The WACC Compiler: Milestone 2
Compiler Back-End

CO261 - Second Year Computing Laboratory
Department of Computing
Imperial College London

Summary

So far you have only built the front-end of your WACC compiler. 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 the ARM11 architecture.

Details

Recall the four stages of the compilation process:

1. **Perform Lexical Analysis:** splitting the input file into tokens.
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 fist 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 the ARM11 architecture (more specifically the ARM1176JZF-S processor found on the Raspberry Pi) 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* ARM 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 1st March 2019

What To Do:

1. Familiarise yourself with the tools for assembling and emulating ARM code on the lab machines.

   We have provided you with a `gcc` cross-compiler that allows you to assemble and link ARM assembly programs. This is installed on the lab machines as `arm-linux-gnueabi-gcc`. You can run the cross-compiler to generate code suitable for the ARM1176JZF-S processor as follows:

   ```
   prompt> arm-linux-gnueabi-gcc -o EXEName -mcpu=arm1176jzf-s -mtune=arm1176jzf-s 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 the ARM1176JZF-S processor as follows:

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

   We suggest that you copy some of the code generated by the reference compiler for a simple program 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 ARM11 code generator for your WACC compiler. Your code generator needs to pass over your internal representation of the input program to construct ARM 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.

   Recall that if you use ANTLR to generate an abstract syntax tree, then it can also output a base visitor class for this tree. This is a good starting point for traversing the tree structure and generating the corresponding code.

Once you have finished this milestone, you should have a full compiler for the WACC programming language.

Testing:

Your compiler will be tested by an automated script which will run the following commands:

```
prompt> make
prompt> ./compile PATH/FILENAME1.wacc
prompt> arm-linux-gnueabi-gcc -o FILENAME1 -mcpu=arm1176jzf-s -mtune=arm1176jzf-s FILENAME1.s
prompt> qemu-arm -L /usr/arm-linux-gnueabi/ FILENAME1
prompt> ./compile PATH/FILENAME2.wacc
prompt> arm-linux-gnueabi-gcc -o FILENAME2 -mcpu=arm1176jzf-s -mtune=arm1176jzf-s FILENAME2.s
prompt> qemu-arm -L /usr/arm-linux-gnueabi/ FILENAME2
  .
  .
  .
```

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 ARM assembly 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`.

Note that if a compilation fails due to syntax or semantic errors, we do not expect any code to be generated.

We will test your generated code by assembling it with the `arm-linux-gnueabi-gcc` cross-compiler and then emulating it with the `qemu-arm` emulator, as described above. The behaviour of your generated code should match the input/output behaviour of the code generated for the reference compiler.

To give a concrete example, the automated test program may run:

```
prompt> make
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 ARM11 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> make
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 "line 7:10 mismatched input `'end'` expecting { `'['`, `'='` }". 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: 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/advice as soon as possible.

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

We have also provided you with a Ruby script `refCompile` that provides command-line access to the web interface. Note that this script makes use of the `rest-client` and `json` gems (both of these are installed as standard on the lab machines).

A full user guide for the reference compiler is included in the `wacc_examples` GitLab repository and can also be found on-line.

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 ARM assembly code. The basename of this file should be the same as that of the target WACC program will. 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_asm`) 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 emulated. You can get the reference compiler to assemble and emulate 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 generating easier.

Your code generator will almost certainly take the form of a tree walker that visits each node of the tree and generates assembly code in a recursive fashion.

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.

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

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 run on the command-line,
- A script `compile` in the root directory which runs your compiler when `./compile` is run on the command-line.

`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 CATe 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 CATe.

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

You should submit the chosen version of your code to CATe by 19:00 on Friday 1st March 2019.

Assessment

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

Code Generation: basic programs (`skip` and `exit`)	4
Code Generation: sequencing (c_1;c_2)	4
Code Generation: input and output (`read`, `print` and `println`)	4
Code Generation: basic variables and types	4
Code Generation: expressions	4
Code Generation: use of arrays	4
Code Generation: conditionals (`if` statements)	4
Code Generation: loops (`while-do` statements)	4
Code Generation: scopes/nested statements	4
Code Generation: simple functions and return statements	4
Code Generation: nested functions and recursion	4
Code Generation: runtime errors	4
Code Generation: heap manipulation (`newpair`, `fst`, `snd` and `free`)	8
Backwards Compatibility: Correctly Identifying Syntax/Semantic Errors	4
Design, Style and Readability (includes CI set-up and internal assembly code representation)	40

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 the 4th March, where personalised feedback will be given to your group.