1

Implementation

Word budget: ~4500–5400 words

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

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

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

1.1 Repository Overview

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

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

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

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

1.2 System Architecture

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

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

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

1.3 Front End

The front end of the compiler consists of the lexer and the parser; it takes C source code as input and outputs an abstract syntax tree. I wrote a custom lexer, because this is necessary to support `typedef` definitions in C. I used the LALRPOP parser generator [1] to convert the tokens emitted by the lexer into an AST.

1.3.1 Preprocessor

I used the GNU C preprocessor (cpp) [2] to handle any preprocessor directives in the source code, for example macro definitions. However, since I do not support linking, I removed any `#include` directives before running the preprocessor and handled them myself.

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

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

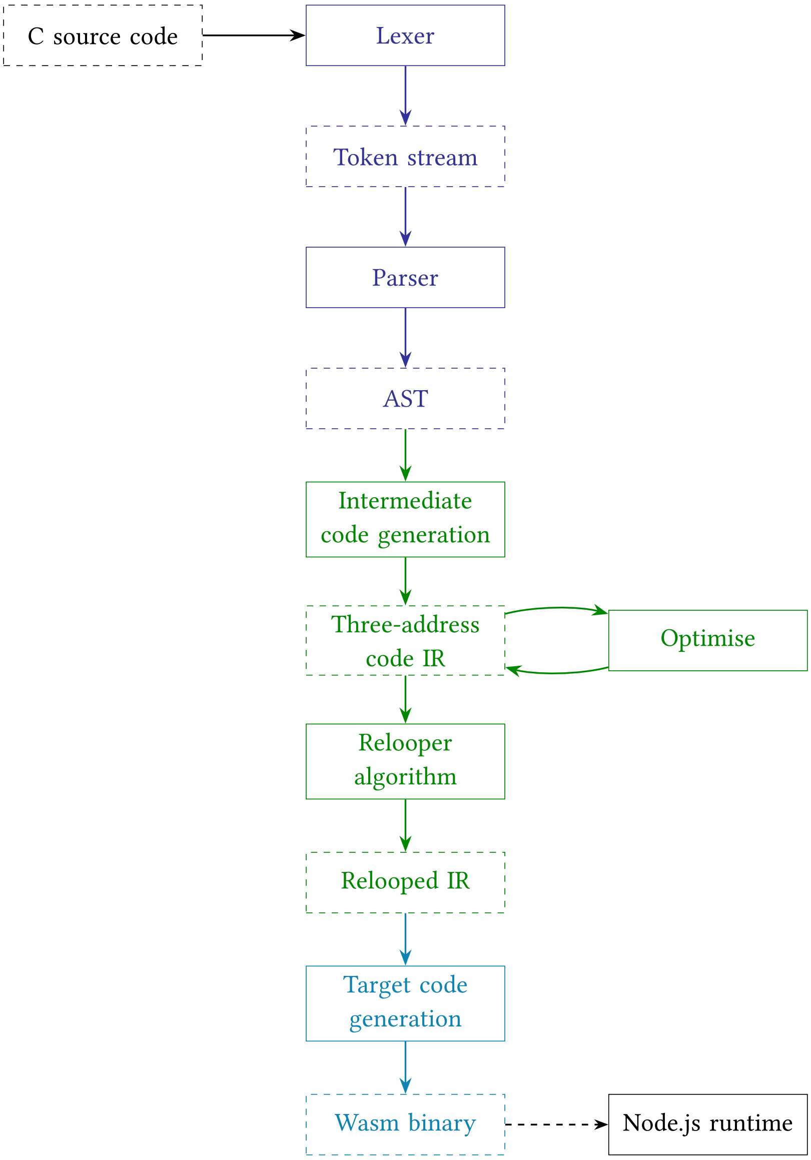


Figure 1.1: Project structure, highlighting the front end, middle end, and back end.

1.3.2 Lexer

The grammar of the C language is mostly context-free, however the presence of **`typedef`** definitions makes it context-sensitive [3, Section 5.10.3]. For example, the statement `foo (bar);` can be interpreted in two ways:

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

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

[1] The brackets will be ignored.

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

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

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

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

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

1.3.3 Parser

I used the LALRPOP parser generator [1] to generate parsing code from the input grammar I wrote. It generates an AST that I defined the structure of. Microsoft's C Language Syntax Summary [4] and C: A Reference Manual [3] were very useful references to ensure I captured the subtleties of C's syntax when writing my grammar. My grammar is able to parse all of the core features of the C language, omitting some of the recent language additions. I chose to make my parser handle a larger subset of C than the compiler as a whole supports; the middle end rejects or ignores nodes of the AST that it doesn't handle. For example, the parser handles storage class specifiers (e.g. **`static`**) and type qualifiers (e.g. **`const`**), and the middle end simply ignores them.

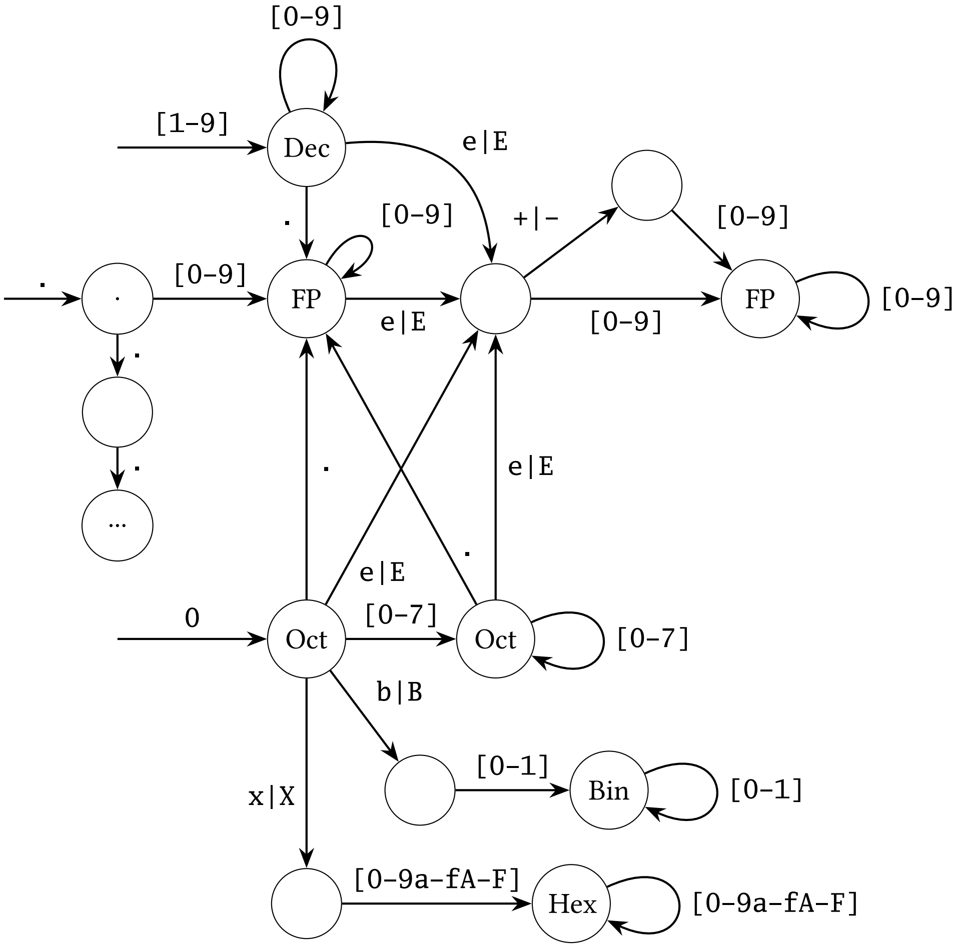


Figure 1.2: Finite state machine for lexing number literals.

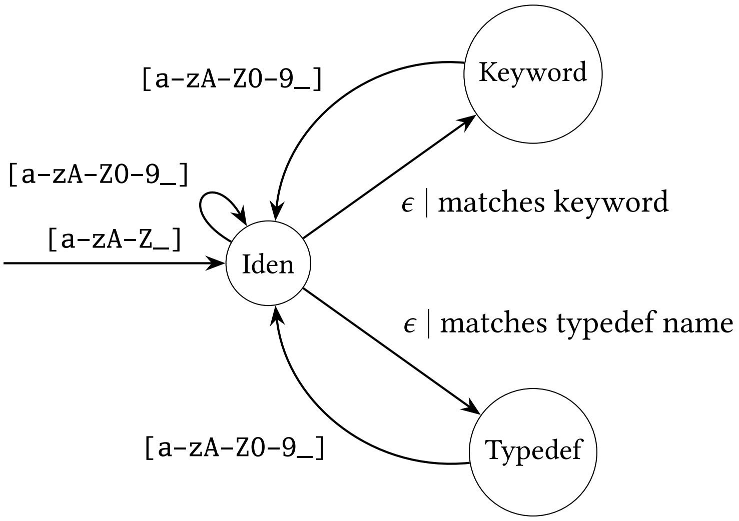


Figure 1.3: Finite state machine for lexing identifiers.

1.3.3.1 Dangling Else Ambiguity

A naive grammar for C (Listing 1.1) contains an ambiguity around **if**/**else** statements [3, 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 [5].

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

Listing 1.1: Ambigious `if`/`else` grammar.

An example of the dangling else problem is shown in Listing 1.2. According to the grammar in Listing 1.1, there are two possible parses of this program. Either the `else` belongs to the inner or the outer `if` (Listing 1.3).

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

Listing 1.2: Example of the dangling else problem.

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

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

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

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

Listing 1.3: Possible parsings of Listing 1.2.

C resolves the ambiguity by always associating the `else` with the closest possible `if`. We can encode this into the grammar with the concept of 'open' and 'closed' statements [6]. Listing 1.4 shows how I introduce this into my grammar for `if`/`else` statements. All other forms of statement must also be converted to the new structure. Any basic statements, i.e. statements that have no sub-statements, are added to `closed-stmt`. All other statements that have sub-statements, such as `while`, `for`, and `switch` statements, must be duplicated to both `open-stmt` and `closed-stmt`.

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

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

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

Listing 1.4: Using open and closed statements to solve the dangling else problem

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

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.

1.3.3.2 LALRPOP Parser Generator

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

Listing 1.5 is an example of the LALRPOP syntax for addition expressions. The left-hand side of the => describes the grammar rule, and the right-hand side is the code to generate an Expression node. Terminals are represented in double quotes; these are defined to map to the tokens emitted by the lexer. Non-terminals are represented inside angle brackets, with their type and a variable name to use in the AST generation code.

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

(a) The grammar rule for addition expressions.

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

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

Listing 1.5: In LALRPOP, the AST generation and grammar code are combined.

LALRPOP also allows macros to be defined, which allow the grammar to be written in a more intuitive way. For example, I defined a macro to represent a comma-separated list of non-terminals (Listing 1.6). The macro has a generic type T, and automatically collects the list items into a Vec<T>, which can be used by the rules that use the macro.

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

Listing 1.6: LALRPOP macro to parse a comma-separated list of non-terminals.

1.3.3.3 String Escape Sequences

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

1.3.3.4 Parsing Type Specifiers

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

1.4 Middle End

The middle end takes an AST as input, and transforms it to intermediate code. It also runs the Relooper algorithm on the IR. Optimisations are run on the IR in this stage; they are described in Section 1.7.

1.4.1 Intermediate Code Generation

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

Many objects in the IR require unique IDs, such as variables and labels. I created a `Id` trait to abstract this concept, together with a generic `IdGenerator` struct (Listing 1.7). The ID generator internally tracks the highest ID that has been generated so far, so that the IR can create as many IDs as necessary without needing to know anything about their implementation. IDs are generated inductively: each ID knows how to generate the next one.

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

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

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

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

Listing 1.7: Implementation of the `Id` trait and `IdGenerator`.

To convert the AST to IR code, the compiler recursively traverses the tree, generating three-address code instructions and metadata as it does so. At the highest level, the AST contains a list of statements. For each of these, the compiler calls `convert_statement_to_ir(stmt, program, context)`. `program` is the mutable intermediate representation, to which instructions and metadata are added as the AST is traversed. `context` is the context object described in Section 1.4.2, which passes relevant contextual information through the functions recursively.

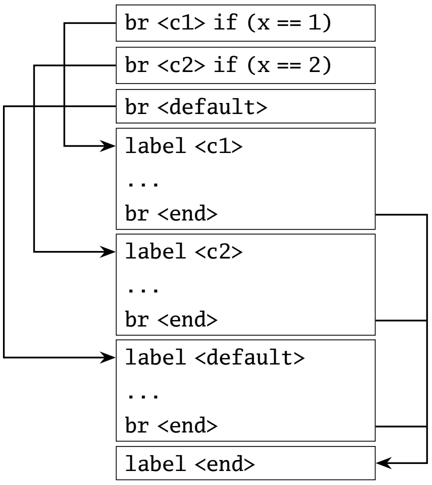
The core of converting a statement or expressions to IR code is pattern matching the AST node, and generating IR instructions according to its structure, recursing into sub-statements and expressions. The case for a **`while`** statement is shown in Listing 1.8; the labels and branches to execute a while loop are added, and the instructions to evaluate the condition and body are generated recursively. Other control-flow structures are similar.

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

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

Listing 1.8: Pseudocode for the `convert_statement_to_ir()` function.

to the next block, and there may be a **`default`** block. I handled this by first generating instructions for each case block, and storing them in a switch context until all blocks have been seen. We then put conditional branches to each case block at the start of the switch statement, followed by each of the block instructions. By doing this, there is no direct fall-through between any blocks; instead, a block that falls through to the next block will end in a branch instruction to the start of that block. This also allows **`default`** blocks to be easily handled; we just add an unconditional branch to the default block after all the conditional branches. If there is no **`default`** block, the conditional branches are followed by an unconditional branch to the end of the **`switch`** statement. Figure 1.4 shows the structure of the generated instructions for **`switch`** statements.



```
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 1.4: IR instructions generated for **`switch`** statements.

Declaration statements have a lot of different cases, each of which I had to handle separately. For example, variables may be declared without being initialised with a value.

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

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

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

1.4.2 Context Object Design Pattern

Throughout the middle and back ends, I used a design pattern of passing a context object through all the function calls. For example, when traversing the AST to generate IR code, the `Context` struct in Listing 1.9 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 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 1.9: The context data structure used when converting the AST to IR code.

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

1.4.3 Types

The following types are supported by the IR, mirroring the types supported by the C language [3, ch. 5]. `Ux` and `Ix` types represent unsigned and signed x-bit integers, respectively. Enumeration types (enums) are supported; their values are encoded as `U64`s. I followed the standard implementation convention for the bit size of `char`s, `short`s, `int`s, and `long`s; 8, 16, 32, and 64 bits respectively[1].

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

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

I implemented the ISO C unary and binary conversions for types [3, pp. 174–176]. They are applied before unary and binary operations respectively. Unary conversion reduces the number of types an operand can be. Smaller integer types are promoted to `I32`/`U32` appropriately, and $\texttt{Array}(T, \_)$ types are converted to $\texttt{Pointer}(T)$. Binary conversions make sure that both operands to a binary operation are of the same type. First, the unary conversions are applied to each operand individually. Then, if both operands are an arithmetic type, and one operand is a smaller type than the other, the smaller type is converted to the larger type. This includes integer types being promoted to float types.

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

1.4.4 The Relooper Algorithm

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

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

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

[1] The C specification doesn’t specify the exact bit widths, only the minimum size.

[2] `I32` is used to represent booleans.

[3] The term ‘label’ is overloaded here, though the two concepts are related; a ‘label instruction’ refers to the intermediate code instruction, whereas a ‘label’ is a basic block that starts with a label instruction.

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

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

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

$\boxed{3}$ 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.

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 the algorithm described in ?? is used to create a Relooper block. The `IdGenerator` struct described above is used to assign each loop and multiple block a unique ID.

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

Maybe talk a bit more about the actual relooper implementation?

1.5 Back End: Target Code Generation

- Int and vector encoding?

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

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

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

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

Most instructions are a variation of this pattern. Listing 1.10 shows the code generated for an add instruction. One subtlety is that `store` instructions take the memory address as their first operand, and the value to store as their second operand. This means the address of the destination variable needs to be pushed onto the stack *before* the source operands are loaded and the operation is performed.

```
c = a + b
```

(a) Intermediate code.

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

(b) Generated WebAssembly code.

Listing 1.10: Transforming an add instruction to target code, assuming `a` and `b` are variables of type `i64`.

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

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

1.5.1 Memory Layout

One of the other main functions of the back end is to generate code that manages the memory layout. A large part of this is the function call stack; pushing new stack frames when functions are called, and popping them when functions return. Figure 1.5 shows the memory layout I defined for the compiler.

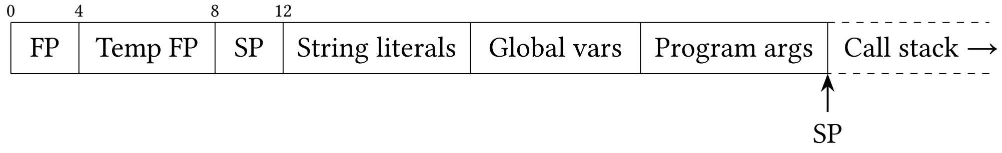


Figure 1.5: Memory structure, with addresses increasing to the right.

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

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

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

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

Programs may accept command-line arguments, which are stored into memory by the runtime environment. This is described in more detail in Section 1.6 below.

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

1.5.2 Stack Frame Operations

Whenever a function is called, a new stack frame is constructed at the top of the stack in memory. Figure 1.6 shows the structure of each stack frame.

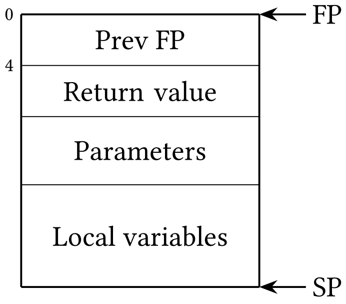


Figure 1.6: Stack frame layout, with addresses increasing downwards.

I defined callee/caller conventions for pushing and popping stack frames. The caller is responsible for constructing the frame with the previous FP value, space for the return address, and any function parameters. The caller also deallocates the stack frame once the function returns. The callee is responsible for allocating space for its local variables on the stack (discussed in Section 1.5.3 below).

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

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

Using the temporary FP here is necessary because if we directly wrote to the FP, we would not be able to copy any variables in as parameters. (Local variables have FP-relative addresses.) We also cannot wait to save the new FP address until after we have copied the variables, because by then the SP has been moved and no longer points to the start of the stack frame. (The SP is updated every time a value is stored on top of the stack.)

Popping a stack frame is simpler:

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

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

1.5.3 Local Variable Allocation

Initially I implemented a naive variable allocation strategy. As an extension, I implemented a more optimal allocation strategy, described in Section 1.7.3.

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

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

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

1.6 Runtime Environment

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

The purpose of the runtime environment is to provide an interface between the WebAssembly binary and the system, allowing it to be executed. I chose to use Node.js [7], because this allows the binary to be run locally. The other option would be to use JavaScript within a web browser, which would be the primary reason to use WebAssembly; however, for the purposes of this project that approach would have added unnecessary complexity.

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

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

Once the module has been instantiated, the runtime stores the command-line program arguments into memory. As is standard for C programs, the first argument is always the name of the program being run (i.e. the name of the WebAssembly binary). Arguments are passed to the `main` function as an array of **`char`** pointers (`argv`), plus the argument count (`argc`). To facilitate this, the runtime first allocates space for the array of pointers, at the compile-time known address immediately after the global variables. After the pointer array, the actual argument values are stored (as strings), and each corresponding pointer is set (Figure 1.7). The runtime then sets the SP to immediately after the last argument.

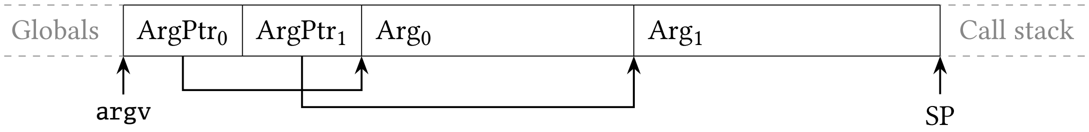


Figure 1.7: Memory structure of program arguments, with addresses increasing to the right. Here, `argc` = 2.

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

1.7 Optimisations

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

1.7.1 Unreachable Procedure Elimination

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

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

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

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

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

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

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

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

1.7.2 Tail-Call Optimisation

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

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

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

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

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

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

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

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

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

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

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

1.7.2.2 Approach 2: Transforming Recursion to Iteration

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

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

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

1.7.3 Stack Allocation Policy

My initial variable allocation policy is described in Section 1.5.3 above. Each variable is allocated a disjoint address range in memory. However, this is very inefficient since most variables are temporary variables that are only used once, and many of them are not used at the same time as each other (i.e. they never *clash*).

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

The optimised allocation is performed in the following steps:

$\boxed{1}$ Remove dead variables.

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

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

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

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

1.7.3.1 Live Variable Analysis

An instruction flowgraph is similar to the function call graph used in Section 1.7.1 above, but on the level of individual instructions within a function. Each node in the graph is a single instruction, and successors of a node are any instructions that can be executed as the next instruction along some execution path. For example, branch instructions will have multiple successors.

The flowgraph is generated by recursing through the program blocks (the output of the Relooper algorithm, Section 1.4.4), creating a node for each instruction, and creating edges along all possible control flow paths.

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

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

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

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

To implement LVA, we keep iterating over every instruction in the flowgraph, applying the above equation to update the set of live variables, until there are no more changes. Initially the set of variables live at each instruction is the empty set. The algorithm is guaranteed to find the smallest set of live variables (it does not add unnecessary overapproximations). Listing 1.11 shows my implementation.

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

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

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

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

Listing 1.11: Live variable allocation implementation.

1.7.3.2 Generating the Clash Graph

The clash graph is generated from the results of LVA. Any variables that are simultaneously live *clash*; i.e. they cannot be allocated to overlapping addresses, because they are in use at the same time. Listing 1.12 describes how the clash graph is generated.

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

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

Listing 1.12: Algorithm to generate the clash graph from LVA.

1.7.3.3 Allocating Variables from the Clash Graph

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

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

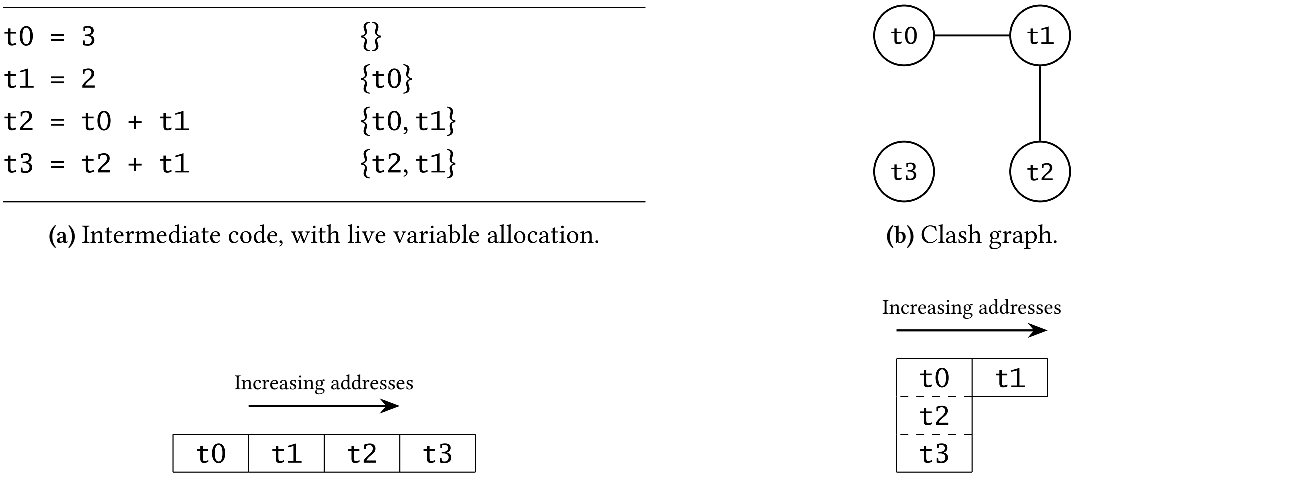
I modified the register allocation method to the following heuristic for variable allocation:

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

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

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

Figure 1.8 shows an example of the impact this optimisation has. In the top left is the intermediate code for which memory is being allocated. The live variables at each instruction—calculated by LVA—are shows to the right of each line. In the top right is the clash graph, showing which variables cannot be allocated to the same memory. The bottom left shows how the naive allocation policy would allocate the variables in memory; sequentially and non-overlapping. In the bottom right is the result of the optimised allocation policy: variables t0, t2, and t3 all use the same memory location because they are never live at the same time. Notice how all three variables use the lowest location, rather than, for example, t3 overlapping with t1. This is due to the heuristic of always allocating to the lowest valid address.



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

(a) Intermediate code, with live variable allocation.

(b) Clash graph.

(c) Naive allocation policy.

(d) Optimised allocation policy.

Figure 1.8: Example of optimised stack allocation policy.