Imperial College London

WACC - Report

Imperial College London

Department of Computing

50007.3 - Advanced Laboratory 2

WACC Group 32:

Pranav Bansal (pb719)
Akshat Tripathi (at619)
Vlad Safta (vs719)
Tudor Stoiean (tas2618)

Date: 1st October 2021

1 The Final Product

Our WACC compiler meets the functional specifications set out in the WACC Language Specification. We verified this using a combination of internal unit tests and the test suite provided by the Department of Computing.

Our WACC compiler is well abstracted, with separate packages for each stage of compilation and special packages to handle typing, error messages and the symbol table. This has the effect of simplifying future development since new language features would need to change only a small number of files. For example, if we wanted to add a new compilation target, we would only need to create a new Emitter struct that implements the Emitter interface.

Our compiler was built with concurrency in mind, which leads to its strong performance characteristics. The compiler parses imported libraries and performs semantic checking concurrently, which give it large performance increases. Concurrency gives us up to 3 times better performance over a non concurrent implementation of our compiler.

We aimed to design the Compiler in a manner that allowed for future extensibility, without unnecessary coupling. For example, we used an intermediary code, a representation of ARM assembly instructions. We chose pseudo registers instead of actual registers or stack locations, since we can add new compilation without thinking of their specific register conventions. Then, the assembly emitter replaces pseudo registers with physical registers.

In the final version of our code, two of the original Expressions tests and Syntax Tests fail, because we did some modifications which overwrite certain preexisting rules of the WACC Language. Now we allow a program to not have any statements in its "main function", in order to have files with only structure, class and/or function definitions. The expression tests fail because of the "++" and "–" side-effecting expressions. These changes have been logged in the readmes/updatedTests.md file in the repository.

2 The Project Management

We roughly followed the SCRUM Agile methodology with daily stand-up meetings where each member of the group answered three pivotal questions:

i) What I did in the past 24 hours?

ii) What I am going to do in the next 24 hours?

iii) Any blockers I may have?

This helped the group keep track of the important design choices being made, catch changes that may cause conflicts later and resolve any difficulties being faced. These meetings also served as a good status update on the group's progress.

Git was used for version control and a feature based branching system was followed. A branch was made for every feature or bug-fix and was merged onto it's source branch after the task was completed.

After the first checkpoint, a "dev" branch was added where all the development took place. The tasks on the backlog were either taken up by a group member on their own or a task was assigned to a pair of members during the daily stand-up meetings. Programming in pairs proved to be beneficial as it was easier to discuss different approaches for a particular problem. The VS Code Live Share also allowed the pair to work in the same workspace concurrently.

When starting to work with the project, we decided to structure our tests into chunks order by our perception of their difficulty level. For the main part, there are 15 chunks, where each one of them can have folders named "syntaxErr", "semanticErr" and "valid". Each of the folder contains tests of the outcome described by their folder: syntax error, semantic error and valid output respectively. The extensions tests are written in a separate folder in tests, where the chunks are named corresponding to their new introduced functionality. The "syntaxErr", "semanticErr" and "valid" pattern is followed here as well.

In order to maximise the efficiency of testing, we wrote test scripts that would automatically compile, link, and execute each test case. Afterwards, the test script would compare these to the outputs from the reference compiler.

A Continuous Integration Pipeline was also added which helped spot bugs with ease. The pipeline also had a linting stage which enforced uniformity of code style throughout the codebase. We were also able to use the pipeline to track the performance of our code, for example during the front-end sprint, we noticed a significant decrease in our compiler's performance where the pipeline took 2 minutes to pass, where it had previously been 5 seconds. We used our pipeline to isolate the change which caused this decrease and revert it.

3 The Design Choices

The implementation language was Go and the parser generator was ANTLR.

3.1 Why go for Go?

The reason the group chose to go for Go was because of its advanced concurrency features and single control flow paradigm. Go believes in having only one way of doing everything which allows for uniformity throughout the codebase. Recently, Go's compiler was bootstrapped (i.e. Go's compiler was written in Go) which serves as empirical proof that Go is a good implementation language.

3.2 Parser Generator: ANTLR

The group decide to use ANTLR which supported and have a context free grammar for WACC. Reaching this decision was an interesting journey because the ANTLR documentation says that it provides support for Go. However, as we found out, ANLTR only supports the listener pattern and does not support the Visitor pattern, which is crucial to having good abstraction in our compiler. So after exploring other alternatives such as Pigeon, a PEG (Parsing Expression Grammar) parser, we finally decided to add support for the visitor pattern in ANTLR ourselves.

3.3 Problems Overcome

Go currently does not support generics, which was a problem when we needed to create a visitor for the generated AST. We were able to solve this problem using code generation with the genny package. [1] We created a visitor generator which would scan our AST package for node types, and use those to generate a visitor pattern template. Using genny we were able to fill in the types of this template before compilation, which allowed us to use generics in Go.

During the back-end sprint, we attempted to make a domain specific language which would simplify the code generation process. This language used regex and Go's templating engine[2], to parse rules and generate the appropriate ARM assembly. However near the end of the sprint we realised that the DSL would need to be Turing complete to fully satisfy its purpose, which lead us to switch to using an emitter pattern instead.

4 The Extensions

4.1 For loops

For loops integrate a usual case of use for while loops in a compact and readable way. The syntactic structure that we have settled for the for loops is very similar to the one in C:

```
for (int i = 1; i < 7; i = i + 1) do
   print i ;
   print " : " ;
   println i * 5 ;
done
```

The first block within the open parenthesis must be a new assignment that declares the iterator set to be used within the loop. It is incorrect to use a normal assignment here. The second block represents the condition that needs to be respected while the loop runs, which evaluates to a boolean value. The third block is an assignment that represents a change of value at the end of every iteration of the for loop.
The body of the loop has its own scope, and the variables within it are separated from the ones outside the loop. The iterator declared in the initialisation statement is bound to the scope of the loop.

4.2 Do while

The "do while" instruction offers a viable alternative for the while loop which comes in handy for certain edge cases. The structure is intuitively done on the basis of the existing while structure:

```
int i = 3
do
    i--
    print i ;
while i > 0
done
```

The notable difference between "while" and "do while" is that "do while" executes the statements in the body at least once.

4.3 Enhanced assignments

The "enhanced assignment" is a form of syntactic sugar for side-effecting statements. There are two types of enhanced assignments.

The first type is illustrated through an integer type variable, an enhanced operand, and another integer type variable. The enhanced operands are "+=", "-=", "*=", "/=", "%=". By using this structure, the identifier is modified in an analogous way for all operands as the following example:
$a+=3$ is equivalent to $a = a + 3$
The second type is either of the form $i{+}{+}$ or $i{-}{-}$, where i is an identifier. They are equivalent to $i = i + 1$ and $i = i - 1$, respectively.

4.4 Ternary operator

The ternary operator is a easier way of returning an expression which depends on a truth value. It has the following form:

```
int value = condition ? value_if_true : value_if_false
```

This is equivalent to:

```
int value
if condition
then
    value = value_if_true
else
    value = value_if_false
fi
```

The variables `value_if_true` and `value_if_false` have to be the same type. We decided upon this design in the interest of idiomatic programming.

4.5 Concurrent Compilation (and Imports)

Our compiler concurrently imports files, and performs semantic checks concurrently.
Our extension allows 2 types of import statement:

1. `import "path/file.wacc";`
2. `import "path/file.wacc" as alias;`

If the compiler encounters an import statement during ast generation, it starts a new goroutine which parses the file and generates its ast. The original goroutine waits for all of its children to terminate before continuing. We synchronise this process using mutexes and channels. A mutex protects the set of all visited files, whereas we send function definitions down the channel. All functions sent down the channel are received by a goroutine, which then adds the to the main AST being built.

4.6 Dynamic Arrays

We added the `make` keyword which would allow us to create variable length arrays. The code `make(type, size)` would create an array of type `type` with `size` elements in it. The free keyword has also been extended to allow arrays to be freed.

4.7 Standard Library

We created standard libraries for strings and arrays, in order to provide programmers with commonly used functions. `stdlib/strings.wacc` contains many functions found in C's `string.h`, such as `strcmp` and `strcpy`. `stdlib/arrays/wacc` contains simple functions for appending to arrays of integers. If we were given more time we would use it to develop the standard library, as well as introduce generics, to allow the standard library to be extended to all types.

4.8 Concurrency

Our concurrency implementation is based on Go's concurrency model. In Go a new Goroutine is created when a function call is prepended with the `go` keyword, this Goroutine then runs concurrently with the rest of the program, and cannot be joined back. We have added similar semantics to WACC, where if a function is called using the `wacc` keyword, the function will run in a new detached thread. We have added the mutex and semaphore primitives to allow for thread safe data structures, and to allow threads to wait for other threads.

4.9 Object Oriented Programming

We added support for classes and structures. Structures consists of fields of any names and types which can be accessed using the dot operator. Classes are just like structures but additionally have methods which can be accessed the same way. The object itself can be accessed in a class method by using the "this" keyword.

When instantiating either a structure or a class, each field needs to initialised by using the constructor which is just the class name followed by the arguments in curly braces. However, if an object is left uninitialised, it receives a default value of a null pointer. Upon instantiation, the objects are declared on the heap.

The following is a code snippet that shows how to declare a class, instantiate it, define a method and then call that method.

```
begin
    # Declaration
    class counter is
        int count

        int increment() is
            this.count += 1
            return this.count
        end

        # Instantiation
        counter c = counter{0};
        int inc_val = c.increment();
        # will print 1
        println c.count
    end
end
```

4.10 Future extensions

One extension which we were unable to complete was a micro implementation of the LISP programming language, based off the LISP micromanual [3]. This would be built on top of imports, classes, the standard library and dynamic arrays. Unfortunately due to time constraints, we were unable to finish this extension.

References

[1] cheekybits. Genny. `https://github.com/cheekybits/genny`, 2020.

[2] Go template. `https://golang.org/pkg/text/template/`, 2016.

[3] John McCarthy. A micro-manual for lisp - not the whole truth. *SIGPLAN Not.*, 13 (8):215–216, August 1978. ISSN 0362-1340. doi: 10.1145/960118.808386. URL `https://doi.org/10.1145/960118.808386`.