Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

TC-3048 Compiler design

miniclj
Design document

Mario Emilio Jiménez Vizcaíno
A01173359

Elda Guadalupe Quiroga
Héctor Gibrán Ceballos

November 24th, 2021

Contents

1 About the project

1.1. Project scope

This project's aim is to create a compiler and virtual machine for a lisp-based language with similar semantics to Clojure. The base functions and data structures will be supported, and they must be accessible either through a Command-Line Interface or inside a web context.

1.2. Requirements

1. The compiler must be able to parse and recognize s-expressions.
2. The compiler must include a specific syntax for creating inline data structures such as lists, vectors, maps and sets
3. The compiler must check for lexic, syntax and semantic errors, and display an appropiate error message in these cases
4. The compiler must emit bytecode similar to quadruples, translating symbols to memory addresses, and the tree-based structure of s-expressions to a list of instructions
5. The virtual machine must be able to execute the bytecode produced by the compiler
6. The virtual machine must check semantic errors that couldn't be checked during compilation, such as the arity of callables and user defined functions
7. Both the compiler and virtual machine must use data structures that enable them to do their job efficiently and without wasting memory

Some test cases for these requirements can be found in section 7: Code examples.

1.3. Development process

The development of the language can be tracked from its GitHub repository: MarioJim/minicIj. The list of commits since the last time this document was generated can also be found in appendix A.

1.3.1. Weekly logs

During the development I've also kept a weekly log in Spanish of my progress. It can be found in the README.md file in the root directory, or in appendix B.

1.3.2. Final thoughts

I would say that this project has helped me learn more about how complex compilers are, because, even though the compiler I wrote is reasonably simple, I've had to build strong abstractions over many of the simple functions of my language, and making sure my abstractions work correctly during compilation and execution has been the hardest challenge I've encountered in this project.

2 About the language

2.1. Language name

I chose the name `miniclj` because this project aims to be a Clojure clone, with a subset of the language's functionality. The syntax and expressions are similar to Clojure's, but some special commands and data structures aren't available, such as support for macros (`defmacro`), symbols (also known as identifiers, they are replaced during compilation) and concurrency primitives (atoms, `swap!`, promises, `deliver`).

2.2. Language features

`miniclj` offers the basic functionality of a lisp-based language, such as a language based on s-expressions and first-class support for lists and lambda functions. Other features inherited from Clojure are more collection types (vectors, sets and maps) and support for strings as lists of characters. For more information check out the User Manual.

An online version of the language can be found in `miniclj`'s playground at mariojim.github.io/miniclj/.

2.3. Errors

The errors for each compilation and execution stage are the following:

2.3.1. Parser errors

This errors are the ones implemented by `lalrpop`, the parser generator library the language uses, and they are variants of the enum `ParseError`, found in the file src/lib.rs from the lalrpop-util crate.

- `InvalidToken`: Returned when the parser encounters a token that isn't part of the language's grammar
- `UnrecognizedEOF`: Returned by the parser when it encounters an EOF it did not expect
- `UnrecognizedToken`: Returned when the parser encounters a token it didn't expect in that position
- `ExtraToken`: Returned when the parser encounters an additional, repeated token
- `User`: Returned by the parser when a custom validation doesn't pass. This type of error is can only be returned while parsing bytecode from its string representation during execution, when a language function isn't recognized or when a memory address couldn't be parsed correctly.

2.3.2. Compiler errors

This errors are implemented as variants of the `CompilationError` enum, file `miniclj-lib/src/compiler/error.rs`.

```
/// Represents the type of errors generated during compilation
#[derive(Debug)]
pub enum CompilationError {
    /// Returned when the compiler finds a symbol that was supposed
    /// to be used as a callable, but isn't defined in the current
    /// scope (wasn't a user-defined function nor a language callable)
    CallableNotDefined(SmolStr),
    /// Returned when a expression tried to call a callable with
    /// no arguments, and the callalbe expects at least one
    EmptyArgs(&'static str),
    /// Returned by the compiler when a symbol wasn't defined
    /// in the current scope (or any other parent scope)
    SymbolNotDefined(SmolStr),
    /// Returned by the compiler when a function receives an argument
    /// that it didn't expect. Although most functions don't check the
    /// type of its arguments during compilation, some functions with
    /// a custom compilation process (such as `fn`, `defn` and `let`)
    /// use their arguments during compilation
    WrongArgument(&'static str, &'static str, &'static str),
    /// Returned when the user tried to call a callable with
    /// the wrong number of arguments
    WrongArity(&'static str, &'static str),
    /// Returned when the user tried to call the `recur` callable
    /// with a different number of arguments than it's corresponding
    /// `loop` call
    WrongRecurCall(usize, usize),
}
```

2.3.3. Runtime errors

This errors are implemented as variants of the `RuntimeError` enum, file `miniclj-lib/src/vm/error.rs`.

```
/// Represents the different errors that can happen during runtime
#[derive(Debug)]
pub enum RuntimeError {
    /// This variant of `RuntimeError` encloses any error that
    /// was caused by a compiler malfunction and should only be
    /// encountered by the user if the compiler has a bug or
    /// if the bytecode was modified
    CompilerError(String),
    /// This variant is returned when a value that was passed
    /// to a parsing function (like `num` and `chr`) couldn't
    /// be correctly processed
    CouldntParse(String, &'static str),
    /// Returned when the user tries to divide a number by zero
    DivisionByZero,
```

```
    /// Returned when the user tries to get a value from
    /// an indexed collection using the callable `nth`
    /// and the collection is shorter than the index
    IndexOutOfBounds(&'static str),
    /// Returned when, inside a function, a value is implicitly
    /// casted to a map entry, but the value isn't a vector
    /// with two elements
    InvalidMapEntry,
    /// Returned when a input/output function returned an error
    /// instead of correctly printing/reading strings
    IOError(&'static str, std::io::Error),
    /// Returned when the user tried to execute a value
    /// as a callable, but it wasn't a language function
    /// nor a user-defined callable
    NotACallable(&'static str),
    /// Returned when the user tried to call a callable
    /// with the wrong number of arguments, variant for functions
    /// with a specific arity
    WrongArityN(&'static str, usize, usize),
    /// Returned when the user tried to call a callable
    /// with the wrong number of arguments, variant for functions
    /// that can be called with different numbers of arguments
    WrongArityS(&'static str, &'static str, usize),
    /// Returned when a callable receives a value with an incorrect
    /// datatype, that the callable didn't expect
    WrongDataType(&'static str, &'static str, &'static str),
}
```

3 About the compiler

3.1. Tools and libraries

The compiler is written in Rust, and it has a couple of dependencies:

- `lalrpop`: used as a lexer and parser for the language
- `num`: used for its implementation of a fraction of 64 bit integers, `Rational64`
- `smol_str`: this package is used to keep small strings (less than 22 bytes) in the stack instead of allocating them in the heap

3.2. Tokens

The language recognizes the following tokens, separated in string literals and regular expressions:

3.2.1. String literals

- `"("`: ParenOpen
- `"#("`: ShorthandLambdaOpen
- `"'("`: ListOpen
- `")"`: ParenClose
- `"["`: BracketOpen
- `"]"`: BracketClose
- `"": BracesOpen`
- `"#{": SetOpen`
- `""`: BracesClose
- `"nil"`: Nil
- "%": ShorthandLambdaArgument
- "=": ComparisonOp::Eq
- `"!="`: ComparisonOp::Ne
- ">": ComparisonOp::Gt
- "<": ComparisonOp::Lt
- "<=": ComparisonOp::Ge
- ">=": ComparisonOp::Le
- "+": FactorOp::Add
- "-": FactorOp::Sub
- `"*"`: FactorOp::Mul
- "/": FactorOp::Div

3.2.2. Regular expressions

- `r"[-]?[0-9]+"`: IntegerLiteral
- `r"[-]?[0-9]+.[0-9]+"`: DecimalLiteral
- `r#""[^"]*""#`: StringLiteral
- `r"[A-Za-z][A-Za-z0-9!?'_-]*"`: UserDefinedSymbol

3.3. Grammar rules

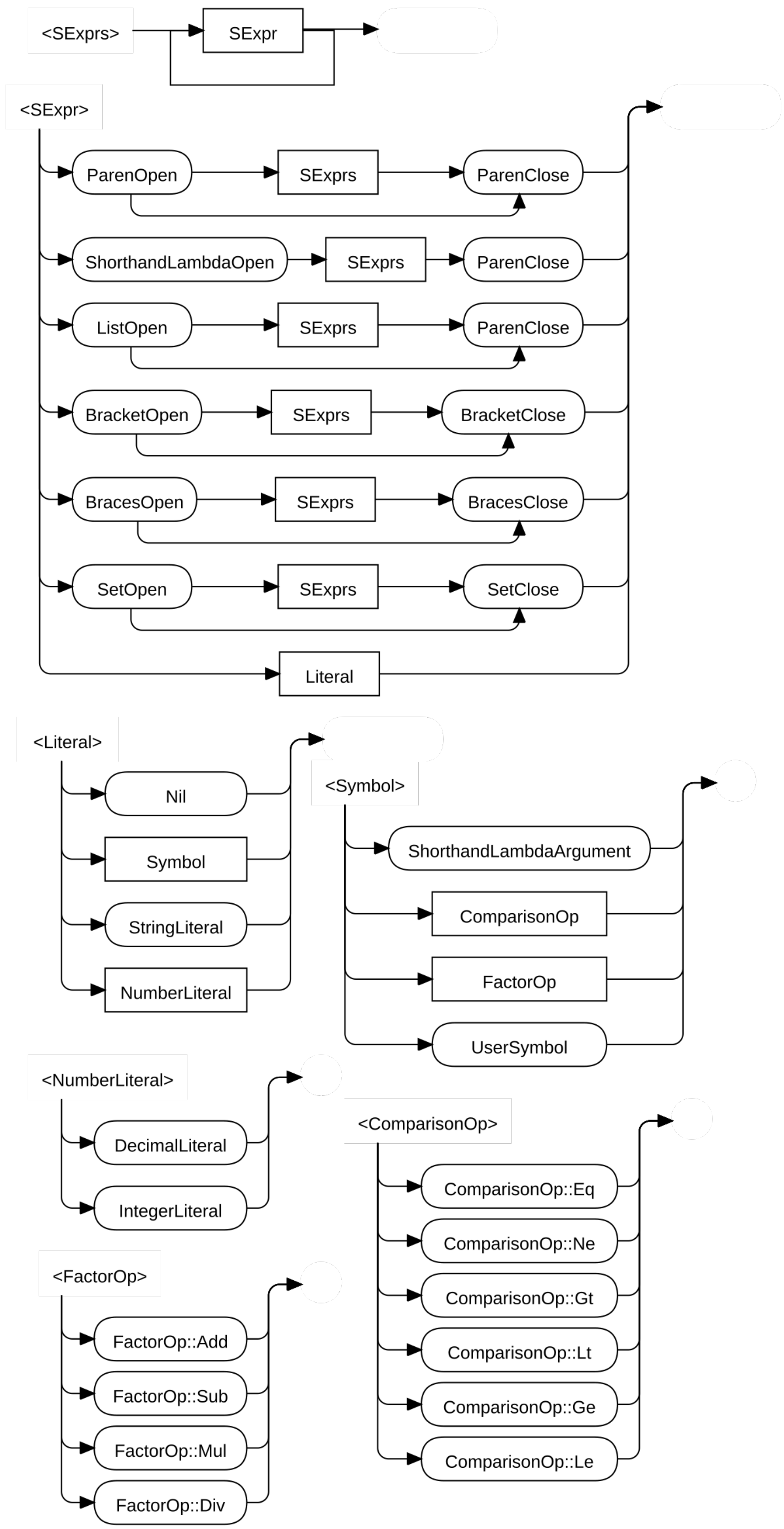
The grammar of the language is described in the file `miniclj-lib/src/parsers/lispparser.lalrpop`, included in appendix C. It describes the following rules:

- **SExprs**
  - **SExpr SExprs**
  - **SExpr**
- **SExpr**
  - ParenOpen **SExprs** ParenClose
  - ParenOpen ParenClose
  - ShorthandLambdaOpen **SExprs** ParenClose
  - ListOpen ParenClose
  - ListOpen **SExprs** ParenClose
  - BracketOpen BracketClose
  - BracketOpen **SExprs** BracketClose
  - BracesOpen BracesClose
  - BracesOpen **SExprs** BracesClose
  - SetOpen BracesClose
  - SetOpen **SExprs** BracesClose
  - **Literal**
- **Literal**
  - Nil
  - **Symbol**
  - StringLiteral
  - **NumberLiteral**
- **NumberLiteral**
  - DecimalLiteral
  - IntegerLiteral
- **Symbol**
  - ShorthandLambdaArgument
  - **ComparisonOp**
  - **FactorOp**
  - UserDefinedSymbol
- **ComparisonOp**
  - ComparisonOp::Eq
  - ComparisonOp::Ne

  - ComparisonOp::Gt
  - ComparisonOp::Lt
  - ComparisonOp::Ge
  - ComparisonOp::Le
- **FactorOp**
  - FactorOp::Add
  - FactorOp::Sub
  - FactorOp::Mul
  - FactorOp::Div

During parsing, the full source code of the file is read, and then, depending on the s-expressions parsed, the bytecode is generated. There aren't any additional actions executed during parsing.

3.4. Syntaxis diagrams



3.5. `CompilerState` struct

The compiler state is enclosed inside the `CompilerState` struct, inside file `miniclj-lib/src/compiler/state.rs`.

```
/// Structure used to process `SExpr`s into bytecode
#[derive(Debug, Default)]
pub struct CompilerState {
    constants: RustHashMap<Constant, MemAddress>,
    instructions: Vec<Instruction>,
    symbol_table: Rc<SymbolTable>,
    loop_jumps_stack: Vec<(InstructionPtr, Vec<MemAddress>)>,
    callables_table: CallablesTable,
}
```

This structure is composed of 5 data structures:

- `constants`: This hashmap stores the relationships between the constants and their memory addresses. I decided to use a map instead of a vector so that repeated constants occupy the same address. This map is accessed by the following methods of the `CompilerState` struct:
  - `insert_constant`: Receives a constant and returns a memory address. This method has two branches: when the constant was already added to the constants map, this metod just returns a copy of the address assigned to the constant. In case the constant wasn't found in the constants table, the compiler finds the next address available by iterating through the map and inserts the constant with that address.
- `instructions`: This vector stores the list of instructions that will be later executed by the VM. It is accessed by the following methods:
  - `add_instruction`: Receives an instruction, appends it to the vector, and returns the index of the new instruction
  - `instruction_ptr`: Returns the length of the instructions vector, used as the index of the following instruction to be inserted
  - `fill_jump`: Receives two instruction pointers: the first one is the index of the jump instruction to be modified, and the second one the instruction that it should point to. If the first instruction pointer doesn't refer to a jump instruction, the compiler crashes.
- `symbol_table`: This custom structure, described in file `miniclj-lib/src/compiler/symboltable.rs` and implemented as a linked list, has three fields:
  - `symbols`: A hashmap of identifiers (declared inside the current scope) to memory addresses
  - `temp_counter`: A counter of how many temporal variables have been created in the current scope
  - `var_counter`: A counter of how many local variables have been assigned in the current scope

  This data structure is accessed by the following methods:

  - `get_symbol`: Receives a reference to a string and returns either the memory address that points to the value of the identifier or no memory address in case that the symbol couldn't be found in the scope
  - `new_address`: Receives a `Lifetime` variant to determine if the new address should be a temporal, local or global address, and returns a new memory address
  - `insert_symbol`: Receives a string and an address, and inserts them into the corresponding symbol table (either the current symbol table if the address is local, or the global symbol table if the address has a global lifetime)
  - `remove_symbol`: Receives a reference to a string and removes the symbol from the scope

- `loop_jumps_stack`: This structure, although represented as a vector, is used as a stack of pairs of instruction pointers and vectors of memory addresses. This stack is useful for loop/recur cycles, where the compiler has to check where was the last `loop` instruction declared, so that when a `recur` instruction is found:

  1. The compiler can check that it has the same number of arguments
  2. It can copy the value from each argument to the memory address of the `loop`'s declaration
  3. It can emit the goto instruction to the `loop` instruction

  This process is documented in file `miniclj-lib/src/callables/scopefns.rs`, and `CompilerState` exposes the following methods to modify the `loop_jumps_stack`:

  - `push_loop_jump`: Receives an instruction pointer and a vector of memory addresses, and appends the pair to the stack
  - `pop_loop_jump`: Returns a pair of instruction pointer and vector of memory addresses, or nothing if the stack is empty

- `callables_table`: This custom structure, implemented as a map between strings and structs that implement the `Callable` trait. It is declared in file `miniclj-lib/src/callables/mod.rs`, and it is used to manage the compilation for callables, that consists of:

  - For most callables, compile the arguments, add the callable to the constants table and emit an `Call` instruction for the callable's address, the resulting address of each argument and the temporal address where the result of the call will be stored.
  - For the other callables, each one may have a different, custom compilation process, like the ones that modify the scope (`def`, `defn`, `let`), the ones used as cycles (`loop`, `recur`) and others (like `fn`)

This structure also exposes a couple of methods that are use throughout the compilation process:

- `compile`: This is the main method of the compiler: it receives an `SExpr`, it modifies its state depending on the variant of s-expression that it received, and returns either the resulting memory address of the expression, or a compiler error.

- `compile_lambda`: This method recieves a list of argument names (of the function that will be compiled) and an `SExpr` that contains the body of the function

- `write_to`: This method is used to serialize the compiler state into its string representation, first writing the constants table to the file, and then writing all the instructions in order. More information about this representation can be found in section 3.6
- `into_parts`: Finally, this method is used when the compiler state, instead of being printed to a file, is decomposed to create the state of a VM. It returns the constants and the instructions of the compiler

3.6. Bytecode representation

A bytecode file produced by the `miniclj` compiler, with the extension `.mclj`, is composed of two parts separated by a line with three asterisks: a list of constant and memory address pairs and a list of instructions. The pairs from the first part are only separated by a space.

3.6.1. Constants

The constants, defined in file `miniclj-lib/src/constant.rs`, have 5 different variants:

- `Callable`: Stores a reference to a language callable
- `Lambda`: Has two fields: the instruction pointer that the VM must jump to to execute the lambda, and the number of arguments that the lambda accepts
- `String`: Stores a string literal inside
- `Number`: Stores a `Rational64` struct inside (`num`'s implementation of fraction between two signed integers of 64 bits)
- `Nil`: The nil value

They are serialized (and deserialized) pretty easily:

- `Callable`: Only the callable name is stored
- `Lambda`: Both numbers are inserted after the string "fn", separated by the at sign (@)
- `String`: The string literal is printed between double quotes
- `Number`: The denominator is printed, then a slash (/) and finally the numerator
- `Nil`: The string "nil" is printed (without quotes)

3.6.2. Memory addresses

Memory addresses are composed of two fields:

- A `lifetime` field, of type `Lifetime`, which specifies the scope of the address. It can be either constant, global, local or temporal.
- The index of the address (represented by an unsigned integer)

They are serialized as unsigned 32 bit integers, where the first 4 bytes are reserved for the lifetime (constant being $1 * (2 << 28)$, global $2 * (2 << 28)$, and so on), and the other 28 bits are reserved for the index of the variable. The string representation of these addresses is just the number printed as is.

3.6.3. Instructions

The enum `Instruction` represents the type of instructions that the VM can execute. It has 6 variants and it is declared on the file `miniclj-lib/src/instruction.rs`. Here's a short description of each type:

- `Call`: Has 3 fields: the memory address of the callable, the list of memory addresses of the arguments, and the memory address where the result should be stored. This instruction is serialized starting with the string "call", then the address of the callable, the arguments and the result, separated by spaces
- `Return`: This instruction represents the return instruction from a lambda function, and it stores only the memory address of the value that the function will return. It is serialized as the keyword "ret", a space, and then the address
- `Assignment`: This instruction is used to copy a value from an address to another one. It stores the source and destination addresses, and is serialized starting with the keyword "mov", a space, the source address, another space, and the destination address
- `Jump`: This instruction represents an inconditional jump, and it stores only the instruction pointer to which the virtual machine should jump to. It is serialized using the word "jmp", a space, and the instruction pointer
- `JumpOnTrue`: This instruction is used when a jump should only be executed if a value is true. It stores the memory address of the value it should check and the instruction pointer it should jump to, and it is serialized with the keyword "jmpT", a space, the address, another space, and the instruction pointer
- `JumpOnFalse`: This instruction is almost the same as the last one, but only executing the jump if the value referenced by the memory address is false, and with being serialized with the keyword "jmpF"

4 About the virtual machine

4.1. Tools and libraries

The virtual machine, also implemented in Rust, uses the same dependencies as the compiler through the `num` callable that parses a number from a string, plus the module `escape8259`, that exports a function used to escape some characters (like `\n` to a newline character) when calling `print` or `println`.

4.2. `VMState` struct

The execution state is stored in the `VMState` structure, declared in file `miniclj-lib/src/vm/state.rs`.

```rust
/// Structure used to execute the bytecode produced by the compiler
#[derive(Debug)]
pub struct VMState {
    constants: HashMap<MemAddress, Constant>,
    instructions: Vec<Instruction>,
    global_scope: Scope,
}
```

This structure is composed of 3 fields:

- `constants`: A map of memory addresses to constants, read and constructed from the first part of the bytecode representation
- `instructions`: A vector of instructions, read from the second part of the bytecode file
- `global_scope`: This field is implemented as a custom structure named `Scope` (declared in file `miniclj-lib/src/vm/scope.rs`, explained in detail in the next section), and it is composed of two vectors of values: one for declared variables and one for temporal values. This structure is used for global and local variables declared in the root scope, and a new `Scope` is created when executing user defined functions with local variables

The main function of the structure `VMState` is `execute`, which calls a private method named `inner_execute`, implemented as a big match expression (like a switch statement) over the instructions that the virtual machine accepts.

Another important method is `execute_lambda` which, as the name implies, executes a lambda function defined by the user. It starts by checking the arity of the function, then creates a new `Scope`, inserts the local parameters at the start of the scope and also calls `inner_execute`.

4.3. `Scope` struct

As described earlier, memory is represented by the structure `Scope`, where values are stored inside two vectors.

```rust
/// Stores the local variables and the temporal values
/// of the current scope
#[derive(Debug, Default)]
pub struct Scope {
    vars: ValuesTable,
    temps: ValuesTable,
}
```

This structure has 4 methods: two `get` methods for temporal values and variables, which accept an index and return either the value of the vector at that index, or a `RuntimeError::CompilerError` when the value wasn't found; and two `store` methods, also for temporal values and variables, which receive an index and a `Value`, which is then stored in the corresponding vector.

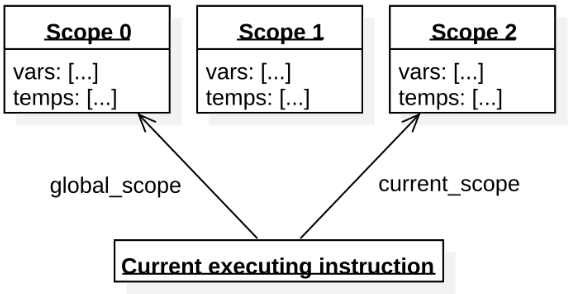
The `get` methods from `Scope` are called by a `get` method in `VMState`, which receives a reference to the current scope and a memory address, and then routes the request depending on the lifetime of the memory address:

- In case that it has a `Constant` lifetime, this method checks the `constants` field in `VMState`

- If the address has a `GlobalVar` lifetime, it checks the `global_scope` field also in `VMState`
- If the address has either a `LocalVar` or `Temporal` lifetime, the request is routed to the current scope passed to the function

The same process happens with the `store` function in `Scope`: `VMState` has a method called `store` which routes its requests depending on the lifetime of the address, with an exception for `Constant` addresses, which aren't supposed to be modified during runtime.

This structure is created once to represent the global scope, and then every time a user-defined function is executed, as shown in the diagram.



During execution, the current instruction has access to two `Scope`s: the current scope, that is replaced every time a new scope is created and is accessed every time a local variable or a temporal value is read or written, and the global scope, accessed when a global variable is read or written. During execution of code in the global scope the reference to the current scope also points to the global scope, and local variables are mixed with global variables.

4.4. `Value` enum

This enum represents any type of value that can be used during runtime in `miniclj`. It is declared in file `miniclj-lib/src/vm/value.rs`.

```rust
/// Represents a value used during execution of `miniclj` code
#[derive(Clone)]
pub enum Value {
    Callable(Box<dyn Callable>),
    Lambda(InstructionPtr, usize),

    List(List),
    Vector(Vec<Value>),
    Set(HashSet<Value>),
    Map(HashMap<Value, Value>),

    String(String),
    Number(Rational64),
    Nil,
}
```

It has the following variants:

- `Callable`, which stores a unique pointer to a structure that implements the trait `Callable`

- `Lambda`, which stores an instruction pointer and the arity of the function
- `List`, which stores a `List` value (explained in the next section)
- `Vector`, with a `Vec` of values inside
- `Set`, with a `HashSet` of values
- `Map`, with a `HashMap` of keys and values `Value`
- `String`
- `Number`, with a `Rational64` structure inside (a fraction of two 64-bit signed integers)
- `Nil`

4.5. `List` enum

This data structure is implemented as an enum with two variants, to closely match Clojure's implementation.

```
/// List type from Clojure
#[derive(Debug, Clone)]
pub enum List {
    Cons(Box<Value>, Box<List>),
    EmptyList,
}
```

Besides implementing a couple of useful functions like "nth" and "len", this enum implements two special functions: `from_iter` which lets the virtual machine create a `List` from any iterator of `Value`s, and `try_from`, which facilitates converting any type of value into a `List`. This last function only works for collections and strings, and for the other types it returns an error with the type of value passed (that couldn't be converted).

```
impl FromIterator<Value> for List {
    fn from_iter<T: IntoIterator<Item = Value>>(iter: T) -> List {
        let mut list = List::EmptyList;
        for value in iter {
            list = List::Cons(Box::new(value), Box::new(list));
        }
        list
    }
}

impl TryFrom<Value> for List {
    type Error = &'static str;

    fn try_from(value: Value) -> Result<List, Self::Error> {
        match value {
            Value::List(list) => Ok(list),
            Value::Vector(vector) => Ok(vector.into_iter().rev().collect()),
            Value::Set(set) => Ok(set.into_iter().collect()),
```

```
            Value::Map(map) => Ok(map
                .into_iter()
                .map(|(key, val)| Value::Vector(vec![key, val]))
                .collect()),
            Value::String(string) => Ok(string
                .chars()
                .map(|char| Value::String(String::from(char)))
                .collect()),
            _ => Err(value.type_str()),
        }
    }
}
```

5 About callables

The name "callable" was originally inherited by Clojure, where a callable refers to any value that can be called, or in other words, used as the first value in a s-expression. In this project, the name "callable" refers to any function exposed by the language or lambda functions defined by the user.

5.1. Language callables and the `Callable` trait

This type of callables implement the trait `Callable`, declared on file `miniclj-lib/src/callables/callable.rs`.

```
/// Base trait that all language callables must implement
pub trait Callable: Display + Debug + DynClone {
    fn name(&self) -> &'static str;

    fn compile(&self, state: &mut CompilerState, args: Vec<SExpr>) -> CompilationResult
↪   {
        self.check_arity(args.len())?;
        self.inner_compile(state, args)
    }

    fn check_arity(&self, num_args: usize) -> Result<(), CompilationError>;

    fn inner_compile(&self, state: &mut CompilerState, args: Vec<SExpr>) ->
↪   CompilationResult {
        let callable_addr = self
            .get_as_address(state)
            .expect("Callable didn't override either get_as_address or inner_compile");

        let arg_addrs = args
            .into_iter()
            .map(|expr| state.compile(expr))
```

```
31             .collect::<Result<Vec<MemAddress>, CompilationError>>()?;
32
33         let res_addr = state.new_address(Lifetime::Temporal);
34         let instruction = Instruction::new_call(callable_addr, arg_addrs, res_addr);
35         state.add_instruction(instruction);
36
37         Ok(res_addr)
38     }
39
40     fn get_as_address(&self, _state: &mut CompilerState) -> Option<MemAddress> {
41         None
42     }
43
44     fn execute(&self, state: &VMState, args: Vec<Value>) -> RuntimeResult<Value>;
45 }
```

A Rust trait shares the same idea as an abstract class in Java, but Rust structures don't actually "inherit" traits, they implement them, meaning polymorphism can't be used to downcast structures that implement the trait `Callable`, and that's why the language uses references to those callables through type `Box<dyn Callable>` (a `Box` is a unique pointer, and the keyword `dyn` means that they implement that trait).

On line 12, the `Callable` trait is declared, and for it to be implemented for a structure, the structure must also implement the traits `Display` (used to display values as strings, in the same spirit as `Object.toString()` in Java), `Debug` (also used to display structures as strings, but with more information) and `DynClone` (makes it easier to clone references to callables and to store them in `Box`es).

The `Callable` trait exposes 6 different functions, as seen in the code snippet above:

- `name`, which returns a static reference to a string. This value is used when compiling code, to link a keyword (for example `defn`) to a callable (struct `Defn`, defined in file `scopefns.rs`).
- `compile`, receives a `CompilerState` structure and a vector of `SExprs` as arguments, is the main function used during compilation to modify the state of the compiler. The default implementation calls the method `check_arity` with the length of the vector of arguments and then calls `inner_compile` with the state and the arguments.
- `check_arity`, receives an unsigned integer and returns nothing, or a `CompilationError`. It doesn't have a default implementation.
- `inner_compile`, accepts the same arguments as `compile`, and has a default implementation (used for language functions, language macros override this implementation), where it executes the following code:
  1. First it calls the method `get_as_address` with the compiler state as the only argument. This function is used to include this function in the constants table of the `CompilerState`, and it returns a variant of the enum `Option<MemAddress>`: either `Some(address)` or `None`. The default implementation of this method returns `None`, and the `expect` call on the result of this method terminates the compiler with the included error message if the result is `None`. This forces any structures that implement `Callabe` to either implement `get_as_address` (for the callable to be used as a function) or override the default implementation of `inner_compile` (for the callable to be used as a macro).

2. Then, it converts the vector of SExprs into an iterator, which is used to compile every s-expression into either a MemAddress or a CompilationError. These results are thenn collected into either a vector of MemAdresses or the first CompilationError that the compiler found. Finally, the question mark at the end of line 30 is an operator in Rust that makes it easier to handle errors: if the result of the iterator was a CompilationError, the whole function returns that CompilationError, and if the result of the iterator was the vector of MemAddresses, the function continues executing as normal.
3. A new temporal address is created for the result of the function, and a new instruction is created to call the memory address of this function, with the memory addresses of its arguments, and lastly with the memory address where the function should save this result.
4. Finally that instruction is inserted into the compiler and the temporal address used for the result is returned.

- get_as_address, as explained above, is used to insert this function in the constants table of the CompilerState, and it returns a variant of the enum Option<MemAddress>: either Some(address) or None.
- execute, this method is used during execution, and it receives the state of the virtual machine VMState and a vector of values. It returns a RuntimeResult<Value>: either a Value or a RuntimeError.

5.1.1. Language functions

Language functions are compiled with the process described above. An example of this type of callables is IsEmpty.

```
#[derive(Debug, Clone)]
pub struct IsEmpty;

impl Callable for IsEmpty {
    fn name(&self) -> &'static str {
        "empty?"
    }

    fn check_arity(&self, num_args: usize) -> Result<(), CompilationError> {
        if num_args == 1 {
            Ok(())
        } else {
            Err(CompilationError::WrongArity(self.name(), "<collection>"))
        }
    }

    fn get_as_address(&self, state: &mut CompilerState) -> Option<MemAddress> {
        Some(state.get_callable_addr(Box::new(self.clone())))
    }

    fn execute(&self, _: &VMState, args: Vec<Value>) -> RuntimeResult<Value> {
        if args.len() != 1 {
            return Err(RuntimeError::WrongArityS(
                self.name(),
```

```
292                 "a collection",
293                 args.len(),
294             ));
295         }
296 
297         let maybe_coll = args.into_iter().next().unwrap();
298         match maybe_coll {
299             Value::List(List::EmptyList) => Ok(true),
300             Value::List(List::Cons(..)) => Ok(false),
301             Value::Vector(v) => Ok(v.is_empty()),
302             Value::Set(s) => Ok(s.is_empty()),
303             Value::Map(m) => Ok(m.is_empty()),
304             Value::String(s) => Ok(s.is_empty()),
305             Value::Nil => Ok(true),
306             _ => Err(RuntimeError::WrongDataType(
307                 self.name(),
308                 "a collection",
309                 maybe_coll.type_str(),
310             )),
311         }
312         .map(Value::from)
313     }
314 }
315 
316 display_for_callable!(IsEmpty);
```

This function is used to test if a collection is empty or not, and is referenced in miniclj code by the symbol "empty?" (the string returned by the method `name`).

It overrides the method `check_arity`, where it returns an error if the number of arguments passed to the function isn't equal to 1.

It also overrides `get_as_address` so that it returns a memory address asigned by the `CompilerState`, through calling its method `get_callable_address`.

Finally, it also provides an implementation for `execute`, where, first, the function, has to check for the number of arguments it received, because functions can be used as a value (for example, in transducers such as "filter").

Then, gets the first argument passed to it, and uses a match expression to check which type of argument it received. If it is a collection or a string, it checks if the underlying collection is empty; if it is "nil" it returns a true (because of a Clojure requirement), and if it is something else, the function returns a `RuntimeError`.

Line 316 calls a Rust macro, `display_for_callable`, defined in `minicl-lib/src/callables/mod.rs`, used to provide a default implementation of the `Display` trait (a trait used in Rust to represent a value as a string). This implementation uses the string provided in `Callable`'s method `name`.

```
macro_rules! display_for_callable {
    ($callable:ty) => {
        impl std::fmt::Display for $callable {
            fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
                write!(f, "{}", self.name())
            }
```

```
        }
    };
}
```

5.1.2. Language macros

This callabes are separated into a different section because they have a different compilation process: they override `Callable`'s method `inner_compile`, with the exception of callable `Recur`, which overrides the method `compile`.

The simplest example of this type of callables is `Do`, defined in `miniclj-lib/src/callables/groupingfns.rs`.

```
#[derive(Debug, Clone)]
pub struct Do;

impl Callable for Do {
    fn name(&self) -> &'static str {
        "do"
    }

    fn check_arity(&self, num_args: usize) -> Result<(), CompilationError> {
        if num_args == 0 {
            Err(CompilationError::EmptyArgs(self.name()))
        } else {
            Ok(())
        }
    }

    fn inner_compile(&self, state: &mut CompilerState, args: Vec<SExpr>) ->
    ↪ CompilationResult {
        let mut args_iter = args.into_iter();
        let mut res_addr = state.compile(args_iter.next().unwrap())?;
        for arg in args_iter {
            res_addr = state.compile(arg)?;
        }

        Ok(res_addr)
    }

    fn execute(&self, _: &VMState, _: Vec<Value>) -> RuntimeResult<Value> {
        Err(RuntimeError::CompilerError(format!(
            "Compiler shouldn't output \"{}\" calls",
            self.name()
        )))
    }
}

display_for_callable!(Do);
```

This macro is usually used to group calls to functions with side-effects. It receives any number of functions and returns only the result of the last one.

The overriden method `inner_compile` does just that: it compiles the first s-expression, saves it's result address in variable `res_addr`, and then iterates through the remaining s-expressions, compiling each one and replacing the variable `res_addr` with the result of the compilation. After the iterator is exhausted, the variable `res_addr` is returned.

Another important point is that every language macro overrides `execute` so that it automatically returns a `RuntimeError::CompilerError` (the variant of `RuntimeError` that indicates a bug in the compiler) with an error message detailing what happened and which function was called.

5.1.3. `CallablesTable` struct

This structure, defined in `miniclj-lib/src/callables/mod.rs`, stores a read-only map of callable names to pointers of callable structures. It is used both during compilation to compile macros and functions) and execution (to link callable names from the constants table to references to the corresponding structures).

```
/// The map of symbols to callables exposed by the language
pub struct CallablesTable(RustHashMap<String, Box<dyn Callable>>);
```

It only has one method: a `get` method which receives a reference to a string and returns either a pointer to the callable with that name or nothing.

```
impl CallablesTable {
    pub fn get(&self, name: &str) -> Option<Box<dyn Callable>> {
        self.0.get(name).cloned()
    }
}
```

5.2. User-defined functions

Although there are three ways to write functions: using a "defn" call, using the "fn" macro, and using the shorthand syntax ("#(...)"), all of them have a similar compilation and execution process.

5.2.1. Compilation

To exemplify the compilation process of these user-defined functions, here's the process for compiling a lambda defined using the shorthand syntax. It starts `CompilerState`'s main function: `compile`, which matches over the type of expression passed to it. Here's the actions executed when it encounters a lambda function defined with the shorthand syntax:

```
            SExpr::ShortLambda(exprs) => {
                let jump_lambda_instr = Instruction::new_jump(None);
                let jump_lambda_instr_ptr = self.add_instruction(jump_lambda_instr);
                let lambda_start_ptr = self.instruction_ptr();
                let lambda_const = Constant::new_lambda(lambda_start_ptr, 1);
                let lambda_addr = self.insert_constant(lambda_const);
```

```

            self.compile_lambda(vec![SmolStr::from("%")], SExpr::Expr(exprs))?;
            self.fill_jump(jump_lambda_instr_ptr, self.instruction_ptr());
            Ok(lambda_addr)
        }
```

First, a new inconditional jump instruction is created, with an unknown instruction pointer as a destination. This instruction is added to the compiler, and the index of this instruction is saved in variable `jump_lambda_instr_ptr`. Then, the instruction pointer to the next instruction is saved, and a new constant is created using that pointer and the arity of the function, which for shorthand lambda functions is always 1. This constant is inserted into the compiler, and the address that corresponds to the lambda is saved in variable `lambda_addr`.

The compiler then calls its own method `compile_lambda` with the argument names (in shorthand lambdas it is only the symbol "%") and the body of the function.

Finally, the jump in position `jump_lambda_instr_ptr` is filled with the current instruction pointer, and the address of the lambda (`lambda_addr`) is returned.

`compile_lambda`, defined in the same file, executes a couple of actions:

```
    pub fn compile_lambda(
        &mut self,
        arg_names: Vec<SmolStr>,
        body: SExpr,
    ) -> Result<(), CompilationError> {
        self.symbol_table = Rc::new(SymbolTable::new_local(
            self.symbol_table.clone(),
            arg_names.len(),
        ));
        for (arg_idx, arg_name) in arg_names.into_iter().enumerate() {
            let addr = MemAddress::new_local_var(arg_idx);
            self.symbol_table.insert(arg_name, addr);
        }
        let res_addr = self.compile(body)?;
        self.symbol_table = self.symbol_table.parent_table().unwrap();

        let ret_instr = Instruction::new_return(res_addr);
        self.add_instruction(ret_instr);
        Ok(())
    }
```

First, it creates a new `SymbolTable` structure using the shared pointer to the current, top-most `SymbolTable` and the number of arguments that the function receives. This last parameter is important because arguments in miniclj are treated as the first local variables in the `Scope` of the function, and if the user would like to create new local variables inside the function, these must have indexes greater than the index of the last parameter.

Then, the compiler iterates through the argument names, and inserts them in order to the `SymbolTable`. The compiler compiles the body of the function, and finally, destroys the created `SymbolTable`.

Lastly, `compile_lambda` generates a return instruction using the returning address of compiling the body of the function, and appends it to the vector of instructions.

5.2.2. Execution

The execution of a user defined function starts when `VMState` tries to process a call instruction, where the address of the callable points to a `Lambda` value.

```
Value::Lambda(new_instruction_ptr, arity) => {
    let result = self.execute_lambda(new_instruction_ptr, arity, args)?;
    self.store(current_scope, *result_addr, result)?;
    instruction_ptr += 1;
    Ok(())
}
```

In this case, the virtual machine calls its own function `execute_lambda`, which receives the instruction pointer to jump to, the arity of the lambda function and the arguments passed to the function (as values). This function returns the value returned from the lambda, stores it in the parent scope of the function and advances the instruction pointer by 1.

```
pub fn execute_lambda(
    &self,
    new_instruction_ptr: InstructionPtr,
    arity: usize,
    args: Vec<Value>,
) -> RuntimeResult<Value> {
    if args.len() != arity {
        return Err(RuntimeError::WrongArityN(
            "User defined callable",
            arity,
            args.len(),
        ));
    }

    let local_scope = Scope::default();
    for (idx, arg) in args.into_iter().enumerate() {
        self.store(&local_scope, MemAddress::new_local_var(idx), arg)?;
    }

    match self.inner_execute(new_instruction_ptr, &local_scope)? {
        Some(return_address) => self.get(&local_scope, &return_address),
        None => Err(RuntimeError::CompilerError(format!(
            "User defined callable at {} never returned",
            new_instruction_ptr
        ))),
    }
}
```

`execute_lambda` is a simple function: it starts by comparing the number of arguments passed to the function and its arity, and if they aren't equal, the virtual machine returns a `RuntimeError::WrongArity` error.

Then, a new `Scope` structure is created, and the arguments passed to the function are inserted into it.

Finally, `VMState`'s method `inner_execute` is called, which returns a `RuntimeResult<Option<MemAddress»`. The error is handled by the question mark at the end of line 59, and we're left with either a memory address or nothing. In the first case, the value is extracted from the scope using the address, and it is returned by `execute_lambda`. In the second case, `execute_lambda` returns a `RuntimeError::CompilerError`, meaning the compiler has a bug and defined a lambda that never returned a value.

6 Project structure

The project is structured in 4 different folders; 3 Rust crates part of the root workspace, and one Next.js project:

6.1. `miniclj-lib`

This crate stores the main logic for the compiler, virtual machine and the shared code between them. This crate's unit tests are run for every new commit pushed to the main branch of the repo in a GitHub Actions worker, following the continuous integration pipeline described in the file `.github/workflows/ci.yml`.

This crate is divided in the following modules:

- `callables`: Stores the callables available in the language, the base `Callable` trait, and the structure `CallablesTable` which exposes the callables implemented inside
- `compiler`: Stores the mechanisms and structures used specifically during the compilation process (the `CompilerState` struct, the `SymbolTable` enum, the `CompilationError` enum, the `SExpr` enum and the `Literal` enum)
- `constant`: Stores the implementation of the `Constant` enum
- `instruction`: Stores the implementation of the `Instruction` enum
- `memaddress`: Stores the implementation of the `MemAddress` struct
- `parsers`: Stores the parsers generated using `lalrpop` (the `SExprsParser`, `BytecodeParser` and the `NumberLiteralParser`)
- `vm`: Stores the mechanisms and structures used specifically during the execution (the `VMState` struct, the `Scope` struct, the `RuntimeError` enum, the `Value` enum and the `List` enum)

6.2. `miniclj`

This crate stores only a couple of files; it exposes the compiler and vm functionality through a Command Line Interface. This crate compiles to an executable that can be called using the following subcommands for a different function each:

- `check`: Check if a source code file can be correctly parsed

- `ast`: Print the abstract syntax tree from a source code file
- `build`: Compile a source code file into a bytecode file
- `exec`: Execute a bytecode file
- `run`: Compile and execute a source code file

6.3. `miniclj-wasm`

This crate compiles to a binary WebAssembly file, and exposes the functionality of the compiler and vm through JavaScript bindings so that they can be ran in a browser context. It exposes three functions, where each one accepts a string as the input code, and outputs either an structure with the output of the function or an error:

- `ast`: This function prints the abstract syntax tree parsed from the code
- `compile`: This function compiles the code and outputs the corresponding bytecode
- `run`: This function compiles and executes the code, but with the following adaptations for the browser context:
  - `read` calls are executed as `window.prompt` calls, where the browser displays an alert with a text input, which is then redirected to the program
  - `print` and `println` instructions append its output to the global variable `window.minicljoutput`

6.4. `playground`

This folder stores a simple, one page Next.js project where the `miniclj-wasm` is imported and executed for the code written in left side panel, and the output or the error for every function is displayed on the right side panel. The playground is built using GitHub Actions for each commit to the repo, following the continuous delivery pipeline described in the file `.github/workflows/cd.yml`

7 Code examples

7.1. Cyclic factorial function

7.1.1. `miniclj` code

```
(defn factorial [n]
  (loop [x n result 1]
    (if (= x 0)
      result
      (recur (- x 1) (* result x)))))

(println "The factorial of" 15 "is" (factorial 15))
```

7.1.2. Output

```
The factorial of 15 is 1307674368000
Finished in 26ms
```

7.1.3. Bytecode

```
268435456 fn@2@1
268435457 1/1
268435458 true?
268435459 =
268435460 0/1
268435461 -
268435462 *
268435463 println
268435464 "The factorial of"
268435465 15/1
268435466 "is"
***
mov 268435456 536870912
jmp 16
mov 805306368 805306369
mov 268435457 805306370
call 268435459 805306369 268435460 1073741824
call 268435458 1073741824 1073741825
jmpF 1073741825 9
mov 805306370 1073741826
jmp 15
call 268435461 805306369 268435457 1073741827
call 268435462 805306370 805306369 1073741828
mov 1073741827 805306369
mov 1073741828 805306370
jmp 4
mov 1073741829 1073741826
ret 1073741826
call 536870912 268435465 1073741824
call 268435463 268435464 268435465 268435466 1073741824 1073741825
```

7.2. Recursive factorial function

7.2.1. miniclj code

```
(defn factorial [n]
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))
```

```

(println "The factorial of" 15 "is" (factorial 15))
```

7.2.2. Output

```
The factorial of 15 is 1307674368000
Finished in 32ms
```

7.2.3. Bytecode

```
268435456 fn@2@1
268435457 true?
268435458 =
268435459 0/1
268435460 1/1
268435461 *
268435462 -
268435463 println
268435464 "The factorial of"
268435465 15/1
268435466 "is"
***
mov 268435456 536870912
jmp 12
call 268435458 805306368 268435459 1073741824
call 268435457 1073741824 1073741825
jmpF 1073741825 7
mov 268435460 1073741826
jmp 11
call 268435462 805306368 268435460 1073741827
call 536870912 1073741827 1073741828
call 268435461 805306368 1073741828 1073741829
mov 1073741829 1073741826
ret 1073741826
call 536870912 268435465 1073741824
call 268435463 268435464 268435465 268435466 1073741824 1073741825
```

7.3. Factorial function using list generation and transducers

7.3.1. miniclj code

```
(defn factorial [n]
  (if (< n 2)
    1
    (reduce * (range 1 (+ n 1)))))
```

```

(println "The factorial of" 15 "is" (factorial 15))
```

7.3.2. Output

```
The factorial of 15 is 1307674368000
Finished in 12ms
```

7.3.3. Bytecode

```
268435456 fn@2@1
268435457 true?
268435458 <
268435459 2/1
268435460 1/1
268435461 reduce
268435462 *
268435463 range
268435464 +
268435465 println
268435466 "The factorial of"
268435467 15/1
268435468 "is"
***
mov 268435456 536870912
jmp 12
call 268435458 805306368 268435459 1073741824
call 268435457 1073741824 1073741825
jmpF 1073741825 7
mov 268435460 1073741826
jmp 11
call 268435464 805306368 268435460 1073741827
call 268435463 268435460 1073741827 1073741828
call 268435461 268435462 1073741828 1073741829
mov 1073741829 1073741826
ret 1073741826
call 536870912 268435467 1073741824
call 268435465 268435466 268435467 268435468 1073741824 1073741825
```

7.4. Cyclic Fibonacci function

7.4.1. miniclj code

```
(defn fibonacci [n]
  (if (<= n 1)
    n
```

```
    (loop [a 0 b 1 idx 2]
      (if (= idx n)
        (+ a b)
        (recur b (+ a b) (+ idx 1))))))

(println "The Fibonacci number" 15 "is" (fibonacci 15))
```

7.4.2. Output

```
The Fibonacci number 15 is 610
Finished in 27ms
```

7.4.3. Bytecode

```
268435456 fn@2@1
268435457 true?
268435458 <=
268435459 1/1
268435460 0/1
268435461 2/1
268435462 =
268435463 +
268435464 println
268435465 "The Fibonacci number"
268435466 15/1
268435467 "is"
***
mov 268435456 536870912
jmp 25
call 268435458 805306368 268435459 1073741824
call 268435457 1073741824 1073741825
jmpF 1073741825 7
mov 805306368 1073741826
jmp 24
mov 268435460 805306369
mov 268435459 805306370
mov 268435461 805306371
call 268435462 805306371 805306368 1073741827
call 268435457 1073741827 1073741828
jmpF 1073741828 16
call 268435463 805306369 805306370 1073741830
mov 1073741830 1073741829
jmp 23
call 268435463 805306369 805306370 1073741831
call 268435463 805306371 268435459 1073741832
mov 805306370 805306369
mov 1073741831 805306370
mov 1073741832 805306371
jmp 10
```

```
mov 1073741833 1073741829
mov 1073741829 1073741826
ret 1073741826
call 536870912 268435466 1073741824
call 268435464 268435465 268435466 268435467 1073741824 1073741825
```

7.5. Recursive Fibonacci function

7.5.1. miniclj code

```
(defn fibonacci [n]
  (if (<= n 1)
    n
    (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))

(println "The Fibonacci number" 15 "is" (fibonacci 15))
```

7.5.2. Output

```
The Fibonacci number 15 is 610
Finished in 143ms
```

7.5.3. Bytecode

```
268435456 fn@2@1
268435457 true?
268435458 <=
268435459 1/1
268435460 +
268435461 -
268435462 2/1
268435463 println
268435464 "The Fibonacci number"
268435465 15/1
268435466 "is"
***
mov 268435456 536870912
jmp 14
call 268435458 805306368 268435459 1073741824
call 268435457 1073741824 1073741825
jmpF 1073741825 7
mov 805306368 1073741826
jmp 13
call 268435461 805306368 268435459 1073741827
call 536870912 1073741827 1073741828
```

```
call 268435461 805306368 268435462 1073741829
call 536870912 1073741829 1073741830
call 268435460 1073741828 1073741830 1073741831
mov 1073741831 1073741826
ret 1073741826
call 536870912 268435465 1073741824
call 268435463 268435464 268435465 268435466 1073741824 1073741825
```

7.6. Find an element in a list

7.6.1. miniclj code

```
(defn find [val list_v]
  (loop [idx 0 list_v list_v]
    (if (= val (first list_v))
      idx
      (recur (+ idx 1) (rest list_v)))))

(def list_val '(2 6 8 4 3 5))
(println "List:" list_val)
(println "Found element" 3 "in position" (find 3 list_val))
```

7.6.2. Output

```
List: '(2 6 8 4 3 5)
Found element 3 in position 4
Finished in 26ms
```

7.6.3. Bytecode

```
268435456 fn@2@2
268435457 0/1
268435458 true?
268435459 =
268435460 first
268435461 +
268435462 1/1
268435463 rest
268435464 list
268435465 2/1
268435466 6/1
268435467 8/1
268435468 4/1
268435469 3/1
268435470 5/1
```

```
268435471 println
268435472 "List:"
268435473 "Found element"
268435474 "in position"
***
mov 268435456 536870912
jmp 17
mov 268435457 805306370
mov 805306369 805306371
call 268435460 805306371 1073741824
call 268435459 805306368 1073741824 1073741825
call 268435458 1073741825 1073741826
jmpF 1073741826 10
mov 805306370 1073741827
jmp 16
call 268435461 805306370 268435462 1073741828
call 268435463 805306371 1073741829
mov 1073741828 805306370
mov 1073741829 805306371
jmp 4
mov 1073741830 1073741827
ret 1073741827
call 268435464 268435465 268435466 268435467 268435468 268435469 268435470 1073741824
mov 1073741824 536870913
call 268435471 268435472 536870913 1073741825
call 536870912 268435469 536870913 1073741826
call 268435471 268435473 268435469 268435474 1073741826 1073741827
```

7.7. Sorting a list

7.7.1. `miniclj` code

```
(defn frequencies [l]
  (loop [l l result {}]
    (if (empty? l)
      result
      (recur
        (rest l)
        (let [val (first l) n (get result val)]
          (if n
            (conj result [val (+ n 1)])
            (conj result [val 1])))))))

(defn cmp-entry [a b]
  (if (> (first a) (first b))
    a b))

(defn sort-list [l]
  (let [freq-map (frequencies l)]
    (loop [freqs freq-map result '()]
```

```
      (if (empty? freqs)
        result
        (let [max-entry (reduce cmp-entry freqs)
              val (first max-entry)
              freq (first (rest max-entry))]
          (recur
            (if (= freq 1)
              (del freqs val)
              (conj freqs [val (- freq 1)]))
            (cons val result)))))))

(def l '(3 6 1 7 8 2 7))

(println "List:" l)
(println "Sorted list:" (sort-list l))
```

7.7.2. Output

```
List: (3 6 1 7 8 2 7)
Sorted list: (1 2 3 6 7 7 8)
Finished in 35ms
```

7.7.3. Bytecode

```
268435456 fn@2@1
268435457 hash-map
268435458 true?
268435459 empty?
268435460 rest
268435461 first
268435462 get
268435463 conj
268435464 vector
268435465 +
268435466 1/1
268435467 fn@32@2
268435468 >
268435469 fn@43@1
268435470 list
268435471 reduce
268435472 =
268435473 del
268435474 -
268435475 cons
268435476 3/1
268435477 6/1
268435478 7/1
268435479 8/1
268435480 2/1
```

```
268435481 println
268435482 "List:"
268435483 "Sorted list:"
***
mov 268435456 536870912
jmp 30
mov 805306368 805306369
call 268435457 1073741824
mov 1073741824 805306370
call 268435459 805306369 1073741825
call 268435458 1073741825 1073741826
jmpF 1073741826 10
mov 805306370 1073741827
jmp 29
call 268435460 805306369 1073741828
call 268435461 805306369 1073741829
mov 1073741829 805306371
call 268435462 805306370 805306371 1073741830
mov 1073741830 805306372
call 268435458 805306372 1073741831
jmpF 1073741831 22
call 268435465 805306372 268435466 1073741833
call 268435464 805306371 1073741833 1073741834
call 268435463 805306370 1073741834 1073741835
mov 1073741835 1073741832
jmp 25
call 268435464 805306371 268435466 1073741836
call 268435463 805306370 1073741836 1073741837
mov 1073741837 1073741832
mov 1073741828 805306369
mov 1073741832 805306370
jmp 5
mov 1073741838 1073741827
ret 1073741827
mov 268435467 536870913
jmp 41
call 268435461 805306368 1073741824
call 268435461 805306369 1073741825
call 268435468 1073741824 1073741825 1073741826
call 268435458 1073741826 1073741827
jmpF 1073741827 39
mov 805306368 1073741828
jmp 40
mov 805306369 1073741828
ret 1073741828
mov 268435469 536870914
jmp 76
call 536870912 805306368 1073741824
mov 1073741824 805306369
mov 805306369 805306370
call 268435470 1073741825
mov 1073741825 805306371
call 268435459 805306370 1073741826
```

```
call 268435458 1073741826 1073741827
jmpF 1073741827 53
mov 805306371 1073741828
jmp 75
call 268435471 536870913 805306370 1073741829
mov 1073741829 805306372
call 268435461 805306372 1073741830
mov 1073741830 805306373
call 268435460 805306372 1073741831
call 268435461 1073741831 1073741832
mov 1073741832 805306374
call 268435472 805306374 268435466 1073741833
call 268435458 1073741833 1073741834
jmpF 1073741834 66
call 268435473 805306370 805306373 1073741836
mov 1073741836 1073741835
jmp 70
call 268435474 805306374 268435466 1073741837
call 268435464 805306373 1073741837 1073741838
call 268435463 805306370 1073741838 1073741839
mov 1073741839 1073741835
call 268435475 805306373 805306371 1073741840
mov 1073741835 805306370
mov 1073741840 805306371
jmp 48
mov 1073741841 1073741828
ret 1073741828
call 268435470 268435476 268435477 268435466 268435478 268435479 268435480 268435478 1073741824
mov 1073741824 536870915
call 268435481 268435482 536870915 1073741825
call 536870914 536870915 1073741826
call 268435481 268435483 1073741826 1073741827
```

7.8. Matrix multiplication

7.8.1. miniclj code

```
(def matrixA
  '('(3 6 7)
    '(5 -3 0)))

(def matrixB
  '('(1 1)
    '(2 1)
    '(3 -3)))

(defn inc [n] (+ n 1))

(defn pos_matrix_mult [A B idxA idxB len]
  (loop [result 0 idx 0]
```

```
    (if (= idx len)
      result
      (recur
        (+ result (* (nth (nth A idxA) idx) (nth (nth B idx) idxB)))
        (inc idx)))))

(defn matrix_mult [A B]
  (let [dA1 (count A) dA2 (count (first A))
        dB1 (count B) dB2 (count (first B))]
    (loop [idxA 0 idxB 0 result [] row []]
      (if (= idxA dA1)
        result
        (if (= idxB dB2)
          (recur (inc idxA) 0 (conj result row) [])
          (recur idxA (inc idxB) result
            (conj row (pos_matrix_mult A B idxA idxB dA2))))))))

(println "Matrix A:" matrixA)
(println "Matrix B:" matrixB)
(println "A x B:" (matrix_mult matrixA matrixB))
(println "B x A:" (matrix_mult matrixB matrixA))
```

7.8.2. Output

```
Matrix A: ((3 6 7) (5 -3 0))
Matrix B: ((1 1) (2 1) (3 -3))
A x B: [[36 -12] [-1 2]]
B x A: [[8 3 7] [11 9 14] [-6 27 21]]
Finished in 18ms
```

7.8.3. Bytecode

```
268435456 list
268435457 3/1
268435458 6/1
268435459 7/1
268435460 5/1
268435461 -3/1
268435462 0/1
268435463 1/1
268435464 2/1
268435465 fn@11@1
268435466 +
268435467 fn@15@5
268435468 true?
268435469 =
268435470 *
268435471 nth
268435472 fn@36@2
```

```
268435473 count
268435474 first
268435475 vector
268435476 conj
268435477 println
268435478 "Matrix A:"
268435479 "Matrix B:"
268435480 "A x B:"
268435481 "B x A:"
***
call 268435456 268435457 268435458 268435459 1073741824
call 268435456 268435460 268435461 268435462 1073741825
call 268435456 1073741824 1073741825 1073741826
mov 1073741826 536870912
call 268435456 268435463 268435463 1073741827
call 268435456 268435464 268435463 1073741828
call 268435456 268435457 268435461 1073741829
call 268435456 1073741827 1073741828 1073741829 1073741830
mov 1073741830 536870913
mov 268435465 536870914
jmp 13
call 268435466 805306368 268435463 1073741824
ret 1073741824
mov 268435467 536870915
jmp 34
mov 268435462 805306373
mov 268435462 805306374
call 268435469 805306374 805306372 1073741824
call 268435468 1073741824 1073741825
jmpF 1073741825 22
mov 805306373 1073741826
jmp 33
call 268435471 805306368 805306370 1073741827
call 268435471 1073741827 805306374 1073741828
call 268435471 805306369 805306374 1073741829
call 268435471 1073741829 805306371 1073741830
call 268435470 1073741828 1073741830 1073741831
call 268435466 805306373 1073741831 1073741832
call 536870914 805306374 1073741833
mov 1073741832 805306373
mov 1073741833 805306374
jmp 17
mov 1073741834 1073741826
ret 1073741826
mov 268435472 536870916
jmp 81
call 268435473 805306368 1073741824
mov 1073741824 805306370
call 268435474 805306368 1073741825
call 268435473 1073741825 1073741826
mov 1073741826 805306371
call 268435473 805306369 1073741827
mov 1073741827 805306372
```

```
call 268435474 805306369 1073741828
call 268435473 1073741828 1073741829
mov 1073741829 805306373
mov 268435462 805306374
mov 268435462 805306375
call 268435475 1073741830
mov 1073741830 805306376
call 268435475 1073741831
mov 1073741831 805306377
call 268435469 805306374 805306370 1073741832
call 268435468 1073741832 1073741833
jmpF 1073741833 57
mov 805306376 1073741834
jmp 80
call 268435469 805306375 805306373 1073741835
call 268435468 1073741835 1073741836
jmpF 1073741836 70
call 536870914 805306374 1073741838
call 268435476 805306376 805306377 1073741839
call 268435475 1073741840
mov 1073741838 805306374
mov 268435462 805306375
mov 1073741839 805306376
mov 1073741840 805306377
jmp 52
mov 1073741841 1073741837
jmp 79
call 536870914 805306375 1073741842
call 536870915 805306368 805306369 805306374 805306375 805306371 1073741843
call 268435476 805306377 1073741843 1073741844
mov 805306374 805306374
mov 1073741842 805306375
mov 805306376 805306376
mov 1073741844 805306377
jmp 52
mov 1073741845 1073741837
mov 1073741837 1073741834
ret 1073741834
call 268435477 268435478 536870912 1073741831
call 268435477 268435479 536870913 1073741832
call 536870916 536870912 536870913 1073741833
call 268435477 268435480 1073741833 1073741834
call 536870916 536870913 536870912 1073741835
call 268435477 268435481 1073741835 1073741836
```

A Commit log

```
2430716 - Initial commit (2021-09-16)
483bee3 - Implement initial parser (2021-09-16)
c50606b - Create CI pipeline (2021-09-20)
0f91a7b - Implement collections and value::Value (#2) (2021-09-20)
f4ab53e - Discard lints from lalrpop-generated code (2021-09-20)
1b7f8cb - Create factor operations (2021-09-21)
5107491 - Create comparison operations (2021-09-21)
6cdfd3b - Create collection functions (2021-09-22)
a69ff1a - Move Callable trait to callables and rm Collection trait (2021-09-22)
129038c - Replace Atom for Value (2021-09-22)
ae200a2 - Create io functions and display fns (2021-09-22)
080ee70 - Create skeleton for functions (2021-09-23)
9d32972 - Pass scope as argument to callables (2021-09-23)
283912c - Make SExpr a Value (2021-09-23)
f7a4dc9 - Implement first and rest for collections (2021-09-23)
fa905d7 - Callable::call now returns an ExecutionResult (2021-09-23)
fc1b8bf - Implement typecasting functions (2021-09-23)
d361ca0 - Add tests for typecasting functions (2021-09-23)
4b00877 - Impl conditionals and From<i64> and <bool> for Value (2021-09-26)
930589d - Create Callable::arity_err() (2021-09-26)
8bbf4f7 - Implement sequence transform functions (2021-09-26)
11b594f - Change the impl of conj and cons (2021-09-26)
ba4c0d8 - Add predefined functions to the root scope (2021-09-26)
b8eb471 - Eval values before calling functions (2021-09-27)
e833dd3 - SExpr isn't a Value (2021-10-03)
9ddb1c3 - Rename Identifier to Symbol (2021-10-03)
eb3d1f3 - Callables accept a Vec<SExpr> instead of &[Values] (2021-10-03)
84b47c7 - Move SExpr out of src/value/, into src/ (2021-10-03)
e3bc9cc - Callables accept a &Rc<Scope> instead of &Scope (2021-10-03)
2f62b4a - Implement lambdas and lambda creation (2021-10-03)
afba256 - Implement SExpr::eval_inside_list (2021-10-03)
a06628e - StringCast prints nil as an empty string (2021-10-03)
4504117 - Implement def and defn (2021-10-03)
f7ec7ff - Move everything into src/compiler (2021-10-09)
7d921f1 - First structure separation (2021-10-14)
5b7d4c4 - Avance 3 (2021-10-16)
06cd131 - Avance 4 (2021-10-26)
e9f28fb - Implement compilation for numerical functions (2021-10-28)
3d4fcaf - Implement writing the compiled output to a file (2021-10-29)
a7879ef - Implement some simple callables (2021-10-29)
154bf6a - Reorganization (2021-10-30)
688a685 - Ignore dead_code warnings for some functions (2021-10-30)
03018e3 - Install rust 1.56 (2021-10-30)
9473949 - Discard datatypes on compilation, implement lambdas (2021-11-01)
05c4250 - Implement virtual machine (2021-11-02)
a00a081 - Implement executions for some callables (2021-11-02)
51525aa - Implement more callables and abstract lambda execution (2021-11-03)
e6cb289 - Pass the VMState when executing the callables (2021-11-03)
0589ec5 - Fix short lambdas, compile collections (2021-11-04)
179719e - Renaming methods and types (2021-11-04)
```

```
654537b - Implement let, loop and recur (2021-11-08)
cf96398 - Update clap version and use SmolStr for symbols (2021-11-08)
125daaf - Impl list as cons, restore tests, run pedantic clippy (2021-11-08)
fcd6597 - Avance 5 (2021-11-08)
911e192 - Move to workspace structure, create playground (2021-11-11)
c99bdbc - Fix base path for the playground (2021-11-11)
3074b82 - Fix playground, miniclj-wasm outputs to window property (2021-11-11)
263ce60 - Restore overriden bindings in a let closure (2021-11-11)
53513b3 - Avance 6 (2021-11-14)
b5543c3 - Fix playground deployment pipeline (2021-11-14)
c0e9268 - Remove rand and regex direct dependencies (2021-11-17)
abd5c21 - Fix loops inside lambdas (2021-11-19)
f47633e - Implement arity checking during runtime (2021-11-19)
aee5c7e - Improve Display impl for Value (2021-11-19)
45aa658 - Return functions as values (2021-11-19)
6aa57f8 - Sort constants, move cons and conj to modification module (2021-11-20)
77c6a42 - Move loop and recur to cycles module (2021-11-21)
6323708 - Avance 7 (2021-11-21)
7b9eb23 - Add comments, encapsulate parsers into a module (2021-11-21)
bd78aa4 - Start documentation (2021-11-17)
107919b - Write the CompilerState part (2021-11-17)
61f9060 - Create examples, write docs about the VM (2021-11-19)
f5d93ef - Finish code examples (2021-11-20)
82017d0 - Start user manual (2021-11-21)
a68a110 - Finish user manual and design doc (2021-11-21)
356fa09 - Include info about modules in miniclj-lib (2021-11-21)
```

B Weekly logs in Spanish

```
## Avance 1

Por el momento he implementado el 90% del lexer/parser (me falta incorporar la definición de map y mejorar la de set).
También implementé casi 30 funciones que formarán parte de mi lenguaje (están listadas en src/scope.rs, pero algunas tienen como cuerpo un todo!()).
Me falta terminar de implementar unas 5 o 6 funciones, el mecanismo de evaluación de los valores y la transformación de SExprs a sus respectivos tipos de dato.

## Avance 2

El jueves me dí cuenta que en realidad el proyecto es hacer un compilador y no un intérprete, por lo que esta semana y la siguiente me dedicaré a separar la parte del compilador y la parte del intérprete, y para esta entrega moví todo lo que tengo a la parte del compilador, mientras diseño el formato de salida del compilador.
También implementé una interfaz de subcomandos para el ejecutable, e incluí 5 opciones por ahora:
```

```markdown

- check, que imprime un error en caso de que el lexer/parser (y próximamente compilador) encuentren una parte de la entrada que no reconozcan
- ast, para imprimir el árbol de sintaxis de un archivo (si no tiene errores de sintaxis)
- build, para compilar un archivo (por ahora no implementado)
- exec, para ejecutar un archivo compilado (tampoco implementado)
- run, para compilar y ejecutar un archivo (por ahora corre el archivo en el intérprete)

Sobre la semántica básica de variables y el cubo semántico, por ahora sólo tengo un tipo de datos numérico (una fracción de enteros de 64bits), y las operaciones aritméticas no aceptan otros tipos.

## Avance 3

Sigo trabajando en separar el compilador y la máquina virtual del intérprete. En esta entrega empecé a definir el estado del compilador y de los espacios en memoria para así definir una función `State::compile` que reciba una expresión y añada al estado del compilador las expresiones descompuestas de la expresión padre.
Todavía tengo algunas dudas sobre cómo será la estructura de los datos en la tabla de símbolos (qué tengo que guardar y cómo) pero en eso avanzaré la siguiente semana.

## Avance 4

Durante esta semana no avancé tanto como me hubiera gustado, pero definí cómo voy a hacer referencias a la memoria durante la ejecución, y estoy empezando a escribir las partes del compilador que imprimen los cuádruplos. Estoy pensando en hacer el compilador sin tipos, y checar eso en la máquina virtual

## Avance 5

Durante la semana i y la semana pasada avancé hasta casi terminar el proyecto: ya compila y ejecuta funciones, condicionales y ciclos. Por ahora tengo un par de ideas "extras", aunque debería empezar con la documentación:

- Añadir funciones como:
  - spit/slurp (recibe el nombre de un archivo y lo escribe/lee como string)
  - inc/dec (incrementan o decrementan un número por uno)
  - mod (módulo de una división entre dos números)
  - rand/rand-int (devuelven un número decimal o entero aleatorio)
  - range (recibe uno, dos o tres números, como la función de Python regresa una lista de números)
  - repeat (repite un valor n veces)
  - sort/sort-by (ordenan una lista por su valor o por el valor regresado por una función)
  - pow (número elevado a otro número)
  - apply (recibe una función y una lista, llama a la función con los elementos de la lista como argumentos)
  - split (para strings, parte una string por un patrón)
  - min/max (encuentra el mínimo y máximo entre dos números)
  - drop/take (tira o toma los primeros n elementos de una lista)
  - drop-while/take-while (tira o toma los elementos de una lista hasta que la condición se vuelva falsa)
```

```
  - into (castea una collección a otro tipo de collección)
  - -> y ->> (reciben una lista de funciones parciales y las encadenan usando el resultado de la anterior como el primer o último argumento de la siguiente llamada)
- Compilar el proyecto en wasm y hacer una página web "playground" en la que de un lado se pueda escribir el código, y del otro poder ver el árbol de sintaxis, o el bytecode del compilador, o directamente el output de ejecutar el código
- Implementar más tests para las funciones del compilador (sólo +,-,\*,/,=,!=,<,>,<=,>= tienen tests unitarios)

## Avance 6

Al final me decidí por compilar el proyecto a wasm y realizar una página web como "playground" (https://mariojim.github.io/miniclj/) basándome en la página de "playground" de swc (https://play.swc.rs/). Para esto tuve que separ la parte del compilador, máquina virtual y código compartido de la interfaz de línea de comandos, y crear una nueva interfaz para el contexto del navegador.

Esta semana también empecé con la documentación del proyecto. Por ahora la estoy haciendo en LaTeX y en inglés.

## Avance 7

Esta semana avancé principalmente en la documentación del compilador y corregí algunos errores de éste y de la máquina virtual. También reorganicé algunas funciones para que estuvieran en módulos más pequeños.
```

C lalrpop grammar

```
use std::str::FromStr;

use num::Rational64;
use smol_str::SmolStr;

use crate::{
    callables::{Callable, ComparisonOp, FactorOp},
    compiler::{Literal, SExpr},
};

grammar;

// Compiler-specific parsers
pub SExprs = List<SExpr>;

SExpr: SExpr = {
    "(" <SExprs?> ")" => SExpr::Expr(<>.unwrap_or_else(Vec::new)),
```

```
    "#(" <SExprs> ")" => SExpr::ShortLambda(<>),
    "'(" <SExprs?> ")" => SExpr::List(<>.unwrap_or_else(Vec::new)),
    "[" <SExprs?> "]" => SExpr::Vector(<>.unwrap_or_else(Vec::new)),
    "{" <SExprs?> "}" => SExpr::Map(<>.unwrap_or_else(Vec::new)),
    "#{" <SExprs?> "}" => SExpr::Set(<>.unwrap_or_else(Vec::new)),
    Literal => SExpr::Literal(<>),
};

Literal: Literal = {
    "nil" => Literal::Nil,
    Symbol => Literal::Symbol(<>),
    StringLiteral => Literal::String(<>),
    NumberLiteral => Literal::Number(<>),
};

pub NumberLiteral: Rational64 = {
    r"[-]?[0-9]+\.[0-9]+" => {
        let num_parts: Vec<&str> = <>.split(".").collect();
        let integer = i64::from_str(num_parts[0]).unwrap();
        let mut decimals = i64::from_str(num_parts[1]).unwrap();
        if integer < 0 {
            decimals *= -1;
        }
        let exp = num_parts[1].len() as u32;
        let numer = (integer * 10_i64.pow(exp)) + decimals;
        Rational64::new(numer, 10_i64.pow(exp))
    },
    r"[-]?[0-9]+" => Rational64::from_str(<>).unwrap(),
};

// Shared parser rules
List<T>: Vec<T> = {
    <mut v:T*> <e:T> => {
        v.push(e);
        v
    }
};

Symbol: SmolStr = {
    "%" => SmolStr::from("%"),
    ComparisonOp => SmolStr::from(<>.name()),
    FactorOp => SmolStr::from(<>.name()),
    r"[A-Za-z][A-Za-z0-9!?'_-]*" => SmolStr::from(<>),
};

ComparisonOp: ComparisonOp = {
    "=" => ComparisonOp::Eq,
    "!=" => ComparisonOp::Ne,
    ">" => ComparisonOp::Gt,
    "<" => ComparisonOp::Lt,
    ">=" => ComparisonOp::Ge,
    "<=" => ComparisonOp::Le,
};
```

```

FactorOp: FactorOp = {
    "+" => FactorOp::Add,
    "-" => FactorOp::Sub,
    "*" => FactorOp::Mul,
    "/" => FactorOp::Div,
};

StringLiteral: String = r#""[^"]*""# => {
    let mut chars = <>.chars();
    chars.next();
    chars.next_back();
    String::from(chars.as_str())
};
```