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

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.

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

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

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

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.

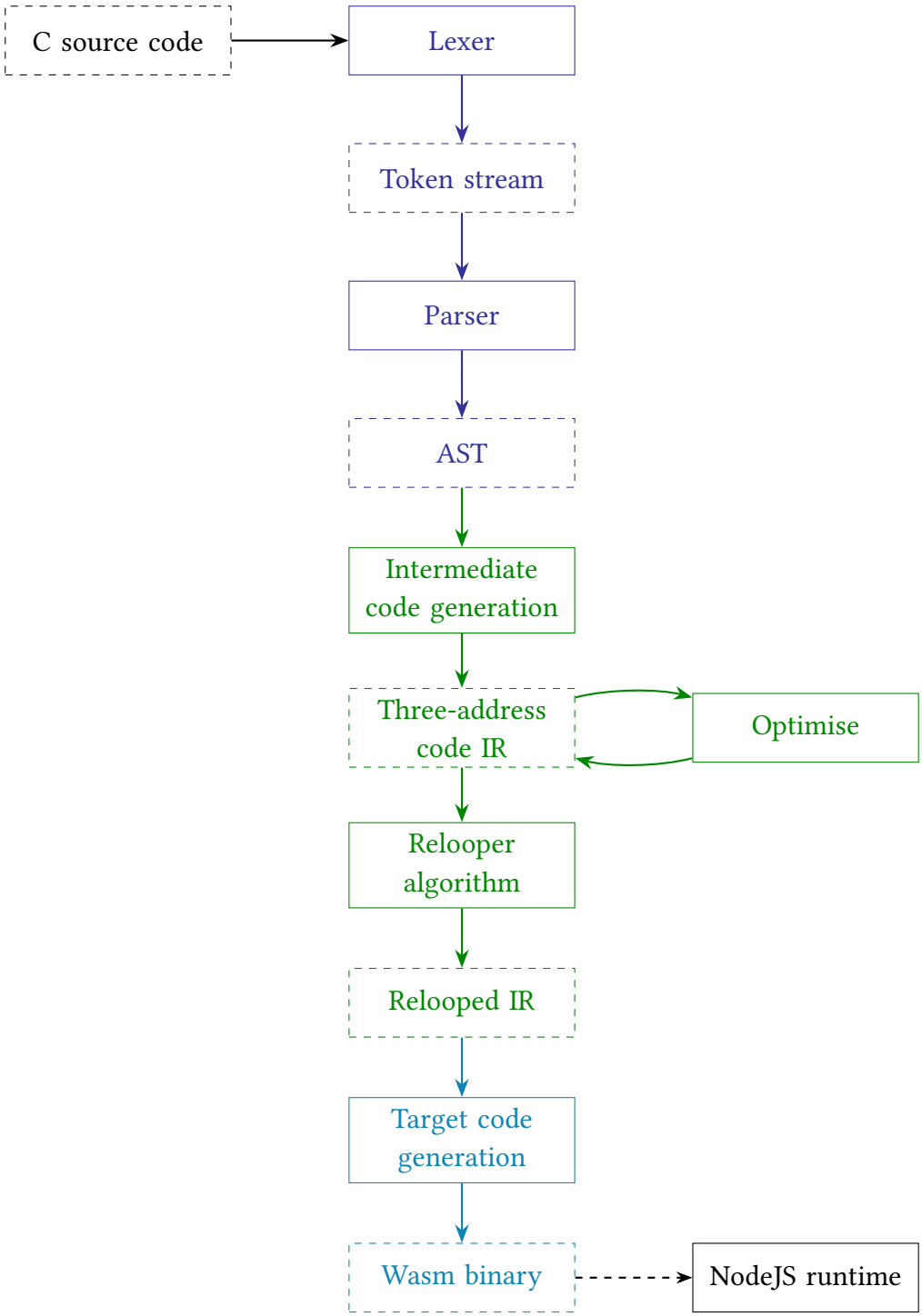

Figure 1.1: Project structure, highlighting the front end, middle end, and back end.

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

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

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 a parser generator to convert the tokens emitted by the lexer into an AST.

1.3.1 Preprocessor

I used the GNU C preprocessor (`cpp`) [1] to handle any preprocessor directives in the source code, for example macro definitions. However, since I am not supporting linking, I removed any `#include` directives before running the preprocessor, and handled them myself.

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

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

1.3.2 Lexer

The grammar of the C language is mostly context-free, however the presence of **typedef** definitions make it context-sensitive [2, Section 5.10.3]. For example, the statement in Listing 1.1 can be interpreted in two ways:

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

```
foo (bar);
```

Listing 1.1: An example of **typedef** name ambiguity in C.

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

The lexer is implemented as a finite state machine. Figures 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

[1] The brackets will be ignored.

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.)

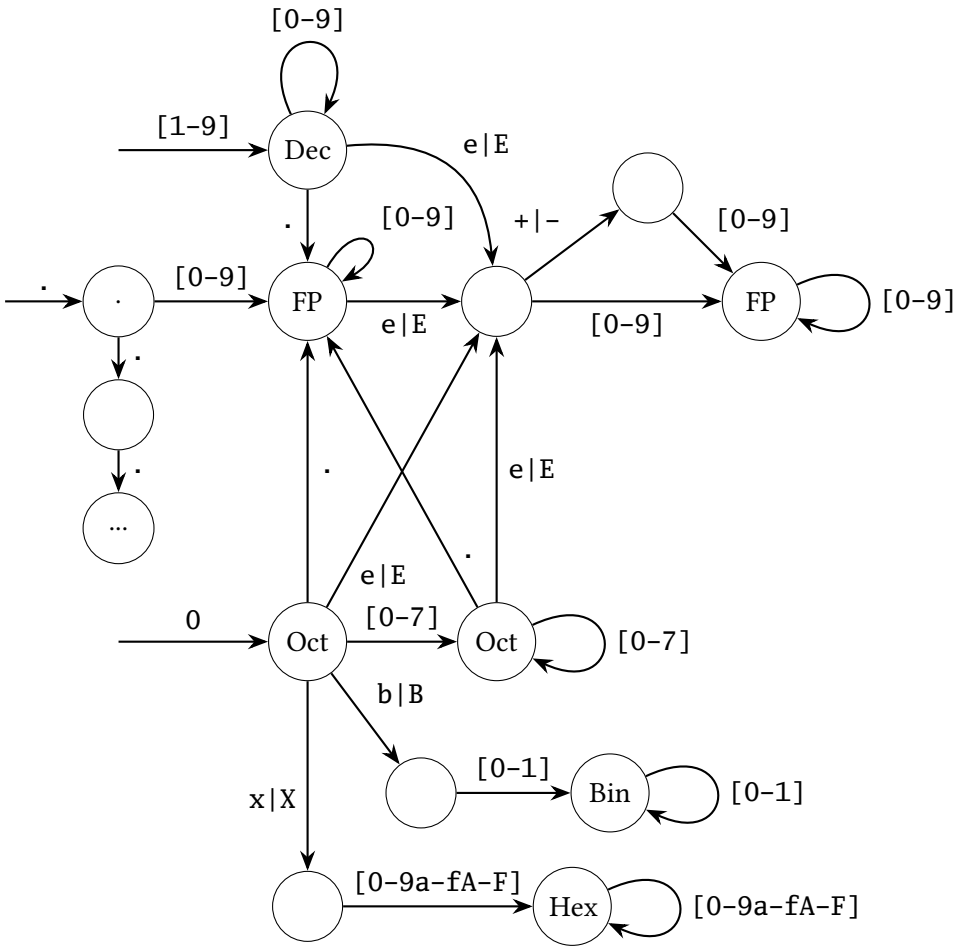

Figure 1.2: Finite state machine for lexing number literals.

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 of a **typedef** name we have encountered this far. (Keywords are given a higher priority of matching.) If a match is found, we move to the corresponding state, represented by the ϵ transitions (since no input is consumed along these transitions). When we reach the end of the token, the three states will emit the corresponding token, either an identifier, keyword, or **typedef** name token respectively. When we emit a **typedef** name token, the lexer pushes it to an array of all the type names that have been declared this far in the program, so we can match future identifiers against it.

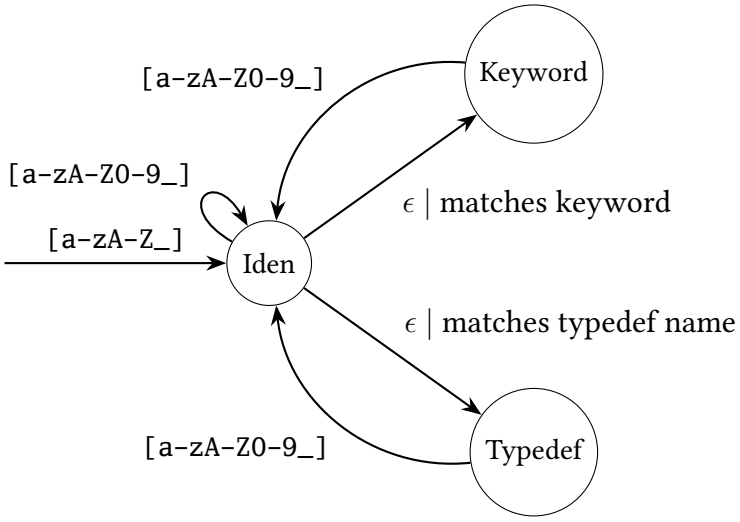

Figure 1.3: Finite state machine for lexing identifiers.

1.3.3 Parser

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

I used the LALRPOP parser generator [3] to generate parsing code from the input grammar I wrote. The Microsoft's C Language Syntax Summary [4] and C: A Reference Manual [2] were very useful references to ensure I captured the subtleties of C's syntax when writing my grammar. My grammar is able to parse all of the core features of the C language, omitting some of the recent language additions. I chose to make my parser handle a larger subset of C than the compiler as a whole supports; the middle end rejects or ignores nodes of the AST that it doesn't handle. For example, the parser handles storage class specifiers (e.g. **static**) and type qualifiers (e.g. **const**), and the middle end simply ignores them.

A naive grammar for C (Listing 1.2) contains an ambiguity around **if**/**else** statements [2, Section 8.5.2]. C permits the bodies of conditional statements to be written without curly brackets if the

body is a single statement. If we have nested **if**/**else** statements that don't use curly brackets, if can be ambiguous which **if** an **else** belongs to. This is known as the dangling else problem.

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

Listing 1.2: Ambigious **if**/**else** grammar.

An example of the dangling else problem is shown in Listing 1.3. According to the grammar in Listing 1.2, there are two possible parses of this program. Either the **else** belongs to the inner or the outer **if** (Listing 1.4).

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

Listing 1.3: Example of the dangling else problem.

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

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

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

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

Listing 1.4: Possible parsings of Listing 1.3.

C resolves the ambiguity by always associating the **else** with the closest possible **if**. We can encode this into the grammar with the concept of 'open' and 'closed' statements [5]. Listing 1.5 shows how we introduce this into our grammar for **if**/**else** statements. All other forms of statement must also be converted to the new structure. Any basic statements, i.e. statements that have no sub-statements, are added to **closed-stmt**. All other statements that have sub-statements, such as **while**, **for**, and **switch** statements, must be duplicated to both **open-stmt** and **closed-stmt**.

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

If we allowed open statements inside the **if** clause of an **open-stmt**, then **open-stmt** and **closed-stmt** would no longer be disjoint, and the grammar would be ambiguous. This is because we wouldn't be able to use the above method for finding the **else** that belongs to the outer **open-stmt**.

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

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

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

Listing 1.5: Using open and closed statements to solve the dangling else problem

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

Listing 1.6 is an example of the LALRPOP syntax for addition expressions. The left-hand side of the => describes the grammar rule, and the right-hand side is the code to generate an `Expression` node. Terminals are represented in double quotes; these are defined to map to the tokens emitted by the lexer. Non-terminals are represented inside angle brackets, with their type and a variable name to use in the AST generation code.

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

(a) The grammar rule for addition expressions.

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

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

Listing 1.6: In LALRPOP, the AST generation and grammar code are combined.

LALRPOP also allows macros to be defined, which allow the grammar to be written in a more intuitive way. For example, I defined a macro to represent a comma-separated list of non-terminals (Listing 1.7). The macro has a generic type `T`, and automatically collects the list items into a `Vec<T>`, which can be used by the rules that use the macro.

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

Listing 1.7: LALRPOP macro to parse a comma-separated list of non-terminals.

1.4 Middle End

Give an overview of the middle end

1.4.1 Intermediate Code Generation

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

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

Many objects in the IR require unique IDs, such as variables and labels. I created a `Id` trait to abstract this concept, together with a generic `IdGenerator` struct (Listing 1.8). The ID generator internally tracks the highest ID that has been generated so far, so that the IR can create IDs as necessary without needing to know anything about their implementation. IDs are generated inductively: each ID knows how to generate the next one.

Throughout the middle and back ends, I used a design pattern of passing a context object through all the function calls. For example, when traversing the AST to generate IR code, the `Context` struct in Listing 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 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 1.8: Implementation of the `Id` trait and `IdGenerator`.

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

Listing 1.9: The context datatype used when converting the AST to IR code.

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

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

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

The core of converting statements and expressions to IR code is matching the type of AST node, and generating IR instructions according to the structure of the statement, recursing into sub-statements and -expressions. The case for a **`while`** statement is shown in Listing 1.10; the labels and branches to execute a while loop are added, and the instructions to evaluate the condition and body are generated recursively. Other control-flow structures are similar.

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

Listing 1.10: Pseudocode for the `convert_statement_to_ir()` function.

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

1.4.2 The Relooper Algorithm

cite Emscripten [6]

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