Martin Walls

C to WebAssembly Compiler

Computer Science Tripos: Part II

Churchill College

30th March, 2023

Declaration of Originality

I, Martin Walls of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed: 

Date: 30th March, 2023

Proforma

Candidate Number:	**TODO**
Project Title:	C to WebAssembly Compiler
Examination:	Computer Science Tripos: Part II
Year:	2023
Dissertation Word Count:	**TODO**[1]
Code Line Count:	**TODO**[2]
Project Originator:	Timothy M. Jones
Supervisor:	Timothy M. Jones

Original Aims of the Project

▷ At most 100 words describing the original aims of the project.

This project aimed to implement a complete compiler pipeline, compiling a subset of the C language into WebAssembly. This consists of a lexer and parser, using a custom abstract syntax tree; a custom three-address code intermediate representation; converting unstructured to structured control flow; and generating WebAssembly binary code. Additionally, I aimed to extend the compiler with optimisations to improve the performance of the compiled code.

Work Completed

▷ At most 100 words summarising the work completed.

The project was entirely successful in completing all the original aims, and in completing an extension. The compiler pipeline is able to transform C source code into correct WebAssembly binaries that can be executed through a JavaScript runtime environment. Each of the pipeline stages maintains correctness as it transforms the code. As well as the planned extensions, I was able to implement an additional extension optimisation, which was successful in significantly reducing the memory usage of programs.

[1]Excluding figures and listings.
[2]Excluding comments and blank lines. Code line count computed with cloc (`https://github.com/AlDanial/cloc`).

Special Difficulties

None.

Contents

1 Introduction

Word budget: ~500–600 words

Explain the main motivation for the project
Show how the work fits into the broad area of surrounding computer science

Talk about why I chose C as a source language

1.1 Background and Motivation

Increasingly in modern society, more and more applications are shifting to cloud computing as one of the primary ways of interacting with computers. We are experiencing a transition away from traditional native applications and towards performing the same tasks in an online environment. However, the standard approach of building web apps with JavaScript fails to deliver the performance necessary for many intensive applications.

WebAssembly (abbreviated Wasm) aims to solve this problem by bringing near-native performance to the browser space. It is a virtual instruction set architecture, which executes on a stack-based virtual machine. Per the Introduction section of the WebAssembly specification [1], it is designed to have "fast, safe, and portable semantics" and an "efficient and portable representation". The next two paragraphs expand on what this entails.

The semantics are designed to be able to be executed efficiently across different hardware, be memory safe (with respect to the surrounding execution environment), and to be portable across source languages, target architectures, and platforms.

The representation is designed with the primary target of the web in mind. It is designed to be compact and modular, allowing it to be efficiently transmitted over the Internet without slowing down page loads. This also includes being streamable and parallelisable, which means it can be decoded while still being received.

1.2 Survey of Related Work

brief survey of previous related work
- original emscripten (LLVM to JS)
- various other compilers to wasm, including LLVM

2

Preparation

Word budget: ~2500–3000 words

Describe the work undertaken before code was written.
”Requirements Analysis” section
-> refer to appropriate software engineering techniques used in the diss
Cite new programming language learnt
Declare starting point
Explain background material required beyond IB
Researching LALRPOP - show good professional use of tools
Talk about revision control strategy, licensing of any libraries I used

Add some intro text here between chapter heading and section title.

2.1 WebAssembly

Before starting to write the compiler, I extensively researched the WebAssembly specification [1], to gain a deep understanding of the binary code the project aims to generate. The following sections provide a high-level overview of the instruction set architecture.

2.1.1 Primitive values

The primitive value types supported by WebAssembly are outlined in Table 2.1, and described in more detail below.

Integers can be interpreted as either signed or unsigned, depending on the operations applied to them. They will also be used to store data such as booleans and memory addresses[1] Integer literals are encoded in the program using the LEB128 variable-length encoding scheme [2]. Listing 2.1 shows how to encode an unsigned integer. The algorithm is almost the same for signed integers (encoded in two’s complement); the only difference is that we sign-extend to a multiple of 7 bits, rather than zero-extend. When decoding, the decoder needs to know if the number is signed or unsigned, to know whether to decode it as a two’s complement number or not. This is why WebAssembly has signed and unsigned variants of some instructions.

[1] I.e. addresses within WebAssembly’s sandboxed linear memory space, rather than function addresses etc.

Type	Constructor	Bit width
Integer	`i32`	32-bit
	`i64`	64-bit
Float	`f32`	32-bit
	`f64`	64-bit
Vector	`v128`	128-bit
Reference	`funcref`	Opaque
	`externref`	

Table 2.1: WebAssembly primitive types.

```
function LEB128EncodeUnsigned(n)
    Zero-extend n to a multiple of 7 bits
    Split n into groups of 7 bits
    Add a 0 bit to the front of the most significant group
    Add a 1 bit to the front of every other group
    return bytes in little-endian order
end function
```

Listing 2.1: Pseudocode for the LEB128 encoding scheme (for unsigned integers).

Floating-point literals are encoded using the IEEE 754-2019 standard [3]. This is the same standard used by many programming languages, including Rust, so the byte representation will not need to be converted between the compiler and the WebAssembly binary.

WebAssembly also provides vector types and reference types. Vectors can store either integers or floats: in either case, the vector is split into a number of evenly-sized numbers. Vector instructions exist to operate on these values. `funcref` values are pointers to WebAssembly functions, and `externref` values are pointers to other types of object that can be passed into WebAssembly. These would be used for indirect function calls, for example. This project does not have a need for using either of these types; the C language doesn't have vector types, and although a compiler could create them, mine will not. Additionally I do not support function references in the scope of this project. I will only use the four main integer and float types.

2.1.2 Instructions

WebAssembly is a stack-based architecture; all instructions operate on the stack. For binary instructions that take their operands from the stack, the first operand is the one that was pushed to the stack first, and the second operand is the one pushed to the stack most recently (see Listing 2.2).

```
i32.const 10 ;; first operand
i32.const 2  ;; second operand
i32.sub
```

Listing 2.2: WebAssembly instructions to calculate `10 - 2`.

All arithmetic instructions specify the type of value that they expect. In Listing 2.2, we put two `i32` values on the stack, and use the `i32` variant of the `sub` instruction. The module would fail to instantiate if the types did not match. Some arithmetic instructions have signed and unsigned variants, such as the less-than instructions `i32.lt_u` and `i32.lt_s`. This is the case for all instructions where the signedness of a number would make a difference to the result.

WebAssembly only supports structured control flow, in contrast to the unstructured control flow found in most instruction sets that feature arbitrary jump instructions. There are three types of block: `block`, `loop`, and `if`. The only difference between `block` and `loop` is the semantics of branch instructions. When referring to a `block`, `br` will jump to the end of it, and when referring to a `loop`, `br` will jump back to the start. This is analogous to `break` and `continue` in C, respectively. It is worth noting that `loop` doesn't loop back to the start implicitly; an explicit `br` instruction is required. `if` blocks, which may optionally have an `else` block, conditionally execute depending on the value on top of the stack. A `br` instruction inside an `if` block behaves in the same way as inside a `block`; it will jump to the end of it.

2.1.3 Modules

A module is a unit of compilation and loading for a WebAssembly program. There is no distinction between a 'library' and a 'program' such as there is in other languages; there are only modules that export functions to the instantiator (i.e. the runtime environment that instantiates the WebAssembly module). Modules are split up into different sections. Each section starts with a section ID and the size of the section in bytes, followed by the body of the section. Each of the sections is described in turn below.

The *type section* defines any function types used in the module, including imported functions. Each type has a type index, allowing the same type to be used by multiple functions.

The *import section* defines everything that is imported to the module from the runtime environment. This includes imported functions as well as memories, tables, and globals.

The *function section* is a map from function indexes to type indexes. This comes before the actual body code of the functions, because this allows the module to be decoded in a single pass. The type signature of all functions will be known before any of the code is read, so all function calls will be able to be properly typed without needing multiple passes.

A table is an array of function pointers, which can be called indirectly with `call_indirect`. It is necessary for WebAssembly to have a separate data structure for this, because functions live outside of the memory visible to the program; tables keep the function addresses opaque to the program, keeping the execution sandboxed. The *table section* stores the size limits of each table; any elements to initialise the tables with are stored in the element section.

The *memory section* is similar to the table section; is specifies the size limits of each linear memory of the program[1], in units of the WebAssembly page size (defined as 64 KiB).

The *global section* defines any global variables used in the program, including an expression to initialise them. Global variables can be either mutable or immutable.

[1]In the current WebAssembly version, only one memory is supported, and is implicitly referenced by memory instructions.

The *export section* defines everything exported from the module to the runtime environment. Normally this is mainly function exports, however it can also include tables, memories, and global variables.

The *start section* optionally specifies a function that should automatically be run when the module is initialised. This could be used to initialise a global variable to a non-constant expression.

The *element section* is used to statically initialise the contents of tables with function addresses. Segments can be active or passive. An active segment will be automatically initialise the table when the module is instantiated, whereas a passive segment needs to be explicitly loaded into a table with a `table.init` instruction.

The *data count section* is an optional additional section used to allow validators to use only a single pass. It specifies how many data segments are in the data section. If the data count section were not present, a validator would not be able to check the validity of instructions that reference data indexes until after reading the data section; but because this comes after the code section, it would require multiple passes over the module. This section has no effect on the actual execution.

The *code section* contains the instructions for each function body. All code in a WebAssembly program is contained in a function. Each function body begins by declaring any local variables, followed by the code for that function.

The *data section* is similar to the element section; it is used to statically initialise the contents of memory. This can be used, for example, to load string literals into memory.

2.2 The Relooper Algorithm

C allows arbitrary control flow, because it has **`goto`** statements. However, WebAssembly only has structured control flow, using **`block`**, **`loop`**, and **`if`** constructs as described above. Therefore, once the intermediate code has been generated, it needs to be transformed to only have structured control flow.

There are several algorithms that achieve this. The most naive solution would be to use a label variable and one big **`switch`** statement containing all the basic blocks of the program; the label variable is set at the end of each block, and determines which block to switch to next. However, this is very inefficient.

The first algorithm that solved this problem in the context of compiling to WebAssembly (and JavaScript, before that) was the Relooper algorithm, introduced by Emscripten in their paper on compiling LLVM to JavaScript [4]. In today's WebAssembly compilers, there are three general methods used to convert to structured control flow [5, 6]: Emscripten/Binaryen's Relooper algorithm, LLVM's CFGStackify, and Cheerp's Stackifier. All of these implementations, including the modern implementation of Relooper, are more optimised than the original Relooper algorithm, however therefore also more complex. The original Relooper algorithm is a greedy algorithm, and is well described in the paper, and is the one I decided to implement.

2.2.1 Input and Output

The Relooper algorithm takes a so-called 'soup of blocks' as input. Each block is a basic block of the flowgraph, that begins at an instruction label and ends with a branch instruction (Figure 2.1). There can be no labels or branch instructions other than at the start or end of the block. The branch instruction may either be a conditional branch (i.e. `if (...) goto x else goto y`), or an unconditional branch.

To avoid overloading the term *block*, these input blocks are referred to as *labels*.

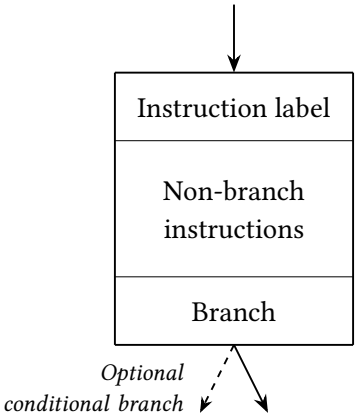


Figure 2.1: Structure of labels (Relooper algorithm input blocks).

The algorithm generates a set of structured blocks, recursively nested to represent the control flow (Figure 2.2). *Simple* blocks represent linear control flow; they contain an *internal* label, which contains the actual program instructions. *Loop* blocks contain an *inner* block, which contains all the labels that can possibly loop back to the start of the loop, along some execution path. *Multiple* blocks represent conditional execution, where execution can flow along one of several paths. They contain *handled* blocks to which execution can pass when we enter the block. All three blocks have a *next* block, where execution will continue.

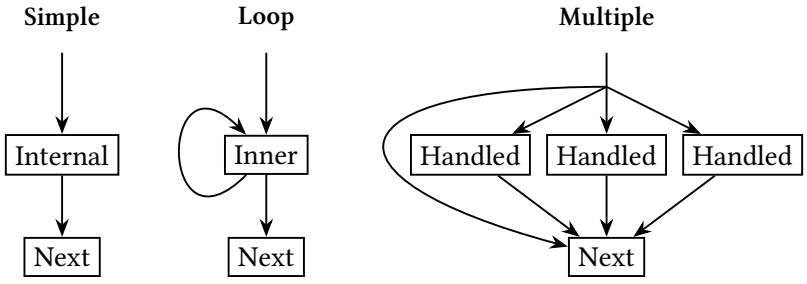


Figure 2.2: Structured blocks, generated by the Relooper algorithm.

2.2.2 Algorithm

Before describing the algorithm, I define the terms used. *Entries* are the subset of labels that can be immediately reached when a block is entered. Each label has a set of *possible branch targets*, which are the labels it directly branches to. It also has a set of labels it can *reach*, known as the *reachability* of the label. This is the transitive closure of the possible branch targets; i.e. all labels that can be reached along some execution path. Labels can always reach themselves.

Given a set of labels, and set of entries, the algorithm to create a block is as follows:

1. Calculate the reachability of each label.
2. If there is only one entry, and execution cannot return to it, create a simple block with the entry as the internal label.
   - Construct the next block from the remaining labels; the entries for the next block are the possible branch targets of the internal label.
3. If execution can return to every entry along some path, create a loop block.
   - Construct the inner block from all labels that can reach at least one of the entries.
   - Construct the next block from the remaining labels; the entries for the next block are all the labels in the next block that are possible branch targets of labels in the inner block.
4. If there is more than one entry, attempt to create a multiple block. (This may not be possible.)
   - For each entry, find any labels that it reaches that no other entry reaches. If any entry has such labels, it is possible to create a multiple block. If not, go to Step 5.
   - Create a handled block for each entry that uniquely reaches labels, containing all those labels.
   - Construct the next block from the remaining labels. The entries are the remaining entries we didn't create handled blocks from, plus any other possible branch targets out of the handled blocks.
5. If Step 4 fails, create a loop block in the same way as Step 3.

The Emscripten paper outlines a proof that this greedy approach will always succeed [4, p. 10]. The core idea is that we can show that (1) whenever the algorithm terminates, the block it outputs is correct with respect to the original program semantics; (2) the problem is strictly simplified every time a block is created, therefore the algorithm must terminate; and (3) the algorithm is always able to create a block from the input labels. Point (3) lies in the observation that if we reach Step 5, we must be able to create a loop block. This holds because if we could not create a loop block, we would not be able to return to any of the entries; however, in that case it would be possible to create a multiple block in Step 4, because the entries would uniquely reach themselves.

The algorithm replaces the branch instructions in the labels as it processes them. When creating a loop block, branch instructions to the start of the loop are replaced with a **`continue`**, and any branches to outside the loop are replaced with a **`break`** (all the branch targets outside the loop will be in the next block). When creating a handled block, branch instructions into the next block are replaced with an **`endHandled`** instruction. Each of these instructions is annotated with the ID of the block they act on, because blocks may be nested. Note that functionally, an **`endHandled`** instruction is equivalent to a **`break`**, but I chose to keep the two distinct because they store IDs of different block types.

To direct control flow when execution enters a multiple block, Relooper makes use of a label variable. Whenever a branch instruction is replaced, an instruction is inserted to set the label variable to the label ID of the branch target. Handled blocks are executed conditional on the value of the label variable. This will generate some overhead of unnecessary instructions, since most of the time the

label variable is set, it will not be checked. However, later stages of the compiler pipeline are able to optimise this away.

2.3 Rust

I learned the Rust programming language [7] for this project. I chose Rust because it is performant and memory efficient. It has a rich type system with pattern matching, which is well suited to writing a compiler. It uses the concept of a borrow checker rather than a garbage collector or other memory management system, which eliminates runtime overhead while guaranteeing memory safety at compile time.

I made use of Rust's excellent online documentation [8, 9], using it to become familiar with the language. The borrow checker was the main new feature of the language to learn. At its core are the concepts of *ownership* and *borrowing*. The main rule is that every value has exactly one owner at any time. When the owner goes out of scope, the value is automatically deallocated; this takes the place of the garbage collector found in other languages. When using a value, it can either be *moved* or *borrowed*. A move will make the new variable the owner of the value, and invalidate the old variable. Borrowing, on the other hand, allows a value to be used without taking ownership; it creates a reference that points to the owned value. There can either be many immutable borrows of a value, or a single mutable borrow. When the borrower is done with the value, it is given back to the owner.

2.4 Project Strategy

TODO put short para here

2.4.1 Requirements Analysis

The requirements for this project were clear from the project description. A C to WebAssembly compiler takes C source code as input, and generates WebAssembly binary code that can be instantiated and executed from JavaScript.

The compiler consists of a pipeline of stages: a front end, middle end, and back end. The front end takes C source code, and outputs an abstract syntax tree (AST). The middle end takes the AST and transforms it to an intermediate representation (IR), in this case three-address code. The back end takes the IR and generates WebAssembly instructions, writing them to a binary file. Each stage of the compiler pipeline is distinct, with defined data models connecting them.

The stages described are the core requirements of the project. Optimisations to the compiler were set as extensions to the project, once the core objectives had been completed. These fall within the middle and back ends, transforming the IR and the target code.

2.4.2 Software Engineering Methodology

Waterfall model: sequential process, each phase completed before starting the next phase

2.4.3 Testing

2.5 Starting Point

TODO put short para here

2.5.1 Knowledge and experience

The *Compiler Construction* course of the Tripos was my starting point for my knowledge of compilers. The Part II course *Optimising Compilers* was useful in extending my project with optimisations, though the optimisations I implemented were not all included in the course.

I had familiarity with the C language from the *Programming in C and C++* course, as well as from previous personal projects. Additionally, I had experience writing JavaScript and Python from personal projects. I gained all my knowledge of WebAssembly and Rust through independent research.

2.5.2 Tools Used

The code for the body of the compiler was written in Rust (see Section 2.3). I wrote the runtime environment in JavaScript [10], using Node.js [11], which is necessary to interface with the WebAssembly binary. Additional development scripts, for automated testing and profiling, were written in Python [12], because of the ease of use, and my familiarity with libraries such as Matplotlib [13]. I used the LALRPOP parser generator library [14] to generate parsing code from the abstract grammar of C.

I used Git [15] for version control, regularly pushing the repository to GitHub [16] as an off-site backup. I used the CLion IDE as my development environment, using the Intellij Rust plugin [17]. CLion has excellent support for C, and the plugin enables native Rust support, including code analysis and debugging.

3 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

Add some intro text here between chapter heading and section title.

3.1 Repository Overview

I developed my project in a GitHub repository, 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.

The high-level structure of the codebase is shown below. All the code for the compiler is in the **`src/`** directory. The other directories contain the runtime environment code, skeleton standard library implementation, and tests and other tools.

`src/`	Compiler source code.
`program_config/`	Compiler constants and runtime options data structures.
`front_end/`	Lexer, parser grammar, AST data structure.
`middle_end/`	IR data structures, definition of intermediate instructions. Converting AST to IR.
`middle_end_optimiser/`	Tail-call optimisation and unreachable procedure elimination.
`relooper/`	Relooper algorithm.
`back_end/`	Target code generation stage.
`wasm_module/`	Data structures to represent a WebAssembly module.
`dataflow_analysis/`	Flowgraph generation, dead code analysis, live variable analysis, clash graph.
`stack_allocation/`	Different stack allocation policies.

data_structures/	Interval tree implementation that I ended up not using.
preprocessor.rs	C preprocessor.
id.rs	Trait for generating IDs used across the compiler.
lib.rs	Contains the main run function.
runtime/	Node.js runtime environment.
headers/	Header files for the parts of the standard library I implemented.
tools/	
profiler.py	Plot stack usage profiles.
testsuite.py	Test runner script.
tests/	Automated test specifications.

3.2 System Architecture

Figure 3.1 describes the high-level structure of the project. The front end, middle end, and back end are denoted by colour.

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 AST representation and IR, which are used as the main data representations in the compiler.

3.3 Front End

The front end of the compiler consists of the lexer and the parser; it takes C source code as input and outputs an abstract syntax tree. I wrote a custom lexer, because this is necessary to support **typedef** definitions in C. I used the LALRPOP parser generator [14] to convert the tokens emitted by the lexer into an AST.

3.3.1 Preprocessor

I used the GNU C preprocessor (cpp) [18] to handle any preprocessor directives in the source code, for example macro definitions. However, since I do not support linking, I removed any `#include` directives before running the preprocessor and handled them myself.

For each `#include <name.h>` directive that is removed, if it is one of the standard library headers that I support, the appropriate library code is copied into the source code from `headers/<name>.h`. One exception is when the name of the header file matches the name of the source program, in which case the contents of the program's header file are inserted into the source code, rather than finding a matching library.

After processing `#include` directives, the compiler spawns `cpp` as a child process, writes the source code to its stdin, and reads the processed source code back from its stdout.

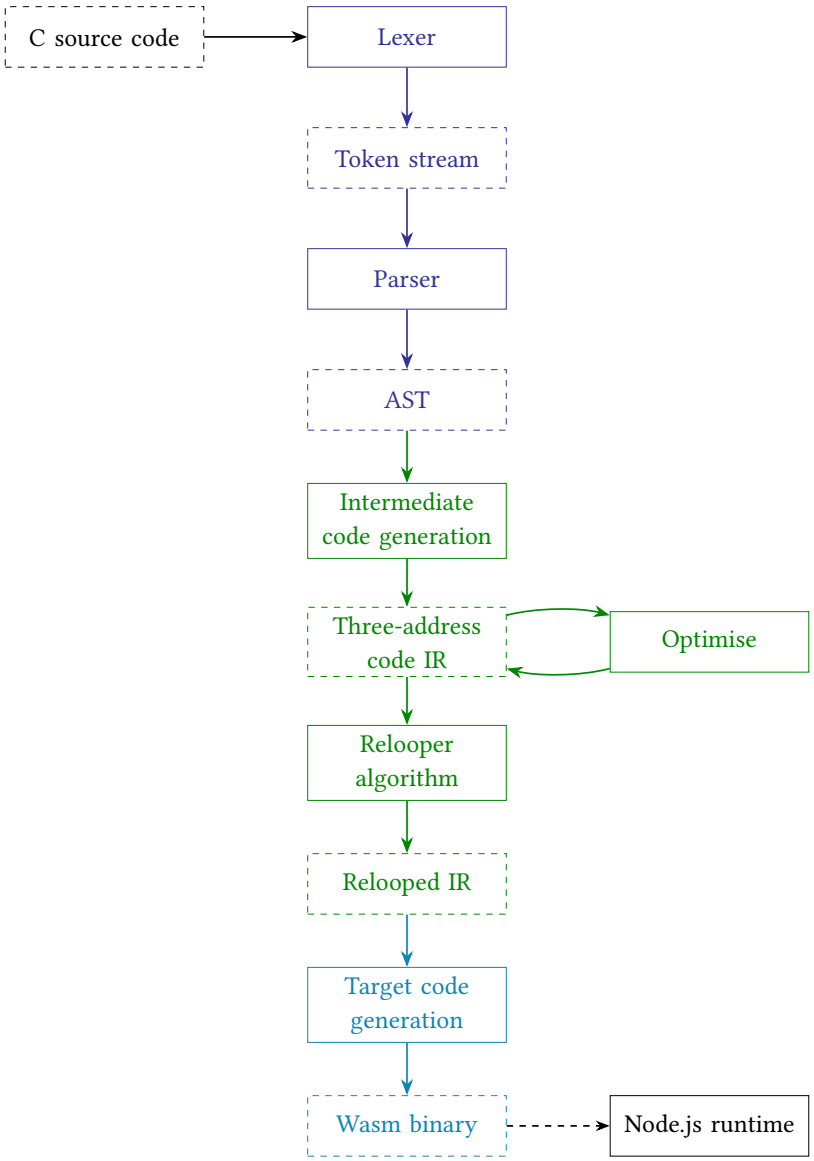


Figure 3.1: Project structure, highlighting the front end, middle end, and back end.

3.3.2 Lexer

The grammar of the C language is mostly context-free, however the presence of **`typedef`** definitions makes it context-sensitive [19, Section 5.10.3]. For example, the statement `foo (bar);` can be interpreted in two ways:

- As a variable declaration, if `foo` has previously been defined as a type name[1]; or
- As a function call, if `foo` is the name of a function.

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 at the current point in the program.

[1]The brackets will be ignored.

The lexer is implemented as a finite state machine. Figures 3.2 and 3.3 highlight portions of the machine; the remaining state transition diagrams can be found in Appendix A. 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.

Figure 3.2 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 3.3 shows the finite state machine for lexing identifiers and **`typedef`** names. This is where we handle the ambiguity introduced into the language. 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 or a **`typedef`** name we have encountered so 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. The lexer stores the **`typedef`** names that have been declared so far so that it can emit the correct type of token for future names.

3.3.3 Parser

I used the LALRPOP parser generator [14] to generate parsing code from the input grammar I wrote. It generates an AST that I defined the structure of. Microsoft's C Language Syntax Summary [20] and C: A Reference Manual [19] were very useful references to ensure I captured the subtleties of C's syntax when writing my grammar. My grammar is able to parse all of the core features of the C language, omitting some of the recent language additions. I chose to make my parser handle a larger subset of C than the compiler as a whole supports; the middle end rejects or ignores nodes of the AST that it doesn't handle. For example, the parser handles storage class specifiers (e.g. **`static`**) and type qualifiers (e.g. **`const`**), and the middle end simply ignores them.

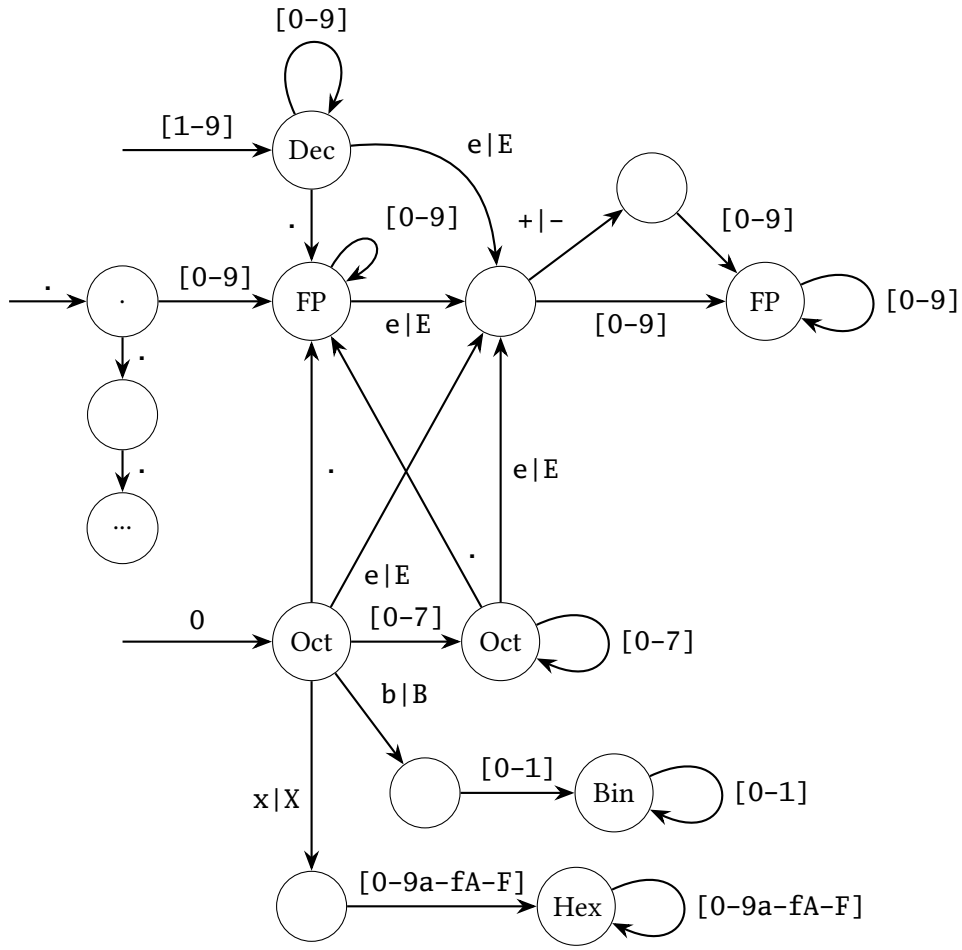


Figure 3.2: Finite state machine for lexing number literals.

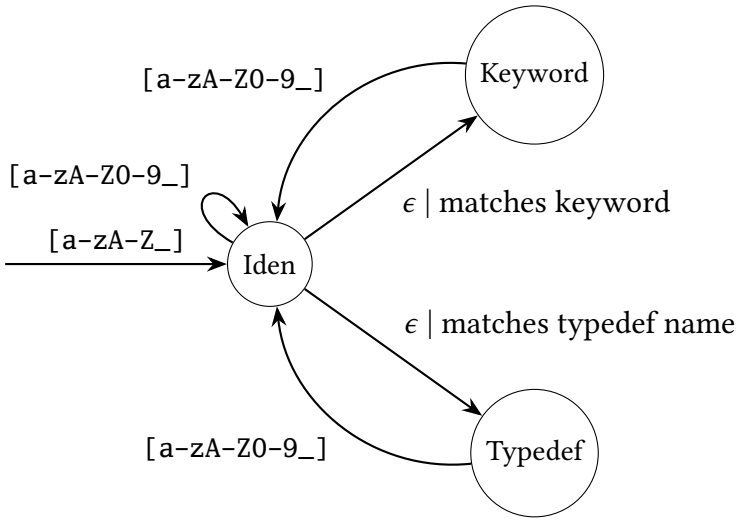


Figure 3.3: Finite state machine for lexing identifiers.

3.3.3.1 Dangling Else Ambiguity

A naive grammar for C (Listing 3.1) contains an ambiguity around **`if`**/**`else`** statements [19, sec. 8.5.2]. C permits the bodies of conditional statements to be written without curly brackets if the body is a single statement. If we have nested **`if`**/**`else`** statements that do not use curly brackets, it can be ambiguous which **`if`** an **`else`** belongs to. This is known as the dangling else problem [21].

```
if-stmt      ::= "if" "(" expr ")" stmt
if-else-stmt ::= "if" "(" expr ")" stmt "else" stmt
```

Listing 3.1: Ambigious **if**/**else** grammar.

An example of the dangling else problem is shown in Listing 3.2. According to the grammar in Listing 3.1, there are two possible parses of this program. Either the **else** belongs to the inner or the outer **if** (Listing 3.3).

```
if (x)
  if (y)
    stmt1;
else
  stmt2;
```

Listing 3.2: Example of the dangling else problem.

```
if (x) {
  if (y) {
    stmt1;
  } else {
    stmt2;
  }
}
```

(a) **else** belongs to inner **if**.

```
if (x) {
  if (y) {
    stmt1;
  }
} else {
  stmt2;
}
```

(b) **else** belongs to outer **if**.

Listing 3.3: Possible parsings of Listing 3.2.

C resolves the ambiguity by always associating the **else** with the closest possible **if**. We can encode this into the grammar with the concept of 'open' and 'closed' statements [22]. Listing 3.4 shows how I introduce this into my grammar for **if**/**else** statements. All other forms of statement must also be converted to the new structure. Any basic statements, i.e. statements that have no sub-statements, are added to **closed-stmt**. All other statements that have sub-statements, such as **while**, **for**, and **switch** statements, must be duplicated to both **open-stmt** and **closed-stmt**.

```
stmt        ::= open-stmt | closed-stmt

open-stmt   ::= "if" "(" expr ")" stmt
              | "if" "(" expr ")" closed-stmt "else" open-stmt
              | ...

closed-stmt ::= "if" "(" expr ")" closed-stmt "else" closed-stmt
              | ...
```

Listing 3.4: Using open and closed statements to solve the dangling else problem

A closed statement always has the same number of **if** and **else** keywords (excluding anything between a pair of brackets, because bracket matching is unambiguous). Thus, in the second alternative of an **open-stmt**, the **else** terminal can be found by counting the number of **if**s and **else**s we encounter since the start of the **closed-stmt**; the **else** belonging to the **open-stmt** is the first **else** once we have more **else**s than **if**s.

If we allowed open statements inside the **if** clause of an **open-stmt**, then **open-stmt** and **closed-stmt** would no longer be disjoint, and the grammar would be ambiguous. This is because we wouldn't be able to use the above method for finding the **else** that belongs to the outer **open-stmt**.

3.3.3.2 LALRPOP Parser Generator

I chose the LALRPOP parser generator because it builds up the AST as it parses the grammar. This is in contrast to some of the other available libraries, which separate the grammar code and the code that generates the AST. LALRPOP provides an intuitive and powerful approach. Each grammar rule contains both the grammar specification and code to generate the corresponding AST node.

Listing 3.5 is an example of the LALRPOP syntax for addition expressions. The left-hand side of the => describes the grammar rule, and the right-hand side is the code to generate an `Expression` node. Terminals are represented in double quotes; these are defined to map to the tokens emitted by the lexer. Non-terminals are represented inside angle brackets, with their type and a variable name to use in the AST generation code.

```
additive-expression ::= additive-expression "+" multiplicative-expression
```

(a) The grammar rule for addition expressions.

```
AdditiveExpression: ast::Expression = {
    <e1:AdditiveExpression> "+" <e2:MultiplicativeExpression>
        => ast::Expression::BinaryOp(
            ast::BinaryOperator::Add,
            Box::new(e1),
            Box::new(e2)
        ),
    ...
};
```

(b) The LALRPOP syntax for the addition grammar rule.

Listing 3.5: In LALRPOP, the AST generation and grammar code are combined.

LALRPOP also allows macros to be defined, which allow the grammar to be written in a more intuitive way. For example, I defined a macro to represent a comma-separated list of non-terminals (Listing 3.6). The macro has a generic type `T`, and automatically collects the list items into a `Vec<T>`, which can be used by the rules that use the macro.

```
CommaSepList<T>: Vec<T> = {
    <mut v:(<T> ",")*> <e:T> => {
        v.push(e);
        v
    }
};
```

Listing 3.6: LALRPOP macro to parse a comma-separated list of non-terminals.

3.3.3.3 String Escape Sequences

The parser had to handle escape sequences in strings. I implemented this by first creating an iterator over the characters of a string, which replaces escape sequences by the character they represent as it emits each character. When the current character is a backslash, instead of emitting it straight away, the iterator consumes the next character and emits the character corresponding to the escape sequence. I wrapped this in an `interpret_string` function that internally creates an instance of the iterator and collects the emitted characters back to a string.

3.3.3.4 Parsing Type Specifiers

Another feature of the C language is that type specifiers (`int`, `signed`, etc.) can appear in any order before a declaration. For example, `signed int x` and `int signed x` are equivalent declarations. To handle this, my parser first consumes all type specifier tokens of a declaration, then constructs an `ArithmeticType` AST node from them. It uses a bitfield where each bit represents the presence of one of the type specifiers in the type. The bitfield is the normalised representation of a type; every possible declaration that is equivalent to a type will have the same bitfield. The declarations above would construct the bitfield `0b00010100`, where the two bits set represent `signed` and `int` respectively. For each type specifier, the corresponding bit is set. Then, the bitfield is matched against the possible valid types, to assign the type to the AST node.

3.4 Middle End

The middle end takes an AST as input, and transforms it to intermediate code. It also runs the Relooper algorithm on the IR. Optimisations are run on the IR in this stage; they are described in Section 3.7.

3.4.1 Intermediate Code Generation

I defined a custom three-address code IR (the instructions are listed in Appendix B). The IR contains both the program instructions and necessary metadata, such as variable type information, the mapping of variable and function names to their IDs, etc. The instructions and metadata are contained in separate sub-structs within the main IR struct, which enables the metadata to be carried forwards through stages of the compiler pipeline while the instructions are transformed at each stage. In

the stage of converting the AST to IR code, the instructions and metadata are encapsulated in the `Program` struct.

Many objects in the IR require unique IDs, such as variables and labels. I created a `Id` trait to abstract this concept, together with a generic `IdGenerator` struct (Listing 3.7). The ID generator internally tracks the highest ID that has been generated so far, so that the IR can create as many IDs as necessary without needing to know anything about their implementation. IDs are generated inductively: each ID knows how to generate the next one.

```rust
pub trait Id {
    fn initial_id() -> Self;
    fn next_id(&self) -> Self;
}

pub struct IdGenerator<T: Id + Clone> {
    max_id: Option<T>,
}

impl<T: Id + Clone> IdGenerator<T> {
    pub fn new() -> Self {
        IdGenerator { max_id: None }
    }

    pub fn new_id(&mut self) -> T {
        let new_id = match &self.max_id {
            None => T::initial_id(),
            Some(id) => id.next_id(),
        };
        self.max_id = Some(new_id.to_owned());
        new_id
    }
}
```

Listing 3.7: Implementation of the `Id` trait and `IdGenerator`.

To convert the AST to IR code, the compiler recursively traverses the tree, generating three-address code instructions and metadata as it does so. At the highest level, the AST contains a list of statements. For each of these, the compiler calls `convert_statement_to_ir(stmt, program, context)`. `program` is the mutable intermediate representation, to which instructions and metadata are added as the AST is traversed. `context` is the context object described in Section 3.4.2, which passes relevant contextual information through the functions recursively.

The core of converting a statement or expressions to IR code is pattern matching the AST node, and generating IR instructions according to its structure, recursing into sub-statements and expressions. The case for a **`while`** statement is shown in Listing 3.8; the labels and branches to execute a while loop are added, and the instructions to evaluate the condition and body are generated recursively. Other control-flow structures are similar.

Some statements took a little more care to ensure the semantic meaning of the program is preserved. For example, **`switch`** statements can contain **`case`** blocks in any order. Some blocks may fall-through

```
function ConvertStatementToIR(stmt, program, context)
    instrs ← []
    match stmt
        case While(condition, body)
            startLabel, endLabel ← create new labels for start and end of loop
            Push new loop context to Context object
            instrs += label <startLabel>
            instrs += ConvertExpressionToIR(condition, program, context)
            instrs += branch <endLabel> if condition == 0
            instrs += ConvertStatementToIR(body, program, context)
            instrs += branch <startLabel>
            instrs += label <endLabel>
            Pop loop context from Context object
        end case
        ... other AST statement nodes ...
    end match
    return instrs
end function
```

Listing 3.8: Pseudocode for the `convert_statement_to_ir()` function.

to the next block, and there may be a **`default`** block. I handled this by first generating instructions for each case block, and storing them in a switch context until all blocks have been seen. We then put conditional branches to each case block at the start of the switch statement, followed by each of the block instructions. By doing this, there is no direct fall-through between any blocks; instead, a block that falls through to the next block will end in a branch instruction to the start of that block. This also allows **`default`** blocks to be easily handled; we just add an unconditional branch to the default block after all the conditional branches. If there is no **`default`** block, the conditional branches are followed by an unconditional branch to the end of the **`switch`** statement. Figure 3.4 shows the structure of the generated instructions for **`switch`** statements.

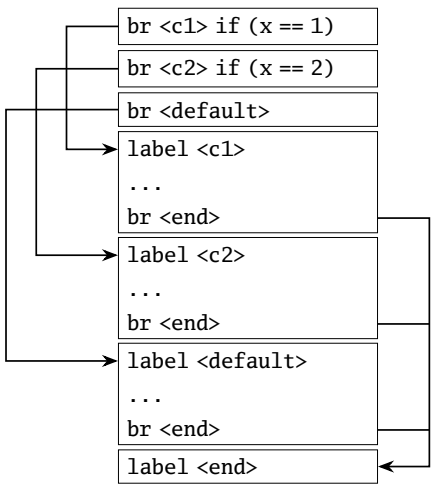


Figure 3.4: IR instructions generated for **`switch`** statements.

Declaration statements have a lot of different cases, each of which I had to handle separately. For

example, variables may be declared without being initialised with a value.

Function declarations only differ syntactically from other declarations by the type of the identifier; so we have to handle them in the same place. For function definitions (i.e. declarations plus body), we transform the body code and add a new function to the IR.

Arrays are complicated because there are multiple ways their length can be specified. It can be given explicitly in the declaration, or implicitly inferred from the length of the initialiser. To further complicate them, an explicit size can either be a static value or a variable that is only known at runtime (creating a variable-length array). To handle variable length arrays, an instruction is inserted to allocate space on the stack for it at runtime. Array and struct initialisers are handled by first allocating memory for the variable, then storing the value of each of the inner members.

Some expressions can be evaluated by the compiler ahead-of-time, for example array length expressions. I implemented a compile-time expression evaluation function that can handle arithmetic expressions and ternaries. Expressions are converted according to their structure, recursing into sub-expressions.

3.4.2 Context Object Design Pattern

Throughout the middle and back ends, I used a design pattern of passing a context object through all the function calls. For example, when traversing the AST to generate IR code, the `Context` struct in Listing 3.9 is used to track information about the current context we are in with respect to the source program. For example, it tracks the stack of nested loops and switch statements, so that when we convert a **`break`** or **`continue`** statement, we know where to branch to.

```
pub struct Context {
    loop_stack: Vec<LoopOrSwitchContext>,
    scope_stack: Vec<Scope>,
    pub in_function_name_expr: bool,
    function_names: HashMap<String, FunId>,
    pub directly_on_lhs_of_assignment: bool,
}
```

Listing 3.9: The context data structure used when converting the AST to IR code.

In an object-oriented language, this would often be achieved by encapsulating the methods in an object and using private state inside the object. Rust, however, is not object oriented, and I found this approach to offer more modularity and flexibility. Firstly, the context information itself is encapsulated inside its own data structure, allowing methods to be implemented on it that gives calling functions access to exactly the context information they need. It also allows creating different context objects for different purposes. In the target code generation stage, the `ModuleContext` stores information about the entire module being generated, whereas the `FunctionContext` is used for each individual function being converted. The `FunctionContext` lives for a shorter lifetime than the `ModuleContext`, so being able to separate the data structures is ideal.

3.4.3 Types

The following types are supported by the IR, mirroring the types supported by the C language [19, ch. 5]. **Ux** and **Ix** types represent unsigned and signed x-bit integers, respectively. Enumeration types (enums) are supported; their values are encoded as **U64**s. I followed the standard implementation convention for the bit size of **char**s, **short**s, **int**s, and **long**s; 8, 16, 32, and 64 bits respectively[1].

$$\begin{aligned} T = {} & \texttt{I8} \mid \texttt{U8} \mid \texttt{I16} \mid \texttt{U16} \mid \texttt{I32} \mid \texttt{U32} \mid \texttt{I64} \mid \texttt{U64} \mid \texttt{F32} \mid \texttt{F64} \mid \texttt{Void} \\ & \mid \texttt{Struct}(T[]) \mid \texttt{Union}(T[]) \\ & \mid \texttt{Pointer}(T) \mid \texttt{Array}(T, \mathit{size}) \\ & \mid \texttt{Function}(T, T[], \mathit{is\_variadic}) \end{aligned}$$

I created a data structure to represent these types, along with methods for operations on those types.

I implemented the ISO C unary and binary conversions for types [19, pp. 174–176]. They are applied before unary and binary operations respectively. Unary conversion reduces the number of types an operand can be. Smaller integer types are promoted to **I32**/**U32** appropriately, and **Array**(T, _) types are converted to **Pointer**(T). Binary conversions make sure that both operands to a binary operation are of the same type. First, the unary conversions are applied to each operand individually. Then, if both operands are an arithmetic type, and one operand is a smaller type than the other, the smaller type is converted to the larger type. This includes integer types being promoted to float types.

When transforming each AST node to intermediate code, the compiler checks that the types of the operands are valid (after unary/binary conversion) and stores the corresponding type of the result variable. For example, before generating a `t = a < b` instruction, the compiler checks whether `a` and `b` are comparable arithmetic types—if not, a compile error is thrown—and sets the type of `t` to **I32**[2]. This ensures that the IR passed to the back end is type safe.

3.4.4 The Relooper Algorithm

The Relooper algorithm is described in Section 2.2. In this section I describe my implementation. The Relooper stage takes the IR as input, and transforming the instructions to a structure of Relooper blocks, but preserving the program metadata.

First, the intermediate code is 'soupified'. This is the process of taking the linear sequence of instructions and producing a set of *labels*[3] (basic blocks), which are the input to the Relooper algorithm itself. Each label starts with a `label` instruction and ends in a `branch`. The process of soupifying is as follows:

1. Remove any label fall-through. By this I mean any label instruction that is not preceded by a branch, where control will flow to directly from the previous instruction. This is removed by inserting a branch instruction before each label instruction, if one does not already exist, branching to that label.

[1] The C specification doesn't specify the exact bit widths, only the minimum size.
[2] **I32** is used to represent booleans.
[3] The term 'label' is overloaded here, though the two concepts are related; a 'label instruction' refers to the intermediate code instruction, whereas a 'label' is a basic block that starts with a label instruction.

This will generate many redundant branch instructions, however they get removed when the Relooper blocks are transformed to target code.

2 Insert an unconditional branch after each conditional branch, to ensure that conditional branches do not occur in the middle of a block.

```
br <l1> if ...    ▷ Original conditional branch
br <l2>           ▷ New unconditional branch
label <l2>        ▷ New label <l2>
...
```

3 If there is no label instruction at the very start of the instructions, add one.

4 Merge any consecutive label instructions into a single instruction, updating branches to the labels accordingly.

5 Finally, divide the instructions into blocks by passing through the instructions sequentially and starting a new block at each label instruction.

Steps 1 to 4 transform the instruction sequence until it can be directly split into label blocks. All control flow between labels is made explicit.

For every function in the IR, the instructions are soupified and then the algorithm described in Section 2.2.2 is used to create a Relooper block. The `IdGenerator` struct described above is used to assign each loop and multiple block a unique ID.

The reachability of each label is calculated by taking the transitive closure of its possible branch targets. Starting with a copy of the set of possible branch targets of a label, we iteratively add the possible branch targets of each label in the set, until there are no more changes. (The reachability is a set so that no duplicates are added.)

Maybe talk a bit more about the actual relooper implementation?

3.5 Back End: Target Code Generation

- Int and vector encoding?

In the back end, I defined data structures to directly represent a WebAssembly module with its constituent sections. I also defined a `WasmInstruction` enum to represent all possible WebAssembly instructions, and data structures to represent value types. The back end generates a `WasmModule` containing `WasmInstructions`, the byte representation of which is written out to a binary file as output of the compiler.

I defined a `ToBytes` trait that is implemented by all data structures that will be written out to the binary. This provides a layer of abstraction between the program data structures and the actual byte values in the binary; each structure only needs to know the byte values specific to itself, e.g. for a specific instruction, and nothing more.

The core function of the back end is to transform IR instructions to WebAssembly instructions. Since WebAssembly is a stack-based architecture, the general pattern for converting an instruction is:

- Push the value of each operand onto the stack (loading any variables from memory).
- Perform the operation, which leaves the result on top of the stack.
- Store the result from the stack back to a variable in memory.

Most instructions are a variation of this pattern. Listing 3.10 shows the code generated for an add instruction. One subtlety is that `store` instructions take the memory address as their first operand, and the value to store as their second operand. This means the address of the destination variable needs to be pushed onto the stack *before* the source operands are loaded and the operation is performed.

```
c = a + b
```

(a) Intermediate code.

```
i32.const <addr of c>  ;; address for the store instruction
i32.const <addr of a>
i64.load               ;; load a onto the stack
i32.const <addr of b>
i64.load               ;; load b onto the stack
i64.add                ;; perform operation on a and b
i64.store              ;; store result from top of stack
```

(b) Generated WebAssembly code.

Listing 3.10: Transforming an add instruction to target code, assuming `a` and `b` are variables of type `i64`.

I defined `load` and `store` functions that take the IR type of a variable and return correctly-typed load and store instructions respectively. The `store` function encapsulates the fact that the address operand has to come before the value to store; this helps to ensure correctness as it is only defined in one place.

I used the same ID generator pattern as in the middle end, to generate unique indices for items in the WebAssembly module, such as functions and types.

3.5.1 Memory Layout

One of the other main functions of the back end is to generate code that manages the memory layout. A large part of this is the function call stack; pushing new stack frames when functions are called, and popping them when functions return. Figure 3.5 shows the memory layout I defined for the compiler.

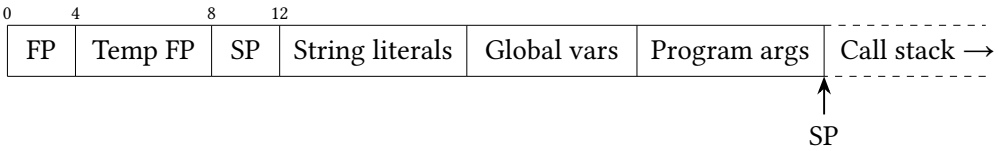


Figure 3.5: Memory structure, with addresses increasing to the right.

The first section of memory contains the frame pointer (FP) and stack pointer (SP), as well as a temporary pointer storage location which is used in intermediate steps of manipulating stack frames.

I chose to call this the 'temporary FP', because it is mainly used to hold the new value that the FP will be set to once we have finished setting up a new stack frame. In a register architecture, these pointers would be stored in registers, however WebAssembly does not have any registers, so instead I chose to allocate them at known locations in memory.

The next section of memory contains any string literals that are defined in the program. In C, a string literal is simply a null-terminated array of characters, which a variable accesses via a **char***. My IR has a dedicated `PointerToStringLiteral` instruction which, in the back end, gets converted to an instruction that pushes the address of the corresponding string onto the stack. All string literals are allocated at compile-time and have compile-time known addresses.

Space is allocated for all global variables at compile-time. These are allocated in the same way as local variables, which is described below.

Programs may accept command-line arguments, which are stored into memory by the runtime environment. This is described in more detail in Section 3.6 below.

The function call stack grows upwards dynamically from this point. I did not implement heap storage, so memory is only used from one end, unlike in standard C compilers such as GCC.

3.5.2 Stack Frame Operations

Whenever a function is called, a new stack frame is constructed at the top of the stack in memory. Figure 3.6 shows the structure of each stack frame.

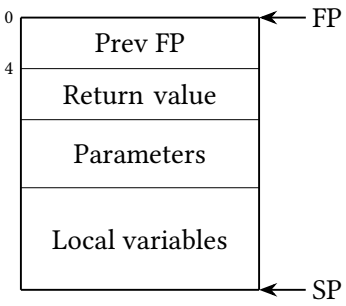


Figure 3.6: Stack frame layout, with addresses increasing downwards.

I defined callee/caller conventions for pushing and popping stack frames. The caller is responsible for constructing the frame with the previous FP value, space for the return address, and any function parameters. The caller also deallocates the stack frame once the function returns. The callee is responsible for allocating space for its local variables on the stack (discussed in Section 3.5.3 below).

The procedure for pushing a new stack frame is the following:

- Store the current FP at the top of the stack.
- Copy the current SP to Temp FP: the address of the start of the stack frame.
- Allocate space for the return value on top of the stack, according to the return type.
- Store each function parameter on top of the stack. This is either copying a variable from the stack frame below or storing a constant.
- Set the FP to the value held in Temp FP.

Using the temporary FP here is necessary because if we directly wrote to the FP, we would not be able to copy any variables in as parameters. (Local variables have FP-relative addresses.) We also

cannot wait to save the new FP address until after we have copied the variables, because by then the SP has been moved and no longer points to the start of the stack frame. (The SP is updated every time a value is stored on top of the stack.)

Popping a stack frame is simpler:

- Set the SP to the current value of the FP. The new top of the stack is the top of the previous stack frame.
- Restore the previous value of the FP.
- If the function returns a result to a variable, copy the return value from the stack frame to the destination variable.

We can pop the frame simply by moving the SP because the data above the SP will never be read; when the stack grows again, it will be overwritten.

3.5.3 Local Variable Allocation

Initially I implemented a naive variable allocation strategy. As an extension, I implemented a more optimal allocation strategy, described in Section 3.7.3.

Variable allocation is done at compile time, so that variable accesses can be static FP offsets rather than requiring a runtime address table. Only in specific cases—variable-length arrays, for example—are variables allocated at runtime. My IR has a dedicated `AllocateVariable` instruction which takes the number of bytes to allocate as an operand.

Each variable, including temporary variables, is allocated a dedicated byte range in memory; none of the variables overlap. This safely maintains the programmer's memory model, however it is an inefficient use of memory, hence the later optimisation.

Local variables are stored as part of the function stack frame, the address of which is runtime-dependent. Therefore, variables are allocated as offsets from the FP. At runtime, a variable is accessed by first loading the FP, then adding the variable offset to get the variable's memory address.

3.6 Runtime Environment

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

The purpose of the runtime environment is to provide an interface between the WebAssembly binary and the system, allowing it to be executed. I chose to use Node.js [11], because this allows the binary to be run locally. The other option would be to use JavaScript within a web browser, which would be the primary reason to use WebAssembly; however, for the purposes of this project that approach would have added unnecessary complexity.

Reason for using Node.js maybe should go in preparation chapter

The runtime is responsible for instantiating the WebAssembly module. A module must be instantiated before any functions it contains can be executed. One of the main functions of the instantiation is to pass imports into the WebAssembly module. The runtime creates a new linear memory for the module to use, and also imports my skeleton library functions. The memory is created by the runtime so that the standard library functions can have access to it. If the memory were created by the WebAssembly module, which is the other option, then the imported functions would not be able to reference it.

Once the module has been instantiated, the runtime stores the command-line program arguments into memory. As is standard for C programs, the first argument is always the name of the program being run (i.e. the name of the WebAssembly binary). Arguments are passed to the `main` function as an array of **`char`** pointers (`argv`), plus the argument count (`argc`). To facilitate this, the runtime first allocates space for the array of pointers, at the compile-time known address immediately after the global variables. After the pointer array, the actual argument values are stored (as strings), and each corresponding pointer is set (Figure 3.7). The runtime then sets the SP to immediately after the last argument.

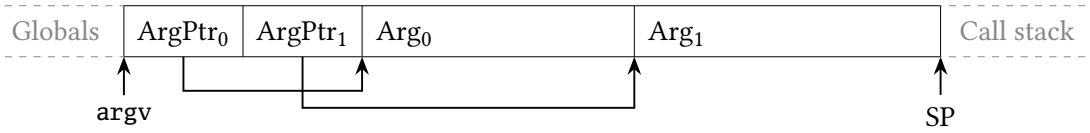


Figure 3.7: Memory structure of program arguments, with addresses increasing to the right. Here, `argc` = 2.

Finally, the runtime calls the exported `main` function, with the values of `argc` and `argv` as parameters.

3.7 Optimisations

After completing the main compiler pipeline, I implemented optimisations in the middle and back ends. I performed unreachable procedure elimination and tail-call optimisation in the middle end, and in the back end I created a more optimal stack allocation policy for local variables.

3.7.1 Unreachable Procedure Elimination

Unreachable procedure elimination removes all functions that are never called. I do not support function pointers; all function calls are direct. Therefore, if no `call` instruction references a particular function, that function is guaranteed to never be called.

First, the call graph is generated. This is a directed graph where each node is a function in the program and each edge represents a syntactic function call. To maintain correctness, the constructed graph is a superset of the *semantic* call graph, which contains all the function calls that can happen in actual execution. The semantic graph is undecidable at compile time[1], so we calculate the syntactic call graph.

I used an adjacency list to store the call graph, since the graph is likely to be sparse. (An adjacency matrix would be more suitable for densely-connected graphs.) The call graph data structure also

[1]Due to the undecidability of arithmetic, it is not possible to decide in general which control flow path will be taken.

has a set of entry nodes, which are functions that are called from the global scope of the program (e.g. `main()`).

To generate the graph, all functions in the IR are added as nodes, then an edge is added for every `call` instruction. For a `call y` instruction found inside function `x`, a directed edge is added from node `x` to node `y`. Any functions that are called from the global scope, as well as the `main` function, are added to the set of entry nodes.

Once the call graph is constructed, it is used to find functions that are never called. First mark every function as unused. I then implemented a breadth-first search (BFS) over the graph, starting from each entry node. Each node we reach is marked as used. Once the search returns, all nodes still marked as unused can be safely removed from the program.

The benefit of using a BFS rather than simply looking for nodes with no incoming edges is that it can handle cycles. A program may contain cycles of functions that call each other but that are never called from the rest of the program, so the cycle is never entered. In this case, the nodes would have incoming edges, but would not be reached in a BFS, allowing them to be removed.

3.7.2 Tail-Call Optimisation

When a function is called recursively, a new stack frame is pushed onto the call stack. A recursive tail-call is when the recursive call is the last operation of the calling function, and the result from the recursive call is directly returned from the caller. In these cases, the caller's stack frame is unneeded as soon as the recursive call is made; when the recursive call returns, the return value is copied to the stack frame below, but nothing more is done. Therefore, tail-call optimisation aims to remove the unnecessary stack frames, thereby reducing the amount of stack memory used. The new stack frame replaces the caller's stack frame, instead of being pushed on top of it.

This can have a dramatic impact on memory usage; it turns a function's memory usage from $\mathcal{O}(n)$ to $\mathcal{O}(1)$ (where n is the recursion depth). Not only does this make a program much more efficient, but for deep enough recursion it allows programs to run that would otherwise crash due to running out of memory.

There is a distinction between general tail-calls and recursive tail-calls. In general, a tail-call is any function call that is the last action of a function[1], whereas a recursive tail-call is specifically a recursive call. Recursive tail-calls are easier to optimise because the function signature is the same, and also have the greatest performance impact because many recursive calls may be made.

Initially I implemented this optimisation by replacing the stack frame in the target code generation stage, however this did not actually work properly. Function calls are implemented using WebAssembly functions and the `call` instruction, which means that the WebAssembly virtual machine has its own call stack as well as the call stack I manage in memory. At large recursion depths, the WebAssembly stack was still running out of memory. Instead, I implemented an approach of transforming the recursion into iteration in the IR stage. This was much more successful. However, I kept the old optimisation for the cases of non-recursive tail-calls, because it does still reduce memory usage there. I will describe both approaches in turn below.

[1] A tail-call necessarily has the same return type as the calling function, because otherwise an explicit type conversion instruction would already have been inserted before the `return` instruction.

3.7.2.1 Approach 1: Replacing the Caller's Stack Frame at Runtime

My first approach was to replace the caller's stack frame at runtime rather than push a new stack frame on top whenever a tail-call is made.

To do this, in the middle end we find all `call` instructions that are directly returned from the function, and replace them with a `tail-call` instruction.

In the target code generation stage, `tail-call` instructions are handled by re-using the current stack frame instead of pushing a new one. The construction of this is very similar to how a stack frame is normally pushed. The frame pointer and space for the return value are left as they are; they don't need to be changed for the new stack frame.

To store the parameters in the new stack frame, first they are copied to a region of unused memory above the stack, then copied to their respective positions in the stack frame. This ensures the needed values in the old stack frame are not overwritten by the new stack frame before they are used.

This approach works for both recursive and non-recursive tail-calls. However, as described above, it is not ideal for recursive tail-calls, because of WebAssembly's own function call stack. Therefore I implemented the second approach below.

3.7.2.2 Approach 2: Transforming Recursion to Iteration

The second approach is entirely contained in the middle end. It only targets recursive tail-calls; these are where the majority of the performance gains are to be found. It removes the recursive calls altogether, and replaces them with iteration back to the start of the function. Therefore, no new stack frame will be allocated.

Similarly to above, we start by finding all `call` instructions that are both recursive and are tail-calls. In the current function, there is a variable holding each parameter. For each argument to the recursive call, we copy it to the variable holding the corresponding parameter in the current function. Once we have done this assignment for each parameter, we add a branch instruction to the start of the function.

This approach solves the problem of the WebAssembly virtual machine's call stack running out of memory, because the function calls have been entirely removed. The recursion is now contained within a single function using iteration.

3.7.3 Stack Allocation Policy

My initial variable allocation policy is described in Section 3.5.3 above. Each variable is allocated a disjoint address range in memory. However, this is very inefficient since most variables are temporary variables that are only used once, and many of them are not used at the same time as each other (i.e. they never *clash*).

Variables that never clash can safely be allocated to the same address range. The challenge of this optimisation was to find a more optimal way of allocating variables such that as little memory is used as possible.

The optimised allocation is performed in the following steps:

1. Remove dead variables.
2. Generate the instruction flowgraph.
3. Perform live variable allocation.
4. Generate the clash graph.
5. Allocate variables from the clash graph.

3.7.3.1 Live Variable Analysis

An instruction flowgraph is similar to the function call graph used in Section 3.7.1 above, but on the level of individual instructions within a function. Each node in the graph is a single instruction, and successors of a node are any instructions that can be executed as the next instruction along some execution path. For example, branch instructions will have multiple successors.

The flowgraph is generated by recursing through the program blocks (the output of the Relooper algorithm, Section 3.4.4), creating a node for each instruction, and creating edges along all possible control flow paths.

Live variable allocation (LVA) is run on the flowgraph to find where each variable is *live*. A variable is said to be *live* (syntactically[1]) at an instruction if the value of the variable is used along any path in the flowgraph before it is redefined.

LVA is a backwards analysis, which means that liveness information is propagated backwards through the flowgraph. First, we define *def* and *ref* sets for each instruction: $def(i)$ is the set of all variables referenced by instruction n, and $ref(n)$ is the set of all variables assigned to by n. We then define $live(n)$, the set of variables that are live immediately before instruction n, as follows:

$$live(n) = \left(\left(\bigcup_{s \in succ(n)} live(s)\right) \setminus def(n)\right) \cup ref(n)$$

We start with all variables live at successors of n, remove all variables that n assigns to, and add any variables that n references. When n assigns to a variable, it is no longer live for previous instructions, because any previous value has been overwritten. When n references a variable, it becomes live for previous instructions, until an instruction assigns to that variable.

To implement LVA, we keep iterating over every instruction in the flowgraph, applying the above equation to update the set of live variables, until there are no more changes. Initially the set of variables live at each instruction is the empty set. The algorithm is guaranteed to find the smallest set of live variables (it does not add unnecessary overapproximations). Listing 3.11 shows my implementation.

One complication of LVA I had to solve was that if a variable is assigned to but never subsequently referenced, it will never be marked as live by the analysis. The next stage of the optimisation then marks it as having no clashes with any other variables. However, the write to the variable may occur while other variable are live; and the compiler would happily allocate this variable in an

[1] Compared to *semantic* liveness, which refers to actual possible execution behaviour, but is undecidable at compile time. Syntactic liveness is a safe overapproximation of semantic liveness.

overlapping location, producing incorrect results. To solve this, I added an optimisation to remove dead variables before running LVA. I made sure to keep any side-effect producing instructions (e.g. function calls) while removing assignments to variables that are not subsequently accessed.

```
pub type LiveVariableMap = HashMap<InstructionId, HashSet<VarId>>;
pub fn live_variable_analysis(flowgraph: &Flowgraph) -> LiveVariableMap {
    // For every instr, which vars are live at that point
    let mut live: LiveVariableMap = LiveVariableMap::new();
    let mut changes = true;
    while changes {
        changes = false;
        for (instr_id, instr) in &flowgraph.instrs {
            // ⋃_{s ∈ succ(n)} live(s)
            let mut out_live: HashSet<VarId> = HashSet::new();
            for successor in flowgraph.successors.get(instr_id).unwrap() {
                out_live.extend(live.get(successor).unwrap_or(&HashSet::new()).to_owned());
            }
            // ∖ def(n)
            for def_var in def_set(instr) {
                out_live.remove(&def_var);
            }
            // ∪ ref(n)
            for ref_var in ref_set(instr) {
                out_live.insert(ref_var);
            }
            // Update live variable set, and compare to previous value for changes
            let prev_live = live.insert(instr_id.to_owned(), out_live.to_owned());
            match prev_live {
                None => {
                    changes = true;
                }
                Some(prev_live_vars) => {
                    if prev_live_vars != out_live {
                        changes = true
                    }
                }
            }
        }
    }
    live // Return the sets of live variables
}
```

Listing 3.11: Live variable allocation implementation.

3.7.3.2 Generating the Clash Graph

The clash graph is generated from the results of LVA. Any variables that are simultaneously live *clash*; i.e. they cannot be allocated to overlapping addresses, because they are in use at the same

time. Listing 3.12 describes how the clash graph is generated.

The clash graph is an approximation whenever variable pointers are present. When a variable has its address taken, the pointer may be passed around the program as a value, so it is no longer possible to track exactly where the variable is accessed. Therefore, to ensure safety of the optimisation, we make each address-taken variable clash with every other variable. I implemented this by storing a set of 'universal clashes' in the clash graph; when checking if two variables clash, if one of them is a 'universal clash', they clash even if they would otherwise not.

---

for instruction n **do**
    $\ell \leftarrow live(n)$
    while $|\ell| > 1$ **do**
        $var_1 \leftarrow \ell$.pop()
        for $var_2 \in \ell$ **do**
            Add edge (v_1, v_2) to clash graph
        end for
    end while
    if n takes the address of its operand var **then**
        Add var as a universal clash to clash graph
    end if
end for

Listing 3.12: Algorithm to generate the clash graph from LVA.

3.7.3.3 Allocating Variables from the Clash Graph

In the Part II Optimising Compilers Course, a register allocation heuristic was described which allocates variables in order of most clashes to least clashes. When each variable is allocated, it is allocated a register that maximally overlaps with already allocated variables, while avoiding registers containing variables it clashes with.

My variable allocation problem is similar in many regards, however it has some key differences. I am allocating variables in memory rather than to registers, so each variable occupies a byte *range* rather than a single location. Variables can have different sizes, and are not required to be aligned, so it is possible that variables may partially overlap (e.g. `t0` $\mapsto [0, 4)$, `t1` $\mapsto [2, 6)$). There is also (effectively) no limit to the amount of memory available, compared to the very limited set of registers most compilers target. The goal of my optimisation is to use as little memory as possible, whereas the goal of register allocation is to most efficiently use the constrained set of registers.

I modified the register allocation method to the following heuristic for variable allocation:

1. Choose a variable with the least number of clashes. Break ties by choosing smaller variables.
2. Remove the variable and its edges from the clash graph.
3. Allocate the variable to the lowest memory address where it doesn't clash with already allocated variables.
4. Repeat from Step 1 until the clash graph is empty.

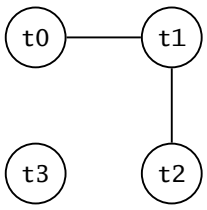
In Step 3, variables are always allocated to the lowest position on the stack possible (satisfying clash constraints). This prioritises overlapping non-clashing variables at lower addresses; only variables that clash lots will be pushed to higher addresses. We want to do this to keep the stack size as small as possible.

I tested allocating variables in both orders: least clashes first and most clashes first. I found that contrary to register allocation, allocating variables with least clashes first resulted in more efficient memory use. I think this is because of the preference to always allocate to lower addresses; this priority does not exist in the register allocation algorithm, since registers are all equal. In my heuristic, it is beneficial to have variables with less clashes at lower addresses, because this allows more variables to overlap there, and fewer variables will be pushed to higher addresses.

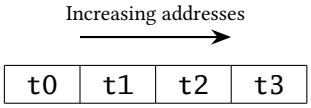
Figure 3.8 shows an example of the impact this optimisation has. In the top left is the intermediate code for which memory is being allocated. The live variables at each instruction—calculated by LVA—are shows to the right of each line. In the top right is the clash graph, showing which variables cannot be allocated to the same memory. The bottom left shows how the naive allocation policy would allocate the variables in memory; sequentially and non-overlapping. In the bottom right is the result of the optimised allocation policy: variables `t0`, `t2`, and `t3` all use the same memory location because they are never live at the same time. Notice how all three variables use the lowest location, rather than, for example, `t3` overlapping with `t1`. This is due to the heuristic of always allocating to the lowest valid address.

`t0 = 3`	{}
`t1 = 2`	{`t0`}
`t2 = t0 + t1`	{`t0`, `t1`}
`t3 = t2 + t1`	{`t2`, `t1`}

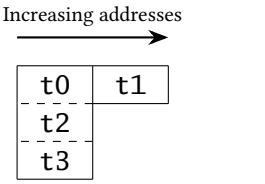
(a) Intermediate code, with live variable allocation.



(b) Clash graph.



(c) Naive allocation policy.



(d) Optimised allocation policy.

Figure 3.8: Example of optimised stack allocation policy.

4

Evaluation

Word budget: ~2000–2400 words

"Signs of success, evidence of thorough and systematic evaluation"
- How many of the original goals were achieved?
- Were they proved to have been achieved?
- Did the program really work?
- Answer questions posed in the introduction
- use appropriate techniques for evaluation, eg. confidence intervals

In this chapter, I evaluate my project against my success criteria, showing that all the success criteria were achieved. In Section 4.2 I demonstrate that the compiler is correct using a variety of test programs. In Section 4.3 I evaluate the impact of the optimisations I implemented, particularly showing significant improvements in memory usage.

4.1 Success Criteria

The success criteria for my project, as defined in my project proposal, are:

- The program generates an abstract syntax tree from C source code.
- The program transforms the abstract syntax tree into an intermediate representation.
- The program uses the Relooper algorithm to transform unstructured to structured control flow.
- The program generates WebAssembly binary code from the intermediate representation.
- The compiler generates binary code that produces the same output as the source program.

All of my success criteria have been met. The first four criteria correspond to the main stages of the compiler pipeline, respectively. The correctness of this pipeline is verified by the correctness of the generated binary code. The generated binary would not be correct if any of the stages had not been successful.

I used a variety of test programs to verify the success of the fifth criteria. This is described in Section 4.2.

4.2 Correctness Testing

I wrote a suite of test programs in C to evaluate the correctness and performance of my compiler. There were two types of program:

- 18 'unit test' programs, which test a specific construct in the C language; and
- 11 'full programs', which represent real workloads that the compiler would be used for.

Programs of the first type are not strictly unit tests by the standard definition. Unit tests verify the functionality of individual units of source code, in isolation from the rest of the application. Each test should test one particular behaviour of that unit, and should be independent from the rest of the program's functionality [23]. My test programs don't test an isolated behaviour of the compiler's source code. Instead, they test a single behaviour of the C source code that is being compiled (for example, dereferencing a pointer), verifying that the compiler pipeline maintains the correct behaviour. This allowed me to trace bugs to the units of code that transformed that particular construct.

Examples of 'full programs' include Conway's Game of Life [24], calculating the Fibonacci numbers (recursively), and finding occurrences of a substring in a string. Six of the test programs I used were sourced from Clib [25], which contains many small utility packages for C. Those that I used had no external dependencies, and were useful in verifying that my compiler worked for other people's code as well as my own. Clib is licensed under the MIT license, which permits use of the software "without restriction". The remaining five full programs and all 18 unit test programs were written by myself.

For all my tests, I used GCC[1] as the reference for correctness. I deemed a program to be correct if it produced the same output as when compiled with GCC. To facilitate this, I made liberal use of `printf`, to output the results of computations.

I wrote a test runner script to ensure that I maintained correctness as I continued to develop the compiler, fix bugs, and implement optimisations. This script read a directory of `.yaml` files, which described the path to each test program, and the arguments to run it with. It then compiled the program with both my compiler and with GCC, and compared the outputs. A test passed if the outputs were identical, and failed otherwise. The script reported which tests, if any, failed.

I also implemented some convenience features into the test script. The command-line interface takes an optional 'filter' argument, which can be used to run a subset of the tests whose name matches the filter. The script can also be used to run one of the test programs without comparing to GCC, printing to the standard output. This allows easier manual testing.

To prevent bugs from accidentally being introduced into my compiler, I set up the test suite to run automatically as a commit hook whenever I committed changes to the Git repository. This would prevent a commit from succeeding if any of the tests failed, allowing me to make corrections first. This ensured that the version of my project in source control was always correct and functioning.

[1] GCC version 11.3.1.

4.3 Impacts of Optimisations

In the following sections, I will evaluate the impacts that the optimisations I implemented had. For unreachable procedure elimination, I will evaluate the reduction in code size. For tail-call optimisation and stack allocation optimisation, I will evaluate the effectiveness at reducing memory usage.

To evaluate the memory usage optimisations, I inserted profiling code when compiling the programs, to measure the size of the stack throughout program execution. When generating instructions that move the stack pointer, the compiler additionally inserts a call to `log_stack_ptr`, a function imported from the JavaScript runtime. `log_stack_ptr` reads the current stack pointer value from memory and appends it to a log file. I wrote a Python script to visualise the resulting data. The plots show how the size of the stack grows and shrinks throughout the execution of the program. Figures 4.1 to 4.4 are generated using this method.

4.3.1 Unreachable Procedure Elimination

In the context of this project, unreachable procedure elimination mainly has benefits in removing unused standard library functions from the compiled binary. When a standard library header is included in the program, the preprocessing stage inserts the entire code of that library[1]. If the program only uses one or two of the functions, most of them will be redundant. Unreachable procedure elimination is able to safely remove these, resulting in a smaller binary.

I only implemented enough of the standard library to allow my test programs to run, so the impact of this optimisation is limited by the number of functions imported. If I were to implement more of the standard library, this optimisation would become more important.

The standard library header with the most functions that I implemented was `ctype.h`. This header contains 13 functions, of which a program might normally use two or three. This is where I saw the biggest improvement from the optimisation. Programs that used the `ctype` library saw an average file-size reduction of 4.7 kB.

The other standard library headers I implemented only contained a few functions, so the impact of this optimisation was much more limited. However, as mentioned above, if I implemented more of the standard library, I would see much more of an improvement.

Due to this difference in the standard library header files, and also to the fact that source programs can arbitrarily contain functions that are never used, it is not meaningful to calculate aggregate metrics across all my test programs. However, testing each program individually does verify that any functions that are unused are removed from the compiled binary.

> Can you graph the results from this section? Benchmarks along the x axis and code size before / after optimisation on the y axis. Or reduction in code size on y?

[1] That is, the entirety of my skeleton implementation of that library, rather than the actual standard library code.

4.3.2 Tail-Call Optimisation

My implementation of tail-call optimisation was successful in reusing the existing function stack frame for tail-recursive calls. The recursion is converted into iteration within the function, eliminating the need for new stack frame allocations. Therefore, the stack memory usage remains constant rather than growing linearly with the number of recursive calls.

One of the functions I used to evaluate this optimisation was the function in Listing 4.1 below, that uses tail-recursion to compute the sum of the first n integers.

```
long sum(long n, long acc) {
    if (n == 0) {
        return acc;
    }
    return sum(n - 1, acc + n);
}
```

Listing 4.1: Tail-recursive function to sum the integers 1 to n

Figure 4.1 compares the stack memory usage with tail-call optimisation disabled and enabled. Without the optimisation, the stack size clearly grows linearly with n. When running the program with $n = 500$, a stack size of 46.3 kB is reached. When the same program is compiled with tail-call optimisation enabled, only 298 bytes of stack space are used; a 99.36 % reduction in memory usage. Of course, the reduction depends on how many iterations of the function are run. In the non-optimised case, the stack usage is $\mathcal{O}(n)$ in the number of iterations, whereas in the optimised case it is $\mathcal{O}(1)$.

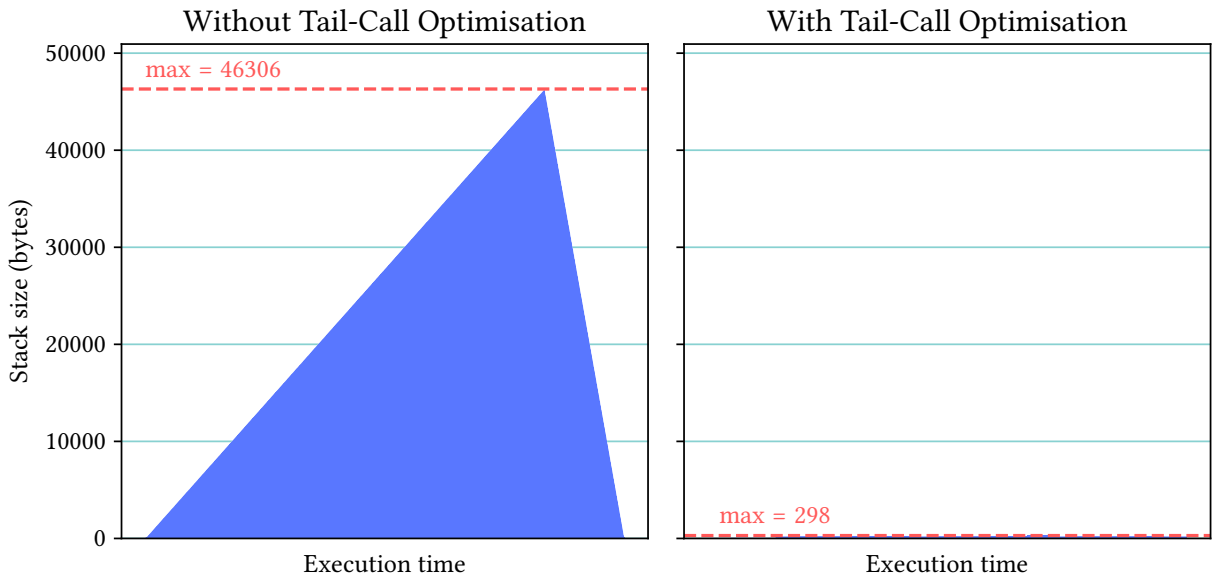


Figure 4.1: Stack usage for calling `sum(500, 0)` (see Listing 4.1)

When testing with large n, the non-optimised version quickly runs out of memory space, and throws an exception. In contrast, the optimised version has no memory constraint on how many iterations can be run. It successfully runs 1 000 000 levels of recursion without using any more memory than for small n; even GCC fails to run that many.

4.3.3 Optimised Stack Allocation Policy

The stack allocation policy that I implemented was successful in reducing the amount of stack memory used.

From my test programs, the largest reduction in memory use was 69.73 % compared to the unoptimised program. The average improvement was 50.28 %.

Figure 4.2 shows the impact that this optimisation had on the different test programs. The full-height bars represent the stack usage of the unoptimised program, which we measure the optimised program against. The darker bars show the stack usage of the optimised program as a percentage of the original stack usage. Shorter bars represent a greater improvement (less memory is being used).

For programs that benefit from tail-call optimisation, I measured the effect of this optimisation on both the optimised and unoptimised versions. I did this because tail-call optimisation also affects how much memory is used, so it may have an impact on the effectiveness of this optimisation.

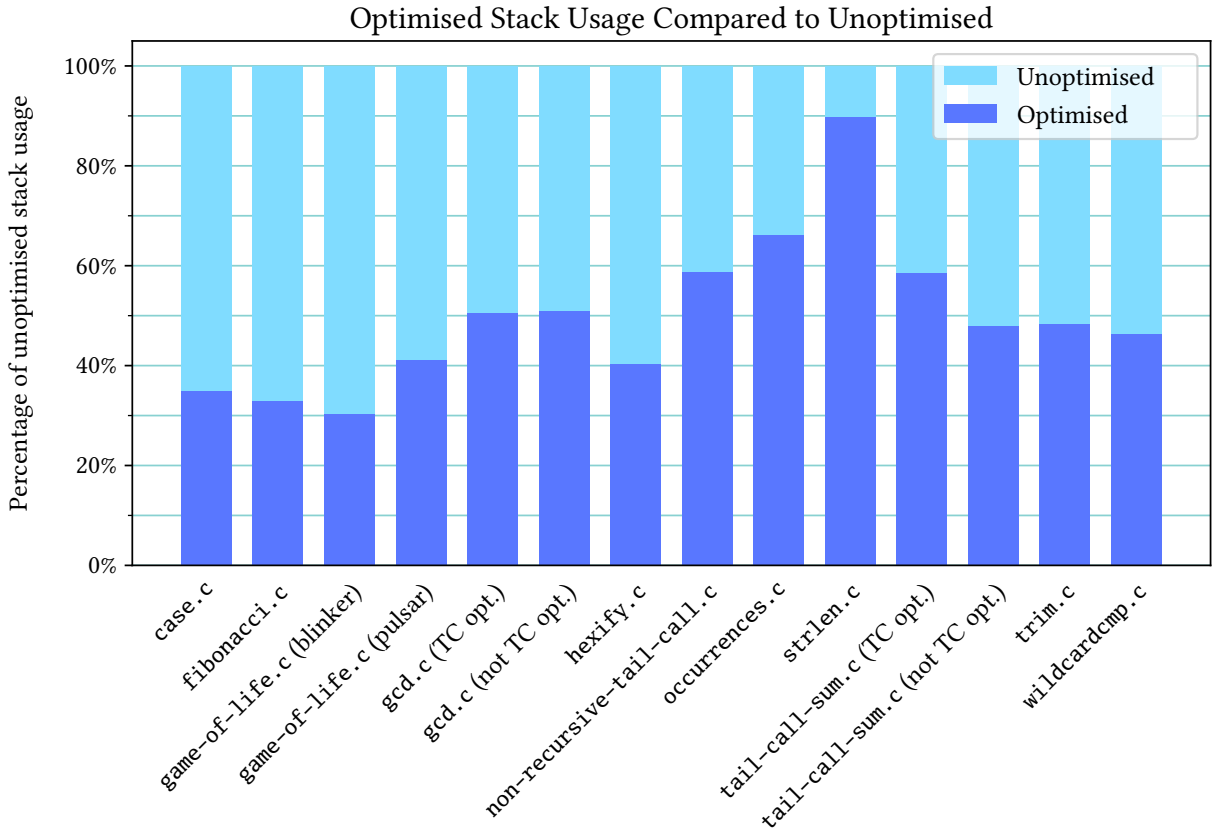


Figure 4.2: Comparing optimised stack usage to unoptimised stack usage. Shorter bars represent greater improvement.

One of the main factors influencing the amount of improvement is the number of temporary variables generated. The more temporary variables generated, the larger each stack frame will be in the unoptimised version, and the more scope there is for the compiler to find non-clashing variables to overlap. Because temporary variables are generated locally for each instruction, the majority of them only have short-range dependencies. Only the variables that correspond to user variables have longer-range dependencies. Therefore the temporary variables offer the compiler more options of independent variables.

The result of this is that as a function increases in its number of operations, the number of temporary variables increases, and so does the scope for optimisation that the compiler is able to exploit.

We can see the effect of this directly when we compare the stack usage of the unoptimised and optimised versions of the same program. Figure 4.3 shows the size of the stack over the execution of a test program that converts strings to upper, lower, or camel case. Since the main stack allocations and deallocations occur on function calls and returns respectively, each spike on the plot corresponds to a function call. We can use this to figure out which parts of the plot correspond to which part of the source program.

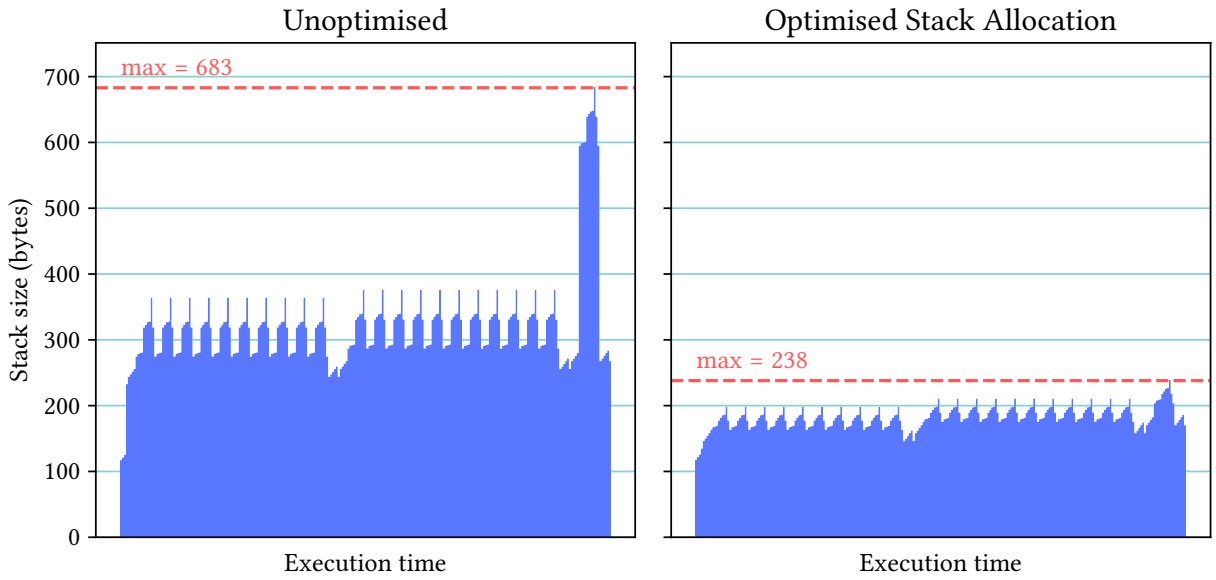


Figure 4.3: Comparing stack usage for `case.c`.

The program in turn calls `case_upper()`, `case_lower()`, and `case_camel()`, which corresponds to the three distinct sections of the plot.

`case_upper()` and `case_lower()` each make repeated calls to `toupper()` and `tolower()`, which corresponds to the many short spikes on the plot (one for each character in the string). Other than a **for** loop, they do not contain many operations, and therefore not many temporary variables are generated.

In contrast, `case_camel()` performs many more operations iteratively in the body of the function. Listing 4.3 shows an extract of its body code. Even in this short section, more temporary variables are created than in the entire body of `case_upper()`. This results in the large spike at the end of Figure 4.3.

```
for (char *s = str; *s; s++) {
    *s = toupper(*s);
}
return str;
```

Listing 4.2: The entire body of `case_upper()`.

```
while (*r && !CASE_IS_SEP(*r)) {
    *w = *r;
    w++;
    r++;
}
```

Listing 4.3: A short section of the body of `case_camel()`.

Due to the differences in temporary variables described above, the compiler is able to optimise `case_camel()` much more than the other functions. This parallels the fact that `case_camel()` had the largest stack frame initially.

Another area that this optimisation has a large impact is for recursive functions. Since this optimisation reduces the size of each stack frame, we will see a large improvement when we have lots of recursive stack frames. Figure 4.4 shows the size of the stack over the execution of calculating the Fibonacci numbers recursively. In this instance, the optimised stack allocation policy reduced the stack size of the program by 67.20 %.

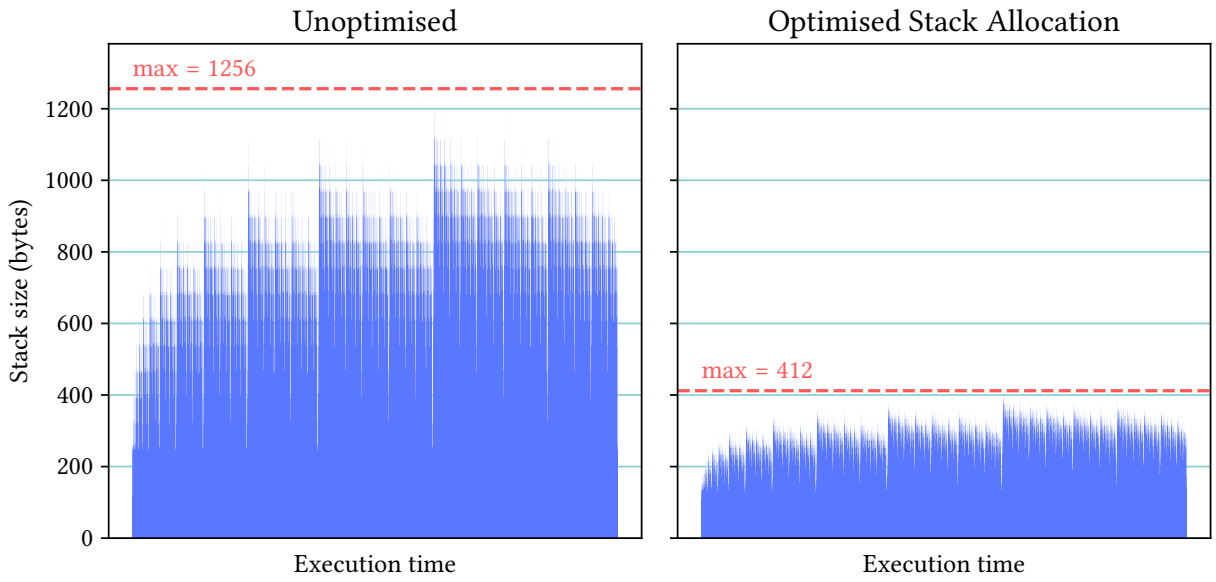


Figure 4.4: Comparing stack usage for `fibonacci.c`.

4.4 Summary

The main objective of this project was to produce a compiler that generated correct WebAssembly binary code. Through the range of testing described above, I have shown that this objective was achieved, with the definition of correctness being that the generated program behaves in the same was as when compiled with GCC.

The objective of adding optimisations is to improve the performance of the compiled programs, while maintaining the semantic meaning of the program. The correctness of my optimisations was verified with the same test suite as was used to test the unoptimised compiler's correctness.

In the previous sections, we have seen measurable evidence that the optimisations did improve performance. Therefore the optimisations were a success.

5 Conclusions

Word budget: ~500–600 words

Likely short, may well refer back to the introduction. Reflection on lessons learned, anything I'd have done differently if starting again with what I know now.
First paragraph should reiterate what the project was about.
Summarise how my evaluation answered the questions this project was asking
Can briefly outline any ideas for further work

5.1 Project Summary

5.2 Lessons Learned

5.3 Further Work

- implement more of stdlib, eg. malloc() and free()

References:

- Relooper algorithm: [4]
- WebAssembly spec: [1]
- C grammar, Microsoft page: [20]
- Avoiding the dangling else ambiguity in LR parsers (Wiki): [**avoiding-dangling-else-ambiguity-wiki**]. Wiki page references [22] ("A Final Solution to the Dangling Else of ALGOL 60 and Related Languages")
- C reference manual book: [19]
- CLRS algorithms textbook, for interval trees: [26]
- LALRPOP tutorial/docs: [14]
- WebAssembly memory guide: [27]
- Addressing Wasm memory: [28]

Bibliography

[1] WebAssembly Community Group. *WebAssembly Core Specification*. Ed. by Andreas Rossberg. Version 2.0. URL: https://webassembly.github.io/spec/core/index.html (visited on 10/14/2022).

[2] Wikipedia contributors. *LEB128*. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=LEB128&oldid=1141111527 (visited on 02/28/2023).

[3] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019* (2019). URL: https://doi.org/10.1109/IEEESTD.2019.8766229.

[4] Alon Zakai. *Emscripten: An LLVM-to-JavaScript Compiler*. Mozilla, 2013. URL: https://raw.githubusercontent.com/emscripten-core/emscripten/main/docs/paper.pdf.

[5] Alon Zakai (@kripken). *A thread on history and terminology of control flow restructuring in the wasm space*. Twitter thread. Mar. 2021. URL: https://twitter.com/kripken/status/1371925248879308801 (visited on 03/02/2023).

[6] Yuri Iozzelli. "Solving the structured control flow problem once and for all". In: (Apr. 2019). URL: https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2.

[7] The Rust Team. *Rust*. URL: https://www.rust-lang.org/ (visited on 03/02/2023).

[8] *Rust by Example*. URL: https://doc.rust-lang.org/rust-by-example/ (visited on 03/04/2023).

[9] Steve Klabnik, Carol Nichols, and The Rust Community. *The Rust Programming Language*. URL: https://doc.rust-lang.org/stable/book/ (visited on 03/04/2023).

[10] MDN contributors. *JavaScript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 03/04/2023).

[11] OpenJS Foundation and Node.js contributors. *Node.js*. URL: https://nodejs.org/en/ (visited on 03/04/2023).

[12] Python Software Foundation. *Python*. URL: https://www.python.org/ (visited on 03/04/2023).

[13] The Matplotlib development team. *Matplotlib: Visualization with Python*. URL: https://matplotlib.org/ (visited on 03/04/2023).

[14] *LALRPOP Documentation*. URL: https://lalrpop.github.io/lalrpop/index.html (visited on 10/14/2022).

[15] *Git*. URL: https://git-scm.com/ (visited on 03/04/2023).

[16] *GitHub*. URL: https://github.com/ (visited on 03/04/2023).

[17] JetBrains. *Intellij Rust*. URL: https://www.jetbrains.com/rust/ (visited on 03/04/2023).

[18] *The C Preprocessor*. URL: https://gcc.gnu.org/onlinedocs/cpp/ (visited on 02/27/2023).

[19] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. 4th ed. 1995. ISBN: 0-13-326224-3.

[20] Microsoft. *C Language Syntax Summary*. URL: https://learn.microsoft.com/en-us/cpp/c-language/c-language-syntax-summary (visited on 10/25/2022).

[21] Wikipedia contributors. *Dangling else*. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Dangling_else&oldid=1136442147 (visited on 03/11/2023).

[22] Paul W. Abrahams. "A Final Solution to the Dangling Else of ALGOL 60 and Related Languages". In: *Communications of the ACM* 9.9 (Sept. 1966), pp. 679–682. URL: https://doi.org/10.1145/365813.365821.

[23] Paul Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. 2004, pp. 1–2. ISBN: 9780596552817.

[24] Martin Gardner. "The fantastic combinations of John Conway's new solitaire game "life"". Mathematical Games. In: *Scientific American* 223.4 (Oct. 1970), pp. 120–123. URL: https://doi.org/10.1038/scientificamerican1070-120.

[25] Clib authors. *Clib Packages*. Licensed under the MIT License. URL: https://github.com/clibs/clib/wiki/Packages (visited on 11/10/2022).

[26] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. 2009. ISBN: 978-0-262-03384-8.

[27] Brian Sletten. "WebAssembly Memory". In: *WebAssembly: The Definitive Guide*. Dec. 2021. Chap. 4. ISBN: 9781492089841.

[28] Rasmus Andersson. *Introduction to WebAssembly. Addressing Memory*. URL: https://rsms.me/wasm-intro#addressing-memory (visited on 02/21/2023).

Appendix A

Lexer Finite State Machine

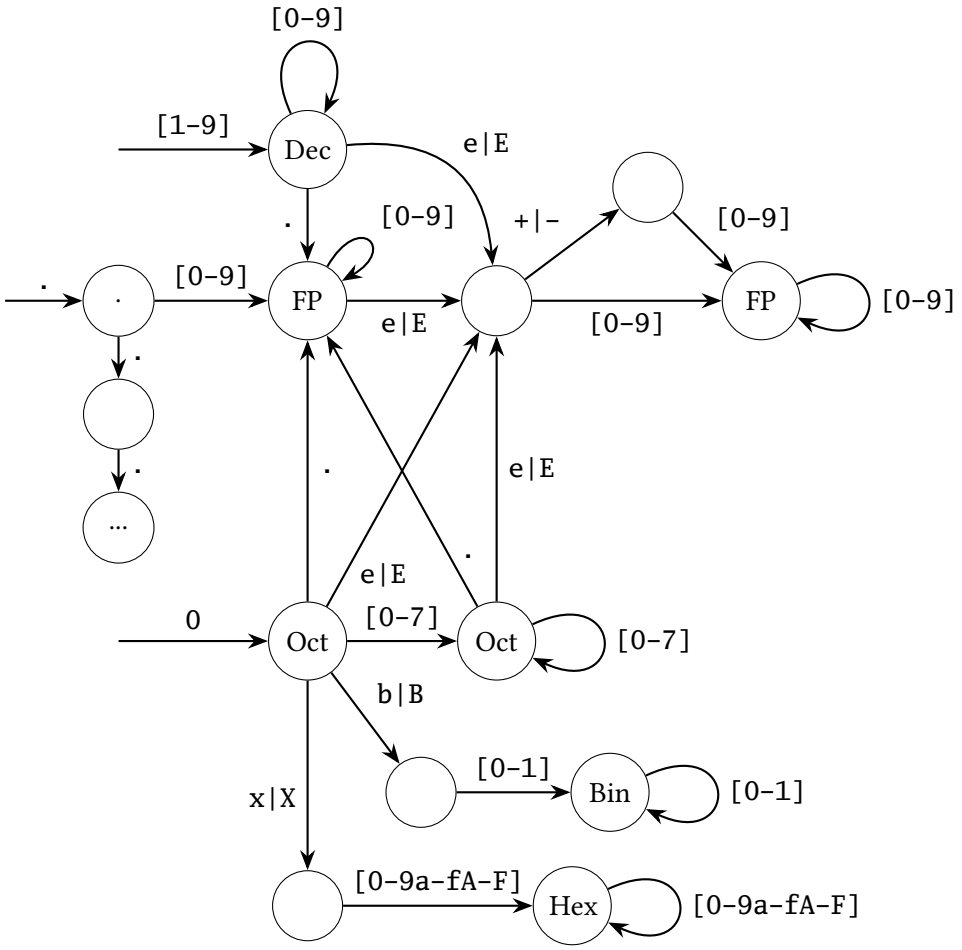


Figure A.1: Finite state machine for lexing number literals.

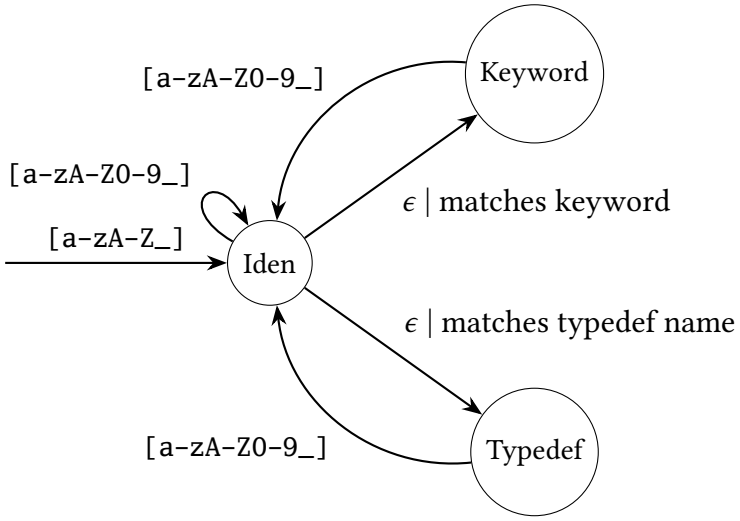


Figure A.2: Finite state machine for lexing identifiers.

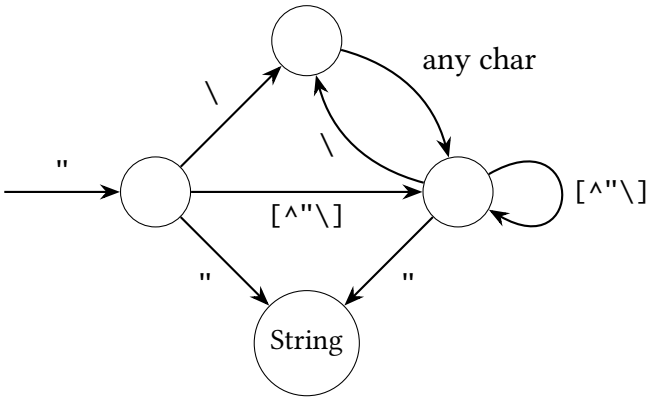


Figure A.3: Finite state machine for lexing string literals.

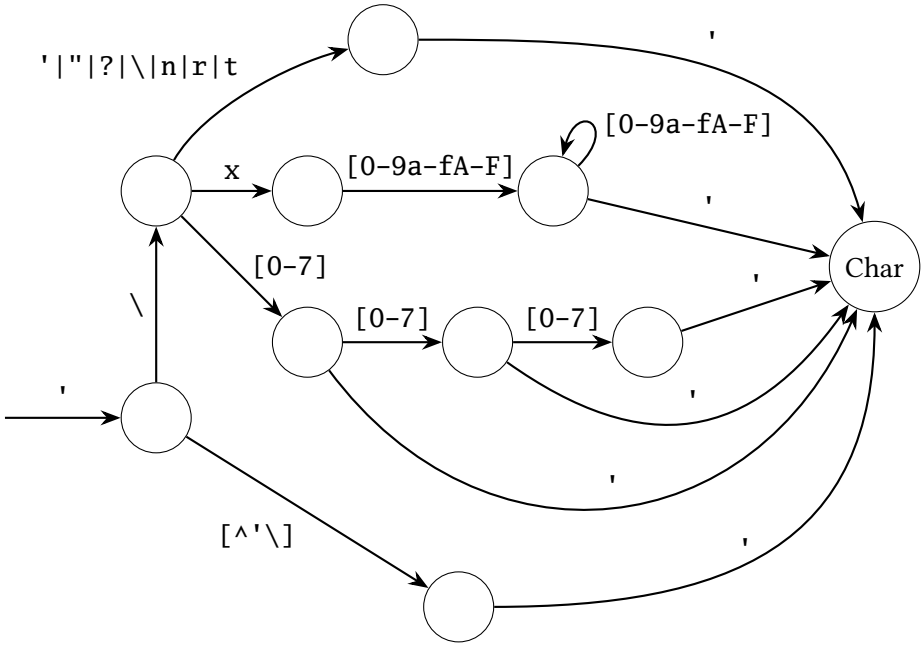


Figure A.4: Finite state machine for lexing character literals.

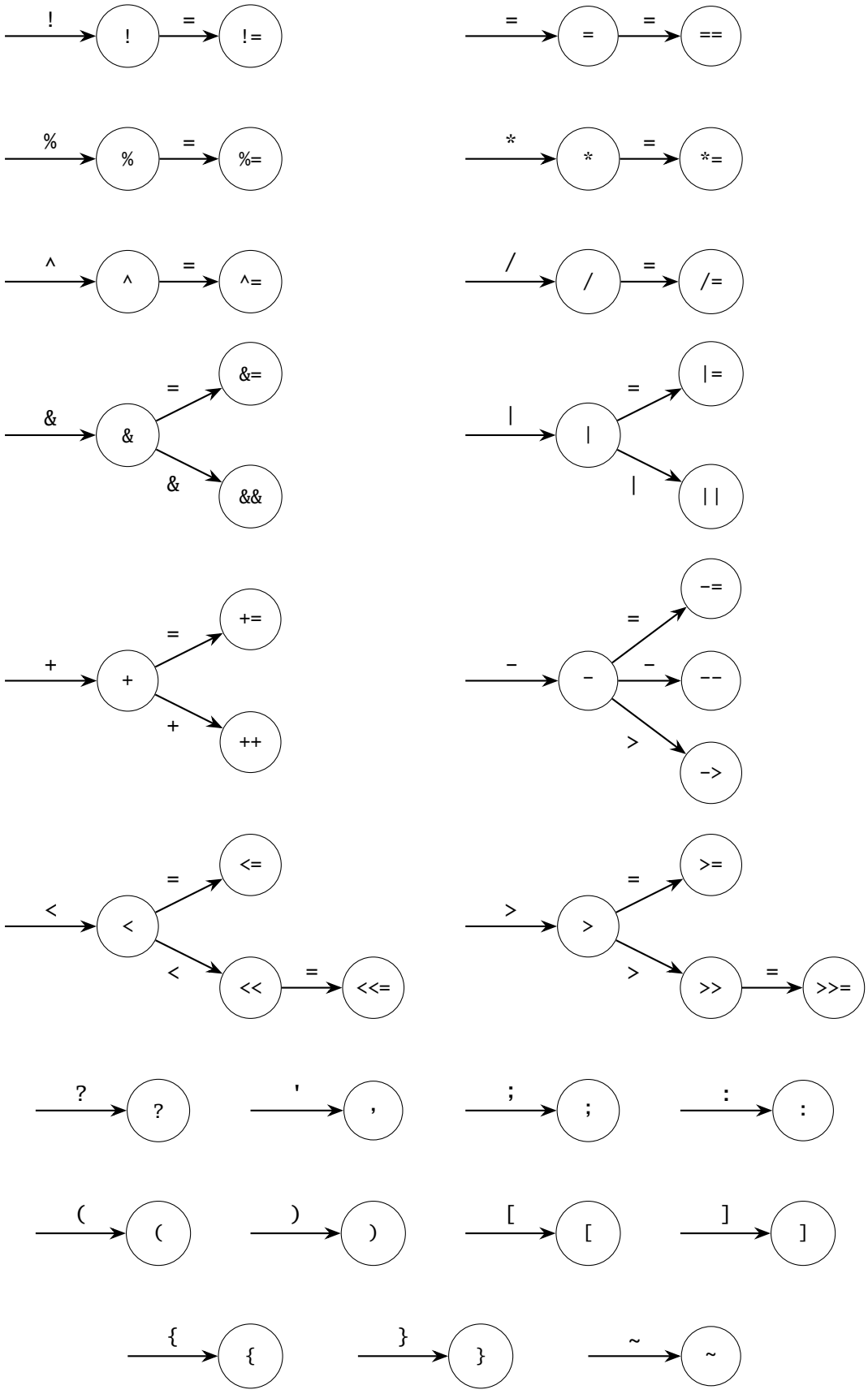


Figure A.5: Finite state machine for lexing operators.

Appendix B

Intermediate Code

`x` and `y` can either be variables or constants, used as operands to instructions. `t` is a destination variable.

`t = x`	Simple Assignment
`t = load from x` `store x to addr y`	Load from and store to memory
`declare var t` `allocate x bytes for var y`	Declare a new variable `t`. Allocate memory for `y` (used for allocating aggregate data structures e.g. arrays).
`reference var x`	A non-executable instruction used internally to mark a variable as live at this program point.
`t = &x`	Address-of operator
`t = ~x`	Bitwise NOT
`t = !x`	Logical NOT
`t = x * y`	Multiplication
`t = x / y`	Division
`t = x % y`	Modulus
`t = x + y`	Addition
`t = x - y`	Subtraction
`t = x << y`	Left-shift
`t = x >> y`	Right-shift (signed-extending for signed `x`, zero-filling for unsigned `x`)
`t = x & y`	Bitwise AND
`t = x | y`	Bitwise OR
`t = x ^ y`	Bitwise XOR
`t = x && y`	Logical AND
`t = x || y`	Logical OR
`t = x < y`	Less-than comparison
`t = x > y`	Greater-than comparison
`t = x <= y`	Less-than or equal comparison
`t = x >= y`	Greater-than or equal comparison
`t = x == y`	Equality comparison
`t = x != y`	Not equal comparison
`t = call f(`p_1`, `p_2`, ...)`	Call function `f` with parameters p_i (either variables or constants)

`tail-call f(p1, p2, ...)`	Call function `f` and return the result from the current function
`return [x]`	Return from the current function. The return value `x` is optional.
`label <l>`	Attach a label to the current program point (immediately before the next instruction).
`br <l>`	Unconditional branch
`br <l> if x == y`	Conditional branch; executed if operands are equal.
`br <l> if x != y`	Conditional branch; executed if operands are not equal.
`t = &<sid>`	Static address of the string literal with id <sid>
`t = (i8 ⟶ i16) x`	Char promotions
`t = (i8 ⟶ u16) x`	
`t = (u8 ⟶ u16) x`	
`t = (u8 ⟶ u16) x`	
`t = (i16 ⟶ i32) x`	Promotions to signed integer
`t = (u16 ⟶ i32) x`	
`t = (i16 ⟶ u32) x`	Promotions to unsigned integer
`t = (u16 ⟶ u32) x`	
`t = (i32 ⟶ u32) x`	
`t = (i32 ⟶ i64) x`	Promotions to signed long
`t = (u32 ⟶ i64) x`	
`t = (i32 ⟶ u64) x`	Promotions to unsigned long
`t = (u32 ⟶ u64) x`	
`t = (i64 ⟶ u64) x`	
`t = (u32 ⟶ f32) x`	Integer to float conversions
`t = (i32 ⟶ f32) x`	
`t = (u64 ⟶ f32) x`	
`t = (i64 ⟶ f32) x`	
`t = (u32 ⟶ f64) x`	Integer to double conversions
`t = (i32 ⟶ f64) x`	
`t = (u64 ⟶ f64) x`	
`t = (i64 ⟶ f64) x`	
`t = (f32 ⟶ f64) x`	Float to double promotion
`t = (f64 ⟶ i32) x`	Double to int conversion
`t = (i32 ⟶ i8) x`	Integer truncation
`t = (u32 ⟶ i8) x`	
`t = (i64 ⟶ i8) x`	
`t = (u64 ⟶ i8) x`	
`t = (i32 ⟶ u8) x`	
`t = (u32 ⟶ u8) x`	
`t = (i64 ⟶ u8) x`	

`t = (u64 ⟶ u8) x`
`t = (i64 ⟶ i32) x`
`t = (u64 ⟶ i32) x`	

`t = (u32 ⟶ *) x`
`t = (i32 ⟶ *) x`
`t = (* ⟶ i32) x`
	Conversions between integer and pointer

`nop`	No-op

`break <loop_block_id>`
`continue <loop_block_id>`
`end handled <multiple_block_id>`
	Control-flow instructions inserted by the Relooper algorithm as it processes branch instructions.

`if x == y {} else {}` `if x != y {} else {}`	Conditional control flow instructions with nested instructions for each branch. These are only inserted by the Relooper algorithm, to replace a conditional branch with conditionally setting the label variable and then branching.

Appendix C
Project Proposal

The original project proposal is included on the following pages.

Part II Project Proposal: C to WebAssembly Compiler

Martin Walls

October 2022

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 Rust, which is memory safe and performant, and has lexer/parser generators I can use.

To test and evaluate the compiler, I will write small benchmark programs that individually test each of the features and optimisations I add. For example, I will use the Fibonacci program to test recursion. I will also test it with Conway's Game of Life, as an example of a larger program, to test and evaluate the functionality of the compiler as a whole.

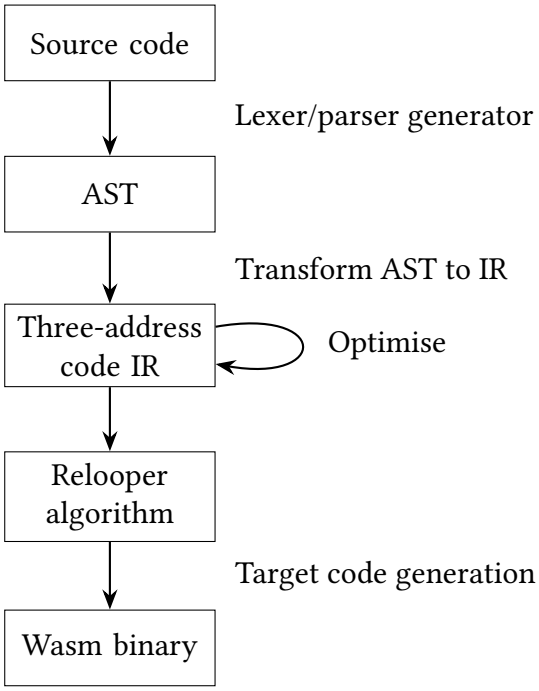
I will use a three-address code style of intermediate representation, because this lends itself to perform optimisations more easily. For example it's easier to see the control flow in three-address code

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.

C allows unstructured control flow (e.g. `goto`), whereas WebAssembly only supports structured control flow. Therefore I will need a step in the compiler to transform unstructured to structured control flow. One algorithm to do this is the Relooper algorithm, which was originally implemented as part of Emscripten, a LLVM to JavaScript compiler[1].

Compiler pipeline overview



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've briefly looked at Rust over the summer, but haven't written anything other than simple programs in it.

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.

[1] `https://github.com/emscripten-core/emscripten/blob/main/docs/paper.pdf`

I have briefly researched lexer and parser generators 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 uses the Relooper algorithm to transform unstructured to structured control flow.
- The program generates WebAssembly binary code from the intermediate representation.
- The compiler generates binary code that produces the same output as the source program.

Optimisations

First I will implement some simple optimisations, before adding some more complicated ones.

One of the simple optimisations I will implement is peephole optimisation, which is where we 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 another quite simple 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.

A more complicated optimisation to add will be tail-call optimisation, which removes unnecessary stack frames when a function call is the last statement of a function.

Other harder optimisations are left as extensions to the project.

Extensions

Extensions to this project will be further optimisations. These optimisations are more complicated and will involve more analysis of the code.

One optimisation would be dead-code elimination, which looks through the code for any variables that are written to but never read. Code that writes to these variables is removed, saving processing power and space.

Another optimisation would be unreachable-code elimination, where we perform analysis to find blocks of code that can never be executed, and removing them. This will involve control flow analysis to determine the possible routes the program can take.

Evaluation

To test and evaluate the compiler, I will use it to compile a variety of different programs. Some of these will be small programs I will write to specifically test the features and optimisations of the compiler individually. I will also write a larger test program to evaluate the compiler as a whole.

In addition, I will use some pre-existing benchmark programs to give a wider range of tests. For example, cBench is a set of programs for benchmarking optimisations, which I could choose appropriate programs from. The source for cBench is no longer available online, but my supervisor is able to give me a copy of them.

For each of these, I will verify that the generated WebAssembly code produces the same output as the source program when run.

To evaluate the impact of the optimisations, I will run the compiler once with optimisations enabled and once with them disabled, on the same set of programs. 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. I will research the WebAssembly instruction set. I will also write test C programs for Fibonacci and Conway's Game of Life. To help with my WebAssembly research, I will implement the same Fibonacci program in WebAssembly by hand. ***Milestone deliverable**: I will write a short LaTeX document explaining the WebAssembly instruction set, from the research I do. C programs of Fibonacci and Conway's Game of Life, and a WebAssembly implementation of Fibonacci.*

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. ***Milestone deliverable**: Lexer and parser generator inputs. The compiler will be able to generate an abstract syntax tree (AST) representation from a source 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 from a source program.*
4	**25th Nov - 9th Dec**	Researching and implementing the Relooper algorithm. ***Milestone deliverable**: The compiler will be able to transform unstructured control flow into structured control flow using the Relooper algorithm. I will also write a short LaTeX document describing the algorithm.*
5	**9th - 23rd 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 a source program. The generated WebAssembly will be able to be run in a web browser.*
		Two weeks off over Christmas
6	**6th - 20th Jan**	*(I'll be more busy during the first week of this with some extracurricular events before term.)* Slack time to finish main implementation if necessary. Implement some peephole optimisations (how many I do here depends on how much of the slack time I need). ***Milestone deliverable**: The basic compiler pipeline will be complete. Some peephole optimisations will be implemented.*
7	**20th Jan - 3rd Feb**	Write progress report. Continue implementing optimisations, in particular implementing tail-call optimisation. ***Milestone deliverable**: Completed progress report. (Deadline 03/02)*
8	**3rd - 17th Feb**	*(I'll be more busy here with extra-curricular events.)* Slack time to finish main optimisations if necessary. If time allows, work on extension optimisations.

		***Milestone deliverable**: The compiler will be able to generate target code with optimisations applied. Evidence to show the impact of the optimisations.*
9	**17th Feb - 3rd Mar**	Evaluate the compiled WebAssembly using a variety of programs (as described above), including correctness and impact of optimisations. Write these evaluations into a draft evaluation chapter. ***Milestone deliverable**: Draft evaluation chapter.*
10	**3rd - 17th Mar**	Write introduction and preparation chapters. ***Milestone deliverable**: Introduction and preparation chapters.*
11	**17th - 31st Mar**	Write implementation chapter. ***Milestone deliverable**: Implementation chapter.*
12	**31st Mar - 14th Apr**	Write conclusions chapter and finish evaluations chapter. ***Milestone deliverable**: Evaluations and conclusions chapter. First draft of complete dissertation.*
13	**14th - 28th Apr**	Adjust dissertation based on feedback. ***Milestone deliverable**: Finished dissertation.*
14	**28th Apr - 12 May**	Slack time in two weeks up to formal deadline, to make any final changes. ***Milestone deliverable**: Final dissertation submitted. (Deadline 12/05)*

Resource declaration

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.

My laptop specifications are:

- Lenovo IdeaPad S540
- CPU: AMD Ryzen 7 3750H
- 8GB RAM
- 2TB SSD
- OS: Fedora 35

I will use Git for version control and will regularly push to an online Git repository on GitHub. 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.