WACC Report

Zeshan Amjad, Rohan Padmanabhan, Rohan Pritchard, & Edward Stow

1 The Product

Our compiler passes all of the supplied test cases, and over 60 additional test cases we wrote to cover areas (mostly in semantic checking) where we found the standard test suite to be lacking. As such, we believe that our compiler meets the functional specification.

Our compiler makes a total of 5 passes over various representations of the program (extra passes are needed for our redundant instructions removal). The first two passes are done by ANTLR to lex and parse the program. The third pass is required to build our own abstract syntax tree (AST). From this point on, it would be possible to produce assembly in one more pass however we chose to build an intermediary code before producing assembly. This requires two passes however it allows us to make efficient register allocation choices rather than using only the stack. We believe this to be a sensible trade-off as it adds a small amount of time to the compile time but provides a large performance boost during runtime.

Throughout the project, we endeavoured to design the Compiler in a manner that allowed for future extensibility, without unnecessary coupling. One example of this is the intermediary code, a representation of ARM assembly instructions, utilising `PseudoRegister`s in place of actual registers or stack locations. This allows us to add new language features without concerning ourselves with memory allocation, as it is possible to create new `PseudoRegister`s whenever we need more space. These `PseudoRegister`s are then replaced with physical registers or stack locations by the register allocator. This means that when adding in new features or functions, the hard work of managing space is done for you, and if we wanted to change the way we did register allocation by for example, using graph colouring, this is simply a matter of swapping out the register allocator object in the final stage.

On the whole, when designing and implementing our extensions, we found the existing codebase to be easy to build on top of, with simple processes for both augmenting existing functionality (extending our library of pre-compiled functions, for example) and creating entirely new constructs (such as classes).

2 The Project Management

To maximise our work efficiency and code quality, we frequently employed pair programming. This meant that there was always another person who understood the task to consult with when we got stuck. When appropriate we were able to work on separate subsections of a task within our pair. This gave us the speed advantages of having all 4 of us writing code but the help and consultation advantages of working in pairs. One example of this was splitting the back end into two distinct parts: generating *internal code* from the AST, and generating *assembly* from the internal code. Two of us worked on each section, and within internal code generation, we were also able to split the task up by each working on different visitors. Despite working independently, we were still able to assist each other when we got stuck, as we were working on very similar tasks.

One issue with splitting tasks like this was that we were sometimes relying on functions that the other pair had not written yet. We did our best to mitigate this problem by writing interfaces to define the interaction between our functions but we would sometimes need to modify these interfaces when we

discovered new issues. This did occasionally cause friction as changing the interface to make one thing easier would often make another harder. However, we were always able to strike a reasonable balance and work through it as a team.

To manage our time, we used an online Kanban board. We kept columns of tasks that were on hold, up next, in progress, and completed. This allowed us to easily visualise our workflow and ensured that it was always clear which task needed to be completed next.

In order to minimise time spent manually testing, we wrote test scripts that would automatically compile, link, and execute each test case. We then compared these to the outputs from the reference compiler (which we had saved to file using another script). However, one issue we encountered was that we were unable to run our test suite with programs that involved `read`. This was because our test suite did not support taking user input and, as such, we had to rely on testing these programs manually.

To collaborate on our code, we used Git as this was the version control system we all had the most experience in. We made sure to frequently merge and to work on separate files (or sections of a file) wherever possible to minimise merge conflicts. Each feature that we worked on was given its own branch, which was merged into `master` once the feature was complete. Once we had a working compiler, we also took great care in ensuring that only one person was ever merging to `master` at a time to make regression testing easier.

3 The Design Choices

The design of the abstract syntax tree (AST) was one of the most consequential and time consuming decisions in our compiler. This is because the AST is the link between the front-end and the back-end of our project. One key design choice we made was to include the variable scopes within the AST by generating scope nodes that fit above the first node that introduces a new scope (for instance a `while` loop), rather than building a hierarchy of scope tables. This meant we did not need to keep track of which nodes in one tree matched to tables in the hierarchy. We did this by having an abstract scope node class which was extended by other nodes that require their own scope. This allowed a single node to contain both the newly initialized variables in its scope and the code to be executed; each node then has a link to it's immediate 'parent scope node' to allow us to efficiently jump up the tree.

We also made extensive use of the visitor pattern. These visitors allowed us to easily traverse both the parse tree and the AST. We utilised our visitors in such a way that each node would also visit each of its children. This meant that the return value contained all the data required about every node beneath the current node in the tree. This meant that when writing each visit function we only needed to worry about the operations required for each node and could assume that each child and parent would be built correctly by a different function.

As mentioned above, clearly dividing the line between the theory of the assembly program and the practicalities of the specific hardware through the use of intermediate code and `PseudoRegister`s reduces the amount of coupling between processes. This means first that it allows us to parallelise our work and separately test the register allocation and the internal code in isolation, and second that the internal code generation can be applied to any type of hardware restrictions, simply by switching out the register allocator.

In addition, we wrote a dependency manager (for which we adopted a factory pattern) to handle the dependencies of our standard functions. For example, our standard function for array bounds checking depended on our standard function for throwing runtime errors. Using the dependency manager meant that whenever we called and included array bounds checking, we did not need to also remember to include runtime error. Whenever we would make a call to the `StandardFunctionsFactory` for a particular function, it would log that it's being used. Upon code generation, the factory would then perform a breadth first search to calculate all dependencies and include these in the output file.

In hindsight, there are a few areas that we would have designed differently from the beginning. One example of this was concerning how we handle scoping of variables. Despite a large amount of consideration upon designing the AST, there was a a test case that caused a lot of headache and design changing nearer the end of the project. Consider this code as an example:

```
int i = 0;
begin
    i = 3;
    int i = 9
end;
print i
```

Originally, we stored variable names as their own local identifiers within the scope nodes, so the variable `i` will be stored just as an `i`. The problem arises in this test case where we have already stored an `i` variable in the second (nested) scope node during AST generation, but are trying to access the `i` variable from the scope above. This means that when we go to reassign the variable `i` on line 3, with no concept of the order that the variables are defined, it will look for the most immediate declaration of this identifier, which will be the one within it's own scope assigned on the line below it. This is clearly not the desired behaviour, and as a result the program would output a `0` instead of a `3`. As we hadn't anticipated this edge case initially, we then had to go through changing the way we stored identifiers so that they had a unique variable name with respect to any scope, which was a fairly large design change so late in the project.

4 The Extensions

4.1 Efficient register allocation

We opted to implement the Linear Scan Register Allocation algorithm, which runs in O(n) time and is much faster to run than a graph colouring method. The Linear Scan approach utilises live intervals as opposed to the live ranges used by the graph colouring algorithm. Live intervals are computed based upon the first and last times a variable is used in the program; variables are then allocated a register for this duration. A variable's register allocation expires once the variable is never used again, freeing the register for use by other variables.

In the case where there are no free registers available for allocation, a variable must be selected to be spilled to memory. Spilling is an expensive process and it was important to minimise the total number of spills that occur. The heuristic we used to do this was selecting the longest-lived variable to be spilled, as this would minimise the number of spills needed in future.

4.2 Classes

WACC is now object oriented. In order to define a new class, we can use the following syntax:

```
class TestClass(string name, int[] values) {

    int getValue(int index) is
        int[] values = this.value;
        return value[index]
    end

}
```

The class fields go inside the curved brackets, and any methods we may want to define go inside the curly braces, object fields inside methods can be accessed with `this.field_identifier`. Classes can have other class types as fields, have methods that take and return class types, and will be accepted as

a type just like any other type after declaration of the class (including for example, having an array of class instances). Upon parsing, class methods become standard functions with one extra argument at the start, being a pointer to the class object. This means that whenever an object method is called, it's a simple front end task of taking that object pointer, adding it as an argument to the function, and logging this as a standard function call. As a future extension given more time, it would have been better to have allowed for function overloading, at which point we wouldn't need to append the class name to the start of the function name to allow for multiple classes to have the same method name, because instead the argument types would have been used to differentiate between different class methods.

Class fields are stored by allocating a single block of memory. Fields are stored in 2 different ways: `int`s and `char`s are stored as immediate values whilst everything else is stored by reference. This allows for efficient retrieval and space allocation of `char`s and `int`s which can be easily stored within the memory block itself. It also means that object freeing is easy too, the command `free [object instance identifier]` will free the single block of memory, wiping immediate values (`char` and `int` fields) and pointers to other fields, whilst leaving the other fields intact in case other pointers refer to them. It is the user's responsibility to 'deep free' any other class fields as they would have manually created them initially.

4.3 For loops

We implemented support for an additional looping construct: `for` loops. `for` loops integrate a common use case for while loops into a much more compact, readable syntax. The syntactic structure of `for` loops is very similar to that in Java, and shown below.

```
for int i = 0, i < 10, i = i + 1 do
    int j = i + 10;
    println j
end
```

The first block of a for loop must be an initialisation assignment: `type varName = value`. The second block is any expression that evaluates to a boolean value. The third block is an assignment statement of the form `varName = newValue`, where varName must already exist in the scope. The loop body has its own local scope which is inaccessible following the `end` statement. The initialisation block and boolean expression in the declaration of the `for` loop are bound to the loop's parent scope and not the internal scope of the `for` loop body.

4.4 Standard functions library

We added a standard functions library to WACC. These functions allow the user to call the following functions without having to write them themselves. These include functions to sort arrays, reverse an array, count the occurrences of an element in an array, check if an element is in an array, find the maximum and minimum array elements, and to calculate the value of one number to the power of another.

To save time during compilation, these standard library functions have been pre-compiled. As such, if a user calls one of these functions the only time it adds to the total compile time is the time taken to retrieve the assembly from the memory. To save space in the output assembly file, these functions are also only added in if they are called in the program. In addition, these library functions also use the aforementioned dependency manager to make adding new functions easier.

If we had more time, we would've made functions such as contains overloaded. At the moment a user needs to either call `containsInt` or `containsChar` depending on the type of the array. With overloading, this could be combined into a single function.

4.5 Instruction optimisation

There are 3 major types of instructional optimisations included in our compiler.

1. This optimiser removes instructions of the form `MOV <register1>, <register2>` where `register1 = register2`. These instructions occur when the register allocator is able to store two variables in the same register at different points. The code generated when visiting the AST treats them as being stored in unique locations, resulting in unnecessary moves.

2. We also optimise out case when instructions like `MOV R1, R2` are immediately followed by `MOV R2, R1`. This inefficiency happens for similar reasons to the previous one, as multiple variables get assigned to the same registers. The optimiser looks at the current and next instruction as it loops through the instruction list and removes both instructions where appropriate.

3. This optimiser replaces unnecessary pushes and pops. For each function call we have to `push` and `pop` R1, R2, and, R3. As a result, when two functions are called one after the other resulting in popping and then immediately pushing the same registers. To make this more efficient the optimiser replaces this pair of instructions with a multiple load instruction.

To coordinate these optimisations we added a flag -o that tells the compiler which optimisations to run and in what order. To further this extension, we could locate more inefficiencies that can be resolved solely by looking at the instructions, or by altering the instruction generation functionality in the AST visitor. An example of the latter optimisation type would be making use of immediate values in arithmetic operations.