Martin Walls

C to WebAssembly Compiler

Computer Science Tripos: Part II

Churchill College

University of Cambridge

22nd April, 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. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: MWalls

Date: 22nd April, 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:	17 722[2]
Project Originator:	Timothy M. Jones
Supervisor:	Timothy M. Jones

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

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 optimisations, such as tail-call optimisation, I was able to implement an additional extension optimisation, which was successful in significantly reducing the memory usage of programs.

Special Difficulties

None.

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

Contents

1
Introduction

1.1 Background and Motivation

In modern society, applications are increasingly 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 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. The WebAssembly specification is designed to have “fast, safe, and portable semantics” and an “efficient and portable representation” [1].

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

The representation is compact and modular, allowing it to be efficiently transmitted over the Internet without slowing page loads. This also includes being streamable and parallelisable, i.e. it can be decoded while still being received.

This project creates a compiler from C to WebAssembly. I chose C as a source language because it is reasonably low-level, which allowed me to have time to work on compiler optimisations rather than just implementing language features. In particular, C has manual memory management, therefore I did not have to implement a garbage collector or similar.

1.2 Survey of Related Work

Many other compilers to WebAssembly exist, covering a large number of source languages [2]. Notably, Emscripten is a compiler to WebAssembly and JavaScript using LLVM [3, 4]. This primarily targets C and C++, however can also be used for any other language using LLVM.

Cheerp is an alternative C/C++ to WebAssembly compiler, which can produce a combination of WebAssembly and JavaScript, and prides itself on generating very optimised code [5].

Compilers to WebAssembly exist for other mainstream languages such as Java, C#, Python, and even JavaScript [6, 7, 8, 9]. It is an area of active development; many more WebAssembly compilers are in progress for other languages [2].

My project covers a subset of the existing C to WebAssembly compilers; I only support a subset of the language features, and focus on correctness over optimality. I have implemented some optimisations that significantly improve the performance of many programs, but do not optimise every possible edge case (as many industry compilers do).

2

Preparation

In this chapter I describe my research into the WebAssembly architecture and the Relooper algorithm, highlight the main features of Rust, the programming language I learnt for this project, and my rationale for using it.

Furthermore, I discuss the requirements of the project, and describe the software engineering approach I employed, before outlining my starting point and previous experience.

2.1 WebAssembly

Initially, I extensively researched the WebAssembly specification to gain a deep understanding of the target binary code [1]. The following sections provide an overview of the instruction set architecture.

2.1.1 Primitive Values

Table 2.1 below outlines the primitive values supported by WebAssembly.

The same representation is used for both signed and unsigned integers; operations determine how to interpret their operands. Integers also store booleans and memory addresses[1]. The LEB128 variable-length encoding scheme is used; Listing 2.1 describes how to encode an unsigned integer [10]. The algorithm is almost identical for signed integers; the only difference is that we sign-extend rather than zero-extend. Some instructions have both signed and unsigned variants, because the decoder needs to know whether to interpret each operand as a two's complement integer or not.

Floating-point literals are encoded using the IEEE 754-2019 standard [11]. This is the same standard used by many programming languages, including Rust, so the byte representation does not need to be converted between the compiler and the WebAssembly binary.

WebAssembly also provides vector types and reference types (mainly used for function references). This project does not have a need for either of these types; the C language has no 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.

[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: Summary of the primitive types supported by WebAssembly. The 'constructor' is the name of the type in code and in the specification.

```
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 [10]. The function takes an integer n and returns the byte sequence to represent it in the compiled program.

2.1.2 Instructions

WebAssembly is a stack-based architecture. Instructions take their operands from the stack, and push their result back to it.

All arithmetic instructions specify the type of value that they expect. A module fails to instantiate if the types do not match. Instructions where the signedness of a number matters have signed and unsigned variants, such as the less-than instructions `i32.lt_u` and `i32.lt_s`.

WebAssembly only supports structured control flow, in contrast to the unstructured control flow found in most instruction sets with 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. Note that **`loop`** does not loop back to the start implicitly; an explicit `br` instruction is required. An **`if`** block conditionally executes depending on the value on top of the stack, and may optionally have an **`else`** block. A `br` instruction referring to an **`if`** block 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' as in other languages; there are only modules that export functions to the instantiator (the runtime environment that instantiates the WebAssembly module).

Modules are split up into sections, described in Table 2.2. Each section is preceded by a header containing the section ID and its size in bytes.

Section	Description
Type	Defines all function types used, including imported functions.
Import	Defines everything imported to the module from the runtime environment: functions, memories, tables, and global variables.
Function	Maps functions to their types.
Table	Stores size information about tables. A table is an array of function pointers, used to call functions indirectly. A separate data structure is needed because function addresses are hidden from the program, keeping the execution sandboxed.
Memory	Specifies the size limits of each linear memory of the program[1] in units of the WebAssembly page size (64 KiB)
Global	Defines global variables, including an expression to initialise them.
Export	Similar to the import section, defines everything exported from the module to the runtime environment.
Start	Optionally specifies a function that should run automatically when the module is initialised.
Element	Statically initialises tables with function addresses. This can either be done automatically when the module is loaded, or explicitly with a `table.init` instruction.
Data count	Optional section used for validation, that specifies how many data segments the data section contains. This allows a validator to use a single pass for static analysis, since it allows them to check the validity of instructions referencing data indexes.
Code	Contains the instructions for each function body. Each function begins by declaring any local variables, followed by the body code.
Data	Used to statically initialise the contents of memory, either automatically or explicitly (like the element section).

Table 2.2: Sections of a WebAssembly module.

2.2 The Relooper Algorithm

C allows arbitrary control flow with **`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.

The most naive solution is 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. This is very inefficient.

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

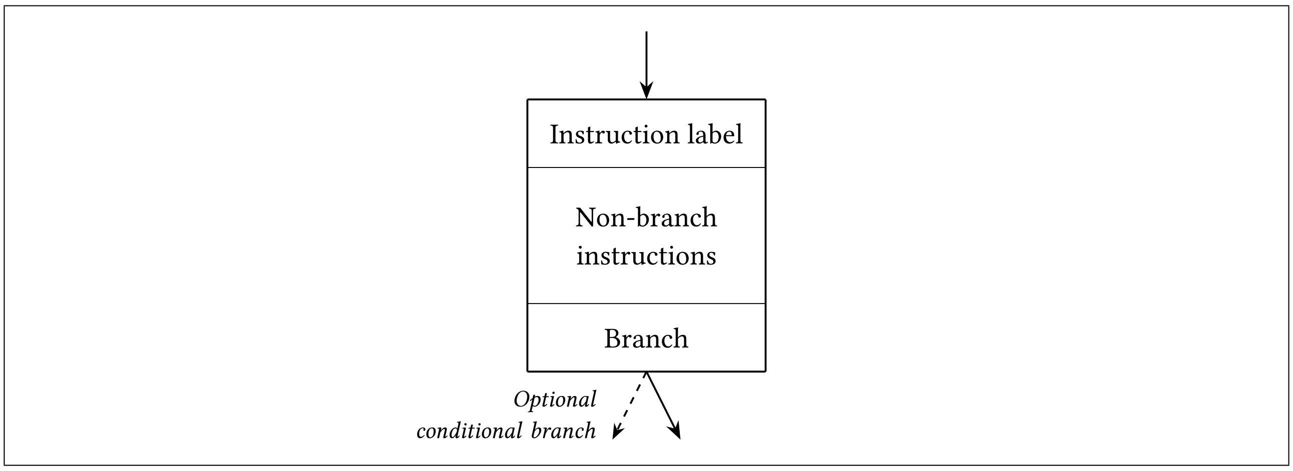


Figure 2.1: Structure of labels (Relooper algorithm input blocks).

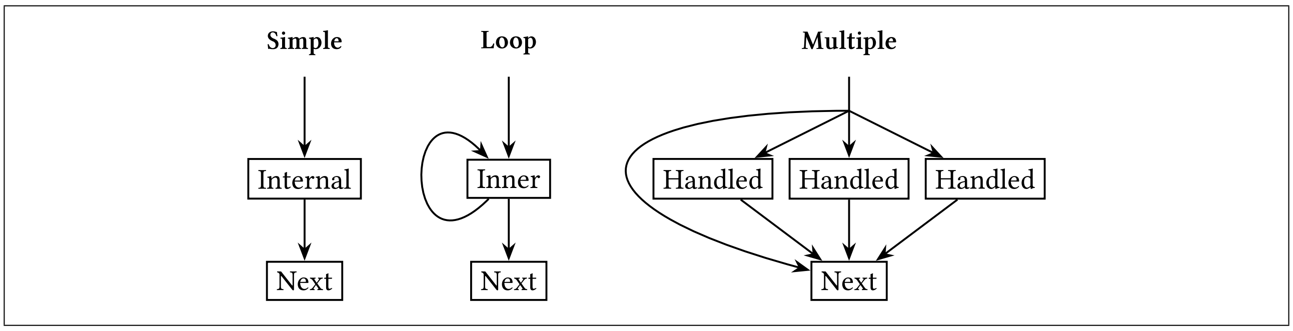


Figure 2.2: Structured blocks, generated by the Relooper algorithm.

The first algorithm that solved this problem in the context of compiling to WebAssembly (and JavaScript, previously) was the Relooper algorithm, introduced by Emscripten [12]. In current WebAssembly compilers, there are three general methods used to convert to structured control flow: Emscripten's Relooper algorithm, LLVM's CFGStackify, and Cheerp's Stackifier [13, 14]. All of these implementations, including the modern implementation of Relooper, are more optimised than the original Relooper algorithm, hence more complex. The original Relooper algorithm is a greedy algorithm, is well described in the paper, and is most suitable to implement in the scope of this project.

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). No labels or branch instructions are permitted in the middle of the block. The branch instruction may either be an unconditional or a conditional branch (`if (...) goto x else goto y`).

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

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, with the actual program instructions. *Loop* blocks contain an *inner* block, which contains all the labels that can reach the start of the loop again along some execution path. *Multiple* blocks represent conditional execution, along one of several paths. They contain *handled* blocks to which execution can pass when entering the block. All three blocks have a *next* block, where execution continues.

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 entries, Algorithm 2.1 creates a block:

$\boxed{1}$ Calculate the reachability of each label.

$\boxed{2}$ 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.

$\boxed{3}$ 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.

$\boxed{4}$ 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 $\boxed{5}$.
- 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 did not create handled blocks from, plus any other possible branch targets out of the handled blocks.

$\boxed{5}$ If Step $\boxed{4}$ fails, create a loop block in the same way as Step $\boxed{3}$.

Algorithm 2.1: The Relooper Algorithm.

The Emscripten paper outlines a proof that this greedy approach will always succeed [12, 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 $\boxed{5}$, 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 $\boxed{4}$, 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 unique identifier (ID) of the block they act on, because blocks may be nested. 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.

In order 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 generates some overhead of unnecessary instructions, since most of the time the label variable is set, it will not be checked. However, later stages of my compiler pipeline will optimise this away.

2.3 Rust

I learnt the Rust programming language for this project [15]. I chose Rust because it is performant and memory efficient. It has a rich type system with pattern matching that is well suited to writing a compiler. It uses the concept of a borrow checker rather than a garbage collector or similar memory management system, which eliminates run-time overhead whilst guaranteeing memory safety at compile time.

I made use of Rust's excellent online documentation to become familiar with the language [16, 17]. The borrow checker was the main new feature to be learnt, with 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 or explicit deallocation found in other languages. When using a value, it can either be *moved* or *borrowed*. Moving a value to another variable makes the new variable the owner of the value, and invalidates 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. The Rust compiler enforces these constraints at compile time.

2.4 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.

2.5 Software Engineering Methodology

I structured my project using the incremental model, which builds upon the waterfall model [18, 19]. The project is broken down into 'increments', which are individual sections of functionality. Each increment is developed consecutively, using the waterfall model to design, implement, and test each section before moving on to the next one.

Each stage of the compiler pipeline is a separate increment. Subsequent stages of the project each depend on the deliverable from the previous one, making this model a natural fit, because it ensures each stage works correctly before continuing.

I used Git as my version control system [20]. I regularly committed my code in small, revertible chunks, and pushed the repository to GitHub to maintain an up-to-date off-site backup [21]. In the later stages of the project, I used pre-commit hooks to run automated tests on the project every time I committed, ensuring no inadvertent bugs were introduced.

2.5.1 Testing

To facilitate testing, I wrote a suite of test programs designed to use different language constructs. After completing each stage of the compiler pipeline, I carefully examined the output of compiling each test program, verifying its correctness.

Once the compiler could generate an AST, I wrote a function to reconstruct the source code from the AST. I compared this output to the original source to confirm the program semantics were maintained.

When the entire pipeline was complete, I used GCC (the GNU Compiler Collection) as a reference compiler; I compiled my test programs with both my compiler and GCC, and verified that the resulting binaries produced the same output [22].

2.6 Starting Point

The *Compiler Construction* course of the Tripos was the starting point for my knowledge of compilers. The Part II course *Optimising Compilers* was useful in extending my project with optimisations, though some of the optimisations I implemented went beyond the course.

I had familiarity with C from the *Programming in C and C++* course, and 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.

2.7 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, using Node.js, which is necessary to interface with the WebAssembly binary [23, 24]. Additional development scripts, for automated testing and profiling, were written in Python, because of ease of use and my familiarity with libraries such as Matplotlib [25, 26].

I used the LALRPOP parser generator library to generate parsing code from the abstract grammar of C [27]. Other parser generator libraries are available for Rust: notably, `nom` and `pest` [28, 29]. Originally `nom` was designed for parsing binary file formats, making it less suitable for parsing source code. Another popular choice is `pest`, however it separates the formal grammar and AST generation, making the code more complicated. LALRPOP provides an intuitive and powerful approach, where each grammar rule contains both the grammar specification and code to generate the corresponding AST node.

I used Git for version control with GitHub as an off-site backup, and the CLion IDE as my development environment using the Intellij Rust plugin [20, 21, 30, 31]. Together they provides excellent support for Rust and C, including code analysis and debugging.

2.8 Licensing

LALRPOP is MIT licensed. I have decided to also use the MIT license for my code.

3

Implementation

This chapter describes the implementation of each stage of the compiler. Section 3.1 gives an overview of the project structure, and subsequent sections explain in more detail.

Code snippets and figures are presented to supplement my explanations. Where code is given, it is simplified to highlight the implementation details being explained. This includes removing boilerplate code, error handling, and other features that are necessary for implementation but unhelpful for clarity.

3.1 System Architecture

Figure 3.1 describes the high-level structure of the project.

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

3.2 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 to convert the tokens emitted by the lexer into an AST [27].

3.2.1 Preprocessor

I used the GNU C preprocessor (cpp) to handle any preprocessor directives in the source code, for example macro definitions [32]. 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.

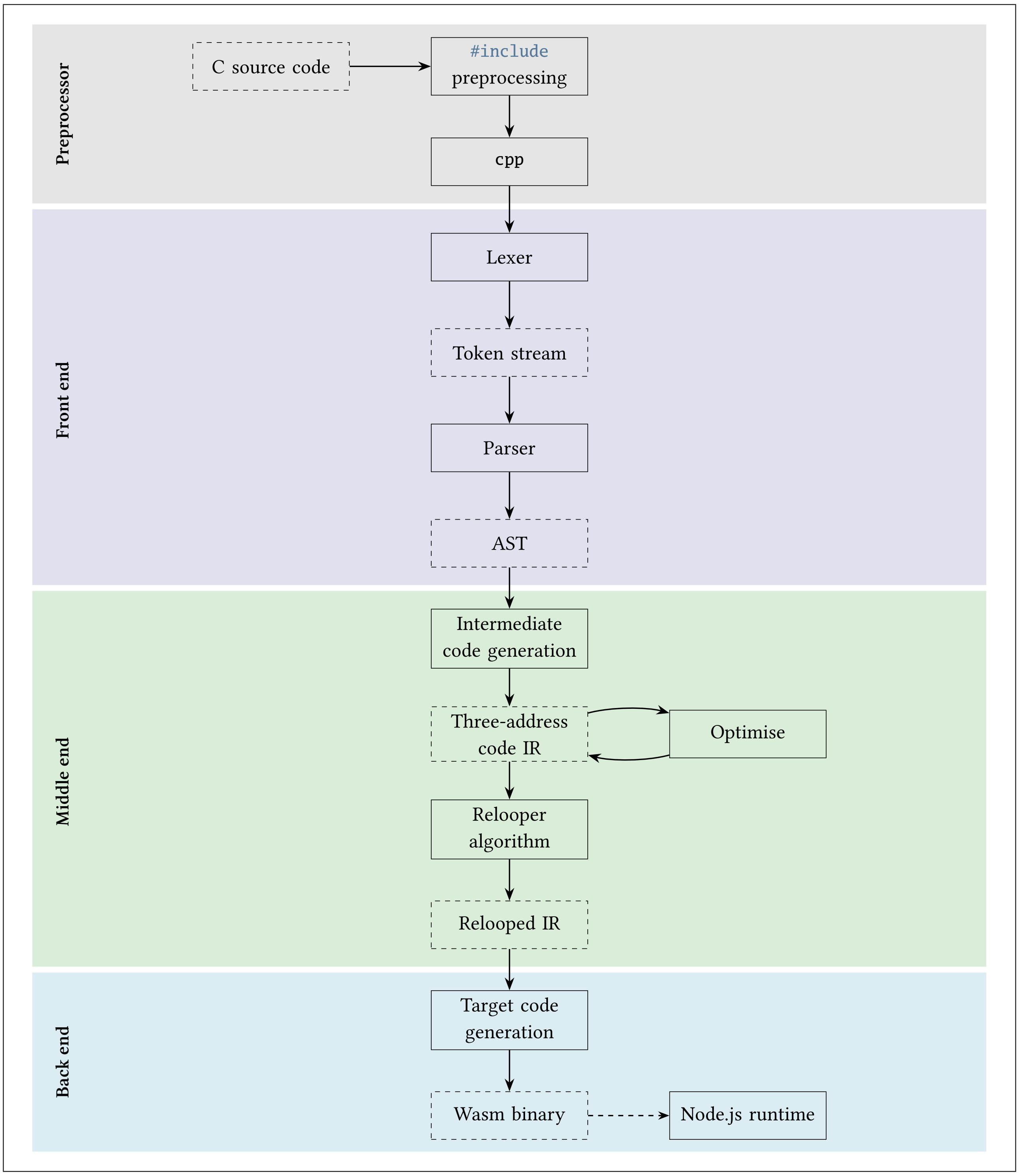


Figure 3.1: Project structure, highlighting the preprocessor, front end, middle end, and back end. Each solid box represents a module of the project, transforming the input data representation into the output representation. The data representations are shown as dashed boxes. Where no new data representation is shown after a module, the output format is the same as the input format (i.e. in the preprocessor).

3.2.2 Lexer

The grammar of the C language is mostly context-free, however the presence of `typedef` definitions makes it context-sensitive [33, sec. 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 already defined at the current program point.

The lexer is implemented as a finite state machine (FSM). Figure 3.2 highlight portions of the machine; the remaining state transition diagrams can be found in Appendix B. The FSM 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 (as the next token in the token stream). 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 lexing error. (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(a) shows the FSM for lexing number literals. This handles all the different number formats that C supports: decimal, binary, octal, hexadecimal, and floating point.

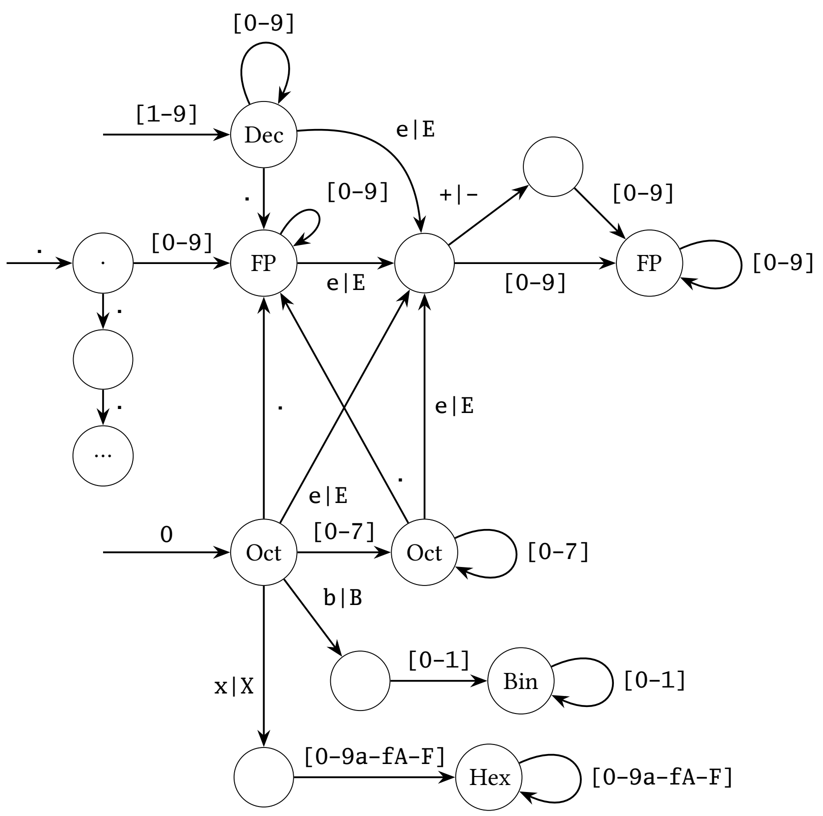
Figure 3.2(b) shows the FSM for lexing identifiers and **`typedef`** names. This is where the ambiguity discussed above is handled. Every time another character of an identifier is consumed, the name is compared to language keywords and **`typedef`** names we have encountered so far[2]. If a match is found, the machine transitions to the corresponding state, along an ϵ transition (no input is consumed). The lexer stores each **`typedef`** name that is declared so that it can emit the correct token for future names.

3.2.3 Parser

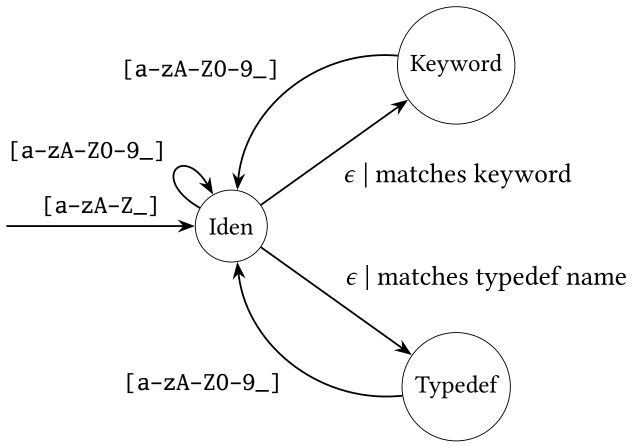
I used the LALRPOP parser generator to generate parsing code from the input grammar that I wrote [27]. It generates an AST for which I defined the structure. Microsoft's C Language Syntax Summary and C: A Reference Manual were very useful references to ensure I captured the subtleties of C's syntax when writing my grammar [33, 34]. My grammar is able to parse all core features of the C language, whilst omitting some of the recent 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 does not 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.

[1]The brackets will be ignored.

[2]Keywords are matched with higher priority.



(a) FSM for lexing number literals.



(b) FSM for lexing identifiers.

Figure 3.2: Sections of the lexer finite state machine (FSM). Named states represent a valid token, and blank states are invalid. Only valid transitions are shown, using regular expressions; if no transition exists for an input character, the end of the current token has been reached. Transitions without a prior state are the initial transitions for the first input character. The token is emitted if the machine stops in a valid (named) state; otherwise a lexing error is raised. In a slight abuse of regular expression notation, '`.`' represents a literal full stop character. No input is consumed along an ϵ transition; the transition is taken if the condition is true.
Tokens: *Dec*: decimal literal; *FP*: floating point literal; *Oct*: octal literal; *Bin*: binary literal; *Hex*: hexadecimal literal; *Iden*: identifier; *Keyword*: a C keyword; *Typedef*: an identifier defined as a type name.
The states leading to the ellipsis token are shown for completeness since they share the starting dot state, even though the token is not a number literal.

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

Listing 3.1: Ambigious **if**/**else** grammar. Terminals are represented in double quotes, and non-terminals in bold. The non-terminals **stmt** and **expr** represent statements and expressions respectively, for which the rules are not shown.

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

(a) Ambiguous source code.

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

(b) Parse: **else** belongs to inner **if**.

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

(c) Parse: **else** belongs to outer **if**.

Listing 3.2: Example of the dangling else problem in C. The C source code (a) could have two potential parse trees ((b) and (c)) when using the grammar in Listing 3.1.

3.2.3.1 Dangling Else Ambiguity

A naive grammar for C (Listing 3.1) contains ambiguity around **if**/**else** statements [33, 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 [35].

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 3.2(b) and Listing 3.2(c), respectively).

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 [36]. Listing 3.3 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**.

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 open statements were allowed 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. The above method for finding the matching **else** would not work.

```
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 3.3: Using open and closed statements to solve the dangling else problem. Other statement rules with no sub-statements are added to `closed-stmt`. Rules with sub-statements, e.g. rules for `for` and `while`, are duplicated across `open-stmt` and `closed-stmt`, having open and closed sub-statements, respectively.

```
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,
            e1,
            e2
        ),
    ...
};
```

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

Listing 3.4: Example of the LALRPOP syntax for grammar rules. (a) is the rule for an addition expression, and (b) is the Rust code to represent it. In LALRPOP, the grammar parsing and AST generation code are combined.

3.2.3.2 LALRPOP Parser Generator

Listing 3.4 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.

LALRPOP 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 3.5). The macro has a generic type `T`, and returns the list items as a `Vec<T>`.

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

Listing 3.5: LALRPOP macro to parse a comma-separated list of non-terminals.

3.2.3.3 Additional Parser Details

The parser contains other features that warrant discussion. The handling of escape sequences in strings is described in Appendix A.1. Subtleties around parsing type specifiers are discussed in Appendix A.2.

3.3 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 3.6. The middle and back ends make use of a context object design pattern, discussed in Appendix A.4.

3.3.1 Intermediate Code Generation

I defined a custom three-address code IR (the instructions are listed in Appendix C). I decided to create a custom instruction set instead of using an existing one as it allowed me to tailor my compiler with exactly the features I support. For example, the IR does not include indirect function calls. I also added instructions specific to the Relooper algorithm. 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 an `Id` trait to abstract this concept, together with a generic `IdGenerator` struct (Listing 3.6). 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.

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)` Listing 3.7. `program` is the mutable intermediate representation, to which instructions and

```rust
trait Id {
    fn initial_id() -> Self;   // Base case
    fn next_id(&self) -> Self; // Generate next ID inductively
}

struct IdGenerator<T: Id + Clone> {
    max_id: Option<T>,         // Internally track the highest ID used so far
}

impl<T: Id + Clone> IdGenerator<T> {
    // Static method to initialise an IdGenerator
    fn new() -> Self {
        IdGenerator { max_id: None }
    }
    // Use the IdGenerator to get a new ID
    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()); // Update the stored highest ID
        new_id
    }
}
```

Listing 3.6: Implementation of the `Id` trait and `IdGenerator` struct, used to inductively generate IDs for objects in the intermediate representation.

metadata are added as the AST is traversed. `context` is the context object described in Appendix A.4, 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 3.7; 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 required care to ensure the semantic meaning of the program was preserved. For example, **switch** statements can contain **case** blocks in any order. Some blocks may fall-through 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. At the start of the **switch** statement, conditional branches to each **case** block are inserted. After these come the blocks themselves. By doing this, there was no direct fall-through between any blocks; instead, such a block will end in a branch instruction to the start of the next 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 3.3 shows the structure of the generated instructions for **switch** statements.

```rust
fn convert_statement_to_ir(stmt: Statement, prog: &mut Program, context: &mut Context)
        -> Vec<Instruction> {
    let mut instrs = Vec::new();
    match stmt {
        Statement::While(cond, body) => {
            let loop_start_label = prog.new_label(); // Create labels for start/end of loop
            let loop_end_label = prog.new_label();
            // Create a new loop context
            context.push_loop(LoopContext::while_loop(loop_start_label, loop_end_label));
            instrs.push(Instruction::Label(loop_start_label));
            // Evaluate loop condition
            let (cond_instrs, cond_var) = convert_expression_to_ir(cond, prog, context);
            instrs.append(cond_instrs);
            // Branch out of loop if condition is false
            instrs.push(Instruction::BrIfEq(cond_var, Constant::Int(0), loop_end_label));
            // Recursively convert loop body
            instrs.append(convert_statement_to_ir(body, prog, context));
            // Branch back to the start of the loop
            instrs.push(Instruction::Br(loop_start_label));
            instrs.push(Instruction::Label(loop_end_label));
            // Remove the loop context we added
            context.pop_loop();
        }
        // ... other AST statement nodes ...
    }
    instrs
}
```

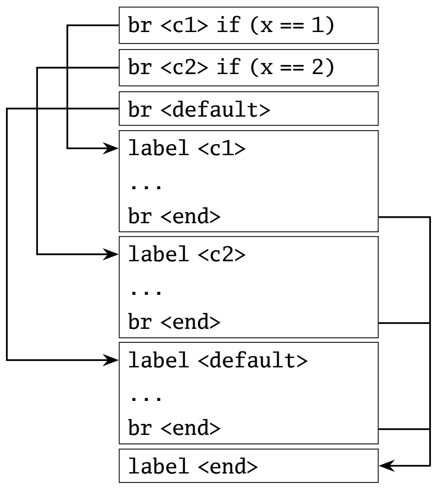
Listing 3.7: Generating IR code for a `while` statement. The AST is pattern matched to handle each type of statement separately. The other cases are not shown here.

Declaration statements have many different formats, each of which I had to handle separately; e.g. variables may be declared without being initialised with a value.

Function declarations only differ syntactically from other declarations by the type of the identifier; therefore they are handled in the same place. For function definitions (i.e. declarations plus body), the body code is converted and the function is added 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. Furthermore, an explicit size can either be a static value or a variable that is only known at run-time (creating a variable-length array). To handle variable-length arrays, an instruction is inserted to allocate space on the stack for it at run-time. 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 are evaluated at compile-time, such as array length expressions. My implementation of this is described in Appendix A.3.



```
br <c1> if (x == 1)
br <c2> if (x == 2)
br <default>
label <c1>
...
br <end>
label <c2>
...
br <end>
label <default>
...
br <end>
label <end>
```

Figure 3.3: IR instructions generated for **switch** statements.

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

Figure 3.4: Supported types in the IR. **Ix** and **Ux** types represent signed and unsigned integers, of bit-width x.

3.3.2 Types

Figure 3.4 outlines the types supported by the IR, mirroring the types supported by the C language [33, 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].

I implemented the ISO C unary and binary conversions for types [33, pp. 174–176]. They are applied before unary and binary operations, respectively. Unary conversion constrains the possible types an operand can have. 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 are of the same type. Firstly, the unary conversions are applied to each operand individually. Then, if both operands are arithmetic, and one operand has a smaller type than the other, the smaller is converted to the larger type. This includes integer promotion 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 the instruction `t = a < b`, 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.

[1]The C specification does not define the exact bit widths, only the minimum size.
[2]**I32** is used to represent booleans.

3.3.3 The Relooper Algorithm

The Relooper algorithm was described in Section 2.2. Here, I describe its implementation. The Relooper stage takes the IR as input, and transforms the instructions into Relooper blocks whilst preserving program metadata.

Firstly, the intermediate code is 'soupified'. This is the process of taking the linear sequence of instructions and producing a set of *labels*[1] (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 Algorithm 3.1:

$\boxed{1}$ Remove any label fall-through. By this I mean any label instruction that is not preceded by a branch in the linear instruction sequence, to which control will flow directly from the previous instruction. A branch instruction is inserted before each label instruction, if one does not already exist, branching to that label. This ensures that a label instruction is always preceded by a branch instruction along every control flow path.

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

$\boxed{2}$ Insert an unconditional branch after each conditional branch, to ensure that conditional branches do not occur in the middle of a block. A new label instruction is created for the unconditional branch, if necessary.

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

$\boxed{3}$ If there is no label instruction at the very start of the instructions, add one.

$\boxed{4}$ Merge any consecutive label instructions into a single instruction, updating branches to the labels accordingly.

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

Algorithm 3.1: 'Soupifying' the intermediate code. The input is a linear instruction sequence, and a set of label blocks is produced.

Steps $\boxed{1}$ to $\boxed{4}$ 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 Algorithm 2.1 (Section 2.2.2) is used to create a Relooper block. The `IdGenerator` struct (Listing 3.6) is used to assign each loop and multiple block a unique ID.

In Algorithm 2.1, the *reachability* set of a label is used to determine which type of block to generate. The reachability of each label is calculated by taking the transitive closure of its possible branch

[1]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.

```
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 3.8: IR code and generated target code for transforming an add instruction, assuming `a` and `b` are variables of type `I64`.

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.) Subsequent steps of the algorithm are implemented as described earlier.

3.4 Back End: Target Code Generation

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`, and its byte representation is written to a binary file as output.

I defined a `ToBytes` trait that is implemented by all data structures that are written 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 and nothing more.

The core function of the back end is to transform IR instructions to WebAssembly instructions. Since WebAssembly has 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 3.8 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.

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 indexes for items in the WebAssembly module, such as functions and types.

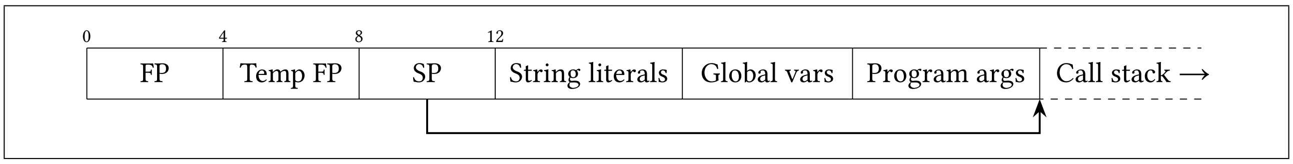


Figure 3.5: Memory structure, containing the frame pointer (FP), temporary frame pointer, and stack pointer (SP), followed by other sections of memory. Addresses increase to the right; compile-time known memory addresses are labelled above.

3.4.1 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.

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 frame pointer', 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 registers, so instead I 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, is 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 (see Section 3.4.3 below).

Programs may accept command-line arguments, which are stored into memory by the runtime environment. This is described in more detail in Section 3.5 below.

The function call stack grows upwards dynamically from this point. I did not implement heap storage, so memory only increases from one end, unlike in standard C compilers such as GCC [22].

3.4.2 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.

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 (see Section 3.4.3 below).

The procedure for pushing a new stack frame is Algorithm 3.2:

```
1  Store the current FP at the top of the stack.
```

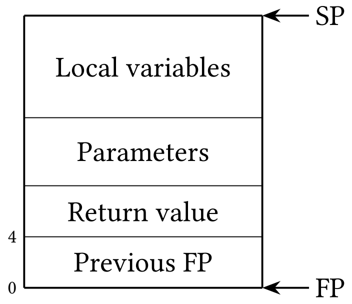


Figure 3.6: Memory layout of a stack frame, showing where the frame pointer (FP) and stack pointer (SP) point to. Addresses increase upwards; compile-time known offsets from the FP are labelled.

$\boxed{2}$ Copy the current SP to Temp FP: the address of the start of the stack frame.

$\boxed{3}$ Allocate space for the return value on top of the stack, according to the return type.

$\boxed{4}$ Store each function parameter on top of the stack. This is either copying a variable from the stack frame below or storing a constant.

$\boxed{5}$ Set the FP to the value held in Temp FP.

Algorithm 3.2: Pushing a new stack frame.

Using the temporary FP here is necessary since overwriting the FP directly would prevent copying local 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:

$\boxed{1}$ Set the SP to the current value of the FP. The new top of the stack is the top of the previous stack frame.

$\boxed{2}$ Restore the previous value of the FP.

$\boxed{3}$ If the function returns a result to a variable, copy the return value from the stack frame to the destination variable.

Algorithm 3.3: Popping a stack frame.

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.

3.4.3 Local Variable Allocation

Initially, I implemented a naive variable allocation strategy. As an extension, I implemented a more optimal allocation strategy, described in Section 3.6.3.

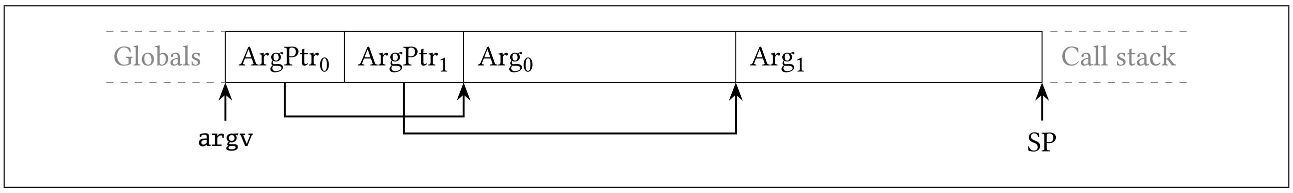


Figure 3.7: Memory structure of program arguments, with addresses increasing to the right. Here, the argument count (argc) is 2. The pointer to the start of the array (argv) and the stack pointer (SP) are labelled.

Variable allocation is performed at compile time, so that variable accesses can be static FP offsets rather than requiring a run-time address table. Only in specific cases—variable-length arrays, for example—are variables allocated at run-time. My IR has a dedicated `AllocateVariable` instruction that 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, but 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 run-time dependent. Therefore, variables are allocated as offsets from the FP. At run-time, a variable is accessed by first loading the FP, then adding the variable offset to get the variable's memory address.

3.5 Runtime Environment

The purpose of the runtime environment is to provide an interface between the WebAssembly binary and the system, thus allowing it to be executed. I chose to use Node.js, which allows the binary to be run locally [24]. Another option was to use JavaScript within a web browser, the primary target for WebAssembly; however, for the purposes of this project it would have added unnecessary complexity.

The runtime is responsible for instantiating the WebAssembly module. A module must be instantiated before any of its functions 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 access it. Alternatively, if the memory were created by the WebAssembly module, 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. Following the C standard, the first argument is always the name of the program being run. 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, 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.

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

3.6 Optimisations

After completing the main compiler pipeline, I implemented optimisations in the middle and back ends. I performed unreachable procedure elimination (Section 3.6.1) and tail-call optimisation (Section 3.6.2) in the middle end, and in the back end I created a more optimal stack allocation policy for local variables (Section 3.6.3).

3.6.1 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 never to be called.

Firstly, 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 it is likely to be sparse. (An adjacency matrix would be more suitable for densely connected graphs.) The call graph data structure also 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. Firstly, 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.

3.6.2 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.

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

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 with deep enough recursion, it allows programs to run that would otherwise crash due to lack 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 at the IR stage. This was much more successful. However, I kept the old optimisation for the cases of non-recursive tail-calls, since this still reduces memory usage there. I will describe both approaches in turn below.

3.6.2.1 Approach 1: Replacing the Caller's Stack Frame at Run-time

My first approach was to replace the caller's stack frame at run-time 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 do not need to be changed for the new stack frame.

To store the parameters in the new stack frame, initially 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 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.

3.6.2.2 Approach 2: Transforming Recursion to Iteration

The second approach is entirely contained within 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.

[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.

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

(a) Original function code.

```c
long sum(long n, long acc) {
start:
    if (n == 0) {
        return acc;
    }
    // Copy recursive arguments to param vars
    long t0 = n - 1;
    long t1 = acc + n;
    n = t0;
    acc = t1;
    // Loop back to start of function
    goto start;
}
```

(b) Tail-call optimised function.

Listing 3.9: Example of transforming tail-recursion to iteration.

Similarly to the previous approach, we start by finding all `call` instructions that are both recursive and are tail-calls. In the current function, there is a variable for each formal parameter. For each actual parameter to the recursive call, we copy it to corresponding formal parameter in the current function. Once done for each parameter, a branch instruction is added to the start of the function. Listing 3.9 shows an example of this transformation. (C code is shown for clarity, however the actual optimisation happens to the intermediate code.)

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.

3.6.3 Stack Allocation Policy

My initial variable allocation policy is described in Section 3.4.3. Each variable is allocated a disjoint address range in memory. However, this is inefficient since most variables are temporary variables that are only used once, and many of them have non-overlapping define-to-use ranges (i.e. they never *clash* in the context of liveness analysis).

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

The optimised allocation is performed with the following steps:

$\boxed{1}$ Remove dead variables.

$\boxed{2}$ Generate the instruction flowgraph.

$\boxed{3}$ Perform live variable analysis.

$\boxed{4}$ Generate the clash graph.

$\boxed{5}$ Allocate variables from the clash graph.

Algorithm 3.4: Optimised variable allocation.

3.6.3.1 Live Variable Analysis

An instruction flowgraph is similar to the function call graph used in Section 3.6.1, but at 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 3.3.3), creating a node for each instruction, and creating edges along all possible control flow paths.

Live variable analysis (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 defined by instruction n, and $ref(n)$ is the set of all variables referenced by n. Equation 3.1 defines $live(n)$, the set of variables that are live immediately before instruction n:

$$live(n) = \left(\left(\bigcup_{s \in succ(n)} live(s) \right) \setminus def(n) \right) \cup ref(n) \qquad \text{(Equation 3.1)}$$

We start with all variables live at successors of n, remove all variables that n assigns to, and then 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 Equation 3.1 to update the set of live variables, until there are no more changes (see Listing 3.10). 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).

One complication of LVA I had to solve was that if a variable is assigned to but is 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 overlapping location, producing incorrect results. To solve this, I added an optimisation to remove dead variables before running LVA.

I handled side-effecting instruction (e.g. function calls) that write to an unused variable by creating a ‘null destination’ variable. The instruction in the IR writes to the null variable. When transforming to target code, no write instruction is generated if the destination variable is the null variable.

[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.

```
// For every instr, which vars are live at that point
type LiveVariableMap = HashMap<InstructionId, HashSet<VarId>>;
fn live_variable_analysis(flowgraph: &Flowgraph) -> LiveVariableMap {
    let mut live: LiveVariableMap = LiveVariableMap::new();
    let mut changes = true;
    while changes {
        changes = false;
        for (instr_id, instr) in &flowgraph.instrs {
            // $\bigcup_{s \in succ(n)} live(s)$
            let mut out_live: HashSet<VarId> = HashSet::new();
            for successor in flowgraph.successors.get(instr_id) {
                out_live.extend(live.get(successor).unwrap_or(&HashSet::new());
            }
            for def_var in def_set(instr) { // $\setminus def(n)$
                out_live.remove(&def_var);
            }
            for ref_var in ref_set(instr) { // $\cup ref(n)$
                out_live.insert(ref_var);
            }
            // Update live variable set, and compare to previous value for changes
            let prev_live = live.insert(instr_id, out_live);
            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 3.10: Live variable analysis implementation, iteratively applying Equation 3.1.

3.6.3.2 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 3.11 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

```
let live_vars = live_variable_analysis(flowgraph);
for (_instr_id, vars_live_at_instr) in live_vars {
    // Add a clash between all vars simultaneously live
    while let Some(var) = vars_live_at_instr.pop() {
        for other_var in vars_live_at_instr {
            clash_graph.add_clash(var, other_var);
        }
    }
}
for instr in flowgraph.instrs {
    if let Instruction::AddressOf(_, _, var) = instr {
        clash_graph.add_universal_clash(var); // Over-approximate address-taken variables
    }
}
```

Listing 3.11: Algorithm to generate the clash graph from the results of live variable analysis.

make each address-taken variable clash with every other variable[1]. 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.

3.6.3.3 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 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:

$\boxed{1}$ Choose a variable with the least number of clashes. Break ties by choosing smaller variables.

$\boxed{2}$ Remove the variable and its edges from the clash graph.

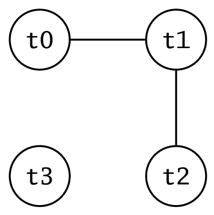
$\boxed{3}$ Allocate the variable to the lowest memory address where it does not clash with already allocated variables.

$\boxed{4}$ Repeat from Step $\boxed{1}$ until the clash graph is empty.

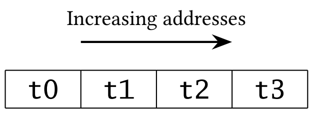
[1]Pointer analysis may be able to improve upon this to reduce the set of clashes for address-taken variables, but cannot solve the problem entirely.

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

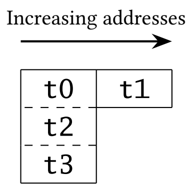
(a) Intermediate code, with live variable analysis.



(b) Clash graph.



(c) Memory allocation (naive policy).



(d) Memory allocation (optimised policy).

Figure 3.8: Example of the optimised stack allocation policy. (a) is the intermediate code for which memory is being allocated, showing the set of live variables at each instruction. (b) is the clash graph generated after running live variable analysis. (c) and (d) show the allocations made by the naive and optimised policies, respectively. Memory addresses increase to the right; variables overlapping vertically, separated by dashed lines, are allocated to the same address.

Algorithm 3.5: Variable allocation heuristic.

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

I tested allocating variables in both orders: least clashes or most clashes first. I found that contrary to register allocation, allocating variables with least clashes first resulted in more efficient memory use. 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 fewer 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.

3.7 Repository Overview

I developed my project in a Git repository, ensuring to regularly push to the cloud for backup purposes. The high-level structure of the codebase is shown in Table 3.1. 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. All code was written by myself.

`src/`	Compiler source code.
`program_config/`	Compiler constants and run-time 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.
`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/`	Auxiliary scripts used for testing and generating graphs.
`profiler.py`	Plot stack usage profiles.
`testsuite.py`	Test runner script.
`tests/`	Automated test specifications.

Table 3.1: Repository structure.

3.8 Summary

I implemented a complete compiler from C to WebAssembly. I created my own IR to represent the program in the middle end, enabling optimisations. I successfully implemented the Relooper algorithm to transform the control flow. I performed tail-call optimisation to significantly reduce memory usage of recursive functions, and I created a novel and more optimal memory allocation algorithm.

4
Evaluation

In this chapter, I evaluate my project against my success criteria (Section 4.1), showing that all the success criteria were achieved. Section 4.2 demonstrates that the compiler is correct using a variety of test programs. Section 4.3 evaluates 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 the success criteria have been met. The first four correspond to the main compiler pipeline. The correctness of these stages is verified by the fifth criterion. Section 4.2 describes the variety of test programs used to verify the compiler's correctness.

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, each testing a specific construct in the C language.
- 11 'full programs', representing real workloads.

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 specific behaviour, and should be independent from the rest of the program's functionality [37]. My test programs do not test an isolated behaviour of the compiler. Instead, they test a single behaviour of the C source code that is being compiled (e.g. pointer dereferencing), verifying that the compiler pipeline maintains the correct behaviour. This allowed me to find and fix bugs more easily.

Examples of ‘full programs’ include Conway’s Game of Life, calculating the Fibonacci numbers, and finding substring occurrences in a string [38, 39]. Six of the programs were sourced from Clib, which contains many 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, permitting use “without restriction” [40]. 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 [22]. I deemed a program to be compiled correctly 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 and extend the compiler. This script reads a directory of YAML files, each specifying a test program and its arguments [41]. It then compiles the programs with both my compiler and GCC, and compares the outputs. If the outputs are identical, the test passes, otherwise it fails.

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

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

4.3 Impacts of Optimisations

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

To evaluate the memory usage optimisations, the compiler inserts profiling code to the programs, which measures the size of the stack throughout program execution. When generating instructions that move the stack pointer, the compiler inserts a call to `log_stack_ptr`, a function imported from the runtime that appends the current stack pointer value to a log file. I wrote a Python script to visualise the resulting data. The plots show how 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 is mainly beneficial in removing unused standard library functions from the compiled binary. When a standard library header is included in a program, the preprocessor inserts the entire code of that library[2]. If the program only uses a few of the functions, the rest are redundant. Unreachable procedure elimination safely removes them, resulting in a smaller binary.

[1]GCC version 11.3.1.

[2]I.e. the entirety of my skeleton implementation of that library.

```c
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*.

The standard library with the most functions that I implemented was `ctype.h`. This 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 libraries I implemented contained only a few functions, so the impact of this optimisation was much more limited. If I implemented more of the standard library, I would see much more improvement.

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

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. 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 (Figure 4.1).

One of the functions I used to evaluate this optimisation uses tail-recursion to compute the sum of the first n integers (Listing 4.1).

Figure 4.1 compares the stack memory usage of this function with and without tail-call optimisation. Without the optimisation, the stack size grows with $\mathcal{O}(n)$, whereas with the optimisation it is $\mathcal{O}(1)$. When running the program with $n = 500$, a stack size of 46.3 kB is reached. When 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 large n is.

When testing with large n, the non-optimised version quickly runs out of memory, and throws an exception. In contrast, the optimised version has no memory constraint on the number of iterations. It successfully runs with $n = 1\,000\,000$, at which point it fails with GCC.

4.3.3 Optimised Stack Allocation Policy

My stack allocation policy was successful in reducing the amount of stack memory used.

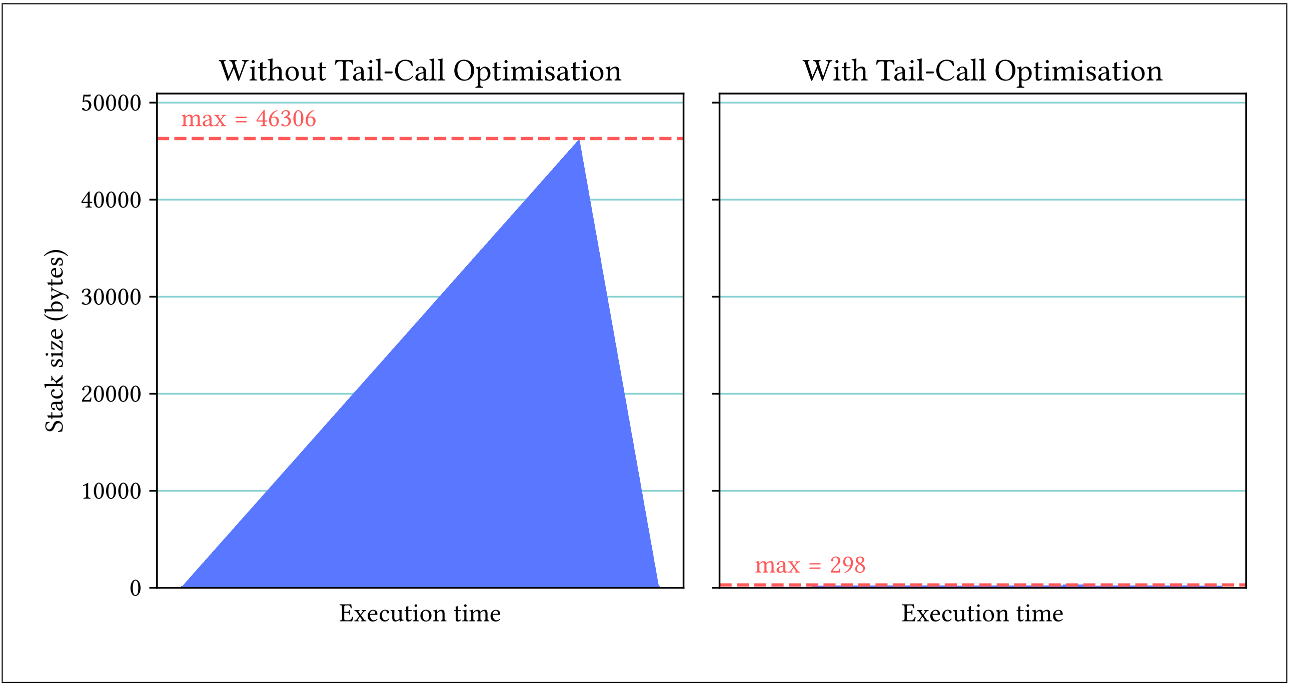


Figure 4.1: Stack usage for calling `sum(500, 0)` (see Listing 4.1)

From the 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, against which the optimised program was measured. 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 the optimisation on both the optimised and unoptimised versions. I did this because tail-call optimisation also affects memory usage, so it may have an impact on the effectiveness of this optimisation.

One of the main factors influencing the degree of improvement is the number of temporary variables generated. The more temporary variables generated, the more scope for the compiler to find non-clashing variables to overlap. Since temporary variables are generated locally for each instruction, they only have short-range dependencies. Only 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 its number of operations and hence the number of temporary variables increases, so too 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 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. Hence we can interpret which parts of the plot correspond to which parts of the source program.

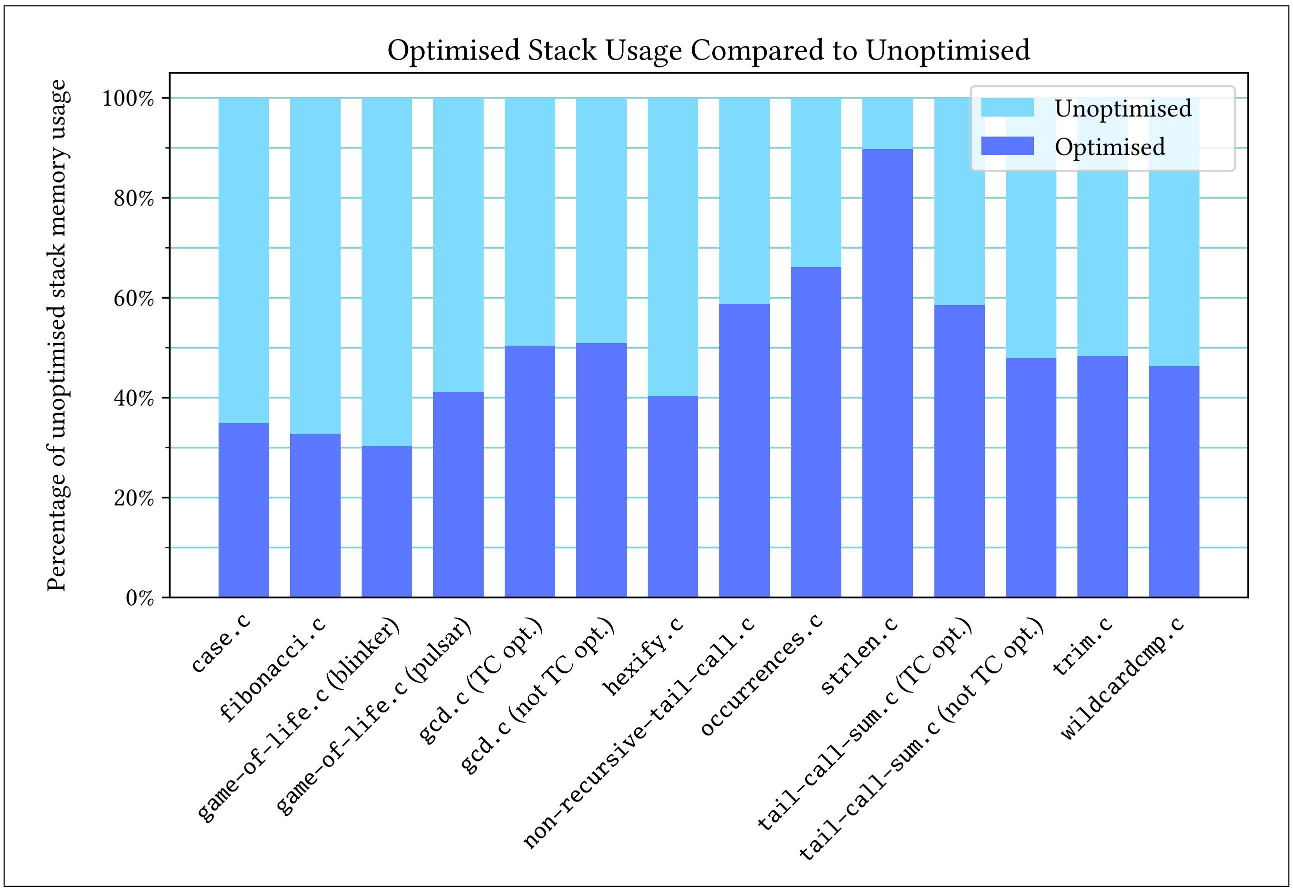


Figure 4.2: Comparing optimised against unoptimised stack memory usage to usage. The shorter the optimised bar, the greater the improvement (less memory is used). *TC opt*: tail-call optimised. *Blinker* and *pulsar* are specific instances of the Game of Life program.

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

`case_upper()` and `case_lower()` each make repeated calls to `toupper()` and `tolower()`, which correspond 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 in the body of the function. Listing 4.2(b) 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()` (Listing 4.2(a)). This causes the larger spike at the end of Figure 4.3.

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.

This optimisation also has a large impact on recursive functions. Since this optimisation reduces the size of each stack frame, the improvement is greater when there are many recursive stack frames. Figure 4.4 shows the size of the stack for calculating the Fibonacci numbers recursively. In this instance, the optimised stack allocation policy reduced the stack size of the program by 67.20 %.

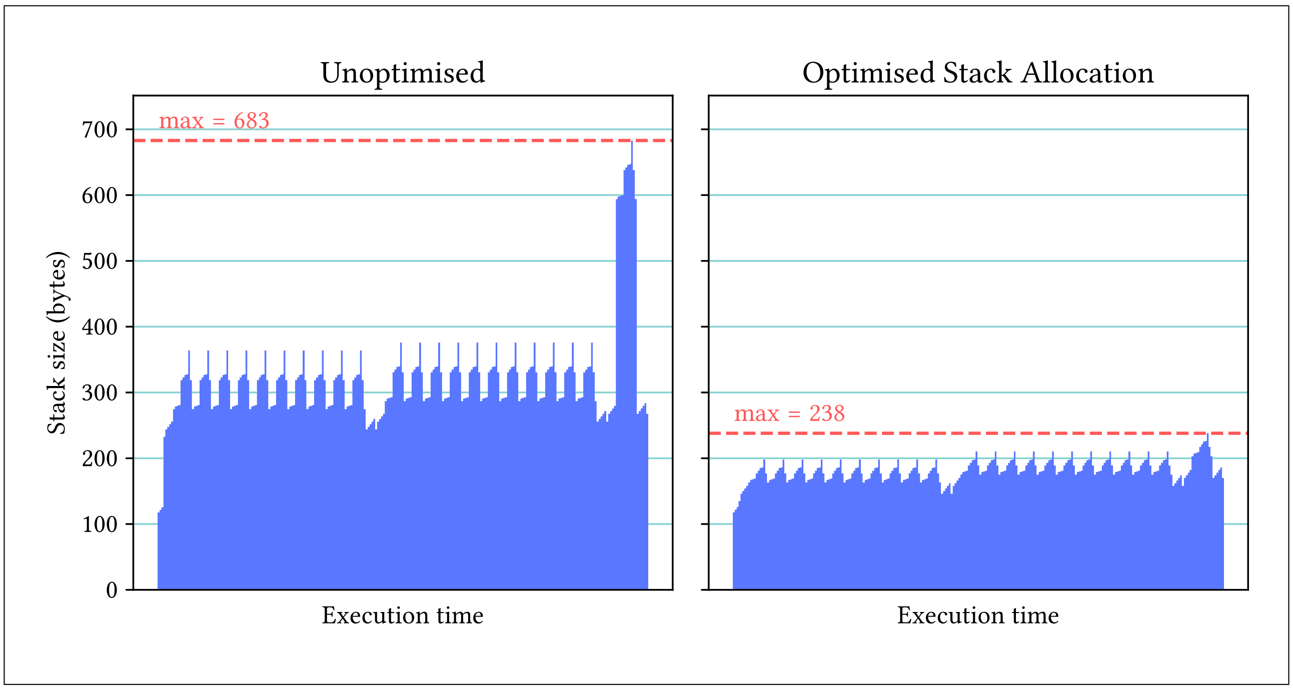


Figure 4.3: Comparing stack usage for `case.c`.

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

(a) The entire body of `case_upper()`.

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

(b) A short section of the body of `case_camel()`.

Listing 4.2: Comparing the body code of `case_upper()` code and `case_camel()`

4.4 Summary

The main objective of this project is to produce a compiler that generates correct WebAssembly binary code. As demonstrated by the testing described above, this objective has been achieved.

The purpose of adding optimisations is to improve the performance of the compiled programs, whilst maintaining the program's semantics. The correctness of my optimisations is verified by the correct output of compiled programs. In the previous sections, we have seen measurable evidence that the optimisations improved performance, particularly by reducing memory usage. Therefore, the optimisations can be considered successful.

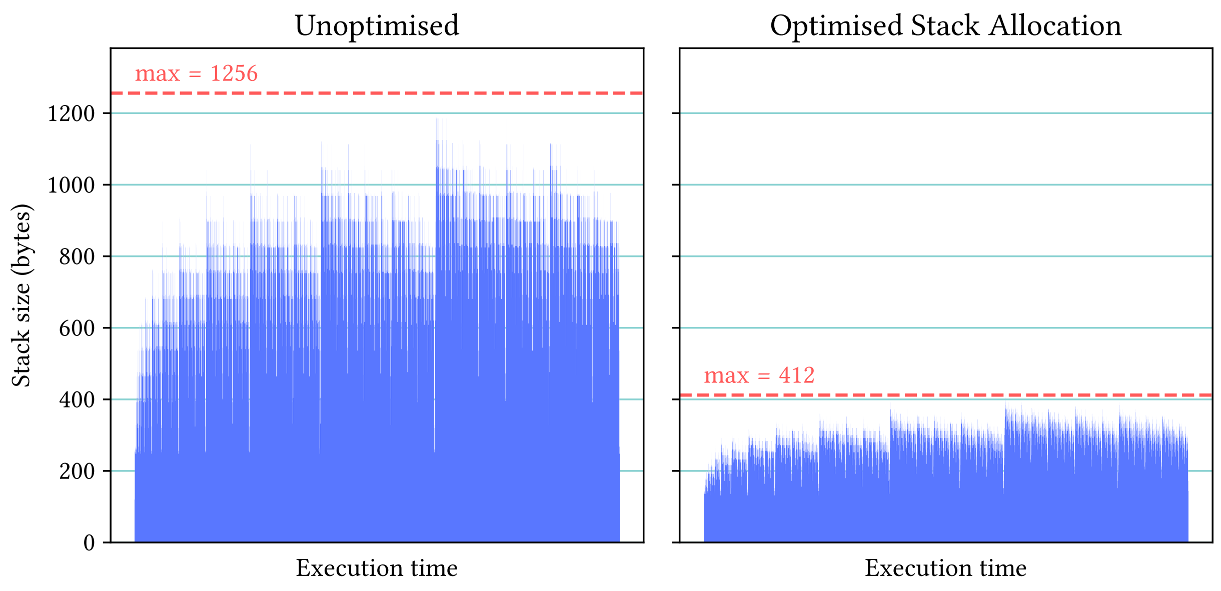


Figure 4.4: Comparing stack usage for `fibonacci.c`.

5 Conclusions

In this chapter I summarise what was achieved in the project and reflect on the lessons learnt. I also offer suggestions of how the work may be taken forwards, if I were to continue the project.

5.1 Project Summary

The project was a success; I met all my success criteria and additionally an extension. I completed an entire compiler pipeline that transforms a subset of the C language into a WebAssembly binary: the front end, which parses the source code into an AST (Section 3.2); the middle end, which transforms the AST into my custom IR (Section 3.3); and the back end, which generates a WebAssembly module from the IR (Section 3.4). I provided a Node.js runtime environment that runs the WebAssembly binary and interfaces to the system and standard library (Section 3.5).

Furthermore, I implemented several optimisations to the program in the middle and back ends of the compiler. Unreachable procedure elimination reduced the size of generated binaries by pruning unused functions from the call graph (Sections 3.6.1 and 4.3.1). I used tail-call optimisation to drastically reduce the amount of memory used by recursive functions, to the extent of being able to run programs not possible with GCC (Sections 3.6.2 and 4.3.2). I created a more optimal stack allocation policy, experimentally tweaking my heuristic to produce significant improvements to programs' memory use (Sections 3.6.3 and 4.3.3).

My compiler produced correct binaries that maintained the semantics of the source program, when comparing my test programs against GCC. My evaluation showed that my optimisations were successful in improving the performance of programs (Section 4.3). Tail-call optimisation reduced $\mathcal{O}(n)$ memory use to $\mathcal{O}(1)$ for recursive functions, and my stack allocation policy significantly decreased memory usage.

I gained experience using Rust, including learning idiomatic ways of structuring my code and how to work with the borrow checker to produce memory-safe code. Throughout the project, I had opportunities to apply theory from the Tripos; ranging from Algorithms course Software and Security Engineering, as well as putting into practice the recent II Optimising Compilers and Advanced Computer Architecture courses. This allowed my to consolidate and expand my skill set.

The project progressed in line with the timetable set out in the proposal; much of the time, I was slightly ahead of schedule, allowing me to be thorough in my testing. This left me enough time to successfully implement the stack allocation policy optimisation as an extension.

5.2 Further Work

The most obvious continuation of this project would be to expand the scope to support a larger subset of C. For example, I could add support for function pointers or linking.

I only implemented a small subset of the standard library. If I were taking this project further, I would implement more of the standard library, such as the string manipulation functions. This would provide support for many more source programs. Notably, I would also implement `malloc()` and `free()`, to provide support for heap memory allocation. In my current memory layout, the heap would grow downwards from high memory. The runtime environment would keep a list of available memory, from which it would allocate when `malloc()` is called. Calling `free()` would return the memory to the available list.

Furthermore, additional optimisations could be added to the compiler, increasing performance of compiled programs in terms of memory usage, execution speed, and binary code size. These can be as sophisticated as desired; there are many analyses that can be done at compile time, including dataflow analysis, constraint-bases analysis, effect systems, and so on. Pointer analysis—such as Andersen's points-to analysis—can be used to narrow down the possible targets of a pointer [42]. This makes other analyses more precise, for example the clash graph in Section 3.6.3.2

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 14/10/2022).

[2] Stephen Akinyemi (appcypher). *Awesome WebAssembly Languages*. GitHub repository. URL: `https://github.com/appcypher/awesome-wasm-langs` (visited on 11/04/2023).

[3] Emscripten contributors. *Emscripten*. URL: `https://emscripten.org/` (visited on 11/04/2023).

[4] The LLVM admin team. *The LLVM Compiler Infrastructure*. URL: `https://llvm.org/` (visited on 11/04/2023).

[5] Learning Technologies. *Cheerp — the C/C++ compiler for Web applications*. URL: `https://docs.leaningtech.com/cheerp/` (visited on 11/04/2023).

[6] Learning Technologies. *CheerpJ — Convert Java to WebAssembly and JavaScript*. URL: `https://docs.leaningtech.com/cheerpj/` (visited on 21/04/2023).

[7] Microsoft. *ASP.NET Core Blazor*. URL: `https://learn.microsoft.com/en-gb/aspnet/core/blazor/` (visited on 21/04/2023).

[8] Pyodide contributors and Mozilla. *Pyodide*. URL: `https://pyodide.org/en/stable/` (visited on 21/04/2023).

[9] Shopify. *Javy: A JavaScript to WebAssembly toolchain*. URL: `https://github.com/shopify/javy` (visited on 21/04/2023).

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

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

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

[13] 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 02/03/2023).

[14] 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`.

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

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

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

[18] Roger S. Pressman. "The Incremental Model". In: *Software Engineering: A Practitioner's Approach*. 6th ed. 2004. Chap. 3.3.1. ISBN: 9780073019338.

[19] Roger S. Pressman. "The Waterfall Model". In: *Software Engineering: A Practitioner's Approach*. 6th ed. 2004. Chap. 3.2. ISBN: 9780073019338.

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

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

[22] Free Software Foundation, Inc and The GCC team. *GCC, the GNU Compiler Collection*. URL: `https://gcc.gnu.org/` (visited on 10/04/2023).

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

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

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

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

[27] *LALRPOP Documentation*. Licensed under the MIT License. URL: `https://lalrpop.github.io/lalrpop/index.html` (visited on 14/10/2022).

[28] *Crate nom*. URL: `https://docs.rs/nom/latest/nom/` (visited on 17/04/2023).

[29] *A thoughtful introduction to the pest parser*. URL: `https://pest.rs/book/` (visited on 17/04/2023).

[30] JetBrains. *CLion*. URL: `https://www.jetbrains.com/clion/` (visited on 04/03/2023).

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

[32] Free Software Foundation, Inc and The GCC team. *The C Preprocessor*. URL: `https://gcc.gnu.org/onlinedocs/cpp/` (visited on 27/02/2023).

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

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

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

[36] 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`.

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

[38] 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`.

[39] OEIS Foundation Inc. *The On-Line Encyclopedia of Integer Sequences. Sequence A000045.* URL: `https://oeis.org/A000045` (visited on 21/04/2023).

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

[41] *YAML: YAML Ain’t Markup Language.* URL: `https://yaml.org/` (visited on 21/04/2023).

[42] Lars Ole Andersen. “Program Analysis and Specialization for the C Programming Language”. PhD thesis. May 1994.

Appendix A
Supplementary Implementation Details

Add a chapter overview text here

A.1 Parsing String Escape Sequences

In C, string literals can contain escape sequences, such as `\n` to insert a new line character [33, sec. 2.7.5]. The parser needs to be convert these to the literal characters they represent. My compiler supports the escape codes described in Table A.1.

I implemented this by creating an iterator over the characters of the string, that replaces escape sequences as it emits characters (Listing A.1). When the current character is a backslash (‘\’), the next one or more characters form an escape sequence. The iterator consumes the characters of the escape sequence, and matches it against the valid escape sequences to perform the appropriate character substitution, which is emitted as the next character from the iterator. When the current character is not a backslash, it is emitted as-is.

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 (Listing A.2). I also wrote a set of unit tests to verify the iterator was functioning correctly.

Escape sequence	Represented character
`\'`	Single quote
`\"`	Double quote
`\?`	Question mark
`\\`	Backslash
`\n`	New line (line feed)
`\r`	Carriage return
`\t`	Horizontal tab
`\xhh…`	The byte given by hh… interpreted as a hexadecimal number
`\nnn`	The byte given by nnn interpreted as an octal number

Table A.1: Supported string escape sequences.

```rust
struct InterpretString<'a> {
    s: Peekable<Chars<'a>>,
}
impl Iterator for InterpretString<'_> {
    fn next(&mut self) -> Option<Self::Item> {
        match self.s.next() {
            Some(c) => Some(match c {
                '\\' => match self.s.next() {
                    Some('n') => Ok('\n'),  // New line
                    // \', \", \t, \r, etc.
                    Some('x') => { // Hexadecimal character code
                        // Check whether the next digit is a valid hex digit
                        match self.s.peek() {
                            Some(c) if c.is_ascii_hexdigit() => {
                                // Store the hex digits in a buffer
                                let mut buffer = String::new();
                                // Keep consuming hex digits
                                while let Some(c) = self.s.next() {
                                    buffer.push(c); // Consume the next hex digit
                                    match self.s.peek() { // Check the following char
                                        Some(c) if c.is_ascii_hexdigit() => continue,
                                        Some(_) | None => break,
                                    }
                                }
                                // Convert the hex code to the corresponding char
                                let hex_code = u32::from_str_radix(&buffer, 16).unwrap();
                                match char::from_u32(hex_code) {
                                    Some(c) => Ok(c),
                                    None => Err(InvalidCharCode(buffer)),
                                }
                            }
                            Some(c) => Err(InvalidEscapeChar(*c)),
                            None => Err(EscapeCharAtEndOfString),
                        }
                    }
                    Some(c) if c.is_digit(8) => {} // Octal character code similarly
                    Some(c) => Err(InvalidEscapeChar(c)), // Invalid escape code
                    None => Err(EscapeCharAtEndOfString),
                },
                c => Ok(c), // All other (non-escape) characters
            }),
            None => None,
        }
    }
}
```

Listing A.1: Implementation of an iterator to handle string escape sequences.

```
pub fn interpret_string(s: &str) -> Result<String, Error> {
    InterpretString::new(s).collect()
}
```

Listing A.2: Function wrapper around the `InterpretString` iterator (see Listing A.1).

A.2 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.

A.3 Compile-time Expression Evaluation

Some expressions are evaluated at compile-time, such as array length expressions. The expressions much be constant expressions, which means they are not permitted to contain assignments, function calls, and increments or decrements [33, sec. 7.11]. I implemented a compile-time expression evaluator that supports arithmetic expressions and ternaries (conditional expressions of the form `condition ? expr_if_true : expr_if_false`).

Listing A.3 shows the function that evaluates expressions. An expression at compile-time can either evaluate to an integer or a float, however only integers are supported as array lengths etc. The `eval` function recurses into the structure of an AST expression, evaluating subtrees before combining their results. An expression either returns a result or an error, that it cannot be evaluated as a constant expression at compile time. Errors will short-circuit the evaluation.

I do not support variables in compile-time evaluated expressions. A more sophisticated compile-time evaluator could look further up the AST to determine if a variable's value is a constant expression, and if so, use it.

A.4 Context Object Design Pattern

Throughout the middle and back ends, I used a design pattern of passing a context object through all function calls. For example, when traversing the AST to generate IR code, the `Context` struct in Listing A.4 was used to track information about the current context with respect to the source program. It tracks the stack of nested loops and switch statements, so that when a **`break`** or **`continue`** statement is converted, the branch target is known.

```rust
// Types that can be evaluated at compile-time
enum ConstantExpressionType {
    Int(i128),
    Float(f64),
}

fn eval(expr: Expression) -> Result<ConstantExpressionType, MiddleEndError> {
    match expr {
        Expression::Constant(c) => match c {
            AstConstant::Int(i) => Ok(Int(i as i128)),
            AstConstant::Float(f) => Ok(Float(f)),
            AstConstant::Char(c) => Ok(Int(c as i128)),
        },
        Expression::BinaryOp(op, left, right) => {
            let left_result = eval(left)?;
            let right_result = eval(right)?;
            match op {
                BinaryOperator::Add => match (left_result, right_result) {
                    (Int(l), Int(r)) => {
                        Ok(Int(l + r))
                    }
                    // ... other combinations of int and float ...
                },
                // ... other binary operators ...
            }
        }
        Expression::Ternary(cond, true_expr, false_expr) => {
            let cond_result = eval(cond)?;
            let cond_value = match cond_result {
                Int(i) => i != 0,
                Float(f) => f != 0.,
            };
            if cond_value {
                eval(true_expr)
            } else {
                eval(false_expr)
            }
        }
        Expression::Identifier(_) => Err(CantEvaluateAtCompileTime),
        Expression::FunctionCall(_, _) => Err(InvalidConstantExpression),
        // ... other expressions ...
    }
}
```

Listing A.3: Compile-time expression evaluator, supporting arithmetic expressions and ternaries. If the given expression is a supported constant expression, its result is returned; if not, an error is returned.

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

Listing A.4: 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 offered more modularity and flexibility. Firstly, the context information is encapsulated inside a separate data structure, allowing methods to be implemented on it that give calling functions access to the exact context information needed. It also enables 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` has a shorter lifetime than the `ModuleContext`, hence separating the data structures is ideal.

Appendix B

Lexer Finite State Machine

This chapter contains the full finite state machine (FSM) for the lexer (see Section 3.2.2).

Named states represent valid tokens, and blank states are invalid. Transitions are labelled with regular expressions, matching the input character. No input is consumed along an ϵ transition; the transition is automatically taken if the condition is true. Only valid transitions are shown; if no transition matches the input character, either the end of the current token has been reached, or the token is invalid. Transitions with no prior state are the initial transitions for the first input character.

If the machine stops in a named state, the corresponding token is emitted to the token stream. However, if the machine stops in an unnamed state, the token is invalid and a lexing error is raised.

In a slight abuse of regular expression notation, the dot character '`.`' and the backslash character '\' represent the respective characters literally.

Tokens labelled as symbols represent themselves. Named tokens correspond to the tokens described in Table B.1.

Label	**Represented token**
Dec	Decimal literal
FP	Floating point literal
Bin	Binary literal
Oct	Octal literal
Hex	Hexadecimal literal
Iden	Identifier
Keyword	C language keyword
Typedef	An identifier defined as a type name
String	String literal
Char	Character literal

Table B.1: Key to token names.

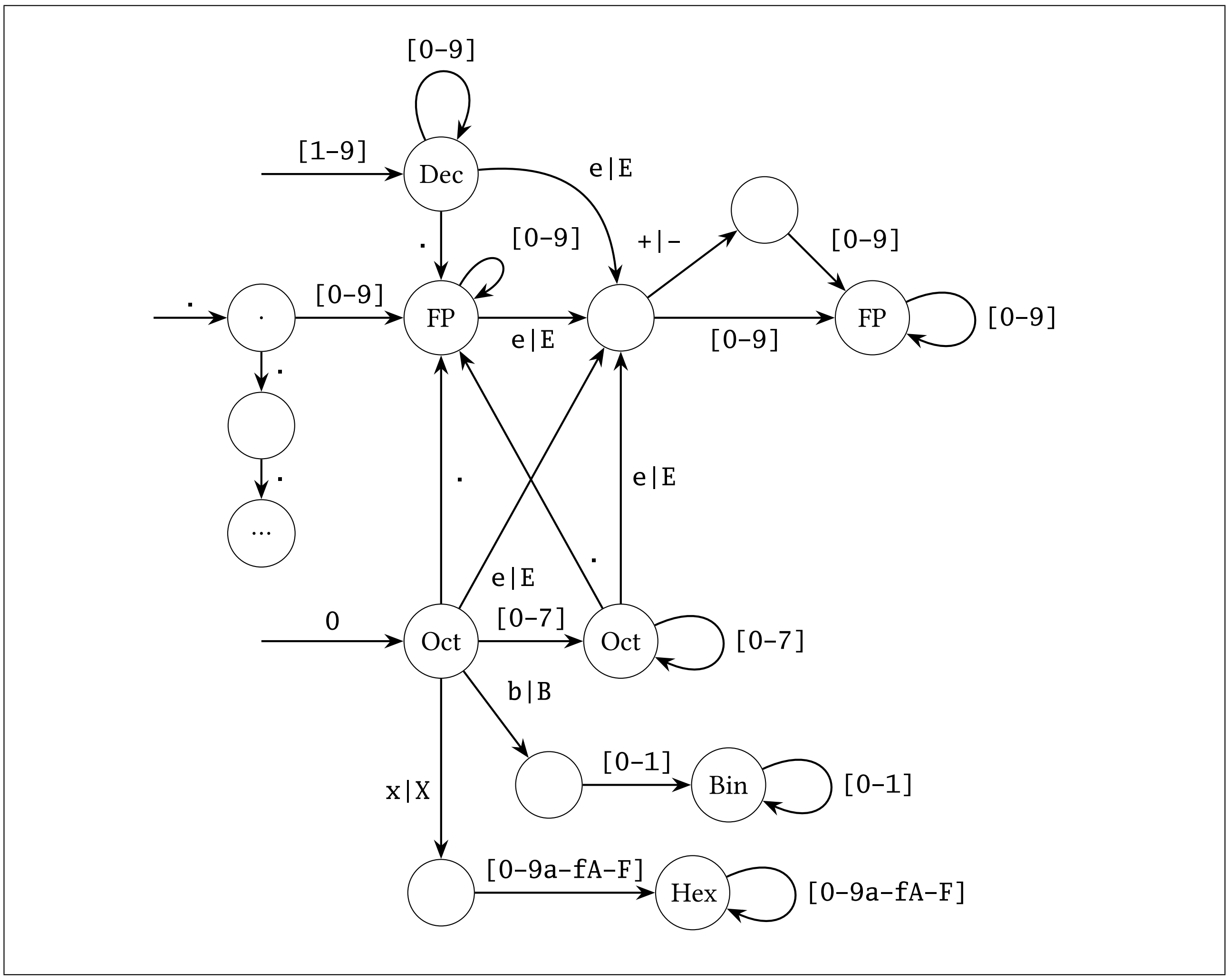


Figure B.1: Finite state machine for lexing number literals (decimal, floating point, binary, octal, and hexadecimal).

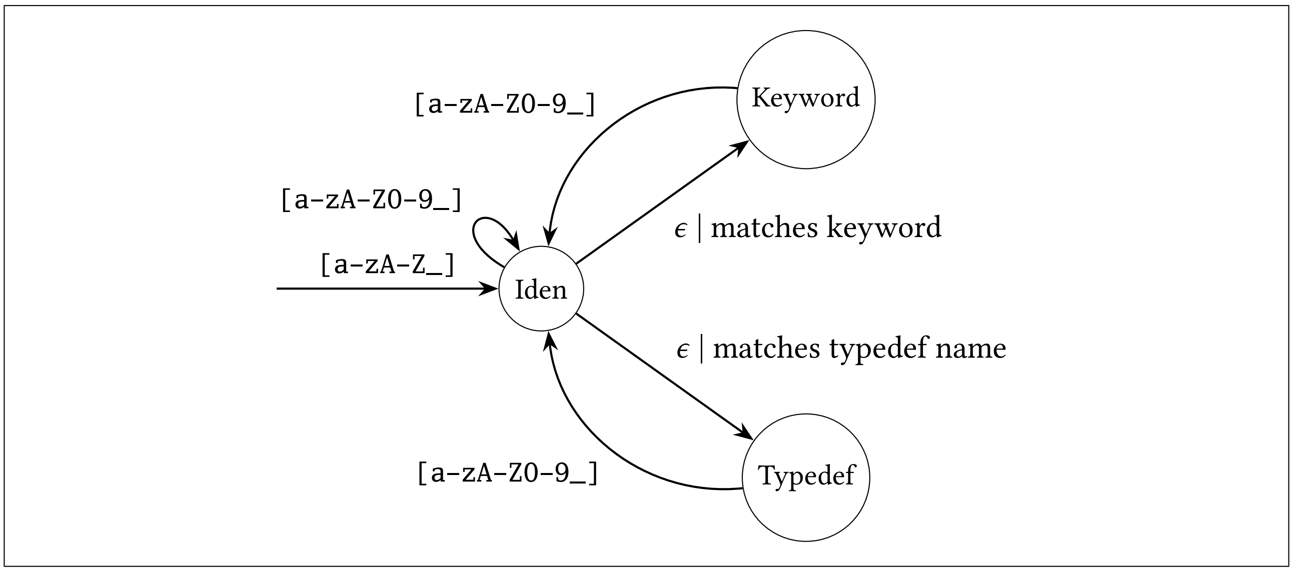


Figure B.2: Finite state machine for lexing identifiers, and matching them against keywords and typedef names.

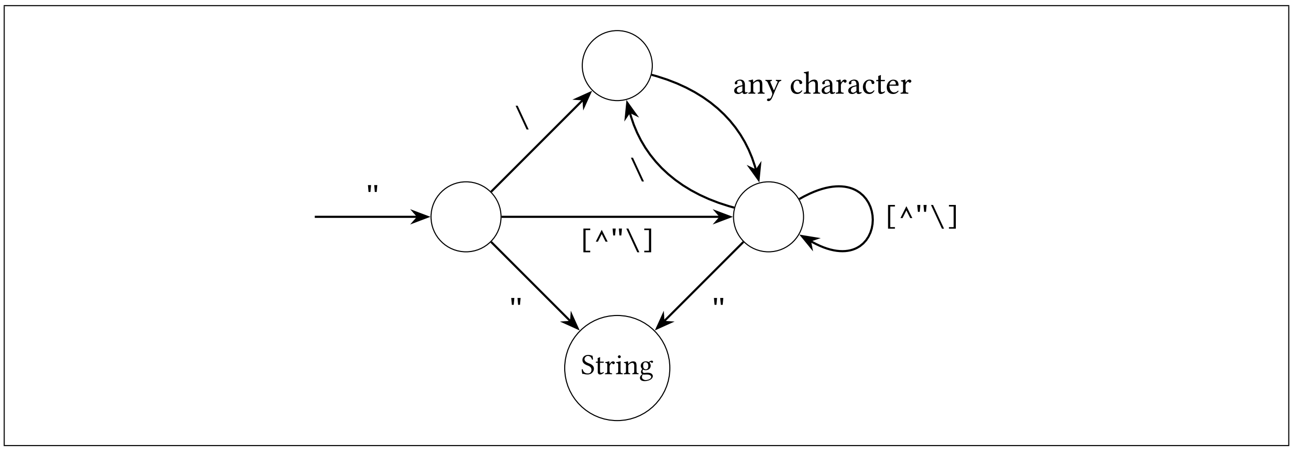


Figure B.3: Finite state machine for lexing string literals, enclosed in double quotes.

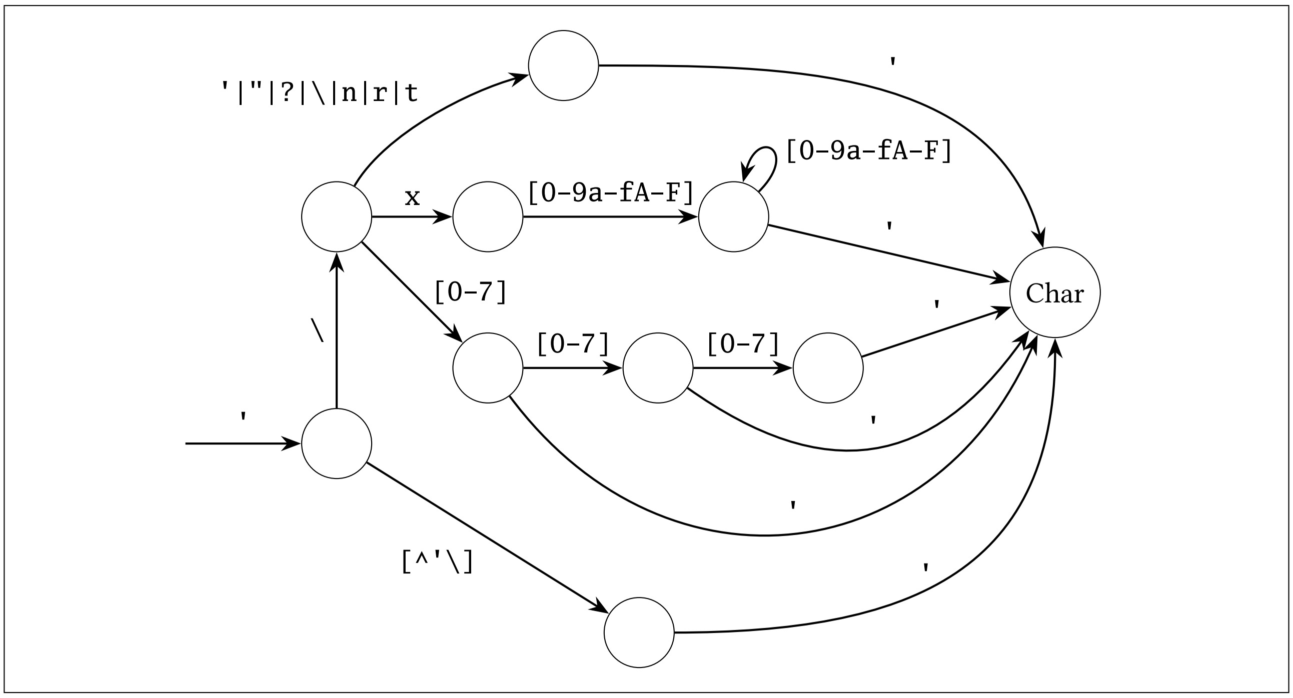


Figure B.4: Finite state machine for lexing character literals, enclosed in single quotes. The machine handles all allowed escape sequences.

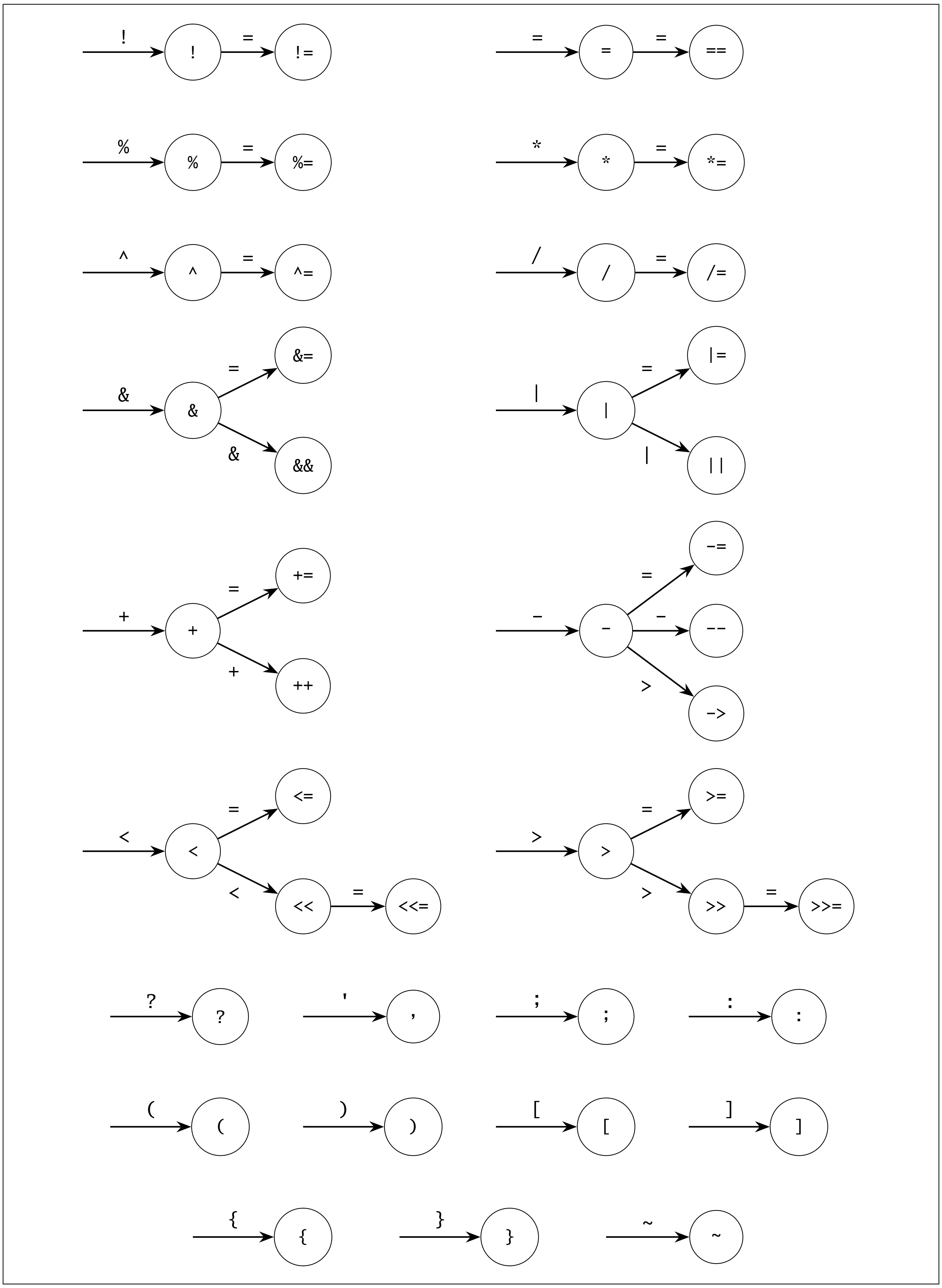


Figure B.5: Finite state machine for lexing operators and symbols.

Appendix C
Intermediate Code

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

Table C.1: IR code instructions.

`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.

Table C.1: IR code instructions (continued).

`t = call f(p₁, p₂, ...)`	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`
`t = (i8 ⟶ u16) x`
`t = (u8 ⟶ u16) x`
`t = (u8 ⟶ u16) x`
	Char promotions.

`t = (i16 ⟶ i32) x`
`t = (u16 ⟶ i32) x`
	Promotions to signed integer.

`t = (i16 ⟶ u32) x`
`t = (u16 ⟶ u32) x`
`t = (i32 ⟶ u32) x`
	Promotions to unsigned integer.

`t = (i32 ⟶ i64) x`
`t = (u32 ⟶ i64) x`
	Promotions to signed long.

`t = (i32 ⟶ u64) x`
`t = (u32 ⟶ u64) x`
`t = (i64 ⟶ u64) x`
	Promotions to unsigned long.

`t = (u32 ⟶ f32) x`
`t = (i32 ⟶ f32) x`
`t = (u64 ⟶ f32) x`
`t = (i64 ⟶ f32) x`
	Integer to float conversions.

`t = (u32 ⟶ f64) x`
`t = (i32 ⟶ f64) x`
`t = (u64 ⟶ f64) x`
`t = (i64 ⟶ f64) x`
	Integer to double conversions.

`t = (f32 ⟶ f64) x`	Float to double promotion.
`t = (f64 ⟶ i32) x`	Double to int conversion.

`t = (i32 ⟶ i8) x`
`t = (u32 ⟶ i8) x`
`t = (i64 ⟶ i8) x`
	Integer truncation.

Table C.1: IR code instructions (continued).

`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 D
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.

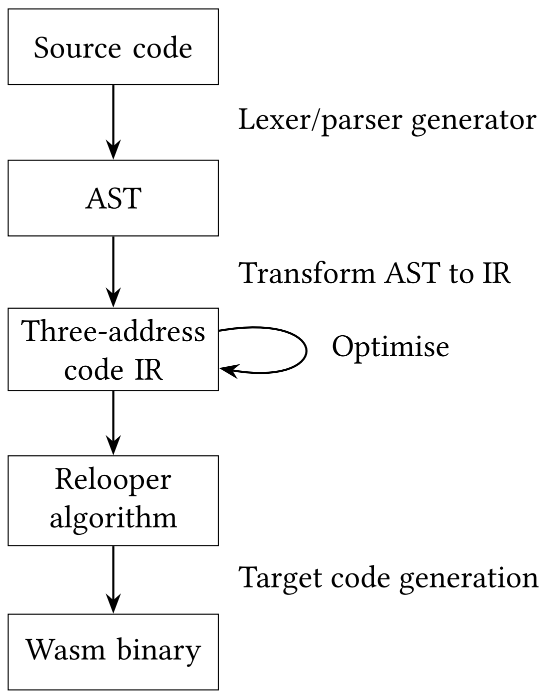
To test and evaluate the compiler, I will write small benchmark programs that individually test each of the features and optimisations I add. For example, I will use the Fibonacci program to test recursion. I will also test it with Conway's Game of Life, as an example of a larger program, to test and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