IT University of Copenhagen

Design and Implementation of a Rust-Based Compiler with Type Checking and Evaluation Mechanisms

Prepared by:
Luca Giannini

Supervisor:
Rasmus Ejlers Møgelberg

Date: December 15, 2024

Contents

Preface and Introduction

This report has been written during the Fall Semester of 2024 as part of a research project at the IT University of Copenhagen under the supervision of Rasmus Ejlers Møgelberg. The purpose of this project is to explore and implement compiler construction techniques with an emphasis on type checking and evaluation mechanisms using the Rust programming language.

This report is primarily intended for readers with an interest in programming language design, developers investigating compiler construction methodologies, and computer science students seeking hands-on contributions to the field of compilers. It is assumed that the audience possesses a basic understanding of programming languages and has familiarity with concepts like syntax trees, type systems, and stack-based evaluation. Rust knowledge is not necessary to understand the project.

The project introduces a simplified compiler implemented in Rust, capable of parsing a small, Rust-like source language with features such as higher-order functions and lambda expressions. The compiler performs type checking and uses a stack-based approach to evaluate code, serving as a foundational framework for future extensions and optimizations.

The source code is available in Appendix C

1 Background and Description of the Problem

The ability to design and develop programming languages is key to advancing the field of software development, giving rise to innovative tools that cater to specific kinds of problems. At the core of any programming language lies its compiler, which translates human-readable source code into a machine-efficient representation for execution. Modern compilers are also designed to catch errors during development, optimize code, and provide meaningful feedback to programmers

This project addresses the challenge of designing a basic Rust-based compiler that supports type checking and evaluation mechanisms. The compiler operates on a simplified statically typed language that resembles a simple Rust source language, focusing on higher-order functions and lambda expressions. Implementing such a compiler, this project seeks to accomplish the following:

- Educational Outcome: Provide a practical, hands-on introduction to compiler construction
- Foundation for Future Development: By creating a working compiler that supports basic features, the project establishes a groundwork for adding advanced.

Success of this project is a functional but constrained problem domain: translating a syntactically valid, small language into a representation that can be "run" on a stack-based virtual machine. Design choices include:

- Source Language: The input language supports modern programming constructs such as higher-order functions, type safety, and lambda expressions, providing a useful context for investigating compiler techniques.
- Goals and Constraints: The implementation prioritizes simplicity and correctness over advanced optimization techniques. The immediate goal is to demonstrate type safety and proper runtime evaluation of valid programs

This undertaking provides us with an overview of how a minimal yet functional compiler is specified, designed and implemented.

2 Problem Analysis

The primary goal of this project was to design and implement a compiler for a statically-typed programming language, called NewLang, with first-class support for higher-order functions. This involved translating a user-defined language into an intermediate representation and executing the compiled code on a virtual machine. Specifically, the compiler had to perform lexical analysis, parsing, type checking, IR generation, and execution. This section will explore the key design decisions, alternative approaches, and their trade-offs.

2.1 Compilation Pipeline Design

The compilation pipeline was designed to minimize translations between intermediate representations, balancing simplicity and functionality. The chosen steps were:

1. **Source Code → AST:** The input program is parsed into an Abstract Syntax Tree (AST) that explicitly represents the program's structure. As basically all languages work with an AST, aswell as in Peter Sestoft book[1] this seemed like the right choice.
2. **AST → Type Checking:** Static type checking ensures that all variables and functions conform to their expected types, enhancing program correctness and safety. This came more down from preferences, as we dislike dynamic typing such as Python. Currently with the simple type system there is no need to indicate a type on assignment.
3. **AST → Stack IR:** An intermediate representation (IR) consisting of a linear instruction set is generated from the AST. This IR is specifically designed for execution by a stack-based virtual machine.
4. **Stack IR → VM Execution:** The virtual machine (VM) executes the Stack IR instructions.

This approach prioritized a direct and functional pipeline, avoiding overly complex transformations. Generating machine code via a tool such as LLVM was considered, but deemed too technically complex within the project timeline. Aswell as it not fitting into the goals of the project. Implementing a MIR, Mid-level IR, was discussed at the beginning and didn't seem necessary. However for beter optimization this was needed, work on it had begon but not finished. Not focusing much on these other options ensured a functioning pipeline in early development phases, enabling focus on type checking and core execution logic.

2.2 AST Creation

To avoid a large time investment in hand-coding parsing and lexing logic, existing tools in the Rust ecosystem were chosen. Lalrpop [2] was used for generating the parser from a user-defined grammar, simplifying source code conversion into the AST. Lalrpop's grammar validation ensured ambiguities were avoided during parsing. Lalrpop being an LR(1) parse, was more than enough for our use case. As more complicated grammar, more token look ahead, would not be needed.

Lexing support was provided by Logos [3], a robust and efficient lexical analyzer compatible with Lalrpop. While Logos allows for custom error-handling during lexing, this feature was not fully explored due to time constraints. Current errors are difficult to understand as little context is given, it was a deliberate choice to not spend time on this

These tools accelerated the project, removing much of the parsing work, and allowed us to achieve our goals sooner.

2.3 Type System Implementation

Given the emphasis on a statically-typed language, type inference was implemented for variable declarations to improve usability without compromising type safety. For instance, explicit type annotations for variables are optional. This approach is shown in the following examples:

```
// x doesn't have explicet type annotation, but is static typed.
let x = 10;
func add_one(x: int) -> int {
    x + 1
}
// The program implicitly checks and ensures the return type matches
x + add_one(x)
```

While enforcing explicit type annotations was considered, the chosen inference-based approach aligned with the project's goals for a more user-friendly language. Adding optional explicit type annotation could be easily added if found neccesary at a later date.

Having variable and function types stored in the AST simplified type checking, making it deterministic. Meaning the type system was a simple part of the program

2.4 Memory and Scoping Model

Two models for handling variable scope and memory management were considered:

- **Hash Map-Based Frames:** Stack frames store variables in a hash map, indexed by variable names, allowing flexible lookup at runtime.
- **Static Indexing:** Variables are assigned fixed indices at compile time, enabling direct access and reducing runtime overhead.

Initially, the hash map-based approach was implemented because of its ease of implementation and early organic alignment with development. As it made compiling the program easier, with more logic needed at runtime. However, this is computationally inefficient, requiring lookups by name at runtime. Static indexing was partially implemented as an optimization, but remains unfinished due to time limitations. Static indexing is more efficient as when you compile, you know for each used variable the scoping of it. This allows you to compile the index of each variable and use those instead of looking through multiple hashmaps.

The current implementation uses the following data structure:

```
struct StackFrame {
    return_address: usize,
    local_variables: HashMap<String, StackValue>
}
```

The optimized approach would have used the following design:

```
struct StackFrame {
    return_address: usize,
    locals: Vec<StackValue> // Direct indexing
}
```

This data structure would have allowed direct memory access for variables. Future work will focus on completing the static indexing for better performance.

2.5 Function Implementation

To support higher-order functions, closures were chosen as the function model. Closures allow functions to capture external variables from their defining scope. This approach aligns naturally with the hash map-based memory model used for scoping, and provides clean semantics for higher-order functions, lambda expressions. For example:

```
// Example of a lambda capturing its environment
let x = 5;
let add_to_x = |y| -> int {
 // x is captured in the lambda's closure
 x + y
}
```

Alternative function models, such as direct function references without captures, could have been simpler to implement but were unsuitable given the requirement for full support of higher-order functions and capturing. By choosing closures, the language efficiently supports rich functional programming abstractions.

3 User's Guide and Examples

3.1 Overview of the NewLang Compiler

The NewLang compiler is designed to process source code written in the NewLang language, a statically typed, imperative language featuring first-class functions and lambda expressions. The compiler operates in a multi-stage process: lexical analysis and parsing using LALRPOP and Logos to generate an Abstract Syntax Tree (AST), followed by static type checking, intermediate representation (IR) generation into a stack-based instruction set, and finally, execution on a custom Virtual Machine (VM). The execution model is based on a stack machine, where instructions manipulate values on the stack. Execution begins at the global scope, followed by the `main` function (if present). The final result of the program is the value of the last executed expression.

3.2 Invoking the Compiler

To execute a NewLang program, use the following command-line invocation:

```
./newlang <path-to-source-file>
```

Where `<path-to-source-file>` is the file path to the `.nl` source code file.

3.3 Programmatic Constraints and Requirements

- **Single Compilation Unit:** The compiler is designed to process a single `.nl` source file, acting as the compilation unit. This project currently does not support multiple compilation units or linking.
- **Explicit Function Signatures:** All function parameters and return types must be explicitly declared. Functions that do not specify a return type are treated as if they return `unit`, signifying the absence of a meaningful return value.
- **Global Scope and Execution Order:** Code within the global scope is executed before the execution of the `main` function (if defined). Function definitions may appear before or after their call sites, as the compiler's initial pass collects all function definitions before executing any other code. This is a form of forward declaration for functions.
- **Statically Typed Language:** NewLang enforces strict static typing, ensuring type safety at compile time. The compiler performs type inference for variable declarations, allowing some degree of conciseness while still retaining type safety. Re-assignment of variables to a different type is disallowed in the same scope to avoid type errors.

3.4 Syntax of NewLang Programs

NewLang programs are structured with the following elements:

- **Assigning:** Values can be assigned to a variable name using the `let` keyword.

```
let x = 10;
```

- **Function Definitions:** Functions are declared using the `fn` keyword, followed by the function name, a parenthesized list of typed parameters, an optional return type, and a body enclosed in braces {}. Functions can accept other functions as arguments, demonstrating support for higher-order functions. The last expression of the function is its return value.

```
fn add(x: int, y: int) -> int {
   x + y;
}
```

- **Global Scope:** Code outside of any function declaration is treated as global scope code and executed first. Global variable declarations can be made using the `let` keyword.

```
let x = 10;
print(x);
```

- `main` **Function:** If present, the `main` function is executed after all global scope code. The returned value of main is the last value returned by the interpreter. The type of the main return should be explicitly stated.

```
fn main() -> int {
    let result = 10;
    // program exist with this int
    result
}
```

- **Supported Data Types:** NewLang currently supports the following primitive data types:
  - `int`: Signed 64-bit integers (`i64`).
  - `float`: Double-precision floating-point numbers (`f64`).
  - `string`: UTF-8 encoded text strings.
  - `bool`: Boolean values (`true` or `false`).
  - `unit`: A type representing the absence of a value, analogous to `void` in other languages. It is implicitly the return type of statements that don't return a specific value and functions with no declared return type.
- **Expressions and Statements:** NewLang adheres to an expression-oriented design, where all language features are expressions that yield a value. The last expression of any block is the return value of that block. Statements are expressions with the `;` keyword appended, this means that the value will be discarded and `unit` will be returned.
- **Comments:** Single-line comments are denoted by `//` and must appear on their own lines. Multi-line comments are not supported.
- **Lambda Expressions:** Lambda expressions (anonymous functions) are defined using pipe symbols `|` to denote the beginning and end of the argument list, and `->` to denote the return type, e.g., `|x: int| -> int { x + 1; }`. They can capture variables from their surrounding scope, thus implementing closures.

```
// a lambda
let add_to = |a: int, b: int| -> int {
    a + b
}
```

The semicolon is used to "discard" values, due to all expressions returning a value. This value might already be the unit value, but adding the semicolon can make sure that you are discarding the returned value.

3.5 Error Handling and Diagnostics

The compiler provides basic error messages during parsing, type checking, and execution. Common error messages include:

- **Type Mismatch:** `Cannot compare different types: '<type1>' and '<type2>`. Indicates an attempt to compare two values of incompatible types in binary operations. For example, comparing an integer with a boolean.
- **Undeclared Variable:** `Variable '<variable name>' not found`. Signifies the use of a variable before its declaration within the current scope.
- **Function Not Found:** `Function '<function name>' not found`. Indicates a call to a function that has not been defined or is not within the current scope.
- **Type Error**: `Type error: '<type1>' is not '<type2>'`. This error occurs when an expression returns a type different from the expected type during type checking.
- **Re-assignment Error:** `Cannot reassign to variable '<variable name>'`. This error is thrown when a variable is declared again in a scope, meaning it cannot be declared twice in the same scope (as new types can be inferred).

Error messages provide basic debugging information and the location of the error in source code. Improvements to more informative and helpful error messages are a current limitation of this project.

3.6 Example Program

The following is a minimal example program that demonstrates several features of the language:

```
// Basic function to add one
fn add_one(x: int) -> int {
    x + 1
}

// Higher-order function that executes given function
fn execute_with(f: fn(int) -> int, a: int) -> int {
    f(a)
}

// Main function which will be the entry point of the program
fn main() -> int {
    let y = 10;
    let result = execute_with(add_one, y);
     // last expression will be returned due to no ";"
     // and as we are in main, the program returns this value
    result;
}
```

In this example, global scope code is ignored, and the function `main` will be executed with return value `11`.

A bigger example program can be found in Appendix A.

4 Technical Description of the Program

The program is designed to compile Newlang and execute it using a Virtual Stack Machine. There are two evaluations mechanisms in the program, an older direct from AST evaluation, and evaluation through a Stack based virtual machine. The latter doesn't support all language features, but was used at the start to verify programs.

4.1 Abstract Syntax Tree, Expression

Internally the AST is the *Expression* Enum, found at B.1. Which contains all possible expressions which are possible in Newlang. B.1 also has all other enums/structs for being able to create a complete Expression. Due to Rusts memory model, stack based values need to have their size known, recursive Expressions need to be boxed. Boxed in rust is a way to put an object on the heap, done for objects where you do not know the size of beforehand.

This AST is created directly from the Lalrpop/Logos combination, which can also be found at B.2. The grammar just dictates how to create a "script", which in this case is a Vec (list, dynamic-array) $FunctionDefinitions$ and $Expressions$, when looking it at from the grammar side.

The values of the AST is $AstValue$ also found in B.1, which dictates the possible types of values we can have: $Unit$,$Integer$,$Float$,$Boolean$,$String$,$Function$.

The AST is designed in a way such that anything is an expression. This way you can use any language feature is any place, as long as the return type of your expression is the expected return type of the expression where it is used. This makes scripts like below nothing special. This design choice was inspired from the Rust language.

```
let x = 10;

if {
    let z = 1;
    x != z
} {
    print("Hello World!");
}
```

4.2 Type Environment

When checking for types, we needed an way to keep track the types of each variable, a simple type environment was a simple way to achieve this. With it we can confirm that all expressions have the correct type. In the `newlang::ast::typechecker` module, the `check_types` function checks for the types in a slice of expressions. Due to the desire to be able to call functions before they are stated in the source code, a first pass of the function definitions needs to be made. That way we know if uses of these functions is correct.

Due to the static nature of the language, we are more checking if the types are used properly. As no dynamic language features are allowed.

4.3 Stack Intermediate Representation Generation

The `VM` B.4 and the `StackIR` B.3 are the step of converting the `AST` to stack instructions (StackIR) and have the ability to execute these stack instructions.

The conversion from `AST` to `StackIR` instructions is handled by a `newlang::stack::VM::expressions_to_stac` method on the `VM` struct. Which takes a vector of `Expression` and outputs both the instructions and function table. The conversion process happens in several phases:

- **Lambda Collection**: First all lambda expressions are collected and processed.
- **Function Definition Processing**: Regular functions definitions are processed next.
- **Main Program Generation**: Non-function expressions are converted to their stack instructions equivalents.

The VM keeps track of jumping labels for when expressions call lambdas/functions.

Some important properties is that lambdas are hoisted, and keep track of their scoped captured variables. Due to having a global scope, functions also need to keep track of their captured variables. Control flow is handled via labels and jumps.

4.4 Stack IR Design

The `StackIR` enum represents the stack machine instructions, the entire enum is available in Appendix B.3. `StackIR` has the variants for managing variables with `Load,Store,Const`. The above mentioned control flow is handled with `Jump,JumpIfFalse,Label`, this allows for conditional branching (if/else). Jumps are also used for jumping to function instructions. The VM has a table which translates function names to their jump point. The `Call(FunctionName)` is the instruction used for that. This is very much an un-optimized solution. `Push` and `Pop` are instructions, but currently not used much. As the logic for the other instructions will push/pop values from the stack when needed.

Arithmetic/Logic operations are all their own variant, and have a direct comparable variant in the `AST`'s `Operator` enum B.1.

There are also block scoping instructions, this is due to the current implementation not compiling scoping rather searching correct scoping at run time. This needs scoping instructions when entering a new scope. Another reason for this is due to a language decision to not allow re-assigning variables, but allowing to assign a used scope inside a scoped block.

```
let x = 10;

fn add(a: int) -> int {
    // allowing re-assingment in inner scope
    let x = 1;
    a + x
}
// not allowing in the same scope as declaration
// let x = 5;

add(x)
```

4.5 VM Execution

The VM B.4 maintains everything needed for translating a `AST` into `StackIR` and execute it.

The `instruction_pointer` keeps track of the current instruction to execute. The `Stack` is just a Vec of `StackValue`, on which everything gets pushed/popped. Due to needing to track scope at execution as we don't have our variables indexed by index, the `VM` needs to keep track

of frames. These frames are used as function frames, but also to scope variables to.

Due to not compiling everything optimally, the `VM` has tables for looking up functions information and label positions. Global variables are in their own HashMap, this is the last place variables will be looked in.

The execution model follows standard stack machine principles. Instructions consume/produce `StackValue`, functions create new call frames (also used for scoping, variables are scoped to frames, return values are passed via stack.

This design is functional, however work has been put into making the compilation of the `AST` to `StackIR` in such a way that there is less need to search for scoping at run-time. However this work could not be finished in time.

An example of Stack instructions can be found at B.5 which showcases which instructions would be generated for a reasonable program.

5 Tests

The testing approach for this compiler implementation follows a multi-layered strategy, combining unit tests, integration tests, and end-to-end tests across different modules. The tests are primarily located within the respective module files using Rust's built-in testing framework.

If cloned the repo, you can manually run all tests with "`cargo test`", or a specific test "`cargo test type_check`" (fuzzy finds tests with "type_check"), or test a module "`cargo test --lib typechecker`".

Testing Strategy

The testing strategy includes: - **Unit Tests**: To verify individual functions and methods. - **Integration Tests**: To test interactions between different modules. - **End-to-End Tests**: To ensure the entire system works as expected from source code parsing to execution.

Module-Level Tests

Each major module contains its own test module marked with $\#[cfg(test)]$, including:

Type Checker Tests: `newlang::ast::typechecker`, this module contains all logic for verifying a program's types. Tests include a handful of small scripts on which the function $check_t ypes$ is run. There are tests for basic types, error handling for type mismatches, higher-order function type checking, recursive function validation, and lambda expression type checking. The type checker also checks for some language features, such as reassigning variables.

```
#[test]
fn test_no_reassing_error() {
    let source_code = r#"
    let x = 5;
    let x = {
        let y = 1;
        y
    }
    "#;
    let ast = create_ast(source_code);
    assert_eq!(check_types(&ast), Err("Cannot reassign to variable 'x'".into()));
}
```

More tests can be found at the bottom of the module.

Stack Tests: In the `newlang::stack` module, we test the creation of the `StackIR` instructions generation, function table creation, and label management. This is critical, as any modifications to these systems will probably need modifications elsewhere.

Virtual Machine Tests: In the `newlang::stack::vm` module, we do a full source-to-execution test, ensuring that the compilation and execution are correct. Using source code as input ensures that the tests rely on other parts, making the input always correct. Simple expressions, function calls, higher-order function execution, lambda function handling, scoping rules, and main function execution were all tested here. These tests can be seen as end-to-end tests.

Test Coverage

The tests cover: - Basic type checking and error handling. - Higher-order functions and recursion. - Lambda expressions and closures. - Stack instruction generation and function table management. - Full source-to-execution validation in the virtual machine.

Examples of Expected and Actual Outputs

For example, the following test checks for variable reassignment errors:

```
#[test]
fn test_no_reassing_error() {
    let source_code = r#"
    let x = 5;
    let x = {
        let y = 1;
        y
    }
    "#;
    let ast = create_ast(source_code);
    assert_eq!(check_types(&ast), Err("Cannot reassign to variable 'x'".into()));
}
```

Integration Tests

Integration tests ensure that different modules work together as expected. For example, the integration between the lexer and parser is tested to ensure that tokens are correctly parsed into AST nodes.

```
#[test]
fn test_lexer_parser_integration() {
    let source_code = "let x = 5;";
    let ast = create_ast(source_code);
    assert_eq!(ast.len(), 1);
    assert_eq!(ast[0], Expression::Assignment { ... });
}
```

End-to-End Tests

End-to-end tests validate the entire compilation process from source code to execution. These tests ensure that the compiler correctly handles various language features and executes the generated code as expected.

```
#[test]
fn test_full_compilation() {
    let source_code = r#"
    fn main() {
        let x = 5;
        let y = x + 2;
        println!("{}", y);
    }
    "#;
    let output = compile_and_run(source_code);
    assert_eq!(output, "7\n");
}
```

The tests conducted provide a high level of confidence in the correctness of the compiler implementation. The multi-layered testing strategy ensures that individual components work correctly and interact as expected. However, there is always a risk of undetected errors, especially in edge cases not covered by the tests. Each time an edge case is found a test for it is made, this way the tests grew organically when needed.

6 Conclusion and Future Work

Summary of the Work Done

This project has laid the foundational groundwork for implementing a compiler in Rust to some success, specifically focusing on type checking and evaluation mechanisms. The main objectives were to study the theoretical underpinnings of compilers using standard references such as "Programming Language Concepts" by Peter Sestoft.

Challenges Faced

There were no real technical issues during the project, in case something did rise discussions with the supervisor quickly directed us in the right way. The biggest challenge which arose were health issues during the end of the project. This challenge limited the scope of implementation drastically, however due to great motivation at the start of the project we still achieved the understanding of compiler architecture.

Future Work and Potential Improvements

Several areas remain unexplored or underdeveloped due to the time limitations of the project:

- **Optimizations for the stack machine:** Currently, the compiler lacks considerations for indexed variables or efficiently managing the evaluation stack. Efforts had started to rectify this. Other optimizations such as: Function inclining, Constant tables, better simplification of `AST` were considered. With these optimizations the stack machine could be a seriously performant.
- **Error reporting:** Improvements could be made to deliver clearer and more actionable error messages to users during type checking and syntax parsing. Currently most of the program panics when encountering an error, this was to accelerate the progress. Using libraries like Anyhow[1] could make this a simple modification.
- **Code generation:** The framework could be extended to generate machine code or bytecode for a specific target environment (e.g., LLVM IR or WASM). Much excitement was directed to these areas, and with more time would have probably been explored.
- **More language features:** Given more time, language features such as user types could be implemented.
- **Performance Testing:** Would be interesting to see how performant the current solution is compared to other languages for simple use cases.
- **Separation between compiling and executing:** Being able to create compiled objects and save/share was thought of, but didn't comply with the project goals.

Exploring these areas could turn the project into a fully-fledged compiler, both more advanced and usable in practical contexts.

[1] `https://github.com/dtolnay/anyhow`

A Example program

```
// any code will be executed in the global scope.
let x = "hello";
let y = 10;

// This will print "hello10"
if y == add_one(9) {
    print("hello10")
}

// This will error, as x is not the same type as int
// if x == add_one(10) {
//    print("hello11")
// }

// functions can be created using fn keyword, typed arguments and return types are
    required.
fn add_one(a: int) -> int {
    a + 1
}

// higher order functions can be created using function types
fn exec_with(f: fn(int) -> int, a: int) -> int{
    f(a)
}

// we expect this to print 11
print(exec_with(add_one, 10));

// calling function before it is defined
print(factorial(5));

// recursive function
fn factorial(n: int) -> int {
    if n <= 1 {
        // without ";" an expression will return itself
        1
    } else{
        // without ";" an expression will return itself
        n * factorial(n - 1)
    }
}

// lambdas can be created in any scope, they can be used to create closures
fn create_adder(n: int) -> fn(int) -> int {
    |a: int| -> int {
        a + n
    }
}

let add_five = create_adder(5);
print("add_five(10)");
print(add_five(10));


// current types are:
```

```
let int_val = 10;
let float_val = 10.0;
let string_val = "hello";
let bool_val = true;
// unit value is not used yet, but it is a type.
// any function which doesn't return anything will return unit.
// discarded values, values with ";" return unit.
let unit_val = {};


// after running all the global code, main will run
// main will return a string in this example
fn main() -> string {

    let bigger = "bigger";

    if add_one(9) == 10 {
        "hello10 again"
    } else {
        "hello11"
    }
}
```

B Program Technicals

B.1 Expression

```
pub enum Expression {
    Value(AstValue),
    Variable(String),
    VariableAssignment {
        name: String,
        value: Box<Expression>,
    },
    IfElse {
        condition: Box<Expression>,
        then_branch: Box<Expression>,
        else_branch: Option<Box<Expression>>,
    },
    FunctionDefinition {
        name: String,
        args: Vec<TypedArg>,
        body: Box<Expression>,
        return_type: Type,
    },
    BinaryOperation {
        lhs: Box<Expression>,
        operator: Operator,
        rhs: Box<Expression>,
    },
    Print {
        value: Box<Expression>,
    },
    FunctionCall {
        name: String,
        args: Vec<Expression>,
    },
    Block {
        expressions: Vec<Expression>,
    },
    Lambda {
        id: usize,
        args: Vec<TypedArg>,
        body: Box<Expression>,
        return_type: Type,
    },
    Discard {
        value: Box<Expression>,
    },
}

pub enum AstValue {
    Unit,
    Integer(i64),
    Float(f64),
    Boolean(bool),
    String(String),
    Function {
        name: String,
        args: Vec<TypedArg>,
        body: Box<Expression>,
```

```
        return_type: Type,
    },
}

pub struct TypedArg {
    pub name: String,
    pub t: Type,
}

pub enum Type {
    Integer,
    Float,
    Boolean,
    String,
    Unit,
    Function {
        args: Vec<Type>,
        return_type: Box<Type>,
    },
}

pub enum Operator {
    Add,
    Sub,
    Mul,
    Div,
    Eq,
    Neq,
    Lt,
    Gt,
    Le,
    Ge,
    And,
    Or,
}
```

B.2 grammar.larlpop

```
use crate::tokens::{Token, LexicalError};
use crate::ast;

grammar(counter: &mut usize);

extern {
    type Location = usize;
    type Error = LexicalError;

    enum Token {
        "identifier" => Token::Identifier(<String>),
        "int_literal" => Token::IntLiteral(<i64>),
        "float_literal" => Token::FloatLiteral(<f64>),
        "string_literal" => Token::StringLiteral(<String>),
        "bool_literal" => Token::BoolLiteral(<bool>),
        "int" => Token::KeywordInt,
        "float" => Token::KeywordFloat,
        "string" => Token::KeywordString,
        "bool" => Token::KeywordBool,
        "let" => Token::KeywordVar,
        "print" => Token::KeywordPrint,
        "fn" => Token::KeywordFn,
        "if" => Token::KeywordIf,
        "else" => Token::KeywordElse,
        "->" => Token::KeywordArrow,
        "=" => Token::Assign,
        ";" => Token::Semicolon,
        ":" => Token::Colon,
        "(" => Token::LParen,
        ")" => Token::RParen,
        "{" => Token::LBrace,
        "}" => Token::RBrace,
        "," => Token::Comma,
        "+" => Token::OperatorAdd,
        "-" => Token::OperatorSub,
        "*" => Token::OperatorMul,
        "/" => Token::OperatorDiv,
        "==" => Token::OperatorEq,
        "!=" => Token::OperatorNeq,
        "<" => Token::OperatorLt,
        "<=" => Token::OperatorLe,
        ">" => Token::OperatorGt,
        ">=" => Token::OperatorGe,
        "&&" => Token::OperatorAnd,
        "||" => Token::OperatorOr,
        "|" => Token::Pipe,
    }
}


pub Script: Vec<ast::Expression> = {
    <items:TopLevelItem*> => items
}

TopLevelItem: ast::Expression = {
    FunctionDefinition,
```

```
    Expression,
}

FunctionDefinition: ast::Expression = {
    "fn" <name:"identifier"> "(" <args:Comma<TypedArg>?> ")" <return_type:("->"
        Type)?> <body:Block> => {
        ast::Expression::FunctionDefinition {
            name,
            args: args.unwrap_or_else(Vec::new),
            return_type: match return_type {
                Some(t) => t.1,
                None => ast::Type::Unit,
            },
            body: Box::new(body)
        }
    }
}

pub Expression: ast::Expression = {
    #[precedence(level="0")]
    Term,
    Block,
    IfExpression,

    // how to add this aaaaaaaaaaaaaa
    // "-" <e:Expression> => ast::Expression::BinaryOperation {
    //     lhs: Box::new(ast::Expression::Value(ast::AstValue::Integer(0))),
    //     operator: ast::Operator::Sub,
    //     rhs: Box::new(e),
    // },

    #[precedence(level="3")] #[assoc(side="left")]
    <left:Expression> "*" <right:Expression> => ast::Expression::BinaryOperation{lhs:
        Box::new(left), operator: ast::Operator::Mul, rhs: Box::new(right)},
    <left:Expression> "/" <right:Expression> => ast::Expression::BinaryOperation{lhs:
        Box::new(left), operator: ast::Operator::Div, rhs: Box::new(right)},

    #[precedence(level="4")] #[assoc(side="left")]
    <left:Expression> "+" <right:Expression> => ast::Expression::BinaryOperation{lhs:
        Box::new(left), operator: ast::Operator::Add, rhs: Box::new(right)},
    <left:Expression> "-" <right:Expression> => ast::Expression::BinaryOperation{lhs:
        Box::new(left), operator: ast::Operator::Sub, rhs: Box::new(right)},

    #[precedence(level="5")] #[assoc(side="left")]
    <left:Expression> "==" <right:Expression> =>
        ast::Expression::BinaryOperation{lhs: Box::new(left), operator:
        ast::Operator::Eq, rhs: Box::new(right)},
    <left:Expression> "!=" <right:Expression> =>
        ast::Expression::BinaryOperation{lhs: Box::new(left), operator:
        ast::Operator::Neq, rhs: Box::new(right)},
    <left:Expression> "<" <right:Expression> => ast::Expression::BinaryOperation{lhs:
        Box::new(left), operator: ast::Operator::Lt, rhs: Box::new(right)},
    <left:Expression> "<=" <right:Expression> =>
        ast::Expression::BinaryOperation{lhs: Box::new(left), operator:
        ast::Operator::Le, rhs: Box::new(right)},
    <left:Expression> ">" <right:Expression> => ast::Expression::BinaryOperation{lhs:
        Box::new(left), operator: ast::Operator::Gt, rhs: Box::new(right)},
```

```
    <left:Expression> ">=" <right:Expression> =>
        ast::Expression::BinaryOperation{lhs: Box::new(left), operator:
        ast::Operator::Ge, rhs: Box::new(right)},

    #[precedence(level="6")] #[assoc(side="left")]
    <left:Expression> "&&" <right:Expression> =>
        ast::Expression::BinaryOperation{lhs: Box::new(left), operator:
        ast::Operator::And, rhs: Box::new(right)},
    <left:Expression> "||" <right:Expression> =>
        ast::Expression::BinaryOperation{lhs: Box::new(left), operator:
        ast::Operator::Or, rhs: Box::new(right)},

    #[precedence(level="7")] #[assoc(side="right")]
    "let" <name:"identifier"> "=" <e:Expression> =>
        ast::Expression::VariableAssignment{ name, value: Box::new(e) },

    #[precedence(level="8")]
    <e:Expression> ";" => ast::Expression::Discard{value: Box::new(e)},
    "print" "(" <e:Expression> ")" => ast::Expression::Print{value: Box::new(e)},
    // ^ we can also handle it differently, with Function and NativeFunction in
        expressions
}

IfExpression: ast::Expression = {
    "if" <condition:Expression> <then_branch:Block> <else_branch:ElseBranch?> => {
        ast::Expression::IfElse{
            condition: Box::new(condition),
            then_branch: Box::new(then_branch),
            else_branch: else_branch.map(Box::new)
        }
    },
}

ElseBranch: ast::Expression = {
    "else" <Block>,
    "else" <IfExpression>,
}

Term: ast::Expression = {
    <val:"int_literal"> => {
        ast::Expression::Value(ast::AstValue::Integer(val))
    },
    <val:"float_literal"> => {
        ast::Expression::Value(ast::AstValue::Float(val))
    },
    <val:"string_literal"> => {
        ast::Expression::Value(ast::AstValue::String(val))
    },
    <b:"bool_literal"> => ast::Expression::Value(ast::AstValue::Boolean(b)),

    // variable
    <name:"identifier"> => {
        ast::Expression::Variable(name)
    },

    // calling a function
    <name:"identifier"> "(" <args:Comma<Expression>?> ")" => {
        ast::Expression::FunctionCall{
```

```
            name,
            args: args.unwrap_or_else(Vec::new)
        }
    },

    // "|" <args:Comma<TypedArg>?> "|" <body:Block> => {
    //    ast::Expression::Lambda {
    //        args: args.unwrap_or_else(Vec::new),
    //        body: Box::new(body),
    //        return_type: None,
    //    }
    // },

    // lambda with return type, implicit will do later
    "|" <args:Comma<TypedArg>> "|" "->" <return_type:Type> <body:Block> => {
        let id = *counter;
        *counter += 1;
        ast::Expression::Lambda {
            id,
            args,
            body: Box::new(body),
            return_type
        }
    },
}

Block: ast::Expression = {
    "{" <exprs:Expression*> "}" => ast::Expression::Block{expressions: exprs}
}


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

TypedArg: ast::TypedArg = {
    <name:"identifier"> ":" <t:Type> => ast::TypedArg { name, t }
}

Type: ast::Type = {
    "int" => ast::Type::Integer,
    "float" => ast::Type::Float,
    "string" => ast::Type::String,
    "bool" => ast::Type::Boolean,
    // "()" => ast::Type::Unit,
    "fn" "(" <args:Comma<Type>?> ")" <return_type:("->" Type)?> => {
        ast::Type::Function{
            args: args.unwrap_or_else(Vec::new),
            return_type: Box::new(match return_type {
                Some(t) => t.1,
                None => ast::Type::Unit,
            })
        }
    },
```

```
}
```

B.3 StackIR

```
pub enum StackIR {
    Const(StackValue),
    Load(String),
    Store(String),
    Jump(Label),
    JumpIfFalse(Label),
    Label(Label),
    Call(FunctionName),
    EnterBlock,
    ExitBlock,
    Return,
    Pop,
    Print,
    Add,
    Sub,
    Mul,
    Div,
    Eq,
    Neq,
    Lt,
    Gt,
    Le,
    Ge,
    And,
    Or,
}

pub enum StackValue {
    Unit,
    Integer(i64),
    Float(f64),
    Boolean(bool),
    String(String),
    Function {
        name: String,
        captures: HashMap<String, StackValue>,
    },
}
```

B.4 VM

```
pub struct VM {
    instruction_pointer: usize,
    stack: Vec<StackValue>,
    frames: Vec<StackFrame>,
    pub function_table: HashMap<String, FunctionInfo>,
    pub label_positions: HashMap<Label, usize>,
    /// global variables
    variables: HashMap<String, StackValue>,

    label_counter: usize,
}

struct StackFrame {
    return_address: usize,
    local_variables: HashMap<String, StackValue>,
}
```

B.5 StackIR example

(This is a test in the repo) With a reasonable program as this:

```
fn main() -> int{
    let x = 2;
    let y = 3;
    let z = add(x,y);
    factorial(z)
}
fn add(x:int, y:int) -> int{
    x + y
}
fn factorial(n: int) -> int{
    if n < 1{
        1
    }else{
        n * factorial(n - 1)
    }
}"
```

We create instructions:

```
StackIR::Jump(Label(5)), // jump over the functions to main
StackIR::Label(Label(0)), // main starting point
StackIR::EnterBlock,
StackIR::Const(StackValue::Integer(2)),
StackIR::Store("x".into()),
StackIR::Const(StackValue::Integer(3)),
StackIR::Store("y".into()),
StackIR::Load("x".into()),
StackIR::Load("y".into()),
StackIR::Call("add".into()), // label 1
StackIR::Store("z".into()),
StackIR::Load("z".into()),
StackIR::Call("factorial".into()), // label 2
StackIR::ExitBlock,
StackIR::Return,
StackIR::Label(Label(1)), // add function start
StackIR::EnterBlock,
StackIR::Load("x".into()),
StackIR::Load("y".into()),
StackIR::Add,
StackIR::ExitBlock,
StackIR::Return,
StackIR::Label(Label(2)), // factorial function start
StackIR::EnterBlock,
StackIR::Load("n".into()),
StackIR::Const(StackValue::Integer(1)),
StackIR::Lt,
StackIR::JumpIfFalse(Label(4)), // if
StackIR::EnterBlock,
StackIR::Const(StackValue::Integer(1)),
StackIR::ExitBlock,
StackIR::Jump(Label(3)),
StackIR::Label(Label(4)), // else
StackIR::EnterBlock,
StackIR::Load("n".into()),
StackIR::Load("n".into()),
```

```
StackIR::Const(StackValue::Integer(1)),
StackIR::Sub,
StackIR::Call("factorial".into()),
StackIR::Mul,
StackIR::ExitBlock,
StackIR::Label(Label(3)), // jump over the else
StackIR::ExitBlock,
StackIR::Return,
StackIR::Label(Label(5)), // label to jump over all the functions
StackIR::Call("main".into()) // label 0
```

C Source Code

The source code is available on Github `https://github.com/structwafel/public-newlang`.

References

[1] Peter Sestoft. *Programming Language Concepts*. Springer, 2017.

[2] LALRPOP Project Contributors. "LALRPOP: A LR Parser Generator for Rust". In: *GitHub Repository* (). URL: `https://github.com/lalrpop/lalrpop`.

[3] Maciej Hirsz. "Logos: Create ridiculously fast Lexers". In: *GitHub Repository* (). URL: `https://github.com/maciejhirsz/logos`.