*public void emitInstructions ( String instruction )*

* This function takes a string parameter and splits it by space into tokens. Every time when a token is written to the memory, we add it to an *ArrayList<String> instructionCounter* so we can keep track of the address to write to the memory. If a machine instruction is detected, then it will write memory as *Machine.Instruction* to the runtime stack. If a token is not a machine instruction, then it will be considered as a value and write to the memory in short.

*public SymbolTableEntry lookup ( String varname, boolean local )*

* This function takes the name of a variable and lookup in the *SymbolTable*. If a variable already exists, then it returns the *SymbolTableEntry*. Otherwise, it returns null. If we only wish to search in the local scope, then local as true. If we wish to search through the entire *SymbolTable*, then local as false.

*public int lookup\_offset (String varname, boolean local )*

* This function is similar to the *lookup* function, except that it returns the offset of a variable on the stack

*public int lookup\_lex ( String varname, boolean local )*

* This function is similar to *lookup\_offset* but it returns the lexical level of a variable

*public void addEntry(String varname, Type type, SymbolKind kind, AST node, int off, int lex)*

* This function takes the name, type, symbol kind, node, offset, and lexical level and add it to the *SymbolTable*. It returns nothing if the no *SymbolTable* exists.

*public void enterScope(ScopeKind kind, RoutineDecl routine)*

* Create a new *MajorScope* and a new *SymbolTable.*

*public void exitScope()*

* Remove the scope

*public Boolean visit(Program stmt)*

* This function is the beginning of the program. First we push the current stack address to the stack and then store it as the lexical level 0. We call *enterScope* to create a new scope and then we start visiting the statements stored in the body. After all nodes in the body are visited, we then pop all the local variables on the stack and stop the machine.

*public Boolean visit(ArrayDeclPart decl)*

* When an array is declared, we store the name of the variable in our *SymbolTable* and allocate space according to the size of the array.

*public Boolean visit(MultiDeclarations decl)*

* *MultiDeclarations* is visited whenever there is one or multiple variables declared. Therefore we check the type of each variable and do the appropriate visit accordingly. No variables are declared here.

*public Boolean visit(RoutineDecl decl)*

* When a function or procedure is declared, we need to skip all its content first. Therefore we need to know where the end of the *RoutineDecl* is and branch to that address. We first allocate a space by pushing a 0 and later change it to the correct address.
* When ever a function or procedure is called, we then branch back here to execute the body of the declaration. Thus, we need to store the address in hash to allow routine call to branch back. Then we create a new scope for the routine and then set current address a new display register.
* Since the parameters are already declared from the caller, we simply add them to the *SymbolTable* for the current routine. Then we proceed to visiting the body statements.
* Finally we remove the scope and pop all its local variables and its parameters, and then branch back to the caller.

*public Boolean visit(ScalarDecl decl)*

* When a scalar variable is declared, we simply store it in the *SymbolTable* and allocate a space on the stack

*public Boolean visit(AnonFuncExpn expn)*

* TODO: Winston?

*public Boolean visit(ArithExpn expn)*

* When an arithmetic statement is called, we visit the BinaryExpn to find the according values and perform the operation according to the operation symbol.

*public Boolean visit(BinaryExpn expn)*

* This expression contains left and right sub-expressions. Therefore we need to visit them to get their values to perform the according operations. If the left or right is an instance of *IdentExpn*, which returns the address of the variable, then we need to load that address to the value. If they are functions or procedures calls, which the return values would already be a real value, not an address, we do not load them.

*public Boolean visit(BoolConstExpn expn)*

* Since this is a boolean constant, we simply push the expression to the stack.

*public Boolean visit(BoolExpn expn)*

* TODO: Winston?

public Boolean visit(CompareExpn expn)

* First we visit the left and right nodes of this expression, which then would return their values. Then we perform operations according to the operators.
* For less than *a < b*, we simple emit a machine instruction *LT* to the machine.
* For less than or equal to *(a <= b)*, we interpret it as *not(b < a)*, so we first swap the two values and do a less than comparison. Then we negate the value by comparing to false.
* For more than *(a > b)*, we simple swap the 2 values and do a less than comparison.
* For more than or equal to (a >= b), we follow the similar steps as less than or equal to comparison.

*public Boolean visit(EqualsExpn expn)*

* We first visit the left and right nodes to get their values and then we perform the operation according to the operator. If it’s an equal comparison, then we emit *EQ*. Otherwise, we emit *EQ* and negate that value by comparing it to false.

*public Boolean visit(FunctionCallExpn expn)*

* When a function call expression is called, we first need to allocate a space each for the return value and return address so that they can be later changed to the correct value or address. Then we take all the parameters and push them onto the stack. Since visiting *IdentExpn* would return an address instead of a value, when an argument is an instance of *IdentExpn*, we need to find its real value for the ease of function or procedure operations. Finally we branch to the callee function and start its executions.

*public Boolean visit(IdentExpn expn)*

* If the expression is not a function, we treat it like a variable and push its address to the stack. Otherwise, we need to perform function operation for such expression.

*public Boolean visit(IntConstExpn expn)*

* Push the integer value directly to the stack.

*public Boolean visit(NotExpn expn)*

* TODO: Winston

*public Boolean visit(SkipConstExpn expn)*

* TODO: Winston

public Boolean visit(SubsExpn expn)

* When dealing with an array, it’s more complicated to calculate the exact location of its allocation. The formula to calculate the location of a one-dimensional array is…

and offset can be calculated by…

Therefore to calculate the offset, we first visit *Subscript1* to get the value of the index, then if it’s an *IdentExpn*, then we load for the value, and if not, then we already have the value. Then we subtract the value by its lower bound and add it to the stack address of the variable.

* On the other hand for two dimensional array the offset is calculated differently…

Therefore we push the appropriate values on the stack and do the arithmetic calculations according to the formula and add it to the address of the variable.

*public Boolean visit(TextConstExpn expn)*

* TODO: winston

*public Boolean visit(UnaryExpn expn)*

* TODO: Winston

*public Boolean visit(UnaryMinusExpn expn)*

* TODO: Winston

*public Boolean visit(AssignStmt stmt)*

* First we visit left and right nodes for its value. We load the address the right node produces for its value. Then we store the value to the address produced by the left node.

*public Boolean visit(ExitStmt stmt)*

* TODO: Winston

*public Boolean visit(GetStmt stmt)*

* TODO: Winston

*public Boolean visit(IfStmt stmt)*

* TODO: Winston

*public Boolean visit(LoopingStmt stmt)*

* TODO: Winston

*public Boolean visit(LoopStmt stmt)*

* TODO: Winston

*public Boolean visit(ProcedureCallStmt stmt)*

* Similar to function call expression except it does not allocate space for a return value.

*public Boolean visit(PutStmt stmt)*

* TODO: Winston

*public Boolean visit(ReturnStmt stmt)*

* Return statement finds the location of a return value of a function and store the desire value to it.

*public Boolean visit(Scope stmt)*

* First we need declare all local variables, so we call *findDeclarations*. Then we visit the rest of the body statements.

*private void findDeclarations(ASTList<Stmt> body)*

* This is a helper function to find all the local variables in the body statements. It’s called reclusively to look for local statements inside loops and if statements.

*public Boolean visit(WhileDoStmt stmt)*

* TODO: Winston