The DoC Lab WACC Compiler: User Manual

Second Year Computing Laboratory
Department of Computing
Imperial College London

1 What is WACC?

WACC (pronounced “whack”) is a simple variant on the While family of languages encountered in many program reasoning/verification courses (in particular in the Models of Computation course taught to our 2nd year undergraduates). It features all of the common language constructs you would expect of a While-like language, such as program variables, simple expressions, conditional branching, looping and no-ops. It also features a rich set of extra constructs, such as simple types, functions, arrays and basic tuple creation on the heap.

The WACC language is intended to help unify the material taught in our more theoretical courses (such as Models of Computation) with the material taught in our more practical courses (such as Compilers). The core of the language should be simple enough to reason about and the extensions should pose some interesting challenges and design choices for anyone implementing it.

2 WACC Compiler Reference Implementation

The lab has written a reference implementation of the WACC compiler which will enable you to explore the behaviour of the WACC language. The core of this compiler has been written in Scala, making use of the powerful parser combinator library Parsley (`https://github.com/j-mie6/parsley`).

There are two ways in which you can access the reference compiler:

1) via a web interface that lets you upload a WACC program and set the compiler options;

2) via a Ruby script that provides a programmatic interface to the web-service.

We discuss each of these in more detail below and provide a brief guide to help get you started with using the reference compiler.

2.1 Web Interface

A web interface to the reference compiler can be found on-line at:

- `https://teaching.doc.ic.ac.uk/wacc_compiler`.

As well an the interface to the reference compiler, the site provides some basic information on the project and links to related resources, the most important of these being a repository of example WACC programs (`WACC_Examples`) hosted on the department’s GitLab service (`https://gitlab.doc.ic.ac.uk/`).

For the interested: the site itself is just a simple HTML page which makes use of the popular Bootstrap framework. All of the dynamic content is managed by a small amount of JavaScript code which captures the contents of the input form and then calls a simple server-side Ruby script which passes this information on to the reference compiler. The compiler results are then passed back to the page as a JSON sting, which is processed and displayed by another piece of JavaScript.

2.2 Command-Line Interface

A Ruby script `refCompile`, which provides a traditional command-line interface to the reference WACC compiler, can be found on the web interface pages described above. The script is also included in the provided Git repository for the 2nd Year Compiler Lab and can be found in `/vol/lab/secondyear/bin` if working on a DoC lab machine. The script allows for programmatic access to the reference compiler, which you will probably find useful when testing your own WACC compiler.

The script is written in Ruby and requires the '`rest-client`' and '`json`' gems to be installed on your machine. These are included in the standard setup on a DoC lab machine.

3 Usage Information

To see a full list of the compiler's options, simply run the compiler without any arguments or with the `-h` (or `--help`) option:

```
prompt> ./refCompile --help
Usage: ./refCompile [options] <target.wacc>
  options:
    -p, --only-parse Parse only. Check the input file for syntax errors and generate an AST.
    -s, --only-typecheck Semantic check. Parse the file for syntax and semantic errors and generate an AST.
    -c, --full-compile Full Compilation (default). Run the full compilation process.
    -t, --target [ARCH] arget. Select target architecture (default arm32, options: x86-64-intel or x86-64).
    -o, --optimise Optimise. Run ARM Peephole optimisations over the generated assembly code.
    -a, --print-assembly View Assembly. Display ARM assembly code generated by the code generator.
    -x, --execute Execute. Assemble and Emulate the generated ARM code and display its output.
    -d, --directory Give directory of wacc files.
    -h, --help Show this message

  target.wacc: path to wacc program file to compile (or target directory if --directory option set)
```

Note that the 'Help' option `-h` (or `--help`) is not available on the web interface, but similar usage information is displayed on the website.

The reference compiler parses the input file `target.wacc` and, if this was successful, generates assembly code for the input file. The compiler expects the target file to have the `.wacc` extension and will throw a warning if this is not the case (although it will still try to run over the file).

By default (with no option flags set) the compiler runs the full compilation process, generating temporary files on the server that are cleaned up as soon as the compiler terminates. This mode can also be forced with the 'Full-Compile' option `-c` (or `--full-compile`), which overrides any other compiler stage options. If you want to see the contents of the compiler's output files, then you can use the 'View-Assembly' `-a` (or `--print_assembly`) option that displays the generated assembly code (if the compiler gets that far).

The reference compiler can be stopped midway through the compilation process, if you just want to know if a program is syntactically or semantically correct. The 'Parse-Only' option `-p` (or `--only-parse`) stops the compiler after the program has been run through the lexer and parser, whilst the 'Semantic-Check' option `-s` (or `--only-typecheck`) additionally performs semantic analysis of the target program. In both cases the compiler exits before generating any assembly code, so the 'View-Assembly' `-a` option will be ignored.

By default the compiler will produce ARM assembly code (`-t arm32` or `--target arm32`), but it can also be configured to produce x86_64 Intel code (`-t x86-64-intel` or `--target x86-64-intel`) or x86_64 AT&T code (`-t x86-64` or `--target x86-64`) via the corresponding 'Target' options.

The default behaviour of the reference compiler generates functionally correct, but only tolerably efficient assembly code. It tries to make sensible use of CPU registers, but doesn't spot repeated patterns or redundant operations. The generated code can be improved by telling the compiler to run peephole optimisations on the generated assembly code via the 'Optimise' option `-o` (or `--optimise`). Note that this optimisation is currently ARM-specific, thus only available for the ARM32 target architecture.

The reference compiler can also simulate the execution of the generated assembly code using the 'Execute' option `-x` (or `--execute`). This works for both ARM and x86_64 targets. The user will first be prompted for an input stream that will later be passed to the compiled program. For example, running:

```
prompt> ./refCompile -x wacc_examples/valid/print.wacc
```

will prompt the user for an input stream and then compile the program and simulate the generated code for the `print.wacc` program from the provided `wacc_examples` repository. Note that the reference compiler automatically imposes a 5 second timeout on the execution of any WACC program. This is to prevent long-running or non-terminating programs from clogging up the server.

The `refCompile` script additionally allows you to run the reference compiler over all of the `.wacc` files within a target directory with the 'Directory' option `-d` (or `--directory`). For example, running:

```
prompt> ./refCompile -d wacc_examples
```

Will run the compiler over all of the WACC programs in the provided `wacc_examples` repository If the 'Execute' `-x` option is also enabled, then the script will prompt the user for an input stream for each program.

You may find it useful to pipe the output from running the `refCompile` script to a file, particularly if you are running it in 'Directory' mode. For example, running:

```
promp> ./refCompile -x -d wacc_examples | tee refCompile.out
```

Note that the use of the `tee` command means that the output from the program will be sent to both the console and the output file `refCompile.out`. You will find this helpful when the script is asking you to provide an input stream for each program.

Acknowledgements

The WACC language and original reference compiler (implemented in Java, using the parser generator tool ANTLR `http://www.antlr.org/`) were developed by Tienchai Wirojsaksaree as part of a UROP placement with Mark Wheelhouse over the summer in 2013. That compiler was later extended to perform various optimisations by Luke Cheeseman and Zuhayr Chagpar as part of a UROP placement with Mark Wheelhouse over the summer in 2014.

The web-service and scripting front-ends were developed by Mark Wheelhouse, with additional improvements later added by Luke Cheeseman and Zuhayr Chagpar.

A new (and significantly improved) version of the reference compiler (impelemented in Scala, using the parser combinator library Parsley `https://github.com/j-mie6/parsley`) was written by Jamie Willis in 2022 and deployed behind the web-service interface in January 2023.

Over the years a number of students have contributed to the quality of the WACC Laboratory by suggesting changes to language specification document and reporting bugs in the reference compiler implementation. A list of these students can be found on the "WACC Bug Hunters" web-page at:

`https://teaching.doc.ic.ac.uk/wacc_compiler/bughunters.html`