# CSC488 Assignment 4

Group 10


| | | |
|---|---|---|
| Daphne Ippolito | 998400476 | g1daphne |
| Adam Robinson-Yu | 998244341 | g2robint |
| Diego Santos | 1001087138 | c4santou |
| Lisa Zhou | 998590332 | g2zhouli |


## Table of Contents

# 1. General

## 1.1 Visitor Pattern

We will continue to use the visitor pattern established during the semantic analysis assignment. We will create a *CodeGenVisitor* class which will generate code for each node in the AST. It will also call the appropriate methods to ensure that the children nodes are also visited. One ASTNode may need to generate code before, in between, and after visiting each of its children, so the sequence in which children nodes are visited needs to be carefully thought out.

When generating branch statements, we do not always know in advance what memory addresses we need to jump to. To handle this, we will do a second pass of the AST to patch branch statements. This will be handled by the *BranchPatchVisitor.* During the first pass, *CodeGenVisitor* will do two things to help *BranchPatchVisitor*; it will store on the AST node the location of the assembly lines that need to be patched, and it will store any line numbers (memory addresses) that it may need to reference. During the second pass, we will enforce that each line requiring a patch is assigned one of the referable line numbers. This will be discussed in more detail in control-flow statements [Section 5] of the report.

As a special case, we also need to handle the left-hand side of assignment statements differently. These expressions should not evaluate to a value, but instead an address to be assigned. *CodeGenVisitor* will pass off its work to *LeftHandAssignmentVisitor* when it reaches these statements. This special visitor will handle the identifiers of scalar and array variables differently, but it will use the standard *CodeGenVisitior* functionality when evaluating any subscript expressions. More information about this technique will be discussed in [Section 5.1].

## 1.2 Code Generation Helper Functions

To streamline the code generation, we will create a class full of generation helper functions. This class will keep track of the current line of memory we should write the next instruction to. All writes to machine memory should pass through these functions. Some of the functions will correspond to multiple lines of assembly. Each function will store debug information about the line generated and return the first address in memory that was written to.

## 1.3 Initialization of the Machine

After the code has been generated, the machine is initialized with the PC at 0. The MSP will be set to the beginning of the memory stack, and the MLP will be just before the constants near the end of memory. Program termination will be handled by a HALT command after all of the other code has been generated.

# 2. Storage of Variables and Program Scope

In order to keep track of scope and variable declarations, we will use a modified version of the *SymbolTable* class from the semantic analysis. The symbol table will keep track of all declared variables and declared routines. The lexical level and the address offset of each variable will be added to the table entries. The program counter for the beginning of each routine will also be stored in the table so that calls to the routine can branch there. This section of the report will discuss variable storage in greater detail.

## 2.1 Major Scopes

All variables in the main program, procedures, and functions, are going to be associated with a major scope. During runtime, each major scope will have an activation record and a control block. The runtime setup and teardown of these records will be discussed in more detail in [Section 4], but it should be noted that the main program scope will also be given a control block.

While generating code, a *Scope* object will be popped onto a list of open scopes each time a new nested major scope is observed. Each nested major scope will increment the *'lexicalLevel'* and store that level in the *Scope* object. The *Scope* object will also keep track of an *'offset'* field which is used for variable declaration.

## 2.2 Variable Declaration

During runtime, declared variables will be stored in allocated space on the stack. The space for all the local variables will be reserved on entry to the major scope. While generating code, the *Scope* object will be initialized so that the *'offset'* field starts after the control block. Each time a new variable is declared, we must increment the *'offset'* field by the amount of memory the variable will take up.

The symbol table already keeps track of variable declarations. When a variable is declared, a new *Symbol* object is created and associated with the current scope. To support code generation, these *Symbol* objects will need to be modified to include their own *'offset'* and *'lexicalLevel'* fields which will be used when referencing variables. When the *Symbol* is created, these fields are set to be the same as the *SymbolTable* at that time.

In our language, each boolean and integer will each take up one unit of memory (16 bits). Arrays will need (length1 * length2) units of space, so the symbol table entries for arrays will need to store the dimensions of the array.

Our language supports negative array indices. However, these are more complicated to deal with. When storing the dimensions of the array, we will normalize each dimension such that the first index is 0. For each dimension we will also store the normalization offset.

Figure 1: Examples of how to store array information in the symbol table

```
EXAMPLE 1: integer a[-10..10]
    In the ArraySymbol object we will store:
    { length1 = 20, length2 = NULL, offset1 = -10, offset2 = NULL }
EXAMPLE 2: integer b[5, 10..20].
    In the ArraySymbol object we will store:
    { length1 = 5, length2 = 10, offset1 = 0, offset2 = 10 }
```

When we are finished generating code for a nested scope, we will record total amount of memory needed to store all the local variables for the scope. Before the scope is removed, we will patch the function prologue [Section 4.2] to allocate the correct amount of space.

## 2.3 Minor Scopes

In our implementation, the minor scope variables will be associated with the nearest major scope. They will be allocated during the major scope setup and teardown just like the major scope variables. As a stretch goal, we can implement a more efficient use of memory by overlaying space needed for minor scope variables.

## 2.4 Constants

Integer and boolean constants will be directly encoded into our assembly code. We will also encode text constants directly into the program - this will be handled as a special case of the *'put'* statement. Since they are only ever used when printing text, the *'put'* statement can generate a list of commands that pushes each character onto the stack (in backwards order), then prints each element from the stack.

Figure 2: A demonstration of how string constants are encoded into the assembly.

| | |
|---|---|
| **put "hello"** | **PUSH 'o'**<br>**PUSH 'l'**<br>**PUSH 'l'**<br>**PUSH 'e'**<br>**PUSH 'h'**<br>**PRINTC**<br>**PRINTC**<br>**PRINTC**<br>**PRINTC**<br>**PRINTC** |

# 3. Expressions

In most cases, expressions evaluate to a single boolean or integer value. Each AST node that inherits from the *Expr* class will generate code that leaves the resulting value on top of the stack. Every other

sequence of generated code will know the result of an expression can be found on the top of the stack.

## 3.1 Accessing Constants

As described in [Section 2.4], all constants will be hardcoded into the assembly code itself. For constant expressions, the value is left on top of the stack after accessing it.

## 3.2 Accessing Variables

During runtime, we will access scalar variables by loading them from the activation record. When generating code, we need to retrieve the *Symbol* object associated with the identifier to find the lexical level and the offset (from the activation record).

Figure 3: An example of evaluating a scalar variable expression.

```
integer x
put x
```

```
# These values will be filled in using information
# from the symbol table.
ADDR <xLL> <xNO>
LOAD
# Print the resulting expression.
PRINTI
```

Accessing array elements is a little more complicated. First, we need to generate code to resolve the indices of the array to an offset, then add offset to the address of the base element of the array. From the symbol table, it is possible to retrieve the normalization offsets for each dimension [see Section 2.2]. The following steps can then be executed to find the element address:

If there is only one dimension:

- Generate code to evaluate the expression inside the indexer.
- Subtract the offset of that indexer from the result of the expression.
- Load the base address of array.
- Add the result and the base address to get the proper address.
- Retrieve the value at this location.
- Leave the value on top of the stack.

If there are two dimensions, we want to calculate an offset such that A[y,x] becomes A[y*size(A, 2)+x] in row major order:

- Evaluate the expression inside the first index.
- Subtract the offset of the first indexer from first index.
- Multiply the result by the length of the second indexer, leave it on the stack.
- Generate code to evaluate the expression inside the second indexer.
- Subtract the offset of the second indexer from the second index, leave it on the stack.
- Add the two values on top of the stack.
- Retrieve the value at the address from the top of the stack.
- Leave the value on top of the stack.

Figure 4: An example of the code generated for accessing an array element.

```
integer range[5..10]
put range[4 + 4]
```

```
# Evaluate the expression inside the indexer
PUSH 4
PUSH 4
ADD

# Subtract the NORMALIZATION OFFSET of the result
PUSH 5
SUB

# Load the base address of the array
ADDR <rangeLL> <rangeON>

# Add the index to the addr of first element
ADD
LOAD

# Print the result
PRINTI
```

## 3.3 Arithmetic Expressions

Each of the binary arithmetic expressions will evaluate both of the children first, who leave their values on top of the stack. It will then use the appropriate instruction to combine those two values.

- The addition expression will be implemented using the ADD instruction.
- The subtraction expression will be implemented using SUB instruction.
- The multiplication expression will be implemented using MUL instruction.
- The division expression will be implemented using DIV instruction.

The negation expression will be implemented using NEG instruction, and will operate on the value left on top of the stack.

## 3.4 Comparisons and Boolean Operators

Our plan for generating code for these operators are shown as pseudocode in the table below.

Figure 5: **Pseudocode** for implementing the remaining comparison operators.

| | |
|---|---|
| `a = b` | `LOAD a  # Evaluate the right side`<br>`LOAD b  # Evaluate the left side`<br>`EQ` |
| `a < b` | `LOAD a  # Evaluate the right side`<br>`LOAD b  # Evaluate the left side`<br>`LT` |
| `a <= b` | |

| | |
|---|---|
| | **LOAD a** # Evaluate the right side<br>**PUSH 1**<br>**SUB**<br>**LOAD b** # Evaluate the left side<br>**LT** |
| **a > b** | |
| | **LOAD a** # Evaluate the right side<br>**NEG**<br>**LOAD b** # Evaluate the left side<br>**NEG**<br>**LT** |
| **a >= b** | |
| | **LOAD a** # Evaluate the right side<br>**PUSH 1**<br>**ADD**<br>**NEG**<br>**LOAD b** # Evaluate the left side<br>**NEG**<br>**LT** |
| **a != b** | |
| | **LOAD a** # Evaluate the right side<br>**LOAD b** # Evaluate the left side<br>**EQ** # Check for equality<br>**NEG** # Toggle the result<br>**PUSH 1**<br>**ADD** |
| **a \| b** | |
| | **LOAD a** # Evaluate the right side<br>**LOAD b**<br>**OR** |
| **a & b** | |
| | **LOAD a** # Evaluate the right side<br>**LOAD b** # Evaluate the right side<br>**ADD**<br>**PUSH 2**<br>**EQ** # They are both true only if the sum is 2. |
| **!a** | |
| | **LOAD a** # Evaluate the expression<br>**NEG**<br>**PUSH 1**<br>**ADD** # Negating and then adding is the same as NOT. |

## 3.5 Anonymous Functions

According the the bulletin board, the anonymous function is logically equivalent to defining a parameterless function then calling it. This will be achieved by directly generating code as if this were a real function. If the expression is *{ stmts yields expr }* then the body of the function will just be *stmts* followed by *return (expr)*. After the definition of the function, the function will be called and the return value will be left on the top of the stack, just as with any other expression. For the setup/teardown of functions, see [Section 4]. While generating code for the function body, we will need to add a major scope to the symbol table just as if we were processing a function declaration.

# 4. Functions and Procedures

## 4.1 Activation Records

Our activation record will contain the following information (and in the following order):

- Control block
- Parameters to the function or procedure
- Space for local variables

Where the control block contains:

- Space for a return value, if applicable (functions only)
- Return address
- Dynamic link to previous display entry

When we refer to “activation record,” we refer to the abstract grouping of data at runtime; it is not a Java class we will be manipulating during code generation. The return address for a function/procedure, as well as other runtime-dependent information, will be pushed onto the stack prior to invocation (more on this in subsequent sections).

The dynamic link is used to restore the stack pointer of the frame previously occupying the display entry.

The amount of storage needed for local variables is given by the symbol table. While the scope is being processed, the symbol table will keep track of all the memory required for it in the *'offset'* field. In the *RoutineDecl* AST node, we will extract this value before the scope of the function is closed, and then use it to set the allocation size for the activation record.

## 4.2 Entrance and Exit Code Strategies

When a function/procedure is invoked, we want to push its activation record onto the stack prior to execution, albeit in stages. The control information and parameter portions of the activation record will be pushed on by the caller of the function, while the local variable information will be handled by the callee.

When a function/procedure gets called, the calling scope must generate code to push the return address for the subroutine onto the stack. We can do so by pushing the current value of the program counter onto the stack. We also need a link to the frame previously occupying the display corresponding to the lexical level of the subroutine being called. When the current subroutine exits, we can restore this entry to the display. We accomplish this by pushing this previous value onto the stack.

From the caller, we also need to pass the parameters (if any) to the subroutine. Note that we are utilizing argument passing by value, since the only arguments allowed in the source language are of type boolean or integer (thus we do not need to handle passing arrays as arguments). If argument expressions must be evaluated and replaced with the corresponding values prior to function execution. Also note that parameters are pushed onto the stack after the control block, to minimize clean-up that the caller is made responsible for.

Figure 6: Procedure/function set-up code (caller)

```
PUSH 0              # Space for return value.
PUSH <PC>           # Push return address (current value of pc).

ADDR <LL>, 2        # Save the address of display,
                    # the lexical level is known at compile time.
LOAD

PUSH arg1           # push parameters

…

PUSH argn
```

Once the caller finishes pushing these values to the stack, we can then branch to the subroutine code. At the start of every subroutine, we will emit a prologue which will handle the remainder of the code entrance set-up.

In the prologue code, we must first update the display so that the current subroutine occupies the display entry for its lexical level.

Figure 7: Procedure/function prologue (callee)

```
# Update the display entry for the lexical level.
# This is known at compile time.

PUSHMT
PUSH <hsize>    # hsize = size of activation record data
                # already on stack (return address, etc.)

SUB             # subtract the two to get stack address corresponding
                # to start of activation record, then
SETD <LL>       # set Display to this value

# Allocate space for the local variables

PUSH N          # N = size of local variables.
PUSH 0          # Dummy value.
DUPN
```

Upon exit, the callee will execute clean-up code which will first pop off all of parts of the stack used for local variable storage. Since we previously stored the exact size of the local variable storage region, we can POPN that value once we reach the end of function execution. We then need to replace the display entry as described previously, and lastly clean up the parameters, which are also accompanied by a size value.

Figure 8: Procedure/function epilogue (callee)

```
PUSH M                  # M = N + size of parameters
POPN                    # free memory
SETD LL                 # restore prev. display
BR
```

For functions, we also need to handle the return value. Again, since return values must be of either boolean or integer type, we can simply push the value onto the stack, without worrying about addressing or alignment. Since we reserved space at the top of the activation record for the return value, we can simply put in that value using STORE.

Lastly, we branch back to the return address stored at the top of the stack. Note that the clean-up code for the subroutine is always stored as the last instructions for the subroutine. If return statements are contained in conditional blocks (such as an IF statement), they will need to branch to the epilogue, the address of which will only be determined after the first pass through code generation. We use the *BranchPatchVisitor* to accomplish patch this in later [see Section 5.5]. Due to this design, there is no clean-up code for the caller to handle, so we are finished.

## 4.3 Function & Procedure Calling

To call a function or procedure, we will need to refer to the symbol table to locate its address. The symbol table stores the memory address of the first instruction of a function or procedure declared in the main scope. For functions declared within nested levels, since our symbol table clears information once a scope is closed, it is assumed that such functions can only be called from within those nested scopes. Thus code generation will emit instructions to branch to nested function addresses within the instruction block of the enclosing function.

Figure 9: Calling a function.

```
# Assuming myfunc() exists and
# returns an integer
integer i = myfunc();
```

```
# myfuncaddr is determined by symbol
# table
PUSH <myfuncaddr>
BR
```

## 4.4 Display Management

As previously noted, the display is mainly managed from within a subroutine's prologue/epilogue code. Our display update policy ensures that each function or procedure call will retain the address of the activation record previously occupying the display entry for its lexical level, so that value may be restored. We assume that the display entry for each lexical level contains at minimum some default value. Even if no such activation record exists corresponding to that value (for instance for the first function call of a lexical level), we will nevertheless store and restore this value to the display entry to simplify logic and maintain generality.

The target machine has a finite number of display registers, thus theoretically placing a restriction on the maximum nesting depth. Though extremely unlikely, it is possible that a program exceeds the depth enforced by the number of available display registers. For the purposes of simplicity, if this occurs we will merely throw an error.

# 5. Statements

## 5.1 Assignment Statements

There are two main steps to handling an assignment. First, the left side of the assignment needs to be resolved to a memory address. Second, instructions need to be generated to compute the result of the expression on the righthand side. Finally, an instruction must be generated to store the computed result at the memory address found in the first step.

To complete the first step, a *LeftHandAssignmentVisitor* will be called. It will evaluate the lefthand expression, and generate code to push the memory address of the variable onto the stack. How this evaluation is done will vary somewhat between scalars and arrays.

For scalars (integers/booleans), the *Symbol* corresponding the the lefthand variable needs to be retrieved form the symbol table. The *offset* can then be gotten from this Symbol object. The *Symbol* also contains a reference to its lexical level. By looking up the base of the activation record in the display, we now have the two pieces of information needed to find the absolute address of the lefthand variable. This result is then pushed onto the stack.

For arrays, a similar process will be followed to find the base memory address of the array. Afterwards, the subscript expressions can be evaluated using the regular *CodeGenVisitor*. For one-dimensional arrays, the address to be returned is found by adding the subscript value to the memory address of the first element. For two-dimensional arrays, we have elected to store elements in row major order. We will use the same method to compute the memory address that is described in [Section 3.2].

Whether the lefthand side is scalar or an array, the end result of the *LeftHandAssignmentVisitor* is that a memory address has been pushed onto the stack. After this has been done, the result of the right-hand expression can be evaluated using the method described in Section 3. The result of this evaluation is pushed onto the stack. Finally, a STORE instruction can be added to place the value into the correct memory address.

Figure 10: A simple example of variable assignment.

```
# x is an integer var
x <= 5
```

```
# Suppose LL for the current scope is 0
# and we've found in the symbol table
# that the offset for variable x is 10.
ADDR 0 10  # push the address for x onto stack
PUSH 5     # push the value 5 onto stack
STORE      # store the value at the address
```

## 5.2. If Statements

The if statement ASTNode will generate most of its machine code on the first pass through its children. However, a couple addresses will need to be modified after all of the children have been visited. The order of visitation of the children is as follows.

1. Visit the predicate expression. Doing so will generate the instructions to evaluate the expression and push the result onto the stack.
2. Issue instructions to push value *j* to the stack, where *j* equals the address of the first instruction **in** the else block (or immediately after the else block if no such block exists). Since, we don't know what this address is yet, we'll just push UNDEFINED for now. In the IfStmt node, keep track of the memory address at which *j* should be filled.
3. Generate a statement to branch to if the expression is false.
4. Generate the instructions for the statements in the *if-true* block. Since if statements are minor scopes, no changes to the display registers need to be made before we start doing this.
5. If there is an else block, perform steps 6-8:
    6. Issue instructions to push value *f* to the stack, where *f* equals the address of the first instruction **following** the else block. Since, we don't know what this address is yet, we'll just push UNDEFINED for now. Keep track of the memory address at which *f* is located.
    7. Generate an unconditional branch instruction.
    8. Generate the instructions for the statements in the else block.
9. Now that the instructions for both the if block and the else block (if there was one) have been generated, it is possible to fill in the values for *j* and *h* by caching the appropriate memory addresses along the way. This will be done during the second pass of the AST.

Figure 11: The machine code generated before applying Step 6.

```
if (true)
begin
    print 1
else
    print 0
end
```

```
PUSH 1    # Result from expression evaluation
PUSH ?
BF
PRINTI 1  # Do the print statement
PUSH ?    # Skip the else block to go to the end of the if
BR
PRINTI 0
...       # Out of the if statement
```

Figure 12: The machine code generated after applying Step 6. Numbers represent mem addresses.

```
if (true)
begin
    print 1
else
    print 0
end
```

```
...
11: PUSH 1     # Result from expression evaluation
12: PUSH 16
13: BF
14: PRINTI 1   # Do the print statement
15: PUSH 17    # Skip the else block
15: BR
16: PRINTI 0
17: ...        # Out of the if statement
```

## 5.3 While and Loop Statements

The while statement ASTNode will generate code according to this template below.

Figure 13: A template of the machine instructions generated for a while loop.

```
while (<expression>)
    <actions>
end
```

```
A : <eval expression>   # Result is pushed to stack
    PUSH B              # Jump to address B if
    BF                  # expression result is false
    <perform actions>
    PUSH A              # Re-evaluate the while
    BR                  # predicate
B : ...                 # Out of the while loop
```

When generating the code during the first pass, we store the memory address for A and B, and the lines of code that will need patching on the while-loop AST node. During the second pass, we patch these values appropriately.

For the infinite loop statement, we only need one pass. We can store the address of the beginning of the loop, then use that branch back at the end.

Figure 14: A template of the machine instructions generated for an infinite loop.

```
loop
   <actions>
end
```

```
A : <perform actions>  # Beginning of loop.
    PUSH A             # After the actions have been
                       # performed.
    BR                 # Branch back to the beginning.
```

## 5.4. Exit Statements

Exit statements are translated to the code snippets below. Notice that we do not know the branching address at the time of generation.

Figure 15: An example of an exit statement.

| | |
|---|---|
| **exit** | **PUSH ?** # the address of out of the loop/while<br>**BR** |

Figure 16: Example of an exit when true statement.

| | |
|---|---|
| **exit when (true)** | **...** # The result of the expression.<br>**NEG** **#** Negate the result of the expression<br>**PUSH 1**<br>**ADD** **# since when want to jump if it is true.**<br>(It follows the same formula of the NOT X > (-X)+1 )<br>**PUSH ?** # Address of the end of the loop.<br>**BF** # Jump if false (which means the<br># expression was true) |

While doing the first pass, the *ExitStmt* AST node will store the memory addresses that need to be patched. In addition, any looping statement will cache the address of the next line of code after the loop in its AST node. During the second pass, whenever a loop is entered, a minor loop scope will be opened. This loop scope will also be given a reference to the AST node representing the loop. When patching the *ExitStmt*, we will find the loop scope from the symbol table, and use the cached value to patch the branch statement.

## 5.5. Return Statements

Return statements either appear standalone or accompanied by a return value. A return statement by itself will simply generate a branch instruction to the location of the clean-up code.

Figure 17: Example of a return statement.

| | |
|---|---|
| **return** | **PUSH ?** **# X is the address of the epilogue**<br>**BR** |

The return address for the epilogue will be stored on the function AST node during the first pass. On the second pass, the function scope will hold a reference to that AST node, which the *ReturnStmt* node will use to patch its line.

A return statement which has a return value will need to update the return value, then branch back. Recall that in the activation record, space for the return value is at offset 0.

| Figure 18: Example of a return statement. | |
|---|---|
| `return (1)` | `ADDR LL 0    # store the return value`<br>`PUSH 1       # in the stack`<br>`STORE`<br>`PUSH ?       # x is the address of the epilogue`<br>`BR` |

## 5.6. Put Statement

The AST subclass *PutStmt*, will visit its *Printable* children, each of whom will generate a sequence of PUSH and PRINT instructions. Expressions that return integers will call PUSH with the return value, and then call PRINTI. The *TextConstExpn* will iterate over each of the characters in a text constant backwards, calling PUSH with the character code. It then calls PRINTC the appropriate number of times.

| Figure 18: Example of a put statement. | |
|---|---|
| `put "hi", 4` | `PUSH 'i'`<br>`PUSH 'h'`<br>`PRINTC`<br>`PRINTC`<br>`PUSH 4`<br>`PRINTI` |

## 5.7 Get Statements

The get statement can be resolved by the first pass of the code generation. First, we push the variable address to the stack. Then, we read in the input integer(s). Finally, we store the value in the variable address.

| Figure 19: Example of a get statement. | |
|---|---|
| `get x` | `READI`<br>`ADDR 0,3   #the address of the variable x`<br>`STORE` |