

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	3
1.1. Project scope	3
1.2. Requirements	3
1.3. Development process	3
1.3.1. Weekly logs	3
1.3.2. Final thoughts	3
2. About the language	3
2.1. Language name	3
2.2. Language features	3
2.3. Errors	4
2.3.1. Parser errors	4
2.3.2. Compiler errors	4
2.3.3. Runtime errors	5
3. About the compiler	5
3.1. Tools and libraries	5
3.2. Lexer and Parser	6
3.3. Compiler state	7
3.4. Bytecode representation	9
3.4.1. Constants	9
3.4.2. Memory addresses	9
3.4.3. Instructions	10
4. About the virtual machine	10
4.1. Tools and libraries	10
4.2. Memory representation	10
5. Code example	11
5.1. Bytecode generated	11
5.2. Terminal output	11
6. Project structure	11
6.1. minicljs-lib	11
6.2. minicljs	11
6.3. minicljs-wasm	12
6.4. playground	12
A. Commit log	12
B. Weekly logs in Spanish	13
C. Language grammar	15
D. CompilerState struct	17

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/minicljl>. 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 `minicljl` 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're replaced during compilation) and concurrency primitives (`atom`, `swap!`, `promise`, `deliver`).

2.2. Language features

`minicljl` 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

These 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

These 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 an expression tried to call a callable with no arguments, and the callable 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 its corresponding `loop` call

2.3.3. Runtime errors

These 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 an input/output function returned an error instead of correctly printing/reading strings
- `NotACallable`: Returned when a value was tried to 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` or in the appendix D.

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 method 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 receives 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 `.mcclj`, 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 unconditional 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

TODO

4.1. Tools and libraries

TODO

4.2. Memory representation

TODO

5 Code example

TODO

5.1. Bytecode generated

TODO

5.2. Terminal output

TODO

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. minicl-j-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.minicl-j-output`

6.4. playground

This folder stores a simple, one page Next.js project where the `minicl-j-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

```
1 2430716 - Initial commit (2021-09-16)
2 483bee3 - Implement initial parser (2021-09-16)
3 c50606b - Create CI pipeline (2021-09-20)
4 0f91a7b - Implement collections and value::Value (#2) (2021-09-20)
5 f4ab53e - Discard lints from lalrpop-generated code (2021-09-20)
6 1b7f8cb - Create factor operations (2021-09-21)
7 5107491 - Create comparison operations (2021-09-21)
8 6cdfd3b - Create collection functions (2021-09-22)
9 a69ff1a - Move Callable trait to callables and rm Collection trait (2021-09-22)
10 129038c - Replace Atom for Value (2021-09-22)
11 ae200a2 - Create io functions and display fns (2021-09-22)
12 080ee70 - Create skeleton for functions (2021-09-23)
13 9d32972 - Pass scope as argument to callables (2021-09-23)
14 283912c - Make SExpr a Value (2021-09-23)
15 f7a4dc9 - Implement first and rest for collections (2021-09-23)
16 fa905d7 - Callable::call now returns an ExecutionResult (2021-09-23)
17 fc1b8bf - Implement typecasting functions (2021-09-23)
18 d361ca0 - Add tests for typecasting functions (2021-09-23)
19 4b00877 - Impl conditionals and From<i64> and <bool> for Value (2021-09-26)
20 930589d - Create Callable::arity_err() (2021-09-26)
```

21 8bbf4f7 - Implement sequence transform functions (2021-09-26)
 22 11b594f - Change the impl of conj and cons (2021-09-26)
 23 ba4c0d8 - Add predefined functions to the root scope (2021-09-26)
 24 b8eb471 - Eval values before calling functions (2021-09-27)
 25 e833dd3 - SExpr isn't a Value (2021-10-03)
 26 9ddb1c3 - Rename Identifier to Symbol (2021-10-03)
 27 eb3d1f3 - Callables accept a Vec<SExpr> instead of &[Values] (2021-10-03)
 28 84b47c7 - Move SExpr out of src/value/, into src/ (2021-10-03)
 29 e3bc9cc - Callables accept a &Rc<Scope> instead of &Scope (2021-10-03)
 30 2f62b4a - Implement lambdas and lambda creation (2021-10-03)
 31 afba256 - Implement SExpr::eval_inside_list (2021-10-03)
 32 a06628e - StringCast prints nil as an empty string (2021-10-03)
 33 4504117 - Implement def and defn (2021-10-03)
 34 f7ec7ff - Move everything into src/compiler (2021-10-09)
 35 7d921f1 - First structure separation (2021-10-14)
 36 5b7d4c4 - Avance 3 (2021-10-16)
 37 06cd131 - Avance 4 (2021-10-26)
 38 e9f28fb - Implement compilation for numerical functions (2021-10-28)
 39 3d4fcfa - Implement writing the compiled output to a file (2021-10-29)
 40 a7879ef - Implement some simple callables (2021-10-29)
 41 154bf6a - Reorganization (2021-10-30)
 42 688a685 - Ignore dead_code warnings for some functions (2021-10-30)
 43 03018e3 - Install rust 1.56 (2021-10-30)
 44 9473949 - Discard datatypes on compilation, implement lambdas (2021-11-01)
 45 05c4250 - Implement virtual machine (2021-11-02)
 46 a00a081 - Implement executions for some callables (2021-11-02)
 47 51525aa - Implement more callables and abstract lambda execution (2021-11-03)
 48 e6cb289 - Pass the VMState when executing the callables (2021-11-03)
 49 0589ec5 - Fix short lambdas, compile collections (2021-11-04)
 50 179719e - Renaming methods and types (2021-11-04)
 51 654537b - Implement let, loop and recur (2021-11-08)
 52 cf96398 - Update clap version and use SmolStr for symbols (2021-11-08)
 53 125daaf - Impl list as cons, restore tests, run pedantic clippy (2021-11-08)
 54 fcd6597 - Avance 5 (2021-11-08)
 55 911e192 - Move to workspace structure, create playground (2021-11-11)
 56 c99bdbc - Fix base path for the playground (2021-11-11)
 57 3074b82 - Fix playground, miniclj-wasm outputs to window property (2021-11-11)
 58 263ce60 - Restore overridden bindings in a let closure (2021-11-11)
 59 53513b3 - Avance 6 (2021-11-14)
 60 b5543c3 - Fix playground deployment pipeline (2021-11-14)

B Weekly logs in Spanish

```

1  ## Avance 1
2
3  Por el momento he implementado el 90% del lexer/parser (me falta incorporar la definición
   ↪ de map y mejorar la de set).
4  También implementé casi 30 funciones que formarán parte de mi lenguaje (están listadas en
   ↪ src/scope.rs, pero algunas tienen como cuerpo un todo!()).
  
```

```

5 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.
6
7 ## Avance 2
8
9 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.
10 También implementé una interfaz de subcomandos para el ejecutable, e incluí 5 opciones
  ↳ por ahora:
11
12 - check, que imprime un error en caso de que el lexer/parser (y próximamente compilador)
  ↳ encuentren una parte de la entrada que no reconozcan
13 - ast, para imprimir el árbol de sintaxis de un archivo (si no tiene errores de sintaxis)
14 - build, para compilar un archivo (por ahora no implementado)
15 - exec, para ejecutar un archivo compilado (tampoco implementado)
16 - run, para compilar y ejecutar un archivo (por ahora corre el archivo en el intérprete)
17
18 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.
19
20 ## Avance 3
21
22 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.
23 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.
24
25 ## Avance 4
26
27 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
28
29 ## Avance 5
30
31 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:
32
33 - Añadir funciones como:
34   - spit/slurp (recibe el nombre de un archivo y lo escribe/lee como string)
35   - inc/dec (incrementan o decrementan un número por uno)
36   - mod (módulo de una división entre dos números)
37   - rand/rand-int (devuelven un número decimal o entero aleatorio)
38   - range (recibe uno, dos o tres números, como la función de Python regresa una lista de
  ↳ números)
39   - repeat (repite un valor n veces)

```

```

40 - sort/sort-by (ordenan una lista por su valor o por el valor regresado por una
    ↪ función)
41 - pow (número elevado a otro número)
42 - apply (recibe una función y una lista, llama a la función con los elementos de la
    ↪ lista como argumentos)
43 - split (para strings, parte una string por un patrón)
44 - min/max (encuentra el mínimo y máximo entre dos números)
45 - drop/take (tira o toma los primeros n elementos de una lista)
46 - drop-while/take-while (tira o toma los elementos de una lista hasta que la condición
    ↪ se vuelva falsa)
47 - into (castea una colección a otro tipo de colección)
48 - -> 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)
49 - 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
50 - Implementar más tests para las funciones del compilador (sólo +,-,\*,/,=,!=,<,>,<=,>=
    ↪ tienen tests unitarios)
51
52 ## Avance 6
53
54 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.
55
56 Esta semana también empecé con la documentación del proyecto. Por ahora la estoy haciendo
    ↪ en LaTeX y en inglés.

```

C Language grammar

```

1  use std::str::FromStr;
2
3  use num::Rational64;
4  use smol_str::SmolStr;
5
6  use crate::{
7      callables::{Callable, ComparisonOp, FactorOp},
8      compiler::{Literal, SExpr},
9  };
10
11 grammar;
12
13 // Compiler-specific parsers
14 pub SExprs = List<SExpr>;
15
16 SExpr: SExpr = {
17     "(" <SExprs> ")" => SExpr::Expr(<>),
18     "#(" <SExprs> ")" => SExpr::ShortLambda(<>),

```

```

19     "(" <SExprs> ")" => SExpr::List(<>),
20     "[" <SExprs> "]" => SExpr::Vector(<>),
21     "{" <SExprs> "}" => SExpr::Map(<>),
22     "#{" <SExprs> "}" => SExpr::Set(<>),
23     Literal => SExpr::Literal(<>),
24 };
25
26 Literal: Literal = {
27     "nil" => Literal::Nil,
28     Symbol => Literal::Symbol(<>),
29     StringLiteral => Literal::String(<>),
30     NumberLiteral => Literal::Number(<>),
31 };
32
33 pub NumberLiteral: Rational64 = {
34     r"[-]?[0-9]+\.[0-9]+" => {
35         let num_parts: Vec<&str> = <>.split(".").collect();
36         let integer = i64::from_str(num_parts[0]).unwrap();
37         let mut decimals = i64::from_str(num_parts[1]).unwrap();
38         if integer < 0 {
39             decimals *= -1;
40         }
41         let exp = num_parts[1].len() as u32;
42         let number = (integer * 10_i64.pow(exp)) + decimals;
43         Rational64::new(number, 10_i64.pow(exp))
44     },
45     r"[-]?[0-9]+" => Rational64::from_str(<>).unwrap(),
46 };
47
48 // Shared parser rules
49 List<T>: Vec<T> = {
50     <mut v:T*> <e:T> => {
51         v.push(e);
52         v
53     }
54 };
55
56 Symbol: SmolStr = {
57     "%" => SmolStr::from("%"),
58     ComparisonOp => SmolStr::from(<>.name()),
59     FactorOp => SmolStr::from(<>.name()),
60     r"[A-Za-z][A-Za-z0-9!'_-]*" => SmolStr::from(<>),
61 };
62
63 ComparisonOp: ComparisonOp = {
64     "=" => ComparisonOp::Eq,
65     "!=" => ComparisonOp::Ne,
66     ">" => ComparisonOp::Gt,
67     "<" => ComparisonOp::Lt,
68     ">=" => ComparisonOp::Ge,
69     "<=" => ComparisonOp::Le,
70 };
71
72 FactorOp: FactorOp = {

```



```

73     "+" => FactorOp::Add,
74     "-" => FactorOp::Sub,
75     "*" => FactorOp::Mul,
76     "/" => FactorOp::Div,
77 };
78
79 StringLiteral: String = r#"^[^"]*"# => {
80     let mut chars = <>.chars();
81     chars.next();
82     chars.next_back();
83     String::from(chars.as_str())
84 };

```

D CompilerState struct

```

1  use std::{collections::HashMap as RustHashMap, io::Write, rc::Rc};
2
3  use smol_str::SmolStr;
4
5  use crate::{
6      callables::{Callable, CallablesTable, HashMap, List, Set, Vector},
7      compiler::{CompilationError, CompilationResult, Literal, SExpr, SymbolTable},
8      constant::Constant,
9      instruction::{Instruction, InstructionPtr},
10     memaddress::{Lifetime, MemAddress},
11 };
12
13 #[derive(Debug, Default)]
14 pub struct CompilerState {
15     constants: RustHashMap<Constant, MemAddress>,
16     instructions: Vec<Instruction>,
17     symbol_table: Rc<SymbolTable>,
18     loop_jumps_stack: Vec<(InstructionPtr, Vec<MemAddress>>>,
19     callables_table: CallablesTable,
20 }
21
22 impl CompilerState {
23     pub fn compile(&mut self, expr: SExpr) -> CompilationResult {
24         match expr {
25             SExpr::Expr(exprs) => {
26                 let mut exprs_iter = exprs.into_iter();
27                 let first_expr = match exprs_iter.next() {
28                     Some(first_expr) => first_expr,
29                     None => return self.compile(SExpr::List(Vec::new())),
30                 };
31                 if let SExpr::Literal(Literal::Symbol(symbol)) = first_expr.clone() {
32                     // Check that there isn't an override for the function
33                     if self.symbol_table.get(&symbol).is_none() {
34                         return match self.callables_table.get(&symbol) {

```

```

35         Some(callable) => callable.compile(self, exprs_iter.collect()),
36         None => Err(CompilationError::CallableNotDefined(symbol)),
37     };
38 }
39 }
40
41 let callable_addr = self.compile(first_expr)?;
42
43 let arg_addrs = exprs_iter
44     .map(|expr| self.compile(expr))
45     .collect::<Result<Vec<MemAddress>, CompilationError>>()?;
46
47 let res_addr = self.new_address(Lifetime::Temporal);
48 let instruction = Instruction::new_call(callable_addr, arg_addrs, res_addr);
49 self.add_instruction(instruction);
50
51 Ok(res_addr)
52 }
53 SExpr::ShortLambda(exprs) => {
54     let jump_lambda_instr = Instruction::new_jump(None);
55     let jump_lambda_instr_ptr = self.add_instruction(jump_lambda_instr);
56     let lambda_start_ptr = self.instruction_ptr();
57     let lambda_const = Constant::new_lambda(lambda_start_ptr, 1);
58     let lambda_addr = self.insert_constant(lambda_const);
59
60     self.compile_lambda(vec![SmolStr::from("%")], SExpr::Expr(exprs))?;
61     self.fill_jump(jump_lambda_instr_ptr, self.instruction_ptr());
62     Ok(lambda_addr)
63 }
64 SExpr::List(exprs) => List.compile(self, exprs),
65 SExpr::Vector(exprs) => Vector.compile(self, exprs),
66 SExpr::Set(exprs) => Set.compile(self, exprs),
67 SExpr::Map(exprs) => HashMap.compile(self, exprs),
68 SExpr::Literal(literal) => {
69     if let Literal::Symbol(symbol) = literal {
70         self.symbol_table
71             .get(&symbol)
72             .ok_or(CompilationError::SymbolNotDefined(symbol))
73     } else {
74         Ok(self.insert_constant(literal.into()))
75     }
76 }
77 }
78 }
79
80 pub fn compile_lambda(
81     &mut self,
82     arg_names: Vec<SmolStr>,
83     body: SExpr,
84 ) -> Result<(), CompilationError> {
85     self.symbol_table = Rc::new(SymbolTable::new_local(self.symbol_table.clone()));
86     for (arg_idx, arg_name) in arg_names.into_iter().enumerate() {
87         let addr = MemAddress::new_local_var(arg_idx);
88         self.symbol_table.insert(arg_name, addr);

```

```

89     }
90     let res_addr = self.compile(body)?;
91     self.symbol_table = self.symbol_table.parent_table().unwrap();
92
93     let ret_instr = Instruction::new_return(res_addr);
94     self.add_instruction(ret_instr);
95     Ok(())
96 }
97
98 pub fn get_symbol(&self, symbol: &str) -> Option<MemAddress> {
99     self.symbol_table.get(symbol)
100 }
101
102 pub fn new_address(&self, lifetime: Lifetime) -> MemAddress {
103     self.symbol_table.new_address(lifetime)
104 }
105
106 pub fn insert_symbol(&self, symbol: SmolStr, address: MemAddress) {
107     self.symbol_table.insert(symbol, address);
108 }
109
110 pub fn remove_symbol(&self, symbol: &str) {
111     self.symbol_table.remove_local(symbol);
112 }
113
114 pub fn insert_constant(&mut self, constant: Constant) -> MemAddress {
115     if let Some(addr) = self.constants.get(&constant) {
116         *addr
117     } else {
118         let next_idx = self
119             .constants
120             .iter()
121             .map(|(_, a)| a.idx() + 1)
122             .max()
123             .unwrap_or(0);
124         let addr = MemAddress::new_const(next_idx);
125         self.constants.insert(constant, addr);
126         addr
127     }
128 }
129
130 /// Returns the index of the next instruction to be inserted
131 pub fn instruction_ptr(&self) -> usize {
132     self.instructions.len()
133 }
134
135 pub fn add_instruction(&mut self, instruction: Instruction) -> InstructionPtr {
136     self.instructions.push(instruction);
137     self.instructions.len() - 1
138 }
139
140 pub fn fill_jump(&mut self, instruction_ptr: InstructionPtr, goto: InstructionPtr) {
141     let instr = self.instructions.get_mut(instruction_ptr).unwrap();
142     match instr {

```

```

143         Instruction::Jump(ptr)
144         | Instruction::JumpOnTrue(_, ptr)
145         | Instruction::JumpOnFalse(_, ptr) => *ptr = goto,
146         _ => panic!("Trying to fill a jump where a different instruction was found"),
147     };
148 }
149
150 pub fn push_loop_jump(&mut self, instruction_ptr: InstructionPtr, addresses: Vec<MemAddress>) {
151     self.loop_jumps_stack.push((instruction_ptr, addresses));
152 }
153
154 pub fn pop_loop_jump(&mut self) -> Option<(InstructionPtr, Vec<MemAddress>)> {
155     self.loop_jumps_stack.pop()
156 }
157
158 pub fn get_callable_addr(&mut self, callable: Box<dyn Callable>) -> MemAddress {
159     self.insert_constant(callable.into())
160 }
161
162 pub fn write_to<T: Write>(&self, writer: &mut T) -> std::io::Result<()> {
163     for (constant, addr) in &self.constants {
164         writer.write_fmt(format_args!("{}", addr, constant))?;
165     }
166     writer.write_all(b"***\n")?;
167     for instruction in &self.instructions {
168         writer.write_fmt(format_args!("{}", instruction))?;
169     }
170     Ok(())
171 }
172
173 pub fn into_parts(self) -> (RustHashMap<Constant, MemAddress>, Vec<Instruction>) {
174     (self.constants, self.instructions)
175 }
176 }

```