1
Implementation

Word budget: ~4500–5400 words

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

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

1.1 Repository Overview

Don't put github repo url – it's got my name in it

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 below
...	
tests/	Test specification files
...	
tools/	

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

```
    ├── 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
└── 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

Figure 1.1 describes the high-level structure of the project. The front end, middle end, and back end are denoted by colour.

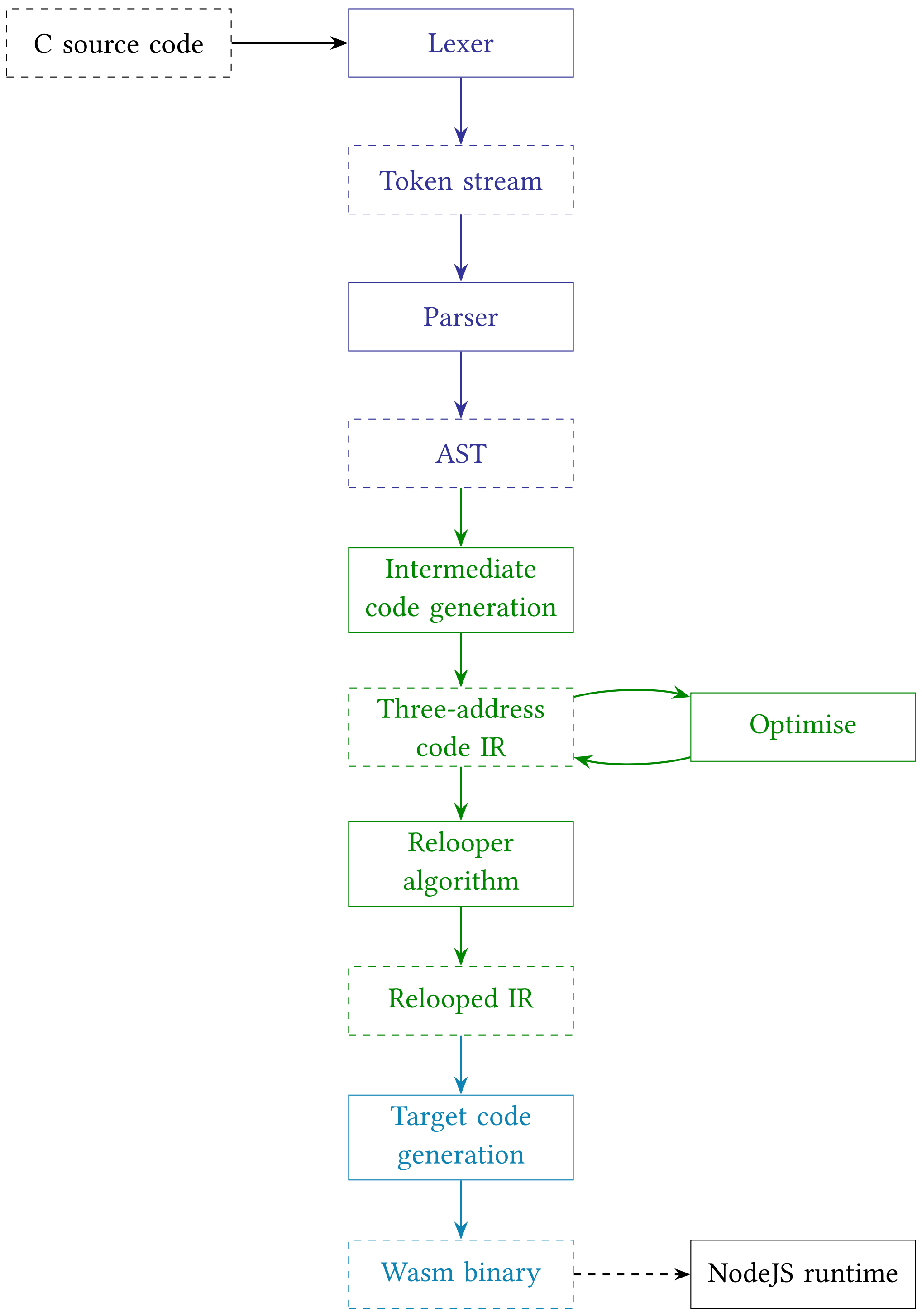

Figure 1.1: Project structure, highlighting the front end, middle end, and back end.

Each solid box represents a module of the project, transforming the input data representation into the output representation. The data representations are shown as dashed boxes.

I created my own Abstract Syntax Tree (AST) representation and Intermediate Representation (IR), which are used as the main data representations in the compiler.

1.3 Front End

Give an overview of the front end

1.3.1 Lexer

The grammar of the C language is mostly context-free, however the presence of typedef definitions make it context-sensitive [1, Section 5.10.3, p. 151]. For example, the statement in Listing 1.1 can be interpreted in two ways:

- As a variable declaration, if `foo` has previously been defined as a type name[2]; or
- As a function call, if `foo` is the name of a function.

```
foo (bar);
```

Listing 1.1: An example of typedef name ambiguity in C.

This ambiguity is impossible to resolve with the language grammar. The solution is to preprocess the typedef names during the lexing stage, and emit distinct type name and identifier tokens to the parser. Therefore, I implemented a custom lexer that is aware of the type names that have already been defined the current point in the program.

The lexer is implemented as a finite state machine. Figures 1.2 and 1.3 highlight portions of the machine; the remaining state transition diagrams can be found in ??. The diagrams show the input character, as a regular expression, along each transition arrow. (Note: in a slight abuse of regular expression notation, the dot character '`.`' represents a literal full stop character, and the backslash character '\' represents a literal backslash.) It is assumed that when no state transition is shown for a particular input, the end of the current token has been reached. Transition arrows without a prior state are the initial transitions for the first input character. Node labels represent the token that will be emitted when we finish in that state.

The finite state machine consumes the source code one character at a time, until the end of the token is reached (i.e. there is no transition for the next input character). Then, the token corresponding to the current state is emitted to the parser. Some states have no corresponding token to emit, because they occur part-way through parsing a token; if the machine finishes in one of these states, this raises a lex error. (In other words, every state labelled with a token is an accepting state of the machine.) For tokens such as number literals and identifiers, the lexer appends the input character to a string buffer on each transition, and when the token is complete, the string is stored inside the emitted token. This gives the parser access to the necessary data about the token, for example the name of the literal.

If, when starting to lex a new token, there is no initial transition corresponding to the input character, then there is no valid token for this input. This raises a lex error, and the compiler will exit.

[2]The brackets will be ignored.

Figure 1.2 shows the finite state machine for lexing number literals. This handles all the different forms of number that C supports: decimal, binary, octal, hexadecimal, and floating point. (Note: the states leading to the ellipsis token are shown for completeness, even though the token is not a number literal, since they share the starting dot state.)

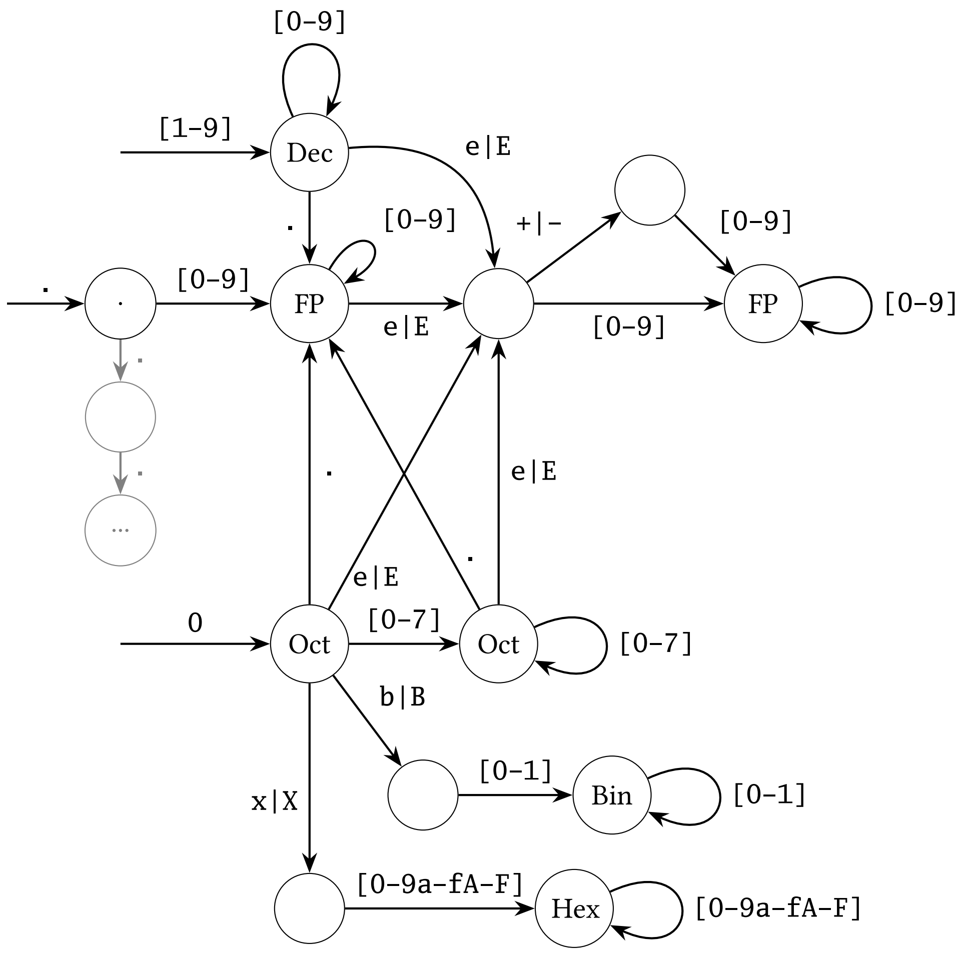

Figure 1.2: Finite state machine for lexing number literals.

Figure 1.3 shows the finite state machine for lexing identifiers and typedef names. This is where we handle the ambiguity introduced into the language by typedef names. Every time we consume another character of an identifier, we check whether the current name (which we have stored in the string buffer) matches either a keyword of the language of a typedef name we have encountered this far. (Keywords are given a higher priority of matching.) If a match is found, we move to the corresponding state, represented by the ϵ transitions (since no input is consumed along these transitions). When we reach the end of the token, the three states will emit the corresponding token, either an identifier, keyword, or typedef name token respectively.

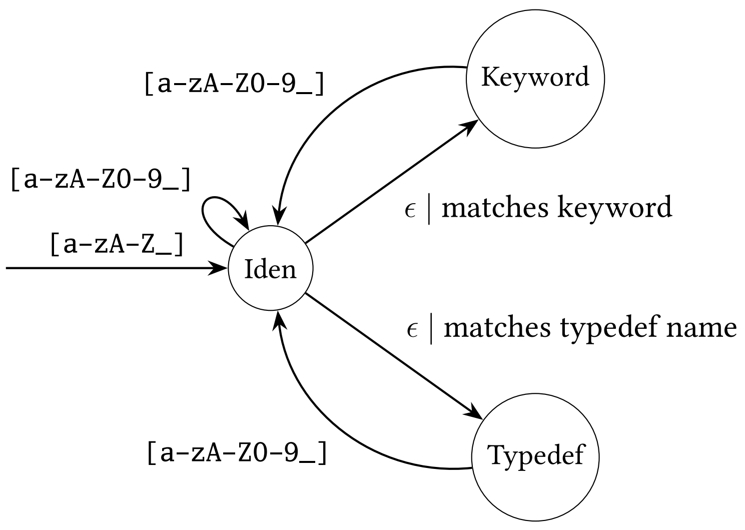

Figure 1.3: Finite state machine for lexing identifiers.

1.3.2 Parser

- 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.
- 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.

1.4 Middle End

Give an overview of the middle end

1.4.1 Intermediate Code Generation

- 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?

1.4.2 The Relooper Algorithm

cite Emscripten [2]

1.5 Back End: Target Code Generation

1.6 Runtime Environment

- Instantiating wasm module
- stdlib functions skeleton implementation
- arg passing + memory initialisation

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