1

Implementation

Word budget: ~4500 words

1.1 Repository Overview

I developed my project in a GitHub repository[1], ensuring to regularly push to the cloud for backup purposes. This repository is a monorepo containing both my research and documentation along with my source code.

```
├── headers/ ........................ Header files for the standard library functions
│   │                                 I implemented
│   ├── stdio.h
│   └── ...
├── runtime/ ........................ NodeJS runtime environment
│   ├── stdlib/ ..................... Implementations of standard library functions
│   │                                 in JS
│   ├── run.mjs
│   └── ...
├── src/ ............................ The source code for the compiler, explained be-
│   │                                 low
│   └── ...
├── tests/ .......................... Test specification files
│   └── ...
└── tools/
    ├── profiler.py ................. Code to plot stack usage profiles
    └── testsuite.py ................ Test runner
src/
├── back_end/
├── data_structures/
├── front_end/
├── middle_end/
├── program_config/
├── relooper/
├── fmt_indented.rs
├── id.rs
├── lib.rs
└── main.rs
```

[1] https://github.com/martin-walls/cam-part-ii-c-webassembly-compiler

```
└─ preprocessor.rs
```

Finish this. Will have to see if it'll be better to have comments on the right of dirs, or to highlight the main structure below

1.2 System Architecture

Compiler Pipeline overview – include the diagram here

1.3 Front End: Lexer and Parser

Describe what was actually produced.
Describe any design strategies that looked ahead to the testing phase, to demonstrate professional approach

- wrote parser grammar
Talk about avoiding ambiguities - eg. dangling else - by using Open/Closed statement in grammar
Talk about my `interpret_string` implementation, to handle string escaping. Implemented using an iterator.
- wrote custom lexer - cos typedefs make C context-sensitive, so handle them as we see them so they don't get mixed with identifiers
- created AST representation
Talk about structure of my AST
Talk about how I parsed type specifiers into a standard type representation. Used a bitfield to parse arithmetic types, cos they can be declared in any order.

Describe high-level structure of codebase.
Say that I wrote it from scratch.
-> mention LALRPOP parser generator used for .lalrpop files

1.4 Middle End: Intermediate Representation

- Defined my own three-address code representation
- for every ast node, defined transformation to 3AC instructions
- created IR data structure to hold instructions + all necessary metadata
- Talk about auto-incrementing IDs - abstraction of the Id trait and generic IdGenerator struct
- handled type information - created data structure to represent possible types
- making sure instructions are type-safe, type converting where necessary - talk about unary/binary conversions, cite the C reference book
- Compile-time evaluation of expressions, eg. for array sizes
- Talk about the Context design pattern I used throughout – maybe research this and see if it's been done before?

1.5 The Relooper Algorithm

cite Emscripten [1]

1.6 Back End: Target Code Generation

1.7 Optimisations

1.7.1 Unreachable Procedure Elimination

1.7.2 Tail-Call Optimisation

Defn of tail-call optimisation
Why do the optimisation

1.8 Summary