The WACC Compiler: Milestone 2
Compiler Back-End

COMP50007.1 - Laboratory 2
Department of Computing
Imperial College London

Summary

So far in the WACC project you have only built the front-end of your WACC compiler. For this project milestone, you will now implement a full compiler for the WACC language. That is, you will add a code generator back-end to your WACC compiler for one of the `aarch64`, `arm32` or `x86-64` architectures (you may choose which).

Details

Recall the four stages of the compilation process:

1. **(Optionally) Perform Lexical Analysis:** splitting the input file into tokens (some parsing approaches omit this stage).
2. **Perform Syntactic Analysis:** parsing the tokens and creating a representation of the structure of the input file.
3. **Perform Semantic Analysis:** working out and ensuring the integrity of the meaning of the input file.
4. **Generate Machine Code:** synthesizing output in the target language, maintaining the semantic meaning of the input file.

You should already have built the front-end of your WACC compiler in Milestone 1, which should handle the first 3 stages. This means that you should be able to perform lexical analysis, syntactic analysis and semantic analysis on WACC programs to determine if they are valid. You should also be able to detect invalid WACC programs and report appropriate error messages.

For this milestone you need to work on the back-end of your compiler, the code generation stage. Your compiler must be able to produce assembly code, for one of the `aarch64` (`armv8-a`), `arm32` (`armv6`) or `x86-64` architectures (you may choose which syntax) when given a valid WACC program. It is likely that you may need to slightly modify your parser and/or semantic analyser so that they produce suitable output for the code generator.

As with the previous milestone, you will probably find it useful to refer to the reference implementation of the WACC compiler. The reference compiler generates some *simplistic* assembly code for each input file that it successfully parses. Looking at these output files should help you design the output for your own WACC compiler.

Important! During this milestone of the project you should be concentrating on the **functional correctness** of your compiler. You do **not** need to optimise the code that you generate, although doing so may give you a head-start on the final milestone.

Submit by 19:00 on Friday 28th February 2025

What To Do:

1. Familiarise yourself with the tools for assembling and emulating assembly code on the lab machines. The tools that you will need to use here will depend on your choice of target architecture.

   ARM `aarch64`: 64-bit ARM code will need to be cross-compiled for its intended target architecture. We have provided you with a `gcc` cross-compiler that will assemble and link `aarch64` assembly programs accordingly. This is installed on the lab machines as `aarch64-linux-gnu-gcc`. You can run the cross-compiler to generate code suitable for any `armv8-a` processor as follows:

   ```
   prompt> aarch64-linux-gnu-gcc -o EXEName -z noexecstack -march=armv8-a ARMCode.s
   ```

   We have also provided you with an ARM emulator which allows you to run these assembled programs. This is installed on the lab machines as `qemu-aarch64`. You can run the ARM emulator for a `armv6` processor as follows:

   ```
   prompt> qemu-aarch64 -L /usr/aarch64-linux-gnu/ EXEName
   ```

   ARM `arm32`: 32-bit ARM code will also need to be cross-compiled before it can be emulated, but a different cross-complier to the one above is needed for this. We have provided you with a `gcc` cross-compiler that will assemble and link `arm32` assembly programs accordingly. This is installed on the lab machines as `arm-linux-gnueabi-gcc`. You can run the cross-compiler to generate code suitable for any `armv6` processor as follows:

   ```
   prompt> arm-linux-gnueabi-gcc -o EXEName -z noexecstack -march=armv6 ARMCode.s
   ```

   We have also provided you with an ARM emulator which allows you to run these assembled programs. This is installed on the lab machines as `qemu-arm`. You can run the ARM emulator for a `armv6` processor as follows:

   ```
   prompt> qemu-arm -L /usr/arm-linux-gnueabi/ EXEName
   ```

   `x86-64`: 64-bit `x86` code (in Intel or AT&T syntax) can be assembled via `gcc` as follows:

   ```
   prompt> gcc -o EXEName -z noexecstack X86Code.s
   ```

   The resulting program can then be run directly on an `x86` system (which all Lab Linux machines are) via the command-line as:

   ```
   prompt> ./EXEName
   ```

Regardless of your target architecture, we suggest that you copy some of the code generated by the reference compiler for a simple program in that architecture and run this though the assembly and emulation instructions described above. You should be using these tools to test your generated code during this milestone.

2. Write the code generator for your WACC compiler. Your code generator needs to pass over your internal representation of the input program to construct assembly code that implements the desired behaviour. You could choose to work with the same structure as generated by your parser, or you may have enriched this structure during your semantic analysis of the program.

On completing this milestone, you should have a full compiler for the whole WACC language.

Testing:

Your compiler will be tested by an automated script which will first build your compiler:

```
prompt> make
```

and then run your compiler on each example WACC program

```
prompt> ./compile PATH/FILENAME.wacc
```

before assembling and executing (or emulating) the resulting assembly code via the process described above for your target architecture.

The `make` command should build your compiler and the `compile` command should call your compiler on the supplied file. You must therefore provide a `Makefile` which builds your compiler and a front-end command `compile` which takes the path to a file `FILENAME.wacc` as an argument and runs it through your compiler, either successfully producing assembly code for your target architecture in the file `FILENAME.s` (at the root level of your repository), or generating error messages as appropriate.

As with the previous milestone, your compiler should generate return codes that indicate the success of running the compiler over a target program file. A successful compilation should return the exit status `0`, a compilation that fails due to one or more syntax errors should return the exit status `100` and a compilation that fails due to one or more semantic errors should return the exit status `200`.

Important! If compilation fails due to syntax or semantic errors, no code should be generated.

The behaviour of your generated code should match the input/output behaviour of the code generated for the reference compiler (regardless of target architecture). To give a concrete example, the automated test program may run:

```
prompt> ./compile waccExamples/valid/print/print.wacc
```

and expect to successfully parse the input file `print.wacc` returning the exit status `0` and generating an assembly program in the file `print.s` (at the root level of your repository). The automated test program will then assemble and emulate your generated code, which should produce the output:

```
Hello World!
```

with exit status `0`.

The automated test program may also run your compiler over programs that are known to be ill-formed, for example by running:

```
prompt> ./compile waccExamples/invalid/syntaxErr/basic/skpErr.wacc
```

In this case it will expect the compilation to fail with exit status `100` and an error message along the lines of:

```
Syntax error in skpErr.wacc (11, 11):
    unexpected keyword end
    expected assignment or array index
    |
    |begin skp end
    |          ^^^
    |
```

Since the compilation process failed, we will not try to assemble or emulate any code.

To help you ensure that your code will compile and run as expected in our testing environment we have provided you with the Lab Testing Service: `LabTS`. `LabTS` will clone your `GitLab` repository and run several automated test processes over your work. This will happen automatically after the deadline, but can also be requested during the course of the exercise.

You can access the `LabTS` webpages at: `https://teaching.doc.ic.ac.uk/labts`
Note that you will be required to log-in with your normal college username and password.

If you click through to your `wacc_<group>` repository you will see a list of the different versions of your work that you have pushed. Next to each commit you will see a set of buttons that will allow you to request that this version of your work is run through the automated test process for the different milestones of the project. If you click on one of these buttons your work will be tested (this may take a few minutes) and the results will appear in the relevant column.

Important! In order for us to be able to test your generated assembly code, `LabTS` will need to know which architecture you are targetting. To specify this, you need to provide a **plain text configutaion file** called `wacc.target` in the root level of your project repository. This file should contain a **single string** that specifies your target architecture, one of: `aarch64`, `arm32` or `x86-64` (for Intel or AT&T syntax). If no such file is provided, then we will assume that your target architecture is `aarch64`.

Important! Code that fails to compile and run will be awarded **0 marks** for functional correctness! You should be periodically (but not continuously) testing your code on `LabTS`. If you are experiencing problems with the compilation or execution of your code then please seek help as soon as possible.

Debugging Arm Assembly Code

If you want to debug your generated Arm assembly code, then we recommend doing this via Gdb. You will need to install `gdb-multiarch` on your machine, then you can emulate your Arm code (as described above) adding `-g <port>` to the command options. Then you can run Gdb on this code as follows:

```
prompt> gdb-multiarch
   gdb> target remote localhost:<port>
```

You will need to replace `<port>` with a port number of your choice, e.g. `1234`.

WACC Compiler Reference Implementation

To help you with this lab, we have provided you with restricted access to a reference implementation of a WACC compiler. You can find a web interface to the reference compiler at:

- `https://wacc-vm.doc.ic.ac.uk`.

We have also provided you with a runnable JAR `wacc-reference-cli.jar` that provides command-line access to the web interface. You should ensure it is up-to-date by running `java -jar wacc-reference-cli.jar update`. A full user guide for the reference compiler is included in the `wacc-examples` GitLab repository and can also be found online.

For this milestone, your implementation should **not** directly mimic the behaviour of the reference compiler, as this does not produce any files. Instead, your compiler should create an assembly (`.s`) file at the root level of your repository that contains your generated assembly code. The basename of this file should be the same as that of the target WACC program provided to your compiler. For example, if the compiler is called on the file `foo.wacc` then the generated code should be in the file `foo.s`.

You can view the code generated by the reference compiler by running it with the `-a` (or `--print-assembly`) flag. You do **not** need to generate exactly the same code as the reference compiler (in fact we challenge you to do better!). You should, however, ensure that your generated code has the same input/output behaviour when assembled and executed/emulated. You can get the reference compiler to assemble and execute the code it generates by running it with the `-x` (or `--execute`) flag.

Note that (as before) we are **not** expecting your compiler to handle options flags, we will just be running your `compile` script as discussed above.

Additional Help Getting Started

You will need to ensure that your parser generates a tree describing the structure of the program which has been parsed. It is likely that you have already done this in the previous milestone, but you may want to enrich the structure of this tree to make code generation easier.

Your code generator will almost certainly take the form of a recursive pattern matching function.

It is a good idea to start with simple programs that use the exit system call to return values, rather than trying to implement print statements straight away.

We recommend that, at least initially, you avoid worrying about optimising your generated code. This milestone is principally concerned with the **functional correctness** of the code you generate. Of course, thinking about any future optimisations you will want to make is a good idea so that you can make sure that your code is designed with such extensions in mind.

Some other general points to consider:

- If you discover that some parts of your code from Milestone 1 do not fit with the requirements of this milestone, do not hesitate to apply significant modifications to your previous work.
- As before you are advised to implement iteratively. Do **not** try to implement every feature in one session.
- As with all labs in the second year, time management is key. Do **not** leave everything until the final week, you will **not** be able to complete the work in time: this is the number 1 milestone where students start failing tests because they just couldn't get everything done on time – use the first week!

If you want to assemble and emulate your generated code on your own machine, then you will need to install the appropriate `gcc` cross-compiler (`aarch64-linux-gnu-gcc` or `arm-linux-gnueabi-gcc` and ARM emulator (`qemu-aarch64` or `qemu-arm`. On Linux, these programs can be installed from the `gcc-aarch64-linux-gnu`, `gcc-arm-linux-gnueabi`, `qemu` and `qemu-user` packages.

Submission

As you work, you should *add*, *commit* and *push* your changes to your Git repository. Your `GitLab` repository should contain all of the source code for your program. In particular you should ensure that this includes:

- Any files required to build your compiler,
- A `Makefile` in the root directory which builds the compiler when `make` is executed on the command-line,
- A script `compile` in the root directory which runs your compiler when `./compile` is executed on the command-line.
- A configuration text file `wacc.target` in the root directory which specifies your compiler's target architecture.

`LabTS` can be used to test any revision of your work that you wish. However, you will still need to submit a *revision id* to Scientia so that we know which version of your code you consider to be your final submission.

Prior to submission, you should check the state of your `GitLab` repository using the `LabTS` webpages:
`https://teaching.doc.ic.ac.uk/labts`
If you click through to your `wacc_<group>` repository, and toggle to the WACC Backend Milestone, you will see a list of the different versions of your work that you have pushed. Next to each commit you will see a link to that commit on `GitLab` as well as a button to submit that version of your code to Scientia.

You should submit to Scientia the version of your code that you consdier to be "final". You can change this later by submitting a different version to Scientia. The submission button will be replaced with a confirmation message if the submission has been sucessful.

You should submit the chosen version of your code by 19:00 on Friday 28th February 2024.

Assessment

In total there are 100 marks available in this milestone. These are allocated as follows:

Functional Correctness	**40**
Basic Programs (`skip` and `exit`)	2
Sequencing (c_1 `;` c_2)	2
Input and Output (`read`, `print` and `println`)	3
Basic Variables and Types (declarations and assignments)	3
Expression Evaluation	3
Use of Arrays	3
Conditionals (`if` statements)	3
Loops (`while`-`do` statements)	3
Scopes/Nested Statements	3
Simple Functions and Return Statements	3
Nested Functions and Recursion	3
Runtime Errors	3
Heap Manipulation (`newpair`, `fst`, `snd` and `free`)	3
Backwards Compatibility: Correctly Identifying Syntax/Semantic Errors	3
Design, Style and Readability	**60**
Internal Instruction Representation Quality	16
Code Generator Abstraction	15
Pre-defined Functions/Widget Abstraction	5
Assembly Formatter/Printer Abstraction	8
Git Use	4
Continuous Integration Set-up	5
General Code Hygiene	6
LabTS 🥕 Mark	1

This milestone will constitute 40% of the marks for the WACC Compiler exercise. Your work will be assessed by an interactive code review session during the week beginning on Monday 3rd March, where personalised feedback will be given to your group.