A5 Design Documentation

Overview

For our A5 implementation, we followed the previously used Visitor design pattern to handle code generation. We expanded several existing classes from the last assignment in order to provide the correct behaviour for code generation, as well as created new classes to support functionality for aspects such as address patching, assignment evaluation, and more.

Java Class Index

Here is a list of noteworthy Java classes that we implemented, along with a brief description of what each class encapsulates.

NodeVisitor - a class that calls *accept* on every node in the AST. Extended by the **CodeGenVisitor** and **SemanticsVisitor** classes.

SemanticsVisitor - extends **NodeVisitor** class in order to provide extra functionality on top of the node traversal.

CodeGenVisitor – goes through each node in the AST and determines what code needs to be generated.

ExpnAddressVisitor - a secondary Visitor used to generate the address resulting from evaluating a variable expression, rather than the actual value stored at that address. This was used to generate code for the left-hand side of assignment statements (which will be elaborated upon later).

CodeWriter – Provides an interface (figuratively) with which to interact with the Machine. It handles patching addresses for branch operations, and provides utility methods for writing an instruction and its parameters to the machine.

SymbolTable – the same symbol table used in A3 with some modifications that are described below

Symbol – the same Symbol class used in A3, except with an additional field to store the address offset from the first address of the lexical level in which the symbol was declared.

SemType – an abstract class used to store extra semantics information about each Symbol. (Each Symbol has a SemType field)

PrimitiveSemType – extends SemType and is used to represent symbols that are either integers or booleans.

IntegerSemType and **BooleanSemType** – extend **PrimitiveSemType**

ArraySemType – extends **SemType**, and stores the array's bounds as well as its type (boolean or integer)

RoutineSemType – stores the parameter types, return type, start address, and lexical level of a routine.

SymScope – a class that stores information about an open scope, its type, its lexical level, and how much memory its local variables have claimed so far. A list of **SymScope** objects is stored in the symbol table.

Differences From A3

We attempted to reuse as much code from A3 as possible, although some changes had to be made. In A3, we created a NodeVisitor class that visited each AST node in the AST without performing any specific actions at each node. The SemanticVisitor used in A3 was an extension of this class. We were able to have the **CodeGenVisitor** from A5 also extend this class. However, many of the visit methods were not able to call into the parent method in the NodeVisitor because they needed to write code to the machine before, between, and after handling code generation for various children.

Some small modifications were made to the SymbolTable. For semantic analysis, it was enough to store an enum for each open scope indicating what type it was (routine, loop, etc.). However, for code generation we needed to keep track of much more information about each scope. The list of enums in the SymbolTable was replaced with a list of **SymScope** objects. A **SymScope** object keeps track of its lexical level and how much space the symbols declared in this lexical level are taking up. This is useful so that when a new symbol is declared, an address can be assigned to it. The **SymScope** object also keeps track of all of the return/exit statements that have been seen while generating code for the scope. When the scope is exited, the addresses of the returns/exits need to be patched.

Although we already had **SemTypes** for A3, several fields were added to them for code generation purposes. The **ArraySemType** was given extra information to store the number of elements in each dimension, as well as the offset for each dimension. The offset was generated by taking the bounds provided by the program, and figuring out how much needed to be added to make them start at 1.

Code Generation Features and Designs

Here you will find some of the key features and design decisions we implemented to combat certain code generation concerns and address issues noted from feedback on our code generation templates (Assignment 4).

Assignment Statements
Assignment statements consist of two expressions, one for the left-hand side of the assignment, and one for the right-hand side. However, these two sides need to generate different types of code. The right-hand express needs to generate code that evaluates the expression and pushes the result on to the stack, while the right-hand side needs to determine the memory address of the variable being referenced and push it onto the stack.

To distinguish between these two types of expressions, we added a **ExpnAddressVisitor** which traverses the AST for an expression and generates the address of the variable being referenced. The ExpnAddressVisitor only needs to visit IdentExpn, the AST nodes representing scalar variables, and **SubsExpn**, the AST nodes representing arrays, as these are the only two types of expressions for which addresses need to be generated.

Patching Addresses
In our A4 design, we mentioned that we planned to use a section Visitor that would traverse the entire AST once the **CodeGenVisitor** was finished, and patch any branch addresses that could not be filled in during the first traversal of the AST. However, we found that this was not necessary.

Instead, any time we found an address that needed to be patched, we kept a record of this address in an **AddressPatch** object. By passing references to these **AddressPatches** back to the **CodeGenVisitor,** the **CodeGenVisitor** is able to fill in the missing address once it knows what their values should be.

For example, an “if” statement will first visit the expression of its condition. Then it will write a branch instruction that needs to be patched. Afterwards, it will visit all of the statements in the body of the “if.” By this point, we know that the address of the branch instruction should point to the value of the program counter after the last instruction in the “if” body has been generated. It is now possible to patch the branch using the information stored in the **AddressPatch** object that was returned when the branch instruction was first written.

For loops and routines, there can be any number of addresses to patch: one for each loop or return statement. Every time a “return” or “exit” is seen, the branch instruction is generated, and the **AddressPatch** that was returned is added to a list in the current scope. Once all of the statements within the loop or routine have been generated, these patches can be iterated over and resolved.

Improvements to Expression
We made substantial improvements to our expressions code generation from what we had described in the A4 design. In particular, we fixed some logical errors for comparison operators. We realized that all of the comparisons operators could be formed out of combinations of “less than” and “not.” We also modified our “and” and “or” code to use branching. If the first clause in an “and” fails, then a branch can be immediately issued so that the second clause is not executed. Similarly, if the first clause in an “or” passes, then it is not necessary to evaluate the second clause. This short-circuiting behaviour is consistent with what many compilers and common programming languages provide.

Final Thoughts

Although we are quite content with the design of our compiler in terms of software design principles, we will admit that there is room for improvement. Specifically, one non-uniform aspect of our CodeGenVisitor class is the inconsistency between certain nodes who call their parent’s implementation of the visit methods versus those who walk a more specialized route. Ideally, there would be a way to generalize this behaviour or at minimum, make it clearer which subclasses exhibit this deviant behaviour. However, given the time constraints and relatively lighter expectations of a student compiler, we did not have a chance to explore this possible improvement. Overall, we applied best practices wherever possible and are confident that our implementation is on par with expected standards.