Report on WACC Project and extensions

Kam Chiu, Ho Law, Jiranart Vacheesuthum, Vasin Wongrassamee

December 18, 2015

1 Introduction

This report covers the details of the design of our WACC Project, and our reflections on the work. It includes the following sections:

- The Product
- The Project Management
- The Design Choices
  - Visitor Pattern
  - Instructions and Instruction Builder
- The Extensions
  - Control Flow Analysis
  - Function Overloading
  - Instruction Evaluation
  - Constant Evaluation and Propagation Analysis

2 The Product

We have decided to approach front end with ANTLR tools provided. ANTLR lexer is used to tokenise the input while the parser produces the context for visitor pattern. In the parser we've added extra locals to contain vital fields for the semantic checks such as the 'typename' which is used to compare types while visiting the parse tree by "passing up" the types from terminal nodes of the tree up to where comparison is required. These 'types' are defined by classes to represent the type and its internal structure (e.g. array). Deeper details will be covered in the design choices section. Nevertheless, the front end of the compiler is able to successfully parse a valid WACC program and produce a parse tree to represent the program as well as exit with appropriate error codes for invalid ones.

Our unique approach at the back end has allowed us to produce machine code within the same visit as the semantic check. 'Instruction' lists are used to store code produced along the visit, separated from the header and footer codes (e.g. messages and pre-defined functions). In order to be able to locate variable declarations on the stack, we build our instruction by using instruction 'fragments', which granted us to allocate and look up variable positions within one visit. The details are covered in the design choices section. The implementation of the back end is capable of generating accurate machine code, which executes correctly according to the written WACC program, meeting the functional specification.

However, the drawback of our WACC compiler comes along with generating ARM machine code within the same visitor as semantic checks. One visit certainly makes the compiler faster, but it sacrifices some readability of the code as our visitor class is now twice as long compared to the first milestone. We have tried out best to separate codes from two ends with clear comments but the readability is still not as good as the case where a different visit is performed for back end in a different class. This also follows to potential issues in future development. Now that the

machine code generation is implemented in the same visitor, optimisation has to be carried out here too, and to be able to turn different optimisations on and off, we will either have to sacrifice extra readability in the code or create a new visitor pattern for the optimisation. The latter one follows that there will be code repetition and bug fixes will have to be accomplished in two or more different classes.

Performance-wise, MyWaccVisitor, as mentioned above, visits the parse tree only once to accomplish syntax and semantic check as well as produce ARM code. This means it takes shorter time to execute the compiler then 2 visits approaches, making execution time the main 'edge' of our unique WACC visitor.

3 The Project Management

For front-end before starting any planning, we had each of our group members read the specification and made sure we understand what we had to do. We then planned how we should approach each analysis, especially how to implement our semantic checker (E.g. using a stack or visitor/listener class). Most of the workload was in the semantic analyser so we had one of our group members working on the lexical analysis and another on syntactic analysis. All four of us then worked on semantic analyser together using the visitor pattern, splitting and assigning different visit methods to each group member. From here, we made our own branch on Git in order to test our separate methods more effectively. We also had many shared class and methods between our visit methods and we implemented them along the way. The ANTLR tool was particularly useful in that we can create a local variable which it generates for us in each context. This allowed us to compare the types easily which also made it easier to debug later. Our group met in the labs to do most of the debugging as this allows us to communicate a lot more efficiently so changes made in some of the shared methods and class can be easily accommodated in our code. Front-end went fairly well with even workloads and working implementations although time management could be improved.

The way we approached back-end was similar to how we approached our front-end. We also used our own branch on Git again for each of our assembly code generation. We tried to split the work equally but we soon realised that some parts require very similar implementations (for example, the scope of variables in while and if) and this lead to uneven workloads. A way that we could have prevented this is to not split the work by syntax (E.g. print, while, if) but by how the syntax is implemented in assembly (if and while have similar scope implementation whereas print and branching have similar labels and format). We managed to have a working implementation before the deadline but again, time management could be improved as we fixed our final bug a couple hours before the deadline.

Each of our group members did an extension and their difficulty corresponds to how much work they did in the previous milestones. Extension went smoother due to the fact that there were less shared class and methods. We also started immediately after finishing back-end which gave us ample time to complete our implementations.

Overall the organisation of this project and the use of different tools went reasonably well. We had a clear idea of what we had to do and we also met the deadline that we set by ourselves. We discussed and brainstormed ideas of different ways to implement our compiler before starting any coding although a more in depth planning would have saved some time for our back-end. We had encountered very few issues with eclipse and other tools. Time management, especially for the first two milestones, could definitely be improved but generally, our whole project went very well.

4 The Design Choices

4.1 Visitor Pattern

The visitor pattern is used in the design of this compiler as we have realised that there are many different classes representing the different nodes of a parse tree, and that we require our program to perform a different action on each of the nodes. We have also foreseen the possibility of going through the tree multiple times to perform different actions like the checks in frontend and the code generations in the backend section. By storing each node specific actions in a separate visitor class, the visitor pattern provides us with a neat way to organise what we want to perform on each specific node when it is visited by a specific visitor. In addition, with this design pattern, the program does not need to know in advance the class of node it is visiting.

Using the visitor pattern we have avoided hard coding the different actions we want to perform on each class into each node class itself. This makes our parse tree more reusable, as we could have a different visitor visiting the same parse tree and performing, on each node, sets of actions different from the previous visitor. We have taken advantage of this characteristic when we implemented our extension. In the source code, there are multiple visitors, each visiting the same parse tree, however performing different actions as they visit each node class.

4.2 Instructions and Instruction Builder

We made the design choice to store the generated assembly code using the Instruction class while we visit the parse tree. The use of a list of Instructions allows more flexibility than simply concatenating strings. The Instruction class allows easy access to changing parts of an instruction after we have finished visiting the parse tree. This helps us compile the Wacc programs while only visiting the parse tree once. We store three Instruction lists: the header, which holds string messages, the body, and the footer, which holds common branches such as print string and runtime error. A valid Wacc program is compiled by concatenating the string representations of all Instructions in each list. The string representation of an Instruction is the concatenation of all fragments in the instruction. An instruction is an ordered list of instruction fragments, with pointers that mark 'changeable' fragments.

There are three types of fragments: a position fragment, variable fragment, and string fragment. The position fragment is used when a variable is declared. As the visitor runs, we do not know which position of the stack this variable shall be stored in, because we do not know the total stack size yet. Therefore the position fragment is simply a placeholder to mark the presence of a variable declaration. After the visitor visits the whole program, each position fragment is allocated a specific position on the stack, and is given the string representation of "[sp, #(int)]". A hash map maps the string of the variable declared to its position on the stack. A variable fragment occurs when a variable is accessed. Again it is a placeholder that will obtain the exact string representation after the visitor finishes, but by looking up the formed hash map. A string fragment is simply a string, used for parts of instructions that are known during the visits, and will not change.

Due to the complex construction of an Instruction, we decided to create an Instruction Builder. The builder simplifies and clarifies our code when we create instructions. It allows the gradual addding of components to form an instruction, instead of passing a very long list of fragments to the constructor. The builder avoids confusion when position fragments and variable fragments are used, and improves readability. It also has many default options for quickly creating ready-made instructions of common assembly codes. For example, move, load, and binary operation instructions can be created easily by just giving the source and destination as arguments.

5 The Extensions

5.1 Control Flow Analysis

WaccVisitorWithCFA class is a visitor extended from our basic visitor to support basic control flow analysis. It works by observing the expression context in if-else and while clause if it is true or false. This is followed by elimination of extra redundant code. For example "if true then c1 else c2" with this optimisation turned on will only produce code for c1, without branching. For while loop, true condition will trigger an infinite loop code and false condition will prevent any code from being generated. All these operate on Boolean switches, which are turned on when a true-false expr is detected. These switches control whether or not the ARM machine code should be produce for any section in that program.

5.2 Function Overloading

Our solution for making the language support overloading function is by labelling each function by its name followed by its type before storing it in a symbol table. An example of this would be a function named 'increment' which accepts a single 'int' argument. This function would be recognised as 'increment_int' in our program. This would then allow us to overload a function with the same name and a different argument type.

In the implementation of this feature, we have had to modify our method of visiting function to retrieve the types of the arguments or parameters first before looking it up, putting it into the symbol table, or generating the assembly code in the 'call' statement. This modification has not involve a change in our design.

5.3 Instruction Evaluation

We added an Optimise class which contains an integer array which holds the value of the registers and a hashmap which acts as the memory. Whenever a load instruction is detected, it will check whether or not the register already contains the value by using the array index as the register number. If it is, the instruction will be considered redundant and removed. Similarly, a store instruction into the memory will be checked with the hashmap which maps the integer (offset from the stack pointer) to its value. The memory and registers will be cleared after each branch and label instruction. The offset is calculated each time there is a 'SUB sp, sp'or 'ADD sp, sp' instruction. We also had to take into account that register could be updated with 'ADDS' or 'SUBS' instructions. A lot of string manipulations were used to extract the numbers (offset and register) and we had to traverse the instruction list once to find the redundant instructions.

5.4 Constant Evaluation and Propagation Analysis

This extension was developed in three stages, and can function as a whole to evaluate most constant values at compile time.

The first stage involves extending the existing visitor to change its behaviour when visiting binary operations. Instead of loading the left and right operands into two registers, then applying a binary operation on the two registers, the OptimisedWaccVisitor will first check if both operands are constant values. Constant values may be integers, characters, booleans, or variables that are classified as constant (see stage 2). If both operands are constant values, the instruction to load operands into registers will not be generated. The binary operation will be evaluated by the visitor and the result passed up the binary operations tree. The visitor's return type of Info is useful for this purpose, as it is a flexible type that can store most types of information, and it has a type field

to describe what type of result it is carrying. By detecting the type field of the Info returned when visiting two operands, we can know if the operands are constant, and if so, the type and value of the result evaluated at a deeper level of the tree. The constant value carried in Info is only loaded to register when we reach the top of the binary operations tree, or when the other operand of the binary operation is a variable, and cannot be evaluated at compile time. This means that once we meet the first variable in a binary operation tree, we cannot evaluate the constants that come after it. We attempt to tackle this in stage 3. For evaluating binary operations that might throw runtime errors, we use methods such as AddExact in Java's Math class, that will throw overflow errors. We catch these errors and simulate them in the output code as a direct branch instruction to throw the corresponding runtime error.

The second stage of this extension is the detection and replacement of variables that have a constant value. A variable is defined to have a constant value if it is never re-assigned. A variable (x) that depends on another variable (y) is not constant if (y) is re-assigned after (x) is declared. Only variables of types int, char and bool are included in this extension because they are most commonly used and manipulated in binary operations in the Wacc language. To detect variables that have a constant value, we create a new visitor Variable Visitor that visits the parse tree, and builds Variable dependecies objects, which are networks of each variable it comes across. This includes the variable declaration context, a list of each subsequent access of the variable, and a collection of listeners - variables which are dependent on this variable. We work by assuming that every variable is constant when we visit the variable declaration context, and we store the expression that this variable is assigned to. If the expression contains any other variables, we subscribe to that variable as a listener. When a variable is re-assigned, it is classified as not constant, and it also notifies all of its subscribers that it has been re-assigned. If a variable subscribes to another variable after it was re-assigned, it would not be notified until the next time the variable is re-assigned again. This fits the definitions we have set out, and maximises the number of variables that we can safely evaluate as constants. After the variable visitor finishes running, variables that are classified as constant will have their occurrences replaced. Visitors visiting the variable will be redirected to visiting the constant expression that it was declared to, and the variable declaration statement will not generate any code. This means that any variables that are declared but are unused will not be evaluated. Hence this optimisation causes the runtime error test DivByZero to fail, because the operation is no used and is no longer evaluated.

The third stage attempts to maximise the number of constants we can evaluate in compile time, when a binary operation tree has both constant and variable operands. The binary operation tree "1+1+1+x" can be evaluted to be "3+x", but the tree "1+1+x+1" will only be evaluted to be "2+x+1", despite being mathematically equivalent. This is because as soon as the visitor detects that one operand of a binary operation is not constant, it can no longer continue evaluation during compile time. To optimise the evaluation of all constants, we need to re-arrange the binary operation tree, so that the deepest (left-most) nodes hold constant values, and variables are in the upper (right-most) nodes. The BinopTreeReorder visitor visits the parse tree, and stores a list of constant nodes it encounters at upper levels. If it then finds a variable node deeper down the tree, it swaps the two nodes. While swapping nodes in a binary operation tree, the visitor makes sure that the correct binary operators are moved together with the operands that they are binded to. The visitor also takes care to preserve mathematical and logical operator binding strengths - for example, it will not swap nodes that are binded with multiply operators with nodes that are binded with addition operators.