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. Bytecode representation

`TODO`

3.4. Memory representation

`TODO`

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. `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

```markdown
## 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:

- 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())
    };
```