WACC Report

The Team

- George Soteriou (gs2617)
- Bianca Casapu (bc3717)
- Iulia Ivana (imi17)
- Niranjan Bhat (nb1317)

1. The Product

Our WACC compiler is built on a strong foundation with distinct lexing, parsing, AST building and code generating steps. These steps are unique enough so that working on each one of these separately is possible and code is easily maintainable. Especially, the AST is strongly typed and keeps a very good abstract but informative notation of the code that generated it. The AST classes are also clearly readable and and can be updated with new features easily. The semantic checker applies a type check. Because the type information stored in the AST is sufficient, it is trivial to spot miss matching types in statements. In very few situations, the type is changed to strictly match lhs and rhs of expressions to further increase the strength of the type information stored (for example fully typed arrays) and this can be easily built upon in the future because of our infrastructure. Finally the code generator traverses the AST only once using all the information stored in it (including the built-in scope and dynamic generation of offsets from stack pointer) and creates a list of instructions. Adding optimisations on these instructions would be intuitive as they keep the context of the instruction in strongly typed classes. All the code is very readable, as our chosen language (Kotlin) and the structure of classes we have created makes this possible.

2. Project management

1. Git
    - Our repository was set up in such a way that the master branch only ever contained fully working, production quality code. For each of the three tasks, a develop branch existed (frontend, backend and extensions) to which features were added.
    - For any new branches were branched off of the develop branches, and features were merged back to them, before being checked together with other features and finally pushed to master. This ensured that we always had a good workflow when we implemented independent features without having to merge from other branches.
    - We also implemented a Gitlab runner that runs, builds and tests in order to check that by adding our new features we did not break any of the others already implemented. The tests were run with every commit and merge requests were only merged if the pipeline was successful.
2. Organization
    - We made use of pair programming, so that any code being written had at least two people looking at it and ensuring its correctness.
    - Upon merging into the develop branch for each of the extensions, we sat down as a group and made sure that everyone was quickly brought up to speed with the new features being added.

3. Issues encountered throughout the project

    - Deciding on how the AST should be structured was hard to do, as the whole of the project structure had to be taken into consideration. This took a lot of time and consideration, and had to go undergo repeated overhauls before we finally settled on a structure we thought was best. Though time consuming at first, going the extra mile to ensure that our AST was intuitive helped a lot later on, as writing code involving the AST was a breeze, and also greatly simplified the debugging process as we could easily spot errors.
    - We didn't have too much experience with writing and understanding assembly, so we heavily relied on understanding the assembly generated by the reference compiler. This meant that we were progressing quite slowly at first, but as we got used to how assembly works, generating it became more and more natural.
    - The initial Gradle setup was hard to do as it had to include ANTLR and kotlin as well as work with IntelliJ. We had to reset our progress quite a few times before getting it to work, but once we did, it was a very useful tool. At the end of the front end task, we still had trouble running our main function from terminal as we had to decide the correct way to make it run with Gradle. After a lot of testing, we ended up using Gradle's assemble task that built a binary file of our code and using that to compile.
    - Working on WACC alongside other courseworks was a big part of this project and managing our time to complete WACC proved to be quite challenging at times, but we managed to work through it.

4. What went well

    - Kotlin was a great language for the project, since it has backwards compatibility with Java, hence also with Antlr, and it also has functional programming features.
    - Testing was implemented quite well as JUnit was nice to work with and running a single command to test all WACC examples both locally and on the pipeline was very helpful. Adding single tests was easy to do as well because of the structure we used.
    - The type comparison implemented within semantic checking was a great tool to easily find semantic errors in the code, as well as make it easy to enhance and add array bounds checking later on.

3. The Design Choices and Implementation Details

1. Front-End

    - Lexing and parsing was done with Antlr and we then used the parse tree created with our own visitors, using the visitor design pattern, to create our more convenient AST.
    - The root of the AST is separated in a list of functions, each containing a block, and a separate block for the main code. Blocks are themselves statements. They hold a list of other sub statements, and a scope.
    - Scopes are linked together using an active scope class that points to a current scope, and a parent active scope. This is used for traversal up the nested scopes, when looking up a variable.
    - When a new variable is declared, it is added to the current block's scope.
    - After the AST is build the next step is semantic checking. The AST is traversed and for each statement, matching expression types are checked. Eg: an assignment must have a variable of the same type as the expression on the rhs. Variables types are also stored in the scope. In

addition, a variable must be declared before it is assigned but since all variables have been added to the scope already from the first pass, a declared bool is needed to complete that task. If a variable is assigned without its declare flag being set, an error is reported.
    - Also in `if` and `when` conditions, the expression should resolve to a bool type.
    - When a function is called, functions are looked up in the list of functions, and the type of each expression passed in must match the corresponding parameter of the function, as well as the return type must match the variable that will store the result.

2. Back-End

    - We used an Instruction Class that contains subclasses of all the types of instructions we used to generate our assembly code. Each of these subclasses uses operands that are defined as subclasses of the Operand Class that contain items such as registers, constants, literals etc. This made our instructions strongly typed and impossible to allow wrong instructions being generated.
    - `compileExpression` function takes an expression and a destination register and will generate the assembly required for computing the expression, and place the result in destination register. This function may be called recursively to generate subexpressions. In the case that the register requested is above the max used register, it will also handle pushing and popping the required registers from stack.
    - `compileStatement` function takes a statement and will generate assembly required for that statement. Because of our architecture, each statement will usually contain one or more expressions and `compileExpression` will be called to handle each of them. It will then use the result of the above expressions to fulfill the required function according to its type. In the special case of a block, it will call itself to add instructions for each statement in the block.
    - This design is intuitive to work with as each statement will have different functions and in the case that a similar functionality was observed in two different statements or expressions, a helper function is called to reduce code duplication.
    - Understanding and working with passing parameters to functions was hard to understand and work with but it wasn't a problem once we understood what was happening.

4. Beyond the Specification

- `for` and `do while` statements were implemented on the basis of while statements. A `for` statement compiles to a `while` statement with an embedded declaration before the loop and a counter increase as the last statement of the body of the loop. The `do while` simply adds the body of the while once before the `while` statement. This creates the effect of a `do while` loop.

  The format of the statements is as follows:

```
do
  <statement list>
while <condition expr>

for <declaration>; <condition expr>; <increment expression> do
  <statement list>
end
```

- We extended the `if` statement implementation to support not having an `else` branch, to avoid writing redundant `else` branches that contain just a `skip`. It is efficiently implemented, reusing assembly code for `if` statements, but with an effectively empty `else` instructions. Because of this change, `ifNoelse` test fails.

- Implemented `when` statements, which act nearly like the ones implemented by Kotlin. This statement contains an expression that will be evaluated, using a user-chosen comparison operator, with another expression found in the lines of the when statement. If none of the lines evaluate to true, the when statement won't do anything, as we decided not to include a default case, as a design choice.

  The format of the statement is as follows:

  ```
  when <expr> <comparator binop>
    <expr1> : <statement list>
    <expr2> : <statement list>
    .
    .
    .
  done
  ```

- Extended WACC language to support side-effecting expressions (`++, --, +=, -=, *=, /=, %=, &=, |=`), by compiling them as assignment statements.

- Extended WACC language to allow a semicolon at the end of the penultimate statement. This improves readability and uniformity of the WACC language. (This causes the `extraSeq` test to fail)

- Implemented a way to include other .wacc files into the program, effectively enabling headers and predefined libraries to be imported into the program. (A side effect of implementing include statements was that the WACC language had to be modified to allow for .wacc files to only contain function definitions, with no body. and therefore the tests `noBodyAfterFuncs` and `noBody` fail)

  The format of the includes is as follows:

  ```
  begin
    include <path to .wacc file1>;
    include <path to .wacc file2>;
    ....
  end
  ```

- Implemented function overloading, by allowing functions to have the same name, and possibly different parameters and return types. This was done by using the name of the function, its parameters and return type to generate a unique function name to be used throughout in the AST, and upon encountering function calls, looking up the unique internal function name to jump to the intended function.

- Implemented array bound checking. This was done by making array types stronger by storing the type of each object in the array. This helped us solve the problem of indexing multidimensional arrays as

each index may have a sub array of different length. Using this tool, every time an access to an array is made, the length of the corresponding array is checked and a semantic error is expressed if the index is out of bounds. Because of this change, 3 runtime error test fail as this is now longer a runtime error, but a semantic one.

- We also spent some time working on a sudoku solving project in wacc. There are two files that can be found under `src/test/resources/` called `sudoku.wacc` and `sudoku2.wacc`. They both take an unsolved sudoku grid and return a solved one in under a second. The first is written in the normal WACC language without any extensions implemented. The second one includes our extensions and can be used to make sure that the features we implemented are correct.