Group 35 WACC Report - Talking SMACC

Thomas Grigg Kiran Patel Daniel Slocombe William Springsteen

Product

During the construction of our compiler, we made sure that we followed the given specification very carefully. We used ANTLR to perform lexical and syntactic analysis. We performed semantic analysis on the parse tree generated by ANTLR using the visitor pattern. This visitor pattern used the base visitor pattern generated by ANTLR, which we extended. This semantic analysis checks that the types make sense, and that the given `.wacc` file is valid according to the WACC language specification, catching any semantic errors that the lexical and syntactic analysis didn't catch. In the semantic analyser, custom exceptions are thrown and caught, and an appropriate error message is added to our `ErrorMessageContainer` when the exception is caught. The semantic analyser will carry on traversing the parse tree, adding any more error messages to the `ErrorMessageContainer`. After semantic analysis, all error messages in the `ErrorMessageContainer` are printed to `stderr`, and the code will not be further compiled. The error messages give the line and column number of each error, and are very informative, allowing the programmer to easily resolve any compilation issues. Once we know the code is semantically correct, the compiler can proceed to code generation, by traversing the AST produced in the semantic analyser. All tests on LabTS, except the double free test cases, pass. Also, all of our custom tests pass, as well as the advanced tests in `wacc_examples`, such as `ticTacToe.wacc` and `binarySortTree.wacc`. One potential criticism of our semantic analyser is that there is a lot of code and logic in the first pass. The logic for creating the AST is interwoven with the logic of the semantic analysis. This lack of separation of logic makes the code hard to read, messy, and difficult to debug. However, creating the AST while doing semantic analysis allows both of these tasks to be completed in one pass, making our compiler much more efficient.

Our compiler is set up well for future development. Lots of interfaces/abstract classes are used, of which lots of classes will implement/extend the same interface/abstract class. For example, each node in our AST has a separate class, such as `ArrayElemNode` or `CallNode`, that extends either the `ExprNode` or `StatNode` abstract class. `ExprNode` and `StatNode` both implement the `ASTNode` interface, so every type of node in the AST can be passed as an `ASTNode`. If anything needs to be added to the WACC language, ANTLR can be easily updated to allow this new syntax, a new visit method for this update can be added to the semantic analyser, a new AST node, that extends either `ExprNode` or `StatNode`, can be created, and a new translate method for this update can be added to the translator. Also, a new `ARMNode` will potentially be needed, if an assembly command is needed for this update that isn't currently supported by our compiler. Furthermore, when starting to make the front end of the compiler, we had to think about the optimisations we could do and how we could do things in the front end so that any optimisations could be implemented fairly easily. One example of this kind of consideration was how we stored the generated code. We made a class for each possible ARM assembly instruction (`ARMNode`s for `MOV`, `STR`, etc.) and then each generated instruction was an instance of one of these `ARMNode`s, where the most recently generated instruction was inserted on to the end of the list. This makes it easier to optimise the actual ARM assembly code, as each instruction can easily be accessed, changed, removed, or moved around.

Project Management

Overall, our group was very organised. We met in labs most days so that we could work with help from each other, or to discuss how to go forward with the project. Before writing any code for a milestone, we spent time together planning out exactly what we were going to do for that milestone, and how we would do it, using a whiteboard. This ensured that everybody knew what was going on, and nobody was left in the dark about any part of the project. It also meant that we could come up with the best way to complete each milestone, as we were all together, so we could all put our ideas forward and discuss which ideas would work and which ones wouldn't. We made sure that we started each milestone well before the deadline, so that we would have as much time as possible to get that milestone completed. Our excellent organisation was partly down to our use of online group messaging applications: When we were not in labs, we were frequently communicating on this group chat to to help one another out with our individual WACC goals and to schedule lab sessions.

The whole group made continual use of Git throughout the project, making sure that we all regularly committed any changes we had made. All commit messages were very descriptive, and told the rest of the group exactly what that person had changed in that commit, rather than entering a useless commit message. We made very effective use of branching in Git. This meant that when somebody wanted to add something new, but didn't want to risk messing anything up on the master branch (or current branch the group is working from), they could just make a new branch from the current working branch, then merge it later once they are sure everything they have implemented works. An example of this was when somebody was trying to implement an extension, and had to make sure that it was working as expected before putting it on the current working branch. A new branch would also be made when something disjoint was being added, which meant that the eventual merge with the current working branch would be relatively straight forward. For example, when the AST viewer was being implemented, a new branch was made to contain this code, while everybody else worked on the code generation part of the front end, and then this AST viewer could be easily merged back into the current working branch. Another feature of Git that we used were tags. We used tags to mark when a significant part of the compiler had been completed, such as when the AST construction was completed, and when the compiler passed all tests on LabTS. This enabled us to quickly find a particular commit that we wanted to find.

Initially, we read through the specification as a group, and discussed how we would go about completing the project. We wrote the ANTLR code together, but then quickly split into pairs for the front end. Billy and Kiran worked on the visitor pattern for visiting the parse tree generated by ANTLR, while Dan and Tom worked on a separate branch, implementing the symbol table, type identification, custom exceptions, and a script to help with testing. We then worked together to finish off the semantic analyser, fixing any bugs and improving on the work that we had just done in pairs. We took a very similar approach with the back end. When we split into pairs, Dan and Tom worked on constructing the AST while visiting the parse tree during semantic analysis (So semantic analysis and AST construction happened in one pass) and thought about the sorts of optimisations we could implement later on, and how this would affect how we went about code generation. Billy and Kiran worked on learning the ARM assembly language and thinking about how to traverse the AST to generate correct ARM assembly code. We then all worked on actually generating the code until the back end was complete. The reference compiler was particularly useful when figuring out how to generate the code for a particular part of a WACC program, although we didn't follow the reference compiler completely, as we felt that it wasn't doing things the best way, even though it was still a correct implementation.

Design Choices

Front End

The first design decision we faced was the matter of choosing a language in which to write our compiler. We briefly toyed with the idea of using Haskell for its obvious advantages in terms of pattern-matching, but this feature was outweighed significantly by the object-oriented paradigm offered by Java once we began to develop a concrete plan of how to proceed. This decision was reinforced by the support offered by the department on the use of the ANTLR tool: Use of the ANTLR tool allowed us to focus on reforming a useful and correct structure of the WACC grammar, bypassing the creation of the generic methods and patterns used to build and traverse the parse tree. This slight restructuring of the grammar allowed us to actually carry out semantic analysis on our parse tree whilst simultaneously constructing a highly refined abstract syntax tree, eliminating a second pass over the parse tree. Examples of the changes we made to facilitate semantic analysis included: differentiating between function identifiers and variable identifiers, eliminating the mutual left recursion present in the definition of `<array-type>` and sorting the precedence of binary operators. This first pass is carried out using a visitor appropriately named `WACCFirstPass` which extends the class `WACCParserBaseVisitor` provided by ANTLR. This visitor handles all of the logic for semantic analysis, creating the symbol table and building the AST. There are five main points of discussion for our implementation of `WACCFirstPass`.

Handling error-messages with `ErrorMessageContainer`

Type-checking using `typeStack`

Perhaps the most fundamental part of carrying out semantic analysis was checking the validity and correspondence of types in WACC. Knowing this, we found an elegant solution to the problem: First, we created an abstract class `WACCType` which was extended by more specific type-representative classes such as `IntType` or `CharType` with methods that are used throughout the compilation process. We then create an instance of a stack in `WACCFirstPass` called `typeStack` which has an associated contract: when an expression is visited and its type is evaluated its argument types are popped from the stack and its return type is pushed onto the stack, if a type fails to match at any point, an error is passed to `ErrorMessageContainer`, and an instance of `AnyType` is pushed onto the stack. `AnyType` is a special sub-class of `WACCType` which matches with any other type - this stops an incorrect type from propagating upwards and causing unhelpful and indirect typing errors, thus allowing the semantic analysis to continue and detect the source of all semantic errors in one compilation.

Creating scopes with `symbolTable`

Our `symbolTable` is a custom data-type consisting of hashtables (representing scopes) each of which is connected to the previous scope. Each hashtable maps variable names, represented by strings, to their corresponding `Variable` object. These individual variable objects hold the variable's type and stack offset in the symbol table. A call to the `lookupAll` function with a string identifier will return the first variable object with a matching identifier to occur in the scopes above and including the current one - if an identifier is not found, then an `IdentifierUndeclaredException` is thrown, and this causes a semantic error to be added to the `ErrorMessageContainer`. Whilst visiting the parse tree in `WACCFirstPass`, whenever a new scope is encountered, we build a special `ScopeNode` and pass it a new `symbolTable` which is marked as the `currentScope` so that the visitor can add any declared variables to it.

AST construction and `funcTable`

The decision to construct the AST in the same pass as semantic analysis was made to pre-emptively reduce the number of passes over data-structures in our compiler. This meant handling two distinct pieces of logic in each parse tree node's visit method: Semantic analysis is always carried out first and is normally followed by the creation of the relevant `ASTNode`. If a semantic error is encountered, an invalid AST would be produced, but this doesn't matter as the compilation process does not continue having detected errors of any kind. An important decision we made resulted from our recognition that translation of functions could be carried out disjointly; our AST data-type is not one tree, rather it consists of a `funcTable` holding each WACC function as a separate syntax tree. We generalised this idea to handle the main body of code in a WACC program in an elegant way: the `funcTable` has a special `main` function which corresponds to the syntax tree representing this body. One potential problem that could have arisen from adopting this method is a name-clash with any user-defined function called `main`. This was easily solved: all user-defined function identifiers are implicitly pre-fixed by the string `"f_"` in `funcTable`. In `WACCFistPass` we are able to build the `funcTable` by having a `currentFunction` field whilst we do our visiting; this field is updated whenever we visit a new function declaration in the parse tree.

Prototyping

A problem we quickly encountered in relation to type-checking function calls in `WACCFirstPass` was that a function may not have been visited (and hence not added to `funcTable`). This means we have no knowledge of the function's type-signature and no way of upholding the `typeStack` contract. Our solution was to visit the immediate children of the program node and declare functions in the `funcTable` at the beginning of the first pass. The function bodies are then visited and the resultant syntax trees are mapped to the correct identifiers in the `funcTable`.

DONE UP TO HERE

(Talk about Register class, Operand Two class, messages/Labels and enums in each node for conditional commands)

A custom visitor pattern was followed to carry out the translation from the code into ARM 11 assembly. We passed over the internal representation of our code (the AST) which was created in the first pass recursively. This involved giving each node of the AST a general translate method which called a unique translate method held in a Translator class for that specific node. We would pass through the list of functions in the Function table and start the translation process from the ScopeNode at the head of each function body. ARM instructions were chosen to be represented as instruction node objects which implemented an ARMNode interface. They are added to a Linked list data structure which represented our program as we translated the body of each function. Each ARMNode had a unique toString method which was called when printing each instruction in the list after translation.

Predefined functions were handled by creating the `PredefinedFunctionHandler` class which contained a list of enums that represented predefined functions. The `addfunction` method would be called on this class with one of these enums as its argument and a switch statement would then add a branch statement to the predefined function in the program list. Another helper function is then called inside this class to add the rest of the arm nodes to a separate list which is put in a `hashMap` from Strings to List¡ARMNodes¿. The String key in the map is the unique name of the predefined function who's hash will be the same for equal strings. This prevents a predefined function being added to the map more than once.

Extensions

One of the extensions implemented was function overloading. This means that multiple functions can now have the same name, but different parameters, and the compiler will be able to determine which function to execute at a function call based on the types of the parameters given when the function is called. The main change required to allow function overloading was that the `Hashtable<String, Function> funcTable` field, which mapped the function name as a `String` to the corresponding `Function` object, was changed to `Hashtable<String, HashSet<Function>> funcTable`, which maps from the function name as a `String` to a `HashSet` of `Function` objects, which are all the functions with the same name but different parameters. This change meant that some methods related to the `FunctionTable` and `Function` classes had to be slightly changed, such as the `declare` and `lookupFunction` methods. Also, `Function` objects had to be given a `String baseId` field, which holds the overloaded function name (Can be shared by multiple functions), and a `String id` field, which holds the unique identifier for each function. This will be the label that is printed out in the ouput ARM assembly file. These unique identifiers would look like this for a function `f` that has been declared 3 times with different parameters: `f_f`, `f_1_f` and `f_2_f`.

CONSTANT EVALUATION HERE

BINARY OP IMMEDIATES HERE

All of the extensions added to our compiler are used whenever a program is compiled, and there is no way to compile a program in the compiler without these extensions. This is because the optimisations don't massively change the assembly code, but stops a lot of unnecessary loading into registers, and there is no reason that the programmer would not want this.

If we had more time, we would like to implement some form of efficient register allocation, such as graph colouring, as we feel this would greatly optimise our generated code. However, we didn't have time for this as data flow analysis would have to be done on our intermediate code, then a control flow graph would have to be constructed, then the interference graph would have to be constructed from the calculated live rangers, and then the actual colouring of the interference graph would have to be done. Another extension we would like to implement is peephole optimisation, as we noticed there were a few redundant instructions in our generated assembly code, such as a store instruction storing a value in a register to some memory location, directly followed by a load instruction loading the value at the same memory location back into the same register. However, there are not many of these redundant instructions produced, so it was not one of the extensions we chose to implement in the time frame we were given.