDR 2 4           % d
STORE              % d <= c

% (FUNC_G_RETURN address)
PUSH FUNC_G_RETURN
PUSH PROC_P
BR                       % P
```

```
%3-25
PUSH 12
ADDR 1 3          % v
LOAD
ADDR 2 1          % w
LOAD
ADDR 0 4          % d
LOAD
ADD               % d + w
SUB               % (d + w) - v
LT                      % d + w - v  < 12

PUSH 4
POPN              % deallocate w,x,b,d

SETD 2                  % restore D[2]

(FUNC_G_END address)-------------------------------------------------

%3-27
% (FUNC_Q_RETURN address)
PUSH FUNC_Q_RETURN
PUSH FUNC_G
BR
NEG
PUSH 1
ADD               % !G
ADDR 0 7          % r
LOAD
OR                      % (!G | r)

PUSH 0
ADDR 0 1          % a
LOAD
ADDR 1 1          % t
LOAD
ADDR 1 -2         % n
LOAD
SUB               % n - t
ADD               % (n - t) + a


% (FUNC_Q_RETURN2 address)
PUSH FUNC_Q_RETURN2
PUSH FUNC_F
BR
```

```
PUSH 2
POPN              % pop 2 arguments
PRINTC                  % ouput result of F( t - n + a  , ! G | r )

PUSH 3
POPN              % deallocate t, u, v

SETD 1                  % restore D[1]

(PROC_Q_END address)-----------------------------------------------------

% 3-29
PUSH 0                  % return value of Q

% (RETURN_ADDR4)
PUSH RETURN_ADDR4
PUSH PROC_P
BR                            % P

% 3-30
ADDR 0 5                % p
LOAD
ADDR 0 6                % q
LOAD
EQ
NEG
PUSH 1
ADD                     % (p != q) => !(p == q)

%3-29
PUSH 0                        % return value of outer F
ADDR 0 6                % q
LOAD
NEG
PUSH 1
ADD                     % !q
PUSH 0                        % return value of inner F
ADDR 0 5                % p
LOAD
ADDR 0 4                % b
LOAD
% (RETURN_ADDR)
PUSH RETURN_DDR
PUSH FUNC_F
BR                      % F(b,p)
% (RETURN_ADDR2)
PUSH RETURN_ADDR2
PUSH FUNC_F
```

```
BR                              % F(F(b,p), !q)
ADDR 0 5                  % p
LOAD
NEG
PUSH 1
ADD                       % !p
ADDR 0 6                  % q
LOAD
OR                        % p | q
% (RETURN_ADDR3)
PUSH RETURN_ADDR3
PUSH FUNC_Q
BR                        % Q( ! p | q , F( F( b, p ), !q ), { P yields p != q }
)

PUSH 8
POPN                      % deallocate a,b,c,d,p,q,r,s

SETD 0                    % restore D[0]

HALT
```

Distribution of Labour:

Storage	-	**g4snyder**
Expressions	-	**g2graman**
Functions and Procedures	-	**g3chencm**
Statements	-	**g2yeungw**
Other	-	**g4snyder**

A4-01	-	**g2yeungw**
A4-02	-	**g2graman**
A3-03	-	**g3chencm**

Collectively, we all thoroughly reviewed the document and made changes and edits to each other's work after doing the initial write up of the individual sections.