1 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

1.1 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.

1.1.1 Primitive values

The primitive value types supported by WebAssembly are outlined in Table 1.1, 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 1.1 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 1.1: WebAssembly primitive types.

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

Listing 1.1: 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.

1.1.2 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 1.2).

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

Listing 1.2: WebAssembly instructions to calculate `10` – `2`.

All arithmetic instructions specify the type of value that they expect. In Listing 1.2, 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`.

1.1.3 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.

1.2 The Relooper Algorithm

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

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

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

1.2.1 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 1.1). There can be no labels or branch instructions other than at the start or end of the block. The

branch instruction may either be a conditional branch (i.e. `if (...) goto x else goto y`), or an unconditional branch.

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

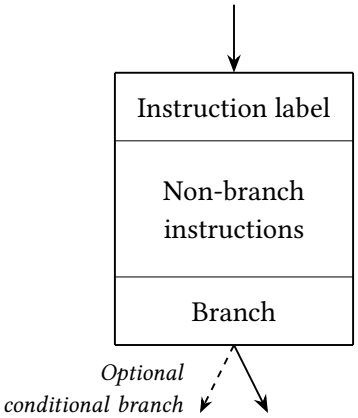

Figure 1.1: Structure of labels (Relooper algorithm input blocks).

The algorithm generates a set of structured blocks, recursively nested to represent the control flow (Figure 1.2). *Simple* blocks represent linear control flow; they contain an internal label, which contains the actual program instructions. *Loop* blocks contain an *inner* block, which contains all the labels that can possibly loop back to the start of the loop, along some execution path. *Multiple* blocks represent conditional execution, where execution can flow along one of several paths. They contain *handled* blocks to which execution can pass when we enter the block. All three blocks have a *next* block, where execution will continue.

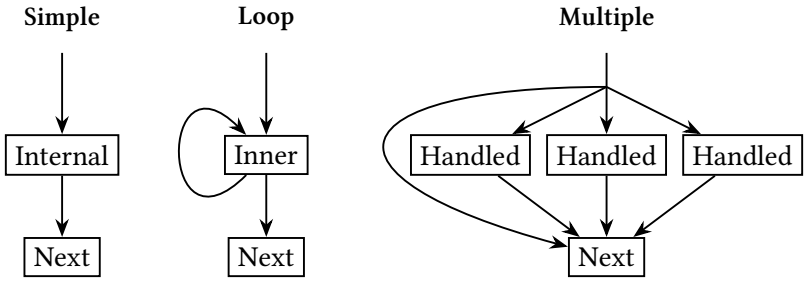

Figure 1.2: Structured blocks, generated by the Relooper algorithm.

1.2.2 Algorithm

Before describing the algorithm, we define the terms we 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.

1. Calculate the reachability of each label.
2. If there is only one entry, and execution cannot return to it, create a simple block.
3. If execution can return to all entries, create a loop block.

4 If there is more than one entry, attempt to create a multiple block. (This may not be possible.)

5 If Step 4 fails, create a loop block in the same way as Step 3.

---

function RELOOP(*labels*, *entries*)
    for $\ell \in$ *labels* **do**
        Calculate REACHABILITY(ℓ)
    end for
    if $|$*entries*$| = 1$ and execution cannot return to the entry **then**
        return CREATESIMPLEBLOCK(*labels*, the single entry)
    end if
    if execution can return to all *entries* **then**
        return CREATELOOPBLOCK(*labels*, *entries*)
    end if
    if $|$*entries*$| > 1$ **then**
        ATTEMPTTOCREATEMULTIPLEBLOCK(*labels*, *entries*)
    end if
end function

function CREATESIMPLEBLOCK(*labels*, *entry*)
    internal $\leftarrow$ *entry*
    next $\leftarrow$ RELOOP(all other labels)
    return Simple(*internal*, *next*)
end function

function CREATELOOPBLOCK(*labels*, *entries*)
    inner $\leftarrow$ RELOOP(all labels that can reach at least one entry)
    next $\leftarrow$ RELOOP(all other labels)
    return Loop(*inner*, *next*)
end function

function ATTEMPTTOCREATEMULTIPLEBLOCK(*labels*, *entries*)
    for $e \in$ *entries* **do**
        See if e reaches any labels that no other entry does
    end for
    if any e uniquely reaches labels **then**
        for each e that uniquely reaches labels **do**
            Construct a Handled block
        end for
    end if
end function

1.3 Rust

1.4 Project Strategy

1.4.1 Requirements Analysis

1.4.2 Software Engineering Methodology

1.4.3 Testing

1.5 Starting Point

1.5.1 Knowledge and experience

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

1.5.2 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