WACC Compiler: Project Report

Group 6: Paul Vidal, Saturnin Pugnet, Gregoire Yharrassarry and Corentin Herbinet

December 18, 2015

1 Product: Quality of our WACC Compiler

1.1 Functional Correctness

We have managed to build a compiler which correctly detects a program's syntax or semantic errors and displays a corresponding erorr message. If the program is valid, an Assembly file is created which produces the correct output when executed.

1.2 Future Development

The way we have built our semantic error checking does not allow an easy addition of certain new features of the WACC language. Type checking is indeed handled at a big cost, as we were not creating a good enough internal representation of the various types, which could lead to a lot of time spent expanding the types of the WACC language. Nevertheless, the variable table handles variable tracking very well thanks to its structure, which was extremely useful during the Assembly code generation. This table could therefore save a lot of time when adding new features to the WACC language, such as class implementation, where new forms of scopes are introduced.

The design of the Assembly code generation is the part of the project our group is the proudest of. The generation of the Assembly commands, through the Command interface, is done in such a way that we are easily able to traverse all commands at the end of the compilation and detect which commands can be optimized. This design allowed us to achieve some of the Assembly code optimisations in the extensions in a relatively short amount of time.

1.3 Performance Issues

The way our compiler is built requires four visits of the generated syntax tree after parsing the code. The first one is achieved by our syntax analyser to check if each function has a return statement. The second pass is made by our semantic analyser which checks the semantic correctness of the given program. The two last visits are made by the Assembly code generator to allow firstly the construction of an internal representation of the code and secondly to generate the Assembly code. For the sake of clarity and design, we decided to build our compiler this way as each step has its own usefulness. However, this creates a lot of redundancy in the crawling of the trees and it could have been done differently, with fewer visits of the trees, especially in the code generation section, where an intermediate representation wasnt essential but useful for the comprehension of the codebase. This is a good example of design that was prioritised over efficiency.

2 Project Management

2.1 Organisation of our Group

We split up the work by identifying when possible two separate tasks in each major task, in order to always work in pairs. For instance, two major task were notable in the Front-End: Lexer/Parser creation and Syntax/Sematic checkers. We decided for each of them to split the group in pairs: one pair was writing the Lexer while the other was working on the Parser. The same occurred for the Syntax/Semantic analysers and later in the Back-End. The idea was to work with different pairs throughout the project while still working as a group overall, as it was easier to design and correct our code.

One of the key aspects of the way we worked was to always try to code together to learn from each other and merge ideas. We discovered that combining design philosophies and ideas leads to a better overall design, which is harder to attain when working on your own.

2.2 Use of Project Management Tools

- *Version control with Git*

This project made our knowledge of Git far more accurate and precise than before and had a strong impact on our efficiency. We realised Git was not only a code storage cloud, but also a way to keep track of the overall progress of the group through detailed commit messages . This is where we understood the importance of every message we were submitting on git, which if badly written, could have a negative impact on the productivity of the group, as we might overwrite each others work.

- *Communication through Slack*

At the start of the project, we decided to use a communication tool used in quite a lot of companies nowadays: Slack. This messaging application allowed us to communicate between each other for issues regarding the project. It is a good filter to make the difference between personal messages and messages relating to work and in our case, the project, which were important and required a quick response most of the time.

2.3 Reflecting on our Project Management

The contribution and the open-mindedness of the group in the design field of the project allowed us to achieve a great structure for the overall project. Everyone understood that it could and had to contribute to the brainstorming sessions at the start of every milestone, allowing us to quickly move forward with a clear idea of what we needed to do.

The area where we struggled the most and that we tried to overcome throughout the project was the readability of the code produced. Many times, we managed to get stuck on a part of the project because we could not fix the problems of the code produced by one of our teammates, due to lack of comprehension. Nevertheless, everyone worked hard to overcome this difficulty during the project with the use of comments and better variabe names, making us more experienced programmers both individually and as a team.

3 Design Choices

3.1 Syntax Analysis Design Choices

The first step we took to achieve the syntax analysis of a program was to extend the basic DefaultErrorStrategy given by Antlr. This allowed us to personalize the error messages in each specific case, for instance a 'missing token' error.

We then created a tree Listener whose method syntaxError() would be called by the new ErrorStrategy every time a syntax error would be detected by Antlr during the compilation of the program. This allowed us to format our error messages correctly and display enough information for the message to be adapted to this specific error. We indeed printed the line numbers and surrounding code, as well as underlined the precise location of the syntax error, so the user could pinpoint his mistake and correct it accordingly.

Finally, we used a tree Visitor for two types of syntax errors that could not be found by the Listener: checking every integer could be written with 32 bits, and checking every function had a return statement. Our Visitor would therefore walk the entire tree, achieving the required checking every time it encountered an Integer Literal node or a Function node.

3.2 Semantic Analysis Design Choices

The way we decided to achieve the analysis of the semantic correctness of the program was using a tree visitor that crawled the tree built by Antlr. On each specific node, we performed the required checking, throwing errors if needed using a class ErrorReporter, specially created with a singleton pattern, which, given some arguments, was always printing the same formatted message with different information depending on the type and context of the semantic error.

To overcome the difficulty of the type checking, we then decided to build an internal representation of the types that could be later placed in the variable table to be used throughout the program. This representation was especially helpful during the Assembly code generation.

The Symbol Table, used to track all the program's variables, was the most important data structure of the whole project as it was essential to link the front-end and the back-end. We decided to create a tree where each node represented a scope (containing variables which had a name, type and various other attributes) and where each child of the node was representing a scope inside the scope the node was representing. The tree was doubly linked as we needed to go back in the parent scope as much as going visiting the childs scope.

We used another singleton pattern to represent the program's General Table. It holds throughout all the compilation the top symbol table (a dummy node with its children being the functions scope and the main scope) and a Function Table which was simply a mapping of function names to a function representation (holding the type of the arguments taken, the return type and other information) also helpful for semantic checking and Assembly code generation. As this table could be accessed from anywhere, it helped us to keep track at any moment of what variables were in the specific scope and which functions could be called.

3.3 Code Generation Design Choices

- *Command interface with class inheritance to be able to build all useful ARM commands*

Knowing that we would possibly need to optimise our commands in the extensions, we decided to create a Command interface, with a method generating the string representation of the command, and ARM instruction representations. Each ARM Command was extending a superclass, depending on the type of Command (e.g. the ADD command extended the LogicalCommand class) which implemented the Command interface. It therefore also implemented the method generating the string representation of the instruction that would be printed on the file. Therefore, all we had to do during our code generation was to construct commands by calling the right constructors, and finally return a list of commands.

- *FileWriter class which takes a list of Commands and writes them in the right Assembly file*

Once all the commands were created in a list, this list wasl passed to a FileWriter which was responsible for formatting the output file and writing the commands one by one on the file by calling the interface method generating their string representation.

- *Factory pattern to build the internal representation of the program*

Having an idea of what we needed to produce (a list of commands), we decided to build an internal representation of the program (a tree). All we had to do was to call on the main Program node a method generating the commands for this program. The program itself was creating some commands and then was asking its children to generate their own commands (without knowing what these children were). The key point of the structure was that each node was generating its own commands and was only asked by their parent to generate these commands. This recursive structure therefore allowed each node to create its own commands independantly.

- *Expression, AssignRhs, AssignLhs and Statement interfaces*

The internal representation of the program had to contain interfaces as some nodes required an Expression, a Statement, an AssignRhs or an AssignLhs child, all of which could take different forms. Therefore, when a node needed an integer expression for example, it was only given a reference to the interface Expression (but its concrete type was an integer expression) which had a method generateCommands(). This expression could generate its own commands as it knew its concrete type and implementation of generateCommands(). Therefore, the node could generate the commands needed for its own representation, and when it needed to add the commands of the expression it had a reference to, it simply called the method generateCommands() on this reference.

- *Hardware Manager singleton pattern to manage registers and memory*

Considering our structure, we needed something to communicate between a node and its child as it had no information on how the registers were managed and which one it should use to continue generating its commands. This is why we created a hardware manager that could be accessed from anywhere during the whole code generation to manage registers and tell the parent node which registers it should use to continue generating its commands. The hardware manager was also responsible for keeping track of the location of variables in the memory in order to compute the right offset when accessing them in a specific command.

4 Our Extensions

4.1 Optimisation: Control Flow Analysis

Building on top of our semantic checker, we added a control flow analysis tool which was evaluating, for IF and WHILE statements, if the condition field always had a fixed value. To do this, we created a class ExpressionAnalyser during the semantic check that were evaluating the boolean value of the condition. If it was always true or false in the case of an IF or always false in the case of WHILE, a function was in charge of rearranging the tree, deleting the unused child node and keeping the correct one. This allowed the Assembly code generator to only generate commands for the right branch in IF statements and skip the code generation in WHILE statements if the condition field was always evaluating to false.

4.2 Optimisation: Instruction Evaluation

Our objective during this extension was to optimise the generated Assembly code in order to reduce the amount of Assembly commands and also the amount of memory operations required (Load and Store) to make the program work correctly. In order to do this, we loop over the generated Commands and check whether we find some cases where an optimisation is possible. We have found three different cases where the Assembly code could be optimised:

- A variable is loaded into a register and then moved to r0 before a function is called in order to make that variable a function argument. In this case, we load whenever possible the variable directly into r0.
- A variable is stored from a register into memory and then loaded back into the same register from memory. In this case, we simply delete both instructions as the variable is already in the right register.
- An integer, boolean or character expression is loaded into a register then used in a Logical (ADD, AND, etc.) or Comparison (CMP) command. In this case, we use whenever possible the expression directly in the logical operation or comparison instruction.

Even though it is an inefficient process, it is theoreticaly correct to loop multiple times over the Assembly code as one case of these optimisations could impact another.

4.3 Optimisation: Constant Propagation Analysis

Following the same design as the Control Flow Analysis, we decided to implement a constant propagation analysis tool in our compiler. We were remembering in our symbol table the value of the variable each time there was an initialisation or an assignment statement (except in special cases such as loops). This allowed us to simplify the tree each time we were meeting an expression. Using the tools created for control flow analysis that allowed us to calculate the specific value of an expression, we computed that value when possible and replaced the partial tree of the whole expression with a new node holding only the simplified value. Once the rearrangement was done, the Assembly code generator was crawling the new tree with the same behaviour as usual. The extension allowed the visitor to see the simplified expression and generate commands only for this simplified expression, saving a lot of manipulation commands that were generated before to calculate the expression value.