Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

TC-3048 Compiler design

miniclj
Final documentation

Mario Emilio Jiménez Vizcaíno
A01173359

Elda Guadalupe Quiroga
Héctor Gibrán Ceballos

November 24th, 2021

Contents

1 About the project

1.1. Project scope

TODO

1.2. Requirements

TODO

1.3. Development process

The development of the language can be tracked from its GitHub repository here: `https://github.com/MarioJim/miniclj`. 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

TODO

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 replaced during compilation) and concurrency primitives (`atom`, `swap!`, `promise`, `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 a couple more collections (vectors, sets and maps) and support for strings as vectors of characters.

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 builtin 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 `src/compiler/error.rs` in the `miniclj-lib` crate.

- `CallableNotDefined`: Returned when the compiler encounters 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 builtin callable)
- `EmptyArgs`: Returned when a expression tried to call a callable with no arguments, and the callalbe expects at least one
- `SymbolNotDefined`: Returned by the compiler when a symbol wasn't defined in the current scope (or any other parent scope)
- `WrongArgument`: 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
- `WrongArity`: Returned when the user tried to call a callable with the wrong number of arguments
- `WrongRecurCall`: Returned when the user tried to call the `recur` function with a different number of arguments than it's corresponding `loop` call

2.3.3. Runtime errors

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

- `CompilerError`: This variant of `RuntimeError` encloses any error that was caused by a compiler malfunction and should be encountered by the user if the compiler has a bug or if the bytecode was modified
- `CouldntParse`: This variant is returned when a value that was passed to a parsing function (like `num` and `chr`) couldn't be correctly processed
- `DivisionByZero`: Returned when the user tries to divide a number by zero
- `IndexOutOfBounds`: 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
- `InvalidMapEntry`: Returned when, inside a function, a value is implicitly casted to a map entry, but the value isn't a vector with two elements
- `IOError`: Returned when a input/output function returned an error instead of correctly printing/reading strings
- `NotACallable`: Returned when a value was tried be executed as a callable, but it wasn't a builtin function nor a user-defined callable
- `WrongArity`: This error has two variants (`WrongArityN` and `WrongArityS`), but both represent the same error: the user tried to call a callable with the wrong number of arguments

- `WrongDataType`: Returned when a callable receives a value with an incorrect datatype, that the callable didn't expect

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. Lexer and Parser

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

- `SExprs`: A list of `SExpr`
- `SExpr`: This rule encompasses the types of expressions `miniclj` accepts:
  - simple expressions (calls to a function with arguments)
  - short lambdas (lambda functions where the argument is a %)
  - lists (implemented as linked lists)
  - vectors (backed by an array)
  - maps (backed by a hashmap)
  - sets (backed by a hashset)
  - simple literals (described the next rule)

  Every expression but the last one accepts a list of s-expressions (rule `SExpr`) between its opening and closing sign.
- `Literal`: This rule has 4 different variants, and each one describes a different type of literal:
  - `"nil"`, the nil value in `miniclj`
  - `Symbol`
  - `StringLiteral`
  - `NumberLiteral`
- `NumberLiteral`: This rule parses a number from a string, and is exposed to the language so that it can be used by calling the builtin function `num`. It has two variants:

  - `r"[-]?[0-9]+.[0-9]+"`: this regular expression describes a decimal number, that is then parsed into a fraction
  - `r"[-]?[0-9]+"`: this regular expression accepts integers, and is also parsed into a fraction with a denominator of 1

- `List<T>`: This is a macro from `lalrpop`, and it is used to parse a list of parsers of type `T` separated by whitespace

- `Symbol`: This rule describes the different type of symbols `miniclj` accepts, and it has 4 variants:
  - `"%"`: The argument for a short lambda
  - `ComparisonOp`
  - `FactorOp`
  - `r"[A-Za-z][A-Za-z0-9!?'_-]*"`: this regular expression accepts most of the characters that can compose a symbol in Clojure, but I chose to discard some of them to simplify my language and reduce the parser conflicts. More information about symbols in Clojure can be found at `https://clojure.org/reference/reader#_symbols`

- `ComparisonOp`: This rule includes the symbols used to compare values in `miniclj`. They are = (equals), `!=` (not equals), > (greater), < (less), >= (greater or equal) and <= (less or equal)

- `FactorOp`: This rule includes the basic math operations: + (addition), - (subtraction), * (multiplication) and / (division)

- `StringLiteral`: This last rule describes a string between double quotes

3.3. Compiler state

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

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 3.4
- `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

3.4. 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.

3.4.1. Constants

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

- `Callable`: Stores a reference to a builtin 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)

3.4.2. 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.

3.4.3. 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. Execution

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

As described earlier, values are stored inside two vectors in structure `Scope`. 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.

5 Code examples

5.1. Cyclic factorial function

5.1.1. minclj 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))
```

5.1.2. Bytecode

```
268435460 0/1
268435465 15/1
268435461 -
268435457 1/1
268435456 fn@2@1
268435459 =
268435464 "The factorial of"
268435466 "is"
268435463 println
268435458 true?
268435462 *
***
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
```

5.1.3. Output

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

5.2. Recursive factorial function

5.2.1. minclj code

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

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

5.2.2. Bytecode

```
268435462 -
268435465 15/1
268435466 "is"
268435463 println
268435460 1/1
268435458 =
268435459 0/1
268435456 fn@2@1
268435461 *
268435457 true?
268435464 "The factorial of"
***
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
```

5.2.3. Output

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

5.3. Cyclic Fibonacci function

5.3.1. minclj 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))
```

5.3.2. Bytecode

```
268435463 +
268435457 true?
268435462 =
268435467 "is"
268435458 <=
268435465 "The Fibonacci number"
268435464 println
268435456 fn@2@1
268435466 15/1
268435460 0/1
268435459 1/1
268435461 2/1
***
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
```

5.3.3. Output

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

5.4. Recursive Fibonacci function

5.4.1. minclj code

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

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

5.4.2. Bytecode

```
268435464 "The Fibonacci number"
268435461 -
268435466 "is"
268435460 +
268435462 2/1
268435459 1/1
268435458 <=
268435457 true?
268435465 15/1
268435456 fn@2@1
268435463 println
***
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
```

5.4.3. Output

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

5.5. Find an element in a list

5.5.1. minclj 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))
```

5.5.2. Bytecode

```
268435461 +
268435456 fn@2@2
268435473 "Found element"
268435466 6/1
268435460 first
268435457 0/1
268435459 =
268435462 1/1
268435465 2/1
268435463 rest
268435470 5/1
268435469 3/1
268435458 true?
268435474 "in position"
268435464 list
268435467 8/1
268435468 4/1
268435472 "List:"
268435471 println
***
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
```

5.5.3. Output

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

5.6. Sorting a list

5.6.1. minclj code

5.6.2. Bytecode

5.6.3. Output

5.7. Matrix multiplication

5.7.1. minclj code

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

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

(defn matrix_mult [A B]
  "TODO")

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

5.7.2. Bytecode

```
268435463 1/1
268435462 0/1
268435460 5/1
268435467 println
268435465 fn@11@2
268435457 3/1
268435469 "Matrix B:"
268435466 "TODO"
268435456 list
268435470 "A x B:"
268435458 6/1
268435468 "Matrix A:"
268435461 -3/1
268435464 2/1
268435459 7/1
***
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 12
ret 268435466
call 268435467 268435468 536870912 1073741831
call 268435467 268435469 536870913 1073741832
call 536870914 536870912 536870913 1073741833
call 268435467 268435470 1073741833 1073741834
```

5.7.3. Output

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`.

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`

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)
```

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
```

```markdown

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:

- 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.
```

C Language 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(<>),
    "#(" <SExprs> ")" => SExpr::ShortLambda(<>),
    "'(" <SExprs> ")" => SExpr::List(<>),
    "[" <SExprs> "]" => SExpr::Vector(<>),
    "{" <SExprs> "}" => SExpr::Map(<>),
    "#{" <SExprs> "}" => SExpr::Set(<>),
    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())
};
```