Martin Walls

C to WebAssembly Compiler

Computer Science Tripos: Part II

Churchill College

2nd March, 2023

Declaration of Originality

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

Signed: 

Date: 2nd March, 2023

Proforma

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

Original Aims of the Project

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

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

Work Completed

At most 100 words summarising the work completed.

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

[1] Excluding figures and listings.

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

Special Difficulties

None.

Contents

1 Introduction

Word budget: ~500–600 words

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

Talk about why I chose C as a source language

1.1 Background and Motivation

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

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

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

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

1.2 Survey of Related Work

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

2 Preparation

Word budget: ~2500-3000 words

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

2.1 WebAssembly

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

2.1.1 Primitive values

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

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

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

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

Table 2.1: WebAssembly primitive types.

```
fn leb128_encode_unsigned(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
}
```

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

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

WebAssembly also provides vector types and reference types. Vectors can store either integers or floats: in either case, the vector is split into a number of evenly-sized numbers. Vector instructions exist to operate on these values. `funcref` values are pointers to WebAssembly functions, and `externref` values are pointers to other types of object that can be passed into WebAssembly. These would be used for indirect function calls, for example. This project does not have a need for using either of these types; C does not have any concept of vector types, and I am not supporting function references. I will only use the four main integer and float types.

2.1.2 Instructions

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

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

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

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

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

2.1.3 Modules

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

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

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

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

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

The *memory section* is similar to the table section; is specifies the size limits of each linear memory of the program[2], in units of page size.

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

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

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

variables.

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

The *element section* is used to statically initialise the contents of tables with function addresses. Each element segment within this section starts with a flag that specifies the mode of the segment. For example, one mode is to automatically initialise the elements to table 0 upon module initialisation.

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

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

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

2.2 The Relooper Algorithm

2.3 Rust

2.4 Project Strategy

2.4.1 Requirements Analysis

2.4.2 Software Engineering Methodology

2.4.3 Testing

2.5 Starting Point

2.5.1 Knowledge and experience

- IB Compilers Course
- Experience with JavaScript + Python (cos I used those for runtime/testing)
- Experience writing C, my source language

2.5.2 Tools Used

- Say here that I learned Rust for this project – talk about the borrow checker and memory safety
- Also used JavaScript for runtime + Python for testing

3 Implementation

Word budget: ~4500–5400 words

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

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

3.1 Repository Overview

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

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

```
front_end/
middle_end/
program_config/
relooper/
fmt_indented.rs
id.rs
lib.rs
main.rs
preprocessor.rs
```

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

3.2 System Architecture

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

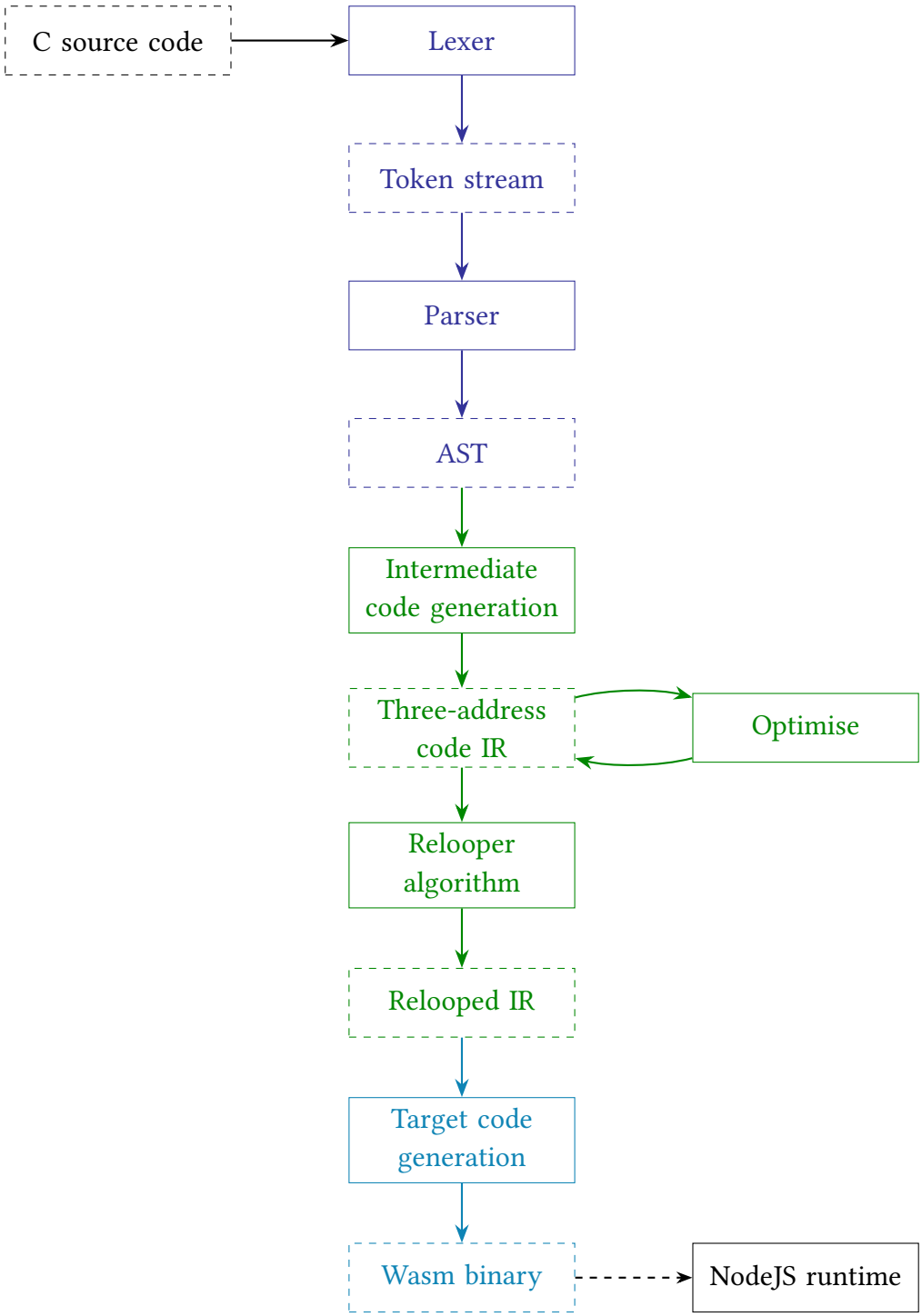


Figure 3.1: Project structure, highlighting the front end, middle end, and back end.

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

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

3.3 Front End

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

3.3.1 Preprocessor

I used the GNU C preprocessor (`cpp`) [4] to handle any preprocessor directives in the source code, for example macro definitions. However, since I am not supporting 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 to 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.

3.3.2 Lexer

The grammar of the C language is mostly context-free, however the presence of `typedef` definitions make it context-sensitive [5, Section 5.10.3]. For example, the statement in Listing 3.1 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.

```
foo (bar);
```

Listing 3.1: An example of `typedef` name ambiguity in C.

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

The lexer is implemented as a finite state machine. Figures 3.2 and 3.3 highlight portions of the machine; the remaining state transition diagrams can be found in Appendix A. The diagrams show the input character, as a regular expression, along each transition arrow. (Note: in a slight abuse of regular expression notation, the dot character '`.`' represents a literal full stop character, and the

[1] The brackets will be ignored.

backslash character '\' represents a literal backslash.) It is assumed that when no state transition is shown for a particular input, the end of the current token has been reached. Transition arrows without a prior state are the initial transitions for the first input character. Node labels represent the token that will be emitted when we finish in that state.

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

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

Figure 3.2 shows the finite state machine for lexing number literals. This handles all the different forms of number that C supports: decimal, binary, octal, hexadecimal, and floating point. (Note: the states leading to the ellipsis token are shown for completeness, even though the token is not a number literal, since they share the starting dot state.)

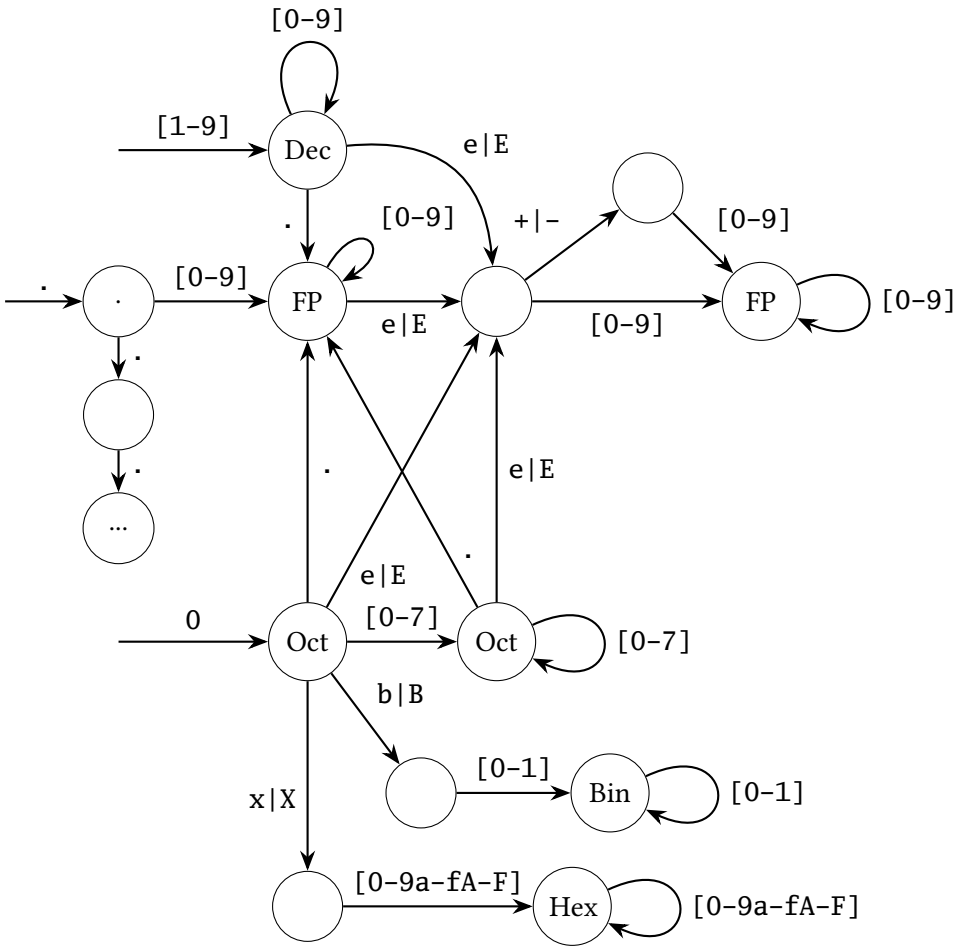


Figure 3.2: Finite state machine for lexing number literals.

Figure 3.3 shows the finite state machine for lexing identifiers and **typedef** names. This is where we handle the ambiguity introduced into the language. Every time we consume another character of an identifier, we check whether the current name (which we have stored in the string buffer) matches either a keyword of the language of a **typedef** name we have encountered this far. (Keywords are given a higher priority of matching.) If a match is found, we move to the corresponding state, represented by the ϵ transitions (since no input is consumed along these transitions). When we reach the end of the token, the three states will emit the corresponding token, either an identifier, keyword, or **typedef** name token respectively. When we emit a **typedef** name token, the lexer pushes it to an array of all the type names that have been declared this far in the program, so we can match future identifiers against it.

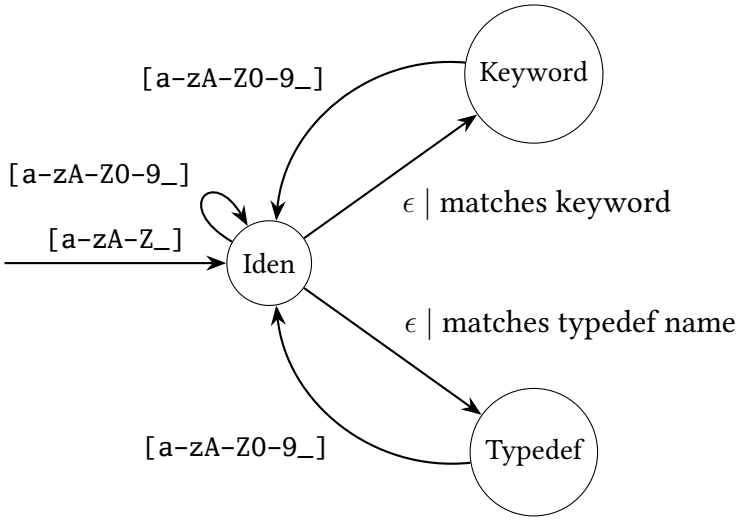


Figure 3.3: Finite state machine for lexing identifiers.

3.3.3 Parser

Talk about my `interpret_string` implementation, to handle string escaping. Implemented using an iterator.
- created AST representation
Talk about structure of my AST
Talk about how I parsed type specifiers into a standard type representation. Used a bitfield to parse arithmetic types, cos they can be declared in any order.

I used the LALRPOP parser generator [6] to generate parsing code from the input grammar I wrote. The Microsoft's C Language Syntax Summary [7] and C: A Reference Manual [5] were very useful references to ensure I captured the subtleties of C's syntax when writing my grammar. My grammar is able to parse all of the core features of the C language, omitting some of the recent language additions. I chose to make my parser handle a larger subset of C than the compiler as a whole supports; the middle end rejects or ignores nodes of the AST that it doesn't handle. For example, the parser handles storage class specifiers (e.g. **static**) and type qualifiers (e.g. **const**), and the middle end simply ignores them.

A naive grammar for C (Listing 3.2) contains an ambiguity around **if**/**else** statements [5, Section 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 don't use curly brackets, if can be ambiguous which **if** an **else** belongs to. This is known as the dangling else problem.

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

Listing 3.2: Ambigious **if**/**else** grammar.

An example of the dangling else problem is shown in Listing 3.3. According to the grammar in Listing 3.2, there are two possible parses of this program. Either the **else** belongs to the inner or the outer **if** (Listing 3.4).

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

Listing 3.3: Example of the dangling else problem.

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

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

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

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

Listing 3.4: Possible parsings of Listing 3.3.

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 [8]. Listing 3.5 shows how we introduce this into our 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 we allowed open statements inside the **if** clause of an **open-stmt**, then **open-stmt** and **closed-stmt** would no longer be disjoint, and the grammar would be ambiguous. This is because we wouldn't be able to use the above method for finding the **else** that belongs to the outer **open-stmt**.

```
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 3.5: Using open and closed statements to solve the dangling else problem

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

Listing 3.6 is an example of the LALRPOP syntax for addition expressions. The left-hand side of the => describes the grammar rule, and the right-hand side is the code to generate an `Expression` node. Terminals are represented in double quotes; these are defined to map to the tokens emitted by the lexer. Non-terminals are represented inside angle brackets, with their type and a variable name to use in the AST generation code.

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

(a) The grammar rule for addition expressions.

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

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

Listing 3.6: In LALRPOP, the AST generation and grammar code are combined.

LALRPOP also allows macros to be defined, which allow the grammar to be written in a more intuitive way. For example, I defined a macro to represent a comma-separated list of non-terminals (Listing 3.7). The macro has a generic type `T`, and automatically collects the list items into a `Vec<T>`, which can be used by the rules that use the macro.

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

Listing 3.7: LALRPOP macro to parse a comma-separated list of non-terminals.

3.4 Middle End

Give an overview of the middle end

3.4.1 Intermediate Code Generation

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

I defined a custom three-address code intermediate representation (IR). The IR contains both the program instructions and necessary metadata, such as variable type information. The instructions and metadata are contained in separate sub-structs within the main IR struct, which enables the metadata to be carried forwards through stages of the compiler pipeline while the instructions are transformed at each stage. In the stage of converting the AST to IR code, the instructions and metadata are encapsulated in the `Program` struct.

Many objects in the IR require unique IDs, such as variables and labels. I created a `Id` trait to abstract this concept, together with a generic `IdGenerator` struct (Listing 3.8). The ID generator internally tracks the highest ID that has been generated so far, so that the IR can create IDs as necessary without needing to know anything about their implementation. IDs are generated inductively: each ID knows how to generate the next one.

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

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

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

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

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

Listing 3.8: Implementation of the `Id` trait and `IdGenerator`.

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

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

In an object-oriented language, this would be achieved by encapsulating the methods in an object and using private state inside the object. Rust, however, is not object oriented, and I believe this approach also offers more modularity and flexibility. Firstly, the context information itself is encapsulated inside its own data structure, allowing methods to be implemented on it that gives calling functions access to exactly the context information they need. Also, it allows separation of the different functions in the middle end, rather than constraining them to all needing to be within one class.

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

as the AST is traversed. `context` is the context object described above, which makes gets passes through the functions recursively so the compiler always has access to relevant contextual information.

The core of converting statements and expressions to IR code is matching the type of AST node, and generating IR instructions according to the structure of the statement, recursing into sub-statements and -expressions. The case for a **`while`** statement is shown in Listing 3.10; 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.

```
fn convert_statement_to_ir(stmt, program, context) {
    instrs = []
    match stmt {
        While(condition, body) => {
            Create new labels for start and end of loop
            Push new loop context to Context object
            instrs += start of loop label
            instrs += convert_expression_to_ir(condition, program, context)
            instrs += BranchIf(condition false, branch to end of loop)
            instrs += convert_statement_to_ir(body, program, context)
            instrs += Branch(start of loop label)
            instrs += end of loop label
            Pop loop context from Context object
        }
        ... other AST statement nodes ...
    }
    return instrs
}
```

Listing 3.10: Pseudocode for the `convert_statement_to_ir()` function.

TODO talk about the more complex cases, eg. switch statements, variable declarations, function declarations

3.4.2 The Relooper Algorithm

cite Emscripten [9]

3.5 Back End: Target Code Generation

3.6 Runtime Environment

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

3.7 Optimisations

3.7.1 Unreachable Procedure Elimination

3.7.2 Tail-Call Optimisation

Defn of tail-call optimisation
Why do the optimisation

3.8 Summary

4

Evaluation

Word budget: ~2000–2400 words

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

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

4.1 Success Criteria

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

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

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

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

4.2 Correctness Testing

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

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

Programs of the first type are not strictly unit tests by the standard definition. Unit tests verify the functionality of individual units of source code, in isolation from the rest of the application. Each test should test one particular behaviour of that unit, and should be independent from the rest of the program's functionality [10]. My test programs don't test an isolated behaviour of the compiler's source code. Instead, they test a single behaviour of the C source code that is being compiled (for example, dereferencing a pointer), verifying that the compiler pipeline maintains the correct behaviour. This allowed me to trace bugs to the units of code that transformed that particular construct.

Examples of 'full programs' include Conway's Game of Life [11], calculating the Fibonacci numbers (recursively), and finding occurrences of a substring in a string. Six of the test programs I used were sourced from Clib [12], which contains many small utility packages for C. Those that I used had no external dependencies, and were useful in verifying that my compiler worked for other people's code as well as my own. Clib is licensed under the MIT license, which permits use of the software "without restriction". The remaining five full programs and all 18 unit test programs were written by myself.

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

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

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

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

[1] GCC version 11.3.1.

4.3 Impacts of Optimisations

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

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

4.3.1 Unreachable Procedure Elimination

In the context of this project, unreachable procedure elimination mainly has benefits in removing unused standard library functions from the compiled binary. When a standard library header is included in the program, the preprocessing stage inserts the entire code of that library[2]. If the program only uses one or two of the functions, most of them will be redundant. Unreachable procedure elimination is able to safely remove these, resulting in a smaller binary.

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

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

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

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

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

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

4.3.2 Tail-Call Optimisation

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

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

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

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

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

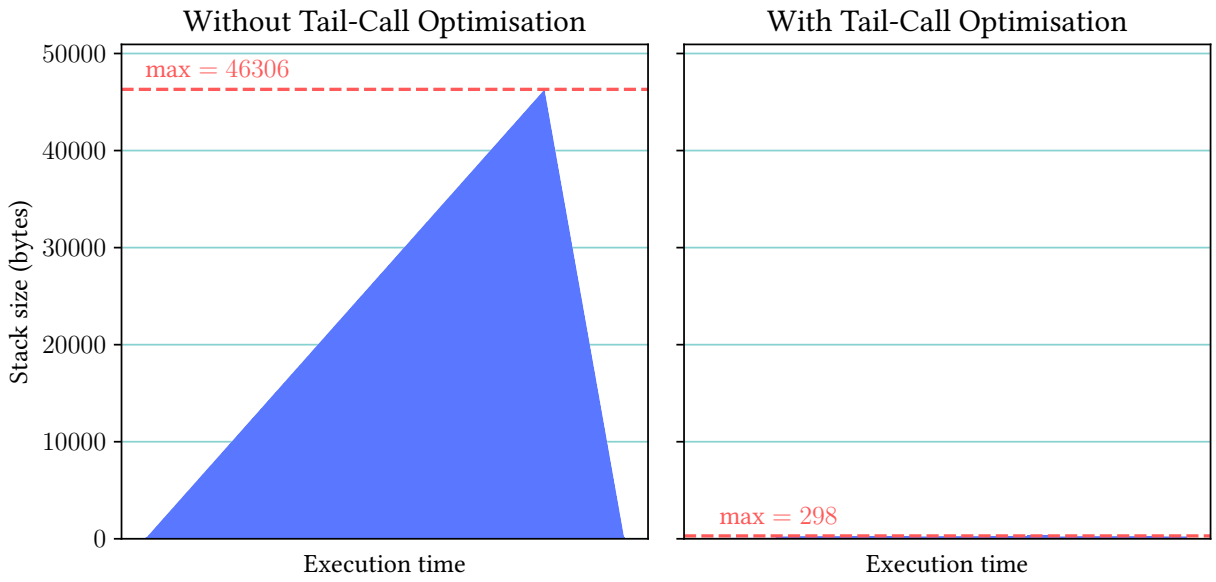


Figure 4.1: Stack usage for calling `sum(500, 0)` (see Listing 4.1)

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

4.3.3 Optimised Stack Allocation Policy

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

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

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

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

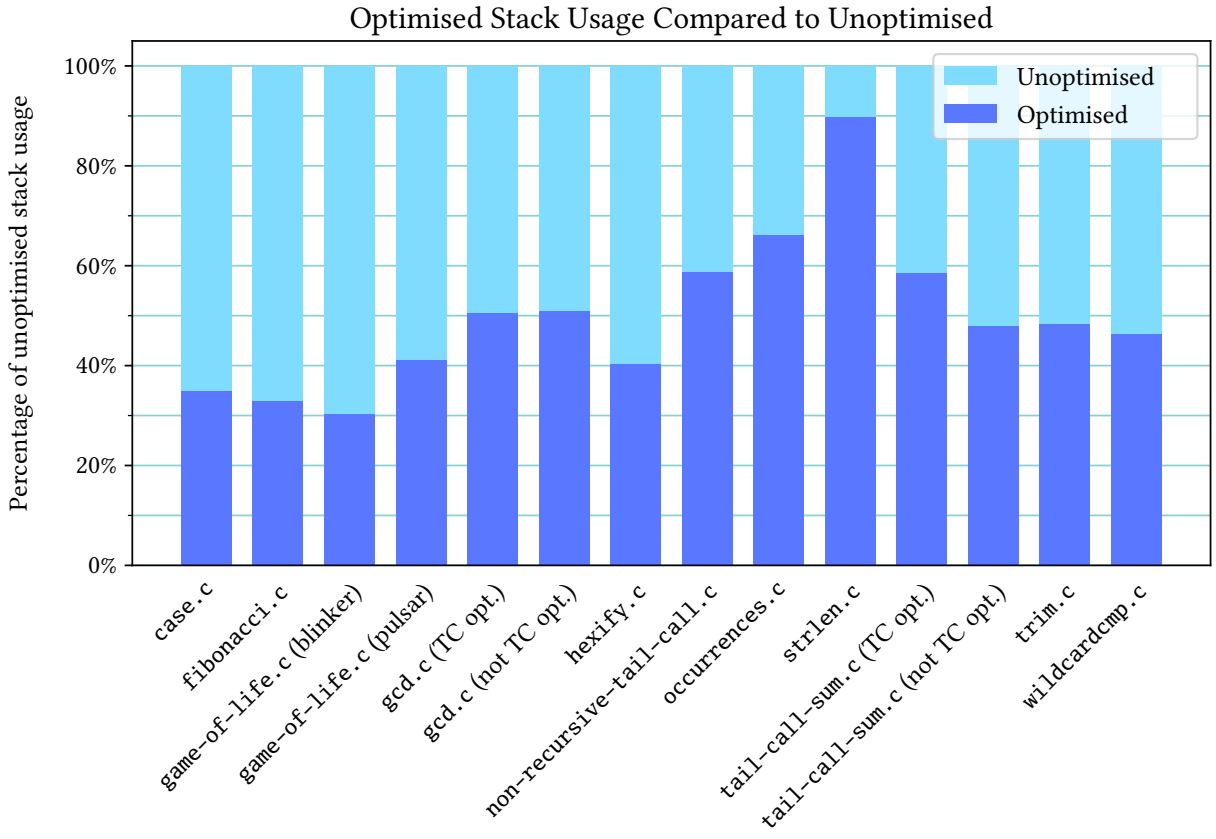


Figure 4.2: Comparing optimised stack usage to unoptimised stack usage. Shorter bars represent greater improvement.

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

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

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

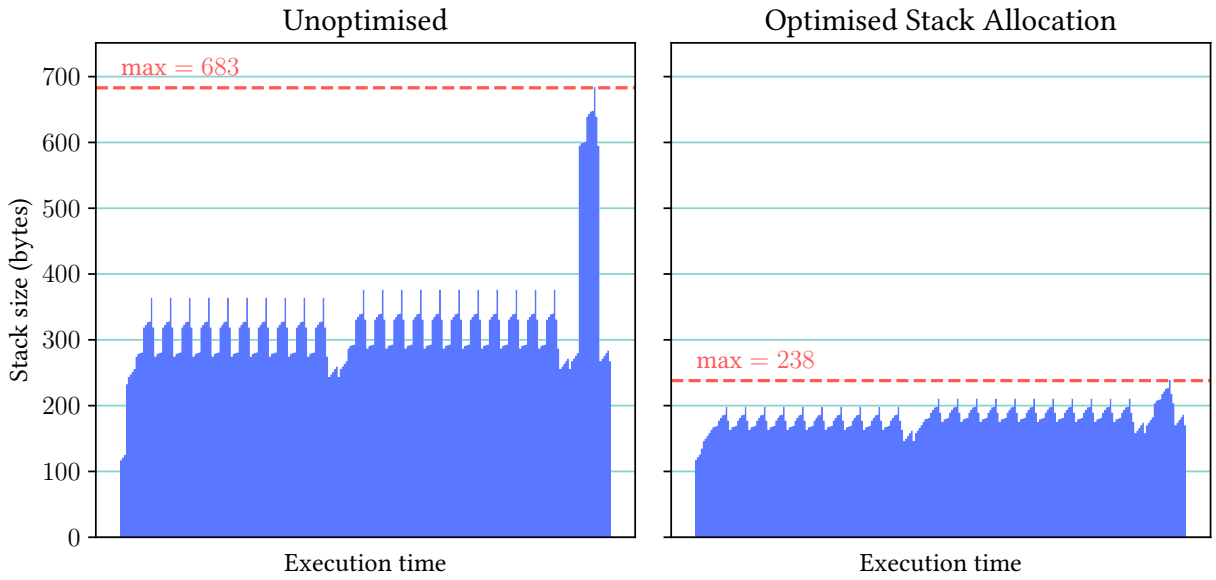


Figure 4.3: Comparing stack usage for `case.c`.

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

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

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

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

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

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

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

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

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

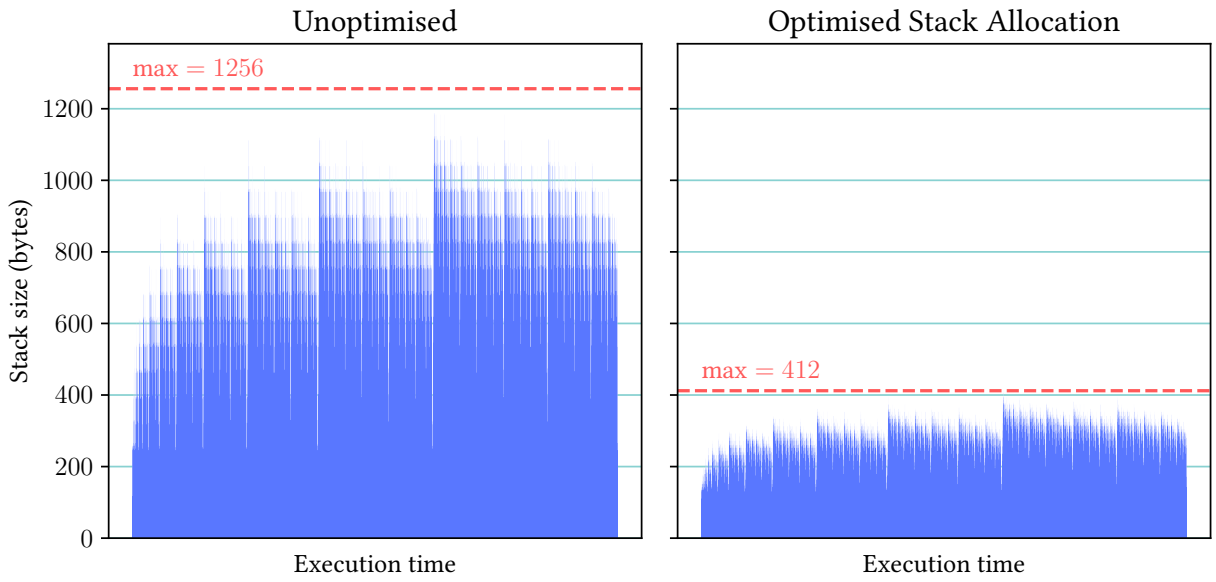


Figure 4.4: Comparing stack usage for `fibonacci.c`.

4.4 Summary

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

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

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

5 Conclusions

Word budget: ~500–600 words

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

5.1 Project Summary

5.2 Lessons Learned

5.3 Further Work

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

References:

- Relooper algorithm: [9]
- WebAssembly spec: [1]
- C grammar, Microsoft page: [7]
- Avoiding the dangling else ambiguity in LR parsers (Wiki): [13]. Wiki page references [8] (“A Final Solution to the Dangling Else of ALGOL 60 and Related Languages”)
- C reference manual book: [5]
- CLRS algorithms textbook, for interval trees: [14]
- LALRPOP tutorial/docs: [6]
- WebAssembly memory guide: [15]
- Addressing Wasm memory: [16]

Bibliography

[1] *WebAssembly Specification*. URL: https://webassembly.github.io/spec/core/index.html (visited on 10/14/2022).

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

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

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

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

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

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

[8] 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.

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

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

[11] 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.

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

[13] Wikipedia contributors. *Dangling else: Avoiding the conflict in LR parsers*. Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Dangling_else&oldid=1136442147#Avoiding_the_conflict_in_LR_parsers (visited on 10/28/2022).

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

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

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

Index

Appendix A

Lexer Finite State Machine

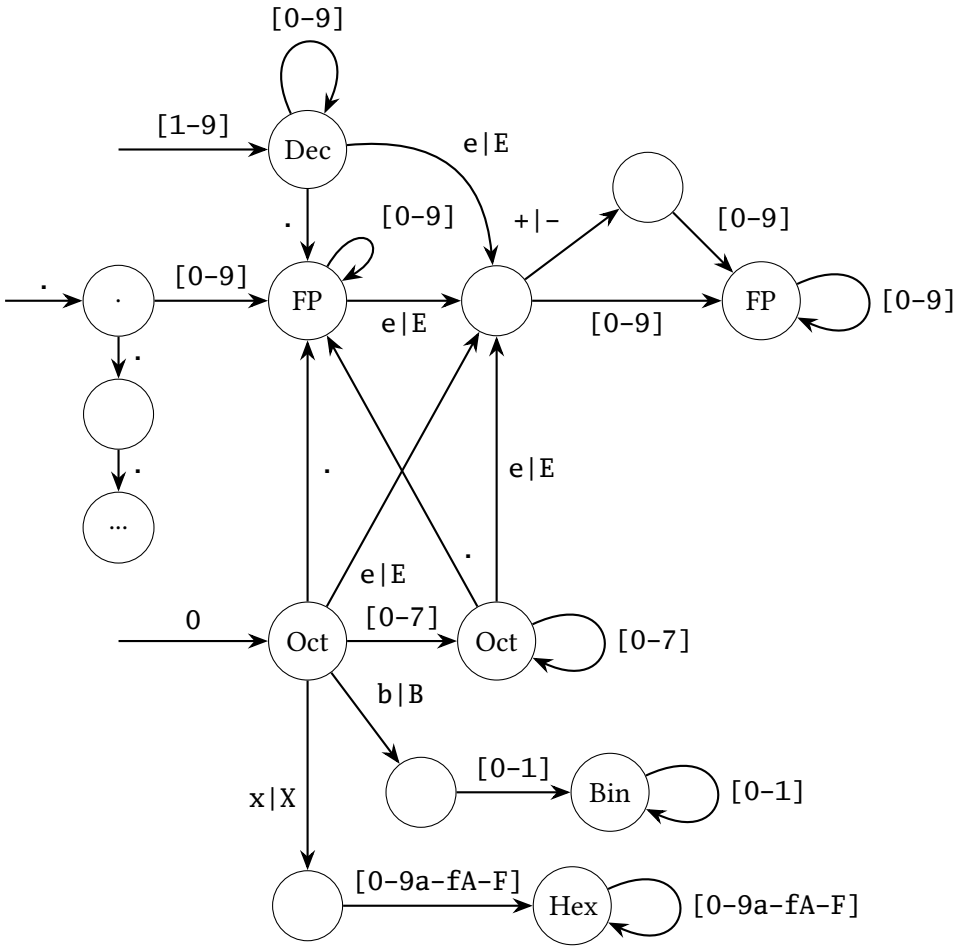


Figure A.1: Finite state machine for lexing number literals.

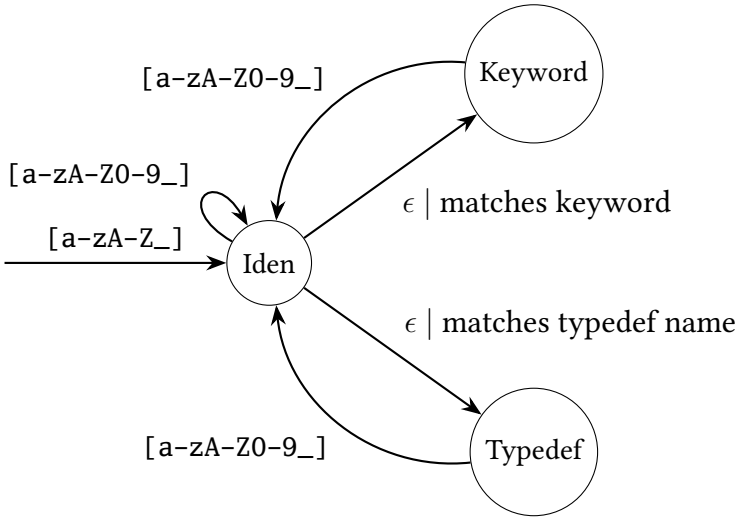


Figure A.2: Finite state machine for lexing identifiers.

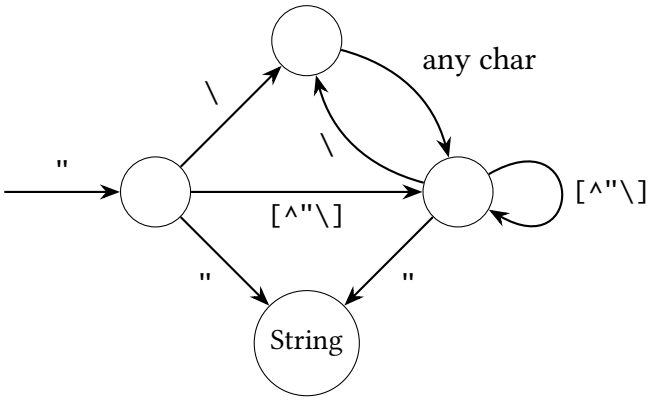


Figure A.3: Finite state machine for lexing string literals.

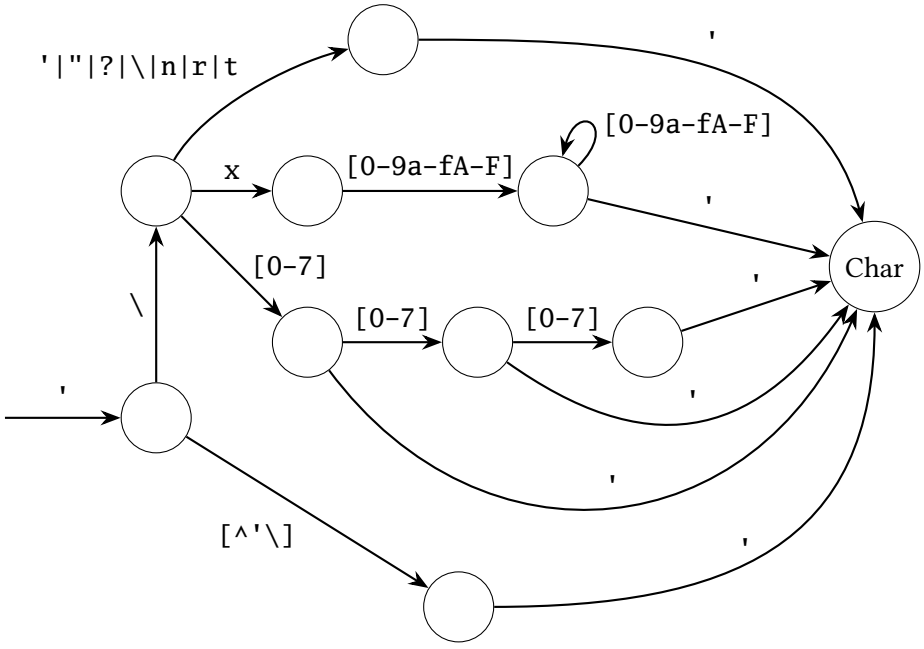


Figure A.4: Finite state machine for lexing character literals.

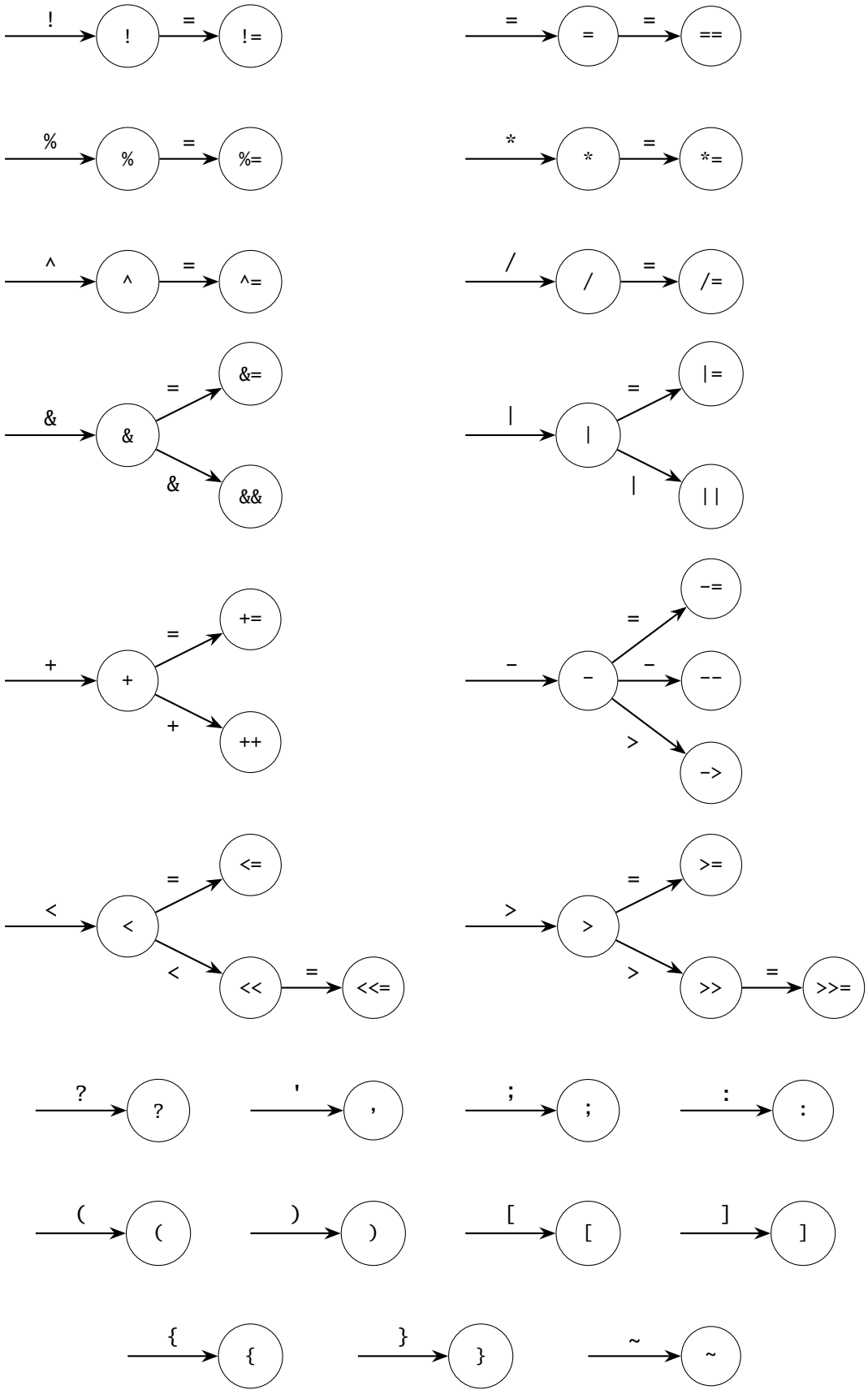


Figure A.5: Finite state machine for lexing operators.

Appendix B
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.

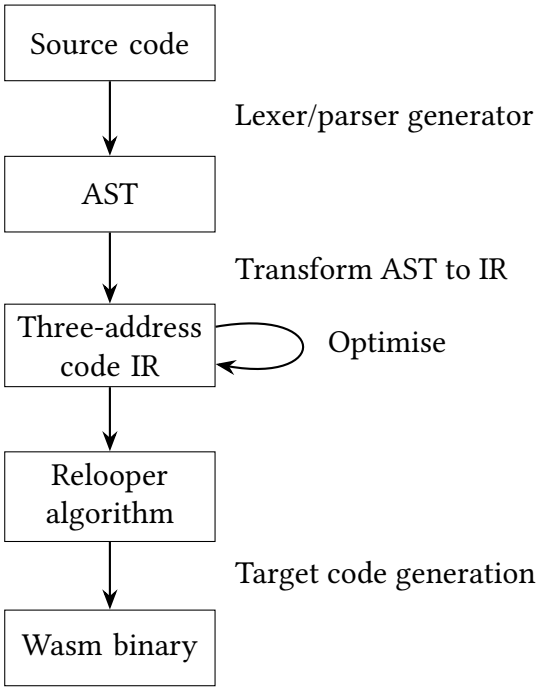
I will use a three-address code style of intermediate representation, because this lends itself to perform optimisations more easily. For example it's easier to see the control flow in three-address code

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.