



Janet Language



Janet is a functional and imperative programming language. It runs on Windows, Linux, macOS, BSDs, and should run on other systems with some porting. The entire language (core library, interpreter, compiler, assembler, PEG) is less than 1MB. You can also add Janet scripting to an application by embedding a single C file and two headers.

[Download](#)[Source](#)[Documentation](#)[Core API](#)

Community

Feel free to ask questions and join discussion on the [Janet Gitter Channel](#). Alternatively, check out [the #janet channel on Freenode](#). For help, you can also checkout [Janet Docs](#) for Janet documentation with user-provided examples.

Use Cases

Janet makes a good system scripting language, or a language to embed in other programs. Think Lua or Guile. Janet also can be used for rapid prototyping, dynamic systems, and other domains where dynamic languages shine. Implemented mostly in standard C99, Janet runs on Windows, Linux and macOS. The few features that are not standard C (dynamic library loading, compiler specific optimizations), are fairly straightforward. Janet can be easily ported to new platforms.

Features

- Minimal setup - one binary and you are good to go!
- First class closures
- Garbage collection
- First class green threads (continuations)
- Python style generators (implemented as a plain macro)
- Mutable and immutable arrays (array/tuple)
- Mutable and immutable hashtables (table/struct)
- Mutable and immutable strings (buffer/string)
- Macros
- Byte code interpreter with an assembly interface, as well as bytecode verification
- Tail call optimization
- Direct interop with C via abstract types and C functions
- Dynamically load C libraries
- Functional and imperative standard library
- Lexical scoping
- Imperative and functional programming
- REPL
- Parsing Expression Grammars built in to the core library
- 300+ functions and macros in the core library
- Interactive environment with detailed stack traces
- Export your projects to standalone executables with a companion build tool, jpm

Try It

```
> (print "hello, world!")
```

Usage

A REPL is launched when the `janet` binary is invoked with no arguments. Pass the `-h` flag to display the usage information. Individual scripts can be run with `janet myscript.janet`

If you are looking to explore, you can print a list of all available macros, functions, and constants by entering the command `all-bindings` into the REPL.

```
$ janet
Janet 1.0.0-dev-cc1ff91 Copyright (c) 2017-2019
Calvin Rose
janet:0:> (+ 1 2 3)
6
janet:10:> (print "Hello, world!")
Hello, world!
nil
janet:34:> (os/exit)
$ janet -h
usage: janet [options] script args...
Options are:
  -h : Show this help
  -v : Print the version string
  -s : Use raw stdin instead of getline like
        functionality
  -e code : Execute a string of janet
  -r : Enter the repl after running all scripts
  -p : Keep on executing if there is a top level
        error (persistent)
  -q : Hide prompt, logo, and repl output (quiet)
  -k : Compile scripts but do not execute
  -m syspath : Set system path for loading global
```

```
modules
  -c source output : Compile janet source code
  into an image
  -n : Disable ANSI color output in the repl
  -l path : Execute code in a file before running
  the main script
  -- : Stop handling options
```

Modules and Libraries

See some auxiliary projects on [GitHub](#). Here is a short list of libraries for Janet to help you get started with some interesting stuff. See [the Janet Package Listing](#) for a more complete list. Packages in the listing can be installed via `jpm install pkg-name`.

- [Circlet](#) - An HTTP server for Janet
- [Joy Web Framework](#) - Framework for web development in Janet
- [JSON](#) - A JSON parser and encoder
- [SQLite3](#) - Bindings to SQLite
- [WebView](#) - Spawn a browser window for creating HTML+CSS UIs on any platform
- [Jaylib](#) - Bindings to Raylib for 2d and 3d game development
- [JHydro](#) - Cryptography for Janet
- [JanetUI](#) - Bindings to [libui](#)

For editor support:

- [Conjure](#) - Interactive evaluation for Neovim (Clojure, Fennel, Janet)
- [janet.vim](#) - Support for Janet syntax in Vim
- [vscode-janet](#) - VSCode plugin for Janet
- [ijanet-mode](#) - Emacs interactive Janet mode
- [janet-mode](#) - Janet major mode for Emacs
- [inf-janet](#) - Inferior lisp Janet mode for Emacs
- [language-janet](#) - Atom Editor support for Janet.

© CAIRNII RUST & COMMUNITIES ZUZU

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Introduction

[< The Janet Programming Language](#) [Syntax and the Parser >](#)

Installation

Install a stable version of Janet from the [releases page](#). Janet is prebuilt for a few systems, but if you want to develop Janet, run Janet on a non-x86 system, or get the latest, you must build Janet from source.

Windows

The recommended way to install on Windows is just to run the installer from the releases page. A Scoop package was maintained previously, but ongoing support is being dropped.

If you want to use `jpm` to install native extensions, you will also need to install Visual Studio, ideally a recent release (2019 or 2017 should work well). Then, running `jpm` from the x64 Native Tools Command Prompt should work for a 64-bit build, and from the Developer Command Prompt should work for 32-bit builds. Using these specific command prompts for `jpm` simply lets `jpm` use the Microsoft compiler to compile native modules on install.

Linux

Arch Linux

You can install either the latest git verion or the latest stable version for Arch Linux from the Arch user repositories with [janet-lang-git](#) or [janet-lang](#).

macOS

.....

Janet is available for installation through the [homebrew](#) package manager as `janet`. The latest version from git can be installed by adding the `--HEAD` flag.

```
brew install janet
```

Compiling and running from source

If you would like the latest version of Janet, are trying to run Janet on a platform that is not macOS or Windows, or would like to help develop Janet, you can build Janet from source. Janet only uses Make and batch files to compile on POSIX and Windows respectively. To configure Janet, edit the header file `src/conf/janetconf.h` before compilation.

macOS and Unix-like

On most platforms, use Make to build Janet. The resulting files (including the `janet` binary) will be in the `build/` directory.

```
cd somewhere/my/projects/janet
make
make test
```

After building, run `make install` to install the `janet` binary and libraries. This will install in `/usr/local` by default, see the Makefile to customize.

FreeBSD

FreeBSD build instructions are the same as the Unix-like build instructions, but you need `gmake` and `gcc` to compile.

```
cd somewhere/my/projects/janet  
gmake CC=gcc  
gmake test CC=gcc
```

Windows

1. Install [Visual Studio](#) or [Visual Studio Build Tools](#)
2. Run a Visual Studio Command Prompt (`c1.exe` and `link.exe` need to be on the PATH)
3. `cd` to the directory with Janet
4. Run `build_win` to compile Janet.
5. Run `build_win test` to make sure everything is working.

To install from source, first follow the steps above, then you will need to

1. Install, or otherwise add to your PATH, the [WiX 3.11 Toolset](#)
2. Run a Visual Studio Command Prompt (`c1.exe` and `link.exe` need to be on the PATH)
3. `cd` to the directory with Janet
4. Run `build_win dist`
5. Then, lastly, execute the resulting `.msi` executable

Emscripten

To build Janet for the web via [Emscripten](#), make sure you have `emcc` installed and on your path. On a Linux or macOS system, use `make emscripten` to build `janet.js` and `janet.wasm` - both are needed to run Janet in a browser or in Node.js. The JavaScript build is what runs the REPL on the main website, but really serves mainly as a proof of concept. Janet will run slower in a browser. Building with Emscripten on

Windows is currently unsupported.

Meson

Janet also has a build file for [Meson](#), a cross-platform build system. This is not currently the main supported build system, but should work on any system that supports Meson. Meson also provides much better IDE integration than Make or batch files.

Small builds

If you want to cut down on the size of the final Janet binary or library, you need to omit features and build with `-Os`. With Meson, this can look something like below:

```
git clone https://github.com/janet-lang/janet.git
cd janet
meson setup SmallBuild
cd SmallBuild
meson configure -Dsingle_threaded=true \
-Dassembler=false -Ddocstrings=false \
-Dreduced_os=true -Dttyped_array=false \
Dsourcemaps=false -Dpeg=false \
-Dint_types=false --optimization=s \
Ddebug=false
ninja
# ./janet should be about 40% smaller than the
default build as of 2019-10-13
```

You can also do this with the Makefile by editing `CFLAGS`, and uncommenting some lines in `src/conf/janetconf.h`.

First program

Following tradition, a simple Janet program will print "Hello, world!".

```
(print "Hello, world!")
```

Put the following code in a file named `hello.janet`, and run `janet hello.janet`. The words "Hello, world!" should be printed to the console, and then the program should immediately exit. You now have a working Janet program!

Alternatively, run the program `janet` without any arguments to enter a REPL, or read-eval-print-loop. This is a mode where Janet works like a calculator, reading some input from the user, evaluating it, and printing out the result, all in an infinite loop. This is a useful mode for exploring or prototyping in Janet.

This hello world program is about the simplest program one can write, and consists of only a few pieces of syntax. The first element is the `print` symbol. This is a function that simply prints its arguments to the console. The second argument is the string literal "Hello, world!", which is the one and only argument to the `print` function. Lastly, the `print` symbol and the string literal are wrapped in parentheses, forming a tuple. In Janet, parentheses and brackets are interchangeable, brackets are used mostly when the resulting tuple is not a function call. The tuple above indicates that the function `print` is to be called with one argument, "Hello, world".

All operations in Janet are in prefix notation: the name of the operator is the first value in the tuple, and the arguments passed to it are in the rest

or the tuple. While this may be familiar in function calls, in Janet this idea extends to arithmetic and all control forms.

The core library

Janet has a built in core library of over 300 functions and macros at the time of writing. For efficient programming in Janet, it is good to be able to make use of many of these functions.

If at any time you want to see short documentation on a binding, use the `doc` macro to view the documentation for it in the REPL.

```
(doc defn) # -> Prints the documentation for  
"defn"
```

To see a list of all global functions in the REPL, type the command

```
(all-bindings)
```

Which will print out every global binding in the Janet REPL. You can also browse the [core library API](#) on the website.

[< The Janet Programming Language](#)

[Syntax and the Parser >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Syntax and the Parser

[< Introduction](#)

[Special Forms >](#)

A Janet program begins life as a text file, just a sequence of bytes like any other file on the system. Janet source files should be UTF-8 or ASCII encoded. Before Janet can compile or run the program, it must transform the source code into a data structure. Like Lisp, Janet source code is homoiconic - code is represented as Janet's core data structures - thus all the facilities in the language for the manipulation of tuples, strings and tables can easily be used for manipulation of the source code as well.

However, before Janet code transforms into a data structure, it must be read, or parsed, by the Janet parser. The parser, often called the reader in Lisp, is a machine that takes in plain text and outputs data structures which can be used by both the compiler and macros. In Janet, it is a parser rather than a reader because there is no code execution at reading time. This approach is safer and more straightforward, and makes syntax highlighting, formatting, and other syntax analysis simpler. While a parser is not extensible, in Janet the philosophy is to extend the language via macros rather than reader macros.

nil, true and false

The values `nil`, `true` and `false` are all literals that can be entered as such in the parser.

```
nil  
true  
false
```

Symbols

Janet symbols are represented as a sequence of alphanumeric characters not starting with a digit or a colon. They can also contain the characters !, \@, \$, %, ^, &, *, -, _, +, =, |, ~, :, <, >, ., ?, \, , as well as any Unicode codepoint not in the ASCII range.

By convention, most symbols should be all lower case and use dashes to connect words (sometimes called kebab case).

Symbols that come from another module will typically contain a slash that separates the name of the module from the name of the definition in the module.

```
symbol
kebab-case-symbol
snake_case_symbol
my-module/my-fuction
*****
!%$^*__--__._+++=~-crazy-symbol
*global-var*
你好
```

Keywords

Janet keywords are like symbols that begin with the character `:`. However, they are used differently and treated by the compiler as a constant rather than a name for something. Keywords are used mostly for keys in tables and structs, or pieces of syntax in macros.

```
:keyword  
:range  
:0x0x0x0  
:a-keyword  
::  
:
```

Numbers

Janet numbers are represented by IEEE 754 floating point numbers. The syntax is similar to that of many other languages as well. Numbers can be written in base 10, with underscores used to separate digits into groups. A decimal point can be used for floating point numbers. Numbers can also be written in other bases by prefixing the number with the desired base and the character `r`. For example, 16 can be written as `16`, `1_6`, `16r10`, `4r100`, or `0x10`. The `0x` prefix can be used for hexadecimal as it is so common. The radix must be written in base 10, and can be any integer from 2 to 36. For any radix above 10, use the letters as digits (not case sensitive).

```
0
12
-65912
4.98
1.3e18
1.3E18
18r123C
11raaa&a
1_000_000
0xbeef
```

Strings

Strings in Janet are surrounded by double quotes. Strings are 8-bit clean, meaning they can contain any arbitrary sequence of bytes, including embedded 0s. To insert a double quote into a string itself, escape the double quote with a backslash. For unprintable characters, you can either use one of a few common escapes, use the `\xHH` escape to escape a single byte in hexadecimal. The supported escapes are:

- `\xHH` Escape a single arbitrary byte in hexadecimal.
- `\n` Newline (ASCII 10)
- `\t` Tab character (ASCII 9)
- `\r` Carriage Return (ASCII 13)
- `\0` Null (ASCII 0)
- `\z` Null (ASCII 0)
- `\f` Form Feed (ASCII 12)
- `\e` Escape (ASCII 27)
- `\"` Double Quote (ASCII 34)
- `\\"` Backslash (ASCII 92)

Strings can also contain literal newline characters that will be ignored. This lets one define a multiline string that does not contain newline characters.

Long strings

An alternative way of representing strings in Janet is the long string, or the backquote-delimited string. A string can also be defined to start with a certain number of backquotes, and will end the same number of backquoutes. Long strings do not contain escape sequences; all bytes will

backquotes. Long strings do not contain escape sequences, all bytes will be parsed literally until the ending delimiter is found. This is useful for defining multi-line strings with literal newline characters, unprintable characters, or strings that would otherwise require many escape sequences.

```
"This is a string."  
"This\nis\na\nstring."  
"This  
is  
a  
string."  
``  
This  
is  
a  
string  
``  
`  
This is  
a string.  
`
```

Buffers

Buffers are similar to strings except they are mutable data structures. Strings in Janet cannot be mutated after being created, whereas a buffer can be changed after creation. The syntax for a buffer is the same as that for a string or long string, but the buffer must be prefixed with the @ character.

```
@""  
@"Buffer."  
@``Another buffer``  
@`  
Yet another buffer  
`
```

Tuples

Tuples are a sequence of whitespace separated values surrounded by either parentheses or brackets. The parser considers any of the characters ASCII 32, `\t`, `\f`, `\n`, `\r` or `\t` to be whitespace.

```
(do 1 2 3)  
[do 1 2 3]
```

Square brackets indicate that a tuple will be used as a tuple literal rather than a function call, macro call, or special form. The parser will set a flag on a tuple if it has square brackets to let the compiler know to compile the tuple into a constructor. The programmer can check if a tuple has brackets via the `tuple/type` function.

Arrays

Arrays are the same as tuples, but have a leading `@` to indicate mutability.

```
@(:one :two :three)  
@[ :one :two :three]
```

Structs

Structs are represented by a sequence of whitespace-delimited key-value pairs surrounded by curly braces. The sequence is defined as key1, value1, key2, value2, etc. There must be an even number of items between curly braces or the parser will signal a parse error. Any value can be a key or value. Using `nil` or `NaN` as a key, however, will drop that pair from the parsed struct.

```
{}
{ :key1 "value1" :key2 :value2 :key3 3}
{ (1 2 3) (4 5 6)}
{@[] @[]}
{1 2 3 4 5 6}
```

Tables

Tables have the same syntax as structs, except they have the `@` prefix to indicate that they are mutable.

```
@{}  
{@{:key1 "value1" :key2 :value2 :key3 3}  
@{(1 2 3) (4 5 6)}  
{@[@[] @[]]}  
{@{1 2 3 4 5 6}}
```

Comments

Comments begin with a `#` character and continue until the end of the line. There are no multiline comments.

Shorthand

Often called reader macros in other programming languages, Janet provides several shorthand notations for some forms. In Janet, this syntax is referred to as prefix forms and they are not extensible.

' x

Shorthand for (quote x)

;
x

Shorthand for (splice x)

~
x

Shorthand for (quasiquote x)

,
x

Shorthand for (unquote x)

| (body \$)

Shorthand for (short-fn (body \$))

These shorthand notations can be combined in any order, allowing forms

like `' 'x ((quote (quote x)))`, or `, ;x ((unquote (splice x)))`.

Syntax Highlighting

For syntax highlighting, there is some preliminary Vim syntax highlighting in [janet.vim](#). Generic lisp syntax highlighting should, however, provide good results. One can also generate a `janet.tmLanguage` file for other programs with `make grammar` from the Janet source code.

Grammar

For anyone looking for a more succinct description of the grammar, a PEG grammar for recognizing Janet source code is below. The PEG syntax is itself similar to EBNF. More info on the PEG syntax can be found in the [PEG section](#).

```
(def grammar
~{:ws (set "\t\r\f\n\0\v")
:readermac (set "';~|")
:symchars (+ (range "09" "AZ" "az"
"\x80\xFF") (set "!$%&*+-./:<?=>@^_"))
:token (some :symchars)
:hex (range "09" "af" "AF")
:escape (* "\\\" (+ (set "ntrzfev0\\\\\\\"")
(* "x" :hex :hex)
(error (constant "bad hex
escape"))))
:comment (* "#" (any (if-not (+ "\n" -1) 1)))
:symbol :token
:keyword (* ":" (any :symchars))
:constant (+ "true" "false" "nil")
:bytes (* "\\" (any (+ :escape (if-not "\\" 1))) "\\\"")
:string :bytes
:buffer (* "@" :bytes)
:long-bytes {:delim (some ``)
:open (capture :delim :n)
:close (cmt (* (not (> -1 ``))
(-> :n) ':delim) ,=)
:main (drop (* :open (any (if-
not :close 1)) :close))})
:long-string :long-bytes
```

```
:long-buffer (* "@" :long-bytes)
:number (cmt (<- :token) ,scan-number)
:raw-value (+ :comment :constant :number
:keyword
                           :string :buffer :long-string
:long-buffer
                           :parray :barray :ptuple :btuple
:struct :dict :symbol)
:value (* (any (+ :ws :readermac)) :raw-value
(any :ws))
:root (any :value)
:root2 (any (* :value :value))
:ptuple (* "(" :root (+ ")" (error "")))
:btuple (* "[" :root (+ "]" (error "")))
:struct (* "{" :root2 (+ "}" (error "")))
:parray (* "@" :ptuple)
:barray (* "@" :btuple)
:dict (* "@" :struct)
:main :root})
```

[< Introduction](#)

[Special Forms >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Special Forms

[< Syntax and the Parser](#)

[Numbers and Arithmetic >](#)

This document serves as an overview of all of the special forms in Janet.

Tuples are used to represent function calls, macros, and special forms. Most functionality is exposed through functions, some through macros, and a minimal amount through special forms. Special forms are neither functions nor macros — they are used by the compiler to directly express a low-level construct that cannot be expressed through macros or functions. Special forms can be thought of as forming the real 'core' language of Janet. There are only 12 special forms in Janet.

(def name meta... value)

This special form binds a value to a symbol. The symbol can be substituted for the value in subsequent expressions for the same result. A binding made by `def` is a constant and cannot be updated. A symbol can be redefined to a new value, but previous uses of the binding will refer to the previous value of the binding.

```
(def anumber (+ 1 2 3 4 5))  
(print anumber) # prints 15
```

`def` can also take a tuple, array, table or struct to perform destructuring on the value. This allows us to do multiple assignments in one `def`.

```
(def [a b c] (range 10))  
(print a " " b " " c) # prints 0 1 2  
  
(def {:x x} #{@:x (+ 1 2)})  
(print x) # prints 3  
  
(def [y {:x x}] @[:hi #{@:x (+ 1 2)}])  
(print y x) # prints hi3
```

`def` can also append metadata and a docstring to the symbol when in the global scope. If not in the global scope, the extra metadata will be ignored.

```
(def mydef :private 3) # Adds the :private key to  
the metadata table.  
(def mydef2 :private "A docstring" 4) # Add a
```

```
docstring
```

```
# The metadata will be ignored here because mydef  
is  
# not accessible outside of the do form.  
(do  
  (def mydef :private 3)  
  (+ mydef 1))
```

(var name meta... value)

Similar to `def`, but bindings set in this manner can be updated using `set`. In all other respects it is the same as `def`.

```
(var a 1)
(defn printa [] (print a))

(printa) # prints 1
(++)
(printa) # prints 2
(set a :hi)
(printa) # prints hi
```

(fn name? args body...)

Compile a function literal (closure). A function literal consists of an optional name, an argument list, and a function body. The optional name is allowed so that functions can more easily be recursive. The argument list is a tuple of named parameters, and the body is 0 or more forms. The function will evaluate to the last form in the body. The other forms will only be evaluated for side effects.

Functions also introduce a new lexical scope, meaning the `def`s and `var`s inside a function body will not escape outside the body.

```
(fn []) # The simplest function literal. Takes no arguments and returns nil.  
(fn [x] x) # The identity function  
(fn identity [x] x) # The name will make stacktraces nicer  
(fn [] 1 2 3 4 5) # A function that returns 5  
(fn [x y] (+ x y)) # A function that adds its two arguments.  
  
(fn [& args] (length args)) # A variadic function that counts its arguments.  
  
# A function that doesn't strictly check the number of arguments.  
# Extra arguments are ignored.  
(fn [w x y z &] (tuple w w x x y y z z))
```

For more information on functions, see the [Functions section](#).

(do body...)

Execute a series of forms for side effects and evaluates to the final form.
Also introduces a new lexical scope without creating or calling a function.

```
(do 1 2 3 4) # Evaluates to 4

# Prints 1, 2 and 3, then evaluates to (print 3),
# which is nil
(do (print 1) (print 2) (print 3))

# Prints 1
(do
  (def a 1)
  (print a))

# a is not defined here, so fails
a
```

(quote x)

Evaluates to the literal value of the first argument. The argument is not compiled and is simply used as a constant value in the compiled code. Preceding a form with a single quote is shorthand for (quote expression).

```
(quote 1) # evaluates to 1
(quote hi) # evaluates to the symbol hi
(quote quote) # evaluates to the symbol quote

'(1 2 3) # Evaluates to a tuple (1 2 3)
'(print 1 2 3) # Evaluates to a tuple (print 1 2
3)
```

(if condition when-true when-false?)

Introduce a branching construct. The first form is the condition, the second form is the form to evaluate when the condition is true, and the optional third form is the form to evaluate when the condition is false. If no third form is provided it defaults to `nil`.

The `if` special form will not evaluate the when-true or when-false forms unless it needs to - it is a lazy form, which is why it cannot be a function or macro.

The condition is considered false only if it evaluates to `nil` or `false` - all other values are considered `true`.

```
(if true 10) # evaluates to 10
(if false 10) # evaluates to nil
(if true (print 1) (print 2)) # prints 1 but not
2
(if 0 (print 1) (print 2)) # prints 1
(if nil (print 1) (print 2)) # prints 2
(if @[] (print 1) (print 2)) # prints 1
```

(splice x)

The `splice` special form is an interesting form that allows an array or tuple to be put into another form inline. It only has an effect in two places - as an argument in a function call or literal constructor, or as the argument to the `unquote` form. Outside of these two settings, the `splice` special form simply evaluates directly to its argument `x`. The shorthand for `splice` is prefixing a form with a semicolon. The `splice` special form has no effect on the behavior of other special forms, except as an argument to `unquote`.

In the context of a function call, `splice` will insert the contents of `x` in the parameter list.

```
(+ 1 2 3) # evaluates to 6  
(+ @[1 2 3]) # bad  
(+ (splice @[1 2 3])) # also evaluates to 6  
(+ ;@[1 2 3]) # Same as above  
(+ ;(range 100)) # Sum the first 100 natural numbers  
(+ ;(range 100) 1000) # Sum the first 100 natural numbers and 1000  
[;(range 100)] # First 100 integers in a tuple instead of an array.  
(def ;[a 10]) # this will not work as def is a
```

special form.

Notice that this means we rarely need the `apply` function, as the `splice` operator is more flexible.

A `splice` form can also be used as the argument to an `unquote` form, where it will behave like an `unquote-splicing` special in Common Lisp. Using the short form of both of these specials, this can be abbreviated `, ; some-array-expression`.

(while condition body...)

The `while` special form compiles to a C-like `while` loop. The body of the form will be continuously evaluated until the condition is `false` or `nil`. Therefore, it is expected that the body will contain some side effects or the loop will go on forever. The `while` loop always evaluates to `nil`.

```
(var i 0)
(while (< i 10)
  (print i)
  (++ i))
```

(break value?)

Break from a `while` loop or return early from a function. The `break` special form can only break from the inner-most loop. Since a `while` loop always returns `nil`, the optional `value` parameter has no effect when used in a `while` loop, but when returning from a function, the `value` parameter is the function's return value.

The `break` special form is most useful as a low level construct for macros. You should try to avoid using it in handwritten code, although it can be very useful for handling early exit conditions without requiring deeply indented code (try the `cond` macro first, though).

```
# Breaking from a while loop
}while true
  (def x (math/random))
  (if (> x 0.95) (break))
  (print x))
```

```
# Early exit example using (break)
(fn myfn
 [x]
 (if (= x :one) (break))
 (if (= x :three) (break x))
 (if (and (number? x) (even? x)) (break))
 (print "x = " x)
 x)
```

```
# Example using (cond)
(fn myfn
 [x]
```

```
(cond
  (= x :one) nil
  (= x :three) x
  (and (number? x) (even? x)) nil
  (do
    (print "x = " x)
    x)))
```

(set l-value r-value)

Update the value of the var `l-value` with the new `r-value`. The `set` special form will then evaluate to `r-value`.

The `r-value` can be any expression, and the `l-value` should be a bound var or a pair of a data structure and key. This allows `set` to behave like `setf` or `setq` in Common Lisp.

```
(var x 10)
(defn prx [] (print x))
(prx) # prints 10
(set x 11)
(prx) # prints 11
(set x nil)
(prx) # prints nil

(def tab @{})
(set (tab :property) "hello")
(pp tab) # prints @{:property "hello"}
```

(quasiquote x)

Similar to (quote x), but allows for unquoting within x. This makes quasiquote useful for writing macros, as a macro definition often generates a lot of templated code with a few custom values. The shorthand for quasiquote is a leading tilde ~ before a form. With that form, (unquote x) will evaluate and insert x into the unquote form. The shorthand for (unquote x) is ,x.

(unquote x)

Unquote a form within a `quasiquote`. Outside of a `quasiquote`, `unquote` is invalid.

[< Syntax and the Parser](#)

[Numbers and Arithmetic >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Numbers and Arithmetic

[< Special Forms](#)

[Bindings \(def and var\) >](#)

Any programming language will have some way to do arithmetic. Janet is no exception, and supports the basic arithmetic operators.

```
# Prints 13
# (1 + (2*2) + (10/5) + 3 + 4 + (5 - 6))
(print (+ 1 (* 2 2) (/ 10 5) 3 4 (- 5 6)))
```

Just like the `print` function, all arithmetic operators are entered in prefix notation. Janet also supports the remainder operator, or `%`, which returns the remainder of division. For example, `(% 10 3)` is 1, and `(% 10.5 3)` is 1.5. The lines that begin with `#` are comments.

All Janet numbers are IEEE 754 double precision floating point numbers. They can be used to represent both integers and real numbers to a finite precision.

Numeric literals

Numeric literals can be written in many ways. Numbers can be written in base 10, with underscores used to separate digits into groups. A decimal point can be used for floating point numbers. Numbers can also be written in other bases by prefixing the number with the desired base and the character 'r'. For example, 16 can be written as `16`, `1_6`, `16r10`, `4r100`, or `0x10`. The `0x` prefix can be used for hexadecimal as it is so common. The radix must be themselves written in base 10, and can be any integer from 2 to 36. For any radix above 10, use the letters as digits (not case sensitive).

Numbers can also be in scientific notation such as `3e10`. A custom radix can be used for scientific notation numbers (the exponent will share the radix). For numbers in scientific notation with a radix other than 10, use the `&` symbol to indicate the exponent rather than `e`.

Some example numbers:

```
0
+0.0
-10_000
16r1234abcd
0x23.23
1e10
1.6e-4
7r343_111_266.6&+10 # a base 7 number in
scientific notation.
# evaluates to 1.72625e+13 in base 10
```

Janet will allow some pretty wacky formats for numbers. However, no matter what format you write your number in, the resulting value will

matter what format you write your number in, the resulting value will always be a double precision floating point number.

Arithmetic functions

Besides the 5 main arithmetic functions, Janet also supports a number of math functions taken from the C library `<math.h>`, as well as bit-wise operators that behave like they do in C or Java. Functions like `math/sin`, `math/cos`, `math/log`, and `math/exp` will behave as expected to a C programmer. They all take either 1 or 2 numeric arguments and return a real number (never an integer!). Bit-wise functions are all prefixed with `b`. They are `bnot`, `bor`, `bxor`, `band`, `blshift`, `brshift`, and `brushift`. Bit-wise functions only work on integers.

See the [Math API](#) for information on functions in the `math/` namespace.

[< Special Forms](#)

[Bindings \(def and var\) >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Bindings (def and var)

[< Numbers and Arithmetic](#)

[Flow >](#)

Values can be bound to symbols for later use using the keyword `def`. Using undefined symbols will raise an error.

```
(def a 100)
(def b (+ 1 a))
(def c (+ b b))
(def d (- c 100))
```

Bindings created with `def` have lexical scoping. Additionally, bindings created with `def` are immutable; they cannot be changed after definition. For mutable bindings, like variables in other programming languages, use the `var` keyword. The assignment special form `set` can then be used to update a var.

```
(def a 100)
(var myvar 1)
(print myvar)
(set myvar 10)
(print myvar)
```

In the global scope, you can use the `:private` option on a def or var to prevent it from being exported to code that imports your current module. You can also add documentation to a function by passing a string to the

`def` or `var` command.

```
(def mydef :private "This will have private  
scope. My doc here." 123)  
(var myvar "docstring here" 321)
```

Scopes

Defs and vars (collectively known as bindings) live inside what is called a scope. A scope is simply where the bindings are valid. If a binding is referenced outside of its scope, the compiler will throw an error. Scopes are useful for organizing your bindings and they can expand your programs. There are two main ways to create a scope in Janet.

The first is to use the `do` special form. `do` executes a series of statements in a scope and evaluates to the last statement. Bindings created inside the form do not escape outside of its scope.

```
(def a :outera)  
  
(do  
  (def a 1)  
  (def b 2)  
  (def c 3)  
  (+ a b c)) # -> 6  
  
a # -> :outera  
b # -> compile error: "unknown symbol \"b\""  
c # -> compile error: "unknown symbol \"c\""
```

Any attempt to reference the bindings from the `do` form after it has finished executing will fail. Also notice that defining `a` inside the `do`

form did not overwrite the original definition of `a` for the global scope.

The second way to create a scope is to create a closure. The `fn` special form also introduces a scope just like the `do` special form.

There is another built in macro, `let`, that does multiple `def`s at once, and then introduces a scope. `let` is a wrapper around a combination of `def`s and `do`s, and is the most "functional" way of creating bindings.

```
(let [a 1
      b 2
      c 3]
  (+ a b c)) # -> 6
```

The above is equivalent to the example using `do` and `def`. This is the preferable form in most cases. That said, using `do` with multiple `def`s is fine as well.

[< Numbers and Arithmetic](#)

[Flow >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Flow

[< Bindings \(def and var\)](#)

[Functions >](#)

Janet has only two built in primitives to change flow while inside a function. The first is the `if` special form, which behaves as expected in most functional languages. It takes two or three parameters: a condition, an expression to evaluate to if the condition is Boolean true (ie: not `nil` or `false`), and an optional condition to evaluate to when the condition is `nil` or `false`. If the optional parameter is omitted, the `if` form evaluates to `nil`.

```
(if (> 4 3)
    "4 is greater than 3"
    "4 is not greater than three") # Evaluates to
the first statement

(if true
    (print "Hey")) # Will print

(if false
    (print "Oy!")) # Will not print
```

The second primitive control flow construct is the `while` loop. The `while` form behaves much the same as in many other programming languages, including C, Java, and Python. The `while` loop takes two or more parameters: the first is a condition (like in the `if` statement), that

is checked before every iteration of the loop. If it is `nil` or `false`, the `while` loop ends and evaluates to `nil`. Otherwise, the rest of the parameters will be evaluated sequentially and then the program will return to the beginning of the loop.

```
# Loop from 100 down to 1 and print each time
(var i 100)
(while (pos? i)
  (print "the number is " i)
  (-- i))

# Print ... until a random number in range [0, 1)
is >= 0.9
# (math/random evaluates to a value between 0 and
1)
(while (> 0.9 (math/random)))
  (print "..."))
```

Besides these special forms, Janet has many macros for both conditional testing and looping that are much better for the majority of cases. For conditional testing, the `cond`, `case`, and `when` macros can be used to great effect. `cond` can be used for making an if-else chain, where using just raw `if` forms would result in many parentheses. `case` is similar to `switch` in C without fall-through, or `case` in some Lisps, though simpler. `when` is like `if`, but returns `nil` if the condition evaluates to `nil` or `false`, similar to the macro of the same name in Common Lisp and Clojure. For looping, `loop`, `seq`, and `generate` implement Janet's form of list comprehension, as in Python or Clojure.

[< Bindings \(def and var\)](#)

[Functions >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Functions

[< Flow](#)

[Strings, Keywords, and Symbols >](#)

Janet is a functional language - that means that one of the basic building blocks of your program will be defining functions (the other is using data structures). Because Janet is a Lisp-like language, functions are values just like numbers or strings - they can be passed around and created as needed.

Functions can be defined with the `defn` macro, like so:

```
(defn triangle-area
  "Calculates the area of a triangle."
  [base height]
  (print "calculating area of a triangle...")
  (* base height 0.5))
```

A function defined with `defn` consists of a name, a number of optional flags for `def`, and finally a function body. The example above is named `triangle-area` and takes two parameters named `base` and `height`. The body of the function will print a message and then evaluate to the area of the triangle.

Once a function like the above one is defined, the programmer can use the `triangle-area` function just like any other, say `print` or `+`.

```
# Prints "calculating area of a triangle..." and  
then "25"  
(print (triangle-area 5 10))
```

Note that when nesting function calls in other function calls like above (a call to `triangle-area` is nested inside a call to `print`), the inner function calls are evaluated first. Additionally, arguments to a function call are evaluated in order, from first argument to last argument).

Because functions are first-class values like numbers or strings, they can be passed as arguments to other functions as well.

```
(print triangle-area)
```

This prints the location in memory of the function `triangle-area`.

Functions don't need to have names. The `fn` keyword can be used to introduce function literals without binding them to a symbol.

```
# Evaluates to 40  
((fn [x y] (+ x x y)) 10 20)  
# Also evaluates to 40  
((fn [x y &] (+ x x y)) 10 20)  
  
# Will throw an error about the wrong arity  
((fn [x] x) 1 2)  
# Will not throw an error about the wrong arity  
((fn [x &] x) 1 2)
```

The first expression creates an anonymous function that adds twice the first argument to the second, and then calls that function with arguments 10 and 20. This will return $(10 + 10 + 20) = 40$.

There is a common macro `defn` that can be used for creating functions and immediately binding them to a name. `defn` works as expected at both the top level and inside another form. There is also the corresponding macro `defmacro` that does the same kind of wrapping for macros.

```
(defn myfun [x y]
  (+ x x y))

# You can think of defn as a shorthand for def
and fn together
(def myfun-same (fn [x y]
                  (+ x x Y)))

(myfun 3 4) # -> 10
```

Janet has many macros provided for you (and you can write your own). Macros are just functions that take your source code and transform it into some other source code, usually automating some repetitive pattern for you.

Optional arguments

Most Janet functions will raise an error at runtime if not passed exactly the right number of arguments. Sometimes, you want to define a function with optional arguments, where the arguments take a default value if not supplied by the caller. Janet has support for this via the `&opt` symbol in parameter lists, where all parameters after `&opt` are `nil` if not supplied.

```
(defn my-opt-function
  "A dumb function with optional arguments."
  [a b c &opt d e f]
  (default d 10)
  (default e 11)
  (default f 12)
  (+ a b c d e f))
```

The `(default)` macro is a useful macro for setting default values for parameters. If a parameter is `nil`, `(default)` will redefine it to a default value.

Variadic functions

Janet also provides first-class support for variadic functions. Variadic functions can take any number of parameters, and gather them up into a tuple. To define a variadic function, use the `&` symbol as the second to last item of the parameter list. Parameters defined before the last variadic argument are not optional, unless specified as so with the `&opt` symbol.

```
(defn my-adder
  "Adds numbers in a dubious way."
  [& xs]
  (var accum 0)
  (each x xs
    (+= accum (- (* 2 x) x)))
  accum)

(my-adder 1 2 3) # -> 6
```

Ignoring extra arguments

Sometimes you may want to have a function that can take a number of extra arguments but not use them. This happens because a function may be part of an interface, but the function itself doesn't need those arguments. You can write a function that will simply drop extra parameters by adding & as the last parameter in the function.

```
(defn ignore-extra
  [x &]
  (+ x 1))

(ignore-extra 1 2 3 4 5) # -> 2
```

Keyword-style arguments

Sometimes, you want a function to have many arguments, and calling such a function can get confusing without naming the arguments in the call. One solution to this problem is passing a table or struct with all of the arguments you want to use. This is in general a good approach, as now your original arguments can be identified by the keys in the struct.

```
(defn make-recipe
  "Create some kind of cake recipe..."
  [args]
  (def dry [(args :flour) (args :sugar) (args
  :baking-soda)])
  (def wet [(args :water) (args :eggs) (args
  :vanilla-extract)])
  {:_name "underspecified-cake" :wet wet :dry
  dry})

# Call with an argument struct
(make-recipe
  {:_flour 1
   :sugar 1
   :baking-soda 0.5
   :water 2
   :eggs 2
   :vanilla-extract 0.5})
```

This is often good enough, but there are a couple downsides. The first is that our semantic arguments are not documented, the docstring will just have a single argument "args" which is not helpful. The docstring should have some indication of the keys that we expect in the struct. We also need to write out an extra pair of brackets for the struct - this isn't a huge deal, but it would be nice if we didn't need to write this explicitly.

usual, but it would be nice if we didn't need to write this explicitly.

We can solve this first problem by using destructuring in the argument.

```
(defn make-recipe-2
  "Create some kind of cake recipe with
destructuring..."
  [{:flour flour
    :sugar sugar
    :baking-soda soda
    :water water
    :eggs eggs
    :vanilla-extract vanilla}])
(def dry [flour sugar soda])
(def wet [water eggs vanilla])
{:name "underspecified-cake" :wet wet :dry dry}

# We can call the function in the same manner as
before.
(make-recipe-2
  {:flour 1
   :sugar 1
   :baking-soda 0.5
   :water 2
   :eggs 2
   :vanilla-extract 0.5})
```

The docstring of the improved function will contain a list of arguments that our function takes. To fix the second issue, we can use the `&keys` symbol in a function parameter list to automatically create our big argument struct for use on invocation.

```
(defn make-recipe-3
  "Create some kind of recipe using &keys..."
  [&keys {:flour flour
          :sugar sugar}
```

```

:baking-soda soda
:water water
:eggs eggs
:vanilla-extract vanilla}]

(def dry [flour sugar soda])
(def wet [water eggs vanilla])
{:name "underspecified-cake" :wet wet :dry
dry}

# Calling this function is a bit different now -
no struct
(make-recipe-3
:flour 1
:sugar 1
:baking-soda 0.5
:water 2
:eggs 2
:vanilla-extract 0.5)

```

Usage of this last variant looks cleaner than the previous two recipe-making functions, but there is a caveat. Having all of the arguments packaged as a struct is often useful, as the struct can be passed around and threaded through an application efficiently. Functions that require many arguments often pass those arguments to subroutines, so in practice this issue is quite common. The `&keys` syntax is therefore most useful where terseness and visual clarity is more important.

It is not difficult to convert between the two calling styles, if need be. To pass a struct to a function that expects `&keys`-style arguments, the `kvs` function works well. To convert arguments in the other direction, from `&keys`-style to a single struct argument, is trivial - just wrap your arguments in curly brackets!

```
(def args
```

```
{:flour 1
:sugar 1
:baking-soda 0.5
:water 2
:eggs 2
:vanilla-extract 0.5})

# Turn struct into an array
key,value,key,value,... and splice into a call
(make-recipe-3 ;(kvs args))
```

[< Flow](#)

[Strings, Keywords, and Symbols >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Strings, Keywords, and Symbols

[< Functions](#)

[Looping >](#)

Janet supports several varieties of types that can be used as labels for things in your program. The most useful type for this purpose is the keyword type. A keyword begins with a colon, and then contains 0 or more alphanumeric or a few other common characters. For example, `:hello`, `:my-name`, `::`, and `:ABC123_--*&^%$` are all keywords.

Keywords, symbols, and strings all behave similarly and can be used as keys for tables and structs. Symbols and keywords are optimized for fast equality checks, so are preferred for table keys. Keywords, symbols, and strings are all immutable.

```
# Evaluates to :monday
:monday

# Will throw a compile error as monday is not
defined
monday

# Quote it - evaluates to the symbol monday
'monday

# Our first define, 'monday'
(def monday "It is monday")

# Now the evaluation should work - monday
```

evaluates to "It is monday"
monday

Keywords

The most common thing to do with a keyword is to check it for equality or use it as a key into a table or struct.

```
# Evaluates to true
(= :hello :hello)

# Evaluates to false, everything in janet is case
sensitive
(= :hello :HeLlO)

# Look up into a table - evaluates to 25
(get {
  :name "John"
  :age 25
  :occupation "plumber"
} :age)
```

Symbols

The difference between symbols and keywords is that keywords evaluate to themselves, while symbols evaluate to whatever they are bound to. To have a symbol evaluate to itself, it must be quoted.

Strings

Strings can be used similarly to keywords, but their primary usage is for defining either text or arbitrary sequences of bytes. Strings (and symbols) in Janet are what is sometimes known as “8-bit clean”; they can hold any number of bytes, and are completely unaware of things like character encodings. This is completely compatible with ASCII and UTF-8, two of the most common character encodings. By being encoding agnostic, Janet strings can be simple, fast, and useful for other uses besides holding text.

Literal text can be entered inside quotes, as we have seen above.

```
"Hello, this is a string."  
  
# We can also add escape characters for newlines,  
double quotes, backslash, tabs, etc.  
"Hello\nThis is on line two\n\tThis is  
indented\n"  
  
# For long strings where you don't want to type a  
lot of escape characters,  
# you can use 1 or more backticks (``) to delimit  
a string.  
# To close this string, simply repeat the opening  
sequence of backticks  
``  
This is a string.  
Line 2  
    Indented  
"We can just type quotes here", and backslashes \  
no problem.  
``
```

Strings, symbols, and keywords can all contain embedded UTF-8. It is recommended that you embed UTF-8 literally in strings rather than escaping it if it is printable.

```
"Hello, "
```

Substrings

The `string/slice` function is used to get substrings from a string. Negative integers can be used to index from the end of the string.

```
(string/slice "abcdefg") # -> "abcdefg"  
(string/slice "abcdefg" 1) # -> "bcdefg"  
(string/slice "abcdefg" 2 -2) # -> "cdef"  
(string/slice "abcdefg" -4 -2) # -> "ef"
```

Finding substrings

Janet has multiple functions for finding and replacing strings. The Janet string finding functions do not work on patterns or regular expressions; they only work on string literals. For more flexible searching and replacing, see the [PEG section](#).

```
(string/find "h" "h h h h") #-> 0  
(string/find-all "h" "h h h h") #-> @[0 2 4 6]  
(string/replace "a" "b" "a a a a") #-> "b a a a"  
(string/replace-all "a" "b" "a a a a") #-> "b b b  
b"
```

Splitting strings

The `string/split` function can be used to split strings or buffers on a delimiting character. This can be used as a quick and dirty function for getting fields from a CSV line or words from a sentence.

```
(string/split "," "abc,def,ghi") #-> @["abc"  
"def" "ghi"]  
(string/split " " "abc def ghi") #-> @["abc"  
"def" "ghi"]
```

Concatenating strings

There are many ways to concatenate strings. The first, most common way is the `string` function, which takes any number of arguments and creates a string that is the concatenation of all of the arguments.

```
(string "abc" 123 "def") # -> "abc123def"
```

The second way is the `string/join` function, which takes an array or tuple of strings and joins them together. `string/join` can also take an optional separator string which is inserted between items in the array. All items in the array or tuple must be byte sequences.

```
(string/join @["abc" "123" "def"]) #->  
"abc123def"  
(string/join @["abc" "123" "def"] ",") # ->  
"abc,123,def"
```

This has the advantage over the `string` function that one can specify a separator. Otherwise, the behavior can be easily emulated using the

`splice` special form.

```
(= (string ;@["a" "b" "c"])
  (string/join @["a" "b" "c"])) # -> true
```

Upper and lower case

The string library also provides facilities for converting strings to upper case and lower case. However, only ASCII characters will be transformed. The functions `string/ascii-upper` and `string/ascii-lower` return a new string that has all character in the appropriate case.

```
(string/ascii-upper "aBcdef--*") #-> "ABCDEF--*"
(string/ascii-lower "aBc676A--*") #-> "abc676a--*
```

[< Functions](#)

[Looping >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Looping

[< Strings, Keywords, and Symbols](#)

[Macros >](#)

A very common and essential operation in all programming is looping. Most languages support looping of some kind, either with explicit loops or recursion. Janet supports both recursion and a primitive `while` loop. While recursion is useful in many cases, sometimes it is more convenient to use an explicit loop to iterate over a collection like an array.

Example 1: Iterating a range

Suppose you want to calculate the sum of the first 10 natural numbers 0 through 9. There are many ways to carry out this explicit calculation. A succinct way in Janet is:

```
(+ ;(range 10))
```

We will limit ourselves however to using explicit looping and no functions like `(range n)` which generate a list of natural numbers for us.

For our first version, we will use only the `while` form to iterate, similar to how one might sum natural numbers in a language such as C.

```
(var sum 0)
(var i 0)
(while (< i 10)
  (+= sum i)
  (++ i))
(print sum) # prints 45
```

This is a very imperative program, and it is not the best way to write this in Janet. We are manually updating a counter `i` in a loop. Using the macros `+=` and `++`, this style code is similar in density to C code. It is recommended to instead use either macros (such as the `loop` or `for` macros) or a functional style in Janet.

Since this is such a common pattern, Janet has a macro for this exact purpose. The `(for x start end body)` form captures this behavior of incrementing a counter in a loop.

```
(var sum 0)
(for i 0 10 (+= sum i))
(print sum) # prints 45
```

We have completely wrapped the imperative counter in a macro. The `for` macro, while not very flexible, is very terse and covers a common case of iteration: iterating over an integer range. The `for` macro will be expanded to something very similar to our original version with a `while` loop.

We can do something similar with the more flexible `loop` macro.

```
(var sum 0)
(loop [i :range [0 10]] (+= sum i))
(print sum) # prints 45
```

This is slightly more verbose than the `for` macro, but can be more easily extended. Let's say that we wanted to only count even numbers towards the sum. We can do this easily with the `loop` macro.

```
(var sum 0)
(loop [i :range [0 10] :when (even? i)] (+= sum i))
(print sum) # prints 20
```

The `loop` macro has several verbs (`:range`) and modifiers (`:when`) that let the programmer more easily generate common looping idioms. The `loop` macro is similar to the Common Lips `loop` macro, but smaller in scope and with a much simpler syntax. As with the `for` macro, the `loop` macro expands to similar code as our original `while` expression.

Example 2: Iterating over an indexed data structure

Another common usage for iteration in any language is iterating over the items in a data structure, like items in an array, characters in a string, or key-value pairs in a table.

Say we have an array of names that we want to print out. We will again start with a simple `while` loop which we will refine into more idiomatic expressions.

First, we will define our array of names:

```
(def names
  @["Jean-Paul Sartre"
    "Bob Dylan"
    "Augusta Ada King"
    "Frida Kahlo"
    "Harriet Tubman"])
```

With our array of names, we can use a `while` loop to iterate through the indices of names, get the values, and then print them.

```
(var i 0)
(def len (length names))
(while (< i len)
  (print (get names i))
  (++ i))
```

This is rather verbose. Janet provides the `each` macro for iterating

through the items in a tuple or array, or the bytes in a buffer, symbol, or string.

```
(each name names (print name))
```

We can also use the `loop` macro for this case as well using the `:in` verb.

```
(loop [name :in names] (print name))
```

Lastly, we can use the `map` function to apply a function over each value in the array.

```
(map print names)
```

The `each` macro is actually more flexible than the normal loop, as it is able to iterate over data structures that are not like arrays. For example, `each` will iterate over the values in a table.

Example 3: Iterating a dictionary

In the previous example, we iterated over the values in an array. Another common use of looping in a Janet program is iterating over the keys or values in a table. We cannot use the same method as iterating over an array because a table or struct does not contain a known integer range of keys. Instead we rely on a function `next`, which allows us to visit each of the keys in a struct or table. Note that iterating over a table will not visit the prototype table.

As an example, let's iterate over a table of letters to a word that starts with that letter. We will print out the words to our simple children's book.

```
(def alphabook
  @{"A" "Apple"
    "B" "Banana"
    "C" "Cat"
    "D" "Dog"
    "E" "Elephant"})
```

As before, we can evaluate this loop using only a `while` loop and the `next` function. The `next` function is the primary way to iterate in Janet, and is overloaded to support all iterable types. Given a data structure and a key, it returns the next key in the data structure. If there are no more keys left, it returns `nil`.

```
(var key (next alphabook nil))
(while (not= nil key)
  (print key " is for " (get alphabook key))
  (set key (next alphabook key)))
```

However, we can do better than this with the `loop` macro using the `:pairs` or `:keys` verbs.

```
(loop [[letter word] :pairs alphabook]
      (print letter " is for " word))
```

Using the `:keys` verb and shorthand notation for indexing a data structure:

```
(loop [letter :keys alphabook]
      (print letter " is for " (alphabook letter)))
```

As an alternative to the `loop` macro here, we can also use the macros `eachk` and `eachhp`, which behave like `each` but `loop` over the keys of a data structure and the key-value pairs in a data structure respectively.

Data structures like tables and structs can be called like functions that look up their arguments. This allows for writing shorter code than what is possible with `(get alphabook letter)`.

We can also use the core library functions `keys` and `pairs` to get arrays of the keys and pairs respectively of the alphabook.

```
(loop [[letter word] :in (pairs alphabook)]
      (print letter " is for " word))

(loop [letter :in (keys alphabook)]
      (print letter " is for " (alphabook letter)))
```

Notice that iteration through the table is in no particular order. Iterating the keys of a table or struct guarantees no order. If you want to iterate in

a specific order, use a different data structure or the `(sort indexed)` function.

[< Strings, Keywords, and Symbols](#)

[Macros >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Macros

[< Looping](#)

[Destructuring >](#)

Janet supports macros: routines that take code as input and return transformed code as output. A macro is like a function, but transforms the code itself rather than data, so it is more flexible in what it can do than a function. Macros let you extend the syntax of the language itself.

You have seen some macros already. The `let`, `loop`, and `defn` forms are macros. When the compiler sees a macro, it evaluates the macro and then compiles the result. We say the macro has been expanded after the compiler evaluates it. A simple version of the `defn` macro can be thought of as transforming code of the form

```
(defn1 myfun [x] body)
```

into

```
(def myfun (fn myfun [x] body))
```

We could write such a macro like so:

```
(defmacro defn1 [name args body]
  (tuple 'def name (tuple 'fn name args body)))
```

There are a couple of issues with this macro, but it will work for simple

functions quite well.

The first issue is that our `defn1` macro can't define functions with multiple expressions in the body. We can make the macro variadic, just like a function. Here is a second version of this macro:

```
(defmacro defn2 [name args & body]
  (tuple 'def name (apply tuple 'fn name args
  body)))
```

Great! Now we can define functions with multiple elements in the body.

We can still improve this macro even more though. First, we can add a docstring to it. If someone is using the function later, they can use `(doc defn3)` to get a description of the function. Next, we can rewrite the macro using Janet's builtin quasiquoting facilities.

```
(defmacro defn3
  "Defines a new function."
  [name args & body]
  ~(def ,name (fn ,name ,args ,;body)))
```

This is functionally identical to our previous version `defn2`, but written in such a way that the macro output is more clear. The leading tilde `~` is shorthand for the `(quasiquote x)` special form, which is like `(quote x)` except we can unquote expressions inside it. The comma in front of `name` and `args` is an unquote, which allows us to put a value in the quasiquote. Without the unquote, the symbol `name` would be put in the returned tuple, and every function we defined would be called `name!`

Similar to `name`, we must also unquote `body`. However, a normal unquote doesn't work. See what happens if we use a normal unquote for

body as well.

```
(def name 'myfunction)
(def args '[x y z])
(defn body '[(print x) (print y) (print z)])

~(def ,name (fn ,name ,args ,body))
# -> (def myfunction (fn myfunction (x y z)
((print x) (print y) (print z))))
```

There is an extra set of parentheses around the body of our function! We don't want to put the body inside the form `(fn args ...)`, we want to splice it into the form. Luckily, Janet has the `(splice x)` special form for this purpose, and a shorthand for it, the `;` character. When combined with the unquote special, we get the desired output:

```
~(def ,name (fn ,name ,args ,;body))
# -> (def myfunction (fn myfunction (x y z)
(print x) (print y) (print z)))
```

Hygiene

Sometimes when we write macros, we must generate symbols for local bindings. Ignoring that this could be written as a function, consider the following macro:

```
(defmacro max1
  "Get the max of two values."
  [x y]
  ~(if (> ,x ,y) ,x ,y))
```

This almost works, but will evaluate both `x` and `y` twice. This is because both show up in the macro twice. For example, `(max1 (do (print 1) 1) (do (print 2) 2))` will print both 1 and 2 twice, which would be surprising to a user of this macro.

We can do better:

```
(defmacro max2
  "Get the max of two values."
  [x y]
  ~(let [x ,x
         y ,y]
    (if (> x y) x y)))
```

Now we have no double evaluation problem! But we now have an even more subtle problem. What happens in the following code?

```
(def x 10)
(max2 8 (+ x 4))
```

We want the maximum to be 14, but this will actually evaluate to 12! This can be understood if we expand the macro. You can expand a macro once in Janet using the `(macex1 x)` function. (To expand macros until there are no macros left to expand, use `(macex x)`. Be careful: Janet has many macros, so the full expansion may be almost unreadable).

```
(macex1 '(max2 8 (+ x 4)))
# -> (let (x 8 y (+ x 4)) (if (> x y) x y))
```

After expansion, `y` wrongly refers to the `x` inside the macro (which is bound to 8) rather than the `x` defined to be 10. The problem is the reuse of the symbol `x` inside the macro, which overshadowed the original binding. This problem is called the [hygiene problem](#) and is well known in many programming languages. Some languages provide complicated solutions to this problem, but Janet opts for a much simpler—if not primitive—solution.

Janet provides a general solution to this problem in terms of the `(gensym)` function, which returns a symbol which is guaranteed to be unique and not collide with any symbols defined previously. We can define our macro once more for a fully correct macro.

```
janet
(defmacro max3
  "Get the max of two values."
  [x y]
  (def $x (gensym))
  (def $y (gensym))
  ~(let [,$x ,x
        ,,$y ,y]
    (if (> ,$x ,$y) ,$x ,$y)))
```

Since it is quite common to create several gensyms for use inside a

macro body, Janet provides a macro `with-syms` to make this definition a bit terser.

```
(defmacro max4
  "Get the max of two values."
  [x y]
  (with-syms [$x $y]
    ~(let [,$x ,x
          ,$y ,y]
      (if (> ,$x ,$y) ,$x ,$y))))
```

As you can see, macros are very powerful but are also prone to subtle bugs. You must remember that at their core, macros are just functions that output code, and the code that they return must work in many contexts! Many times a function will suffice and be more useful than a macro, as functions can be more easily passed around and used as first class values.

[< Looping](#)

[Destructuring >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Destructuring

[Macros](#)

[Data Structures](#)

Janet uses the `get` function to retrieve values from inside data structures. In many cases, however, you do not need the `get` function at all. Janet supports destructuring, which means both the `def` and `var` special forms can extract values from inside structures themselves.

```
# Without destructuring, we might do
(def my-array @[:mary :had :a :little :lamb])
(def lamb (get my-array 4))
(print lamb) # Prints :lamb

# Now, with destructuring,
(def [__ __ lamb] my-array)
(print lamb) # Again, prints :lamb

# Destructuring works with tables as well
(def person @{:name "Bob Dylan" :age 77})
(def
  {:name person-name
   :age person-age} person)
(print person-name) # Prints "Bob Dylan"
(print person-age) # Prints 77
```

Destructuring works in many places in Janet, including `let` expressions, function parameters, and `var`. It is a useful shorthand for

extracting values from data structures and is the recommended way to get values out of small structures.

[< Macros](#)

[Data Structures >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Data Structures

[< Destructuring](#)

Once you have a handle on functions and the primitive value types, you might be wondering how to work with collections of things. Janet has a small number of built-in data structure types that are very versatile. Tables, structs, arrays, tuples, strings, and bytes are the 6 main built-in data structure types. These data structures can be summarized in the following table:

Interface	Mutable	Immutable
Indexed	Array	Tuple
Dictionary	Table	Struct
Bytes	Buffer	String (Symbol, Keyword)

Indexed types are linear lists of elements than can be accessed in constant time via an integer index. Indexed types are backed by a single chunk of memory for efficiency. They are indexed from 0 as in C. Dictionary types associate keys with values. The key difference between dictionaries and indexed types is that dictionaries are not limited to integer keys. They are backed by a hash table and also offer constant time lookup (and insertion in the mutable case). Finally, the 'bytes' abstraction is any type that contains a sequence of bytes. A 'bytes' value or byteseq associates integer keys (the indices) with byte values between 0 and 255 (the byte values). In this way, they behave much like a tuple of bytes. However, one cannot put non-integer values into a byteseq.

The table below summarizes the big-O complexity of various operations and information on the built-in data structures. All primitive operations on data structures run in constant time regardless of the number of items in the data structure.

Data Structure	Access	Insert/Append	Delete	Space Complexity
Array *	O(1)	O(1)	O(1)	O(n)
Tuple	O(1)	-	-	O(1)
Table	O(1)	O(1)	O(1)	O(1)
Struct	O(1)	-	-	O(1)
Buffer	O(1)	O(1)	O(1)	O(1)
String/Keyword/Symbol	-	-	-	O(1)

*: Append and delete for an array correspond to `array/push` and `array/pop`. Removing or inserting elements at random indices will run in $O(n)$ time where n is the number of elements in the array.

```
(def mytuple (tuple 1 2 3))

(def myarray @(1 2 3))
(def myarray (array 1 2 3))

(def mystruct {
  :key "value"
  :key2 "another"
  1 2
  4 3})

(def another-struct
```

```
(struct :a 1 :b 2))

(def my-table @{
  :a :b
  :c :d
  :A :qwerty})
(def another-table
  (table 1 2 3 4))

(def my-buffer @"thisismutable")
(def my-buffer2 @``This is also mutable``)
```

To read the values in a data structure, use the `get` function. The first parameter is the data structure itself, and the second parameter is the key. An optional third parameter can be used to specify a default if the value is not found.

```
(get @{:a 1} :a) # -> 1
(get {:a 1} :a) # -> 1
(get @[:a :b :c] 2) # -> :c
(get (tuple "a" "b" "c") 1) # -> "b"
(get @"hello, world" 1) # -> 101
(get "hello, world" 0) # -> 104
(get {:a :b} :a) # -> :b
(get {:a :b} :c :d) # -> :d
```

Similar to the `get` function and added in v1.5.0, the `in` function also gets values contained in a data structure but will throw errors on bad keys to arrays, tuples, and structs. You should prefer `in` going forward as it can be used to better detect tables and structs, `in` behaves identically to `get`.

```
(in @[:a :b :c] 2) # -> :c
(in @[:a :b :c] 3) # -> raises error
```

To update a mutable data structure, use the `put` function. It takes 3 arguments:

structure, the key, and the value, and returns the data structure. The allow and values depend on the data structure passed in.

```
(put @[] 100 :a)
(put @{} :key "value")
(put @"" 100 92)
```

Note that for arrays and buffers, putting an index that is outside the length structure will extend the data structure and fill it with `nil`s in the case of an array or `0`s in the case of the buffer.

The last generic function for all data structures is the `length` function. This function returns the number of values in a data structure (the number of keys in a dictionary type).

[< Destructuring](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Arrays

[< Data Structures](#)

[Buffers >](#)

Arrays are a fundamental datatype in Janet. Arrays are values that contain a sequence of other values, indexed from 0. Arrays are also mutable, meaning that values can be added or removed in place. Many functions in the Janet core library will also create arrays, such as `map`, `filter`, and `interpose`.

Array length

The length of an array is the number of elements in the array. The last element of the array is therefore at index `length - 1`. To get the length of an array, use the `(length x)` function.

Creating arrays

There are many ways to create arrays in Janet, but the easiest is the array literal.

```
(def my-array @[1 2 3 "four"])
(def my-array2 @(1 2 3 "four"))
```

An array literal begins with an at symbol, `@`, followed by square brackets or parentheses with 0 or more values inside.

To create an empty array that you will fill later, use the `(array/new capacity)` function. This creates an array with a reserved capacity for a number of elements. This means that appending elements to the array will not re-allocate the memory in the array. Using an empty array literal would not pre-allocate any space, so the resulting operation would be less efficient.

```
(def arr (array/new 4))
arr # -> @[]
(put arr 0 :one)
(put arr 1 :two)
(put arr 2 :three)
(put arr 3 :four)
arr # -> @[:one :two :three :four]
```

Getting values

Arrays are not of much use without being able to get and set values inside them. To get values from an array, use the `in` or `get` function. `in` will require an index within bounds and will throw an error otherwise. `get` requires an index and an optional default value. If the index is out of bounds it will return `nil` or the given default value.

```
(def arr @[:a :b :c :d])
(in arr 1) # -> :b
(get arr 1) # -> :b
(get arr 100) # -> nil
(get arr 100 :default) # -> :default
```

Instead of `in` you can also use the array as a function with the index as an argument or call the index as a function with the array as the first argument.

```
(arr 2) # -> :c
(0 arr) # -> :a
```

Note that a non-integer key will throw an error when using `in`.

Setting values

To set values in an array, use either the `put` function or the `set` special. The `put` function is a function that allows putting values in any associative data structure. This means that it can associate keys with values for arrays, tables, and buffers. If an index is given that is past the end of the array, the array is first padded with `nil`s so that it is large enough to accommodate the new element.

```
(def arr @[])
(put arr 0 :hello) # -> @[:hello]
(put arr 2 :hello) # -> @[:hello nil :hello]

(set (arr 0) :hi) # -> :hi
arr # -> @[:hi nil :hello]
```

The syntax for the `set` special is slightly different, as it is meant to mirror the syntax for getting an element out of a data structure. Another difference is that while `put` returns the data structure, `set` evaluates to the new value.

Using an array as a stack

Arrays can also be used for implementing efficient stacks. The Janet core library provides three functions that can be used to treat an array as a stack.

- `(array(push stack value))`
- `(array(pop stack))`
- `(array(peek stack))`

`(array(push stack value))`

Appends a value to the end of the array and returns the array.

`(array(pop stack))`

Removes the last value from stack and returns it. If the array is empty, returns `nil`.

`(array(peek stack))`

Returns the last element in stack but does not remove it. Returns `nil` if the stack is empty.

More array functions

There are several more functions in the `array/` namespace and many more functions that create or manipulate arrays in the core library. A short list of the author's favorites are below:

- `array/slice`
- `map`
- `filter`
- `interpose`
- `frequencies`

For documentation on these functions, use the `doc` macro in the REPL or consult the [Array API](#).

[< Data Structures](#)

[Buffers >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Buffers

[< Arrays](#)

[Tables >](#)

Buffers in Janet are the mutable version of strings. Since strings in Janet can hold any sequence of bytes, including zeros, buffers share this same property and can be used to hold any arbitrary memory, which makes them very simple but versatile data structures. They can be used to accumulate small strings into a large string, to implement a bitset, or to represent sound or images in a program.

Creating buffers

A buffer literal looks like a string literal, but prefixed with an at symbol @ .

```
(def my-buf @"This is a buffer.")

(defn make-buffer
  "Creates a buffer"
  []
  @"a new buffer")

(make-buffer) # -> @"a new buffer"
(= (make-buffer) (make-buffer)) # -> false
# All buffers are unique - buffers are equal only
to themselves.
```

Getting bytes from a buffer

To get bytes from a buffer, use the `get` function that works on all built-in data structures. Bytes in a buffer are indexed from 0, and each byte is considered an integer from 0 to 255.

```
(def buf @"abcd")  
  
(get buf 0) # -> 97  
(0 buf) # -> 97  
  
# Use destructuring to print 4  
# bytes of a buffer.  
(let [[b1 b2 b3 b4] buf]  
  (print b1)  
  (print b2)  
  (print b3)  
  (print b4))
```

One can also use the `string/slice` or `buffer/slice` functions to get sub strings (or sub buffers) from inside any sequence of bytes.

```
(def buf @"abcdefg")  
(def buf1 (buffer/slice buf)) # -> @"abcdef", but  
a different buffer  
(def buf2 (buffer/slice buf 2)) # -> @"cdefg"  
(def buf3 (buffer/slice buf 2 -2)) # -> @"cdef"  
  
# string/slice works the same way, but returns  
strings.
```

Setting bytes in a buffer

Buffers are mutable, meaning we can add bytes or change bytes in an already created buffer. Use the `put` function to set individual bytes in a buffer at a given byte index. Buffers will be expanded as needed if an index out of the range provided.

```
(def b @"")
(put b 0 97) #-> @"a"
(put b 10 97) #-> @"a\0\0\0\0\0\0\0\0\0\0a"
```

The `set` special form also works for setting bytes in a buffer.

```
(def b @"")
(set (b 0) 97) #-> @"a"
(set (b 10) 97) #-> @"a\0\0\0\0\0\0\0\0\0\0a"
```

The main difference between `set` and `put` is that `set` will return the byte value, whereas `put` will return the buffer value.

Pushing bytes to a buffer

There are also many functions to push data into a buffer. Since buffers are commonly used to accumulate data, there are a variety of functions for pushing data into a buffer.

- `buffer/push-byte`
- `buffer/push-word`
- `buffer/push-string`

See the documentation for these functions in the [Buffer API](#).

[< Arrays](#)

[Tables >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Tables

[< Buffers](#)

[Structs >](#)

The table is one of the most flexible data structures in Janet and is modeled after the associative array or dictionary. Values are put into a table with a key, and can be looked up later with the same key. Tables are implemented with an underlying open hash table, so they are quite fast and cache friendly.

Any Janet value except `nil` and `Nan` can be a key or a value in a Janet table, and a single Janet table can have any mixture of types as keys and values.

Creating tables

The easiest way to create a table is via a table literal.

```
(def my-tab @{  
  :key1 1  
  "hello" "world!"  
  1 2  
  '(1 0) @{:another :table}})
```

Another way to create a table is via the `table` function. This has the advantage that `table` is an ordinary function that can be passed around like any other. For most cases, a table literal should be preferred.

```
(def my-tab (table  
            :key1 1  
            "hello" "world!"  
            1 2  
            '(1 0) @{:another :table}))
```

Getting and setting values

Like other data structures in Janet, values in a table can be retrieved with the `get` function, and new values in a table can be added via the `put` function or the `set` special. Inserting a value of `nil` into a table removes the key from the table. This means tables cannot contain `nil` values.

```
(def t @{})

(get t :key) #-> nil

(put t :key "hello")
(get t :key) #-> "hello"

(set (t :cheese) :cake)
(get t :cheese) #-> :cake

# Tables can be called as functions
# that look up the argument
(t :cheese) #-> :cake
```

Prototypes

All tables can have a prototype, a parent table that is checked when a key is not found in the first table. By default, tables have no prototype. Prototypes are a flexible mechanism that, among other things, can be used to implement inheritance in Janet. Read more in the [documentation for prototypes](#)

Useful functions for tables

The Janet core library has many useful functions for manipulating tables. Below is a non-exhaustive list.

- frequencies
- keys
- kvs
- length
- merge-into
- merge
- pairs
- post-walk
- pre-walk
- table/getproto
- table/setproto
- update
- values
- walk
- zipcoll

See the [Table API](#) documentation for table-specific functions.

[< Buffers](#)

[Structs >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Structs

[Tables](#)

[Tuples](#)

Structs are immutable data structures that map keys to values. They are semantically similar to [tables](#), but are immutable. They also can be used as keys in tables and other structs, and follow expected equality semantics. Like tables, they are backed by an efficient, native hash table.

To create a struct, you may use a struct literal or the `struct` function.

```
(def my-struct {:key1 2  
                :key2 4})  
  
(def my-struct2 (struct  
                  :key1 2  
                  :key2 4))  
  
# Structs with identical contents are equal  
(= my-struct my-struct2) #-> true
```

As with other data structures, you may retrieve values in a struct via the `get` function, or call the struct as a function. Since structs are immutable, you cannot add key-value pairs

```
(def st {:a 1 :b 2})  
  
(get st :a) #-> 1
```

```
(st :b) #-> 2
```

Converting a table to a struct

A table can be converted to a struct via the `table/to-struct` function. This is useful for efficiently creating a struct by first incrementally putting key-value pairs in a table.

```
(def accum @{})
(for i 0 100
  (put accum (string i) i))
(def my-struct (table/to-struct accum))
```

Using structs as keys

Because a struct is equal to any struct with the same contents, structs make useful keys for tables or other structs. For example, we can use structs to represent Cartesian points and keep a mapping from points to other values.

```
(def points @{
  {:_x 10 :_y 12} "A"
  {:_x 12 :_y 10} "B"
  {:_x 0  :_y 0}   "C"))

(defn get-item
  "Get item at a specific point"
  [x y]
  (points {:_x x :_y y}))

(get-item 10 12) # -> "A"
(get-item 0 0)   # -> "C"
(get-item 5 5)   # -> nil
```

[< Tables](#)

[Tuples >](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Tuples

[< Structs](#)

[Modules >](#)

Tuples are immutable, sequential types that are similar to arrays. They are represented in source code with either parenthesis or brackets surrounding their items. Note that Janet differs from traditional Lisps here, which commonly use the term "lists" to describe forms wrapped in parenthesis. Like all data structures, tuple contents can be retrieved with the `get` function and their length retrieved with the `length` function.

The two most common ways to create a tuple are using the literal form with bracket characters `[]` or calling the `tuple` function directly.

```
# Four ways to create the same tuple:  
  
(def mytup1 [1 2 3 4])          # using bracket  
literals  
(def mytup2 (tuple 1 2 3 4)) # using the (tuple)  
function  
  
# the quote prevents form evaluation and returns  
the tuple directly  
(def mytup3 '(1 2 3 4))  
  
# quasiquote works similarly to quote, but allows  
for some  
# forms to be evaluated inside via the ,  
character
```

```
(def mytup4 ~(1 2 3 , (+ 2 2)))  
  
# these all result in the same tuple:  
(assert (= mytup1 mytup2 mytup3 mytup4)) # true
```

As table keys

Tuples can be used as table keys because two tuples with the same contents are considered equal:

```
(def points @{[0 0] "A"  
           [1 3] "B"  
           [7 5] "C"})  
  
(get points [0 0])          # "A"  
(get points (tuple 0 0))    # "A"  
(get points [1 3])          # "B"  
(get points [8 5])          # nil (ie: not found)
```

Sorting tuples

Tuples can also be used to sort items. When sorting tuples via the `<` or `>` comparators, the first elements are compared first. If those elements are equal, we move on to the second element, then the third, and so on. We could use this property of tuples to sort all kind of data, or sort one array by the contents of another array.

```
(def inventory [  
  ["ermie" 1]  
  ["banana" 18]  
  ["cat" 5]  
  ["dog" 3]  
  ["flamingo" 23]  
  ["apple" 2]])  
  
(def sorted-inventory (sorted inventory))  
  
(each [item n] sorted-inventory (print item ":" "  
n"))  
# apple: 2  
# banana: 18  
# cat: 5  
# dog: 3  
# ermie: 1  
# flamingo: 23
```

Bracketed tuples

Under the hood, there are two kinds of tuples: bracketed and non-bracketed. We have seen above that bracket tuples are used to create a tuple with `[]` characters (ie: a tuple literal). The way a tuple literal is interpreted by the compiler is one of the few ways in which bracketed tuples and non-bracketed tuples differ:

- Bracket tuples are interpreted as a tuple constructor rather than a function call by the compiler.
- When printed via `pp`, bracket tuples are printed with square brackets instead of parentheses.
- When passed as an argument to `tuple/type`, bracket tuples will return `:brackets` instead of `:parens`.

In all other ways, bracketed tuples behave identically to normal tuples. It is not recommended to use bracketed tuples for anything outside of macros and tuple constructors (ie: tuple literals).

More functions

Most functions in the core library that work on arrays also work on tuples, or have an analogous function for tuples. See the [Tuple API](#) for a list of functions that operate on tuples.

[< Structs](#)

[Modules >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Modules

[< Tuples](#)

[Fiber Overview >](#)

As programs grow, they should be broken into smaller pieces for maintainability. Janet has the concept of modules to this end. A module is a collection of bindings and its associated environment. By default, modules correspond one-to-one with source files in Janet, although you may override this and structure modules however you like.

Installing a module

Using `jpm`, the `path` module can be installed like so from the command line:

```
sudo jpm install https://github.com/janet-lang/path.git
```

The use of `sudo` is not required in some setups, but is often needed for a global install on POSIX systems. You can pass in a git repository URL to the `install` command, and `jpm` will install that package globally on your system.

If you are not using `jpm`, you can place the file `path.janet` from the repository in your current directory, and Janet will be able to import it as well. However, for more complicated packages or packages containing native C extensions, `jpm` will usually be much easier.

Importing a module

To use a module, the best way is use the `(import)` macro, which looks for a module with the given name on your system and imports its symbols into the current environment, usually prefixed with the name of the module.

```
(import path)  
(path/join (os/cwd) "temp")
```

Once `path` is imported, all of its symbols are available to the host program, but prefixed with `path/`. To import the symbols with a custom prefix or without any prefix, use the `:prefix` argument to the `import` macro.

```
(import path :prefix "")  
(join (os/cwd) "temp")
```

You may also use the `:as` argument to specify the prefix in a more natural way.

```
(import path :as p)  
(p/join (os/cwd) "temp")
```

Custom loaders (module/paths and module/loaders)

The `module/paths` and `module/loaders` data structures determine how Janet will load a given module name. `module/paths` is a list of patterns and methods to use to try and find a given module. The entries in `module/paths` will be checked in order, as it is possible that multiple entries could match. If a module name matches a pattern (which usually means that some file exists), then we use the corresponding loader in `module/loaders` to evaluate that file, source code, URL, or whatever else we want to derive from the module name. The resulting value, usually an environment table, is then cached so that it can be reused later without evaluating a module twice.

By modifying `module/paths` and `module/loaders`, you can create custom module schemes, handlers for file extensions, or even your own module system. It is recommended to not modify existing entries in `module/paths` or `module/loader`, and simply add to the existing entries. This is rather advanced usage, but can be incredibly useful in creating DSLs that feel native to Janet.

module/paths

This is an array of file paths to search for modules in the file system. Each element in this array is a tuple `[path type predicate]`, where `path` is a templated file path, which determines what path corresponds to a module name, and where `type` is the loading method used to load

the module. `type` can be one of `:native`, `:source`, `:image`, or any key in the `module/loaders` table.

`predicate` is an optional function or file extension used to further filter whether or not a module name should match. It's mainly useful when `path` is a string and you want to further refine the pattern.

`path` can also be a function that checks if a module name matches. If the module name matches, the function should return a string that will be used as the main identifier for the module. Most of the time, this should be the absolute path to the module, but it can be any unique key that identifies the module such as an absolute URL. It is this key that is used to determine if a module has been loaded already. This mechanism lets `./mymod` and `mymod` refer to the same module even though they are different names passed to `import`.

module/loaders

Once a primary module identifier and module type has been chosen, Janet's import machinery (defined mostly in `require` and `module/find`) will use the appropriate loader from `module/loaders` to get an environment table. Each loader in the table is just a function that takes the primary module identifier (usually an absolute path to the module) as well as optionally any other arguments passed to `require` or `import`, and returns the environment table. For example, the `:source` type is a thin wrapper around `dofile`, the `:image` type is a wrapper around `load-image`, and the `:native` type is a wrapper around `native`.

URL loader example

An example from `examples/urlloader.janet` in the source repository shows how a loader can be a wrapper around curl and `dofile` to load source files from HTTP URLs. To use it, simply evaluate this file somewhere in your program before you require code from a URL. Don't use this as is in production code, as if `url` contains spaces in `load-url` the module will not load correctly. A more robust solution would quote the URL, or better yet use `os/execute` and write to a temporary file instead of `file/popen`.

```
(defn- load-url
  [url args]
  (def f (file/popen (string "curl " url)))
  (def res (dofile f :source url ;args))
  (try (file/close f) ([err] nil))
  res)

(defn- check-http-url
  [path]
  (if (or (string/has-prefix? "http://" path)
          (string/has-prefix? "https://" path))
    path))

# Add the module loader and path tuple to right
places
(array(push module/paths [check-http-url :janet-
http])
(put module/loaders :janet-http load-url))
```

Relative imports

You can include files relative to the current file by prefixing the module name with a `./`. For example, if you have a file tree that is structured like so:

```
mymod/
  init.janet
  deps/
    dep1.janet
    dep2.janet
```

With the above structure, `init.janet` can import `dep1` and `dep2` with relative imports. This is less brittle as the entire `mymod/` directory can be installed without any chance that `dep1` and `dep2` will overwrite other files, or that `init.janet` will accidentally import a different file named `dep1.janet` or `dep2.janet`. `mymod` can even be a sub-directory in another Janet source tree and work as expected.

`init.janet`

```
(import ./deps/dep1 :as dep1)
(import ./deps/dep2 :as dep2)

...
```

Writing a module

Writing a module in Janet is mostly about exposing only the public functions that you want users of your module to be able to use. All top level functions defined via `defn`, macros defined `defmacro`, constants defined via `def`, and vars defined via `var` will be exported in your module. To mark a function or binding as private to your module, you may use `defn-` or `def-` at the top level. You can also add the `:private` metadata to the binding.

Sample module:

```
# Put imports and other requisite code up here

(def api-constant
  "Some constant."
  1000)

(def- private-constant
  "Not exported."
  :abc)

(var *api-var*
  "Some API var. Be careful with these, dynamic
bindings may be better."
  nil)

(var *private-var* :private
  "var that is not exported."
  123)

(defn- private-fun
```

```
"Sum three numbers."  
[x y z]  
(+ x y z))  
  
(defn api-fun  
  "Do a thing."  
  [stuff]  
  (+ 10 (private-fun stuff 1 2)))
```

To import our sample module given that it stored on disk at `mymod.janet`, we could do something like the following (this also works in a REPL):

```
(import mymod)  
  
mymod/api-constant # evaluates to 10000  
  
(mymod/api-fun 10) # evaluates to 23  
  
mymod/*api-var* # evaluates to nil  
(set mymod/*api-var* 10)  
mymod/*api-var* # evaluates to 10  
  
(mymod/private-fun 10) # will not compile
```

[< Tuples](#)

[Fiber Overview >](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Fiber Overview

[< Modules](#)

[Dynamic Bindings >](#)

Janet has support for single-core asynchronous programming via coroutines or fibers. Fibers allow a process to stop and resume execution later, essentially enabling multiple returns from a function. This allows many patterns such as schedules, generators, iterators, live debugging, and robust error handling. Janet's error handling is actually built on top of fibers (when an error is thrown, control is returned to the parent fiber).

A temporary return from a fiber is called a yield, and can be invoked with the `yield` function. To resume a fiber that has been yielded, use the `resume` function. When `resume` is called on a fiber, it will only return when that fiber either returns, yields, throws an error, or otherwise emits a signal.

Different from traditional coroutines, Janet's fibers implement a signaling mechanism, which is used to differentiate different kinds of returns. When a fiber yields or throws an error, control is returned to the calling fiber. The parent fiber must then check what kind of state the fiber is in to differentiate errors from return values from user-defined signals.

To create a fiber, user the `fiber/new` function. The fiber constructor take one or two arguments. The first, necessary argument is the function that the fiber will execute. This function must accept an arity of zero. The next optional argument is a collection of flags checking what kinds of

signals to trap and return via `resume`. This is useful so the programmer does not need to handle all different kinds of signals from a fiber. Any untrapped signals are simply propagated to the previous calling fiber.

```
(def f (fiber/new (fn []
                     (yield 1)
                     (yield 2)
                     (yield 3)
                     (yield 4)
                     5)))

# Get the status of the fiber (:alive, :dead,
:debug, :new, :pending, or :user0-:user9)
(print (fiber/status f)) # -> :new

(print (resume f)) # -> prints 1
(print (resume f)) # -> prints 2
(print (resume f)) # -> prints 3
(print (resume f)) # -> prints 4
(print (fiber/status f)) # -> print :pending
(print (resume f)) # -> prints 5
(print (fiber/status f)) # -> print :dead
(print (resume f)) # -> throws an error because
the fiber is dead
```

Using fibers to capture errors

Besides being used as coroutines, fibers can be used to implement error handling (ie: exceptions).

```
(defn my-function-that-errors []
  (print "start function")
  (error "oops!")
  (print "never gets here"))

# Use the :e flag to only trap errors.
(def f (fiber/new my-function-that-errors :e))
(def result (resume f))
(if (= (fiber/status f) :error)
  (print "result contains the error")
  (print "result contains the good result"))
```

Since the above code is rather verbose, Janet provides a `try` macro which is similar to try/catch in other languages. It wraps the body in a new fiber, `resumes` the fiber, and then checks the result. If the fiber has errored, an error clause is evaluated.

```
(try
(error 1)
([err] (print "got error: " err)))
# evaluates to nil and prints "got error: 1"

(try
(+ 1 2 3)
([err] (print "oops")))
# Evaluates to 6 - no error thrown
```

[< Modules](#)

[Dynamic Bindings >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Dynamic Bindings

[< Fiber Overview](#)

[Error Handling >](#)

There are situations where the programmer would like to thread a parameter through multiple function calls, without passing that argument to every function explicitly. This can make code more concise, easier to read, and easier to extend. Dynamic bindings are a mechanism that provide this in a safe and easy to use way. This is in contrast to lexically-scoped bindings, which are usually superior to dynamically-scoped bindings in terms of clarity, composability, and performance. However, dynamic scoping can be used to great effect for implicit contexts, configuration, and testing. Janet supports dynamic scoping as of version 0.5.0 on a per-fiber basis — each fiber contains an environment table that can be queried for values. Using table prototypes, we can easily emulate dynamic scoping.

Setting a value

To set a dynamic binding, use the `setdyn` function.

```
# Sets a dynamic binding :my-var to 10 in the
current fiber.
(setdyn :my-var 10)
```

Getting a value

To get a dynamically-scoped binding, use the `dyn` function.

```
(dyn :my-var) # returns nil  
(setdyn :my-var 10)  
(dyn :my-var) # returns 10
```

Creating a dynamic scope

Now that we can get and set dynamic bindings, we need to know how to create dynamic scopes themselves. To do this, we can create a new fiber and then use `fiber/setenv` to set the dynamic environment of the fiber. To inherit from the current environment, we set the prototype of the new environment table to the current environment table.

Below, we set the dynamic binding `:pretty-format` to configure the pretty print function `pp`.

```
# Body of our new fiber
(defn myblock
  []
  (pp [1 2 3]))

# The current env
(def curr-env (fiber/getenv (fiber/current)))

# The dynamic bindings we want to use
(def my-env {:pretty-format "Inside myblock:
%.20P"})

# Set up a new fiber
(def f (fiber/new myblock))
(fiber/setenv f (table/setproto my-env curr-env))

# Run the code
(pp [1 2 3]) # prints "[1 2 3]"
(resume f) # prints "Inside myblock: [1 2 3]"
(pp [1 2 3]) # prints "[1 2 3]"
```

This is verbose so the core library provides a macro, `with-dyns`, that makes it much clearer in the common case.

```
(pp [1 2 3]) # prints "[1 2 3]"
# prints "Inside with-dyns: [1 2 3]"
(with-dyns [:pretty-format "Inside with-dyns:
%.20P"]
  (pp [1 2 3]))
(pp [1 2 3]) # prints "[1 2 3]"
```

When to use dynamic bindings

Dynamic bindings should be used when you want to pass around an implicit, global context, especially when you want to automatically reset the context if an error is raised. Since a dynamic binding is tied to the current fiber, when a fiber exits the context is automatically unset. This is much easier and often more efficient than manually trying to detect errors and unset context. Consider the following example code, written once with a global var and once with a dynamic binding.

Using a global var

```
(var *my-binding* 10)

(defn may-error
  "A function that may error."
  []
  (if (> (math/random) *my-binding*) (error "uh
oh")))

(defn do-with-value
  "Set *my-binding* to a value and run may-error."
  [x]
  (def oldx *my-binding*)
  (set *my-binding* x)
  (may-error)
  (set *my-binding* oldx))
```

This example is a bit verbose, but most importantly it fails to reset `*my-binding*` if an error is thrown. We could fix this with a `try`, but even

that may have subtle bugs if the fiber yields but is never resumed. However, there is a better solution with dynamic bindings.

Using a dynamic binding

```
(defn may-error
  "A function that may error."
  []
  (if (> (math/random) (dyn :my-binding)) (error
    "uh oh")))

(defn do-with-value
  [x]
  (with-dyns [:my-binding x]
    (may-error)))
```

This looks much cleaner, thanks to a macro, but is also correct in handling errors and any other signal that a fiber may emit. In general, prefer dynamic bindings over global vars. Global vars are mainly useful for scripts or truly program-global configuration.

Advanced use cases

Dynamic bindings work by a table associated with each fiber, called the fiber environment (often "env" for short). This table can be accessed by all functions in the fiber, so it serves as place to store implicit context. During compilation, this table also contains top-level bindings available for use, and is what is returned from a `(require ...)` expression.

With this in mind, there is no requirement that the first argument to `(setdyn name value)` and `(dyn name)` be keywords. These functions can be used to quickly put values in and get values from the current environment. As such, these can be used for getting metadata for a given symbol.

```
(dyn 'pp) # -> prints all metadata in the current environment for pp.  
(setdyn 'pp nil) # -> will not work, as 'pp is defined in the current environments prototype table.  
(setdyn 'pp @{}) # -> will define 'pp as nil.  
(def a 10)  
(setdyn 'a nil) # -> will remove the binding to 'a in the current environment.
```

Macros can modify this table to do things that are otherwise not possible in a macro.

```
(defmacro make-defs  
  "Add many defs at the top level, binding them to random numbers determined at compile time."  
  [x])
```

```
(each name ['a 'b 'c 'd 'e 'f 'g]
  (setdyn name @{:value (math/random)}))
nil)
```

[< Fiber Overview](#)

[Error Handling >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Error Handling

[< Dynamic Bindings](#)

[Object-Oriented Programming >](#)

One of the main uses of fibers is to encapsulate and handle errors. Janet offers no direct try/catch mechanism like some languages, and instead builds error handling on top of fibers. When a function throws an error, Janet creates a signal and throws that signal in the current fiber. The signal will be propagated up the chain of active fibers until it hits a fiber that is ready to handle the error signal. This fiber will trap the signal and return the error from the last call to `resume`.

```
(defn block
  []
  (print "inside block...")
  (error "oops"))

# Pass the :e flag to trap errors (and only
errors).
# All other signals will be propagated up the
fiber stack.
(def f (fiber/new block :e))

# Get result of resuming the fiber in res.
# Because the fiber f traps only errors, we know
# that after resume returns, f either exited
normally
# (has status :dead), or threw an error (has
status :error)
```

```
(def res (resume f))

(if (= (fiber/status f) :error)
  (print "caught error: " res)
  (print "value returned: " res))
```

Janet also provides a simple macro `try` to make this a bit easier in the common case.

```
(try
  (do
    (print "inside block...")
    (error "oops")
    (print "will never get here"))
  ([err] (print "caught error: " err)))
```

[< Dynamic Bindings](#)

[Object-Oriented Programming >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Object-Oriented Programming

[< Error Handling](#)

[Parsing Expression Grammars >](#)

Although Janet is primarily a functional language, it has support for object-oriented programming as well, where objects and classes are represented by tables. Object-oriented programming can be very useful in domains that are less analytical and more about modeling, such as some simulations, GUIs, and game scripting. More specifically, Janet supports a form of object-oriented programming via prototypical inheritance, which was pioneered in the [Self](#) language and is used in languages like Lua and JavaScript. Janet's implementation of object-oriented programming is designed to be simple, terse, flexible, and efficient in that order of priority.

Please see the [prototypes section](#) for more information on table prototypes.

For example:

```
# Create a new object called Car with two
methods, :say and :honk.
(def Car
  @{:type "Car"
    :color "gray"
    :say (fn [self msg] (print "Car says: " msg))
    :honk (fn [self] (print "beep beep! I am "
      (self :color) "!"))})
```

```
# Red Car inherits from Car
(def RedCar
  (table/setproto @{:color "red"} Car))

(:honk Car) # prints "beep beep! I am gray!"
(:honk RedCar) # prints "beep beep! I am red!"

# Pass more arguments
(:say Car "hello!") # prints "Car says: hello!"
```

In the above example, we could replace the method call `(:honk Car)` with `((get Car :honk) Car)`, and this is exactly what the runtime does when it sees a keyword called as a function. In any method call, the object is always passed as the first argument to the method. Since functions in Janet check that their arity is correct, make sure to include a self argument to methods, even when it is not used in the function body.

Factory functions

Rather than constructors, creating objects in Janet can usually be done with a factory function. One can wrap the boilerplate of initializing fields and setting the table prototype using a single function to create a nice interface.

```
(defn make-car
  []
  (table/setproto @{:serial-number (math/random)}
Car))

(def c1 (make-car))
(def c2 (make-car))
(def c3 (make-car))

(c1 :serial-number) # 0.840188
(c2 :serial-number) # 0.394383
(c3 :serial-number) # 0.783099
```

One could also define a set of macros for creating objects with useful factory functions, methods, and properties. The core library does not currently provide such functionality, as it should be fairly simple to customize.

Structs

In addition to tables, Structs may also be used as objects. Structs are of limited use, however, because they do not have prototypes. This means that they cannot implement any form of inheritance. The above example would work for the first object `Car`, but `RedCar` could not inherit from `Car`.

```
# Create a new struct object called Car with two
methods, :say and :honk.
(def Car
  {:type "car"
   :color "gray"
   :say (fn [self msg] (print "Car says: " msg))
   :honk (fn [self] (print "beep beep! I am "
                           (self :color) "!)))}

(:honk Car) # prints "beep beep! I am gray!"

# Pass more arguments
(:say Car "hello!") # prints "Car says: hello!"
```

Abstract types

While tables make good objects, Janet also strives for good interoperation with C. Many C libraries expose pseudo object-oriented interfaces to their libraries, so Janet allows methods to be added to abstract types. The built in abstract type `:core/file` can be manipulated with methods as well as the `file/` functions.

```
(file/read stdin :line)  
# or  
(:read stdin :line)
```

[< Error Handling](#)

[Parsing Expression Grammars >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Parsing Expression Grammars

[< Object-Oriented Programming](#)

[Prototypes >](#)

A common programming task is recognizing patterns in text, be it filtering emails from a list or extracting data from a CSV file. Programming languages and libraries usually offer a number of tools for this, including prebuilt parsers, simple operations on strings (splitting a string on commas), and regular expressions. The pre-built or custom-built parser is usually the most robust solution, but can be very complex to maintain and may not exist for many languages. String functions are not powerful enough for a large class of languages, and regular expressions can be hard to read (which characters are escaped?) and under-powered (don't parse HTML with regex!).

PEGs, or Parsing Expression Grammars, are another formalism for recognizing languages. PEGs are easier to write than a custom parser and more powerful than regular expressions. They also can produce grammars that are easily understandable and fast. PEGs can also be compiled to a bytecode format that can be reused. Janet offers the `peg` module for writing and evaluating PEGs.

Janet's `peg` module borrows syntax and ideas from both LPeg and REBOL/Red parse module. Janet has no built-in regex module because PEGs offer a superset of the functionality of regular expressions.

Below is a simple example for checking if a string is a valid IP address.

Notice how the grammar is descriptive enough that you can read it even if you don't know the PEG syntax (example is translated from a [RED language blog post](#)).

```
(def ip-address
  '{:dig (range "09")
    :0-4 (range "04")
    :0-5 (range "05")
    :byte (choice
            (sequence "25" :0-5)
            (sequence "2" :0-4 :dig)
            (sequence "1" :dig :dig)
            (between 1 2 :dig)))
    :main (sequence :byte "." :byte "." :byte "."
                    :byte}))}

(peg/match ip-address "0.0.0.0") # -> []
(peg/match ip-address "elephant") # -> nil
(peg/match ip-address "256.0.0.0") # -> nil
(peg/match ip-address "0.0.0.0more text") # ->
@[]
```

The API

The `peg` module has few functions because the complexity is exposed through the pattern syntax. Note that there is only one match function, `peg/match`. Variations on matching, such as parsing or searching, can be implemented inside patterns. PEGs can also be compiled ahead of time with `peg/compile` if a PEG will be reused many times.

`(peg/match peg text [,start=0]
& arguments)`

Match a PEG against some text. Returns an array of captured data if the text matches, or `nil` if there is no match. The caller can provide an optional `start` index to begin matching, otherwise the PEG starts on the first character of text. A PEG can either be a compiled PEG object or PEG source.

`(peg/compile peg)`

Compiles a PEG source data structure into a new PEG. Throws an error if there are problems with the PEG code.

Primitive patterns

Larger patterns are built up with primitive patterns, which recognize individual characters, string literals, or a given number of characters. A character in Janet is considered a byte, so PEGs will work on any string of bytes. No special meaning is given to the `0` byte, or the string terminator as in many languages.

Pattern Signature	What it matches
<code>string ("cat")</code>	The literal string.
<code>integer (3)</code>	Matches a number of characters, and advances that many characters. If negative, matches if not that many characters and does not advance. For example, <code>-1</code> will match the end of a string
<code>(range "az" "AZ")</code>	Matches characters in a range and advances 1 character. Multiple ranges can be combined together.
<code>(set "abcd")</code>	Match any character in the argument string. Advances 1 character.

Primitive patterns are not that useful by themselves, but can be passed to `peg/match` and `peg/compile` like any other pattern.

```
(peg/match "hello" "hello") # -> @[]
(peg/match "hello" "hi") # -> nil
```

```
(peg/match 1 "hi") # -> @[]
(peg/match 1 "") # -> nil
(peg/match '(range "AZ") "F") # -> @[]
(peg/match '(range "AZ") "-") # -> nil
(peg/match '(set "AZ") "F") # -> nil
(peg/match '(set "ABCDEFGHIJKLMNOPQRSTUVWXYZ") "F") # -> @[]
```

Combining patterns

These primitive patterns can be combined with several combinators to match a wide number of languages. These combinators can be thought of as the looping and branching forms in a traditional language (that is how they are implemented when compiled to bytecode).

Pattern Signature	What it matches
(choice a b c ...)	Tries to match a, then b, and so on. Will succeed on the first successful match, and fails if none of the arguments match the text.
(+ a b c ...)	Alias for (choice a b c ...)
(sequence a b c)	Tries to match a, b, c and so on in sequence. If any of these arguments fail to match the text, the whole pattern fails.
(* a b c ...)	Alias for (sequence a b c ...)
(any x)	Matches 0 or more repetitions of x.
(some x)	Matches 1 or more repetitions of x.
(between min max x)	Matches between min and max (inclusive) repetitions of x.

(at-least n x)	Matches at least n repetitions of x.
(at-most n x)	Matches at most n repetitions of x.
(if cond patt)	Tries to match patt only if cond matches as well. cond will not produce any captures.
(if-not cond patt)	Tries to match only if cond does not match. cond will not produce any captures.
(not patt)	Matches only if patt does not match. Will not produce captures or advance any characters.
(! patt)	Alias for (not patt)
(look offset patt)	Matches only if patt matches at a fixed offset. offset can be any integer. patt will not produce captures and the peg will not advance any characters.
(> offset patt)	Alias for (look offset patt)
(opt patt)	Alias for (between 0 1 patt)
(? patt)	Alias for (between 0 1 patt)

PEGs try to match an input text with a pattern in a greedy manner. This means that if a rule fails to match, that rule will fail and not try again. The only backtracking provided in a PEG is provided by the (choice x y

`z ...)` special, which will try rules in order until one succeeds, and the whole pattern succeeds. If no sub-pattern succeeds, then the whole pattern fails. Note that this means that the order of `x y z` in choice does matter. If `y` matches everything that `z` matches, `z` will never succeed.

Captures

So far we have only been concerned with "does this text match this language?". This is useful, but it is often more useful to extract data from text if it does match a PEG. The `peg` module uses that concept of a capture stack to extract data from text. As the PEG is trying to match a piece of text, some forms may push Janet values onto the capture stack as a side effect. If the text matches the main PEG language, `(peg/match)` will return the final capture stack as an array.

Capture specials will only push captures to the capture stack if their child pattern matches the text. Most captures specials will match the same text as their first argument pattern. In addition, most specials that produce captures can take an optional argument `tag` that applies a keyword tag to the capture. These tagged captures can then be recaptured via the `(backref tag)` special in subsequent matches. Tagged captures, when combined with the `(cmt)` special, provide a powerful form of look-behind that can make many grammars simpler.

Pattern Signature	What it matches
<code>(capture patt ?tag)</code>	Captures all of the text in patt if patt matches, If patt contains any captures, then those captures will be pushed to the capture stack before the total text.
<code>(<- patt ? tag)</code>	Alias for <code>(capture patt ?tag)</code>

(quote patt ?tag)	Another alias for (capture patt ?tag). This allows code like 'patt to capture a pattern.
(group patt ?tag)	Captures an array of all of the captures in patt.
(replace patt subst ?tag)	Replaces the captures produced by patt by applying subst to them. If subst is a table or struct, will push (get subst last-capture) to the capture stack after removing the old captures. If a subst is a function, will call subst with the captures of patt as arguments and push the result to the capture stack. Otherwise, will push subst literally to the capture stack.
(/ patt subst ?tag)	Alias for (replace patt subst ?tag)
(constant k ?tag)	Captures a constant value and advances no characters.
(argument n ?tag)	Captures the nth extra argument to the match function and does not advance.
(position ? tag)	Captures the current index into the text and advances no input.
(\$?tag)	Alias for (position ?tag).
(accumulate patt ?tag)	Capture a string that is the concatenation of all captures in patt. This will try to be efficient and not create intermediate strings if possible.

(% patt ? tag)	Alias for (accumulate patt ?tag)
(cmt patt fun ?tag)	Invokes fun with all of the captures of patt as arguments (if patt matches). If the result is truthy, then captures the result. The whole expression fails if fun returns false or nil.
(backref tag ?tag)	Duplicates the last capture with the tag tag . If no such capture exists then the match fails.
(-> tag ? tag)	Alias for (backref tag) .
(error patt)	Throws a Janet error if patt matches. The error thrown will be the last capture of patt, or a generic error if patt produces no captures.
(drop patt)	Ignores (drops) all captures from patt.
(lenprefix n patt)	Matches n repetitions of a pattern, where n is supplied from other parsed input and is not a constant.

Grammars and recursion

The feature that makes PEGs so much more powerful than pattern matching solutions like (vanilla) regex is mutual recursion. To do recursion in a PEG, you can wrap multiple patterns in a grammar, which is a Janet struct. The patterns must be named by keywords, which can then be used in all sub-patterns in the grammar.

Each grammar, defined by a struct, must also have a main rule, called `:main`, that is the pattern that the entire grammar is defined by.

An example grammar that uses mutual recursion:

```
(def my-grammar
'{:a (* "a" :b "a")
:b (* "b" (+ :a 0) "b")
:main (* "(" :b ")")})

(peg/match my-grammar "(bb)") # -> []
(peg/match my-grammar "(babbab)") # -> []
(peg/match my-grammar "(baab)") # -> nil
(peg/match my-grammar "(babaabab)") # -> nil
```

Keep in mind that recursion is implemented with a stack, meaning that very recursive grammars can overflow the stack. The compiler is able to turn some recursion into iteration via tail-call optimization, but some patterns may fail on large inputs. It is also possible to construct (very poorly written) patterns that will result in long loops and be very slow in general.

Built-in patterns

Besides the primitive patterns and pattern combinator given above, the `peg` module also provides a default grammar with a handful of commonly used patterns. All of these shorthands can be defined with the combinators above and primitive patterns, but you may see these aliases in other grammars and they can make grammar simpler and easier to read.

Name	Expanded	Description
<code>:d</code>	<code>(range "09")</code>	Matches an ASCII digit.
<code>:a</code>	<code>(range "az" "AZ")</code>	Matches an ASCII letter.
<code>:w</code>	<code>(range "az" "AZ" "09")</code>	Matches an ASCII digit or letter.
<code>:s</code>	<code>(set " \t\r\n\f")</code>	Matches an ASCII whitespace character.
<code>:D</code>	<code>(if-not :d 1)</code>	Matches a character that is not an ASCII digit.
<code>:A</code>	<code>(if-not :a 1)</code>	Matches a character that is not an ASCII letter.
<code>:W</code>	<code>(if-not :w 1)</code>	Matches a character that is not an ASCII digit or letter.
<code>:S</code>	<code>(if-not :s 1)</code>	Matches a character that is not ASCII whitespace.

1)		ASCII whitespace.
:d+	(some :d)	Matches 1 or more ASCII digits.
:a+	(some :a)	Matches 1 or more ASCII letters.
:w+	(some :w)	Matches 1 or more ASCII digits and letters.
:s+	(some :s)	Matches 1 or more ASCII whitespace characters.
:d*	(any :d)	Matches 0 or more ASCII digits.
:a*	(any :a)	Matches 0 or more ASCII letters.
:w*	(any :w)	Matches 0 or more ASCII digits and letters.
:s*	(any :s)	Matches 0 or more ASCII whitespace characters.

All of these aliases are defined in `default-peg-grammar`, which is a table that maps from the alias name to the expanded form. You can even add your own aliases here which are then available for all PEGs in the program. Modifying this table will not affect already compiled PEGs.

String searching and other idioms

Although all pattern matching is done in anchored mode, operations like global substitution and searching can be implemented with the PEG module. A simple Janet function that produces PEGs that search for strings shows how captures and looping specials can be composed, and how quasiquoting can be used to embed values in patterns.

```
(defn finder
  "Creates a peg that finds all locations of str
  in the text."
  [str]
  (peg/compile ~(any (+ (* ($), str) 1)))))

(def where-are-the-dogs? (finder "dog"))

(peg/match where-are-the-dogs? "dog dog cat dog")
# -> @[0 4 12]

# Our finder function also works any pattern, not
just strings.

(def find-cats (finder '(* "c" (some "a") "t")))

(peg/match find-cats "cat ct caat caaaaat cat") #
-> @[0 7 12 20]
```

We can also wrap a PEG to turn it into a global substitution grammar with the accumulate special (%) .

```
(defn replacer
  "Creates a peg that replaces instances of patt
```

```
with subst."
[patt subst]
(peg/compile ~(% (any (+ (/ (<- ,patt) ,subst)
(<- 1))))))
```

[< Object-Oriented Programming](#)

[Prototypes >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Prototypes

[< Parsing Expression Grammars](#)

[The Janet Abstract Machine >](#)

To support basic generic programming, Janet tables support a prototype table. A prototype table contains default values for a table if certain keys are not found in the original table. This allows many similar tables to share contents without duplicating memory.

```
# Simple Object Oriented behavior in Janet
(def proto1 @{:type :custom1
              :behave (fn [self x] (print
"behaving " x)))})
(def proto2 @{:type :custom2
              :behave (fn [self x] (print
"behaving 2 " x)))}

(def thing1 (table/setproto @{} proto1))
(def thing2 (table/setproto @{} proto2))

(print (thing1 :type)) # prints :custom1
(print (thing2 :type)) # prints :custom2

(:behave thing1 :a) # prints "behaving :a"
(:behave thing2 :b) # prints "behaving 2 :b"
```

Looking up a value in a table with a prototype can be summed up with the following algorithm.

1. `(get my-table my-key)` is called.
2. `my-table` is checked for the key `my-key`. If there is a value for the key, it is returned.
3. If there is a prototype table for `my-table`, set `my-table = my-table's prototype` and go to 2. Otherwise go to 4.
4. Return `nil` as the key was not found.

Janet will check up to about a 1000 prototypes recursively by default before giving up and returning `nil`. This is to prevent an infinite loop. This value can be changed by adjusting the `JANET_RECURSION_GUARD` value in `janet.h`.

Note that Janet prototypes are not as expressive as metatables in Lua and many other languages. This is by design, as adding Lua- or Python-like capabilities would not be technically difficult. Users should prefer plain data and functions that operate on them rather than mutable objects with methods. However, some object-oriented capabilities are useful in Janet for systems that require extra flexibility.

[< Parsing Expression Grammars](#)

[The Janet Abstract Machine >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

The Janet Abstract Machine

[< Prototypes](#)

[Multithreading >](#)

The Janet language is implemented on top of an abstract machine (AM). The compiler converts Janet data structures to this bytecode, which can then be efficiently executed from inside a C program. To understand Janet bytecode, it is useful to understand the abstractions used inside the Janet AM, as well as the C types used to implement these features.

The stack = the fiber

A Janet fiber is the type used to represent multiple concurrent processes in Janet. It is basically a wrapper around the idea of a stack. The stack is divided into a number of stack frames (`JanetStackFrame *` in C), each of which contains information such as the function that created the stack frame, the program counter for the stack frame, a pointer to the previous frame, and the size of the frame. Each stack frame also is paired with a number registers.

```
X: Slot  
  
X  
X - Stack Top, for next function call.  
----  
Frame next  
----  
X  
X  
X  
X  
X  
X  
X  
X - Stack 0  
----  
Frame 0  
----  
X  
X  
X - Stack -1  
----  
Frame -1  
----
```

```
X  
X  
X  
X  
X - Stack -2  
-----  
Frame -2  
-----  
...  
...  
...  
-----  
Bottom of stack
```

Fibers also have an incomplete stack frame for the next function call on top of their stacks. Making a function call involves pushing arguments to this temporary stack, and then invoking either the `CALL` or `TCALL` instructions. Arguments for the next function call are pushed via the `PUSH`, `PUSH2`, `PUSH3`, and `PUSHA` instructions. The stack of a fiber will grow as large as needed, although by default Janet will limit the maximum size of a fiber's stack. The maximum stack size can be modified on a per-fiber basis.

The slots in the stack are exposed as virtual registers to instructions. They can hold any Janet value.

Closures

All functions in Janet are closures; they combine some bytecode instructions with 0 or more environments. In the C source, a closure (hereby the same as a function) is represented by the type

`JanetFunction *`. The bytecode instruction part of the function is represented by `JanetFuncDef *`, and a function environment is represented with `JanetFuncEnv *`.

The function definition part of a function (the 'bytecode' part, `JanetFuncDef *`), also stores various metadata about the function which is useful for debugging, as well as constants referenced by the function.

C functions

Janet uses C functions to bridge to native code. A C function (`JanetCFunction *` in C) is a C function pointer that can be called like a normal Janet closure. From the perspective of the bytecode instruction set, there is no difference in invoking a C function and invoking a normal Janet function.

Bytecode format

Janet bytecode operates on an array of identical registers that can hold any Janet value (`Janet *` in C). Most instructions have a destination register, and 1 or 2 source registers. Registers are simply indices into the stack frame, which can be thought of as a constant sized array.

Each instruction is a 32-bit integer, meaning that the instruction set is a constant width RISC instruction set like MIPS. The opcode of each instruction is the least significant byte of the instruction. The highest bit of this leading byte is reserved for debugging purpose, so there are 128 possible opcodes encodable with this scheme. Not all of these possible opcodes are defined, and undefined opcodes will trap the interpreter and emit a debug signal. Note that this means an unknown opcode is still valid bytecode, it will just put the interpreter into a debug state when executed.

X - Payload bits
0 - Opcode bits

4	3	2	1
+---+	---	---	---
XX	XX	XX	00
+---+	---	---	---

Using 8 bits for the opcode leaves 24 bits for the payload, which may or may not be utilized. There are a few instruction variants that divide these payload bits.

- 0 arg - Used for noops, returning `nil`, or other instructions that take no arguments. The payload is essentially ignored.

- 1 arg - All payload bits correspond to a single value, usually a signed or unsigned integer. Used for instructions of 1 argument, like returning a value, yielding a value to the parent fiber, or doing a (relative) jump.
- 2 arg - Payload is split into byte 2 and bytes 3 and 4. The first argument is the 8-bit value from byte 2, and the second argument is the 16-bit value from bytes 3 and 4 (`instruction >> 16`). Used for instructions of two arguments, like move, normal function calls, conditionals, etc.
- 3 arg - Bytes 2, 3, and 4 each correspond to an 8-bit argument. Used for arithmetic operations, emitting a signal, etc.

These instruction variants can be further refined based on the semantics of the arguments. Some instructions may treat an argument as a slot index, while other instructions will treat the argument as a signed integer literal, an index for a constant, an index for an environment, or an unsigned integer. Keeping the bytecode fairly uniform makes verification, compilation, and debugging simpler.

Instruction reference

A listing of all opcode values can be found in `janet.h`. The Janet assembly short names can be found in `src/core/asm.c`. In this document, we will refer to the instructions by their short names as presented to the assembler rather than their numerical values.

Each instruction is also listed with a signature, which are the arguments the instruction expects. There are a handful of instruction signatures, which combine the arity and type of the instruction. The assembler does not do any type-checking per closure, but does prevent jumping to invalid instructions and failure to return or error.

Notation

- The '\$' prefix indicates that a instruction parameter is acting as a virtual register (slot). If a parameter does not have the '\$' suffix in the description, it is acting as some kind of literal (usually an unsigned integer for indexes, and a signed integer for literal integers).
- Some operators in the description have the suffix 'i' or 'r'. These indicate that these operators correspond to integers or real numbers only, respectively. All bit-wise operators and bit shifts only work with integers.
- The `>>>` indicates unsigned right shift, as in Java. Because all integers in Janet are signed, we differentiate the two kinds of right bit shift.
- The 'im' suffix in the instruction name is short for immediate.

Reference table

Instruction	Signature	Description
add	(add dest lhs rhs)	\$dest = \$lhs + \$rhs
addim	(addim dest lhs im)	\$dest = \$lhs + im
band	(band dest lhs rhs)	\$dest = \$lhs & \$rhs
bnot	(bnot dest operand)	\$dest = ~\$operand
bor	(bor dest lhs rhs)	\$dest = \$lhs \$rhs
bxor	(bxor dest lhs rhs)	\$dest = \$lhs ^ \$rhs
call	(call dest callee)	\$dest = call(\$callee, args)
clo	(clo dest index)	\$dest = closure(defs[\$index])
cmp	(cmp dest lhs rhs)	\$dest = janet_compare(\$lhs, \$rhs)
div	(div dest lhs rhs)	\$dest = \$lhs / \$rhs

<code>divim</code>	<code>(divim dest lhs im)</code>	$\$dest = \lhs / im
<code>eq</code>	<code>(eq dest lhs rhs)</code>	$\$dest = \$lhs == \$rhs$
<code>eqim</code>	<code>(eqim dest lhs im)</code>	$\$dest = \$lhs == im$
<code>eqn</code>	<code>(eqn dest lhs im)</code>	$\$dest = \$lhs .== \$rhs$
<code>err</code>	<code>(err message)</code>	Throw error \$message.
<code>get</code>	<code>(get dest ds key)</code>	$\$dest = \$ds[\$key]$
<code>geti</code>	<code>(geti dest ds index)</code>	$\$dest = \$ds[index]$
<code>gt</code>	<code>(gt dest lhs rhs)</code>	$\$dest = \$lhs > \$rhs$
<code>gten</code>	<code>(gten dest lhs rhs)</code>	$\$dest = \$lhs .>= \$rhs$
<code>gtim</code>	<code>(gtim dest lhs im)</code>	$\$dest = \$lhs .> im$
<code>gtn</code>	<code>(gtn dest lhs rhs)</code>	$\$dest = \$lhs .> \$rhs$
<code>jmp</code>	<code>(jmp label)</code>	$pc = \text{label}$, $pc += \text{offset}$
		if \$cond $pc = \text{label}$ else

<code>jmpif</code>	<code>(jmpif cond label)</code>	<code>pc++</code>
<code>jmpno</code>	<code>(jmpno cond label)</code>	<code>if \$cond pc++ else pc = label</code>
<code>ldc</code>	<code>(ldc dest index)</code>	<code>\$dest = constants[index]</code>
<code>ldf</code>	<code>(ldf dest)</code>	<code>\$dest = false</code>
<code>ldi</code>	<code>(ldi dest integer)</code>	<code>\$dest = integer</code>
<code>ldn</code>	<code>(ldn dest)</code>	<code>\$dest = nil</code>
<code>lds</code>	<code>(lds dest)</code>	<code>\$dest = current closure (self)</code>
<code>ldt</code>	<code>(ldt dest)</code>	<code>\$dest = true</code>
<code>ldu</code>	<code>(ldu dest env index)</code>	<code>\$dest = envs[env][index]</code>
<code>len</code>	<code>(len dest ds)</code>	<code>\$dest = length(ds)</code>
<code>lt</code>	<code>(lt dest lhs rhs)</code>	<code>\$dest = \$lhs < \$rhs</code>
<code>lten</code>	<code>(lten dest lhs rhs)</code>	<code>\$dest = \$lhs .<= \$rhs</code>
<code>ltim</code>	<code>(ltim dest lhs im)</code>	<code>\$dest = \$lhs .< im</code>

<code>ltn</code>	<code>(ltn dest lhs rhs)</code>	$\$dest = \$lhs < \$rhs$
<code>mkarr</code>	<code>(mkarr dest)</code>	$\$dest = \text{call(array, args)}$
<code>mkbtp</code>	<code>(mkbtp dest)</code>	$\$dest = \text{call(tuple/brackets, args)}$
<code>mkbuf</code>	<code>(mkbuf dest)</code>	$\$dest = \text{call(buffer, args)}$
<code>mkstr</code>	<code>(mkstr dest)</code>	$\$dest = \text{call(string, args)}$
<code>mkstu</code>	<code>(mkstu dest)</code>	$\$dest = \text{call(struct, args)}$
<code>mktab</code>	<code>(mktab dest)</code>	$\$dest = \text{call(table, args)}$
<code>mktup</code>	<code>(mktup dest)</code>	$\$dest = \text{call(tuple, args)}$
<code>movf</code>	<code>(movf src dest)</code>	$\$dest = \src
<code>movn</code>	<code>(movn dest src)</code>	$\$dest = \src
<code>mul</code>	<code>(mul dest lhs rhs)</code>	$\$dest = \$lhs * \$rhs$
<code>mulim</code>	<code>(mulim dest lhs im)</code>	$\$dest = \$lhs * im$
<code>noop</code>	<code>(noop)</code>	Does nothing.
<code>prop</code>	<code>(prop val fiber)</code>	Propagate (Re-raise) a signal that has been caught.
		Push \$val on arg

<code>push</code>	<code>(push val)</code>	
<code>push2</code>	<code>(push2 val1 val3)</code>	Push \$val1, \$val2 on args
<code>push3</code>	<code>(push3 val1 val2 val3)</code>	Push \$val1, \$val2, \$val3, on args
<code>pusha</code>	<code>(pusha array)</code>	Push values in \$array on args
<code>put</code>	<code>(put ds key val)</code>	\$ds[\$key] = \$val
<code>puti</code>	<code>(puti ds index val)</code>	\$ds[index] = \$val
<code>res</code>	<code>(res dest fiber val)</code>	\$dest = resume \$fiber with \$val
<code>ret</code>	<code>(ret val)</code>	Return \$val
<code>retn</code>	<code>(retn)</code>	Return nil
<code>setu</code>	<code>(setu env index val)</code>	envs[env][index] = \$val
<code>sig</code>	<code>(sig dest value sigtype)</code>	\$dest = emit \$value as sigtype
<code>sl</code>	<code>(sl dest lhs rhs)</code>	\$dest = \$lhs << \$rhs
<code>slim</code>	<code>(slim dest</code>	\$dest = \$lhs << shamt

	lhs shamt)	
sr	(sr dest lhs rhs)	\$dest = \$lhs >> \$rhs
srim	(srim dest lhs shamt)	\$dest = \$lhs >> shamt
sru	(sru dest lhs rhs)	\$dest = \$lhs >>> \$rhs
sruim	(sruim dest lhs shamt)	\$dest = \$lhs >>> shamt
sub	(sub dest lhs rhs)	\$dest = \$lhs - \$rhs
tcall	(tcall callee)	Return call(\$callee, args)
tchck	(tcheck slot types)	Assert \$slot matches types

[< Prototypes](#)

[Multithreading >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Multithreading

[< The Janet Abstract Machine](#)

[jpm >](#)

Starting in version 1.6.0, Janet has the ability to do true, non-cooperative multithreading with the `thread/` functions. Janet threads correspond to native threads on the host operating system, which may be either pthreads on POSIX systems or Windows threads. Each thread has its own Janet heap, which means threads behave more like processes that communicate by message passing. However, this does not prevent native code from sharing memory across these threads. Without native extensions, however, the only way for two Janet threads to communicate directly is through message passing.

Creating threads

Use `(thread/new func &opt capacity)` to create a new thread. This thread will start and wait for a message containing a function that it will run as the main body. This function must also be able to take 1 parameter, the parent thread. Each thread has its own mailbox, which can asynchronously receive messages. These messages are added to a queue, which has a configurable maximum capacity according to the second optional argument passed to `thread/new`. If unspecified, the mailbox will have an initial capacity of 10.

```
(defn worker
  [parent]
  (print "New thread started!"))

# New thread's mailbox has capacity for 32
messages.
(def thread (thread/new worker 32))
```

Sending and receiving messages

Threads in Janet do not share a memory heap and must communicate via message passing. Use `thread/send` to send a message to a thread, and `thread/receive` to get messages sent to the current thread.

```
(defn worker
  [parent]
  (print "waiting for message..."))
  (def msg (thread/receive))
  (print "got message: " msg))

(def thread (thread/new worker))
# Thread objects support the :send method as an
alias for thread/send.
(:send thread "Hello!")
```

Limitations of messages

Since threads do not share Janet heaps, all values sent as messages are first marshalled to a byte sequence by the sending thread, and then unmarshalled by the receiving thread. For marshalling and unmarshalling, threads use the two tables `make-image-dict` and `load-image-dict`. This means you can send over core bindings, even if the underlying value cannot be marshalled. Semantically, messages sent with `(thread/send to msg)` are first converted to a buffer via `(marshal msg make-image-dict mailbox-buf)`, and then unmarshalled by the receiving thread via `(unmarshal mailbox-buf load-image-dict)`.

Values that cannot be marshalled, including thread values, cannot be sent as messages to other threads, or even as part of a message to another thread. For example, the following will not work because open file handles cannot be marshalled.

```
(def file (file/open "myfile.txt" :w))
(defn worker
  [parent]
  (with-dyns [:out file]
    (print "Writing to file."))

# Will throw an error, as worker contains a
# reference to file, which cannot be marshalled.
(thread/new worker)
```

The fix here is to move the file creation inside the worker function, since every worker thread is going to have its own copy of the file handle.

```
(defn worker
  [parent]
  (def file (file/open "myfile.txt" :w))
  (with-dyns [:out file]
    (print "Writing to file.")))

# No error
(thread/new worker)
```

This limitation has been removed for some values (C functions and raw pointers) in version 1.9.0, to make use of threads with native modules more seamless, although certain abstract types like open files can still not be sent to threads via `thread/send` or `thread/new`.

Blocking and non-blocking sends and receives

All sends and receives can be blocking or non-blocking by providing an optional timeout value. A timeout of `0` makes `(thread/send to msg &opt timeout)` and `(thread/receive &opt timeout)` non-blocking, while a positive timeout value indicates a blocking send or receive that will resume the current thread by throwing an error after timeout seconds. Timeouts are limited to 30 days, with exception the using `math/inf` for a timeout means that a timeout will never occur.

```
(defn worker
  [parent]
  (for i 0 10
    (os/sleep 0.5)
    (:send parent :ping))
  # sleep a long while to make the parent timeout
  (os/sleep 2)
  (:send parent :done))

# Will print the first 10 pings, but timeout and
# throw an error before :done
(try
  (let [t (thread/new worker)]
    (for i 0 11
      (print "got message: " (thread/receive
        1)))
    ([err] (print "error: " err)))

  # Flush :done message
  (thread/receive math/inf))
```

```
# Will not error and work successfully
(let [t (thread/new worker)]
  (for i 0 11
    (print "got message: " (thread/receive
  math/inf))))
```

[< The Janet Abstract Machine](#)

[jpm >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

jpm

[< Multithreading](#)

[Core API >](#)

Most default installs of Janet come with a build tool, `jpm`, that makes managing dependencies, building Janet code, and distributing Janet applications easy. While `jpm` is not required by Janet, it is a very handy build tool that removes much of the boilerplate for creating new Janet projects, and is updated in step with Janet. `jpm` also works on Windows, Linux, and macOS so you don't need to write multiple build scripts and worry about platform quirks to make your scripts, binaries, and applications cross-platform.

`jpm`'s main functions are installing dependencies and building native Janet modules, but it is meant to be used for much of the life-cycle for Janet projects. Since Janet code doesn't usually need to be compiled, you don't always need `jpm`, especially for scripts, but `jpm` comes with some functionality that is difficult to duplicate, like compiling Janet source code and all imported modules into a statically linked executable for distribution.

Glossary

A self contained unit of Janet source code as recognized by `jpm` is called a project. A project is a directory containing a `project.janet` file, which contains build recipes. Often, a project will correspond to a single git repository, and contain a single library. However, a project's `project.janet` file can contain build recipes for as many libraries, native extensions, and executables as it wants. Each of these recipes builds an artifact. Artifacts are the output files that will either be distributed or installed on the end-user or developer's machine.

Building projects with `jpm`

Once you have the project on your machine, building the various artifacts should be pretty simple.

Global install

```
sudo jpm deps  
jpm build  
jpm test  
sudo jpm install
```

(On Windows, `sudo` is not required. Use of `sudo` on POSIX systems depends on whether you installed `janet` to a directory owned by the root user.)

Local install

```
export JANET_PATH=/home/me/janet  
jpm deps  
jpm build  
jpm test  
jpm install
```

Dependencies

`jpm deps` is a command that installs Janet libraries that the project depends on recursively. It will automatically fetch, build, and install all required dependencies for you. As of August 2019, this only works with

git, which you need to have installed on your machine to install dependencies. If you don't have git you are free to manually obtain the requisite dependencies and install them manually with `sudo jpm install`.

Building

Next, we use the `jpm build` command to build artifacts. All built artifacts will be created in the `build` subdirectory of the current project. Therefore, it is probably a good idea to exclude the `build` directory from source control. For building executables and native modules, you will need to have a C compiler on your PATH where you run `jpm build`. For POSIX systems, the compiler is `cc`.

If you make changes to the source code after building once, `jpm` will try to only rebuild what is needed on a rebuild. If this fails for any reason, you can delete the entire build directory with `jpm clean` to reset things.

Windows

For Windows, the C compiler used by `jpm` is `cl.exe`, which is part of MSVC. You can get it with Visual Studio, or standalone with the C and C++ Build Tools from Microsoft. You will then need to run `jpm build` in a Developer Command Prompt, or source `vcvars64.bat` in your shell to add `cl.exe` to the PATH.

Testing

Once we have built our software, it is a good idea to test it to verify that it works on the current machine. `jpm test` will run all Janet scripts in the

`test` directory of the project and return a non-zero exit code if any fail.

Installing

Finally, once we have built our software and tested that it works, we can install it on our system. For an executable, this means copying it to the `bin` directory, and for libraries it means copying them to the global syspath. You can optionally install into any directory if you don't want to pollute your system or you don't have permission to write to the directory where `janet` itself was installed. You can specify the path to install modules to via the `--modpath` option, and the path to install binaries to with the `--binpath` option. These need to be given before the subcommand `install`.

The `project.janet` file

To create your own software in Janet, it is a good idea to understand what the `project.janet` file is and how it defines rules for building, testing, and installing software. The code in `project.janet` is normal Janet source code that is run in a special environment.

A `project.janet` file is loaded by `jpm` and evaluated to create various recipes, or rules. For example, `declare-project` creates several rules, including `"install"`, `"build"`, `"clean"`, and `"test"`. These are a few of the rules that `jpm` expects `project.janet` to create when executed.

Declaring a project

Use the `declare-project` as the first `declare-` macro towards the beginning of your `project.janet` file. You can also pass in any metadata about your project that you want, and add dependencies on other Janet projects here.

```
(declare-project
  :name "mylib" # required
  :description "a library that does things" # some example metadata.

  # Optional urls to git repositories that contain required artifacts.
  :dependencies ["https://github.com/janet-lang/json.git"]))
```

Creating a module

A 100% Janet library is the easiest kind of software to distribute in Janet. Since it does not need to be built and since installing it means simply moving the files to a system directory, we only need to specify the files that comprise the library in `project.janet`.

```
(declare-source
  # :source is an array or tuple that can contain
  # source files and directories that will be
  installed.
  # Often will just be a single file or single
  directory.
  :source ["mylib.janet"])
```

For information on writing modules, see [the modules docpage](#).

Creating a native module

Once you have written your C code that defines your native module (see the [embedding](#) page on how to do this), you must declare it in `project.janet` in order for `jpm` to build the native modules for you.

```
(declare-native
  :name "mynative"
  :source ["mynative.c" "mysupport.c"]
  :embedded ["extra-functions.janet"])
```

This makes `jpm` create a native module called `mynative` when `jpm build` is run, the arguments for which should be pretty straightforward. The `:embedded` argument is Janet source code that will be embedded

as an array of bytes directly into the C source code. It is not recommended to use the `:embedded` argument, as one can simply create multiple artifacts, one for a pure C native module and one for Janet source code.

Creating an executable

The declaration for an executable file is pretty simple.

```
(declare-executable
  :name "myexec"
  :entry "main.janet")
```

`jpm` is smart enough to figure out from the one entry file what libraries and other code your executable depends on, and bundles them into the final application for you. The final executable will be located at `build/myexec`, or `build\myexec.exe` on Windows.

Also note that the entry of an executable file should look different than a normal Janet script. It should define a `main` function that can receive a variable number of parameters, the command-line arguments. It will be called as the entry point to your executable.

```
(import mylib1)
(import mylib2)

# This will be printed when you run `jpm build`
(print "build time!")

(defn main [& args]
  # You can also get command-line arguments
  through (dyn :args)
```

```
(print "args: " ;(interpolate ", " args))
(mylib1/do-thing)
(mylib2/do-thing))
```

It's important to remember that code at the top level will run when you invoke `jpm build`, not at executable runtime. This is because in order to create the executable, we marshal the `main` function of the app and write it to an image. In order to create the main function, we need to actually compile and run everything that it references, in the above case `mylib1` and `mylib2`.

This has a number of benefits, but the largest is that we only include bytecode for the functions that our application uses. If we only use one function from a library of 1000 functions, our final executable will not include the bytecode for the other 999 functions (unless our one function references some of those other functions, of course). This feature, called tree-shaking, only works for Janet code. Native modules will be linked to the final executable statically in full if they are used at all. A native module is considered "used" if is imported at any time during `jpm build`. This may change, but it is currently the most reliable way to check if a native modules needs to be linked into the final executable.

There are some limitations to this approach. Any dynamically required modules will not always be linked into the final executable. If `require` or `import` is not called during `jpm build`, then the code will not be linked into the executable. The module can still be required if it is available at runtime, though.

For an example Janet executable built with `jpm`, see <https://github.com/bakpakin/littleserver>.

[< Multithreading](#)

[Core API >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#))

Core API

[< jpm](#) [Array Module >](#)

Index

% %≡ * *≡ + ++ +≡ - -≡ -> ->> -?> -?>> / /≡ ≤ ≤≡ ≡ ≥ ≥≡ abstract?
accumulate accumulate2 all all-bindings all-dynamics and apply array
array(concat array/ensure array/fill array/insert array/new array/new-filled
array/peek array/pop array/push array/remove array/slice array? as-> as?
-> asm assert bad-compile bad-parse band blshift bnot boolean? bor
brshift brushift buffer buffer/bit buffer/bit-clear buffer/bit-set buffer/bit-
toggle buffer/blit buffer/clear buffer/fill buffer/format buffer/new
buffer/new-filled buffer/popn buffer/push-byte buffer/push-string
buffer/push-word buffer/slice buffer? bxor bytes? case cfunction? chr cli-
main comment comp compile complement comptime cond coro count
debug debug/arg-stack debug/break debug/fbreak debug/lineage
debug/stack debug/stacktrace debug/step debug/unbreak
debug/unbreak debugger-env dec deep-not= deep= def- default default-
peg-grammar defer defglobal defmacro defmacro- defn defn- describe
dictionary? disasm distinct doc doc* doc-format dofile drop drop-until
drop-while dyn each eachk eachp edefer eflush empty? env-lookup eprin
eprintf eprint eprintf error errorf eval eval-string even? every? extreme
false? fiber/can-resume? fiber/current fiber/getenv fiber/maxstack
fiber/new fiber/root fiber/setenv fiber/setmaxstack fiber/status fiber?
file/close file/flush file/open file/popen file/read file/seek file/temp file/write
filter find find-index first flatten flatten-into flush for freeze frequencies
function? gccollect ginterval gcsetinterval generate gensym get get-in
getline hash idempotent? identity if-let if-not if-with import import* in inc
indexed? int/s64 int/u64 int? interleave interpose invert janet/build
janet/config-bits janet/version juxtapose juxtapose* keep keys keyword keyword? kvs
label last length let load-image load-image-dict loop macex macex1
make-env make-image make-image-dict map mapcat marshal match
math/-inf math/abs math/acos math/acosh math/asin math/asinh

[math/atan](#) [math/atan2](#) [math/atanh](#) [math/cbrt](#) [math/ceil](#) [math/cos](#)
[math/cosh](#) [math/e](#) [math/erf](#) [math/erfc](#) [math/exp](#) [math/exp2](#) [math/expm1](#)
[math/floor](#) [math/gamma](#) [math/hypot](#) [math/inf](#) [math/log](#) [math/log10](#)
[math/log1p](#) [math/log2](#) [math/next](#) [math/pi](#) [math/pow](#) [math/random](#)
[math/rng](#) [math/rng-buffer](#) [math/rng-int](#) [math/rng-uniform](#) [math/round](#)
[math/seedrandom](#) [math/sin](#) [math/sinh](#) [math/sqrt](#) [math/tan](#) [math/tanh](#)
[math/trunc](#) [max](#) [mean](#) [merge](#) [merge-into](#) [min](#) [mod](#) [module/add-paths](#)
[module/cache](#) [module/expand-path](#) [module/find](#) [module/loaders](#)
[module/loading](#) [module/paths](#) [nan?](#) [nat?](#) [native](#) [neg?](#) [net/chunk](#) [net/close](#)
[net/connect](#) [net/read](#) [net/server](#) [net/write](#) [next](#) [nil?](#) [not](#) [not=](#) [number?](#)
[odd?](#) [one?](#) [or](#) [os/arch](#) [os/cd](#) [os/chmod](#) [os/clock](#) [os/cryptorand](#) [os/cwd](#)
[os/date](#) [os/dir](#) [os/environ](#) [os/execute](#) [os/exit](#) [os/getenv](#) [os/link](#) [os/lstat](#)
[os/mkdir](#) [os/mktime](#) [os/perm-int](#) [os/perm-string](#) [os/readlink](#) [os.realpath](#)
[os/rename](#) [os/rm](#) [os/rmdir](#) [os/setenv](#) [os/shell](#) [os/sleep](#) [os/stat](#) [os/symlink](#)
[os/time](#) [os/touch](#) [os/umask](#) [os/which](#) [pairs](#) [parse](#) [parser/byte](#) [parser/clone](#)
[parser/consume](#) [parser/eof](#) [parser/error](#) [parser/flush](#) [parser/has-more](#)
[parser/insert](#) [parser/new](#) [parser/produce](#) [parser/state](#) [parser/status](#)
[parser/where](#) [partial](#) [partition](#) [peg/compile](#) [peg/match](#) [pos?](#) [postwalk](#) [pp](#)
[prewalk](#) [prin](#) [printf](#) [print](#) [printf](#) [product](#) [prompt](#) [propagate](#) [protect](#) [put](#) [put-in](#)
[quit](#) [range](#) [reduce](#) [reduce2](#) [repl](#) [require](#) [resume](#) [return](#) [reverse](#) [root-env](#)
[run-context](#) [scan-number](#) [seq](#) [setdyn](#) [short-fn](#) [signal](#) [slice](#) [slurp](#) [some](#) [sort](#)
[sort-by](#) [sorted](#) [sorted-by](#) [spit](#) [stderr](#) [stdin](#) [stdout](#) [string](#) [string/ascii-lower](#)
[string/ascii-upper](#) [string/bytes](#) [string/check-set](#) [string/find](#) [string/find-all](#)
[string/format](#) [string/from-bytes](#) [string/has-prefix?](#) [string/has-suffix?](#)
[string/join](#) [string/repeat](#) [string/replace](#) [string/replace-all](#) [string/reverse](#)
[string/slice](#) [string/split](#) [string/trim](#) [string/triml](#) [string/trimr](#) [string? struct](#)
[struct?](#) [sum](#) [symbol](#) [symbol?](#) [table](#) [table/clone](#) [table/getproto](#) [table/new](#)
[table/rawget](#) [table/setproto](#) [table/to-struct](#) [table?](#) [take](#) [take-until](#) [take-while](#)
[tarray/buffer](#) [tarray/copy-bytes](#) [tarray/length](#) [tarray/new](#) [tarray/properties](#)
[tarray/slice](#) [tarray/swap-bytes](#) [thread/close](#) [thread/current](#) [thread/new](#)
[thread/receive](#) [thread/send](#) [trace](#) [true?](#) [truthy?](#) [try](#) [tuple](#) [tuple/brackets](#)
[tuple/setmap](#) [tuple/slice](#) [tuple/sourcemap](#) [tuple/type](#) [tuple?](#) [type](#) [unless](#)

[unmarshal](#) [untrace](#) [update](#) [update-in](#) [use](#) [values](#) [var-](#) [varfn](#) [varglobal](#) [walk](#)
[when](#) [when-let](#) [when-with](#) [with](#) [with-dyns](#) [with-syms](#) [with-vars](#) [yield](#) [zero?](#)
[zipcoll](#)

%₀

function

(% dividend divisor)

Returns the remainder of dividend / divisor.

EXAMPLES

```
(% 10 3) # -> 1
(% -10 3) # -> -1
(% 10 -3) # -> 1
(% -10 -3) # -> -1
(% 1.4 1) # -> 0.4
(% -1.4 1) # -> -0.4
(% 1.4 1) # -> 0.4
(% -1.4 1) # -> -0.4
(% -1.4 0) # -> -nan
(% -1.4 0) # -> -nan
```

%₌

macro

[source](#)

(%₌ x n)

Shorthand for (set x (% x n)).

EXAMPLES

```
(var x 10) # -> 10
(%= x 3) # -> 1
x # -> 1
```

*

function

(* & xs)

Returns the product of all elements in xs. If xs is empty, returns 1.

EXAMPLES

```
(*) # -> 1
(* 10) # -> 10
(* 10 20) # -> 200
(* 1 2 3 4 5 6 7) #-> 5040

# Can take product of array with splice, but
'product' is better
(* ;(range 1 20)) #-> 1.21645e+17
(product (range 1 20)) #-> 1.21645e+17
```

*=

macro

[source](#)

(*= x n)

Shorthand for (set x (* x n)).

EXAMPLES

```
(var x 100) # -> 100
x # -> 100
(*= x 10) # -> 1000
x # -> 1000
```

+

function

(+ & xs)

Returns the sum of all xs. xs must be integers or real numbers only. If xs is empty, return 0.

EXAMPLES

```
(+) # -> 0
(+ 10) # -> 10
(+ 1 2) # -> 3
(+ 1.4 -4.5) # -> -3.1
(+ 1 2 3 4 5 6 7 8 9 10) # -> 55

# Splice can be used to sum arrays, but 'sum'
is better
(+ ;(range 101)) # -> 5050
(sum (range 101)) # -> 5050

# Janet can add types that support the :+ or
:r+ method
(+ (int/s64 "10") 10) # -> <core/s64 20>

# Bad types give errors
(+ nil 10) # -> error: could not find method :+
```

```
for nil, or :r+ for 10
```

`++`

macro

[source](#)

```
(++ x)
```

Increments the var x by 1.

`+=`

macro

[source](#)

```
(+= x n)
```

Increments the var x by n.

EXAMPLES

```
(var x 100) # -> 100
x # -> 100
(+= x 10) # -> 110
x # -> 110
```

-

function

```
(- & xs)
```

Returns the difference of xs. If xs is empty, returns 0. If xs has one

element, returns the negative value of that element. Otherwise, returns the first element in xs minus the sum of the rest of the elements.

EXAMPLES

```
(-) # -> 0
(- 10) # -> -10
(- 1 2) # -> -1
(+ 1.4 -4.5) # -> 5.9

# Equivalent to (- first (+ ;rest))
(- 1 2 3 4 5 6 7 8 9 10) # -> -53

# Janet can subtract types that support the :- or :r- method
(- (int/s64 "10") 10) # -> <core/s64 0>
```

--
macro
[source](#)

(-- x)

Decrements the var x by 1.

--
macro
[source](#)

(-= x n)

Decrements the var x by n.

EXAMPLES

```
(var x 10) # -> 10
(-= x 20) # -> -10
x # -> -10
```

->

macro

[source](#)

(-> x & forms)

Threading macro. Inserts x as the second value in the first form in forms, and inserts the modified first form into the second form in the same manner, and so on. Useful for expressing pipelines of data.

->>

macro

[source](#)

(->> x & forms)

Threading macro. Inserts x as the last value in the first form in forms, and inserts the modified first form into the second form in the same manner, and so on. Useful for expressing pipelines of data.

-?>

macro

[source](#)

(-?> x & forms)

Short circuit threading macro. Inserts x as the second value in the first form in forms, and inserts the modified first form into the second form in the same manner, and so on. The pipeline will return nil if an intermediate value is nil. Useful for expressing pipelines of data.

-?>>

macro

[source](#)

(-?>> x & forms)

Short circuit threading macro. Inserts x as the last value in the first form in forms, and inserts the modified first form into the second form in the same manner, and so on. The pipeline will return nil if an intermediate value is nil. Useful for expressing pipelines of data.

/

function

(/ & xs)

Returns the quotient of xs. If xs is empty, returns 1. If xs has one value x, returns the reciprocal of x. Otherwise return the first value of xs repeatedly divided by the remaining values.

EXAMPLES

```
(/) # -> 1
(/ 10) # -> 0.1
(/ 10 5) # -> 2
(/ 10 5 4) # -> 0.5
```

```
(/ 10 5 4 5) # -> 0.1  
  
# More arguments is the same as repeated  
division.  
(/ ;(range 1 20)) # -> 8.22064e-18
```

/=

macro

[source](#)

(/= x n)

Shorthand for (set x (/ x n)).

EXAMPLES

```
(var x 10) # -> 10  
(/= x 5) # -> 2  
x # -> 2  
(/= x 0.2) # -> 10  
x # -> 10
```

<

function

(< & xs)

Check if xs is in ascending order. Returns a boolean.

<=

function

(\leq & xs)

Check if xs is in non-descending order. Returns a boolean.

=

function

(= & xs)

Check if all values in xs are equal. Returns a boolean.

>

function

(> & xs)

Check if xs is in descending order. Returns a boolean.

\geq

function

(>= & xs)

Check if xs is in non-ascending order. Returns a boolean.

abstract?

cfunction

(abstract? x)

Check if x is an abstract type.

accumulate

function

[source](#)

(accumulate f init ind)

Similar to reduce, but accumulates intermediate values into an array.
The last element in the array is what would be the return value from
reduce. The init value is not added to the array. Returns a new array.

accumulate2

function

[source](#)

(accumulate2 f ind)

The 2 argument version of accumulate that does not take an
initialization value.

all

function

[source](#)

(all pred xs)

Returns true if all xs are truthy, otherwise the result of first falsey
predicate value, (pred x).

all-bindings

function

[source](#)

(all-bindings &opt env local)

Get all symbols available in an environment. Defaults to the current fiber's environment. If local is truthy, will not show inherited bindings (from prototype tables).

all-dynamics

function

[source](#)

(all-dynamics &opt env local)

Get all dynamic bindings in an environment. Defaults to the current fiber's environment. If local is truthy, will not show inherited bindings (from prototype tables).

and

macro

[source](#)

(and & forms)

Evaluates to the last argument if all preceding elements are truthy, otherwise evaluates to the first falsey argument.

apply

function

(apply f & args)

Applies a function to a variable number of arguments. Each element in args is used as an argument to f, except the last element in args, which is expected to be an array-like. Each element in this last argument is then also pushed as an argument to f. For example:

(apply + 1000 (range 10))

sums the first 10 integers and 1000.

EXAMPLES

```
(apply + (range 10)) # -> 45
(apply + []) # -> 0
(apply + 1 2 3 4 5 6 7 [8 9 10]) # -> 55
(apply + 1 2 3 4 5 6 7 8 9 10) # -> error:
expected array|tuple, got number

# Can also be used to call macros like
functions.
# Will return the macro expanded code of the
original macro.
(apply for 'x 0 10 ['(print x)])
# -> (do (var _000000 0) (def _000001 10)
(while ...
```

array

cfunction

(array & items)

Create a new array that contains items. Returns the new array.

array(concat)

cfunction

(array(concat arr & parts))

Concatenates a variadic number of arrays (and tuples) into the first argument which must be an array. If any of the parts are arrays or tuples, their elements will be inserted into the array. Otherwise, each part in parts will be appended to arr in order. Returns the modified array arr.

array(ensure)

cfunction

(array(ensure arr capacity growth))

Ensures that the memory backing the array is large enough for capacity items at the given rate of growth. Capacity and growth must be integers. If the backing capacity is already enough, then this function does nothing. Otherwise, the backing memory will be reallocated so that there is enough space.

array(fill)

cfunction

(array(fill arr &opt value))

Replace all elements of an array with value (defaulting to nil) without changing the length of the array. Returns the modified array.

array/insert

cfunction

(array/insert arr at & xs)

Insert all of xs into array arr at index at. at should be an integer 0 and the length of the array. A negative value for at will index from the end of the array, such that inserting at -1 appends to the array. Returns the array.

array/new

cfunction

(array/new capacity)

Creates a new empty array with a pre-allocated capacity. The same as (array) but can be more efficient if the maximum size of an array is known.

EXAMPLES

```
(def arr (array/new 100)) # -> @[]  
  
# Now we can fill up the array without  
triggering a resize  
(for i 0 100  
  (put arr i i))
```

array/new-filled

cfunction

(array/new-filled count &opt value)

Creates a new array of count elements, all set to value, which defaults to nil. Returns the new array.

array/peek

cfunction

(array/peek arr)

Returns the last element of the array. Does not modify the array.

array/pop

cfunction

(array/pop arr)

Remove the last element of the array and return it. If the array is empty, will return nil. Modifies the input array.

array/push

cfunction

(array(push arr x)

Insert an element in the end of an array. Modifies the input array and returns it.

array/remove

cfunction

(array/remove arr at &.opt n)

(array/remove arr at &opt n)

Remove up to n elements starting at index at in array arr. at can index from the end of the array with a negative index, and n must be a non-negative integer. By default, n is 1. Returns the array.

array/slice

cfunction

(array/slice arrtup &opt start end)

Takes a slice of array or tuple from start to end. The range is half open, [start, end). Indexes can also be negative, indicating indexing from the end of the end of the array. By default, start is 0 and end is the length of the array. Note that index -1 is synonymous with index (length arrtup) to allow a full negative slice range. Returns a new array.

array?

function

[source](#)

(array? x)

Check if x is an array.

as->

macro

[source](#)

(as-> x as & forms)

Thread forms together, replacing as in forms with the value of the

previous form. The first for is the value x. Returns the last value.

as?->

macro

[source](#)

(as?-> x as & forms)

Thread forms together, replacing as in forms with the value of the previous form. The first for is the value x. If any intermediate values are falsey, return nil; otherwise, returns the last value.

asm

cfunction

(asm assembly)

Returns a new function that is the compiled result of the assembly. The syntax for the assembly can be found on the Janet website. Will throw an error on invalid assembly.

assert

function

[source](#)

(assert x &opt err)

Throw an error if x is not truthy.

bad-compile

function

[source](#)

(bad-compile msg macrof where)

Default handler for a compile error.

bad-parse

function

[source](#)

(bad-parse p where)

Default handler for a parse error.

band

function

(band & xs)

Returns the bit-wise and of all values in xs. Each x in xs must be an integer.

blshift

function

(blshift x & shifts)

Returns the value of x bit shifted left by the sum of all values in shifts. x and each element in shift must be an integer.

bnot

function

(bnot x)

Returns the bit-wise inverse of integer x.

boolean?

function

[source](#)

(boolean? x)

Check if x is a boolean.

bor

function

(bor & xs)

Returns the bit-wise or of all values in xs. Each x in xs must be an integer.

brshift

function

(brshift x & shifts)

Returns the value of x bit shifted right by the sum of all values in shifts. x and each element in shift must be an integer.

brushift

function

(brushift x & shifts)

Returns the value of x bit shifted right by the sum of all values in shifts. x and each element in shift must be an integer. The sign of x is not preserved, so for positive shifts the return value will always be positive.

buffer

cfunction

(buffer & xs)

Creates a new buffer by concatenating values together. Values are converted to bytes via describe if they are not byte sequences. Returns the new buffer.

buffer/bit

cfunction

(buffer/bit buffer index)

Gets the bit at the given bit-index. Returns true if the bit is set, false if not.

buffer/bit-clear

cfunction

(buffer/bit-clear buffer index)

Clears the bit at the given bit-index. Returns the buffer.

buffer/bit-set

cfunction

(buffer/bit-set buffer index)

Sets the bit at the given bit-index. Returns the buffer.

buffer/bit-toggle

cfunction

(buffer/bit-toggle buffer index)

Toggles the bit at the given bit index in buffer. Returns the buffer.

buffer/blit

cfunction

(buffer/blit dest src &opt dest-start src-start src-end)

Insert the contents of src into dest. Can optionally take indices that indicate which part of src to copy into which part of dest. Indices can be negative to index from the end of src or dest. Returns dest.

buffer/clear

cfunction

(buffer/clear buffer)

Sets the size of a buffer to 0 and empties it. The buffer retains its memory so it can be efficiently refilled. Returns the modified buffer.

buffer/fill

cfunction

(buffer/fill buffer &opt byte)

Fill up a buffer with bytes, defaulting to 0s. Does not change the buffer's length. Returns the modified buffer.

buffer/format

cfunction

(buffer/format buffer format & args)

Snprintf like functionality for printing values into a buffer. Returns the modified buffer.

buffer/new

cfunction

(buffer/new capacity)

Creates a new, empty buffer with enough backing memory for capacity bytes. Returns a new buffer of length 0.

buffer/new-filled

cfunction

(buffer/new-filled count &opt byte)

Creates a new buffer of length count filled with byte. By default, byte is 0. Returns the new buffer.

buffer/popn

cfunction

(buffer/popn buffer n)

Removes the last n bytes from the buffer. Returns the modified buffer.

buffer/push-byte

cfunction

(buffer/push-byte buffer x)

Append a byte to a buffer. Will expand the buffer as necessary.
Returns the modified buffer. Will throw an error if the buffer overflows.

buffer/push-string

cfunction

(buffer/push-string buffer str)

Push a string onto the end of a buffer. Non string values will be converted to strings before being pushed. Returns the modified buffer.
Will throw an error if the buffer overflows.

buffer/push-word

cfunction

(buffer/push-word buffer x)

Append a machine word to a buffer. The 4 bytes of the integer are appended in two's complement, little endian order, unsigned. Returns the modified buffer. Will throw an error if the buffer overflows.

buffer/slice

cfunction

(buffer/slice bytes &opt start end)

Takes a slice of a byte sequence from start to end. The range is half open, [start, end). Indexes can also be negative, indicating indexing from the end of the end of the array. By default, start is 0 and end is the length of the buffer. Returns a new buffer.

buffer?

function

[source](#)

(buffer? x)

Check if x is a buffer.

bxor

function

(bxor & xs)

Returns the bit-wise xor of all values in xs. Each in xs must be an integer.

bytes?

function

[source](#)

(bytes? x)

Check if x is a string, symbol, or buffer.

case

macro

[source](#)

(case dispatch & pairs)

Select the body that equals the dispatch value. When pairs has an odd number of arguments, the last is the default expression. If no match is found, returns nil.

cfunction?

function

[source](#)

(cfunction? x)

Check if x a cfunction.

chr

macro

[source](#)

(chr c)

Convert a string of length 1 to its byte (ascii) value at compile time.

cli-main

function

[source](#)

(cli-main args)

Entrance for the Janet CLI tool. Call this functions with the command line arguments as an array or tuple of strings to invoke the CLI interface.

comment

macro

[source](#)

(comment &)

Ignores the body of the comment.

comp

function

[source](#)

(comp & functions)

Takes multiple functions and returns a function that is the composition
of those functions

of those functions.

compile

cfunction

(compile ast &opt env source)

Compiles an Abstract Syntax Tree (ast) into a function. Pair the compile function with parsing functionality to implement eval. Returns a new function and does not modify ast. Returns an error struct with keys :line, :column, and :error if compilation fails.

complement

function

[source](#)

(complement f)

Returns a function that is the complement to the argument.

comptime

macro

[source](#)

(comptime x)

Evals x at compile time and returns the result. Similar to a top level unquote.

cond

macro

[source](#)

(cond & pairs)

Evaluates conditions sequentially until the first true condition is found, and then executes the corresponding body. If there are an odd number of forms, the last expression is executed if no forms are matched. If there are no matches, return nil.

coro

macro

[source](#)

(coro & body)

A wrapper for making fibers. Same as (fiber/new (fn [] ;body) :yi).

count

function

[source](#)

(count pred ind)

Count the number of items in ind for which (pred item) is true.

debug

function

(debug &opt x)

Throws a debug signal that can be caught by a parent fiber and used to inspect the running state of the current fiber. Returns the value

passed in by resume.

debug/arg-stack

cfunction

(debug/arg-stack fiber)

Gets all values currently on the fiber's argument stack. Normally, this should be empty unless the fiber signals while pushing arguments to make a function call. Returns a new array.

debug/break

cfunction

(debug/break source byte-offset)

Sets a breakpoint with source a key at a given line and column. Will throw an error if the breakpoint location cannot be found. For example

(debug/break "core.janet" 1000)

wil set a breakpoint at the 1000th byte of the file core.janet.

debug/fbreak

cfunction

(debug/fbreak fun &opt pc)

Set a breakpoint in a given function. pc is an optional offset, which is in bytecode instructions. fun is a function value. Will throw an error if the offset is too large or negative

~~offset is too large or negative.~~

debug/lineage

cfunction

(debug/lineage fib)

Returns an array of all child fibers from a root fiber. This function is useful when a fiber signals or errors to an ancestor fiber. Using this function, the fiber handling the error can see which fiber raised the signal. This function should be used mostly for debugging purposes.

debug/stack

cfunction

(debug/stack fib)

Gets information about the stack as an array of tables. Each table in the array contains information about a stack frame. The top most, current stack frame is the first table in the array, and the bottom most stack frame is the last value. Each stack frame contains some of the following attributes:

- :c - true if the stack frame is a c function invocation
- :column - the current source column of the stack frame
- :function - the function that the stack frame represents
- :line - the current source line of the stack frame
- :name - the human friendly name of the function
- :pc - integer indicating the location of the program counter
- :source - string with the file path or other identifier for the source code
- :slots - array of all values in each slot

:tail - boolean indicating a tail call

debug/stacktrace

cfunction

(debug/stacktrace fiber err)

Prints a nice looking stacktrace for a fiber. The error message err must be passed to the function as fiber's do not keep track of the last error they have thrown. Returns the fiber.

debug/step

cfunction

(debug/step fiber &opt x)

Run a fiber for one virtual instruction of the Janet machine. Can optionally pass in a value that will be passed as the resuming value. Returns the signal value, which will usually be nil, as breakpoints raise nil signals.

debug/unbreak

cfunction

(debug/unbreak source line column)

Remove a breakpoint with a source key at a given line and column. Will throw an error if the breakpoint cannot be found.

debug/unfbreak

cfunction

(debug/unfbreak fun &opt pc)

Unset a breakpoint set with debug/fbreak.

debugger-env

table

[source](#)

An environment that contains dot prefixed functions for debugging.

dec

function

[source](#)

(dec x)

Returns x - 1.

deep-not=

function

[source](#)

(deep-not= x y)

Like not=, but mutable types (arrays, tables, buffers) are considered equal if they have identical structure. Much slower than not=.

deep=

function

[source](#)

(deep= x y)

Like =, but mutable types (arrays, tables, buffers) are considered equal if they have identical structure. Much slower than =.

def-

macro

[source](#)

(def- name & more)

Define a private value that will not be exported.

EXAMPLES

```
# In a file module.janet
(def- private-thing :encapsulated)
(def public-thing :exposed)

# In a file main.janet
(import module)

module/private-thing # -> Unknown symbol
module/public-thing # -> :exposed

# Same as normal def with :private metadata
(def :private x private-thing :encapsulated)
```

default

macro

[source](#)

(default sym val)

Define a default value for an optional argument. Expands to (def sym
(if (= nil sym) val sym))

default-peg-grammar

table

[source](#)

The default grammar used for pegs. This grammar defines several common patterns that should make it easier to write more complex patterns.

defer

macro

[source](#)

(defer form & body)

Run form unconditionally after body, even if the body throws an error. Will also run form if a user signal 0-4 is received.

EXAMPLES

```
# Evaluates to 6 after printing "scope left!"  
(defer (print "scope left!")  
      (+ 1 2 3))  
  
# cleanup will always be called, even if there  
is a failure  
(defer (cleanup))
```

```
(step-1)
(step-2)
(if (< 0.1 (math/random)) (error "failure"))
(step-3))
```

defglobal

function

[source](#)

(defglobal name value)

Dynamically create a global def.

defmacro

macro

[source](#)

(defmacro name & more)

Define a macro.

defmacro-

macro

[source](#)

(defmacro- name & more)

Define a private macro that will not be exported.

defn

macro

macro

[source](#)

(defn name & more)

Define a function. Equivalent to (def name (fn name [args] ...)).

EXAMPLES

```
(defn simple
  [x]
  (print (+ x 1)))

(simple 10) # -> 11

(defn long-body
  [y]
  (print y)
  (print (+ y 1))
  (print (+ y 2))
  (+ y 3))

(defn with-docstring
  "This function has a docstring"
  []
  (print "hello!"))

(defn with-tags
  :tag1 :tag2 :private
  "Also has a docstring and a variadic argument
'more'!"
  [x y z & more]
  [x y z more])

(with-metadata 1 2) # raises arity error
(with-metadata 1 2 3) # -> (1 2 3 ())
(with-metadata 1 2 3 4) # -> (1 2 3 (4))
```

```
(with-metadata 1 2 3 4 5) # -> (1 2 3 (4 5))

# Tags (and other metadata) are (usually)
visible in the environment.
(dyn 'with-tags) # -> @{:tag2 true :value
<function with-tags> :doc "(with-tags x y z &
more)\n\nAlso has a docstring..." :source-map
("repl" 4 1) :tag1 true :private true}
```

defn-

macro

[source](#)

(defn- name & more)

Define a private function that will not be exported.

EXAMPLES

```
# In a file module.janet
(defn- not-exposed-fn
  [x]
  (+ x x))
(not-exposed-fn 10) # -> 20

# In a file main.janet
(import module)

(module/not-exposed-fn 10) # -> Unknown symbol
error

# Same as
(defn not-exposed-fn
```

```
:private  
[x]  
(+ x x))
```

describe

cfunction

```
(describe x)
```

Returns a string that is a human readable description of a value x.

dictionary?

function

[source](#)

```
(dictionary? x)
```

Check if x a table or struct.

disasm

cfunction

```
(disasm func)
```

Returns assembly that could be used be compile the given function.
func must be a function, not a c function. Will throw on error on a badly
typed argument.

distinct

function

[source](#)

(distinct xs)

Returns an array of the deduplicated values in xs.

doc

macro

[source](#)

(doc &opt sym)

Shows documentation for the given symbol, or can show a list of available bindings. If sym is a symbol, will look for documentation for that symbol. If sym is a string or is not provided, will show all lexical and dynamic bindings in the current environment with that prefix (all bindings will be shown if no prefix is given).

doc*

function

[source](#)

(doc* &opt sym)

Get the documentation for a symbol in a given environment. Function form of doc.

doc-format

function

[source](#)

(doc-format text &opt width)

Reformat text to wrap at a given line.

dofile

function

[source](#)

(dofile path &keys {:expander expander :env env :exit exit :source src :evaluator evaluator})

Evaluate a file and return the resulting environment. :env, :expander, and :evaluator are passed through to the underlying run-context call. If exit is true, any top level errors will trigger a call to (os/exit 1) after printing the error.

drop

function

[source](#)

(drop n ind)

Drop first n elements in an indexed type. Returns new indexed instance.

drop-until

function

[source](#)

(drop-until pred ind)

Same as (drop-while (complement pred) ind).

drop-while

function

[source](#)

(drop-while pred ind)

Given a predicate, remove elements from an indexed type that satisfy the predicate, and abort on first failure. Returns a new array.

dyn

cfunction

(dyn key &opt default)

Get a dynamic binding. Returns the default value (or nil) if no binding found.

each

macro

[source](#)

(each x ds & body)

Loop over each value in ds. Returns nil.

eachk

macro

[source](#)

(eachk x ds & body)

Loop over each key in ds. Returns nil.

eachp

macro

[source](#)

(eachp x ds & body)

Loop over each (key, value) pair in ds. Returns nil.

edefer

macro

[source](#)

(edefer form & body)

Run form after body in the case that body terminates abnormally (an error or user signal 0-4). Otherwise, return last form in body.

EXAMPLES

```
# Half of the time, return "ok", the other
# half of the time, print there was an error
# and throw "oops".
(edefer (print "there was an error")
  (if (< (math/random) 0.5)
    (error "oops")
    "ok"))
```

eflush

cfunction

(eflush)

Flush (dyn :err stderr) if it is a file, otherwise do nothing.

empty?

function

[source](#)

(empty? xs)

Check if xs is empty.

env-lookup

cfunction

(env-lookup env)

Creates a forward lookup table for unmarshalling from an environment.
To create a reverse lookup table, use the invert function to swap keys
and values in the returned table.

eprin

cfunction

(eprin & xs)

Same as prin, but uses (dyn :err stderr) instead of (dyn :out stdout).

eprinf

cfunction

(eprintf fmt & xs)

Like eprintf but with no trailing newline.

eprint

cfunction

(eprint & xs)

Same as print, but uses (dyn :err stderr) instead of (dyn :out stdout).

eprintf

cfunction

(eprintf fmt & xs)

Prints output formatted as if with (string/format fmt ;xs) to (dyn :err stderr) with a trailing newline.

error

function

(error e)

Throws an error e that can be caught and handled by a parent fiber.

errorf

function

[source](#)

(errorf fmt & args)

(error fmt & args)

A combination of error and string/format. Equivalent to (error (string/format fmt ;args))

eval

function

[source](#)

(eval form)

Evaluates a form in the current environment. If more control over the environment is needed, use run-context.

EXAMPLES

```
(eval '(+ 1 2 3)) # -> 6
(eval '(error :oops)) # -> raises error :oops
(eval '(+ nil nil)) # -> raises error
```

eval-string

function

[source](#)

(eval-string str)

Evaluates a string in the current environment. If more control over the environment is needed, use run-context.

EXAMPLES

```
(eval-string "(+ 1 2 3 4)") # -> 10
```

```
(eval-string ")") # -> parse error  
(eval-string "(bloop)") # -> compile error  
(eval-string "(+ nil nil)") # -> runtime error
```

even?

function

[source](#)

(even? x)

Check if x is even.

every?

function

[source](#)

(every? ind)

Returns true if each value in is truthy, otherwise the first falsey value.

extreme

function

[source](#)

(extreme order args)

Returns the most extreme value in args based on the function order.
order should take two values and return true or false (a comparison).
Returns nil if args is empty.

false?

function

[source](#)

(false? x)

Check if x is false.

fiber/can-resume?

cfunction

(fiber/can-resume? fiber)

Check if a fiber is finished and cannot be resumed.

fiber/current

cfunction

(fiber/current)

Returns the currently running fiber.

fiber/getenv

cfunction

(fiber/getenv fiber)

Gets the environment for a fiber. Returns nil if no such table is set yet.

fiber/maxstack

cfunction

(fiber/maxstack fib)

Gets the maximum stack size in janet values allowed for a fiber. While memory for the fiber's stack is not allocated up front, the fiber will not be allocated more than this amount and will throw a stack-overflow error if more memory is needed.

fiber/new

cfunction

(fiber/new func &opt sigmask)

Create a new fiber with function body func. Can optionally take a set of signals to block from the current parent fiber when called. The mask is specified as a keyword where each character is used to indicate a signal to block. The default sigmask is :y. For example,

(fiber/new myfun :e123)

blocks error signals and user signals 1, 2 and 3. The signals are as follows:

- a - block all signals
- d - block debug signals
- e - block error signals
- t - block termination signals: error + user[0-4]
- u - block user signals
- y - block yield signals
- 0-9 - block a specific user signal

The sigmask argument also can take environment flags. If any mutually

exclusive flags are present, the last flag takes precedence.

i - inherit the environment from the current fiber

p - the environment table's prototype is the current environment table

fiber/root

cfunction

(fiber/root)

Returns the current root fiber. The root fiber is the oldest ancestor that does not have a parent.

fiber/setenv

cfunction

(fiber/setenv fiber table)

Sets the environment table for a fiber. Set to nil to remove the current environment.

fiber/setmaxstack

cfunction

(fiber/setmaxstack fib maxstack)

Sets the maximum stack size in janet values for a fiber. By default, the maximum stack size is usually 8192.

fiber/status

cfunction

(fiber/status fib)

Get the status of a fiber. The status will be one of:

- :dead - the fiber has finished
 - :error - the fiber has errored out
 - :debug - the fiber is suspended in debug mode
 - :pending - the fiber has been yielded
 - :user(0-9) - the fiber is suspended by a user signal
 - :alive - the fiber is currently running and cannot be resumed
 - :new - the fiber has just been created and not yet run
-

fiber?

function

[source](#)

(fiber? x)

Check if x is a fiber.

file/close

cfunction

(file/close f)

Close a file and release all related resources. When you are done reading a file, close it to prevent a resource leak and let other processes read the file. If the file is the result of a file/popen call, close waits for and returns the process exit status.

file/flush

cfunction

(file/flush f)

Flush any buffered bytes to the file system. In most files, writes are buffered for efficiency reasons. Returns the file handle.

file/open

cfunction

(file/open path &opt mode)

Open a file. path is an absolute or relative path, and mode is a set of flags indicating the mode to open the file in. mode is a keyword where each character represents a flag. If the file cannot be opened, returns nil, otherwise returns the new file handle. Mode flags:

- r - allow reading from the file
 - w - allow writing to the file
 - a - append to the file
 - b - open the file in binary mode (rather than text mode)
 - + - append to the file instead of overwriting it
-

file/popen

cfunction

(file/popen path &opt mode)

Open a file that is backed by a process. The file must be opened in either the :r (read) or the :w (write) mode. In :r mode, the stdout of the

process can be read from the file. In :w mode, the stdin of the process can be written to. Returns the new file.

file/read

cfunction

(file/read f what &opt buf)

Read a number of bytes from a file into a buffer. A buffer can be provided as an optional fourth argument, otherwise a new buffer is created. 'what' can either be an integer or a keyword. Returns the buffer with file contents. Values for 'what':

:all - read the whole file

:line - read up to and including the next newline character

n (integer) - read up to n bytes from the file

file/seek

cfunction

(file/seek f &opt whence n)

Jump to a relative location in the file. 'whence' must be one of

:cur - jump relative to the current file location

:set - jump relative to the beginning of the file

:end - jump relative to the end of the file

By default, 'whence' is :cur. Optionally a value n may be passed for the relative number of bytes to seek in the file. n may be a real number to handle large files of more than 4GB. Returns the file handle.

file/temp

cfunction

(file/temp)

Open an anonymous temporary file that is removed on close. Raises an error on failure.

file/write

cfunction

(file/write f bytes)

Writes to a file. 'bytes' must be string, buffer, or symbol. Returns the file.

filter

function

[source](#)

(filter pred ind)

Given a predicate, take only elements from an array or tuple for which (pred element) is truthy. Returns a new array.

find

function

[source](#)

(find pred ind)

Find the first value in an indexed collection that satisfies a predicate.
Returns nil if not found. Note there is no way to differentiate a nil from
the indexed collection and a not found. Consider find-index if this is an
issue.

find-index

function

[source](#)

(find-index pred ind)

Find the index of indexed type for which pred is true. Returns nil if not
found.

first

function

[source](#)

(first xs)

Get the first element from an indexed data structure.

flatten

function

[source](#)

(flatten xs)

Takes a nested array (tree), and returns the depth first traversal of that
array. Returns a new array.

flatten-into

function

[source](#)

(flatten-into into xs)

Takes a nested array (tree), and appends the depth first traversal of that array to an array 'into'. Returns array into.

flush

cfunction

(flush)

Flush (dyn :out stdout) if it is a file, otherwise do nothing.

for

macro

[source](#)

(for i start stop & body)

Do a c style for loop for side effects. Returns nil.

freeze

function

[source](#)

(freeze x)

Freeze an object (make it immutable) and do a deep copy, making child values also immutable. Closures, fibers, and abstract types will

not be recursively frozen, but all other types will.

frequencies

function

[source](#)

(frequencies ind)

Get the number of occurrences of each value in a indexed structure.

function?

function

[source](#)

(function? x)

Check if x is a function (not a cfunction).

gccollect

cfunction

(gccollect)

Run garbage collection. You should probably not call this manually.

gcinterval

cfunction

(gcinterval)

Returns the integer number of bytes to allocate before running an

iteration of garbage collection.

gcsetinterval

cfunction

(gcsetinterval interval)

Set an integer number of bytes to allocate before running garbage collection. Low values for interval will be slower but use less memory. High values will be faster but use more memory.

generate

macro

[source](#)

(generate head & body)

Create a generator expression using the loop syntax. Returns a fiber that yields all values inside the loop in order. See loop for details.

EXAMPLES

```
# An infinite stream of random numbers, but doubled.  
(def g (generate [_ :iterate true :repeat 2]  
  (math/random)))  
# -> <fiber 0x5562863141E0>  
  
(resume g) # -> 0.487181  
(resume g) # -> 0.487181  
(resume g) # -> 0.507917  
(resume g) # -> 0.507917  
# ...
```

gensym

cfunction

(gensym)

Returns a new symbol that is unique across the runtime. This means it will not collide with any already created symbols during compilation, so it can be used in macros to generate automatic bindings.

get

function

(get ds key &opt dflt)

Get the value mapped to key in data structure ds, and return dflt or nil if not found. Similar to in, but will not throw an error if the key is invalid for the data structure unless the data structure is an abstract type. In that case, the abstract type getter may throw an error.

get-in

function

[source](#)

(get-in ds ks &opt dflt)

Access a value in a nested data structure. Looks into the data structure via a sequence of keys.

getline

cfunction

(getline &opt prompt buf env)

Reads a line of input into a buffer, including the newline character, using a prompt. An optional environment table can be provided for auto-complete. Returns the modified buffer. Use this function to implement a simple interface for a terminal program.

hash

cfunction

(hash value)

Gets a hash for any value. The hash is an integer can be used as a cheap hash function for all values. If two values are strictly equal, then they will have the same hash value.

idempotent?

function

[source](#)

(idempotent? x)

Check if x is a value that evaluates to itself when compiled.

identity

function

[source](#)

(identity x)

A function that returns its first argument.

if-let

macro

[source](#)

(if-let bindings tru &opt fal)

Make multiple bindings, and if all are truthy, evaluate the tru form. If any are false or nil, evaluate the fal form. Bindings have the same syntax as the let macro.

if-not

macro

[source](#)

(if-not condition then &opt else)

Shorthand for (if (not condition) else then).

if-with

macro

[source](#)

(if-with [binding ctor dtor] truthy &opt falsey)

Similar to with, but if binding is false or nil, evaluates the falsey path. Otherwise, evaluates the truthy path. In both cases, ctor is bound to binding.

import

macro

[source](#)

(import path & args)

Import a module. First requires the module, and then merges its symbols into the current environment, prepending a given prefix as needed. (use the :as or :prefix option to set a prefix). If no prefix is provided, use the name of the module as a prefix. One can also use :export true to re-export the imported symbols. If :exit true is given as an argument, any errors encountered at the top level in the module will cause (os/exit 1) to be called. Dynamic bindings will NOT be imported.

import*

function

[source](#)

(import* path & args)

Function form of import. Same parameters, but the path and other symbol parameters should be strings instead.

in

function

(in ds key &opt dflt)

Get value in ds at key, works on associative data structures. Arrays, tuples, tables, structs, strings, symbols, and buffers are all associative and can be used. Arrays, tuples, strings, buffers, and symbols must use integer keys that are in bounds or an error is raised. Structs and tables can take any value as a key except nil and will return nil or dflt if

names can take any value as a key except nil and will return nil or unit if not found.

inc

function

[source](#)

(inc x)

Returns x + 1.

indexed?

function

[source](#)

(indexed? x)

Check if x is an array or tuple.

int/s64

cfunction

(int/s64 value)

Create a boxed signed 64 bit integer from a string value.

int/u64

cfunction

(int/u64 value)

Create a boxed unsigned 64 bit integer from a string value.

int?

cfunction

(int? x)

Check if x can be exactly represented as a 32 bit signed two's complement integer.

interleave

function

[source](#)

(interleave & cols)

Returns an array of the first elements of each col, then the second, etc.

interpose

function

[source](#)

(interpose sep ind)

Returns a sequence of the elements of ind separated by sep. Returns a new array.

invert

function

[source](#)

..

(invert ds)

Returns a table of where the keys of an associative data structure are the values, and the values of the keys. If multiple keys have the same value, one key will be ignored.

janet/build

string

The build identifier of the running janet program.

janet/config-bits

number

The flag set of config options from janetcconf.h which is used to check if native modules are compatible with the host program.

janet/version

string

The version number of the running janet program.

juxt

macro

[source](#)

(juxt & funs)

Macro form of `juxt*`. Same behavior but more efficient.

juxt*

function

[source](#)

(juxt* & funs)

Returns the juxtaposition of functions. In other words, ((juxt* a b c) x) evaluates to [(a x) (b x) (c x)].

keep

function

[source](#)

(keep pred ind)

Given a predicate, take only elements from an array or tuple for which (pred element) is truthy. Returns a new array of truthy predicate results.

keys

function

[source](#)

(keys x)

Get the keys of an associative data structure.

keyword

cfunction

(keyword & xs)

Creates a keyword by concatenating values together. Values are converted to bytes via describe if they are not byte sequences. Returns the new keyword.

keyword?

function

[source](#)

(keyword? x)

Check if x is a keyword.

kvs

function

[source](#)

(kvs dict)

Takes a table or struct and returns and array of key value pairs like @[k v k v ...]. Returns a new array.

label

macro

[source](#)

(label name & body)

Set a label point that is lexically scoped. Name should be a symbol that will be bound to the label.

last

function

[source](#)

(last xs)

Get the last element from an indexed data structure.

length

function

(length ds)

Returns the length or count of a data structure in constant time as an integer. For structs and tables, returns the number of key-value pairs in the data structure.

let

macro

[source](#)

(let bindings & body)

Create a scope and bind values to symbols. Each pair in bindings is assigned as if with def, and the body of the let form returns the last value.

load-image

function

[source](#)

(load-image image)

The inverse operation to make-image. Returns an environment.

load-image-dict

table

[source](#)

A table used in combination with unmarshal to unmarshal byte sequences created by make-image, such that (load-image bytes) is the same as (unmarshal bytes load-image-dict).

loop

macro

[source](#)

(loop head & body)

A general purpose loop macro. This macro is similar to the Common Lisp loop macro, although intentionally much smaller in scope. The head of the loop should be a tuple that contains a sequence of either bindings or conditionals. A binding is a sequence of three values that define something to loop over. They are formatted like:

binding :verb object/expression

Where binding is a binding as passed to def, :verb is one of a set of keywords, and object is any expression. The available verbs are:

:iterate - repeatedly evaluate and bind to the expression while it is truthy.

:range - loop over a range. The object should be two element tuple with a start and end value, and an optional positive step. The

range is half open, [start, end).

:range-to - same as :range, but the range is inclusive [start, end].

:down - loop over a range, stepping downwards. The object should be two element tuple with a start and (exclusive) end value, and an optional (positive!) step size.

:down-to - same as down, but the range is inclusive [start, end].

:keys - iterate over the keys in a data structure.

:pairs - iterate over the keys value pairs as tuples in a data structure.

:in - iterate over the values in a data structure.

:generate - iterate over values yielded from a fiber. Can be paired with the generator function for the producer/consumer pattern.

loop also accepts conditionals to refine the looping further.

Conditionals are of the form:

:modifier argument

where :modifier is one of a set of keywords, and argument is keyword dependent. :modifier can be one of:

:while expression - breaks from the loop if expression is falsey.

:until expression - breaks from the loop if expression is truthy.

:let bindings - defines bindings inside the loop as passed to the let macro.

:before form - evaluates a form for a side effect before of the next inner loop.

:after form - same as :before, but the side effect happens after the next inner loop.

:repeat n - repeats the next inner loop n times.

:when condition - only evaluates the loop body when condition is true.

The loop macro always evaluates to nil.

EXAMPLES

```
# -> prints 0123456789 (not followed by
newline)
(loop [x :range [0 10]]
  (prin x))

# Cartesian product (nested loops)

# -> prints 00010203101112132021222330313233
# Same as (for x 0 4 (for y 0 4 (prin x y)))
(loop [x :range [0 4]
       y :range [0 4]]
  (prin x y))

# -> prints bytes of "hello, world" as numbers
(loop [character :in "hello, world"]
  (print character))

# -> prints 1, 2, and 3, in an unspecified
order
(loop [value :in {:a 1 :b 2 :c 3}]
  (print value))

# -> prints 0 to 99 inclusive
(loop [x :in (range 100)]
  (print x))

# Complex body
(loop [x :in (range 10)]
  (print x)
  (print (inc c))
  (print (+ x 2)))
```

```

# Iterate over keys
(loop [k :keys {:a 1 :b 2 :c 3}]
  (print k))
# print a, b, and c in an unspecified order

(loop [index :keys [:a :b :c :d]]
  (print index))
# print 0, 1, 2, and 3 in order.

(defn print-pairs
  [x]
  (loop [[k v] :pairs x]
    (printf "[%v]=%v" k v)))

(print-pairs [:a :b :c])
# [0]=:a
# [1]=:b
# [2]=:c

(print-pairs {:a 1 :b 2 :c 3})
# [:a]=1
# [:b]=2
# [:c]=3

# Some modifiers - allow early termination and
conditional execution
# of the loop

(loop [x :range [0 100] :when (even? x)]
  (print x))
# prints even numbers 0, 2, 4, ..., 98

(loop [x :range [1 100] :while (pos? (% x 7))]
  (print x))
# prints 1, 2, 3, 4, 5, 6

```

```

# Consume fibers as generators
(def f
  (fiber/new
    (fn []
      (for i 0 100
        (yield i)))))

(loop [x :generate f]
  (print x))
# print 0, 1, 2, ... 99

# Modifiers in nested loops

(loop [x :range [0 10]
       :after (print)
       y :range [0 x]]
  (prin y " "))
# 0
# 0 1
# 0 1 2
# 0 1 2 3
# 0 1 2 3 4
# 0 1 2 3 4 5
# 0 1 2 3 4 5 6
# 0 1 2 3 4 5 6 7
# 0 1 2 3 4 5 6 7 8

```

macex

function

[source](#)

(macex x &opt on-binding)

Expand macros completely. on-binding is an optional callback whenever a normal symbolic binding is encountered. This allows macros

to easily see all bindings used by their arguments by calling macex on their contents. The binding itself is also replaced by the value returned by on-binding within the expand macro.

macex1

function

[source](#)

(macex1 x &opt on-binding)

Expand macros in a form, but do not recursively expand macros. See macex docs for info on on-binding.

make-env

function

[source](#)

(make-env &opt parent)

Create a new environment table. The new environment will inherit bindings from the parent environment, but new bindings will not pollute the parent environment.

make-image

function

[source](#)

(make-image env)

Create an image from an environment returned by require. Returns the image source as a string.

make-image-dict

table

[source](#)

A table used in combination with marshal to marshal code (images), such that (make-image x) is the same as (marshal x make-image-dict).

map

function

[source](#)

(map f & inds)

Map a function over every element in an indexed data structure and return an array of the results.

mapcat

function

[source](#)

(mapcat f ind)

Map a function over every element in an array or tuple and use array to concatenate the results.

marshal

cfunction

(marshal x &opt reverse-lookup buffer)

Marshal a value into a buffer and return the buffer. The buffer can be

later be unmarshalled to reconstruct the initial value. Optionally, one can pass in a reverse lookup table to not marshal aliased values that are found in the table. Then a forwardlookup table can be used to recover the original value when unmarshalling.

match

macro

[source](#)

(match x & cases)

Pattern matching. Match an expression x against any number of cases. Each case is a pattern to match against, followed by an expression to evaluate to if that case is matched. A pattern that is a symbol will match anything, binding x's value to that symbol. An array will match only if all of its elements match the corresponding elements in x. A table or struct will match if all values match with the corresponding values in x. A tuple pattern will match if its first element matches, and the following elements are treated as predicates and are true. The last special case is the '_' symbol, which is a wildcard that will match any value without creating a binding. Any other value pattern will only match if it is equal to x.

math/-inf

number

The number representing negative infinity

math/abs

cfunction

(math/abs x)

Return the absolute value of x.

math/acos

cfunction

(math/acos x)

Returns the arccosine of x.

math/acosh

cfunction

(math/acosh x)

Return the hyperbolic arccosine of x.

math/asin

cfunction

(math/asin x)

Returns the arcsine of x.

math/asinh

cfunction

(math/asinh x)

Return the hyperbolic arcsine of x.

math/atan

cfunction

(math/atan x)

Returns the arctangent of x.

math/atan2

cfunction

(math/atan2 y x)

Return the arctangent of y/x. Works even when x is 0.

math/atanh

cfunction

(math/atanh x)

Return the hyperbolic arctangent of x.

math/cbrt

cfunction

(math/cbrt x)

Returns the cube root of x.

math/ceil

cfunction

(math/ceil x)

Returns the smallest integer value number that is not less than x.

math/cos

cfunction

(math/cos x)

Returns the cosine of x.

math/cosh

cfunction

(math/cosh x)

Return the hyperbolic cosine of x.

math/e

number

The base of the natural log.

math/erf

cfunction

(math/erf x)

Returns the error function of x.

math/erfc

cfunction

(math/erfc x)

Returns the complementary error function of x.

math/exp

cfunction

(math/exp x)

Returns e to the power of x.

math/exp2

cfunction

(math/exp2 x)

Returns 2 to the power of x.

math/expm1

cfunction

(math/expm1 x)

Returns e to the power of x minus 1.

math/floor

cfunction

(math/floor x)

Returns the largest integer value number that is not greater than x.

math/gamma

cfunction

(math/gamma x)

Returns gamma(x).

math/hypot

cfunction

(math/hypot a b)

Returns the c from the equation $c^2 = a^2 + b^2$

math/inf

number

The number representing positive infinity

math/log

cfunction

(math/log x)

Returns log base natural number of x.

math/log10

cfunction

(math/log10 x)

Returns log base 10 of x.

math/log1p

cfunction

(math/log1p x)

Returns ($\log_e x + 1$) more accurately than (+ (math/log x) 1)

math/log2

cfunction

(math/log2 x)

Returns log base 2 of x.

math/next

cfunction

(math/next x y)

Returns the next representable floating point value after x in the direction of y.

math/pi

number

The value pi.

math/pow

cfunction

(math/pow a x)

Return a to the power of x.

math/random

cfunction

(math/random)

Returns a uniformly distributed random number between 0 and 1.

math/rng

cfunction

(math/rng &opt seed)

Creates a Psuedo-Random number generator, with an optional seed. The seed should be an unsigned 32 bit integer or a buffer. Do not use this for cryptography. Returns a core/rng abstract type.

math/rng-buffer

cfunction

(math/rng-buffer rng n &opt buf)

Get n random bytes and put them in a buffer. Creates a new buffer if no buffer is provided, otherwise appends to the given buffer. Returns the buffer.

math/rng-int

cfunction

(math/rng-int rng &opt max)

Extract a random random integer in the range [0, max] from the RNG. If no max is given, the default is $2^{31} - 1$.

math/rng-uniform

cfunction

(math/rng-seed rng seed)

Extract a random number in the range [0, 1) from the RNG.

math/round

cfunction

(math/round x)

Returns the integer nearest to x.

math/seedrandom

cfunction

(math/seedrandom seed)

Set the seed for the random number generator. seed should be an integer or a buffer.

math/sin

cfunction

(math/sin x)

Returns the sine of x.

math/sinh

cfunction

(math/sinh x)

Return the hyperbolic sine of x.

math/sqrt

cfunction

(math/sqrt x)

Returns the square root of x.

math/tan

~function~

cfunction

(math/tan x)

Returns the tangent of x.

math/tanh

cfunction

(math/tanh x)

Return the hyperbolic tangent of x.

math/trunc

cfunction

(math/trunc x)

Returns the integer between x and 0 nearest to x.

max

function

[source](#)

(max & args)

Returns the numeric maximum of the arguments.

mean

function

[source](#)

(mean xs)

Returns the mean of xs. If empty, returns NaN.

merge

function

[source](#)

(merge & colls)

Merges multiple tables/structs to one. If a key appears in more than one collection, then later values replace any previous ones. Returns a new table.

merge-into

function

[source](#)

(merge-into tab & colls)

Merges multiple tables/structs into a table. If a key appears in more than one collection, then later values replace any previous ones.
Returns the original table.

min

function

[source](#)

(min & args)

Returns the numeric minimum of the arguments.

mod

function

(mod dividend divisor)

Returns the modulo of dividend / divisor.

module/add-paths

function

[source](#)

(module/add-paths ext loader)

Add paths to module/paths for a given loader such that the generated paths behave like other module types, including relative imports and syspath imports. ext is the file extension to associate with this module type, including the dot. loader is the keyword name of a loader that is module/loaders. Returns the modified module/paths.

module/cache

table

[source](#)

Table mapping loaded module identifiers to their environments.

module/expand-path

cfunction

(module/expand-path path template)

Expands a path template as found in module/paths for module/find.

This takes in a path (the argument to require) and a template string, template, to expand the path to a path that can be used for importing files. The replacements are as follows:

- :all: the value of path verbatim
 - :cur: the current file, or (dyn :current-file)
 - :dir: the directory containing the current file
 - :name: the name component of path, with extension if given
 - :native: the extension used to load natives, .so or .dll
 - :sys: the system path, or (dyn :syspath)
-

module/find

function

[source](#)

(module/find path)

Try to match a module or path name from the patterns in module/paths. Returns a tuple (fullpath kind) where the kind is one of :source, :native, or image if the module is found, otherwise a tuple with nil followed by an error message.

module/loaders

table

[source](#)

A table of loading method names to loading functions. This table lets require and import load many different kinds of files as module.

module/loading

table

native

[source](#)

Table mapping currently loading modules to true. Used to prevent circular dependencies.

module/paths

array

[source](#)

The list of paths to look for modules, templated for module/expand-path. Each element is a two element tuple, containing the path template and a keyword :source, :native, or :image indicating how require should load files found at these paths.

A tuple can also contain a third element, specifying a filter that prevents module/find from searching that path template if the filter doesn't match the input path. The filter can be a string or a predicate function, and is often a file extension, including the period.

nan?

function

[source](#)

(nan? x)

Check if x is NaN

nat?

cfunction

(nat? x)

Check if x can be exactly represented as a non-negative 32 bit signed two's complement integer.

native

cfunction

(native path &opt env)

Load a native module from the given path. The path must be an absolute or relative path on the file system, and is usually a .so file on Unix systems, and a .dll file on Windows. Returns an environment table that contains functions and other values from the native module.

neg?

function

[source](#)

(neg? x)

Check if x is less than 0.

net/chunk

cfunction

(net/chunk stream nbytes &opt buf)

Same a net/read, but will wait for all n bytes to arrive rather than return early.

net/close

cfunction

(net/close stream)

Close a stream so that no further communication can occur.

net/connect

cfunction

(net/connect host port)

Open a connection to communicate with a server. Returns a duplex stream that can be used to communicate with the server.

net/read

cfunction

(net/read stream nbytes &opt buf)

Read up to n bytes from a stream, suspending the current fiber until the bytes are available. If less than n bytes are available (and more than 0), will push those bytes and return early. Returns a buffer with up to n more bytes in it.

net/server

cfunction

(net/server host port handler)

Start a TCP server. handler is a function that will be called with a *stream on each connection to the server. Returns a new stream that is*

~~stream on each connection to the server. Returns a new stream that is neither readable nor writeable.~~

net/write

function

(net/write stream data)

Write data to a stream, suspending the current fiber until the write completes. Returns stream.

next

function

(next ds &opt key)

Gets the next key in a data structure. Can be used to iterate through the keys of a data structure in an unspecified order. Keys are guaranteed to be seen only once per iteration if the data structure is not mutated during iteration. If key is nil, next returns the first key. If next returns nil, there are no more keys to iterate through.

nil?

function

[source](#)

(nil? x)

Check if x is nil.

not

function

(not x)

Returns the boolean inverse of x.

not=

function

(not= & xs)

Check if any values in xs are not equal. Returns a boolean.

number?

function

[source](#)

(number? x)

Check if x is a number.

odd?

function

[source](#)

(odd? x)

Check if x is odd.

one?

function

[source](#)

(one? x)

Check if x is equal to 1.

or

macro

[source](#)

(or & forms)

Evaluates to the last argument if all preceding elements are falsey, otherwise evaluates to the first truthy element.

os/arch

cfunction

(os/arch)

Check the ISA that janet was compiled for. Returns one of:

- :x86
- :x86-64
- :arm
- :aarch64
- :sparc
- :wasm
- :unknown

os/cd

cfunction

(os/cd path)

Change current directory to path. Returns nil on success, errors on failure.

os/chmod

cfunction

(os/chmod path mode)

Change file permissions, where mode is a permission string as returned by os/perm-string, or an integer as returned by os/perm-int. When mode is an integer, it is interpreted as a Unix permission value, best specified in octal, like 8r666 or 8r400. Windows will not differentiate between user, group, and other permissions, and thus will combine all of these permissions. Returns nil.

os/clock

cfunction

(os/clock)

Return the number of seconds since some fixed point in time. The clock is guaranteed to be non decreasing in real time.

os/cryptorand

cfunction

/os/cryptorand n <int> buf

`(os/cryptos rand n &opt buf)`

Get or append n bytes of good quality random data provided by the OS. Returns a new buffer or buf.

os/cwd

cfunction

`(os/cwd)`

Returns the current working directory.

os/date

cfunction

`(os/date &opt time local)`

Returns the given time as a date struct, or the current time if no time is given. Returns a struct with following key values. Note that all numbers are 0-indexed. Date is given in UTC unless local is truthy, in which case the date is formatted for the local timezone.

- `:seconds` - number of seconds [0-61]
- `:minutes` - number of minutes [0-59]
- `:hours` - number of hours [0-23]
- `:month-day` - day of month [0-30]
- `:month` - month of year [0, 11]
- `:year` - years since year 0 (e.g. 2019)
- `:week-day` - day of the week [0-6]
- `:year-day` - day of the year [0-365]
- `:dst` - If Day Light Savings is in effect

os/dir

cfunction

(os/dir dir &opt array)

Iterate over files and subdirectories in a directory. Returns an array of paths parts, with only the file name or directory name and no prefix.

os/environ

cfunction

(os/environ)

Get a copy of the os environment table.

os/execute

cfunction

(os/execute args &opts flags env)

Execute a program on the system and pass it string arguments. Flags is a keyword that modifies how the program will execute.

:e - enables passing an environment to the program. Without :e, the current environment is inherited.

:p - allows searching the current PATH for the binary to execute. Without this flag, binaries must use absolute paths.

env is a table or struct mapping environment variables to values. Returns the exit status of the program.

os/exit

cfunction

(os/exit &opt x)

Exit from janet with an exit code equal to x. If x is not an integer, the exit with status equal the hash of x.

os/getenv

cfunction

(os/getenv variable &opt dflt)

Get the string value of an environment variable.

os/link

cfunction

(os/link oldpath newpath &opt symlink)

Create a link at newpath that points to oldpath and returns nil. Iff symlink is truthy, creates a symlink. Iff symlink is falsey or not provided, creates a hard link. Does not work on Windows.

os/lstat

cfunction

(os/lstat path &opt tab|key)

Like os/stat, but don't follow symlinks.

os/mkdir

cfunction

(os/mkdir path)

Create a new directory. The path will be relative to the current directory if relative, otherwise it will be an absolute path. Returns true if the directory was created, false if the directory already exists, and errors otherwise.

os/mktime

cfunction

(os/mktime date-struct &opt local)

Get the broken down date-struct time expressed as the number of seconds since January 1, 1970, the Unix epoch. Returns a real number. Date is given in UTC unless local is truthy, in which case the date is computed for the local timezone.

Inverse function to os/date.

os/perm-int

cfunction

(os/perm-int bytes)

Parse a 9 character permission string and return an integer that can be used by chmod.

os/perm-string

cfunction

(os/perm-string int)

Convert a Unix octal permission value from a permission integer as returned by os/stat to a human readable string, that follows the formatting of unix tools like ls. Returns the string as a 9 character string of r, w, x and - characters. Does not include the file/directory/symlink character as rendered by `ls`.

os/readlink

cfunction

(os/readlink path)

Read the contents of a symbolic link. Does not work on Windows.

os/realpath

cfunction

(os/realpath path)

Get the absolute path for a given path, following ../, ./, and symlinks. Returns an absolute path as a string. Will raise an error on Windows.

os/rename

cfunction

(os/rename oldname newname)

Renames a file or dirk to a new path. Returns nil

`RENAME A FILE OR DISK TO A NEW PATH. RETURNS NIL.`

os/rm

cfunction

`(os/rm path)`

Delete a file. Returns nil.

os/rmdir

cfunction

`(os/rmdir path)`

Delete a directory. The directory must be empty to succeed.

os/setenv

cfunction

`(os/setenv variable value)`

Set an environment variable.

os/shell

cfunction

`(os/shell str)`

Pass a command string str directly to the system shell.

os/sleep

cfunction

(os/sleep nsec)

Suspend the program for nsec seconds. 'nsec' can be a real number.

Returns nil.

os/stat

cfunction

(os/stat path &opt tab|key)

Gets information about a file or directory. Returns a table If the third argument is a keyword, returns only that information from stat. If the file or directory does not exist, returns nil. The keys are

- :dev - the device that the file is on
- :mode - the type of file, one of :file, :directory, :block, :character, :fifo, :socket, :link, or :other
- :int-permissions - A Unix permission integer like 8r744
- :permissions - A Unix permission string like "rwxr--r--"
- :uid - File uid
- :gid - File gid
- :nlink - number of links to file
- :rdev - Real device of file. 0 on windows.
- :size - size of file in bytes
- :blocks - number of blocks in file. 0 on windows
- :blocksize - size of blocks in file. 0 on windows
- :accessed - timestamp when file last accessed
- :changed - timestamp when file last changed (permissions changed)

modified timestamp when file last modified / content changed

`.modified` - timestamp when file last modified (content changed)

os/symlink

cfunction

(os/symlink oldpath newpath)

Create a symlink from oldpath to newpath, returning nil. Same as (os/link oldpath newpath true).

os/time

cfunction

(os/time)

Get the current time expressed as the number of seconds since January 1, 1970, the Unix epoch. Returns a real number.

os/touch

cfunction

(os/touch path &opt actime modtime)

Update the access time and modification times for a file. By default, sets times to the current time.

os/umask

cfunction

(os/umask mask)

Set a new umask, returns the old umask.

os/which

cfunction

(os/which)

Check the current operating system. Returns one of:

- :windows
- :macos
- :web - Web assembly (emscripten)
- :linux
- :freebsd
- :openbsd
- :netbsd
- :posix - A POSIX compatible system (default)

May also return a custom keyword specified at build time.

pairs

function

[source](#)

(pairs x)

Get the values of an associative data structure.

parse

function

[source](#)

(parse str)

Parse a string and return the first value. For complex parsing, such as for a repl with error handling, use the parser api.

parser/byte

cfunction

(parser/byte parser b)

Input a single byte into the parser byte stream. Returns the parser.

parser/clone

cfunction

(parser/clone p)

Creates a deep clone of a parser that is identical to the input parser. This cloned parser can be used to continue parsing from a good checkpoint if parsing later fails. Returns a new parser.

parser/consume

cfunction

(parser/consume parser bytes &opt index)

Input bytes into the parser and parse them. Will not throw errors if there is a parse error. Starts at the byte index given by index. Returns the number of bytes read.

parser/eof

cfunction

(parser/eof parser)

Indicate that the end of file was reached to the parser. This puts the parser in the :dead state.

parser/error

cfunction

(parser/error parser)

If the parser is in the error state, returns the message associated with that error. Otherwise, returns nil. Also flushes the parser state and parser queue, so be sure to handle everything in the queue before calling parser/error.

parser/flush

cfunction

(parser/flush parser)

Clears the parser state and parse queue. Can be used to reset the parser if an error was encountered. Does not reset the line and column counter, so to begin parsing in a new context, create a new parser.

parser/has-more

cfunction

,

"

\

(parser/nas-more parser)

Check if the parser has more values in the value queue.

parser/insert

cfunction

(parser/insert parser value)

Insert a value into the parser. This means that the parser state can be manipulated in between chunks of bytes. This would allow a user to add extra elements to arrays and tuples, for example. Returns the parser.

parser/new

cfunction

(parser/new)

Creates and returns a new parser object. Parsers are state machines that can receive bytes, and generate a stream of values.

parser/produce

cfunction

(parser/produce parser)

Dequeue the next value in the parse queue. Will return nil if no parsed values are in the queue, otherwise will dequeue the next value.

parser/state

cfunction

(parser/state parser &opt key)

Returns a representation of the internal state of the parser. If a key is passed, only that information about the state is returned. Allowed keys are:

:delimiters - Each byte in the string represents a nested data structure. For example, if the parser state is '(["', then the parser is in the middle of parsing a string inside of square brackets inside parentheses. Can be used to augment a REPL prompt.

:frames - Each table in the array represents a 'frame' in the parser state. Frames contain information about the start of the expression being parsed as well as the type of that expression and some type-specific information.

parser/status

cfunction

(parser/status parser)

Gets the current status of the parser state machine. The status will be one of:

:pending - a value is being parsed.
:error - a parsing error was encountered.
:root - the parser can either read more values or safely terminate.

parser/where

cfunction

(parser/where parser)

Returns the current line number and column of the parser's internal state.

partial

function

[source](#)

(partial f & more)

Partial function application.

partition

function

[source](#)

(partition n ind)

Partition an indexed data structure into tuples of size n. Returns a new array.

peg/compile

cfunction

(peg/compile peg)

Compiles a peg source data structure into a <core/peg>. This will speed up matching if the same peg will be used multiple times.

peg/match

function

(peg/match peg text &opt start & args)

Match a Parsing Expression Grammar to a byte string and return an array of captured values. Returns nil if text does not match the language defined by peg. The syntax of PEGs is documented on the Janet website.

pos?

function

[source](#)

(pos? x)

Check if x is greater than 0.

postwalk

function

[source](#)

(postwalk f form)

Do a post-order traversal of a data structure and call (f x) on every visitation.

pp

function

[source](#)

(pp x)

Pretty print to stdout or (dyn :out). The format string used is (dyn :pretty-format "%q").

prewalk

function

[source](#)

(prewalk f form)

Similar to postwalk, but do pre-order traversal.

prin

cfunction

(prin & xs)

Same as print, but does not add trailing newline.

printf

cfunction

(printf fmt & xs)

Like printf but with no trailing newline.

print

cfunction

(print & xs)

Print values to the console (standard out). Value are converted to strings if they are not already. After printing all values, a newline character is printed. Use the value of (dyn :out stdout) to determine what to push characters to. Expects (dyn :out stdout) to be either a core/file or a buffer. Returns nil.

printf

cfunction

(printf fmt & xs)

Prints output formatted as if with (string/format fmt ;xs) to (dyn :out stdout) with a trailing newline.

product

function

[source](#)

(product xs)

Returns the product of xs. If xs is empty, returns 1.

EXAMPLES

```
(product []) # -> 1
(product @[1 2 3]) # -> 6
(product [0 1 2 3]) # -> 0
(product (range 1 10)) # -> 362880

# Product over byte values [0-255] in a string
(product "hello") # -> 1.35996e+10
```

```
# Product over values in a table or struct  
(sum { :a 1 :b 2 :c 4}) # -> 8
```

prompt

macro

[source](#)

(prompt tag & body)

Set up a checkpoint that can be returned to. Tag should be a value that is used in a return statement, like a keyword.

propagate

function

(propagate x fiber)

Propagate a signal from a fiber to the current fiber. The resulting stack trace from the current fiber will include frames from fiber. If fiber is in a state that can be resumed, resuming the current fiber will first resume fiber. This function can be used to re-raise an error without losing the original stack trace.

protect

macro

[source](#)

(protect & body)

Evaluate expressions, while capturing any errors. Evaluates to a tuple of two elements. The first element is true if successful, false if an error,

and the second is the return value or error.

put

function

(put ds key value)

Associate a key with a value in any mutable associative data structure. Indexed data structures (arrays and buffers) only accept non-negative integer keys, and will expand if an out of bounds value is provided. In an array, extra space will be filled with nils, and in a buffer, extra space will be filled with 0 bytes. In a table, putting a key that is contained in the table prototype will hide the association defined by the prototype, but will not mutate the prototype table. Putting a value nil into a table will remove the key from the table. Returns the data structure ds.

put-in

function

[source](#)

(put-in ds ks v)

Put a value into a nested data structure. Looks into the data structure via a sequence of keys. Missing data structures will be replaced with tables. Returns the modified, original data structure.

quit

function

[source](#)

(quit &opt value)

Tries to exit from the current repl or context. Does not always exit the application. Works by setting the :exit dynamic binding to true. Passing a non-nil value here will cause the outer run-context to return that value.

range

function

[source](#)

(range & args)

Create an array of values [start, end) with a given step. With one argument returns a range [0, end). With two arguments, returns a range [start, end). With three, returns a range with optional step size.

EXAMPLES

```
(range 10) # -> @[0 1 2 3 4 6 7 8 9]
(range 5 10) # -> @[5 6 7 8 9]
(range 5 10 2) # -> @[5 7 9]
(range 5 11 2) # -> @[5 7 9]
(range 10 0 -1) # -> @[10 9 8 7 6 5 4 3 2 1]
```

reduce

function

[source](#)

(reduce f init ind)

Reduce, also known as fold-left in many languages, transforms an indexed type (array, tuple) with a function to produce a value.

reduce2

function

[source](#)

(reduce2 f ind)

The 2 argument version of reduce that does not take an initialization value. Instead the first element of the array is used for initialization.

repl

function

[source](#)

(repl &opt chunks onsignal env)

Run a repl. The first parameter is an optional function to call to get a chunk of source code that should return nil for end of file. The second parameter is a function that is called when a signal is caught. One can provide an optional environment table to run the repl in.

require

function

[source](#)

(require path & args)

Require a module with the given name. Will search all of the paths in module/paths. Returns the new environment returned from compiling and running the file.

resume

function

(resume fiber &opt x)

Resume a new or suspended fiber and optionally pass in a value to the fiber that will be returned to the last yield in the case of a pending fiber, or the argument to the dispatch function in the case of a new fiber.

Returns either the return result of the fiber's dispatch function, or the value from the next yield call in fiber.

return

function

[source](#)

(return to &opt value)

Return to a prompt point.

reverse

function

[source](#)

(reverse t)

Reverses the order of the elements in a given array or tuple and returns a new array.

EXAMPLES

```
(reverse [1 2 3]) # -> @[3 2 1]
(reverse "abcdef") # -> @[102 101 100 99 98 97]
```

```
(reverse :abc) # -> @[99 98 97]
```

root-env

table

[source](#)

The root environment used to create environments with (make-env)

run-context

function

[source](#)

(run-context opts)

Run a context. This evaluates expressions in an environment, and is encapsulates the parsing, compilation, and evaluation. Returns (in environment :exit-value environment) when complete. opts is a table or struct of options. The options are as follows:

:chunks - callback to read into a buffer - default is getline

:on-parse-error - callback when parsing fails - default is bad-parse

:env - the environment to compile against - default is the current env

:source - string path of source for better errors - default is "<anonymous>"

:on-compile-error - callback when compilation fails - default is bad-compile

:evaluator - callback that executes thunks. Signature is (evaluator thunk source env where)

:on-status - callback when a value is evaluated - default is

.on-status - callback when a value is evaluated - default is debug/stacktrace

:fiber-flags - what flags to wrap the compilation fiber with. Default is :ia.

:expander - an optional function that is called on each top level form before being compiled.

scan-number

cfunction

(scan-number str)

Parse a number from a byte sequence and return that number, either an integer or a real. The number must be in the same format as numbers in janet source code. Will return nil on an invalid number.

seq

macro

[source](#)

(seq head & body)

Similar to loop, but accumulates the loop body into an array and returns that. See loop for details.

EXAMPLES

```
(seq [x :range [0 5]] (* 2 x)) # -> @[0 2 4 6  
8]
```

setdyn

cfunction

(setdyn key value)

Set a dynamic binding. Returns value.

short-fn

macro

[source](#)

(short-fn arg)

fn shorthand.

usage:

(short-fn (+ \$ \$)) - A function that double's its arguments.

(short-fn (string \$0 \$1)) - accepting multiple args

|(+ \$ \$) - use pipe reader macro for terse function literals

|(+ \$&) - variadic functions

signal

cfunction

(signal what x)

Raise a signal with payload x.

slice

cfunction

(slice x &opt start end)

Extract a sub-range of an indexed data structure or byte sequence.

EXAMPLES

```
(slice @[1 2 3]) # -> (1 2 3) (a new array!)
(slice @[:a :b :c] 1) # -> (:b :c)
(slice [:a :b :c :d :e] 2 4) # -> (:c :d)
(slice [:a :b :d :d :e] 2 -1) # -> (:c :d :e)
(slice [:a :b :d :d :e] 2 -2) # -> (:c :d)
(slice [:a :b :d :d :e] 2 -4) # -> ()
(slice [:a :b :d :d :e] 2 -10) # -> error:
range error
(slice "abcdefg" 0 2) # -> "ab"
(slice @"abcdefg" 0 2) # -> "ab"
```

slurp

function

[source](#)

(slurp path)

Read all data from a file with name path and then close the file.

some

function

[source](#)

(some pred xs)

Returns nil if all xs are false or nil. otherwise returns the result of the

first truthy predicate, (pred x).

sort

function

[source](#)

(sort a &opt by)

Sort an array in-place. Uses quick-sort and is not a stable sort.

sort-by

function

[source](#)

(sort-by f ind)

Returns a new sorted array that compares elements by invoking a function on each element and comparing the result with <.

sorted

function

[source](#)

(sorted ind &opt by)

Returns a new sorted array without modifying the old one.

sorted-by

function

[source](#)

(sorted-by f ind)

Returns a new sorted array that compares elements by invoking a function on each element and comparing the result with <.

split

function

[source](#)

(split path contents &opt mode)

Write contents to a file at path. Can optionally append to the file.

stderr

core/file

The standard error file.

stdin

core/file

The standard input file.

stdout

core/file

The standard output file.

string

function

cfunction

(string & parts)

Creates a string by concatenating values together. Values are converted to bytes via describe if they are not byte sequences. Returns the new string.

string/ascii-lower

cfunction

(string/ascii-lower str)

Returns a new string where all bytes are replaced with the lowercase version of themselves in ASCII. Does only a very simple case check, meaning no unicode support.

string/ascii-upper

cfunction

(string/ascii-upper str)

Returns a new string where all bytes are replaced with the uppercase version of themselves in ASCII. Does only a very simple case check, meaning no unicode support.

string/bytes

cfunction

(string/bytes str)

Returns an array of integers that are the byte values of the string.

string/check-set

cfunction

(string/check-set set str)

Checks that the string str only contains bytes that appear in the string set. Returns true if all bytes in str appear in set, false if some bytes in str do not appear in set.

string/find

cfunction

(string/find patt str)

Searches for the first instance of pattern patt in string str. Returns the index of the first character in patt if found, otherwise returns nil.

string/find-all

cfunction

(string/find-all patt str)

Searches for all instances of pattern patt in string str. Returns an array of all indices of found patterns. Overlapping instances of the pattern are not counted, meaning a byte in string will only contribute to finding at most one occurrence of pattern. If no occurrences are found, will return an empty array.

string/format

cfunction

(string/format format & values)

Similar to sprintf, but specialized for operating with Janet values.

Returns a new string.

string/from-bytes

cfunction

(string/from-bytes & byte-vals)

Creates a string from integer parameters with byte values. All integers will be coerced to the range of 1 byte 0-255.

string/has-prefix?

cfunction

(string/has-prefix? pfx str)

Tests whether str starts with pfx.

string/has-suffix?

cfunction

(string/has-suffix? sfx str)

Tests whether str ends with sfx.

string/join

cfunction

(string/join parts &opt sep)

Joins an array of strings into one string, optionally separated by a separator string sep.

string/repeat

cfunction

(string/repeat bytes n)

Returns a string that is n copies of bytes concatenated.

string/replace

cfunction

(string/replace patt subst str)

Replace the first occurrence of patt with subst in the string str. Will return the new string if patt is found, otherwise returns str.

string/replace-all

cfunction

(string/replace-all patt subst str)

Replace all instances of patt with subst in the string str. Will return the new string if patt is found, otherwise returns str.

string/reverse

cfunction

(string/reverse str)

Returns a string that is the reversed version of str.

string/slice

cfunction

(string/slice bytes &opt start end)

Returns a substring from a byte sequence. The substring is from index start inclusive to index end exclusive. All indexing is from 0. 'start' and 'end' can also be negative to indicate indexing from the end of the string. Note that index -1 is synonymous with index (length bytes) to allow a full negative slice range.

string/split

cfunction

(string/split delim str &opt start limit)

Splits a string str with delimiter delim and returns an array of substrings. The substrings will not contain the delimiter delim. If delim is not found, the returned array will have one element. Will start searching for delim at the index start (if provided), and return up to a maximum of limit results (if provided).

string/trim

cfunction

(string/trim str &opt set)

Trim leading and trailing whitespace from a byte sequence. If the argument set is provided, consider only characters in set to be whitespace.

string/triml

cfunction

(string/triml str &opt set)

Trim leading whitespace from a byte sequence. If the argument set is provided, consider only characters in set to be whitespace.

string/trimr

cfunction

(string/trimr str &opt set)

Trim trailing whitespace from a byte sequence. If the argument set is provided, consider only characters in set to be whitespace.

string?

function

[source](#)

(string? x)

Check if x is a string.

struct

cfunction

(struct & kvs)

Create a new struct from a sequence of key value pairs. kvs is a sequence k1, v1, k2, v2, k3, v3, ... If kvs has an odd number of elements, an error will be thrown. Returns the new struct.

struct?

function

[source](#)

(struct? x)

Check if x a struct.

sum

function

[source](#)

(sum xs)

Returns the sum of xs. If xs is empty, returns 0.

EXAMPLES

```
(sum []) # -> 0
(sum @[1]) # -> 1
(sum (range 100)) # -> 4950

# Sum over bytes values [0-255] in a string
(sum "hello") # -> 532
```

```
# Sum over values in a table or struct  
(sum {:a 1 :b 2 :c 4}) # -> 7
```

symbol

cfunction

(symbol & xs)

Creates a symbol by concatenating values together. Values are converted to bytes via describe if they are not byte sequences. Returns the new symbol.

symbol?

function

[source](#)

(symbol? x)

Check if x is a symbol.

table

cfunction

(table & kvs)

Creates a new table from a variadic number of keys and values. kvs is a sequence k1, v1, k2, v2, k3, v3, ... If kvs has an odd number of elements, an error will be thrown. Returns the new table.

table/clone

cfunction

(table/clone tab)

Create a copy of a table. Updates to the new table will not change the old table, and vice versa.

table/getproto

cfunction

(table/getproto tab)

Get the prototype table of a table. Returns nil if a table has no prototype, otherwise returns the prototype.

table/new

cfunction

(table/new capacity)

Creates a new empty table with pre-allocated memory for capacity entries. This means that if one knows the number of entries going to go in a table on creation, extra memory allocation can be avoided.

Returns the new table.

table/rawget

cfunction

(table/rawget tab key)

Gets a value from a table without looking at the prototype table. If a

~~Gets a value from a table without looking at the prototype table. If a table tab does not contain t directly, the function will return nil without checking the prototype. Returns the value in the table.~~

table/setproto

cfunction

(table/setproto tab proto)

Set the prototype of a table. Returns the original table tab.

table/to-struct

cfunction

(table/to-struct tab)

Convert a table to a struct. Returns a new struct. This function does not take into account prototype tables.

table?

function

[source](#)

(table? x)

Check if x a table.

take

function

[source](#)

...

(take n ind)

Take first n elements in an indexed type. Returns new indexed instance.

take-until

function

[source](#)

(take-until pred ind)

Same as (take-while (complement pred) ind).

take-while

function

[source](#)

(take-while pred ind)

Given a predicate, take only elements from an indexed type that satisfy the predicate, and abort on first failure. Returns a new array.

tarray/buffer

cfunction

(tarray/buffer array|size)

Return typed array buffer or create a new buffer.

tarray/copy-bytes

cfunction

Digitized by srujanika@gmail.com

(tarray/copy-bytes src sindex dst dindex &opt count)

Copy count elements (default 1) of src array from index sindex to dst array at position dindex memory can overlap.

tarray/length

cfunction

(tarray/length array|buffer)

Return typed array or buffer size.

tarray/new

cfunction

(tarray/new type size &opt stride offset tarray|buffer)

Create new typed array.

tarray/properties

cfunction

(tarray/properties array)

Return typed array properties as a struct.

tarray/slice

cfunction

(tarray/slice tarr &opt start end)

Takes a slice of a typed array from start to end. The range is half open, [start, end). Indexes can also be negative, indicating indexing from the end of the end of the typed array. By default, start is 0 and end is the size of the typed array. Returns a new janet array.

tarray/swap-bytes

cfunction

(tarray/swap-bytes src sindex dst dindex &opt count)

Swap count elements (default 1) between src array from index sindex and dst array at position dindex memory can overlap.

thread/close

cfunction

(thread/close thread)

Close a thread, unblocking it and ending communication with it. Note that closing a thread is idempotent and does not cancel the thread's operation. Returns nil.

thread/current

cfunction

(thread/current)

Get the current running thread.

thread/new

cfunction

(thread/new func &opt capacity)

Start a new thread that will start immediately. If capacity is provided, that is how many messages can be stored in the thread's mailbox before blocking senders. The capacity must be between 1 and 65535 inclusive, and defaults to 10. Returns a handle to the new thread.

thread/receive

cfunction

(thread/receive &opt timeout)

Get a message sent to this thread. If timeout is provided, an error will be thrown after the timeout has elapsed but no messages are received.

thread/send

cfunction

(thread/send thread msg)

Send a message to the thread. This will never block and returns thread immediately. Will throw an error if there is a problem sending the message.

trace

cfunction

(trace func)

Enable tracing on a function. Returns the function.

true?

function

[source](#)

(true? x)

Check if x is true.

truthy?

function

[source](#)

(truthy? x)

Check if x is truthy.

try

macro

[source](#)

(try body catch)

Try something and catch errors. Body is any expression, and catch should be a form with the first element a tuple. This tuple should contain a binding for errors and an optional binding for the fiber wrapping the body. Returns the result of body if no error, or the result of catch if an error.

tuple

cfunction

(tuple & items)

Creates a new tuple that contains items. Returns the new tuple.

tuple/brackets

cfunction

Set the sourcemap metadata on a tuple. line and column indicate should be integers.

tuple/slice

cfunction

(tuple/slice arrtup [,start=0 [,end=(length arrtup)]])

Take a sub sequence of an array or tuple from index start inclusive to index end exclusive. If start or end are not provided, they default to 0 and the length of arrtuple respectively. 'start' and 'end' can also be

negative to indicate indexing from the end of the input. Note that index -1 is synonymous with index '(length arrtuple)' to allow a full negative slice range. Returns the new tuple.

tuple/sourcemap

cfunction

(tuple/sourcemap tup)

Returns the sourcemap metadata attached to a tuple, which is another tuple (line, column).

tuple/type

cfunction

(tuple/type tup)

Checks how the tuple was constructed. Will return the keyword :brackets if the tuple was parsed with brackets, and :parens otherwise. The two types of tuples will behave the same most of the time, but will print differently and be treated differently by the compiler.

tuple?

function

[source](#)

(tuple? x)

Check if x is a tuple.

type

cfunction

(type x)

Returns the type of x as a keyword. x is one of

- :nil
- :boolean
- :number
- :array
- :tuple
- :table
- :struct
- :string
- :buffer
- :symbol
- :keyword
- :function
- :cfunction

or another keyword for an abstract type.

EXAMPLES

```
(type nil) # -> :nil
(type true) # -> :boolean
(type false) # -> :boolean
(type 1) # -> :number
(type :key) # -> :keyword
(type (int/s64 "100")) # -> :core/s64
```

unless

macro
[source](#)

(unless condition & body)

Shorthand for (when (not condition) ;body).

unmarshal

cfunction

(unmarshal buffer &opt lookup)

Unmarshal a value from a buffer. An optional lookup table can be provided to allow for aliases to be resolved. Returns the value unmarshalled from the buffer.

untrace

cfunction

(untrace func)

Disables tracing on a function. Returns the function.

update

function
[source](#)

(update ds key func & args)

Accepts a key argument and passes its associated value to a function. The key is the re-associated to the function's return value. Returns the updated data structure ds.

update-in

function

[source](#)

(update-in ds ks f & args)

Update a value in a nested data structure by applying f to the current value. Looks into the data structure via a sequence of keys. Missing data structures will be replaced with tables. Returns the modified, original data structure.

use

macro

[source](#)

(use & modules)

Similar to import, but imported bindings are not prefixed with a namespace identifier. Can also import multiple modules in one shot.

values

function

[source](#)

(values x)

Get the values of an associative data structure.

var-

~~~

macro

[source](#)

(var- name & more)

Define a private var that will not be exported.

---

**varfn**

macro

[source](#)

(varfn name & body)

Create a function that can be rebound. varfn has the same signature as defn, but defines functions in the environment as vars. If a var 'name' already exists in the environment, it is rebound to the new function. Returns a function.

---

**varglobal**

function

[source](#)

(varglobal name init)

Dynamically create a global var.

---

**walk**

function

[source](#)

(walk f form)

Iterate over the values in ast and apply f to them. Collect the results in

a data structure. If ast is not a table, struct, array, or tuple, returns form.

---

## **when**

macro

[source](#)

(when condition & body)

Evaluates the body when the condition is true. Otherwise returns nil.

---

## **when-let**

macro

[source](#)

(when-let bindings & body)

Same as (if-let bindings (do ;body)).

---

## **when-with**

macro

[source](#)

(when-with [binding ctor dtor] & body)

Similar to with, but if binding is false or nil, returns nil without evaluating the body. Otherwise, the same as with.

---

## **with**

macro

[source](#)

(with [binding ctor dtor] & body)

Evaluate body with some resource, which will be automatically cleaned up if there is an error in body. binding is bound to the expression ctor, and dtor is a function or callable that is passed the binding. If no destructor (dtor) is given, will call :close on the resource.

## EXAMPLES

```
# Print all of poetry.txt, and close the file
when done,
# even when there is an error.
(with [f (file/open "poetry.txt")]
  (print (:read f :all)))
```

---

## with-dyns

macro

[source](#)

(with-dyns bindings & body)

Run a block of code in a new fiber that has some dynamic bindings set. The fiber will not mask errors or signals, but the dynamic bindings will be properly unset, as dynamic bindings are fiber local.

---

## with-syms

macro

[source](#)

(with-syms syms & body)

Evaluates body with each symbol in syms bound to a generated, unique symbol.

---

## **with-vars**

macro

[source](#)

(with-vars vars & body)

Evaluates body with each var in vars temporarily bound. Similar signature to let, but each binding must be a var.

---

## **yield**

function

(yield &opt x)

Yield a value to a parent fiber. When a fiber yields, its execution is paused until another thread resumes it. The fiber will then resume, and the last yield call will return the value that was passed to resume.

---

## **zero?**

function

[source](#)

(zero? x)

Check if x is zero.

---

## **zipcoll**

function

[source](#)

(zipcoll ks vs)

Creates a table from two arrays/tuples. Returns a new table.

[< jpm](#)

[Array Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Array Module

[< Core API](#)

[Buffer Module >](#)

Arrays are the mutable indexed type in Janet. They contain a sequence of values that are keyed with an index. They can also be used to implement stacks and queues. See the [Arrays](#) documentation for more info.

# Index

[array\(concat\)](#) [array\(ensure\)](#) [array\(fill\)](#) [array\(insert\)](#) [array\(new\)](#) [array\(new-filled\)](#)  
[array\(peek\)](#) [array\(pop\)](#) [array\(push\)](#) [array\(remove\)](#) [array\(slice\)](#)

---

## **array(concat)**

cfunction

(array(concat arr & parts))

Concatenates a variadic number of arrays (and tuples) into the first argument which must an array. If any of the parts are arrays or tuples, their elements will be inserted into the array. Otherwise, each part in parts will be appended to arr in order. Return the modified array arr.

---

## **array(ensure)**

cfunction

(array(ensure arr capacity growth))

Ensures that the memory backing the array is large enough for capacity items at the given rate of growth. Capacity and growth must be integers. If the backing capacity is already enough, then this function does nothing. Otherwise, the backing memory will be reallocated so that there is enough space.

---

## **array(fill)**

cfunction

(array/fill arr &opt value)

Replace all elements of an array with value (defaulting to nil) without changing the length of the array. Returns the modified array.

---

### array/insert

cfunction

(array/insert arr at & xs)

Insert all of xs into array arr at index at. at should be an integer 0 and the length of the array. A negative value for at will index from the end of the array, such that inserting at -1 appends to the array. Returns the array.

---

### array/new

cfunction

(array/new capacity)

Creates a new empty array with a pre-allocated capacity. The same as (array) but can be more efficient if the maximum size of an array is known.

## EXAMPLES

```
(def arr (array/new 100)) # -> @[]

# Now we can fill up the array without
triggering a resize
(for i 0 100
  (put arr i i))
```

---

---

## **array/new-filled**

cfunction

(array/new-filled count &opt value)

Creates a new array of count elements, all set to value, which defaults to nil. Returns the new array.

---

## **array/peek**

cfunction

(array/peek arr)

Returns the last element of the array. Does not modify the array.

---

## **array/pop**

cfunction

(array/pop arr)

Remove the last element of the array and return it. If the array is empty, will return nil. Modifies the input array.

---

## **array/push**

cfunction

(array(push arr x)

Insert an element in the end of an array. Modifies the input array and

returns it.

---

## **array/remove**

cfunction

(array/remove arr at &opt n)

Remove up to n elements starting at index at in array arr. at can index from the end of the array with a negative index, and n must be a non-negative integer. By default, n is 1. Returns the array.

---

## **array/slice**

cfunction

(array/slice arrtup &opt start end)

Takes a slice of array or tuple from start to end. The range is half open, [start, end). Indexes can also be negative, indicating indexing from the end of the array. By default, start is 0 and end is the length of the array. Note that index -1 is synonymous with index (length arrtup) to allow a full negative slice range. Returns a new array.

[< Core API](#)

[Buffer Module >](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Buffer Module

[< Array Module](#)    [Debug Module >](#)

# Index

[buffer/bit](#) [buffer/bit-clear](#) [buffer/bit-set](#) [buffer/bit-toggle](#) [buffer/blit](#)  
[buffer/clear](#) [buffer/fill](#) [buffer/format](#) [buffer/new](#) [buffer/new-filled](#)  
[buffer/popn](#) [buffer/push-byte](#) [buffer/push-string](#) [buffer/push-word](#)  
[buffer/slice](#)

---

## **buffer/bit**

cfunction

(buffer/bit buffer index)

Gets the bit at the given bit-index. Returns true if the bit is set, false if not.

---

## **buffer/bit-clear**

cfunction

(buffer/bit-clear buffer index)

Clears the bit at the given bit-index. Returns the buffer.

---

## **buffer/bit-set**

cfunction

(buffer/bit-set buffer index)

Sets the bit at the given bit-index. Returns the buffer.

---

## **buffer/bit-toggle**

cfunction

(buffer/bit-toggle buffer index)

Toggles the bit at the given bit index in buffer. Returns the buffer.

---

## **buffer/blit**

cfunction

(buffer/blit dest src &opt dest-start src-start src-end)

Insert the contents of src into dest. Can optionally take indices that indicate which part of src to copy into which part of dest. Indices can be negative to index from the end of src or dest. Returns dest.

---

## **buffer/clear**

cfunction

(buffer/clear buffer)

Sets the size of a buffer to 0 and empties it. The buffer retains its memory so it can be efficiently refilled. Returns the modified buffer.

---

## **buffer/fill**

cfunction

(buffer/fill buffer &opt byte)

Fill up a buffer with bytes, defaulting to 0s. Does not change the

length of the buffer. Returns the modified buffer.

buffer's length. Returns the modified buffer.

---

## **buffer/format**

cfunction

(buffer/format buffer format & args)

Sprintf like functionality for printing values into a buffer. Returns the modified buffer.

---

## **buffer/new**

cfunction

(buffer/new capacity)

Creates a new, empty buffer with enough backing memory for capacity bytes. Returns a new buffer of length 0.

---

## **buffer/new-filled**

cfunction

(buffer/new-filled count &opt byte)

Creates a new buffer of length count filled with byte. By default, byte is 0. Returns the new buffer.

---

## **buffer/popn**

cfunction

(buffer/popn buffer n)

Removes the last n bytes from the buffer. Returns the modified buffer.

---

## **buffer/push-byte**

cfunction

(buffer/push-byte buffer x)

Append a byte to a buffer. Will expand the buffer as necessary.

Returns the modified buffer. Will throw an error if the buffer overflows.

---

## **buffer/push-string**

cfunction

(buffer/push-string buffer str)

Push a string onto the end of a buffer. Non string values will be converted to strings before being pushed. Returns the modified buffer. Will throw an error if the buffer overflows.

---

## **buffer/push-word**

cfunction

(buffer/push-word buffer x)

Append a machine word to a buffer. The 4 bytes of the integer are appended in twos complement, little endian order, unsigned. Returns the modified buffer. Will throw an error if the buffer overflows.

---

## **buffer/slice**

cfunction

(buffer/slice bytes &opt start end)

Takes a slice of a byte sequence from start to end. The range is half open, [start, end). Indexes can also be negative, indicating indexing from the end of the array. By default, start is 0 and end is the length of the buffer. Returns a new buffer.

[< Array Module](#)

[Debug Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Debug Module

[< Buffer Module](#)    [Fiber Module >](#)

# Index

[debug/arg-stack](#) [debug/break](#) [debug/fbreak](#) [debug/lineage](#) [debug/stack](#)  
[debug/stacktrace](#) [debug/step](#) [debug/unbreak](#) [debug/unfbreak](#)

---

## **debug/arg-stack**

cfunction

(debug/arg-stack fiber)

Gets all values currently on the fiber's argument stack. Normally, this should be empty unless the fiber signals while pushing arguments to make a function call. Returns a new array.

---

## **debug/break**

cfunction

(debug/break source byte-offset)

Sets a breakpoint with source a key at a given line and column. Will throw an error if the breakpoint location cannot be found. For example

(debug/break "core.janet" 1000)

wil set a breakpoint at the 1000th byte of the file core.janet.

---

## **debug/fbreak**

cfunction

(debug/fbreak fun &opt pc)

Set a breakpoint in a given function. pc is an optional offset, which is in bytecode instructions. fun is a function value. Will throw an error if the offset is too large or negative.

---

## **debug/lineage**

cfunction

(debug/lineage fib)

Returns an array of all child fibers from a root fiber. This function is useful when a fiber signals or errors to an ancestor fiber. Using this function, the fiber handling the error can see which fiber raised the signal. This function should be used mostly for debugging purposes.

---

## **debug/stack**

cfunction

(debug/stack fib)

Gets information about the stack as an array of tables. Each table in the array contains information about a stack frame. The top most, current stack frame is the first table in the array, and the bottom most stack frame is the last value. Each stack frame contains some of the following attributes:

- :c - true if the stack frame is a c function invocation
- :column - the current source column of the stack frame
- :function - the function that the stack frame represents
- :line - the current source line of the stack frame

:name - the human friendly name of the function  
:pc - integer indicating the location of the program counter  
:source - string with the file path or other identifier for the source code  
:slots - array of all values in each slot  
:tail - boolean indicating a tail call

---

## **debug/stacktrace**

cfunction

(debug/stacktrace fiber err)

Prints a nice looking stacktrace for a fiber. The error message err must be passed to the function as fiber's do not keep track of the last error they have thrown. Returns the fiber.

---

## **debug/step**

cfunction

(debug/step fiber &opt x)

Run a fiber for one virtual instruction of the Janet machine. Can optionally pass in a value that will be passed as the resuming value. Returns the signal value, which will usually be nil, as breakpoints raise nil signals.

---

## **debug/unbreak**

cfunction

(debug/unbreak source line column)

Remove a breakpoint with a source key at a given line and column.  
Will throw an error if the breakpoint cannot be found.

---

## **debug/unbreak**

cfunction

(debug/unbreak fun &opt pc)

Unset a breakpoint set with debug/fbreak.

[< Buffer Module](#)

[Fiber Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Fiber Module

[< Debug Module](#)    [File Module >](#)

# Index

[fiber/can-resume?](#) [fiber/current](#) [fiber/getenv](#) [fiber/maxstack](#) [fiber/new](#)  
[fiber/root](#) [fiber/setenv](#) [fiber/setmaxstack](#) [fiber/status](#)

---

## **fiber/can-resume?**

cfunction

(fiber/can-resume? fiber)

Check if a fiber is finished and cannot be resumed.

---

## **fiber/current**

cfunction

(fiber/current)

Returns the currently running fiber.

---

## **fiber/getenv**

cfunction

(fiber/getenv fiber)

Gets the environment for a fiber. Returns nil if no such table is set yet.

---

## **fiber/maxstack**

cfunction

(fiber/maxstack fib)

Gets the maximum stack size in janet values allowed for a fiber. While memory for the fiber's stack is not allocated up front, the fiber will not be allocated more than this amount and will throw a stack-overflow error if more memory is needed.

---

## **fiber/new**

cfunction

(fiber/new func &opt sigmask)

Create a new fiber with function body func. Can optionally take a set of signals to block from the current parent fiber when called. The mask is specified as a keyword where each character is used to indicate a signal to block. The default sigmask is :y. For example,

(fiber/new myfun :e123)

blocks error signals and user signals 1, 2 and 3. The signals are as follows:

- a - block all signals
- d - block debug signals
- e - block error signals
- t - block termination signals: error + user[0-4]
- u - block user signals
- y - block yield signals
- 0-9 - block a specific user signal

The sigmask argument also can take environment flags. If any mutually

exclusive flags are present, the last flag takes precedence.

i - inherit the environment from the current fiber

p - the environment table's prototype is the current environment table

---

## **fiber/root**

cfunction

(fiber/root)

Returns the current root fiber. The root fiber is the oldest ancestor that does not have a parent.

---

## **fiber/setenv**

cfunction

(fiber/setenv fiber table)

Sets the environment table for a fiber. Set to nil to remove the current environment.

---

## **fiber/setmaxstack**

cfunction

(fiber/setmaxstack fib maxstack)

Sets the maximum stack size in janet values for a fiber. By default, the maximum stack size is usually 8192.

---

## **fiber/status**

cfunction

(fiber/status fib)

Get the status of a fiber. The status will be one of:

- :dead - the fiber has finished
- :error - the fiber has errored out
- :debug - the fiber is suspended in debug mode
- :pending - the fiber has been yielded
- :user(0-9) - the fiber is suspended by a user signal
- :alive - the fiber is currently running and cannot be resumed
- :new - the fiber has just been created and not yet run

[< Debug Module](#)

[File Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# File Module

[< Fiber Module](#)    [Integer Types Module >](#)

# Index

[file/close](#) [file/flush](#) [file/open](#) [file/popen](#) [file/read](#) [file/seek](#) [file/temp](#) [file/write](#)

---

## **file/close**

cfunction

(file/close f)

Close a file and release all related resources. When you are done reading a file, close it to prevent a resource leak and let other processes read the file. If the file is the result of a file/popen call, close waits for and returns the process exit status.

---

## **file/flush**

cfunction

(file/flush f)

Flush any buffered bytes to the file system. In most files, writes are buffered for efficiency reasons. Returns the file handle.

---

## **file/open**

cfunction

(file/open path &opt mode)

Open a file. path is an absolute or relative path, and mode is a set of flags indicating the mode to open the file in. mode is a keyword where

each character represents a flag. If the file cannot be opened, returns nil, otherwise returns the new file handle. Mode flags:

- r - allow reading from the file
  - w - allow writing to the file
  - a - append to the file
  - b - open the file in binary mode (rather than text mode)
  - + - append to the file instead of overwriting it
- 

## **file/popen**

cfunction

(file/popen path &opt mode)

Open a file that is backed by a process. The file must be opened in either the :r (read) or the :w (write) mode. In :r mode, the stdout of the process can be read from the file. In :w mode, the stdin of the process can be written to. Returns the new file.

---

## **file/read**

cfunction

(file/read f what &opt buf)

Read a number of bytes from a file into a buffer. A buffer can be provided as an optional fourth argument, otherwise a new buffer is created. 'what' can either be an integer or a keyword. Returns the buffer with file contents. Values for 'what':

- :all - read the whole file
- :line - read up to and including the next newline character

n (integer) - read up to n bytes from the file

---

## **file/seek**

cfunction

(file/seek f &opt whence n)

Jump to a relative location in the file. 'whence' must be one of

- :cur - jump relative to the current file location
- :set - jump relative to the beginning of the file
- :end - jump relative to the end of the file

By default, 'whence' is :cur. Optionally a value n may be passed for the relative number of bytes to seek in the file. n may be a real number to handle large files of more than 4GB. Returns the file handle.

---

## **file/temp**

cfunction

(file/temp)

Open an anonymous temporary file that is removed on close. Raises an error on failure.

---

## **file/write**

cfunction

(file/write f bytes)

Writes to a file. 'bytes' must be string, buffer, or symbol. Returns the

file.

[< Fiber Module](#)

[Integer Types Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Integer Types Module

[< File Module](#)

[Math Module >](#)

# Index

[int/s64](#) [int/u64](#)

---

## **int/s64**

cfunction

(int/s64 value)

Create a boxed signed 64 bit integer from a string value.

---

## **int/u64**

cfunction

(int/u64 value)

Create a boxed unsigned 64 bit integer from a string value.

[< File Module](#)

[Math Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Math Module

[< Integer Types Module](#)    [Module Module >](#)

# Index

[math/-inf](#) [math/abs](#) [math/acos](#) [math/acosh](#) [math/asin](#) [math/asinh](#)  
[math/atan](#) [math/atan2](#) [math/atanh](#) [math/cbrt](#) [math/ceil](#) [math/cos](#)  
[math/cosh](#) [math/e](#) [math/erf](#) [math/erfc](#) [math/exp](#) [math/exp2](#) [math/expm1](#)  
[math/floor](#) [math/gamma](#) [math/hypot](#) [math/inf](#) [math/log](#) [math/log10](#)  
[math/log1p](#) [math/log2](#) [math/next](#) [math/pi](#) [math/pow](#) [math/random](#)  
[math/rng](#) [math/rng-buffer](#) [math/rng-int](#) [math/rng-uniform](#) [math/round](#)  
[math/seedrandom](#) [math/sin](#) [math/sinh](#) [math/sqrt](#) [math/tan](#) [math/tanh](#)  
[math/trunc](#)

---

## **math/-inf**

number

The number representing negative infinity

---

## **math/abs**

cfunction

(math/abs x)

Return the absolute value of x.

---

## **math/acos**

cfunction

(math/acos x)

Returns the arccosine of x.

---

## **math/acosh**

cfunction

(math/acosh x)

Return the hyperbolic arccosine of x.

---

## **math/asin**

cfunction

(math/asin x)

Returns the arcsine of x.

---

## **math/asinh**

cfunction

(math/asinh x)

Return the hyperbolic arcsine of x.

---

## **math/atan**

cfunction

(math/atan x)

Returns the arctangent of x.

---

## **math/atan2**

cfunction

(math/atan2 y x)

Return the arctangent of y/x. Works even when x is 0.

---

## **math/atanh**

cfunction

(math/atanh x)

Return the hyperbolic arctangent of x.

---

## **math/cbrt**

cfunction

(math/cbrt x)

Returns the cube root of x.

---

## **math/ceil**

cfunction

(math/ceil x)

Returns the smallest integer value number that is not less than x.

---

## **math/cos**

cfunction

(math/cos x)

Returns the cosine of x.

---

**math/cosh**

cfunction

(math/cosh x)

Return the hyperbolic cosine of x.

---

**math/e**

number

The base of the natural log.

---

**math/erf**

cfunction

(math/erf x)

Returns the error function of x.

---

**math/erfc**

cfunction

(math/erfc x)

Returns the complementary error function of x.

---

## **math/exp**

cfunction

(math/exp x)

Returns e to the power of x.

---

## **math/exp2**

cfunction

(math/exp2 x)

Returns 2 to the power of x.

---

## **math/expm1**

cfunction

(math/expm1 x)

Returns e to the power of x minus 1.

---

## **math/floor**

cfunction

(math/floor x)

Returns the largest integer value number that is not greater than x.

---

## **math/gamma**

cfunction

(math/gamma x)

Returns gamma(x).

---

### **math/hypot**

cfunction

(math/hypot a b)

Returns the c from the equation  $c^2 = a^2 + b^2$

---

### **math/inf**

number

The number representing positive infinity

---

### **math/log**

cfunction

(math/log x)

Returns log base natural number of x.

---

### **math/log10**

cfunction

(math/log10 x)

Returns log base 10 of x.

---

## **math/log1p**

cfunction

(math/log1p x)

Returns ( $\log_e x + 1$ ) more accurately than (+ (math/log x) 1)

---

## **math/log2**

cfunction

(math/log2 x)

Returns log base 2 of x.

---

## **math/next**

cfunction

(math/next x y)

Returns the next representable floating point value after x in the direction of y.

---

## **math/pi**

number

The value pi.

---

## **math/pow**

cfunction

(math/pow a x)

Return a to the power of x.

---

## **math/random**

cfunction

(math/random)

Returns a uniformly distributed random number between 0 and 1.

---

## **math/rng**

cfunction

(math/rng &opt seed)

Creates a Psuedo-Random number generator, with an optional seed. The seed should be an unsigned 32 bit integer or a buffer. Do not use this for cryptography. Returns a core/rng abstract type.

---

## **math/rng-buffer**

cfunction

(math/rng-buffer rng n &opt buf)

Get n random bytes and put them in a buffer. Creates a new buffer if no buffer is provided, otherwise appends to the given buffer. Returns the buffer.

---

## **math/rng-int**

cfunction

(math/rng-int rng &opt max)

Extract a random random integer in the range [0, max] from the RNG.  
If no max is given, the default is  $2^{31} - 1$ .

---

### **math/rng-uniform**

cfunction

(math/rng-seed rng seed)

Extract a random number in the range [0, 1) from the RNG.

---

### **math/round**

cfunction

(math/round x)

Returns the integer nearest to x.

---

### **math/seedrandom**

cfunction

(math/seedrandom seed)

Set the seed for the random number generator. seed should be an integer or a buffer.

---

### **math/sin**

cfunction

(math/sin x)

Returns the sine of x.

---

### **math/sinh**

cfunction

(math/sinh x)

Return the hyperbolic sine of x.

---

### **math/sqrt**

cfunction

(math/sqrt x)

Returns the square root of x.

---

### **math/tan**

cfunction

(math/tan x)

Returns the tangent of x.

---

### **math/tanh**

cfunction

(math/tanh x)

— . . . . —

Return the hyperbolic tangent of x.

---

**math/trunc**

cfunction

(math/trunc x)

Returns the integer between x and 0 nearest to x.

[< Integer Types Module](#)

[Module Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Module Module

[< Math Module](#)    [Net Module >](#)

# Index

[module/add-paths](#) [module/cache](#) [module/expand-path](#) [module/find](#)  
[module/loaders](#) [module/loading](#) [module/paths](#)

---

## **module/add-paths**

function

[source](#)

(module/add-paths ext loader)

Add paths to module/paths for a given loader such that the generated paths behave like other module types, including relative imports and syspath imports. ext is the file extension to associate with this module type, including the dot. loader is the keyword name of a loader that is module/loaders. Returns the modified module/paths.

---

## **module/cache**

table

[source](#)

Table mapping loaded module identifiers to their environments.

---

## **module/expand-path**

cfunction

(module/expand-path path template)

Expands a path template as found in module/paths for module/find.

This takes in a path (the argument to require) and a template string, template, to expand the path to a path that can be used for importing files. The replacements are as follows:

- :all: the value of path verbatim
- :cur: the current file, or (dyn :current-file)
- :dir: the directory containing the current file
- :name: the name component of path, with extension if given
- :native: the extension used to load natives, .so or .dll
- :sys: the system path, or (dyn :syspath)

---

## **module/find**

function

[source](#)

(module/find path)

Try to match a module or path name from the patterns in module/paths. Returns a tuple (fullpath kind) where the kind is one of :source, :native, or image if the module is found, otherwise a tuple with nil followed by an error message.

---

## **module/loaders**

table

[source](#)

A table of loading method names to loading functions. This table lets require and import load many different kinds of files as module.

---

## **module/loading**

table

[source](#)

Table mapping currently loading modules to true. Used to prevent circular dependencies.

---

## module/paths

array

[source](#)

The list of paths to look for modules, templated for module/expand-path. Each element is a two element tuple, containing the path template and a keyword :source, :native, or :image indicating how require should load files found at these paths.

A tuple can also contain a third element, specifying a filter that prevents module/find from searching that path template if the filter doesn't match the input path. The filter can be a string or a predicate function, and is often a file extension, including the period.

[< Math Module](#)

[Net Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Net Module

[< Module Module](#)    [OS Module >](#)

# Index

[net/chunk](#) [net/close](#) [net/connect](#) [net/read](#) [net/server](#) [net/write](#)

---

## **net/chunk**

cfunction

(net/chunk stream nbytes &opt buf)

Same a net/read, but will wait for all n bytes to arrive rather than return early.

---

## **net/close**

cfunction

(net/close stream)

Close a stream so that no further communication can occur.

---

## **net/connect**

cfunction

(net/connect host port)

Open a connection to communicate with a server. Returns a duplex stream that can be used to communicate with the server.

---

## **net/read**

cfunction

(net/read stream nbytes &opt buf)

Read up to n bytes from a stream, suspending the current fiber until the bytes are available. If less than n bytes are available (and more than 0), will push those bytes and return early. Returns a buffer with up to n more bytes in it.

---

## **net/server**

cfunction

(net/server host port handler)

Start a TCP server. handler is a function that will be called with a stream on each connection to the server. Returns a new stream that is neither readable nor writeable.

---

## **net/write**

cfunction

(net/write stream data)

Write data to a stream, suspending the current fiber until the write completes. Returns stream.

[< Module](#) [Module >](#)

[OS Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# OS Module

[< Net Module](#)

[PEG Module >](#)

The `os` module contains most Operating System specific functionality as well as routines for interacting with the host OS. There is also some functionality for interacting with the file system. The functionality in this module can be much reduced by setting the `JANET_REDUCED_OS` define in `janetconf.h`.

# Index

[os/arch](#) [os/cd](#) [os/chmod](#) [os/clock](#) [os/cryptorand](#) [os/cwd](#) [os/date](#) [os/dir](#)  
[os/environ](#) [os/execute](#) [os/exit](#) [os/getenv](#) [os/link](#) [os/lstat](#) [os/mkdir](#)  
[os/mktime](#) [os/perm-int](#) [os/perm-string](#) [os/readlink](#) [os.realpath](#) [os/rename](#)  
[os/rm](#) [os/rmdir](#) [os/setenv](#) [os/shell](#) [os/sleep](#) [os/stat](#) [os/symlink](#) [os/time](#)  
[os/touch](#) [os/umask](#) [os/which](#)

---

## **os/arch**

cfunction

(os/arch)

Check the ISA that janet was compiled for. Returns one of:

- :x86
- :x86-64
- :arm
- :aarch64
- :sparc
- :wasm
- :unknown

---

## **os/cd**

cfunction

(os/cd path)

Change current directory to path. Returns nil on success, errors on

failure.

---

## **os/chmod**

cfunction

(os/chmod path mode)

Change file permissions, where mode is a permission string as returned by os/perm-string, or an integer as returned by os/perm-int. When mode is an integer, it is interpreted as a Unix permission value, best specified in octal, like 8r666 or 8r400. Windows will not differentiate between user, group, and other permissions, and thus will combine all of these permissions. Returns nil.

---

## **os/clock**

cfunction

(os/clock)

Return the number of seconds since some fixed point in time. The clock is guaranteed to be non decreasing in real time.

---

## **os/cryptorand**

cfunction

(os/cryptorand n &opt buf)

Get or append n bytes of good quality random data provided by the OS. Returns a new buffer or buf.

---

## **os/cwd**

cfunction

(os/cwd)

Returns the current working directory.

---

## **os/date**

cfunction

(os/date &opt time local)

Returns the given time as a date struct, or the current time if no time is given. Returns a struct with following key values. Note that all numbers are 0-indexed. Date is given in UTC unless local is truthy, in which case the date is formatted for the local timezone.

- :seconds - number of seconds [0-61]
- :minutes - number of minutes [0-59]
- :hours - number of hours [0-23]
- :month-day - day of month [0-30]
- :month - month of year [0, 11]
- :year - years since year 0 (e.g. 2019)
- :week-day - day of the week [0-6]
- :year-day - day of the year [0-365]
- :dst - If Day Light Savings is in effect

---

## **os/dir**

cfunction

(os/dir dir &opt array)

---

Iterate over files and subdirectories in a directory. Returns an array of paths parts, with only the file name or directory name and no prefix.

---

## **os/environ**

cfunction

(os/environ)

Get a copy of the os environment table.

---

## **os/execute**

cfunction

(os/execute args &opts flags env)

Execute a program on the system and pass it string arguments. Flags is a keyword that modifies how the program will execute.

:e - enables passing an environment to the program. Without :e, the current environment is inherited.

:p - allows searching the current PATH for the binary to execute. Without this flag, binaries must use absolute paths.

env is a table or struct mapping environment variables to values.  
Returns the exit status of the program.

---

## **os/exit**

cfunction

(os/exit &opt x)

Exit from janet with an exit code equal to x. If x is not an integer, the exit with status equal the hash of x.

---

### **os/getenv**

cfunction

(os/getenv variable &opt dflt)

Get the string value of an environment variable.

---

### **os/link**

cfunction

(os/link oldpath newpath &opt symlink)

Create a link at newpath that points to oldpath and returns nil. Iff symlink is truthy, creates a symlink. Iff symlink is falsey or not provided, creates a hard link. Does not work on Windows.

---

### **os/lstat**

cfunction

(os/lstat path &opt tab|key)

Like os/stat, but don't follow symlinks.

---

### **os/mkdir**

cfunction

(os/mkdir path)

Create a new directory. The path will be relative to the current directory if relative, otherwise it will be an absolute path. Returns true if the directory was created, false if the directory already exists, and errors otherwise.

---

### **os/mktime**

cfunction

(os/mktime date-struct &opt local)

Get the broken down date-struct time expressed as the number of seconds since January 1, 1970, the Unix epoch. Returns a real number. Date is given in UTC unless local is truthy, in which case the date is computed for the local timezone.

Inverse function to os/date.

---

### **os/perm-int**

cfunction

(os/perm-int bytes)

Parse a 9 character permission string and return an integer that can be used by chmod.

---

### **os/perm-string**

cfunction

(os/perm-string int)

Convert a Unix octal permission value from a permission integer as returned by os/stat to a human readable string, that follows the formatting of unix tools like ls. Returns the string as a 9 character string of r, w, x and - characters. Does not include the file/directory/symlink character as rendered by `ls`.

---

## **os/readlink**

cfunction

(os/readlink path)

Read the contents of a symbolic link. Does not work on Windows.

---

## **os/realpath**

cfunction

(os/realpath path)

Get the absolute path for a given path, following ../, ./, and symlinks. Returns an absolute path as a string. Will raise an error on Windows.

---

## **os/rename**

cfunction

(os/rename oldname newname)

Rename a file on disk to a new path. Returns nil.

---

## **os/rm**

experimental

cfunction

(os/rm path)

Delete a file. Returns nil.

---

**os/rmdir**

cfunction

(os/rmdir path)

Delete a directory. The directory must be empty to succeed.

---

**os/setenv**

cfunction

(os/setenv variable value)

Set an environment variable.

---

**os/shell**

cfunction

(os/shell str)

Pass a command string str directly to the system shell.

---

**os/sleep**

cfunction

(os/sleep nsec)

Suspend the program for nsec seconds. 'nsec' can be a real number.  
Returns nil.

---

## **os/stat**

cfunction

(os/stat path &opt tab|key)

Gets information about a file or directory. Returns a table If the third argument is a keyword, returns only that information from stat. If the file or directory does not exist, returns nil. The keys are

- :dev - the device that the file is on
- :mode - the type of file, one of :file, :directory, :block, :character, :fifo, :socket, :link, or :other
- :int-permissions - A Unix permission integer like 8r744
- :permissions - A Unix permission string like "rwxr--r--"
- :uid - File uid
- :gid - File gid
- :nlink - number of links to file
- :rdev - Real device of file. 0 on windows.
- :size - size of file in bytes
- :blocks - number of blocks in file. 0 on windows
- :blocksize - size of blocks in file. 0 on windows
- :accessed - timestamp when file last accessed
- :changed - timestamp when file last changed (permissions changed)
- :modified - timestamp when file last modified (content changed)

---

## **os/symlink**

cfunction

(os/symlink oldpath newpath)

Create a symlink from oldpath to newpath, returning nil. Same as (os/link oldpath newpath true).

---

## **os/time**

cfunction

(os/time)

Get the current time expressed as the number of seconds since January 1, 1970, the Unix epoch. Returns a real number.

---

## **os/touch**

cfunction

(os(touch path &opt actime modtime)

Update the access time and modification times for a file. By default, sets times to the current time.

---

## **os/umask**

cfunction

(os/umask mask)

Set a new umask, returns the old umask.

---

## **os/which**

cfunction

(os/which)

Check the current operating system. Returns one of:

- :windows
- :macos
- :web - Web assembly (emscripten)
- :linux
- :freebsd
- :openbsd
- :netbsd
- :posix - A POSIX compatible system (default)

May also return a custom keyword specified at build time.

[< Net Module](#)

[PEG Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# PEG Module

[< OS Module](#)

[Parser Module >](#)

See [The peg documentation](#) for more information.

# Index

## [peg/compile](#) [peg/match](#)

---

### **peg/compile**

cfunction

(`peg/compile` `peg`)

Compiles a peg source data structure into a <core/peg>. This will speed up matching if the same peg will be used multiple times.

---

### **peg/match**

cfunction

(`peg/match` `peg` `text` &`opt` `start` & `args`)

Match a Parsing Expression Grammar to a byte string and return an array of captured values. Returns nil if text does not match the language defined by peg. The syntax of PEGs is documented on the Janet website.

[< OS Module](#)

[Parser Module >](#)

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Parser Module

[< PEG Module](#)    [String Module >](#)

# Index

[parser/byte](#) [parser/clone](#) [parser/consume](#) [parser/eof](#) [parser/error](#)  
[parser/flush](#) [parser/has-more](#) [parser/insert](#) [parser/new](#) [parser/produce](#)  
[parser/state](#) [parser/status](#) [parser/where](#)

---

## **parser/byte**

cfunction

(parser/byte parser b)

Input a single byte into the parser byte stream. Returns the parser.

---

## **parser/clone**

cfunction

(parser/clone p)

Creates a deep clone of a parser that is identical to the input parser. This cloned parser can be used to continue parsing from a good checkpoint if parsing later fails. Returns a new parser.

---

## **parser/consume**

cfunction

(parser/consume parser bytes &opt index)

Input bytes into the parser and parse them. Will not throw errors if there is a parse error. Starts at the byte index given by index. Returns

the number of bytes read.

---

### **parser/eof**

cfunction

(parser/eof parser)

Indicate that the end of file was reached to the parser. This puts the parser in the :dead state.

---

### **parser/error**

cfunction

(parser/error parser)

If the parser is in the error state, returns the message associated with that error. Otherwise, returns nil. Also flushes the parser state and parser queue, so be sure to handle everything in the queue before calling parser/error.

---

### **parser/flush**

cfunction

(parser/flush parser)

Clears the parser state and parse queue. Can be used to reset the parser if an error was encountered. Does not reset the line and column counter, so to begin parsing in a new context, create a new parser.

---

## **parser/has-more**

cfunction

(parser/has-more parser)

Check if the parser has more values in the value queue.

---

## **parser/insert**

cfunction

(parser/insert parser value)

Insert a value into the parser. This means that the parser state can be manipulated in between chunks of bytes. This would allow a user to add extra elements to arrays and tuples, for example. Returns the parser.

---

## **parser/new**

cfunction

(parser/new)

Creates and returns a new parser object. Parsers are state machines that can receive bytes, and generate a stream of values.

---

## **parser/produce**

cfunction

(parser/produce parser)

Dequeue the next value in the parse queue. Will return nil if no parsed values are in the queue. Otherwise will dequeue the next value.

values are in the queue, otherwise will enqueue the next value.

---

## **parser/state**

cfunction

(parser/state parser &opt key)

Returns a representation of the internal state of the parser. If a key is passed, only that information about the state is returned. Allowed keys are:

:delimiters - Each byte in the string represents a nested data structure. For example, if the parser state is '(["', then the parser is in the middle of parsing a string inside of square brackets inside parentheses. Can be used to augment a REPL prompt. :frames - Each table in the array represents a 'frame' in the parser state. Frames contain information about the start of the expression being parsed as well as the type of that expression and some type-specific information.

---

## **parser/status**

cfunction

(parser/status parser)

Gets the current status of the parser state machine. The status will be one of:

:pending - a value is being parsed.

:error - a parsing error was encountered.

:root - the parser can either read more values or safely terminate.

---

## **parser/where**

cfunction

(parser/where parser)

Returns the current line number and column of the parser's internal state.

[< PEG Module](#)

[String Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# String Module

[< Parser Module](#)    [Table Module >](#)

# Index

[string/ascii-lower](#) [string/ascii-upper](#) [string/bytes](#) [string/check-set](#)  
[string/find](#) [string/find-all](#) [string/format](#) [string/from-bytes](#) [string/has-prefix?](#)  
[string/has-suffix?](#) [string/join](#) [string/repeat](#) [string/replace](#) [string/replace-all](#)  
[string/reverse](#) [string/slice](#) [string/split](#) [string/trim](#) [string/triml](#) [string/trimr](#)

---

## **string/ascii-lower**

cfunction

(string/ascii-lower str)

Returns a new string where all bytes are replaced with the lowercase version of themselves in ASCII. Does only a very simple case check, meaning no unicode support.

---

## **string/ascii-upper**

cfunction

(string/ascii-upper str)

Returns a new string where all bytes are replaced with the uppercase version of themselves in ASCII. Does only a very simple case check, meaning no unicode support.

---

## **string/bytes**

cfunction

(string/bytes str)

Returns an array of integers that are the byte values of the string.

## string/check-set

## cfunction

(string/check-set set str)

Checks that the string str only contains bytes that appear in the string set. Returns true if all bytes in str appear in set, false if some bytes in str do not appear in set.

## string/find

## cfunction

(string/find patt str)

Searches for the first instance of pattern *patt* in string *str*. Returns the index of the first character in *patt* if found, otherwise returns nil.

## string/find-all

## cfunction

(string/find-all patt str)

Searches for all instances of pattern `patt` in string `str`. Returns an array of all indices of found patterns. Overlapping instances of the pattern are not counted, meaning a byte in `string` will only contribute to finding at most one occurrence of pattern. If no occurrences are found, will return an empty array.

---

## **string/format**

cfunction

(string/format format & values)

Similar to sprintf, but specialized for operating with Janet values.

Returns a new string.

---

## **string/from-bytes**

cfunction

(string/from-bytes & byte-vals)

Creates a string from integer parameters with byte values. All integers will be coerced to the range of 1 byte 0-255.

---

## **string/has-prefix?**

cfunction

(string/has-prefix? pfx str)

Tests whether str starts with pfx.

---

## **string/has-suffix?**

cfunction

(string/has-suffix? sfx str)

Tests whether str ends with sfx.

---

## **string/join**

cfunction

(string/join parts &opt sep)

Joins an array of strings into one string, optionally separated by a separator string sep.

---

## **string/repeat**

cfunction

(string/repeat bytes n)

Returns a string that is n copies of bytes concatenated.

---

## **string/replace**

cfunction

(string/replace patt subst str)

Replace the first occurrence of patt with subst in the string str. Will return the new string if patt is found, otherwise returns str.

---

## **string/replace-all**

cfunction

(string/replace-all patt subst str)

Replace all instances of patt with subst in the string str. Will return the new string if patt is found, otherwise returns str.

---

## **string/reverse**

cfunction

(string/reverse str)

Returns a string that is the reversed version of str.

---

## **string/slice**

cfunction

(string/slice bytes &opt start end)

Returns a substring from a byte sequence. The substring is from index start inclusive to index end exclusive. All indexing is from 0. 'start' and 'end' can also be negative to indicate indexing from the end of the string. Note that index -1 is synonymous with index (length bytes) to allow a full negative slice range.

---

## **string/split**

cfunction

(string/split delim str &opt start limit)

Splits a string str with delimiter delim and returns an array of substrings. The substrings will not contain the delimiter delim. If delim is not found, the returned array will have one element. Will start searching for delim at the index start (if provided), and return up to a maximum of limit results (if provided).

---

## **string/trim**

cfunction

(string/trim str &opt set)

Trim leading and trailing whitespace from a byte sequence. If the argument set is provided, consider only characters in set to be whitespace.

---

### **string/triml**

cfunction

(string/triml str &opt set)

Trim leading whitespace from a byte sequence. If the argument set is provided, consider only characters in set to be whitespace.

---

### **string/trimr**

cfunction

(string/trimr str &opt set)

Trim trailing whitespace from a byte sequence. If the argument set is provided, consider only characters in set to be whitespace.

[< Parser Module](#)

[Table Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Table Module

[< String Module](#)    [Threads >](#)

# Index

[table/clone](#) [table/getproto](#) [table/new](#) [table/rawget](#) [table/setproto](#) [table/to-struct](#)

---

## **table/clone**

cfunction

(table/clone tab)

Create a copy of a table. Updates to the new table will not change the old table, and vice versa.

---

## **table/getproto**

cfunction

(table/getproto tab)

Get the prototype table of a table. Returns nil if a table has no prototype, otherwise returns the prototype.

---

## **table/new**

cfunction

(table/new capacity)

Creates a new empty table with pre-allocated memory for capacity entries. This means that if one knows the number of entries going to go in a table on creation. extra memory allocation can be avoided.

Returns the new table.

---

### **table/rawget**

cfunction

(table/rawget tab key)

Gets a value from a table without looking at the prototype table. If a table tab does not contain t directly, the function will return nil without checking the prototype. Returns the value in the table.

---

### **table/setproto**

cfunction

(table/setproto tab proto)

Set the prototype of a table. Returns the original table tab.

---

### **table/to-struct**

cfunction

(table/to-struct tab)

Convert a table to a struct. Returns a new struct. This function does not take into account prototype tables.

[< String Module](#)

[Threads >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Threads

[< Table Module](#)    [Tuple Module >](#)

# Index

[thread/close](#) [thread/current](#) [thread/new](#) [thread/receive](#) [thread/send](#)

---

## **thread/close**

cfunction

(thread/close thread)

Close a thread, unblocking it and ending communication with it. Note that closing a thread is idempotent and does not cancel the thread's operation. Returns nil.

---

## **thread/current**

cfunction

(thread/current)

Get the current running thread.

---

## **thread/new**

cfunction

(thread/new func &opt capacity)

Start a new thread that will start immediately. If capacity is provided, that is how many messages can be stored in the thread's mailbox before blocking senders. The capacity must be between 1 and 65535 inclusive. and defaults to 10. Returns a handle to the new thread.

---

## **thread/receive**

cfunction

(thread/receive &opt timeout)

Get a message sent to this thread. If timeout is provided, an error will be thrown after the timeout has elapsed but no messages are received.

---

## **thread/send**

cfunction

(thread/send thread msg)

Send a message to the thread. This will never block and returns thread immediately. Will throw an error if there is a problem sending the message.

[< Table Module](#)

[Tuple Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Tuple Module

[< Threads](#)

[Type Array Module >](#)

# Index

[tuple/brackets](#) [tuple/setmap](#) [tuple/slice](#) [tuple/sourcemap](#) [tuple/type](#)

---

## **tuple/brackets**

cfunction

(tuple/brackets & xs)

Creates a new bracketed tuple containing the elements xs.

---

## **tuple/setmap**

cfunction

(tuple/setmap tup line column)

Set the sourcemap metadata on a tuple. line and column indicate  
should be integers.

---

## **tuple/slice**

cfunction

(tuple/slice arrtup [,start=0 [,end=(length arrtup)]])

Take a sub sequence of an array or tuple from index start inclusive to  
index end exclusive. If start or end are not provided, they default to 0  
and the length of arrtup respectively. 'start' and 'end' can also be  
negative to indicate indexing from the end of the input. Note that index  
-1 is synonymous with index '(length arrtup)' to allow a full negative

slice range. Returns the new tuple.

---

## **tuple/sourcemap**

cfunction

(tuple/sourcemap tup)

Returns the sourcemap metadata attached to a tuple, which is another tuple (line, column).

---

## **tuple/type**

cfunction

(tuple/type tup)

Checks how the tuple was constructed. Will return the keyword :brackets if the tuple was parsed with brackets, and :parens otherwise. The two types of tuples will behave the same most of the time, but will print differently and be treated differently by the compiler.

[< Threads](#)

[Type Array Module >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Type Array Module

[< Tuple Module](#)    [The Janet C API >](#)

# Index

[tarray/buffer](#) [tarray/copy-bytes](#) [tarray/length](#) [tarray/new](#) [tarray/properties](#)  
[tarray/slice](#) [tarray/swap-bytes](#)

---

## **tarray/buffer**

cfunction

(tarray/buffer array|size)

Return typed array buffer or create a new buffer.

---

## **tarray/copy-bytes**

cfunction

(tarray/copy-bytes src sindex dst dindex &opt count)

Copy count elements (default 1) of src array from index sindex to dst array at position dindex memory can overlap.

---

## **tarray/length**

cfunction

(tarray/length array|buffer)

Return typed array or buffer size.

---

## **tarray/new**

cfunction

(tarray/new type size &opt stride offset tarray|buffer)

Create new typed array.

---

## **tarray/properties**

cfunction

(tarray/properties array)

Return typed array properties as a struct.

---

## **tarray/slice**

cfunction

(tarray/slice tarr &opt start end)

Takes a slice of a typed array from start to end. The range is half open, [start, end). Indexes can also be negative, indicating indexing from the end of the end of the typed array. By default, start is 0 and end is the size of the typed array. Returns a new janet array.

---

## **tarray/swap-bytes**

cfunction

(tarray/swap-bytes src sindex dst dindex &opt count)

Swap count elements (default 1) between src array from index sindex and dst array at position dindex memory can overlap.

[< Tuple Module](#)

[The Janet C API >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# The Janet C API

[< Type Array Module](#)

[Wrapping Types >](#)

One of the fundamental design goals of Janet is to extend it via the C programming language, as well as embed the interpreter in a larger program. To this end, Janet exposes most of its low level functionality in well defined C API. Unlike Janet, the Janet C API is not safe — it's possible to write programs in C that will crash, use memory after freeing it, or simply cause undefined behavior. However, using the C API will let you use high performance libraries on almost any system and access native functionality that is not possible in pure Janet.

Janet's C API assumes that you will have one Janet interpreter per OS thread (if the system you are on has threads). This means that Janet must have some global, thread local state. This is a problem for some host languages, like Go, that expect all code to be fully re-entrant on any thread. This design choice, however, makes the Janet APIs more pleasant to use in languages like C and C++, as there is no need to pass around a context object to all api functions.

# What defines the C API?

The C API is defined by the header `janet.h`, which should be included by programs to both embed Janet in a larger program, or to write a Janet module.

# Using The Amalgamation

When integrating Janet into a host project (usually some larger application, but it could be a small wrapper around Janet), it is convenient to be able to add only a few source files to your project rather than a large dependency. The amalgamated source, `janet.c`, contains all of the source code for Janet concatenated into one file. This file can be added to the host project, along with `janet.h` and `janetconf.h`. There is also the option of including `shell.c`, which is used in the standalone interpreter, but not usually needed for host programs.

On a POSIX system, building the amalgamated source is straightforward. With `janet.c`, `janet.h`, `janetconf.h`, and `shell.c` in the current directory, building is as follows:

```
cc -std=c99 -Wall -Werror -O2 -fPIC -shared  
janet.c shell.c -o janet -lm -ldl -lrt -lpthread
```

The exact flags required will depend on the system and on the configuration of the amalgamation. For example, linking to pthreads is not needed if `JANET_SINGLE_THREADED` is defined in `janetconf.h`.

# Writing a Module

The easiest way to get started with the Janet C API is to write a Janet native module. A native module is native code that can be loaded dynamically at runtime in the interpreter.

A basic skeleton for a native module in C is below.

```
#include <janet.h>

static Janet myfun(int32_t argc, Janet *argv) {
    janet_fixarity(argc, 0);
    printf("hello from a module!\n");
    return janet_wrap_nil();
}

static const JanetReg cfuns[] = {
    {"myfun", myfun, "(mymod/myfun)\n\nPrints a
hello message."},
    {NULL, NULL, NULL}
};

JANET_MODULE_ENTRY(JanetTable *env) {
    janet_cfuns(env, "mymod", cfuns);
}
```

This module is a very simple module that exposes one native function called `myfun`. The module is called `mymod`, although when importing the module, the user can rename it to whatever he or she likes.

## Compiling the Module

Once you have written your native code in a file `mymod.c`, you will need to compile it with your system's C compiler. The easiest way to do this is to use `jpm` and write a `project.janet` file. `jpm` will handle all of the linking and flags for you so that you can write portable modules. Since `project.janet` is just a Janet script, you can detect what platform you are running on and do conditional compilation there if needed.

Inside `project.janet`:

```
(declare-project :name "mymod")
(declare-native :name "mymod" :source
@["mymod.c"])
```

Once you have `project.janet` written, run `jpm build` to build the module. If all goes well, `jpm` should have built a file `build/mymod.so`. In your repl, you can now import your module and use it.

```
(import build/mymod :as mymod)
(mymod/myfun) # Prints the hello message
```

Congratulations, you have a native module.

[< Type Array Module](#)

[Wrapping Types >](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Wrapping Types

[< The Janet C API](#)

[Embedding >](#)

Janet has several built-in fundamental data types. These data types are the bread and butter of the Janet C API, and most functions and macros in the API expect at least some of their arguments to be of these types. However, as Janet is a dynamically typed language, Janet internally uses a boxed representation of these types, simply called `Janet`.

Functions that are specific to a particular type expect the unboxed or unwrapped type, while operations that are generic to Janet values will typically expect a `Janet`. It is important to be able to convert between these two representations in order to interact with Janet from C.

# Wrapping Functions

These functions convert a C type into a generic Janet value.

```
Janet janet_wrap_nil(void);
Janet janet_wrap_number(double x);
Janet janet_wrap_true(void);
Janet janet_wrap_false(void);
Janet janet_wrap_boolean(int x);
Janet janet_wrap_string(const uint8_t *x);
Janet janet_wrap_symbol(const uint8_t *x);
Janet janet_wrap_keyword(const uint8_t *x);
Janet janet_wrap_array(JanetArray *x);
Janet janet_wrap_tuple(const Janet *x);
Janet janet_wrap_struct(const JanetKV *x);
Janet janet_wrap_fiber(JanetFiber *x);
Janet janet_wrap_buffer(JanetBuffer *x);
Janet janet_wrap_function(JanetFunction *x);
Janet janet_wrap_cfunction(JanetCFunction x);
Janet janet_wrap_table(JanetTable *x);
Janet janet_wrap_abstract(void *x);
Janet janet_wrap_pointer(void *x);
Janet janet_wrap_integer(int32_t x);
```

# Unwrapping Functions

These functions convert from a `Janet` to a more specific C type. If the `Janet` being unwrapped is not actually the returned type, the behavior is undefined.

```
const JanetKV *janet_unwrap_struct(Janet x);
const Janet *janet_unwrap_tuple(Janet x);
JanetFiber *janet_unwrap_fiber(Janet x);
JanetArray *janet_unwrap_array(Janet x);
JanetTable *janet_unwrap_table(Janet x);
JanetBuffer *janet_unwrap_buffer(Janet x);
const uint8_t *janet_unwrap_string(Janet x);
const uint8_t *janet_unwrap_symbol(Janet x);
const uint8_t *janet_unwrap_keyword(Janet x);
void *janet_unwrap_abstract(Janet x);
void *janet_unwrap_pointer(Janet x);
JanetFunction *janet_unwrap_function(Janet x);
JanetCFunction janet_unwrap_cfunction(Janet x);
int janet_unwrap_boolean(Janet x);
double janet_unwrap_number(Janet x);
int32_t janet_unwrap_integer(Janet x);
```

# Janet Types

Before unwrapping a Janet value, one should check the value is of the correct Janet type. Janet provides two macros for checking this, `janet_checktype` and `janet_checktypes`. These macros may be more efficient than the more obvious way of checking types via `janet_type(x) == JANET_ARRAY`, and so are preferred.

The enum `JanetType` represents the base type of a Janet value. There are exactly 16 basic types.

```
typedef enum JanetType {
    JANET_NUMBER,
    JANET_NIL,
    JANET_BOOLEAN,
    JANET_FIBER,
    JANET_STRING,
    JANET_SYMBOL,
    JANET_KEYWORD,
    JANET_ARRAY,
    JANET_TUPLE,
    JANET_TABLE,
    JANET_STRUCT,
    JANET_BUFFER,
    JANET_FUNCTION,
    JANET_CFUNCTION,
    JANET_ABSTRACT,
    JANET_POINTER
} JanetType;
```



```
/* macro */
int janet_checktype(Janet x, JanetType type);
```

Checks the type `x`, and returns a non zero value if `x` is the correct type, and 0 otherwise. Usually, you will want to use this function before unwrapping a value.

---

```
/* macro */
int janet_checktypes(Janet x, int typeflags);
```

Similar to `janet_checktype`, but allows the programmer to check multiple types at once, in an efficient manner. `typeflags` should be a bit set of all of the types to check membership for. For example, to check if `x` is one of nil or a number, you would call

```
janet_checktypes(x, (1 << JANET_NIL) | (1 << JANET_NUMBER))
```

There are also several aliases so you don't need to do the shift yourself.

```
#define JANET_TFLAG_NIL (1 << JANET_NIL)
#define JANET_TFLAG_BOOLEAN (1 << JANET_BOOLEAN)
#define JANET_TFLAG_FIBER (1 << JANET_FIBER)
#define JANET_TFLAG_NUMBER (1 << JANET_NUMBER)
#define JANET_TFLAG_STRING (1 << JANET_STRING)
#define JANET_TFLAG_SYMBOL (1 << JANET_SYMBOL)
#define JANET_TFLAG_KEYWORD (1 << JANET_KEYWORD)
#define JANET_TFLAG_ARRAY (1 << JANET_ARRAY)
#define JANET_TFLAG_TUPLE (1 << JANET_TUPLE)
#define JANET_TFLAG_TABLE (1 << JANET_TABLE)
#define JANET_TFLAG_STRUCT (1 << JANET_STRUCT)
```

```
#define JANET_TFLAG_BUFFER (1 << JANET_BUFFER)
#define JANET_TFLAG_FUNCTION (1 <<
JANET_FUNCTION)
#define JANET_TFLAG_CFUNCTION (1 <<
JANET_CFUNCTION)
#define JANET_TFLAG_ABSTRACT (1 <<
JANET_ABSTRACT)
#define JANET_TFLAG_POINTER (1 << JANET_POINTER)

/* Some abstractions */
#define JANET_TFLAG_BYTES (JANET_TFLAG_STRING |  

JANET_TFLAG_SYMBOL | JANET_TFLAG_BUFFER |  

JANET_TFLAG_KEYWORD)
#define JANET_TFLAG_INDEXED (JANET_TFLAG_ARRAY |  

JANET_TFLAG_TUPLE)
#define JANET_TFLAG_DICTIONARY (JANET_TFLAG_TABLE  

| JANET_TFLAG_STRUCT)
#define JANET_TFLAG_LENGTHTABLE (JANET_TFLAG_BYTES  

| JANET_TFLAG_INDEXED | JANET_TFLAG_DICTIONARY)
#define JANET_TFLAG_CALLABLE  

(JANET_TFLAG_FUNCTION | JANET_TFLAG_CFUNCTION | \  

JANET_TFLAG_LENGTHTABLE |  

JANET_TFLAG_ABSTRACT)
```

The resulting call would look like:

```
janet_checktypes(x, JANET_TFLAG_NIL |  

JANET_TFLAG_NUMBER)
```

---

```
JANET_API JanetType janet_type(Janet x);
```

Returns the type of Janet value.

[< The Janet C API](#)

[Embedding >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Embedding

[< Wrapping Types](#)

[Configuration >](#)

Janet can be embedded in a host program, usually written in C or C++, in several ways. The easiest way is to use the Janet's amalgamated build, which is a single giant C source file that can be copied into the host program's source tree. This method has many upsides, including simplicity and stability. You can also link to `libjanet.so` to include Janet as shared object.

# Clients

In the context of embedding Janet in a program, we can call the larger program a client which usually uses the Janet library to run Janet source code. Most Janet clients have a number of things in common, especially some boilerplate required to get Janet code running.

The simplest Janet client will need to initialize the VM, load the core environment, run some Janet code, and then deinitialize the environment. A simple example program that does this is below.

```
#include <janet.h>

int main(int argc, const char *argv[]) {
    // Initialize the virtual machine. Do this
    // before any calls to Janet functions.
    janet_init();

    // Get the core janet environment. This
    // contains all of the C functions in the core
    // as well as the code in
    // src/boot/boot.janet.
    JanetTable *env = janet_core_env(NULL);

    // One of several ways to begin the Janet vm.
    janet_dostring(env, "(print `hello,
world!`)", "main", NULL);

    // Use this to free all resources allocated
    // by Janet.
    janet_deinit();
    return 0;
}
```

---

# Basic Janet Functions

```
JANET_API int janet_init(void);
```

Use this function to initialize global Janet state. This must be called once per thread if using Janet in a multithreaded environment, as all Janet global state is thread local by default.

```
JANET_API void janet_deinit(void);
```

Call this function to free all memory and resources managed by Janet.

```
JanetTable *janet_core_env(JanetTable  
*replacements);
```

Use this function to get the core environment for the Janet language. Replacements is an optional table that can be used to override some of the default bindings in the core environment. Usually, set to NULL.

```
int janet_dobytes(JanetTable *env, const uint8_t  
*bytes, int32_t len, const char *sourcePath,  
Janet *out);
```

Use this function to compile and run some Janet source code from C. If you plan on running the code multiple times, there are more efficient options. However, for code that will only run once, this is a useful function.

```
int janet_destring(JanetTable *env, const char
```

```
*str, const char *sourcePath, Janet *out);
```

Similar to `janet_dobytes`, runs a null-terminated C string of Janet source code.

```
JanetSignal janet_continue(JanetFiber *fiber,  
Janet in, Janet *out);
```

Resumes a new or suspended fiber. Returns a signal that corresponds to the status of the fiber after execution, and places the return/signal value in `out`. When resuming a fiber, the value to resume with should be in the argument `in`, which corresponds to the second argument to the Janet `resume` function.

```
JanetSignal janet_pcall(JanetFunction *fun,  
int32_t argc, const Janet *argv, Janet *out,  
JanetFiber **f);
```

Invoke a function in a protected manner, catching any panics raised. Returns the resulting status code, as well as placing the return value in `*out`. The fiber pointer `f` is a pointer to a fiber pointer will contain the fiber used to run the function `fun`. If `f` is `NULL`, a new fiber will be created. Otherwise, Janet will use the fiber pointed to by `f` to run `fun`.

If no panics are raised, will return `JANET_SIGNAL_OK`. If an error is raised, will return `JANET_SIGNAL_ERROR`. Other signals will be returned as expected.

```
Janet janet_call(JanetFunction *fun, int32_t  
argc, const Janet *argv);
```

Make a simple, re-entrant call back into the Janet interpreter from C. Try to not modify globals about running logic here as this overwrides them

to put mostly simple, short running logic here as this suspends the garbage collector for the duration of the call. Functions called this way also must either return or error. Other signals will be coerced into an error.

```
void janet_stacktrace(JanetFiber *fiber, Janet  
err);
```

Print a janet stacktrace for the given fiber to stderr, or whatever is bound to the `:err` dynamic binding on the current fiber. You also must supply the return value or error message raised, as that is not tracked by the fiber itself.

[< Wrapping Types](#)

[Configuration >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Configuration

[< Embedding](#)

[Array C API >](#)

Janet can be configured by editing the source header `janetconf.h`. To build Janet on certain platforms, disable features to save space, or integrated Janet into certain environments, editing this file may be required, although in most cases the default build of Janet should work.

Not all combinations of flags are guaranteed to work, although in most cases they should. You should probably run the test suite on a particular set of flags to verify that they can work together.

# Defines

```
#define JANET_MAJOR 1
```

```
#define JANET_MINOR 2
```

```
#define JANET_PATCH 3
```

```
#define JANET_EXTRA "-dev"
```

```
#define JANET_VERSION "1.2.3-  
dev"
```

Use these defines to set the janet version. The major, minor, and patch values must be in sync with the version string.

```
#define JANET_BUILD "my-local-  
build"
```

Set this to any string such as "local" to set what value `janet/build` will have at runtime. By default, is set to a git commit hash.

## `#define JANET_SINGLE_THREADED`

Define this if you know the interpreter will only run in a single threaded program or context. When this define is set, global Janet state will use normal global C variables rather than thread local variables.

Setting this also disables the `thread/` module.

## `#define JANET_NO_DYNAMIC_MODULES`

Define this to disable loading of dynamic modules via the `native` function. This also lets Janet run on platforms without `dlopen` or Windows, as well as letting Janet be compiled without linking to `-ldl` on Posix.

## `#define JANET_NO_ASSEMBLER`

Define this to disable the functions `asm` and `disasm` in the core library. The assembler is not needed to run most programs, can is useful for inspecting functions, debugging, and writing a compiler for the Janet abstract machine.

## `#define JANET_NO_PEG`

Define this to disable the `peg` module. The core Janet library should work without the peg module, and be slightly smaller, but this flag is not recommended as the peg module itself is actually quite small. This

disables the functions `peg/match` and `peg/compile`.

## `#define JANET_NO_TYPED_ARRAY`

Define this to disable the `tarray` module. Typed arrays are an interface besides buffers for interacting with binary data in a structured manner. The core Janet library does not use them, but they could be used by native modules, especially bindings to scientific or numerical libraries.

## `#define JANET_NO_INT_TYPES`

Define this to disable the `int` module. The `int` module contains abstract types and methods for integers that cannot fit in a Janet number. The only integer types currently support are signed and unsigned 64 bit integers. The current implementation makes this module quite large.

## `#define JANET_REDUCED_OS`

Define this to compile the `os` module with only the bar minimum needed for `boot.janet`. This is just `os/exit`, `os/getenv`, and `os/which`. All other functions in the `os` module will be disabled.

## `#define JANET_API __attribute__((visibility ("default")))`

Set this define to change how Janet symbols in the core library are exported. If using dynamic modules, all symbols in the Janet core library

must be publicly visible. It is not recommended to change this unless you are dramatically changing the build system.

```
#define JANET_OUT_OF_MEMORY do  
{ printf("oops!\n"); exit(1); }  
while (0)
```

Set this macro to decide what Janet will do if it runs out of memory. Be default, the program will print an error message and exit, but you may want Janet to clean up resources or `longjmp` somewhere if using Janet in a larger application.

```
#define JANET_RECURSION_GUARD  
1024
```

Set this to a positive integer to prevent recursive functions in C from causing a segfault via a stack overflow. All recursive functions in Janet use a counter to keep them from overflowing the stack, and will error if the recursion gets too deep. Use this to set how many recursions are allowed before errors. Functions that have such a limitation are `compile` and `marshal`.

```
#define JANET_MAX_PROTO_DEPTH  
200
```

Set this define to determine how many tables to recursively check for prototypes before throwing an error. This is needed as it is easily possible to create tables that form a cyclic prototype loop, which would

otherwise stall the VM if we do not detect it.

```
#define JANET_MAX_MACRO_EXPAND  
200
```

Set this define to limit the number of times a form can be macro expanded. This is also to limit the damage that could be caused by a recursive macro, although even simple recursive macros can stall the VM with even modest recursion limits.

```
#define JANET_STACK_MAX 16384
```

Set the default maximum size of a fiber's stack in number of Janet values. This means that a max stack of 1000 will be able to store approximately 1000 Janet values on the stack, NOT that the fiber can hold 1000 stack frames. This can also be changed on a per fiber basis at runtime, although each new fiber will have stack maximum set to this value by default. Setting a maximum stack size per stack is only useful for more quickly detecting stack overflow issues, but each fiber stack starts with a minimum amount of memory and only expands as needed. This means that setting this value to a lower value will not make Janet use less memory.

```
#define JANET_NO_NANBOX
```

Internally, Janet uses a technique called nanboxing to make each Janet value take less memory and allow the VM to be faster (although Janet mainly uses the technique for the first reason). This technique, however, is not valid C of any kind, and will only work correctly on some

architectures. Many 64 bit architectures will have trouble with the nanboxing code, so it is by default turned off on 64 bit, non x86 architectures. However, one may also want to turn off nanboxing for other reasons, so this flag is exposed for that reason. If an unsupported architecture mistakenly tries to use nanboxing, this could be considered a bug and the platform detection macros in `janet.h` should be updated to disable nanboxing for that platform.

[< Embedding](#)

[Array C API >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Array C API

[< Configuration](#)

[Buffer C API >](#)

As with most data structures, arrays are exposed directly to the programmer in the C API. Users are free to set and get elements in the array, even change the count value of the array to change it's size. It is recommended to use the provided API functions to mutate the array aside from setting a single element, though. For getting and setting values in the array, use `array->data[index]` directly.

# Definition

```
/* A dynamic array type. */
struct JanetArray {
    JanetGCObject gc;
    int32_t count;
    int32_t capacity;
    Janet *data;
};
typedef struct JanetArray JanetArray;
```

# Functions

```
JANET_API JanetArray *janet_array(int32_t capacity);
```

Creates a new, empty array with enough preallocated space for capacity elements.

```
JANET_API JanetArray *janet_array_n(const Janet *elements, int32_t n);
```

Creates a new array and fills it with n elements. The elements are copied into the array via `memcpy`.

```
JANET_API void janet_array_ensure(JanetArray *array, int32_t capacity, int32_t growth);
```

Ensure that an array has enough space for capacity elements. If not, resize the backing memory to `capacity * growth` slots. In most cases, growth should be 1 or 2.

```
JANET_API void janet_array_setcount(JanetArray *array, int32_t count);
```

Set the length of a janet array by setting `array->count = count`. If `count` is greater than the current `array->count`, also lengthens the array by appending nils. Do not pass a negative value to `count`.

```
JANET_API void janet_array_push(JanetArray
```

```
*array, Janet x);
```

Insert a value at the end of an array. May panic if the array runs out of capacity.

```
JANET_API Janet janet_array_push(JanetArray  
*array);
```

Remove an element from the end of the array. Returns the element, or nil if the array is empty.

```
JANET_API Janet janet_array_pop(JanetArray  
*array);
```

Returns the last element in the array, not modifying the array.

[< Configuration](#)

[Buffer C API >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Buffer C API

[< Array C API](#)    [Table C API >](#)

# Definition

```
/* A byte buffer type. Used as a mutable string  
or string builder. */  
struct JanetBuffer {  
    JanetGCObject gc;  
    int32_t count;  
    int32_t capacity;  
    uint8_t *data;  
};  
typedef struct JanetBuffer JanetBuffer;
```

# Functions

```
JANET_API JanetBuffer *janet_buffer(int32_t capacity);
JANET_API JanetBuffer
*janet_buffer_init(JanetBuffer *buffer, int32_t capacity);
JANET_API void janet_buffer_deinit(JanetBuffer
*buffer);
JANET_API void janet_buffer_ensure(JanetBuffer
*buffer, int32_t capacity, int32_t growth);
JANET_API void janet_buffer_setcount(JanetBuffer
*buffer, int32_t count);
JANET_API void janet_buffer_extra(JanetBuffer
*buffer, int32_t n);
JANET_API void
janet_buffer_push_bytes(JanetBuffer *buffer,
const uint8_t *string, int32_t len);
JANET_API void
janet_buffer_push_string(JanetBuffer *buffer,
const uint8_t *string);
JANET_API void
janet_buffer_push_cstring(JanetBuffer *buffer,
const char *cstring);
JANET_API void janet_buffer_push_u8(JanetBuffer
*buffer, uint8_t x);
JANET_API void janet_buffer_push_u16(JanetBuffer
*buffer, uint16_t x);
JANET_API void janet_buffer_push_u32(JanetBuffer
*buffer, uint32_t x);
JANET_API void janet_buffer_push_u64(JanetBuffer
*buffer, uint64_t x);
```

[< Array C API](#)

[Table C API >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Table C API

[< Buffer C API](#)    [Fiber C API >](#)

# Definition

```
/* A mutable associative data type. Backed by a
hashtable. */
struct JanetTable {
    JanetGCObject gc;
    int32_t count;
    int32_t capacity;
    int32_t deleted;
    JanetKV *data;
    JanetTable *proto;
};
typedef struct JanetTable JanetTable;
```

# Functions

```
JANET_API JanetTable *janet_table(int32_t
capacity);
JANET_API Janet janet_table_get(JanetTable *t,
Janet key);
JANET_API Janet janet_table_rawget(JanetTable *t,
Janet key);
JANET_API Janet janet_table_remove(JanetTable *t,
Janet key);
JANET_API void janet_table_put(JanetTable *t,
Janet key, Janet value);
JANET_API const JanetKV
*janet_table_to_struct(JanetTable *t);
JANET_API void janet_table_merge_table(JanetTable
*table, JanetTable *other);
JANET_API void
janet_table_merge_struct(JanetTable *table, const
JanetKV *other);
JANET_API JanetKV *janet_table_find(JanetTable
*t, Janet key);
```

[< Buffer C API](#)

[Fiber C API >](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Fiber C API

[< Table C API](#)    [Janet's Memory Model >](#)

# Definition

```
/* Fiber statuses - mostly corresponds to
signals. */
typedef enum {
    JANET_STATUS_DEAD,
    JANET_STATUS_ERROR,
    JANET_STATUS_DEBUG,
    JANET_STATUS_PENDING,
    JANET_STATUS_USER0,
    JANET_STATUS_USER1,
    JANET_STATUS_USER2,
    JANET_STATUS_USER3,
    JANET_STATUS_USER4,
    JANET_STATUS_USER5,
    JANET_STATUS_USER6,
    JANET_STATUS_USER7,
    JANET_STATUS_USER8,
    JANET_STATUS_USER9,
    JANET_STATUS_NEW,
    JANET_STATUS_ALIVE
} JanetFiberStatus;

/* A lightweight green thread in janet. Does not
correspond to
 * operating system threads. */
struct JanetFiber {
    JanetGCObject gc; /* GC Object stuff */
    int32_t flags; /* More flags */
    int32_t frame; /* Index of the stack frame */
    int32_t stackstart; /* Beginning of next args
*/
    int32_t stacktop; /* Top of stack. Where
```

```
values are pushed and popped from. */
    int32_t capacity;
    int32_t maxstack; /* Arbitrary defined limit
for stack overflow */
    JanetTable *env; /* Dynamic bindings table
(usually current environment). */
    Janet *data;
    JanetFiber *child; /* Keep linked list of
fibers for restarting pending fibers */
};

typedef struct JanetFiber JanetFiber;
```

# Functions

```
JANET_API JanetFiber *janet_fiber(JanetFunction
*callee, int32_t capacity, int32_t argc, const
Janet *argv);
JANET_API JanetFiber
*janet_fiber_reset(JanetFiber *fiber,
JanetFunction *callee, int32_t argc, const Janet
*argv);
JANET_API JanetFiberStatus
janet_fiber_status(JanetFiber *fiber);
JANET_API JanetFiber *janet_current_fiber(void);
```

[< Table C API](#)

[Janet's Memory Model >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Janet's Memory Model

[< Fiber C API](#)

[Writing C Functions >](#)

In order to write a functional and correct C extension to Janet, it is necessary to have a good understanding of Janet's memory model. Code that does not follow this model may leak memory, especially on errors (panics), and may trigger use after free bugs. It is the author's recommendation to test your extension with a tool like [valgrind](#) or an address sanitizer to ensure that your extension does not access memory in an invalid way.

# Janet's Garbage Collector

Like most dynamic languages, Janet has a garbage collector to clean up unused objects. The garbage collector cannot "see" objects allocated via `malloc / free`, only memory allocated through the function `janet_galloc`. It is recommended to not use this function at all, and instead use the Janet C API functions for creating Janet data structures that will be collected when needed. These data structures will be collected when no longer reachable, ensuring that your program does not leak memory. Janet's garbage collector can only run at certain times, meaning most of the time when writing a native function you can ignore it. However, any Janet C API function that may re-enter the interpreter may trigger a collection, which could collect any data structures you have just allocated. The notable exception to this is `janet_call`, which suspends garbage collection until it completes for convenience. Other re-entrant functions, like `janet_pcall`, `janet_continue`, `janet_dobytes`, and `janet_destring` may trigger a collection.

# Adding Janet values to the GC Root list

Janet keeps a global list of objects that are considered reachable. To determine all reachable objects, the garbage collector will traverse this list when needed. Any value not in this list, or not accessible via traversing this list, is considered garbage (with the exception of a few values, such as the current fiber). Notably, this means that values allocated in a C function, by default, are considered garbage and will be collected at the next collection. Most C functions take advantage of the fact that the GC will not run until the interpreter is resumed. However, sometimes it is necessary to add a value to the root list so that a temporary value isn't collected before you are finished with it.

Below is an example C function that adds a value to the root list to prevent the garbage collector from deleting it.

```
Janet make_some_table(int32_t argc, Janet *argv)
{
    janet_fixarity(argc, 0);
    JanetTable *table = janet_table(0);
    Janet tablev = janet_wrap_table(table);

    // Root the table before potentially calling
    // collect, as
    // no live objects reference the table
    janet_gcroot(tablev);
    janet_collect();
    janet_gcunroot(tablev);

    return tablev;
```

```
}
```

This does have some downsides, though. First, it may leak memory if the code between `janet_gcroot` and `janet_gcunroot` can panic. This isn't as bad as it sounds, as most if not all of the functions in the Janet API that can enter the interpreter cannot panic, or will swallow panics and return a signal code instead. It is then up to the user to rethrow any panics caught after releasing (or unrooting) the values. Basically, if you use this method, you MUST clean up after yourself!

This is also why it is recommended to use `janet_call` over `janet_pcall` if you can. Since `janet_call` suspends the garbage collector (including all nested calls to things like `janet_pcall`), there is no need to modify the gcroot list on regular basis. It is also recommended to not use too much re-entrant code, as it will often suspend the garbage collector. For example, if you run your entire application inside a call to `janet_call`, the garbage collector will never run.

# The argv array

Another gotcha in the memory model for the C API is the argument stack to native functions. This pointer to a sequence of Janet values is a pointer into the current fiber where the function arguments were pushed, and has a short lifespan. Any function that may cause the stack to resize invalidates this pointer, and the pointer should be considered invalid after the function returns.

Functions that may cause the stack to resize are any function that modifies a fiber in the fiber API, as well as any function that invokes the interpreter. This means functions like `janet_call` will invalidate the `argv` pointer to your function!

Example of buggy code:

```
Janet my_bad_cfunc(int32_t argc, Janet *argv) {
    for (int32_t i = 0; i < argc; i++) {
        JanetFunction *func =
janet_getfunction(argv, i);
        janet_call(func, 0, NULL);
    }
    return janet_wrap_nil();
}
```

This code is incorrect because it tries to access the `argv` pointer after `janet_call` has been called. The usual fix to this is to copy the `argv` data into a new array that will not be resized if the stack is. A tuple can handily be used for this.

Better code:

```
Janet my_bad_cfunc(int32_t argc, Janet *argv) {
    const Janet *argv2 = janet_tuple_n(argv,
    argv);
    for (int32_t i = 0; i < argc; i++) {
        JanetFunction *func =
janet_getfunction(argv2, i);
        janet_call(func, 0, NULL);
    }
    return janet_wrap_nil();
}
```

# Scratch Memory

Many algorithms will need to allocate memory, but cleaning up the memory can be difficult to correctly integrate with Janet's memory model, considering that many functions in Janet's API can panic, longjumping over your clean up code. Rather than trying to prevent panics, it is much easier to accept that they can happen and allocate resources that can automatically be cleaned up for you.

Janet brings the concept of scratch memory for this purpose, or memory that will certainly be cleaned up at the next collection. This memory can also be freed manually if no error is encountered, making sure that we don't generate garbage in the common case.

```
void *janet_malloc(size_t size);
void *janet_srealloc(void *p, size_t size);
void janet_sfree(void *p)
```

# GC API

```
void janet_mark(Janet x);
void janet_sweep(void);
void janet_collect(void);
void janet_clear_memory(void);
void janet_gcroot(Janet root);
int janet_gcunroot(Janet root);
int janet_gcunrootall(Janet root);
int janet_gclock(void);
void janet_gcunlock(int handle)
```

[< Fiber C API](#)

[Writing C Functions >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)



Janet 1.9.1-4ae3722 Documentation

(Other Versions: [1.8.1](#) [1.7.0](#) [1.6.0](#) [1.5.1](#) [1.5.0](#) [1.4.0](#) [1.3.1](#) )

# Writing C Functions

[< Janet's Memory Model](#)

[The Janet Programming Language >](#)

The most common task in creating native extensions for Janet is writing functions in C that can be called from Janet. These functions, called C Functions, often follow a template to ensure consistent and safe behavior.

1. Assert the arity. This prevents callers from calling the function with the wrong arity
2. Extract arguments. This does type checking on arguments and unwraps Janet values into C values for processing.
3. Function logic.
4. Return value or panic

While not all C functions need to follow this pattern exactly, many functions can written in this manner, and the C API was designed with this use in mind. Most of the core's C functions were written in this style. Functions written like this will also give consistent error messages when called improperly, creating a better experience for users.

# C Function Signature

All C Functions must conform to the `JanetCFunction` type defined in `janet.h`. `argc` is the number of arguments the function was called with, and `argv` is an array of arguments with length `argc`. Any API call that may re-size the stack can invalidate `argv`, so calls to `janet_pcall`, `janet_call`, `janet_continue`, and `janet_fiber_*` should happen after all arguments have been extracted from `argv`.

```
static Janet cfun_myfunc(int32_t argc, Janet *argv);
```

# Asserting Arity

Janet comes with 2 arity-asserting API functions. These functions simply check the value of argc and provide helpful error messages if it is invalid.

```
void janet_arity(int32_t argc, int32_t min,  
int32_t max);  
void janet_fixarity(int32_t argc, int32_t arity);
```

Use `janet_arity` to ensure that the arity is in a range between `min` and `max`, inclusive. A value of `-1` for max will disable the maximum, allowing variadic arguments.

Functions that are single arity should use `janet_fixarity`, which is equivalent to `janet_arity(argc, arity, arity)`, but provides a slightly nicer error message.

# Extracting Arguments

For many wrappers, the most laborious task is safely converting Janet values into C types in a safe and flexible manner. We would like to verify that all arguments are of the correct type, and only then unwrap them and assign them to C variables. The C API provides a number of utility functions that both typecheck and unwrap values in one operation, making this boilerplate a bit more natural.

These utility functions will have the signature `type janet_get##type(const Janet *argv, int32_t n)`, and extract the nth argument passed to `argv` while asserting the argument is a compatible type and converting it for you. If the argument is the wrong type, a descriptive error will be raised, indicating a type error in the nth argument. These functions will not bounds check the `argv` array, so be sure to check your arity before calling these functions.

## Getter functions

```
double janet_getnumber(const Janet *argv, int32_t n);
JanetArray *janet_getarray(const Janet *argv,
int32_t n);
const Janet *janet_gettuple(const Janet *argv,
int32_t n);
JanetTable *janet_gettable(const Janet *argv,
int32_t n);
const JanetKV *janet_getstruct(const Janet *argv,
int32_t n);
const uint8_t *janet_getstring(const Janet *argv,
```

```
int32_t n);
const char *janet_getcstring(const Janet *argv,
int32_t n);
const uint8_t *janet_getsymbol(const Janet *argv,
int32_t n);
const uint8_t *janet_getkeyword(const Janet
*argv, int32_t n);
JanetBuffer *janet_getbuffer(const Janet *argv,
int32_t n);
JanetFiber *janet_getfiber(const Janet *argv,
int32_t n);
JanetFunction *janet_getfunction(const Janet
*argv, int32_t n);
JanetCFunction janet_getcfunction(const Janet
*argv, int32_t n);
int janet_getboolean(const Janet *argv, int32_t
n);
void *janet_getpointer(const Janet *argv, int32_t
n);

int32_t janet_getnat(const Janet *argv, int32_t
n);
int32_t janet_getinteger(const Janet *argv,
int32_t n);
int64_t janet_getinteger64(const Janet *argv,
int32_t n);
size_t janet_getsize(const Janet *argv, int32_t
n);
JanetView janet_getindexed(const Janet *argv,
int32_t n);
JanetByteView janet_getbytes(const Janet *argv,
int32_t n);
JanetDictView janet_getdictionary(const Janet
*argv, int32_t n);
void *janet_getabstract(const Janet *argv,
int32_t n, const JanetAbstractType *at);
JanetRange janet_getslice(int32_t argc, const
```

```

Janet *argv);
int32_t janet_gethalfrange(const Janet *argv,
int32_t n, int32_t length, const char *which);
int32_t janet_getargindex(const Janet *argv,
int32_t n, int32_t length, const char *which);
uint64_t janet_getflags(const Janet *argv,
int32_t n, const char *flags);

```

Another common pattern is to allow nil for some arguments, and use a default value if nil or no value is provided. C Functions should try to be consistent with the idea that not providing an optional argument is the same as setting it to nil. API functions in the family `type`

`janet_opt##type(const Janet *argv, int32_t argc,`  
`int32_t n, type dflt)` work similar to the normal argument getter functions. Note that the signature is different as they take in the value of `argc`. These functions perform bounds checks on the argument array (assuming `n >= 0`), which is why they take in `argc` as a parameter.

```

double janet_optnumber(const Janet *argv, int32_t
argc, int32_t n, double dflt);
const Janet *janet_opttuple(const Janet *argv,
int32_t argc, int32_t n, const Janet *dflt);
const JanetKV *janet_optstruct(const Janet *argv,
int32_t argc, int32_t n, const JanetKV *dflt);
const uint8_t *janet_optstring(const Janet *argv,
int32_t argc, int32_t n, const uint8_t *dflt);
const char *janet_optcstring(const Janet *argv,
int32_t argc, int32_t n, const char *dflt);
const uint8_t *janet_optsymbol(const Janet *argv,
int32_t argc, int32_t n, const uint8_t *dflt);
const uint8_t *janet_optkeyword(const Janet
*argv, int32_t argc, int32_t n, const uint8_t
*dflt);
JanetFiber *janet_optfiber(const Janet *argv,
int32_t argc, int32_t n, JanetFiber *dflt);

```

```
JanetFunction *janet_optfunction(const Janet
*argv, int32_t argc, int32_t n, JanetFunction
*dflt);
JanetCFunction janet_optcfunction(const Janet
*argv, int32_t argc, int32_t n, JanetCFunction
dflt);
int janet_optboolean(const Janet *argv, int32_t
argc, int32_t n, int dflt);
void *janet_optpointer(const Janet *argv, int32_t
argc, int32_t n, void *dflt);
int32_t janet_optnat(const Janet *argv, int32_t
argc, int32_t n, int32_t dflt);
int32_t janet_optinteger(const Janet *argv,
int32_t argc, int32_t n, int32_t dflt);
int64_t janet_optinteger64(const Janet *argv,
int32_t argc, int32_t n, int64_t dflt);
size_t janet_optsize(const Janet *argv, int32_t
argc, int32_t n, size_t dflt);
void *janet_optabstract(const Janet *argv,
int32_t argc, int32_t n, const JanetAbstractType
*at, void *dflt);

/* Mutable optional types specify a size default,
and
 * construct a new value if none is provided */
JanetBuffer *janet_optbuffer(const Janet *argv,
int32_t argc, int32_t n, int32_t dflt_len);
JanetTable *janet_opttable(const Janet *argv,
int32_t argc, int32_t n, int32_t dflt_len);
JanetArray *janet_optarray(const Janet *argv,
int32_t argc, int32_t n, int32_t dflt_len);
```

# Panicking

If at any point during execution of the C function an error is hit, you may want to abort execution and raise an error. Errors should not be expected to be frequent, but they should not segfault the program, leak memory, or invoke undefined behavior. The C API provides a few `janet_panic` functions that will abort execution and jump back to a specified error point (the last call to `janet_continue`).

```
void janet_panicv(Janet message);
void janet_panic(const char *message);
void janet_panics(const uint8_t *message);
void janet_panicf(const char *format, ...);
```

Use `janet_panicf` to create descriptive, formatted error messages for functions. The format string is much like that of `printf`, but is specialized for working with Janet values. The pretty printing formatters can take a precision argument to specify the maximum nesting depth to print.

| Formatter | Description                                | Type   |
|-----------|--------------------------------------------|--------|
| %o        | print an octal integer                     | long   |
| %x        | print a hexadecimal integer<br>(lowercase) | long   |
| %X        | print a hexadecimal integer<br>(uppercase) | long   |
| %f        | print a double                             | double |
|           |                                            |        |

|    |                                                                          |                    |
|----|--------------------------------------------------------------------------|--------------------|
| %d | print an integer                                                         | long               |
| %i | print an integer (same as d)                                             | long               |
| %S | print a NULL-terminated C string                                         | const<br>char *    |
| %f | print a floating point number                                            | double             |
| %A | print a hexidecimal floating point<br>number (uppercase)                 | double             |
| %a | print a hexidecimal floating point<br>number (lowercase)                 | double             |
| %E | print a number in scientific notation                                    | double             |
| %g | print a number in its shortest form                                      | double             |
| %G | print a number in its shortest form<br>(uppercase)                       | double             |
| %S | print a Janet string-like                                                | const<br>uint8_t * |
| %t | print the type of a Janet value                                          | Janet              |
| %T | print a Janet type                                                       | JanetType          |
| %v | print a Janet value with (describe<br>x)                                 | Janet              |
| %V | print a Janet value with (tostring x)                                    | Janet              |
| %j | print a Janet value in jdn format.<br>Will fail on cyclic structures and | Janet              |

|    |                                                                   |       |
|----|-------------------------------------------------------------------|-------|
|    | special types                                                     |       |
| %p | pretty print a Janet value                                        | Janet |
| %P | pretty print a Janet value with color                             | Janet |
| %q | pretty print a Janet value on one line                            | Janet |
| %Q | pretty print a Janet value on one line with color                 | Janet |
| %m | pretty print a Janet value (no truncation)                        | Janet |
| %M | pretty print a Janet value (no truncation) with color             | Janet |
| %n | pretty print a Janet value (no truncation) on one line            | Janet |
| %N | pretty print a Janet value (no truncation) on one line with color | Janet |

# Scratch Memory

Panicking is implemented with C's `setjmp` and `longjmp`, which means that resources are not cleaned up when a panic happens. To avoid leaking memory, stick to using Janet's data structures, which are garbage collected, or use Janet's scratch memory API to allocate memory. The scratch memory API exposes functions similar to `malloc` and `free`, but the memory will be automatically freed on the next garbage collection cycle if `janet_sfree` is not called.

```
void *janet_malloc(size_t size);
void *janet_srealloc(void *mem, size_t size);
void janet_sfree(void *mem);
```

# Example

```
/* Set a value in an array with an index that
wraps around */
static Janet cfun_array_ringset(int32_t argc,
Janet *argv) {

    /* Ensure 3 arguments */
    janet_fixarity(argc, 3);

    /* Extract arguments */
    JanetArray *array = janet_getarray(argv, 0);
    int64_t index = janet_getinteger64(argv, 1);
    if (index < 0) {
        janet_panicf("expected non-negative 64
bit integer, got %v",
                      argv[1]);
    }

    /* Set array[index % count] = argv[2] */
    int64_t count64 = (int64_t) array->count;
    int64_t mod_index = index % count64;
    array->data[mod_index] = argv[2];

    /* Return the original array */
    return janet_wrap_array(array);
}
```

[< Janet's Memory Model](#)

[The Janet Programming Language >](#)

© Calvin Rose & contributors 2020

Generated on May 31, 2020 at 05:44:10 (UTC) with Janet 1.9.1-4ae3722

See a problem? Source [on GitHub](#)