Part 61: What's Next?

We've achieved the goal of writing a self-compiling compiler. Now that this goal has been reached, what else could we do with the codebase?

From the start, I'm going to say that there are already a number of working, production-ready C compilers: GCC, LLVM etc. We don't need another production-ready C compiler. But the point of writing this compiler was pedagogical: to explain the basics of how compilers work, and to put this knowledge into practice.

So, I see the future work on the compiler to continue to explain how compilers work and to put this into practice.

With this direction set, let's look at the possibilities.

Code Cleanup

I wrote the compiler fairly quickly, with only a little thought about the overarching design of the code. I think the design is reasonable, but the whole codebase needs a clean up. There's a fair bit of DRY code in places which could be refactored. Some of the code is ugly and could be improved. Also, some of the comments no longer reflect the code. This wouldn't change the compiler's functionality, but it would make it easier to understand.

Fix the Bugs

The compiler, as it stands, purports to implement a specific subset of the C language. But I'm sure there are plenty of bugs in this implementation. We could spend some time identifying these bugs and fixing them, while keeping the compiler's functionality constant.

Write Out the Final BNF Grammar

This suggestions goes along with the previous one. We should document the exact subset of the C language that the compiler supports, as a BNF grammar. I did write snippets of BNF grammar thoughout the journey, but near the end I stopped doing it. It would be good to write out the full, final, BNF grammar.

Support Variadic Functions

The compiler still doesn't check that the number of arguments to a function matches the number of function parameters. We need this because the compiler also doesn't support variadic functions like `printf()` and friends.

So, we need to add in the `...` token, somehow mark a function has having either "exactly N" or "N or more" parameters, and then write the code to use this information.

Add `short`s

It shouldn't be too hard to add a 16-bit signed `short` type. But Nils mentions, in his SubC book, that adding `unsigned` integer to a C compiler is tricky.

Rewrite the Register Allocation and Spilling

Right now, the mechanism for register allocation and register spilling is really awful, especially the spilling of registers before and after a function call. The assembly code is terribly inefficient. I'd like to see this rewritten using some of the theory on register allocation, e.g graph colouring. Even better, if this was written up like the past journey steps, it would help newcomers (like me) understand it better.

AST Optimisations

I did mention the idea of optimising the generated code by restructing the AST trees. An example of this is strength reduction. The SubC compiler does this, and it would be easy to add to our compiler, along with a writeup. There might be other AST tree optimisations that could be done.

Code Generation Optimisation

Another place to do output optimisation is in the code generator. A good example is peephole optimisation. To do this, however, the way the assembly code is generated would have to change. Instead of `fprintf()` ing the output, it should be stored in a data structure to make it easier for the peephole optimiser to traverse the assembly code. That's as far as I've thought, but it would be interesting to do.

Add Debugging Output

I started to do this in step 59. We should be able to output `gdb` directives into the assembly output to allow `gdb` to see the original C source lines, and step through a program line by line. Right now, the compiler is outputting this information but the `gdb` directives are not placed correctly in the assembly output. There's another step in here with a writeup on how to do this properly.

Complete the ARM Backend, plus Others

I did start the ARM back-end, and at the time I promised that I would keep it in sync with the x86-64 back-end. Well, I broke that promise as I got too interested in extending the compiler's functionality. Now that the compiler's functionality is relatively stable, I should go back and complete the ARM back-end. Even better would be a third back-end to prove that the compiler is fairly portable.

Extending the Recognised Grammar

I've left this suggestion to near the end as it doesn't continue the theme of explaining how compilers work. There is always scope to add more elements of the C language to the compiler. We don't need to do this to make the compiler self-compiling, but it would make the compiler more useful as a general-purpose compiler.

Work Out How to Call `ld` Directly

A long time ago, when I was playing around with BSD and Linux systems, I used to be able to link excecutables by hand with the `ld` command. I've been unable to work out how to do this on current Linux systems, and I'm relying on `cc` to do the linking for me. I'd love to learn how to link by hand with `ld` on Linux.

Port the Compiler to non-Linux Systems

Following on from the last point, it would be good to "port" the compiler to non-Linux systems like some of the BSD platforms.

Conclusion

These are all the possible things that I can of (at the moment) to continue the work on our compiler. I will get on to some of them, but at this point I'd be very happy to have other people help out with the project, and/or fork the compiler's code and do their own thing with it! Next step