The C WACC Compiler: Supplementary Lab Exercise

Summary

In this exercise you will be implementing the front-end of a compiler for the WACC language in C. That is, you will be writing a lexer, a parser and a semantic analyser for WACC programs. To help you with this you will be provided with a detailed language specification, numerous code samples and access to a reference implementation of the WACC compiler.

Details

As you will recall from the lectures of the Compilers course, a compiler takes a source input file, does something "magic" to it, and produces an output file in the target language. During this laboratory exercise you will be performing this "magic".

Breaking the process into stages a compiler must:

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.

For this resit exercise you only need to create the front-end of a compiler (the first 3 stages given above). Your compiler front-end should be able to successfully parse any valid WACC program, generating some internal representation of that program. It should also be able to detect an invalid WACC program and generate appropriate error messages. In particular, when your compiler fails to parse an input file it should report the nature of the error.

In practice, it is now very rare to write a parser from scratch as there are a number of tools that exist to help simplify this process. For this exercise we have provided you with the Lex and Yacc lexer/parser tools (also known as Flex and Bison), which integrate well with C.

Submit by midnight on Friday 1st September 2023

What To Do:

1. Get the provided GitLab repository for this exercise by running the command:

   replacing `<login>` with your normal college login. You will be prompted for your normal college login and password.

   The provided repository is set up to illustrate the directory structure we are expecting you to use. It includes the following files and directories:

- The `src` directory contains a simple lexer, parser and printer for a basic calculator grammar. This is where we expect you to write your compiler code. There is more information on the provided code in the `README` files.
- The `compile` script should be edited to provide a front-end interface to your WACC compiler. You are free to change the language used in this script, but do not change its name.
- `Makefile` should be edited so that running `make` in the root directory builds your WACC compiler.
- The `README` files contain some further information about the files and scripts provided in this git repository.

2. Get the examples and documentation GitLab repository, `wacc_examples`, by running the command:

   This repository contains example WACC programs and supporting documentation that should help you with the lab, including:

   - The `valid` directory, that contains a number of examples of well-formed WACC programs.
   - The `invalid` directory, that contains a number of examples of ill-formed WACC programs. Some of these programs are syntactically invalid, while others are semantically invalid.
   - The `refCompile` script provides command-line access to the reference compiler. For more details on its use, see the user manual `WACCRefManual` in the `waccExamples` repository.
   - The WACC language specification `WACCLangSpec.pdf`, which describes the WACC language in detail.
   - The WACC reference compiler user manual `WACCRefManual.pdf`, which describes how to access and use the lab's reference implementation of a WACC compiler.

3. Familiarise yourself with the Lex and Yacc tools. We have provided you with token and grammar specification files and a simple abstract syntax tree (AST) structure that describe a lexer and parser for a simple expression language consisting only of `+`, `-` and integer literals. Try to extend these files so that you can handle more of the WACC language's expressions. You will probably find it useful to create some simple (non-WACC) test cases at this stage.

4. Write the lexer for your WACC compiler. All this needs to be able to do is to match valid input strings and convert them into tokens for your parser. You should be able to use the Lex tool to help you with this task.

5. Write the parser for your WACC compiler. You parser needs to take the tokens generated by your lexer and parse them into some internal representation of the program. You should be able to use the Yacc tool to help you with this task. We have provided you with an example AST structure that could be extended to the full WACC language. Your parser should be checking the syntax of your programs as it is creating this tree, generating errors as necessary.

6. Write the semantic checker for your WACC compiler. Your semantic checker needs to pass over your internal representation of the input program and check that the types make sense and that the program constructs are being applied in the correct fashion. The provided files include an example tree walker that prints out the structure of the AST. This is a good starting point for traversing its structure and analysing the program's properties.

Testing:

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

```
prompt> make
prompt> ./compile FILENAME1.wacc
prompt> ./compile FILENAME2.wacc
prompt> ./compile FILENAME3.wacc
  .
  .
  .
```

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's front-end processes either successfully parsing the file or generating error messages as appropriate.

Important! - Your compiler should generate exit 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 errors, we do not expect any semantic analysis to be carried out on the target program file.

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

```
prompt> make
prompt> ./compile wacc_examples/valid/print/print.wacc
```

and expect to successfully parse the input file `print.wacc` returning the exit status `0`.

It may also run:

```
prompt> make
prompt> ./compile wacc_examples/invalid/syntaxErr/basic/skpErr.wacc
```

and expect the compilation to fail with exit status `100` and an error message along the lines of "line 7:10 mismatched input `'end'` expecting { `'['`, `'='` }".

As a final example, the automated test program may run:

```
prompt> make
prompt> ./compile wacc_examples/invalid/semanticErr/while/whileIntCondition.wacc
```

and expect the compilation to fail with exit status `200` and error message along the lines of "Semantic error detected on line 8: Incompatible type at `15+6` (expected: BOOL, actual: INT)".

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 `lab2sqt_<login>` 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 limited access to a reference implementation of the 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. (These are both setup as standard on the lab machines in the main lab, room 219.)

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

- `http://teaching-dev.doc.ic.ac.uk/wacc_compiler/WACCRefManual.pdf`.

For this exercise, your implementation should mimic the behaviour of the reference implementation when it is called with the `-sc` (Semantic Check) flag. Most importantly, your compiler should generate the same exit codes as the reference compiler. You do not, however, need to generate exactly the same output (in fact we challenge you to do better than the reference compiler!). Note that 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

- We have provided you with a basic Lex/Yacc framework, but if you want to read more documentation, find some tutorials, or see answers to frequently asked questions, then there is a wealth of information at:
  - `https://developer.ibm.com/tutorials/au-lexyacc/`
  - `https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf`
  - `http://www.cs.man.ac.uk/~pjj/cs211/ho/node8.html`
- Think carefully about the design of your compiler. Poorly thought out design will slow down your development and make debugging significantly harder.
- Take the time to read the provided WACC language specification document and example calculator files. If you dive straight into coding you are likely to spend a lot of time undoing mistakes that could have easily have been avoided.
- We recommend that you implement your compiler iteratively. Do not try to implement every feature in one session, but instead try to work on supporting one language feature at a time.
- Use the reference compiler to help you debug your own compiler. If you are struggling to solve a problem, try writing a new test and observe the reference compiler's behaviour on that test.
- Manage your time carefully. Do not leave everything until the last day or you will find it very hard to complete this exercise.

Submission

As you work, you should *add*, *commit* and *push* your changes to your `GitLab` repository. Your 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.

No submission is required on your part, but your repository will be cloned shortly after the deadline for testing purposes. We will assume that the work you want us to mark will be in the most recent commit of the `master` branch, unless you inform us otherwise. It is, therefore, important that you check the final state of your `GitLab` repository using the webpages at `https://gitlab.doc.ic.ac.uk/`. If you click through to the `Projects` tab and then follow the `TF Lab Deployer 2223 Summer / Lab2SQT_<login>` link you will see a list of all the commits that have been pushed to your repository. From the project page you can also view the files under version control and access other useful data.

Assessment

To pass this Supplementary Lab Exercise, your WACC compiler front-end must correctly process at least 50% of the WACC programs passed to it by the automated test script. This must include successfully identifying at least 40% of the valid programs, at least 40% of the syntax errors and at least 40% of the semantic errors. Note that there may be programs used by the automated test script that are not included in the `wacc_examples` repository, so make sure that you test your compiler thoroughly.

We will additionally be assessing the quality of your error messages and the design and style of your code. You will receive a short report on your work via e-mail within two weeks of the submission date.