# Proposal Draft Martin Walls

Project title: C to WebAssembly Compiler

## Overview

With the web playing an ever-increasing role in how we interact with computers, applications are often expected to run in a web browser in the same way as a traditional native application. WebAssembly is a binary code format that runs in a stack-based virtual machine, supported by all major browsers. It aims to bring near-native performance to web applications, with applications for situations where JavaScript isn't performant enough, and for running programs originally written in languages other than JavaScript in a web browser.

I plan to implement a compiler from the C language to WebAssembly. C is a good candidate for this project because it is quite a low-level language, so I can focus on compiler optimisations rather than just implementing language features to make it work. Because C has manual memory management, I won't have to implement a garbage collector or other automatic memory management features. Initially I will provide support for the stack only, and if time allows I will implement `malloc` and `free` functionality to provide heap memory management.

I will compile a subset of the C language, to allow simple C programs to be run in a web browser. A minimal set of features to support will include arithmetic, control flow, variables, and functions (including recursion). I won't initially implement linking, so the compiler will only handle single-file programs. This includes not linking the C standard library, so I will provide simple implementations of some of the standard library myself, as necessary to provide common functionality such as `printf`.

I will use a lexer and parser generator to do the initial source code transformation into an abstract syntax tree. I will focus this project on transforming the abstract syntax tree into an intermediate representation—where optimisations can be done—and then generating the target WebAssembly code.

I plan to write the compiler in C++, because I have a reasonable level of familiarity with it, and there exist well-established lexer and parser generators for it. For example, Flex and Bison respectively, which are included as standard on most GNU/Linux systems.

I will test and evaluate the compiler with a simple test program, which will calculate and output the first ten Fibonacci numbers. So that I can test and evaluate optimisations I add to the compiler, I will write the test program with some inefficiencies to give the compiler scope to improve it.

I will use a three-address code style of intermediate representation, because this lends itself to perform optimisations more easily. For example it's easier to see the control flow in three-address code compared to a stack-based representation. To transform from abstract syntax tree to the intermediate representation, this will involve traversing the abstract syntax tree recursively, and applying a transformation depending on the type of node to three-address code.

To transform from the intermediate representation to WebAssembly, I will need to convert the three-address code representation into a stack-based format, since WebAssembly is stack-based. This stack-based format will have a direct correspondence to WebAssembly instructions, so the final step of the compiler will be writing out the list of program instructions to a WebAssembly binary file.

## Starting point

I don't have any experience in writing compilers beyond the Part IB Compiler Construction course. I haven't previously used any lexer or parser generator libraries. I have knowledge of C++ from the Part IB course but haven't written a larger program in C++ outside the course.

I have briefly looked up the instruction set for WebAssembly and have written a single-function program that does basic arithmetic, in WebAssembly text format. I used `wat2wasm` to convert this to a WebAssembly binary and ran the function using JavaScript.

I have briefly researched lexer and parser generators such as Flex and Bison to see what's out there and to help decide on which language to write my compiler in, but I haven't used them before.

## Success criteria

The project will be a success if:

- The program generates an abstract syntax tree from C source code.
- The program transforms the abstract syntax tree into an intermediate representation.
- The program generates WebAssembly binary code from the intermediate representation.
- The generated binary is correct with respect to the original source code.

## Optimisations

At first, I will implement peephole optimisations, which are optimisations that look at short sections of code and match them against patterns we know can be optimised, then replacing them with the optimised version. For example, redundant operations can be removed, such as writing to the same variable twice in a row (ignoring the first value written), or a stack push followed immediately by a pop. Null operations (operations that have no effect, such as adding zero) can also be removed.

Constant folding is an optimisation that performs some arithmetic at compile time already, if possible. For example, the statement `x = 3 + 4` can be replaced by `x = 7` at compile time; there is no need for the addition operation to be done at runtime.

These optimisations will be run in several passes, because doing one optimisation may then allow another optimisation to be done that wasn't previously available. The optimisation passes will run until no further changes are made.

The stack-based peephole optimisations (such as removing pushes directly followed by a pop) will be done once the three-address code representation has been transformed into the stack-based format in the final stage.

## Extensions

Extensions to this project will be further optimisations.

One optimisation would be unused variable elimination, which looks through the code for any variables that are written to but never read. These variables are removed, saving processing power and space.

Another optimisation would be to perform dead code analysis to remove whole blocks of code that will never be executed, since they cannot possibly be reached. This will involve control flow analysis to determine the possible routes the program can take.

## Evaluation

The main evidence to show the compiler working will be the generated target code that it outputs, when given the test program as input. I will show that the generated code is correct with respect to the source code.

Additionally, I will evaluate the compiler optimisations by running the compiler once with optimisations enabled and once with them disabled. I will then benchmark the performance of the output program to identify the impact of the optimisations on the program's running time, and I will also compare the size of the two programs to assess the impact on storage space.

## Work Plan

1. **14th - 28th Oct**

   Preparatory research, set up project environment, including toolchain for running compiled WebAssembly.

   **Milestone deliverable**: Test program written in C that calculates and outputs the first ten Fibonacci numbers; This will be the target for the compiler to be able to compile. The test program will deliberately include inefficient/redundant code to allow optimisations to be demonstrated. A handwritten WebAssembly implementation of that program.

2. **28th Oct - 11th Nov**

   Lexer and parser generator implementation. This will involve writing the inputs to the lexer and parser generators to describe the grammar of the source code and the different types of tokens. When the generators are run with these inputs, they will generate code for the lexer and parser respectively, which will be used by the compiler.

**Milestone deliverable**: Lexer and parser generator inputs. The compiler will be able to generate an abstract syntax tree (AST) representation of the test program.

3. **11th - 25th Nov**

   Implementation of transforming the AST into the intermediate representation. This will require defining the intermediate code to generate for each type of node in the AST.

   **Milestone deliverable**: The compiler will be able to generate an intermediate representation version of the test program.

4. **25th Nov - 9th Dec**

   Implementation of target code generation from intermediate representation. For each type of instruction in the intermediate representation, I will need to define the transformation that generates WebAssembly from it.

   **Milestone deliverable**: The compiler will be able to generate target code for the test program. The generated WebAssembly will be able to be run in a web browser.

5. **9th - 23rd Dec**

   Implementation of peephole optimisations, as described above.

   **Milestone deliverable**: The generated target code of the test program with optimisations applied. Evidence to show the impact of the optimisations.

   *3 weeks off over Christmas*

6. **13th - 27th Jan**

   Finish implementation of optimisations if necessary.

   Write progress report.

   **Milestone deliverable**: Completed progress report.

7. **27th Jan - 10th Feb** (will be more busy here with extra-curricular events)

   Working on extension optimisations if implementation goals are met, otherwise slack time to finish the implementation.

   **Milestone deliverable**: Finished implementation, with extensions if completed.

8. **10th - 24th Feb**

   Evaluate the compiled WebAssembly, including correctness and impact of optimisations.

   **Milestone deliverable**: Evaluation chapter.

9. **24th Feb - 10th Mar**

   Write introduction chapter.

   **Milestone deliverable**: Introduction chapter.

10. **10th - 24th Mar**

    Write preparation chapter.

    **Milestone deliverable**: Preparation chapter.

11. **24th Mar - 7th Apr**

    Write implementation chapter.

    **Milestone deliverable**: Implementation chapter.

12. **7th - 21st Apr**

    Write conclusions chapter.

    **Milestone deliverable**: Conclusions chapter.

13. **21st Apr - 5th May**

    Adjust dissertation based on feedback.

    **Milestone deliverable**: Finished dissertation.

14. **5th - 12th May**

    Slack time in week up to formal deadline, to make any final changes.

## Resources

I will primarily use my own laptop for development. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

I will use Git for version control and will regularly push to an online Git repository. I will clone this repository to the MCS and regularly update the clone, so that if my machine fails I can immediately continue work on the MCS.