Intro

My project is to build a compiler from a subset of C to WebAssembly.

My project is on track and I have met my success criteria.

Source code → AST

First step: used a parser generator to generate an AST from the C source code.

I wrote the grammar to describe the C language as input to the generator, making sure to avoid ambiguities.

I ended up having to write a custom lexer, to pre-process typedef definitions because they introduce context-sensitivity into the language.

AST → Three-address code IR

I defined my own three-address code representation, and I implemented the transformation from the AST into the intermediate representation.

I overcame multiple areas of complexity:

- The logic for switch statements took some time to be correct for all the different possible structures.
- The semantics around assignment instructions meant I had to take care whether a variable was being written to directly or storing the address to store to.

Relooper algorithm

WebAssembly doesn't allow arbitrary control flow, so I implemented the Relooper algorithm which transforms the code into a block structure.

There are three types of block: Simple blocks are linear control flow, Loop blocks can branch back to the start of the block, and Multiple blocks represent conditional control flow along one of several paths.

I followed the algorithm as described in the Emscripten paper that introduced it. The main challenge here was to understand the algorithm thoroughly enough to actually implement it, because the paper doesn't describe all the fine details.

Target code generation

The last step in the main pipeline was generating WebAssembly binary code.

As well as defining the binary instructions to generate for each IR instruction, the main things I implemented here were managing the function call stack, including moving the frame and stack pointers, and also allocating an address to every variable.

Runtime environment

Once I was able to generate WebAssembly code, I write a NodeJS runtime environment to run the code.

The runtime provides the interface to the command line, and provides a skeleton implementation of some of the standard library, such as printf.

I used a suite of different C programs to test my compiler, and I wrote a test runner that compiles the programs with both GCC and my compiler, and compares the output to make sure they're the same.

This allowed me to work on optimisations while making sure I didn't break anything.

Optimisations

Once the main compiler pipeline was complete, I implemented some optimisations.

I implemented tail-call optimisation to make recursive functions more memory efficient. I did this by transforming the recursive call into iteration back to the start of the function, so no new stack frame has to be allocated.

I also implemented unreachable procedure elimination, by generating a call graph for the program, walking over it, and removing all functions we can't reach.

Stack allocation policy optimisation

As an extension to my project, I've implemented a more optimal stack allocation policy, which is able to reduce the memory needed for each stack frame.

Instead of allocating a new memory address for each temporary variable, it allocates multiple variables that don't clash with each other to the same address.

The example here is the stack size of the Fibonacci program, and we can see the memory usage is a lot lower with the optimisation enabled.

Wrap up

To wrap up, I've met my success criteria, and have implemented an extension optimisation. I plan to start writing up my evaluations this week.