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/	NodeJS 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.

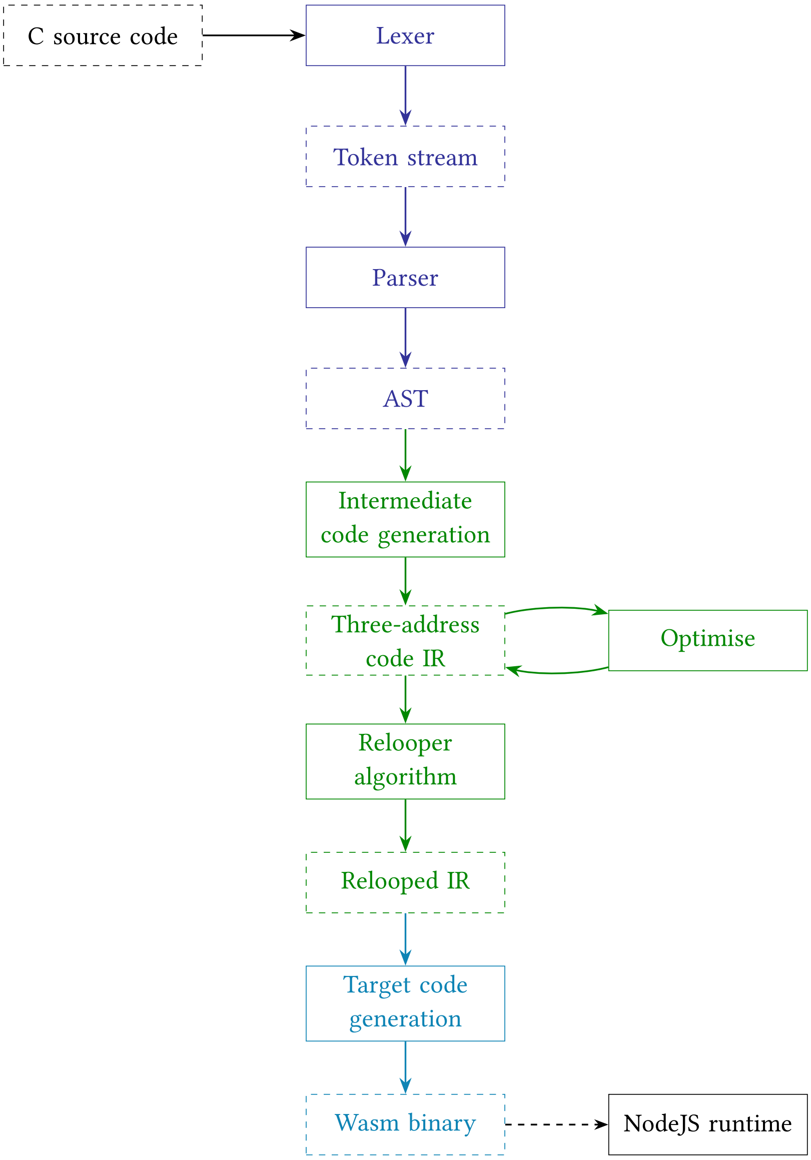


Figure 1.1: Project structure, highlighting the front end, middle end, and back end.

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.

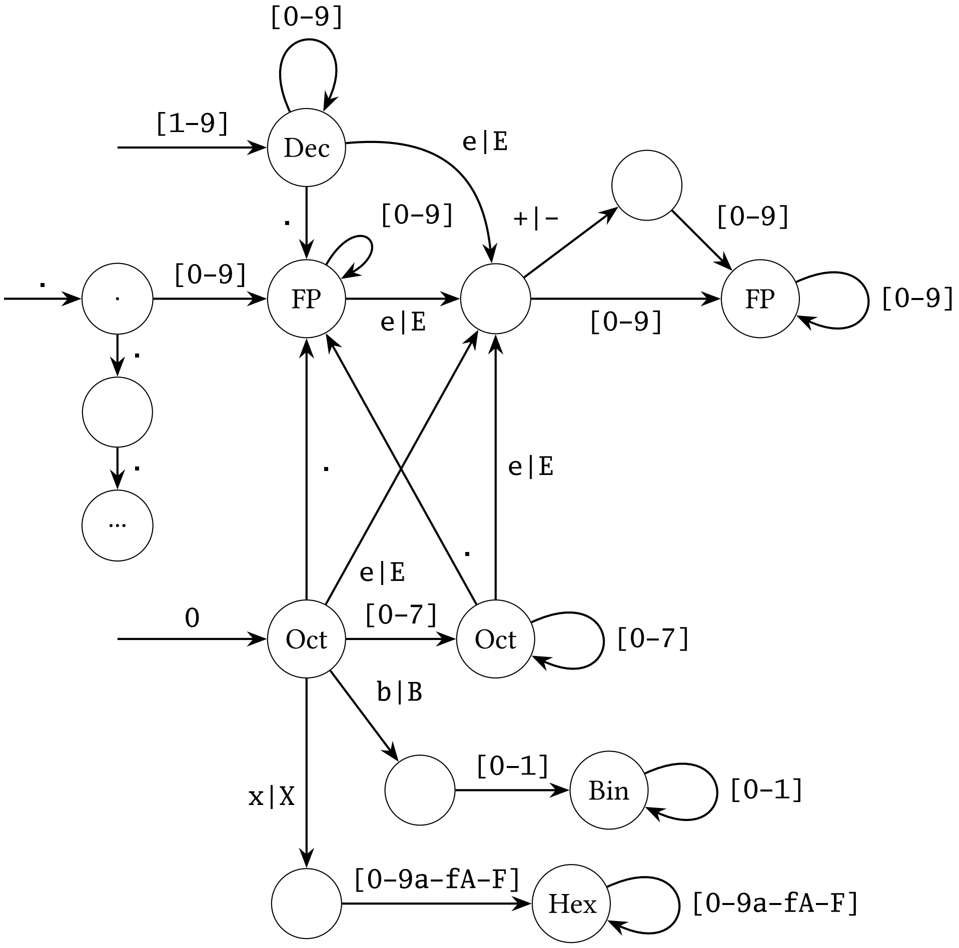


Figure 1.2: Finite state machine for lexing number literals.

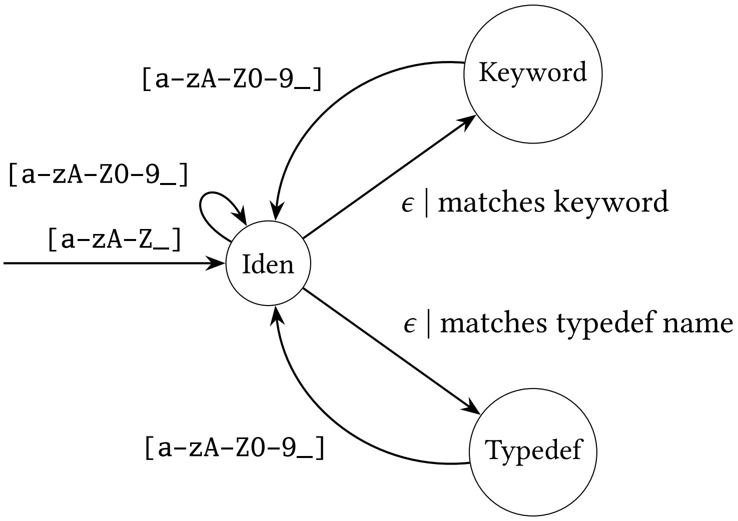


Figure 1.3: Finite state machine for lexing identifiers.

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.

Make this an AST diagram instead?

```
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`**.

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

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

Give an overview of the middle end

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.

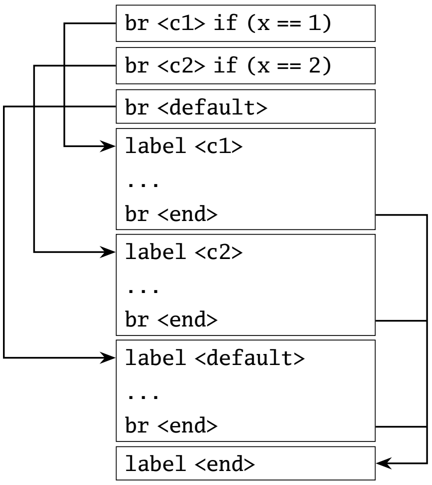
The core of converting a statement or expressions to IR code is pattern matching the AST node, and generating IR instructions according to its structure, recursing into sub-statements and expressions. The case for a **`while`** statement is shown in Listing 1.8; the labels and branches to execute a while 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 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

```
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.

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.

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

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 `Array`$(T, \_)$ types are converted to `Pointer`(T). Binary conversions make sure that both operands to a binary operation are of the same type. First, the unary conversions are applied to each operand individually. Then, if both operands are an arithmetic type, and one operand is a smaller type than the other, the smaller type is converted to the larger type. This includes integer types being promoted to float types.

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

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 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:

$\boxed{1}$ Remove any label fall-through. By this I mean any label 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, if one does not already exist, branching to that label.

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

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

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

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
...
```

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

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

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

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

TODO talk about the implementation of the actual relooper bit

1.5 Back End: Target Code Generation

1.6 Runtime Environment

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

1.7 Optimisations

1.7.1 Unreachable Procedure Elimination

1.7.2 Tail-Call Optimisation

Defn of tail-call optimisation
Why do the optimisation

1.8 Summary