50007 - Wacc project report

Zhili Tian (zt719@ic.ac.uk)
Yeye Chen (yc1319@ic.ac.uk)
Boan Zhu (bz2818@ic.ac.uk)
Xingzhi Qian (xq219@ic.ac.uk)

March 18, 2021

1 The Final Product:

By the end of the second part of the project, out compiler meets most of the functional specification of the project, as it passes all the given tests except the "badIntAssignment".

We built both unit tests and integration tests to make sure the compiler have a sound basis. Most importantly, it not only passes basic tests, but also works on advanced tests. Out of the consideration for the future development, our design separates all stages (lexer, parser, visitor, code generator); extensive amount of inheritance and abstract classes are used to make things easier to be added or modified.

Our compiler should have the average performance. We use ANTLR base lexer and parser to generate parse tree. The parse tree would only be traversed once to building AST, symbol table while checking syntax and semantic errors. Then the assembly codes should be generated by visiting AST once. There are flaws in our compiler. It could not handle bad integer assignment; It does not support multiple error report; No optimization for the assembly code.

During this project we learnt a lot, it is not only we had a good experience to write a compiler by ourselves, but also started to know how to build a large project using all kinds of tools (e.g., GitLabCI and Maven) and how to work as a team. All in all, this is a very helpful precedent for future development.

2 Project Management:

The way we organize our group is to work as pairs, where we usually split the workload for two pairs to do at the same time. In this way, we can have someone to discuss about if we encounter any problems, or even if we are stuck. We would digest the problems within a pair first, and if that doesn't work out, we would turn to the other pair for help. This turned out to be the best for us in terms of speed and efficiency.

We use Git as our project management tool. We use different branches for different features that we wish to implement, then merge them into the master branch. We put usernames in front of each commit messages to indicate which particular feature of the compiler is implemented by whom, so that we can refer to particular group members if we have any issues with that feature. One thing that we could improve is to merge more frequently. We messed up the master branch once by merging two different branches where they have some conflicts, and eventually resolved by reverting to a previous version.

In addition, we have used maven and built up our own CI/CD pipelines to run tests on GitLab to check that if we meet the requirements, which is quite efficient and convenient. This is not only a good tool for automated testing, but also to check that if our newest implementation fits previous behaviors of our compiler. If not, we are alert that something went wrong and would fix the problem immediately. Otherwise, this may lead us to miss critical bugs in our project.

What we did not do well is that although we worked in pairs and divided our jobs, we didn't set a deadline for our works. So we ran out of time in task 1 and we didn't have enough time to implement AST structure entirely and pass all the tests. Compared with other groups, we are a little behind in progress. However, we improved our time management at task 2 and try our best to make up for the problems left by the previous task. Finally, we successfully pass all the tests at the end and got a much higher marks than the previous task.

For the latter parts of the project, we did our testing manually, which takes quite some time. What we will do if possible, in the next lab, is to write test scripts that would automatically compile, and execute each test case that are given.

3 Design Choices and Implementation Details:

We have written our compiler in java, as the ANTLR tool can help us build our own Wacc visitor and listener, which helps and increases the efficiency of the project.

In the semantics checking stage, we used the visitor pattern that ANTLR supports, traversing the parse tree and building the abstract syntax tree at the same time. This avoids traversing the program twice for building the AST, which would potentially lead to worse performance of our compiler.

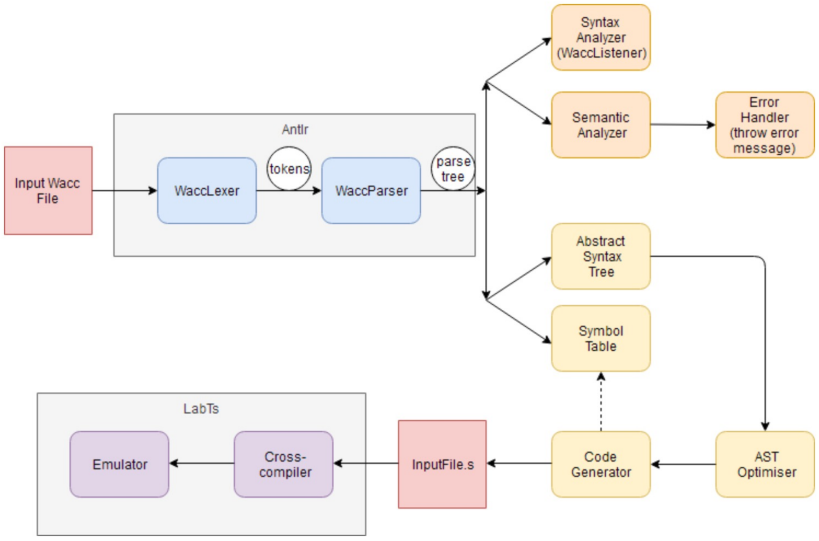
A good design we made during sematic checking stage is the Type class, which is an object that represents all kinds of Wacc data type and is hold by each expression node. Type is recursively defined so it is able to hold nested data types like array and pair. There is also an 'All-type' option in order to clear and compare expressions with unclear pair expression - pair (pair, pair), empty array liter expression - [], etc.

In our implementation of AST, we build lots of different nodes where they are divided into statement nodes and expression nodes. These two kinds of node are represented as abstract classes, extending the super class Node and

are inherited by non-abstract sub-nodes. In this design, we can easily add more features for our compiler if we want to. Each node should contain fields as few as possible, as we want to minimize the size of AST. For example, we extend our compiler to support do-while loop in the extension stage, by simply adding a 'DoWhileNode' as one of the statement nodes, and corresponding translator in our code generator.

Code-generation:

Label system:



4 Beyond the Specification:

Easy:

Additional Loop Control Statement: We implement for loop, do-while loop, break and continue. Both for loop and do-while loop support variable redeclaration. The for loop is specially well structed, as all three attributes are optional. For example, one can either code "for int n = 0; n < 10; n = n + 1 body" to use its normal function, or say "for ; ; body" to use it as a while true loop. However, the compiler is currently not strong enough to warn user if one writes an endless loop. The break and continue should work as intuition, which jump to the end and begin for current innermost loop. Semantic error reporter now should include checking if break or continue keyword appear in global scope.

Additional Branching Statement: We implement if branch (without else branch) and switch branch. For the switch branch, we allow not only switch by variable, but also by pair-elem or array-elem. Nested branch or loop inside any switch branch is supported. Variable redeclaration is supported. We also make the default branch as an optional choice for the users.

Bit-wise Operators: We implement '&', '|' as extra binary operation and '~'

as extra unary operation. The semantics errors are carefully handled as bit-wise operation should only apply on int type.

Macros: We implement variable macros defined before functions defined. Macro variables work exactly as normal declaration statements, they can be accessed and modified in global scope. We strictly ask user to name macros by using only underscore and uppercase letters. Semantic errors are therefore extended.

Optimisation – Constant Evaluation: We implement the constant evaluation for all binary and unary operations. Expressions like "not true", "1 + 2" should now be evaluated as "false" and "3" before the code generation stage. This is realised by adding one more stage – the AST optimisation right after AST generation. Constant evaluation is applied here by traversing the AST once and modified node if one can be simplified. This optimisation is optional and bound with level o1, so one can choose to turn it on by typing "./compile file.wacc -o1".

Medium:

Full Pair Types: We extend and modify the grammar to implement full pair types. Thanks to the Type class, we do not change much code. It is actually an easier language grammar under our design pattern, as all types now can be strictly and clearly typed without worrying compare a pair of all-type and a pair of any other types.

Summary Future Extension:

We carefully build each extension by adding unit tests and always run back-end tests immediately after pushing the updated code to make sure new features would crush with previous sections.

For the future extension, we would like to add more optimisation such as control flow analyse, instruction evaluation, efficient register allocation, constant propagation analyse and so on. Then if we still have time, we would head to implement class and inheritance.