CSC488 Team 2 A5 Code Documentation

Changes Since A4

Since A4 we made the following changes:

- Anonymous functions use the same code generation as functions, but instead of branching around the function and having a return address, the program will directly enter the anonymous function.
    - We set the display register
    - We allocate space for local variables
    - Generate code for the body of the anonymous function
    - Generate code for the return value
- We restructured the activation record for functions and procedures to have the return address right above the return value and under the parameters. This way, the return address will not be popped off before the function can return.
- The activation record for functions and procedures is now the same. For the return value, procedures have UNDEFINED here. Then, after the procedure returns and extra POP is emitted to remove the UNDEFINED value.
- When entering a new scope, we walk through the AST to find all declarations (local variables) and allocate space for them. Then, these declarations are removed from the AST. This way we will allocate the right amount of memory for each scope, and will pop the correct number of arguments and can access these variables easily.
- We changed all the Boolean operators (AND, OR, and NOT) to use branching statements instead of evaluating the actual Boolean. This makes it so that it correctly implements conditional AND and conditional OR. The template for each of these is described later on in the document.
- We use a two dimensional ArrayList to keep track of the addresses that need to be patched for 'exit' and 'exit when'. The first dimension of the ArrayList has an ArrayList at the end which is the current loop. At the beginning of each loop, a new ArrayList is pushed into this list, and at the end of each loop it is popped off. During the generation of code for the loop, the address of any 'exit' statements are pushed into the latest ArrayList.

Code Documentation

The following is the documentation for the class *compiler488.codegen.Codegen*. This was the only class that was implemented for the code generation. Following up with the visitor design pattern used in A3, we recycled the ASTVisitor interface for CodeGen which now implements it. The idea is that each node on the AST is visited and code is generated based on the type of the node. We defined several private variables:

ArrayList<String> instructionCounter: keeps track of the instructions we have generated.

LinkedList<MajorScope> scopes: keeps track of the major scopes in the program
lexicalLevel: keeps track of the current lexical level, used to set the display register. It is incremented at the beginning of a new function declaration and decremented at the end.
HashMap<String, Integer> hash: A symbol table used to keep track of the start addresses of each function.
int num_var: Keeps track of the number of variables in the current scope being visited. Is used to pop off variable off the stack at the end of a scope.
int num_par: Keeps track of the number of parameters in a function. Used to pop parameters off the stack at the end of a scope.

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)

- When visiting an AnonFuncExpn, the code is similar to declaring a new function. However, instead of branching around the function and saving it in our SymbolTable, the function is executed right away.
- First, we enter a new scope, meaning that we keep track of the number of newly declared variables and allocate space for all the declarations in this scope. Then the display register is set.
- Then, the body of this function is visited and the code for it is emitted.
- Afterwards, the scope is exited, and all local variables are popped off the stack.

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)

- First, the left side of the expression is visited and the code for it is generated. After this, the result of the left side will be at the top of the stack (either MACHINE_TRUE or MACHINE_FALSE). If the expression was a variable, then a LOAD will need to be emitted since only the address of the variable will be on the top of the stack and needs to be loaded.
- Then two different things are done depending on the operation:
    - OR operation: The template for the code looks as follows:
      …code to evaluate left side…
      PUSH X
      BF
      PUSH MACHINE_TRUE
      PUSH X2
      BR
      …code to evaluate right side...
      Where X is the first instruction of the right side, and X2 is the first instruction after the code for the right side. The reasoning for this is that if the left side is true, then the right side does not need to be evaluated. So, MACHINE_TRUE will be pushed to the top of stack, and then we branch around the right side so it is not evaluated. However, if the left side was false, then we branch to evaluate the right side. After which, the right side is at the top of stack and determines the Boolean value of the overall expression.
    - AND operation: The template for the code looks as follows:
      …code to evaluate left side…
      PUSH X
      BF
      …code to evaluate right side…
      PUSH X
      BF
      PUSH MACHINE_TRUE
      PUSH X2
      BR
      PUSH MACHINE_FALSE
      Where X is the instruction PUSH MACHINE_FALSE and X2 is the first instruction after PUSH MACHINE_FALSE. The reasoning for this is that if the left side is false, then the right side does not need to be evaluated, so we branch to push MACHINE_FALSE. Otherwise, if the right side is true, then it will follow through to push MACHINE_TRUE and then branch to the instructions after this

Boolean expression. If the right side is false, then it will branch to push MACHINE_FALSE.

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)

- The template for the code is as follows:
  …code to evaluate the operand…
  PUSH X
  BF
  PUSH MACHINE_FALSE
  PUSH X2
  BR
  PUSH MACHINE_TRUE
  Where X is the address of push MACHINE_TRUE and X2 is the address of the instruction after push MACHINE_TRUE. The reasoning for this is that if the expression

is true, then it will not branch, and then push FALSE and then branch to the instruction after push MACHINE_TRUE (the next instruction after the operand). If the expression is false, then it will branch to push MACHINE_TRUE and continue.

public Boolean visit(SkipConstExpn expn)

- This function pushes the ASCII value of a new line character and a PRINTC to print to standard out.

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…

$$location = location\ of\ variable + offset$$

and offset can be calculated by…

$$offset = sub1 - lb1$$

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…

$$offset = (ub2 - lb2 + 1) * (sub1 - lb1) + (sub2 - lb2)$$

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)

- In this function we iterate through every character of the string to be printed and push its ASCII value on the stack followed by a PRINTC for each character.

public Boolean visit(UnaryMinusExpn expn)

- This function is for a negation (i.e. -1). First the expression that the negative is being applied to is evaluated and the code for it is written. At this point the value of the expression is at the top of stack. Then, the machine operation NEG is emitted.

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)

- To keep track of the addresses that need to be patched, we have made a two dimensional ArrayList called loopExits. In loopExits, the first dimension keeps track of the loop we are currently in (the very last one in the list). Then the second dimension keeps track of the addresses that need to be patched for this loop. The reasoning for this design is to

handle nested loops and multiple exit statements. For the nested loop, each ArrayList represents a loop, while each array list has addresses for each exit statement.

- If the exit statement has an expression associated with it then it is an 'exit when' statement. So, to evaluate the conditional, we create a NotExpn out of the expression and then evaluate it. At this point the negation of the conditional is put on the top of stack. Then, we emit a PUSH and BF, keeping track of the address of the PUSH for a patch later on. The address is stored in the last ArrayList within loopExits. This is so that if the NotExpn evaluates to false, then it means that the expression is now true and can branch.
- If the exit statement does not have an expression associated with it, then it is just a regular 'exit statement'. We simple do a PUSH and BR, keeping track of the address of the push using loopExits, to be patched later on.
- Addresses will be patched when the loop body statements are finished being evaluated.

public Boolean visit(GetStmt stmt)

- We iterate through each input being evaluated and emit the code to get the address of the variable, a READI to read from standard input and a STORE.

public Boolean visit(IfStmt stmt)

- First the expression for the If statement is evaluated. A PUSH is evaluated which will be patched with the address of the end of conditional if no else clause exists. If an else clause exists then it will be patched with the start of the else clause.
- If an else clause exists, then at the end of the If clause, a PUSH is emitted. This PUSH will be patched with the address of the end of the else clause.

public Boolean visit(LoopingStmt stmt)

- Visits either a WhileDoStmt or LoopStmt depending on what the statement is.

public Boolean visit(LoopStmt stmt)

- At the start of LoopStmt we add a new ArrayList<Integer> to the end of loopExits to represent this loop and keep track of the exit statements that need to be patched.
- Then the body of the loop is visited and the code for it is evaluated.
- After the statements for the body are emitted, then the last ArrayList form loopExits is popped, and these addresses are filled with the current instruction address (the end of loop).

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)

- Generates code for each output respective of its type.

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 declarations inside loops and if statements.

public Boolean visit(WhileDoStmt stmt)

- Like LoopStmt, a new ArrayList<Integer> is added to loopExits to keep track of the addresses that need to be patched.
- The current address is saved and then the code for the conditional is generated. A PUSH is generated which will be patched with the address of the end of the loop, so that the program will branch to the end of loop if the condition no longer holds.
- The code for the body of the WhileDoStmt is generated, and at the very end a branch back to the conditional is generated.
- Finally, the latest ArrayList<Integer> is popped off of loopExits and each address is patched with the current address in memory.