The WACC Compiler: Milestone 1
Compiler Front-End

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

Summary

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

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 first milestone you only need to create the front-end of your 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. Your compiler must also return the status of the compilation in its exit code (see the Testing section below for more details).

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 ANTLR parser tool, which integrates well with Java and most popular programming languages

If you **really** do not want to work with Java and ANTLR then you may choose to use your own language and tool-set. However, **you must check this with the lab organiser first** and you must be aware that such a choice will limit the support that the laboratory helpers will be able to provide you and a poor choice of language or tool-set could reduce your project marks. You must also understand that it will be up to you to ensure that your compiler can be built and run on the lab machines and LabTS.

Important! - JMC students who are not taking the Compilers module (COMP50006) are **strongly** advised to stick with the recommended Java language and ANTLR tool-set.

Submit by 19:00 on Friday 12th February 2021

What To Do:

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

   ```
   prompt> git clone https://gitlab.doc.ic.ac.uk/lab2021_spring/wacc_<group>.git
   ```

   replacing `<group>` with your group number (which you can find on the GitLab `Projects` page). 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 `antlr_config` directory contains simple example ANTLR lexer and parser specification files (`BasicLexer.g4` and `Basic.g4`), along with a script `antlrBuild` that builds the corresponding Java class files using the ANTLR libraries.
   - The `lib` directory contains the ANTLR library files in `antlr-4.7-complete.jar`.
   - The `src` directory is currently empty (apart from a simple README file) and is where we expect you to write your compiler code.
   - The `grun` script allows you to run the ANTLR TestRig program that can assist you in debugging your lexer and parser.
   - 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. Currently running `make` will call the `antlrBuild` script in `antlr_config` and will then run `javac` over all of the `.java` files found in `src`. `make` has also been configured to include the `antlr-4.7` library in the Java class-path.
   - The `README` file contains 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:

   ```
   prompt> git clone https://gitlab.doc.ic.ac.uk/lab2021_spring/wacc_examples.git
   ```

   You will be prompted for your normal college login and password. Note that you may already have cloned a copy of this repository as part of the first Compilers coursework.

   This repository contains numerous example WACC programs, a script that gives you command line access to the reference compiler and supporting documentation that should help you with this lab. In particular:

   - The `valid` directory contains a number of examples of well-formed WACC programs.
   - The `invalid` directory 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 allows you to access the WACC reference compiler from the command line.
   - The partial WACC language specification `WACCLangSpec.pdf` used in Compilers CW1.
   - The WACC reference compiler user manual `WACCRefManual.pdf` describes how to access and use the lab's reference implementation of a WACC compiler.

3. Familiarise yourself with the ANTLR tool. We have provided you with ANTLR specification files that describe a lexer and parser for a simple expression langauge 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. Set up a Continuous Integration (CI) pipeline so that you can monitor your progress on your compiler. We have already shown you how to do this with GitLabCI during the DevOps lab, but you can use any system that you like for the WACC project. However, we expect to see an automated process that is triggered for each push to your GitLab repo. You might also want to consider how you will control which tests are run, so you can avoid having a long period of "broken" builds early in the project.

5. 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 provided ANTLR grun tool to help you with this task.

6. Write the parser for your WACC compiler. Your parser needs to take the tokens generated by your lexer and parse them into some internal representation of the program. Your parser should be checking the syntax of your programs as it is doing this, generating errors as necessary. Again, you should be able to use the provided ANTLR grun tool to help you with this task. Note that ANTLR can generate a parse tree corresponding to the input program as well as visitor/listener patterns for traversing this tree.

7. 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. If you use ANTLR to generate a parse tree then it can also output a base visitor and/or listener class for this tree, which 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 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`.

To check the return code generated by your compiler you can view the special shell variable `$?` (for example, by running the command `echo $?` after your compiler has terminated). All commands record their exit status in this shell variable, overwriting its previous contents.

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 some concrete examples, 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: "Syntax error 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 `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 first milestone, your implementation should mimic the behaviour of the reference implementation when it is called with the `-s` (or `--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 ANTLR framework, but if you want to read the full documentation, find tutorials, or see answers to frequently asked questions, then there is a wealth of information at `http://www.antlr.org` and in the ANTLR e-book provided for this lab. We will also be demonstrating the use of ANTLR in the Lab Support Lectures.
- 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 ANTLR examples. 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. We also recommend that you structure you CI pipeline to support this iterative development process.
- 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 final week and do not underestimate the time it will take to work on the semantic checker.

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` web-pages:
`https://teaching.doc.ic.ac.uk/labts`
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 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 consider 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 successful.

You should submit the chosen version of your code to CATe by 19:00 on Friday 12th February 2021.

Assessment

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

Correctly Parsing Valid Programs	10
Correctly Identifying Syntax Errors	10
Correctly Identifying Semantic Errors	15
Quality of Error Messages	15
Design, Style and Readability (includes design pattern use, CI set-up and internal program representation)	50

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 15th February, where personalised feedback will be given to your group.