1. Storage

The amount of storage required for variables in the main program, procedures, functions, and minor scopes will be determined during semantic analysis. The symbol table will contain entries for every variable in every scope and from that we can determine the amount of storage to allocate.

If we need to allocate storage for *N* words in memory for these variables, we can do so by pushing *N* ‘undefined’ values onto the stack.

```
PUSH UNDEFINED
PUSH N
DUPN
```

Integer, boolean, and text constants don’t require storage to be allocated to them in advance and can be pushed onto the stack as needed.

Given an integer constant *I*

```
PUSH I
```

Given a boolean constant with value ‘true’

```
PUSH MACHINE_TRUE
```

Given a boolean constant with value ‘false’

```
PUSH MACHINE_FALSE
```

Given a text constant, we push each character *C*

```
PUSH C
```

2. Expressions

Since values of constants are stored by pushing them onto the stack, we can access them by popping them off the stack.

To access values of scalar variables, we can compute the address at which the value is stored and then load in the value and put it at the top of the stack given the lexical level *LL* and the offset *ON* at which the value is stored:

```
ADDR LL ON
LOAD
```

Array Access

To access elements of an array, $A[L_1 : U_1, L_2 : U_2, \ldots, L_n : U_n]$, we first compute the base address, *BASE,* given the lexical level *LL* and the offset *ON* at which the array is stored.

```
ADDR LL ON  // At the top of the stack we now have BASE
```

Since the arrays are arranged in row-major order we can efficiently access an individual array element, $A[E_1, E_2, \ldots E_n]$, by calculating the address using Horners rule:

Given

$stride_i = U_i - L_i + 1$

$const\_part = ((\ldots((L_1 * stride_2 + L_2) * stride_3 + \ldots L_{n-1}) * stride_n + L_n$

$var\_part = ((\ldots((E_1 * stride_2 + E_2) * stride_3 + \ldots E_{n-1}) * stride_n + E_n$

Then

$addr(A[E_1, E_2, \ldots E_n]) = base - const\_part + var\_part$

From the symbol table we will be able to fetch the lower bounds (L_i), upper bounds (U_i), and size of the elements (*SIZE*) Given $L_1, U_1, L_2, U_2, \ldots, L_n, U_n$

Let $STRIDE_i$ be the following steps:

```
PUSH Ui
PUSH Li
SUB
PUSH 1
ADD
```

Let *CONST_PART* be the following steps:

```
PUSH L1
STRIDE2
MUL
```

```
PUSH L2
ADD
STRIDE3
MUL
PUSH L3
ADD
...

STRIDEn
MUL
PUSH Ln
ADD
PUSH SIZE
MUL
```

Then $addr(A[E_1, E_2, ... E_n])$ is the following

```
PUSH E1
STRIDE2
MUL
PUSH E2
ADD
STRIDE3
MUL
PUSH E3
ADD
...

STRIDEn
MUL
PUSH En
ADD
PUSH SIZE
MUL            // At the top of the stack we now have var_part then base
CONST_PART     // Top of the stack: const_part, var_part, base
SUB            // Top of the stack: var_part − const_part, base
ADD            // Top of the stack: base + var_part − const_part
LOAD
```

To access an individual array element of $A[E_1]$ of an array, $A[U_1]$, we do the following steps given the *SIZE* of elements in *A*:

```
ADDR LL ON   // At the top of the stack we now have BASE of A
PUSH E1
PUSH SIZE
MUL          // Calculate VAR_PART in Horners algorithm
ADD          // Base - 0 + VAR_PART, CONST_PART is 0 because L1 = 0
LOAD
```

To access an individual array element of $A[E_1]$ of an array, $A[L_1 : U_1]$, we do the following steps given the *SIZE* of elements in *A*:

```
ADDR LL ON   // At the top of the stack we now have BASE of A
PUSH E1
PUSH L1
SUB
PUSH SIZE
MUL          // Calculate VAR_PART - CONST_PART
ADD          // Calculate BASE + VAR_PART - CONST_PART
```

Arithmetic Operators

There are four arithmetic operators: +, -, *, and / . For each of these, the scheme that will be used is to [recursively] evaluate the expressions on the left- and right-hand sides of the operator, and then -- using these evaluated operands -- combine them using the machine instructions that map directly to the arithmetic operators.

Specifically, `evaluate` is a representation of a recursive evaluation of an expression, and this computed value being pushed on to the `msp` stack. This is necessary for these operators as they are non-trivial and cannot be represented purely in base machine instructions. These operators take in integers as operands, and evaluate to integers.

Expr1 + Expr2: Addition

```
evaluate Expr1
evaluate Expr2
ADD
```

Expr1 - Expr2: Subtraction

```
evaluate Expr1
evaluate Expr2
SUB
```

- Expr: Unary Minus

```
evaluate Expr
NEG
```

Expr1 * Expr2: Multiplication

```
evaluate Expr1
evaluate Expr2
MUL
```

Expr1 / Expr2: Division

```
evaluate Expr1
evaluate Expr2
DIV
```

<u>Comparison Operators</u>

As with the arithmetic operators, the comparison operators require recursive evaluation of the expressions, which are then combined with various base machine instructions. These operators take in integers as operands, and evaluate to booleans. Here, `evaluate` represents the same as what it represented for arithmetic operators.

Expr1 < Expr2: Less-than

```
evaluate Expr1
evaluate Expr2
LT
```

Expr1 <= Expr2: Less-than-or-equal-to

```
evaluate Expr1
evaluate Expr2
SWAP
LT
NOT          // since Expr1 <= Expr2 is the same as:  NOT(Expr2 < Expr1)
```

Expr1 = Expr2: Equal-to

```
evaluate Expr1
```

```
evaluate Expr2
EQ
```

Expr1 != Expr2: Not-equal-to

```
evaluate Expr1
evaluate Expr2
EQ
NOT
```

Expr1 >= Expr2: Greater-than-or-equal-to

```
evaluate Expr1
evaluate Expr2
LT
NOT          // since Expr1 >= Expr2 is the same as:  NOT(Expr1 < Expr2)
```

Expr1 > Expr2: Greater-than

```
evaluate Expr1
evaluate Expr2
SWAP
LT           // since Expr1 > Expr2 is the same as: Expr2 < Expr1
```

<u>Boolean Operators</u>

Once again, we make use of the recursive evaluation of the expressions using `evaluate`. These operators take in booleans (either 1 or 2) as operands, and return booleans.

Expr1 & Expr2: Binary And

- Note: For this operator, because there is no AND machine instruction, we instead make use of DeMorgan's Law: that is, a && b == ¬ (¬ a || ¬ b)

```
evaluate Expr1
PUSH MACHINE_FALSE
EQ           // after this EQ, the msp stack's top is: !Expr1
evaluate Expr2
PUSH MACHINE_FALSE
EQ           // after this EQ, the msp stack's top is: !Expr2 , !Expr1     (!Expr2 is top)
OR           // after this OR, the msp stack's top is: !Expr1 | !Expr2
PUSH MACHINE_FALSE
EQ           // after this EQ, the msp stack's top is: !(!Expr1 | !Expr2) = (Expr1 & Expr2)
```

Expr1 | Expr2: Binary Or

```
evaluate Expr1
evaluate Expr2
OR
```

!Expr: Unary Not

```
evaluate Expr
PUSH MACHINE_FALSE
EQ
```

<u>Conditional Expressions</u>

Here, we must make use of branching to ensure that we generate the correct code for conditional expressions. We will make use of two branches: one for the expression/statements in the “else” portion of the conditional expression, and one for the portion of the code *after* the conditional.

Thus, we generate as below for a conditional of the form:

(*cond ? expr_if_true : expr_if_false)*

```
evaluate cond
PUSH label_false
BF                          // if cond is false, we branch to the if-false/else portion
evaluate expr_if_true       // but if cond is not false, we evaluate the if-true portion
PUSH label_after            //  label for code after the conditional
BR            // we have executed the if-true portion, and we now exit the conditional
label_false      // create label if-false portion, set to current code address
evaluate expr_if_false
label_after      // create after-conditional label, set to current code address
```

3. Functions and procedures

<u>Activation record for functions and procedures</u>

1. return address
2. previous value of `display[LL]`
3. parameters
4. storage for local variables

5. storage for return value

Procedure and function entrance code

```
ADDR LL 0 // save display[LL] for restoration later
PUSHMT // set new display[LL] value
SETD LL
// push parameter values
// reserve space for local variables
PUSH UNDEFINED
PUSH N
DUPN
// if function, reserve space for return value
PUSH UNDEFINED
```

Procedure and function exit code

```
// add next 3 instructions if there is return value
// here we find the address of the caller's placeholder for the
return value and move the return value there
ADDR LL -3-N // N is the number of parameters
SWAP
STORE
PUSHMT
// remove local variables & parameters
ADDR LL 0
SUB
POPN
// restore original value of display[LL]
SETD LL
BR
```

Parameter passing

For each parameter, we evaluate it and push it onto the stack, using the offset from beginning of the display entry to index them (starting at 0).

Function call and function value return

```
PUSH UNDEFINED // placeholder for return value
PUSH return_address
PUSH proc_address
BR
// set return_address to this location
```

Procedure call

```
PUSH return_address
PUSH proc_address
BR
// set return_address to this location
```

Display management strategy

We will use the constant cost display update method where on a call from level *A* to level *B*, we will save `display[B]` in the local storage of the caller, the called procedure will set `display[B]` to the address of its activation record, and then on return it will restore the saved value of `display[B]`.

4. Statements

Assignment Statements

For an assignment expression, *Var := E,* we use the `evaluate` function once again and the given *LL* and *ON* for *Var* in the symbol table.

```
ADDR LL ON
evaluate E
STORE
```

If Statements

For an If statement, *if COND then EXPR* we do the following:

```
evaluate COND
PUSH label_false
```

```
BF                          // if cond is false, we branch to after the loop
evaluate expr_if_true       // but if cond is not false, we evaluate the if-true portion
label_false         // create label if-false portion, set to current code address
```

For an if statement, *If COND then EXP_T else EXP_F*, the steps required are equivalent to the steps described above for Conditional Expressions

While Statements

For a While statement, *while COND do STATEMENT*,

```
loop_label          // set loop_label in label table at the beginning of loop
evaluate COND
PUSH done_label
BF                  // If COND is false branch to done_label
evaluate STATEMENT
PUSH loop_label
BR                  // Branch to loop_label
done_label          // set done_label in label table at the end of loop
```

Repeat Statements

For a Repeat statement, *repeat STATEMENT until COND*,

```
loop_label          // set loop_label in label table at the beginning of loop
evaluate STATEMENT
evaluate COND       // if COND is false branch to done_label
PUSH done_label
BF
PUSH loop_label
BR                  // Branch to loop_label
done_label          // set done_label in label table at the end of loop
```

Exit Statements

For an exit statement, *exit,* when it used inside of a Repeat or While statement with a defined *done_label* we can simply execute the following steps:

```
PUSH done_label
BR
```

For an exit statement with an integer, *exit INTEGER*,

// Since INTEGER is known at compile time, push the done_label at
// (Current Loop Level - INTEGER)
```
PUSH done_label
BR
```

For an exit statement with an expression, *exit when EXP*,
```
evaluate EXP
PUSH done_label
BF
```

Return Statements

For a return statement with an expression, *return with EXP,* we execute the following

```
evaluate EXP
```

Then we exit via a Return statement as it is described in Section 3: “Procedure and function exit code” above

Similarly for a return statement with no expression, *return*, the steps to execute are also described in the same section.

Read and Write Statements

Read:
For a read statement, we process the variables given as arguments to read one by one. For each one, if the type of the variable is Integer we execute the following:

```
ADDR LL ON    % address of the variable
READI
STORE
```

Write:

For a write statement, we process the characters given as arguments one by one. For a string literal, we go through each character and print it, as follows:

```
PUSH 'C'
PRINTC
```

If we have arguments to write which are not strings, we first evaluate them and then print the results for each one:

```
evaluate arg
PRINTI
```

Handling of Minor Scopes

For minor scopes, we don’t allocate a new frame but we just treat all the variables defined inside as independent from variables outside even if they share a name. We allocate for all these variables at compile time (by pushing undefined values onto the stack) since we know exactly the amount of memory we need for them. Any variables referenced inside will refer to the variable with that name at the closest scope to the current one, and variables outside with the same name are not overridden.

5. Other

The initialization of the main program and its termination will be handled as follows:

```
PUSHMT
SETD 0
```
// emit code for major scope
```
HALT
```