miniclj
User Manual

Contents

1 About the language

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

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

1.1 Differences and limitations compared to Clojure

Other than not including a broader standard library compared to Clojure, miniclj has some differences and limitations, like:

- Support for symbols during runtime isn't supported because they must be linked to a memory address during compilation
- Expressions and lists are evaluated eagerly, miniclj doesn't support lazy sequences
- Lambda functions don't capture their enclosing environment/scope
- Support for macros wasn't implemented
- Code is strictly single threaded, and there is no support for using concurrency controls like atoms or promises

2 Data types

2.1 Symbols

Symbols are used to identify values declared in the current scope or in the global scope. They must start with a letter (upper case or lower case), and they can be followed by any number of letters, numbers, or symbols "!", "?", "'", "_" and "-". Other symbols are used for comparison operations (described in section 3.2), factor operations (described in section 3.5), and for the only argument in lambda functions declared using the shorthand form (explained in section 2.9.1).

```
my-var
my_global_var
VaLuE'19_-!?
%
```

2.2 Numbers

Numbers are internally represented as fractions of signed 64 bit integers, using `num`'s `Rational64` structure. Integers are parsed as they are, with a denominator of 1, and decimals are parsed with an power of 10 as a denominator.

```
26
18.34
```

2.3 Strings

Strings are enclosed in double quotes, and they support RFC8259-compliant escaping for unicode characters ("\u0041") and other escape codes such as the newline ("\n"). Strings are stored using Rust' `String`, and therefore adhere to Rust string rules, such as only being constructed from valid UTF-8 characters. In some functions it can be implicitly casted to a list, where its elements are valid UTF-8 chars.

```
"a string"
"a string\n using escaped\n characters"
```

2.4 Nil

A value meaning "nothing" or "no value".

```
nil
```

2.5 Lists

An ordered collection of values represented by a linked list. Insertion and deletion from the front are constant time operations, but searching and getting a value from the middle execute linear time.

```
'()
'(1 2 "string" 3)
```

2.6 Vectors

An ordered collection of values internally represented by a Rust `Vec`. Insertion and deletion from the back, and getting a value using its index execute in constant time.

```
[]
[1 2 3]
```

2.7 Maps

An unordered collection of key-value pairs stored as a Rust HashMap. Insertion, deletion and getting a value by its key are constant time operations.

```
{}
{"key" 23 "another key" 87}
```

2.8 Sets

An unordered collection of values stored as a Rust HashSet. Insertion, deletion and getting a value are constant time operations.

```
#{}
#{"string" 23 87}
```

2.9 Callables

2.9.1 User-defined lambdas

New functions can be declared in two different ways:

- Using the shorthand syntax, in which the body of the function is preceded by a hash symbol (#), and the only argument's name is a percent symbol (%).
- Using the `fn` callable, which expects two arguments: the vector of argument names and the body of the function

```
#(+ % 1)
(fn [arg1 arg2]
    expression)
```

2.9.2 Built-in functions

miniclj includes many different functions, described in the next chapter. Some examples of its uses are:

```
(count collection)
(map #(* % 5) collection)
```

3 Callables

3.1 Collection functions

3.1.1 Access

`first`

```
(first collection)
```

Returns the first item in an ordered collection. In an unordered collection, it returns a random item.

rest

```
(rest collection)
```

Returns a list of the items after the first.

nth

```
(nth collection index)
```

Returns the value of the collection at the index, or throws an `IndexOutOfBounds` runtime error if the index is bigger than the length of the collection.

get

```
(get collection key)
```

For vectors, accepts positive integers as keys and returns the value in that position. For maps, it returns the value of the key passed. For sets it returns the value if it is found. In case the key isn't found, or the first argument isn't any of those collections, it returns `nil`.

count

```
(count collection)
```

Returns the count of items in a collection, or the number of key-value pairs in a map.

empty?

```
(empty? collection)
```

Returns a 0 or 1 number depending on if the collection's length is 0 or greater.

3.1.2 Creation

list

```
(list value1 value2)
```

Used to construct a list. Accepts any number of arguments.

vector

```
(vector value1 value2)
```

Used to construct a vector. Accepts any number of arguments.

set

```
(set value1 value2)
```

Used to construct a set. Accepts any number of arguments.

hash-map

```
(hash-map key1 value2 key2 value2)
```

Used to construct a map. Accepts a pair number of arguments, where the values in odd positions are keys and the values in even positions are used as values for their preceding keys.

3.1.3 Generation

range

```
(range stop)
(range start stop)
(range start stop step)
```

Returns a list of numbers from `start` (inclusive, defaults to 0), to `stop` (exclusive), in steps of size `step` (defaults to 1).

3.1.4 Modification

cons

```
(cons value collection)
```

Creates a new list with the value appended to the start.

conj

```
(conj collection value1 value2)
```

Creates a new collection with the value added to it. In maps it expects a vector of two elements: a key and a value.

del

```
(del collection key1 key2)
```

Creates a new unordered collection with a key removed from it. In maps it removes the key-value pair, and it sets it removes the value.

3.1.5 Transducers

map

```
(map fun collection1 collection2)
```

Returns a list of the results of applying the function `fun` to the first element of every collection, followed by the result of applying the function `fun` to the second element of every collection, and so on until any collection is exhausted. Accepts at least one collection.

filter

```
(filter predicate collection)
```

Returns a list of the elements of the collection where the function `predicate`, applied to the element, returned a truthy value

reduce

```
(reduce accumulator collection)
```

If `collection` is empty, it returns the result of calling the function `accumulator` with no arguments. If `collection` has one element, it returns the element. If `collection` has two or more elements, `reduce` calls `accumulator` with the first two elements, and then with that result and the next element, until there are no more elements.

3.2 Comparison operations

=, !=

```
(= value1 value2)
(!= value1 value2)
```

Checks if two or more values are or aren't equal. If the function receives only one element it returns 1 for = and 0 for `!=`.

>, <, >=, <=

```
(> number1 number2)
(< number1 number2)
(>= number1 number2)
(<= number1 number2)
```

Checks if two or more numbers are in monotonically decreasing order for >, monotonically increasing for <, monotonically non-increasing for >= and monotonically non-decreasing for <=. If the function receives one number it returns 1.

3.3 Conditionals

true?

```
(true? value)
```

Checks if a value is truthy or not. Only falsy values are `nil` and `0`.

if

```
(if condition true-value false-value)
```

Receives three expressions as arguments. If the first argument, `condition`, evaluates to a truthy value, the second argument, `true-value` is evaluated and returned. Otherwise it evaluates and returns the third value, `false-value`.

and

```
(and condition1 condition2)
```

Checks if every condition evaluates to a truthy value. If no conditions are passed, it returns `1`.

or

```
(or condition1 condition2)
```

Checks if at least one condition evaluates to a truthy value. If no conditions are passed, it returns `0`.

3.4 Cycles

loop

```
(loop [symbol1 value1 symbol2 value2]
    expression)
```

Receives two arguments: a vector of symbol-value pairs, saved as local variables, and an expression that can now call `recur` to be evaluated again but with another set of values associated to the initial symbols.

recur

```
(recur value1 value2)
```

Receives as many arguments as the last enclosing `loop` call had symbols. Re-evaluates the last enclosing `loop` call with the values provided as arguments.

3.5 Factor operations

+, -, *, /

```
(+ number1 number2)
(- number1 number2)
(* number1 number2)
(/ number1 number2)
```

Executes the operation specified on the numbers passed as arguments. When passed no arguments:

- + returns 0
- * returns 1
- - and / return a `WrongArity` error

When passed one argument:

- + and * return the argument
- - returns the argument multiplied by -1
- / returns the multiplicative inverse of the argument (1 divided by the argument)

3.6 Grouping functions

do

```
(do
    expression1
    expression2)
```

Evaluates the expressions and returns the result of the last one. When passed no expressions it returns `nil`.

3.7 I/O functions

print, println

```
(print value1 value2)
(println value1 value2)
```

In the CLI version, these functions print to stdout the arguments separated by spaces. In the WASM version, these functions append their output to a variable in the global `window` object named `minicljoutput`. Accepts any number of arguments.

read

```
(read)
```

In the CLI version, this function reads a line from stdin and returns a string. In the WASM version, this function calls `window.prompt` for the user to input a string. This functions accepts no arguments.

3.8 Scope functions

def

```
(def symbol value)
```

Creates a global variable, referred by the identifier `symbol`, with a value of `value`.

defn

```
(defn symbol [argument1 argument2]
    expression)
```

Creates a global user-defined function, referred by the identifier `symbol`. The other two arguments are a vector of argument names and an expression to be evaluated.

let

```
(let [symbol1 value1 symbol2 value2]
    expression)
```

Creates local variables. This callable expects two arguments: a vector of key-value pairs and an expression which can use the variables defined in the vector.

3.9 Typecasting functions

num

```
(num string)
```

Parses a string into a number. Returns a `CouldntParse` error otherwise.

str

```
(str value1 value2)
```

Prints the values to a string. Accepts any number of arguments.

ord

```
(ord string)
```

Returns the numerical value of the first character in the string.

chr

```
(chr number)
```

Returns the UTF-8 character represented by the number, or a `CouldntParse` error in case no character corresponds to that number.