The Janet Abstract Machine

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

The Janet Abstract Machine

Prototypes The Event Loop

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 of registers.

```
X: Slot

X
X - Stack Top, for next function call.
-----
Frame next
-----
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 frame, 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
O - Opcode bits

   4    3    2    1
+----+----+----+----+
| XX | XX | XX | OO |
+----+----+----+----+
```

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 '$' prefix indicates that an instruction parameter is acting as a virtual register (slot). If a parameter does not have the '$' 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)
`cncl`	`(cncl dest fiber err)`	Resume fiber, but raise error immediately

Instruction	Signature	Description
div	(div dest lhs rhs)	$dest = $lhs / $rhs
divf	(divf dest lhs rhs)	$dest = floor($lhs / $rhs)
divim	(divim dest lhs im)	$dest = $lhs / im
eq	(eq dest lhs rhs)	$dest = $lhs == $rhs
eqim	(eqim dest lhs im)	$dest = $lhs == im
err	(err message)	Throw error $message.
get	(get dest ds key)	$dest = $ds[$key]
geti	(geti dest ds index)	$dest = $ds[index]
gt	(gt dest lhs rhs)	$dest = $lhs > $rhs
gte	(gte dest lhs rhs)	$dest = $lhs >= $rhs
gtim	(gtim dest lhs im)	$dest = $lhs > im
in	(in dest ds key)	$dest = $ds[$key] using `in`
jmp	(jmp offset)	pc += offset
jmpif	(jmpif cond offset)	if $cond pc += offset else pc++
jmpni	(jmpni cond offset)	if $cond == nil pc += offset else pc++
jmpnn	(jmpnn cond offset)	if $cond != nil pc += offset else pc++
jmpno	(jmpno cond offset)	if $cond pc++ else pc += offset
ldc	(ldc dest index)	$dest = constants[index]
ldf	(ldf dest)	$dest = false
ldi	(ldi dest integer)	$dest = integer
ldn	(ldn dest)	$dest = nil
lds	(lds dest)	$dest = current closure (self)
ldt	(ldt dest)	$dest = true
ldu	(ldu dest env index)	$dest = envs[env][index]
len	(len dest ds)	$dest = length(ds)
lt	(lt dest lhs rhs)	$dest = $lhs < $rhs
lte	(lte dest lhs rhs)	$dest = $lhs <= $rhs
ltim	(ltim dest lhs im)	$dest = $lhs < im
mkarr	(mkarr dest)	$dest = call(array, args)
mkbtp	(mkbtp dest)	$dest = call(tuple/brackets, args)
mkbuf	(mkbuf dest)	$dest = call(buffer, args)
mkstr	(mkstr dest)	$dest = call(string, args)
mkstu	(mkstu dest)	$dest = call(struct, args)
mktab	(mktab dest)	$dest = call(table, args)
mktup	(mktup dest)	$dest = call(tuple, args)
mod	(mod dest lhs rhs)	$dest = $lhs mod $rhs
movf	(movf src dest)	$dest = $src
movn	(movn dest src)	$dest = $src
mul	(mul dest lhs rhs)	$dest = $lhs * $rhs
mulim	(mulim dest lhs im)	$dest = $lhs * im
neq	(neq dest lhs rhs)	$dest = $lhs != $rhs

Instruction	Signature	Description
neqim	(neqim dest lhs im)	$dest = $lhs != $im
next	(next dest ds key)	$dest = next($ds, $key)
noop	(noop)	Does nothing.
prop	(prop dest val fiber)	Propagate (Re-raise) a signal that has been caught.
push	(push val)	Push $val on args
push2	(push2 val1 val2)	Push $val1, $val2 on args
push3	(push3 val1 val2 val3)	Push $val1, $val2, $val3, on args
pusha	(pusha array)	Push values in $array on args
put	(put ds key val)	$ds[$key] = $val
puti	(puti ds val index)	$ds[index] = $val
rem	(rem dest lhs rhs)	$dest = $lhs % $rhs
res	(res dest fiber val)	$dest = resume $fiber with $val
ret	(ret val)	Return $val
retn	(retn)	Return nil
setu	(setu val env index)	envs[env][index] = $val
sig	(sig dest value sigtype)	$dest = emit $value as sigtype
sl	(sl dest lhs rhs)	$dest = $lhs << $rhs
slim	(slim dest lhs shamt)	$dest = $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
subim	(subim dest lhs im)	$dest = $lhs - im
tcall	(tcall callee)	Return call($callee, args)
tchck	(tchck slot types)	Assert $slot matches types

Prototypes The Event Loop
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures

- Fiber C API
- Memory Model
- Writing C Functions

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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)

Comparison Operators 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.

Comparison Operators Flow
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Comparison Operators

Numbers and Arithmetic Bindings (def and var)
Comparison operators are used for comparing values in Janet, in order to establish equality or ordering. Janet has two types of comparison operators, which we refer to as "primitive" comparison operators, and "polymorphic" comparison operators, respectively.

Primitive Comparison Operators

The primitive comparison operators are =, <, <=, >, >=. In the simple case, each operator can be used to compare two values as follows: `(< a b)` will return true if a < b, and false otherwise. Similarly `(= a b)` will be true only if a == b.

```
(= 3 3)          # true
(< 1 3)          # true
(>= :a :a)       # true
(> "bar" "foo")  # false -- strings compare lexicographically
(<= :bar :foo)   # true -- keywords compare lexicographically by keyword name
(= nil nil)      # true -- nil always equal to itself and only itself
```

More generally, each of these operators can take any number of arguments (from 0) and will return true if the arguments do not violate the ordering implied by the operator. So `(< 1 2 3)` returns true but `(> 3 2 4)` returns false. As an extreme, `(<)` returns true since its (nonexistent) arguments do not violate the ordering.

```
(= 1 1 1)  # true
(< 1 3 5)  # true
(>= 3 1 7) # false
(> 1)      # true
```

The primitive comparison operators provide a total ordering for all Janet types, but importantly, these operators compare values of different types in a way that the user might find surprising. If two arguments of a primitive comparison are of different types, they will be ordered by Janet's internal type number. This is not necessarily what the user needs, for example, when comparing Janet int/s64 types to Janet numbers.

```
# surprisingly evaluates to false:
(= (int/s64 1) (int/u64 1))
# surprisingly evaluates to true but this due to Janet internal type number
# for int/u64 types being greater than the internal type number for numbers!
(< 3 (int/u64 2))
```

If you require comparison between types to be ordered by something other than type number (e.g. "numeric value") then, use the polymorphic comparison operators, described below.

Polymorphic Comparison Operators

The polymorphic comparison operators are used for comparing different types in some manner rather than just by internal Janet type number. The semantics are determined by the types involved. The purpose of these operators is to allow comparison to work in some "less surprising" way (relative to the primitive comparison operators) when two types have a natural ordering between them.

The polymorphic comparison operators are `compare=`, `compare<`, `compare<=`, `compare>`, `compare>=`. In general, they work similarly to the primitive operators:

```
(compare< 1 3)          # true
(compare> "bar" "foo")  # false -- strings compare lexicographically
(compare<= :bar :foo)   # true -- keywords compare lexicographically by keyword name
(compare= nil nil)      # true -- nil always equal to itself and only itself
(compare< 1 2 3)        # true -- just like <
```

However, when comparing between int/s64, int/u64, and number types, these operators will "do the right thing".

```
(compare= (int/s64 1) (int/u64 1))  # true -- they are "semantically" equal
(compare< 3 (int/u64 2))            # false -- semantically 3 is not < 2
```

In general the polymorphic operators are slower than the primitive ones, so use the primitive ones unless you need the extra polymorphic features.

Implementing Polymorphic Comparison

If you just want to use the polymorphic comparison for the built in types, you can skip this section, which is about how to implement polymorphic comparison for your own types.

The polymorphic comparison operators all use a function called `compare` to establish an ordering between janet values. The `compare` function compares two values (here called `a` and `b`), and returns -1, 0, or 1 for `a < b`, `a = b`, `a > b` respectively. This result is used by the comparison operator, like `compare<`, to return false or true. (The comparison operators are extended to work for multiple arguments using multiple calls to `compare`). The algorithm for the `compare` function is as follows:

- if `a` implements `:compare` and `(:compare a b)` is not nil, then return `(:compare a b)`
- if `b` implements `:compare` and `(:compare b a)` is not nil, then return `(- (:compare a b))`
- else return `(cond (< a b) -1 (= a b) 0 1)`

Since compare defers to the primitive operators as a last resort, the polymorphic comparison operators produce a total ordering of Janet types.

The compare method on an abstract type can be implemented in C, and the compare method for a table-based "object" can be implemented in Janet. For more information on the latter see the object oriented programming section and the prototypes section for more information on developing object oriented methods on tables.

Deep Equality Operator

The primitive = operator does not compare the contents of mutable structures (arrays, tables, and buffers). Instead, it checks if the two values are the same object.

```
# Even though they have the same contents these are considered *not* equal
(assert (not= @{:a 1} @{:a 1}))
(assert (not= @[:a :b] @[:a :b]))
(assert (not= @"abc" @"abc"))

# primitive equal will return true if they are same object
(let [x @{:a 1}]
  (assert (= x x)))

# NOTE: = works as expected on immutable structures
```

```
(assert (= [:a :b] [:a :b]))
(assert (= {:a 1} {:a 1}))
(assert (= "abc" "abc"))
```

To compare the contents of mutable structures use the `deep=` function. This function compares buffers like strings, and recursively compares arrays and tables.

```
(assert (deep= [:a :b] [:a :b]))
(assert (deep= {:a 1} {:a 1}))
(assert (deep= "abc" "abc"))

(assert (deep=
  @{ :x @[@{:a 1} 3 @"hi"] :y 2 }
  @{ :x @[@{:a 1} 3 @"hi"] :y 2 }))

# NOTE deep= uses primitive = when the values are not an array, table or buffer.
(assert (deep-not= @{ :x (int/s64 1) } @{ :x (int/u64 1) }))
```

Numbers and Arithmetic Bindings (def and var)
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs
    - Tuples
  - Destructuring
  - Fibers
    - Dynamic Bindings
    - Errors
  - Modules
  - Object-Oriented Programming
  - Parsing Expression Grammars
  - Prototypes
  - The Janet Abstract Machine
  - The Event Loop
  - Multithreading
  - Networking

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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` takes 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 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- buffer
- bundle
- debug
- ev
- ffi
- fiber
- file
- int
- jpm
    - rules
    - cc
    - cgen
    - cli
    - commands
    - config
    - make-config
    - dagbuild
    - pm
    - scaffold
    - shutil
- math
- module
- net
- os
- peg
- parser
- spork
    - argparse
    - base64
    - cc
    - cjanet
    - crc
    - channel
    - cron
    - data
    - ev-utils
    - fmt
    - generators
    - getline
    - htmlgen
    - http
    - httpf
    - infix
    - json
    - mdz
    - math
    - misc
    - netrepl
    - pgp
    - build-rules
    - path

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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)) # -> @"abcdefg", 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\0a"
```

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

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

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

See the documentation for these functions in the Buffer API.

Arrays Tables
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays

- Memory Model
- Writing C Functions

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Data Structures

Macros Arrays

Once you have a handle on functions and the primitive value types, you may be wondering how to work with collections of things. Janet has a small number of core data structure types that are very versatile. Tables, structs, arrays, tuples, strings, and buffers, are the 6 main built-in data structure types. These data structures can be arranged in a useful table describing their relationship to each other.

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 with an integer index. Indexed types are backed by a single chunk of memory for fast access, and are indexed from 0 as in C. Dictionary types associate keys with values. The 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 for 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 integer values between 0 and 255 (the byte values). In this way, they behave much like arrays and tuples. However, one cannot put non-integer values into a byteseq.

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

Data Structure	Access	Insert/Append	Delete	Space Complexity	Mutable
Array *	O(1)	O(1)	O(1)	O(n)	Yes
Tuple	O(1)	-	-	O(n)	No
Table	O(1)	O(1)	O(1)	O(n)	Yes
Struct	O(1)	-	-	O(n)	No
Buffer	O(1)	O(1)	O(1)	O(n)	Yes
String/Keyword/Symbol	-	-	-	O(n)	No

*: 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 my-tuple (tuple 1 2 3))

(def my-array @(1 2 3))
(def my-array (array 1 2 3))

(def my-struct {
 :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` or `in` functions. 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
(in  {:a 1} :a)  # -> 1
(get @[:a :b :c] 2) # -> :c
(in  @[:a :b :c] 2) # -> :c
(get (tuple "a" "b" "c") 1) # -> "b"
(get @"hello, world" 1) # -> 101
(get "hello, world" 1)  # -> 101
(in  "hello, world" 1)  # -> 101
(get {:a :b} :a)    # -> :b
(get {:a :b} :c :d) # -> :d
(in  {:a :b} :c :d) # -> :d
```

The `in` function (added in v1.5.0) for tables and structs behaves identically to `get`. However, the `in` function for arrays, tuples, and string-likes will throw errors on bad keys --- so to better detect errors, prefer `in` for these.

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

To update a mutable data structure, use the `put` function. It takes 3 arguments, the data structure, the key, and the value, and returns the data structure. The allowed types of keys 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 of the data structure will extend the data structure and fill it with `nil`s in the case of the array, or `0`s in the case of the buffer.

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

Macros Arrays
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
    - Syntax and the Parser
    - Special Forms

- ◊ dagbuild
- ◊ pm
- ◊ scaffold
- ◊ shutil

- ♦ math
- ♦ module
- ♦ net
- ♦ os
- ♦ peg
- ♦ parser
- ♦ spork
  - ◊ argparse
  - ◊ base64
  - ◊ cc
  - ◊ cjanet
  - ◊ crc
  - ◊ channel
  - ◊ cron
  - ◊ data
  - ◊ ev-utils
  - ◊ fmt
  - ◊ generators
  - ◊ getline
  - ◊ htmlgen
  - ◊ http
  - ◊ httpf
  - ◊ infix
  - ◊ json
  - ◊ mdz
  - ◊ math
  - ◊ misc
  - ◊ netrepl
  - ◊ pgp
  - ◊ build-rules
  - ◊ path
  - ◊ randgen
  - ◊ rawterm
  - ◊ regex
  - ◊ rpc
  - ◊ schema
  - ◊ sh
  - ◊ msg
  - ◊ stream
  - ◊ tasker
  - ◊ temple
  - ◊ test
  - ◊ tarray
  - ◊ utf8
  - ◊ zip
- ♦ string
- ♦ table

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- ffi
- fiber
- file
- int
- jpm
    - rules
    - cc
    - cgen
    - cli
    - commands
    - config
    - make-config
    - dagbuild
    - pm
    - scaffold
    - shutil
- math
- module
- net
- os
- peg
- parser
- spork
    - argparse
    - base64
    - cc
    - cjanet
    - crc
    - channel
    - cron
    - data
    - ev-utils
    - fmt
    - generators
    - getline
    - htmlgen
    - http
    - httpf
    - infix
    - json
    - mdz
    - math
    - misc
    - netrepl
    - pgp
    - build-rules
    - path
    - randgen
    - rawterm
    - regex
    - rpc

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 `math/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 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs
    - Tuples
  - Destructuring
  - Fibers

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Tuples

Structs Destructuring
Tuples are immutable, sequential types that are similar to arrays. They are represented in source code with either parentheses or brackets surrounding their items. Note that Janet differs from traditional Lisps here, which commonly use the term "lists" to describe forms wrapped in parentheses. 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 Destructuring
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
    - ♦ Syntax and the Parser
    - ♦ Special Forms
    - ♦ Numbers and Arithmetic
    - ♦ Comparison Operators
    - ♦ Bindings (def and var)
    - ♦ Flow
    - ♦ Functions
    - ♦ Strings, Keywords, and Symbols
    - ♦ Looping
    - ♦ Macros
    - ♦ Data Structures
        - ◊ Arrays
        - ◊ Buffers
        - ◊ Tables
        - ◊ Structs
        - ◊ Tuples
    - ♦ Destructuring

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Destructuring

Tuples Fiber Overview

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

# You can also collect trailing destructured values into a "more"
# tuple, though note that it copies, and so may be costly.
(def [p q & more] my-array)
(print p) # :mary
(print q) # :had
(pp more) # (:a :little :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.

Tuples Fiber Overview

© Calvin Rose & contributors 2025

Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3

See a problem? Source on GitHub

- Home
- Documentation
    - ♦ Syntax and the Parser
    - ♦ Special Forms
    - ♦ Numbers and Arithmetic
    - ♦ Comparison Operators
    - ♦ Bindings (def and var)
    - ♦ Flow
    - ♦ Functions
    - ♦ Strings, Keywords, and Symbols
    - ♦ Looping
    - ♦ Macros
    - ♦ Data Structures
        - ◊ Arrays

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Documentation

Spawning a Process jpm
Documenting code is an important way to communicate to users how and when to use a particular piece of functionality.

In many languages, documentation is associated with code through convention (e.g. a comment that comes immediately before a function definition describing the function, its parameters and the return value). In contrast, Janet includes first-class support for documentation through the use of docstrings.

Detecting Docstrings

A docstring is a string that Janet associates with a particular binding in an environment. Janet detects docstrings by looking at the elements in the binding, if a string is located at a particular position, it is used as the docstring.

```
(defn my-function
  "This function adds two to the argument."
  [x]
  (+ x 2))
```

In the above example, Janet treats the string `"This function adds two to the argument."` as the docstring. When creating the binding associated with the symbol `my-function`, Janet will add a key `:doc` with the docstring as its value.

In the case of function and macro definitions, the string that occurs before the tuple of arguments is used as the docstring. In other cases, the string that occurs before the final value is used as the docstring. The string should come after any tags (e.g. `:private`) that are associated with the binding.

```
(def
  my-binding       # the symbol used to name the binding
  :my-tag          # one or more tags that are associated with the binding
  "The docstring"  # the docstring associated with the binding
  42)              # the value associated with the binding
```

Accessing Docstrings

As described above, docstrings are added to a binding under the `:doc` keyword. The docstrings can be accessed from a binding using the `(dyn)` function and accessing the value associated with `:doc`.

```
(defn my-function
  "This function adds two to the argument."
  [x]
  (+ x 2))

(get (dyn 'my-function) :doc)
```

Note that when a docstring is displayed with `(doc)`, Janet adds additional information to the docstring before displaying it. To use this modified string, wrap the call to `(doc)` in a call to `(with-dyns)`.

```
(def b @"")
(with-dyns [:out b]
```

```
(doc my-function))
```

Using Long-Strings

Janet provides two literal forms for expressing a strings: ordinary strings (that is, a sequence of characters delimited by the double quote character, `"`) and long-strings (that is, a sequence of characters delimited by one or more backquote characters, `` ` ``).

While either form can be used for docstrings, long-strings have two advantages over ordinary strings. First, new lines are preserved. This makes it simple to write readable strings in code. Second, Janet will automatically removed indentation (so-called "dedenting") for whitespace that appears before the column in which the long-string began. This is best seen with an example.

```
(defn my-second-function
  ``This function adds three to the argument.

  Note that unlike `my-function` this function returns the value as a string.
  ``
  [x]
  (string (+ 3 x)))
```

Formatting with Markdown

Docstrings are typically read in a plaintext environment. Formatting systems like Markdown are a natural fit for these situations.

Janet's built-in documentation viewer, `(doc)`, understands a subset of Markdown and will indent docstrings that use this subset in an intelligent way. The subset of Markdown includes:

- numbered and unnumbered lists
- codeblocks
- blockquotes

Other Markdown-formatted text (e.g. code spans) are simply treated as ordinary text.

Adding Docstrings to Modules

In addition to documenting bindings, documentation can be added to a file using `(setdyn)`.

```
# my-module.janet

(setdyn :doc "This is the docstring for my-module")

(defn plus-two [x] (+ x 2))
```

At the REPL, this docstring can be retrieved using the path to the module:

```
repl:1:> (import ./my-module)
@{my-module/plus-two @{:private true} _ @{:value <cycle 0>}}
repl:2:> (doc ./my-module)


    module (source)
```

```
my-module.janet

This is the docstring for my-module.


nil
```

Spawning a Process jpm
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs
    - Tuples
  - Destructuring
  - Fibers
    - Dynamic Bindings
    - Errors
  - Modules
  - Object-Oriented Programming
  - Parsing Expression Grammars
  - Prototypes
  - The Janet Abstract Machine
  - The Event Loop
  - Multithreading
  - Networking
  - Process Management
    - Executing
    - Spawning
  - Documentation
  - jpm
  - Linting
  - Foreign Function Interface
- API
  - array
  - buffer

- bundle
- debug
- ev
- ffi
- fiber
- file
- int
- jpm
    - rules
    - cc
    - cgen
    - cli
    - commands
    - config
    - make-config
    - dagbuild
    - pm
    - scaffold
    - shutil
- math
- module
- net
- os
- peg
- parser
- spork
    - argparse
    - base64
    - cc
    - cjanet
    - crc
    - channel
    - cron
    - data
    - ev-utils
    - fmt
    - generators
    - getline
    - htmlgen
    - http
    - httpf
    - infix
    - json
    - mdz
    - math
    - misc
    - netrepl
    - pgp
    - build-rules
    - path
    - randgen

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

The Event Loop

The Janet Abstract Machine Multithreading
Janet comes with a powerful concurrency model out of the box - the event loop. The event loop provides concurrency within a single thread by allowing cooperating fibers to yield instead of blocking forward progress. This is a form of cooperative multi-threading that can be useful in many applications, like situations where there are many concurrent, IO-bound tasks. This means tasks that spend most of their execution time waiting for network or disk, rather than using CPU time.

Most event loop functionality in Janet is exposed through the `ev/` module, but many other functions outside this module will interact with the event loop.

An Overview of the Event Loop

Janet is by no means the first language to provide an event loop, and most languages or tools that provide them work in the same general way.

All programs in Janet are wrapped in an implicit loop that will run until all tasks are complete. The following is pseudo-code describing how the event loop works - the event loop code is all actually implemented directly in the runtime for efficiency.

```
# Functions that yield to the event loop will put (fiber/root) here.
(def pending-tasks @[])

# Pseudo-code of the event loop scheduler
(while (not (empty? pending-tasks))
  (def [root-fiber data] (wait-for-next-event))
  (resume root-fiber data))
```

Each `root-fiber`, or task, is a fiber that will be automatically resumed when an event (or sequence of events) that it is waiting for occurs. Generally, one should not manually resume tasks - the event loop will call `resume` when the completion event occurs.

To be precise, a task is just any fiber that was scheduled to run by the event loop. Therefore, all tasks are fibers, but not all fibers are tasks. You can get the currently executing task in Janet with `(fiber/root)`.

Blocking the Event Loop

When using the event loop, it is important to be aware that CPU bound loops will block all other tasks on the event loop. For example, an infinite while loop that does not yield to the event loop will bring the entire thread to a halt! This is generally considered to be one of the biggest drawbacks to cooperative multi-threading, so be cautious and do not mix in blocking functions into a program that uses the event loop. Many functions in Janet's core library will not block when the event loop is enabled, but a few will. All functions in the `file/` module will block, so be careful when using these with networked files. These functions may be changed or deprecated in the future. Other blocking functions include `os/sleep` and `getline`.

NOTE: This means that getting input from the repl will block the event loop! Be careful of this when trying to use `ev/` functions from the built in repl. Instead, you can use a stream-based repl such as `spork/netrepl` which will interact well with the event loop.

Creating Tasks

A default Janet program has a single task that will run until complete. To create new tasks, Janet provides two built-in functions - `ev/go` and `ev/call`. `ev/call` is implemented in terms of `ev/go`, and other macros and functions are wrappers around these two functions.

```
(defn worker
  "Does some work."
  [name n]
  (for i 0 n
    (print name " working " i "...")
    (ev/sleep 0.5))
  (print name " is done!"))

# Start bob working in a new task with ev/call
(ev/call worker "bob" 10)

(ev/sleep 0.25)

# Start sally working in a new task with ev/go
(ev/go (fiber/new |(worker "sally" 20)))

(ev/sleep 11)
(print "Everyone should be done by now!")
```

This example will start two worker tasks that will run concurrently with each other. The `ev/call` function takes a function as well as its arguments and calls that function inside a new task. `ev/call` returns the newly created task fiber immediately.

The more general way to start new tasks is `ev/go`, which takes as an argument a fiber to be resumed by the event loop scheduler. `ev/go` allows the programmer to customize fiber flags as well as specify a supervisor channel for the newly created task.

Lastly, there is the `ev/spawn` macro for concisely running a series of forms in a new task.

Task Communication

To communicate between tasks, the `ev/` module offers two abstractions: streams and channels. Both abstraction works as FIFO (First In, First Out) data structures, but operate on different kinds of data. Channels allow the programmer to communicate by sending any Janet value as messages, and only work inside a thread - they do not allow communication between threads, processes, or over the network. Streams are wrappers around file descriptors and operate on streams of bytes. Streams can communicate across threads, processes, and across the network.

With these constraints, channels are preferred for things like internal queues and between-task communication, where streams are useful for most everything else.

Channels

Channels are based around a queue abstraction, where producer tasks will add items to the channel with `ev/give`, and consumer tasks will remove items from the channel with `ev/take`. If there are no items in the channel when `ev/take` is called, the consumer will suspend until a producer gives an item to the channel. Similarly, if a task gives to a full channel, it will suspend until a consumer takes an item from the

queue, freeing up space for the producer to give an item to the channel.

Internally, channels are infinitely expanding queues, so one might think that they never need to be "full", but allowing channels to fill up and suspend writers allows for back pressure. Back pressure means that a task can indicate to other tasks that it is "backed up", and that they should stop sending it "work".

```
# Create a new channel with a limit of 10 items.
(def channel (ev/chan 10))

# Write to the channel forever
(ev/spawn
  (forever
    (ev/give channel (math/random))))

(defn consumer
  "Take from the channel forever."
  [name delay]
  (forever
    (def item (ev/take channel))
    (print name " got item " item)
    (ev/sleep delay)))

(ev/call consumer "bob" 1)
(ev/call consumer "sally" 0.63)

# Call (ev/sleep 1) to see some results
```

Streams

Streams let Janet do IO without blocking. They provide a very similar interface to files, with a few extra caveats and capabilities. There are several ways to acquire streams, such as using the `net/` module, `os/open`, and `os/pipe`.

To read from a stream, use `ev/read` or `ev/chunk`. These can also be accessed with `:read` and `:chunk` methods on streams.

To write to a stream, use the function `ev/write`. This can be referenced as the `:write` method on streams as well.

For some examples on how to use streams, see documentation on networking in Janet.

Cancellation

Sometimes, IO operations can take too long or even hang indefinitely. Janet offers `ev/cancel` to interrupt and cancel an ongoing IO operation. This will cause the canceled task to be resumed but immediately error. From the point of view of the canceled task, it will look as though the last function that yielded to the event loop raised an error.

Macros like `try` and `protect` will continue to work just as if the cancellation error was any other error.

```
(def f
  (ev/spawn
    (print "starting long io...")
    (ev/sleep 10000)
```

```
    (print "finished long io!")))

# wait 2 seconds before canceling the long IO.
(ev/sleep 2)
(ev/cancel f "canceled")
```

Supervisor Channels

An interesting problem with supporting multiple tasks is error handling - if a task raises an error (or any other uncaught signal), who should handle it? The programmer often wants to know when a task fails (or finishes), and then continue with some other code. Alternatively, it might be desirable to try again a number of times before really giving up.

Janet has the concept of supervisor channels to support this. When a new task is scheduled with `ev/go`, one can optionally specify a channel to set as the new tasks supervisor. When the task yields to the event loop, or raises any kind of fiber signal, a message is automatically written to the supervisor channel. Another task can then continuously listen on that channel to monitor other tasks, handling errors or restarting them. This is a powerful pattern that can be used to build robust applications that gracefully handle failure.

```
(def channel (ev/chan))

(var counter 0)

(defn worker
  "A worker that is unreliable and sometimes fails."
  []
  (repeat 50
    (print "working...")
    (ev/sleep 0.1)
    (when (> 0.05 (math/random))
      (print "worker errored out!")
      (error "unexpected error"))
    (++ counter))
  (print "worker finished normally!"))

(defn start-worker
  "Start a worker with the given channel as its supervisor"
  []
  (print "starting new worker...")
  (ev/go (fiber/new worker :tp) nil channel))

# make sure there are always two workers until counter gets to 100
(start-worker)
(forever
  (start-worker)
  (def [status fiber] (ev/take channel))
  (print "worker " fiber " completed with status :" status)
  (if (>= counter 100) (break)))
```

The fiber mask argument (`:tp` above) determines what signals are automatically written to the supervisor channel. By default, the fiber will only write to the supervisor channel when it completes successfully, with the message `[:ok fiber]`. The most useful setting here is `:t`, which causes the fiber to write an event to the supervisor channel when it terminates. These events will be tuples of 2 arguments, `(signal fiber)`.

A task can also write custom messages to its supervisor with `ev/give-supervisor`.

© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- int
- jpm
    - rules
    - cc
    - cgen
    - cli
    - commands
    - config
    - make-config
    - dagbuild
    - pm
    - scaffold
    - shutil
- math
- module
- net
- os
- peg
- parser
- spork
    - argparse
    - base64
    - cc
    - cjanet
    - crc
    - channel
    - cron
    - data
    - ev-utils
    - fmt
    - generators
    - getline
    - htmlgen
    - http
    - httpf
    - infix
    - json
    - mdz
    - math
    - misc
    - netrepl
    - pgp
    - build-rules
    - path
    - randgen
    - rawterm
    - regex
    - rpc
    - schema
    - sh
    - msg

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Foreign Function Interface

Linting Core API
Starting in version 1.23.0, Janet includes a Foreign Function Interface module on x86-64, non-Windows systems. This lets programmers more easily call into native code without needing to write extensive native bindings in C, which is often very tedious. While the FFI is convenient and quite general, it lacks both the flexibility, safety, and speed of bindings written against the Janet C API. Programmers should be aware of this before choosing FFI bindings over a traditional native module. It is also possible to use a hybrid approach, where some core functionality is exposed via a C extension module, and the majority of an API is bound via FFI.

The FFI Module contains both the low-level primitives to load dynamic libraries (as with `dlopen` on posix systems), get function pointers from those modules, and call those function pointers. On top of this, there is a macro-based abstraction that makes it convenient to declare bindings and is sufficient in most cases.

Primitive Types

Primitive types in the FFI syntax are specified with keywords, and map directly to primitive type in C. There are a number of aliases for common types, such as the Linux kernel source style aliases for sized integer types. More complex types can be built from these primitive types. On x86-64, long doubles are unsupported.

Primitive Type	Corresponding C Type
`:void`	void
`:bool`	bool
`:ptr`	void *
`:string`	const char *
`:float`	float
`:double`	double
`:int8`	int8_t
`:uint8`	uint8_t
`:int16`	int16_t
`:uint16`	uint16_t
`:int32`	int32_t
`:uint32`	uint32_t
`:int64`	int64_t
`:uint64`	uint64_t
`:size`	size_t
`:ssize`	ptrdiff_t
`:r32`	float
`:r64`	double
`:s8`	int8_t
`:u8`	uint8_t
`:s16`	int16_t
`:u16`	uint16_t

Primitive Type	Corresponding C Type
`:s32`	int32_t
`:u32`	uint32_t
`:s64`	int64_t
`:char`	char
`:short`	short
`:int`	int
`:long`	long
`:byte`	uint8_t
`:uchar`	uint8_t
`:ushort`	unsigned short
`:uint`	unsigned int
`:ulong`	unsigned long

All primitive types with the exception of `:void`, `:bool`, `:ptr`, and `:string` are numeric types. 64 bit integer types can also be mapped to Janet's 64 bit integers if the `int/` module is enabled. The void type can only be used as a return value, and bool maps to either Janet `true` or Janet `false`. The `:string` type will map a Janet string to a `const char *` and vice-versa.

The `:ptr` type is the most flexible, catch-all type. All bytes sequence types, raw pointers, nil and abstract types can be converted to raw pointers (NULL is mapped to nil). If the native function will mutate data in the pointer, be sure not to pass in strings, symbols and keywords, as these are expected to be immutable. Buffers can be mutated freely. Functions returning pointers (either directly or in a struct) will return raw, opaque pointers. Data in the pointer can be inspected with `ffi/read` if needed.

Structs

FFI struct types (not to be confused with Janet structs) can be created with the `ffi/struct` function. All `ffi/` functions that take type arguments will implicitly create structs if passed tuples for convenience, but if you are going to reuse a struct definition, it is recommended to create the struct explicitly. Otherwise, multiple copies of identical struct definitions will be allocated.

Struct creation simply takes all of the types inside the struct in layout order; elements are not named. However, this is sufficient for interfacing with libraries and reduces overhead when mapping to Janet values.

```
(def my-struct (ffi/struct :int :int :ptr))
# Maps to the following in C:
# struct my_struct {
#   int a;
#   int b;
#   void *c;
# }
```

Packed structs are also supported, either for all struct members or for individual members. To specify a single member as packed, precede the member type with the keyword `:pack`. To indicate that all members of the struct should be packed, include `:pack-all` somewhere in the struct definition.

```
(ffi/size (ffi/struct :char :int)) # -> 8
(ffi/size (ffi/struct :char :pack :int)) # -> 5
```

C structs map to Janet tuples - that is, to pass a struct to an FFI function, pass in a tuple, and struct-returning functions will return tuples. To map C structs to other types (such as a Janet struct), you must do the conversion manually.

Array Types

Array types are defined with a Janet array of one or two elements - the first element is the type of array elements, and the optional second element is the number of elements in the array. If there is no second element, the type is a 0 element array which can be used to implement flexible array members as defined in C99.

(Although a zero-length has a size of zero, it has a required alignment so needs to be included in struct definitions.)

```
(ffi/size @[:int 10]) # -> 40
(ffi/size @[:int 0]) # -> 0
(ffi/size [:char]) # -> 1
(ffi/size [:char @[:int]]) # -> 4
```

Using Buffers - `ffi/write` and `ffi/read`

While primitive types and nested struct types will be converted to and from Janet values automatically, the FFI will not dereference pointers for you as a general rule, with the exception of returning string types. You also cannot use the common C idiom of converting between arrays and pointers as needed since Janet values are not laid out in memory as any C ABI specifies. To pass a pointer to a struct or array of values to a native FFI function, one must use `ffi/write` to write Janet values to a buffer. That buffer can then be passed as a `:ptr` type to a function.

```
(ffi/context "./mylib.so")
(def my-type (ffi/struct :char @[:int 4]))
(ffi/defbind takes_a_pointer :void [a :ptr])
(def buf (ffi/write my-type [100 [0 1 2 3]]))
(takes_a_pointer buf)
```

When using buffers in this manner, keep in mind that pointers written to the buffer cannot be followed by the garbage collector. Is up to the programmer to ensure such pointers do not become invalid by either keeping explicit references to these values or (temporarily) turning off the garbage collector.

The inverse of this process is dereferencing a returned pointer. `ffi/read` takes either a byte sequence, an abstract type, or a raw pointer and extracts the data at that address into Janet values.

```
(ffi/context "./mylib.so")
(def my-type (ffi/struct :char @[:int 4]))
(ffi/defbind returns_a_pointer :ptr [])
(def pointer (returns_a_pointer))
(pp (ffi/read my-type pointer))
```

Getting Function Pointers and Calling Them

The FFI module can use any opaque pointer as a function pointer, and while usually you will be loading functions from native modules loaded with `ffi/native`, you can use pointer values obtained from anywhere in your program. Of course, if these pointers are not actually C function pointers, your program will

likely crash.

To load a dynamic library (.so) file, use `(ffi/native path-to-lib)`. This will return an abstract type that can be used to look up symbols. You can pass `nil` as the path to return the current binary's symbols. The function `(ffi/lookup native-module symbol-name)` is then used to get pointers from the shared object.

Once you have a function pointer, you will still need a function signature to call the function. Function signatures are created with `ffi/signature calling-convention return-type & args)`. Since certain functions may use calling conventions besides the default, you may specify the convention, such as `:sysv64`, or use `:default` to use the default calling convention on your system. As of version 1.23.0, `:sysv64` is the only supported calling convention. Not all systems and operating systems will support all calling conventions. Varargs are not supported.

Once you have both a function pointer and a function signature, you can finally make a call to your function with `(ffi/call function-pointer function-signature & arguments)` You will probably want to save the function pointer and signature rather than recalculate them on each use.

```
(def self-symbols (ffi/native))
(def memcpy (ffi/lookup self-symbols "memcpy"))
(def signature (ffi/signature :default :ptr :ptr :ptr :size))

# Example usage of our memcpy binding
(def buffer1 @"aaaa")
(def buffer2 @"bbbb")
(ffi/call memcpy signature buffer1 buffer2 4)
(print buffer1) # prints bbbb
```

High-Level API - `ffi/context` and `ffi/defbind`.

Using the low-level api to manually load dynamic libraries can get rather tedious, so the FFI module has a few macros and functions to make it easier. The function `ffi/context` is used to select a native module that subsequent bindings will refer to. `ffi/defbind` will then lookup function pointers, create signature values, and create Janet wrappers around ffi/call for you. The memcpy example from above would look like so with the high level api:

```
(ffi/context nil)
(ffi/defbind memcpy :ptr
  [dest :ptr src :ptr n :size])

(def buffer1 @"aaaa")
(def buffer2 @"bbbb")
(memcpy buffer1 buffer2 4)
(print buffer1) # prints bbbb
```

This code uses `ffi/native`, `ffi/lookup`, `ffi/signature`, `ffi/call` behind the scenes, and you can mix and match the `ffi/defbind` macro with explicit bindings.

Callbacks

One limitation of Janet's FFI module is passing function pointers to C functions, such as in `qsort`. This is unsupported in the general case, as it requires runtime generation of machine code. Instead, callback functions must be written in C. Often, a C library will allow setting some kind of user data, which will then be passed

back when the callback is invoked by the library. One could put a `JanetFunction *` into that user data slot and have a common "trampoline" native function that can be a sort of universal callback that would call that userdata parameter it received as a JanetFunction. While this is far from general, it is effective in many cases, and so the `ffi/module` provides one such function pointer out of the box with `ffi/trampoline`.

GTK Example

One good use for FFI bindings are interfacing with GUI libraries. For example, one could be building a standalone binary that could detect available GUI libraries on the host system, and use FFI bindings to interact with the host GUI framework that was detected. Below is an example of what FFI with GTK might look like using the high-level, macro based abstraction with `ffi/context` and `ffi/defbind`.

```
(ffi/context "/usr/lib/libgtk-3.so" :lazy true)

(ffi/defbind
  gtk-application-new :ptr
  [title :string flags :uint])

(ffi/defbind
  g-signal-connect-data :ulong
  [a :ptr b :ptr c :ptr d :ptr e :ptr f :int])

(ffi/defbind
  g-application-run :int
  [app :ptr argc :int argv :ptr])

(ffi/defbind
  gtk-application-window-new :ptr
  [a :ptr])

(ffi/defbind
  gtk-button-new-with-label :ptr
  [a :ptr])

(ffi/defbind
  gtk-container-add :void
  [a :ptr b :ptr])

(ffi/defbind
  gtk-widget-show-all :void
  [a :ptr])

(ffi/defbind
  gtk-button-set-label :void
  [a :ptr b :ptr])

(def cb (delay (ffi/trampoline :default)))

(defn on-active
  [app]
  (def window (gtk-application-window-new app))
  (def btn (gtk-button-new-with-label "Click Me!"))
  (g-signal-connect-data btn "clicked" (cb)
                         (fn [btn] (gtk-button-set-label btn "Hello World"))
                         nil 1)
  (gtk-container-add window btn)
  (gtk-widget-show-all window))

(defn main
```

```
[&]
(def app (gtk-application-new "org.janet-lang.example.HelloApp" 0))
(g-signal-connect-data app "activate" (cb) on-active nil 1)
# manually build an array with ffi/write
(g-application-run app 0 nil))
```

Linting Core API
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- debug
- ev
- ffi
- fiber
- file
- int
- jpm
    - rules
    - cc
    - cgen
    - cli
    - commands
    - config
    - make-config
    - dagbuild
    - pm
    - scaffold
    - shutil
- math
- module
- net
- os
- peg
- parser
- spork
    - argparse
    - base64
    - cc
    - cjanet
    - crc
    - channel
    - cron
    - data
    - ev-utils
    - fmt
    - generators
    - getline
    - htmlgen
    - http
    - httpf
    - infix
    - json
    - mdz
    - math
    - misc
    - netrepl
    - pgp
    - build-rules
    - path
    - randgen
    - rawterm

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 provides 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 the 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 prototyp
(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 ti
  [x]
  (each name ['a 'b 'c 'd 'e 'f 'g]
    (setdyn name @{:value (math/random)}))
  nil)
```

Fiber Overview Error Handling
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 Modules
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))
```

Two error-handling forms built on fibers

`try`

Janet provides a simple macro `try` to make error-handling a bit easier in the common case. `try` performs the function of a traditional try/catch block; it will execute its body and, if any error is raised, optionally bind the error and the raising fiber in its second clause.

```
(try
  (do
    (print "inside block...")
    (error "oops")
    (print "will never get here"))
  ([err fib]
    # err and (fiber/last-value fib) are the same here
    (print "caught error: " err)
    (print "fib's last-value: " (fiber/last-value fib))
    # returns the status of fib, in this case, :error
    (fiber/status fib)))
```

Note that in the above example, the caught `err` and `fib`'s last-value are the same, i.e. "oops".

`protect`

Janet also provides the `protect` macro for a slightly different flavor of error-handling, converting caught errors into further expressions.

```
(protect
  (if (> (math/random) 0.42)
    (error "Good luck")
    "Bad luck"))
```

Dynamic Bindings Modules
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs
    - Tuples
  - Destructuring
  - Fibers
    - Dynamic Bindings
    - Errors
  - Modules
  - Object-Oriented Programming
  - Parsing Expression Grammars
  - Prototypes
  - The Janet Abstract Machine
  - The Event Loop
  - Multithreading
  - Networking
  - Process Management
    - Executing
    - Spawning
  - Documentation
  - jpm
  - Linting
  - Foreign Function Interface
- API
  - array
  - buffer
  - bundle

- debug
- ev
- ffi
- fiber
- file
- int
- jpm
  - rules
  - cc
  - cgen
  - cli
  - commands
  - config
  - make-config
  - dagbuild
  - pm
  - scaffold
  - shutil
- math
- module
- net
- os
- peg
- parser
- spork
  - argparse
  - base64
  - cc
  - cjanet
  - crc
  - channel
  - cron
  - data
  - ev-utils
  - fmt
  - generators
  - getline
  - htmlgen
  - http
  - httpf
  - infix
  - json
  - mdz
  - math
  - misc
  - netrepl
  - pgp
  - build-rules
  - path
  - randgen
  - rawterm

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Fiber Overview

Destructuring 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. In all of these cases a value is produced and can be obtained via the function `fiber/last-value`. This function takes a fiber as its sole argument. If a fiber has never been resumed, `fiber/last-value` will return `nil`.

Different from traditional coroutines, Janet's fibers implement a signaling mechanism, which is used to distinguish between 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. This state is referred to as the status of a fiber and can be checked using the `fiber/status` function. This function takes a fiber as its sole argument.

To create a fiber, use the `fiber/new` function. The fiber constructor takes one or two arguments. The first argument (required) is the function that the fiber will execute. This function must accept an arity of zero. The next argument (optional) 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 of the different kinds of signals from a fiber. Any untrapped signals are simply propagated to the previous calling fiber.

Note that the terms "block", "mask", and "capture" are sometimes used to refer to the "trapping" of signals.

```
(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)
(fiber/status f) # -> :new
(fiber/last-value f) # -> fiber hasn't been resumed yet, so returns nil

(resume f) # -> 1
(fiber/last-value f) # -> 1 (again)
(resume f) # -> 2
(resume f) # -> 3
(resume f) # -> 4
(fiber/status f) # -> :pending
(resume f) # -> 5
(fiber/status f) # -> :dead
(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 (i.e. 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, `resume`s 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
```

The `signal` function can be used to raise a particular signal. The function's first argument, `what`, specifies the signal, while the second argument, `x`, is an arbitrary payload value.

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

(defer (pp :hey)
  (signal :user4 :hello)
  (pp :unreached))
# only prints :hey
```

Destructuring Dynamic Bindings
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 then 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 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms

- ◊ dagbuild
- ◊ pm
- ◊ scaffold
- ◊ shutil

- ♦ math
- ♦ module
- ♦ net
- ♦ os
- ♦ peg
- ♦ parser
- ♦ spork
  - ◊ argparse
  - ◊ base64
  - ◊ cc
  - ◊ cjanet
  - ◊ crc
  - ◊ channel
  - ◊ cron
  - ◊ data
  - ◊ ev-utils
  - ◊ fmt
  - ◊ generators
  - ◊ getline
  - ◊ htmlgen
  - ◊ http
  - ◊ httpf
  - ◊ infix
  - ◊ json
  - ◊ mdz
  - ◊ math
  - ◊ misc
  - ◊ netrepl
  - ◊ pgp
  - ◊ build-rules
  - ◊ path
  - ◊ randgen
  - ◊ rawterm
  - ◊ regex
  - ◊ rpc
  - ◊ schema
  - ◊ sh
  - ◊ msg
  - ◊ stream
  - ◊ tasker
  - ◊ temple
  - ◊ test
  - ◊ tarray
  - ◊ utf8
  - ◊ zip
- ♦ string
- ♦ table

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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.

Another way an anonymous function can be created is via the `|` reader macro (or the equivalent `short-fn` macro):

```
# These are functionally equivalent
((fn [x] (+ x x)) 1) # -> 2
(|(+ $ $) 1) # -> 2

# Multiple arguments may be referenced
(|(+ $0 $1 $2) 1 2 3) # -> 6

# All arguments can be accessed as a tuple
(|(map inc $&) -1 0 1 2) # -> @[0 1 2 3]

# | is shorthand for (short-fn ...)
((short-fn (+ $ $)) 1) # -> 2

# Other constructs can be used after |
(|[:x $] :y) # -> [:x :y]
(|{:a $0 :b $1} 1 2) # -> {:a 1 :b 2}

# Will throw an error about the wrong arity
(|(inc $) 1 2)
# Will not throw an error about the wrong arity
(|(get $& 0) 1 2)
```

Within the above sorts of contexts the symbol `$` refers to the first (or zero-th) argument. Similarly, `$0`, `$1`, etc. refer to the arguments at index 0, 1, etc. and `$&` refers to all passed arguments as a tuple.

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.

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))
```

Note that what follows `&keys` does not have to be a struct and can instead be a single symbol. The symbol can then be used as a name within the function body for a struct containing the passed keys and values:

```
(defn feline-counter
  [&keys animals]
  (if-let [n-furries (get animals :cat)]
    (printf "Awww, I see %d cat(s)!" n-furries)
    (print "Shucks, where are my friends?")))

(feline-counter :ant 1 :bee 2 :cat 3)
```

Optional Flags

As a matter of style, some Janet functions allow you to optionally pass in multiple flags from a given set, as a single argument. (A "set" in the generic sense --- Janet doesn't have a `set` type.)

For example, if your `foo` function takes a string argument, and may also be passed zero or more flags from the set `:a`, `:b`, and `:c`, you could call `foo` like so:

```
(foo "hi")       # No flags passed.
(foo "hi" :a)    # Just the `:a` flag.
(foo "hi" :ab)   # The `:a` and `:b` flags.
(foo "hi" :ba)   # Same.
(foo "hi" :cab)  # All three flags specified.
```

The pattern is similar to passing multiple flags to a command in the shell:

```
$ my-command -abc      # is like
$ my-command -a -b -c
```

but with Janet we're using `:` instead of `-`.

An example of using optional flags in your own functions:

```
#!/usr/bin/env janet

(defn my-cool-string
  ``Decorates `some-str` with stars around it.

  `flags` is set of flags; it is a keyword where each
  character indicates a flag from this set:

    * `:u` --- uppercase the string too.
    * `:b` --- put braces around it as well.

  Returns the decorated string.
  ``
  [some-str &opt flags]
  (var u-flag-set false)
  (var b-flag-set false)
  (when flags
    (set u-flag-set (string/check-set flags :u))
    (set b-flag-set (string/check-set flags :b)))
  (let [s (string "*" some-str "*")
        s (if u-flag-set (string/ascii-upper s) s)
        s (if b-flag-set (string "{" s "}")     s)]
    s))

(print (my-cool-string "I â ¥ Janet"))      # *I â ¥ Janet*
(print (my-cool-string "I â ¥ Janet" :u))   # *I â ¥ JANET*
(print (my-cool-string "I â ¥ Janet" :b))   # {*I â ¥ Janet*}
(print (my-cool-string "I â ¥ Janet" :ub))  # {*I â ¥ JANET*}
```

Built-in Janet functions that utilize this style include `file/open`, `fiber/new`, and `os/execute`.

Named arguments

Starting in version 1.23.0, functions support named arguments. This is a more convenient syntax for keyword arguments. Functions with named arguments are implicitly variadic and will not reject badly named arguments. All symbols after the `&named` symbol in the parameter list need to be passed to the function as arguments preceded by a keyword of the same name.

```
(defn named-fn
  [x &named person1 person2]
  (print "giving " x " to " person1)
  (print "giving " x " to " person2))

(named-fn "apple" :person1 "bob" :person2 "sally")
```

Flow Strings, Keywords, and Symbols
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators

- ◊ scaffold
- ◊ shutil

- ♦ math
- ♦ module
- ♦ net
- ♦ os
- ♦ peg
- ♦ parser
- ♦ spork
  - ◊ argparse
  - ◊ base64
  - ◊ cc
  - ◊ cjanet
  - ◊ crc
  - ◊ channel
  - ◊ cron
  - ◊ data
  - ◊ ev-utils
  - ◊ fmt
  - ◊ generators
  - ◊ getline
  - ◊ htmlgen
  - ◊ http
  - ◊ httpf
  - ◊ infix
  - ◊ json
  - ◊ mdz
  - ◊ math
  - ◊ misc
  - ◊ netrepl
  - ◊ pgp
  - ◊ build-rules
  - ◊ path
  - ◊ randgen
  - ◊ rawterm
  - ◊ regex
  - ◊ rpc
  - ◊ schema
  - ◊ sh
  - ◊ msg
  - ◊ stream
  - ◊ tasker
  - ◊ temple
  - ◊ test
  - ◊ tarray
  - ◊ utf8
  - ◊ zip
- ♦ string
- ♦ table
- ♦ misc
- ♦ tuple

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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

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 version or the latest stable version for Arch Linux from the Arch user repositories with janet-lang-git or janet-lang.

Other Linux

See instructions below for compiling and installing from source.

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.

Non-root install (macOS and Unix-like)

To build and install Janet (and jpm) to a local or temporary directory tree instead of the default prefix of `/usr/local`, set the `PREFIX` environment variable while building and installing. You will like likely also

want to update your path to point to `PREFIX/bin` to use `janet`, `jpm`, and any scripts installed via `jpm`.

```
export PREFIX="$HOME"/.local
make -j
make install
make install-jpm-git # you can also do this manually by reading the janet-lang/jpm.git rea
export PATH="$PREFIX"/bin:"$PATH"
```

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.

Next, to install the Janet Project Manager tool (`jpm`), clone the jpm repository, and from within that directory, run:

```
[sudo] janet bootstrap.janet
```

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 (you will need the latest MSVC build tools and the Windows SDK components)
2. Run a Visual Studio Command Prompt (`cl.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 (`cl.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

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 -Dtyped_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!".

Put the following code in a file named `hello.janet`, and run `janet hello.janet`:

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

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 also brackets form a tuple; 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 of the tuple. While this may be familiar in function calls, in Janet this idea extends to arithmetic and all control forms.

Second program

If you create a "main" function, `janet` will automatically call that for you when you run your script. Here's an example that prints out the command line args tuple passed to your program, its length, and a greeting to the whomever's named first on the command line:

```
#!/usr/bin/env janet

(defn print-greeting
  [greetee]
  (print "Hello, " greetee "!"))

(defn main
  [& args]
  (pp args) # pretty-prints the args tuple
  (print (length args))
  (print-greeting (get args 1)))
```

Output:

```
$ ./foo.janet World Famous Fries
("./foo.janet" "World" "Famous" "Fries")
4
Hello, World!
```

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 bindings in the REPL, use the `doc` macro without arguments to print them out.

```
(doc)
```

Note: see the `all-bindings` function for related functionality.

You can also browse the core library API on the website.

© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

jpm

Documentation Linting
JPM is a build tool that can be installed along with Janet to help build and install libraries for Janet. The main uses are installing dependencies, compiling C/C++ to native libraries, and other project management tasks. The source code for JPM can be found at https://github.com/janet-lang/jpm.git. With janet already installed, JPM is also self bootstrapping.

```
git clone --depth=1 https://github.com/janet-lang/jpm.git
cd jpm
sudo janet bootstrap.janet
```

The bootstrap script can also be configured to install jpm to different directories by setting the `DESTDIR` environment variable. Ideally, jpm should be installed to the same tree as Janet, although this is not strictly required. See the README in jpm's repository for more information.

Updating JPM

Once installed and configured, JPM can update itself from the git repository at any time.

```
sudo jpm install jpm
```

`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.)

User local install

The JANET_TREE environment variable can be used to set the tree the jpm installs things to. By default, running `janet` from the command line separately will not use modules in the custom tree, so you will likely want to modify JANET_PATH as well.

```
export JANET_TREE=$HOME/.local/jpm_tree
jpm deps
jpm build
jpm test
jpm install
# alternative: jpm --tree=$HOME/.local/jpm_tree deps
```

Project local install

JPM also has some flags to install dependencies to a tree local to a project. Dependencies will be installed to ./jpm_tree/lib (and binaries installed to ./jpm_tree/bin) when passing the `-l` flag to jpm.

```
jpm -l deps
jpm -l build
jpm -l test
# Run a janet interpreter in the local environment with access to all dependencies installe
jpm -l janet
```

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"
 :install true)
```

`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.

If the optional key-value pair `:install true` is specified in the `declare-executable` form, by default, the appropriate `jpm install` command will install the resulting executable to the `JANET_BINPATH` (but see the jpm man page for further details).

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: " ;(interpose ", " 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.

Other `declare-` callables

Some additional `declare-` callables are:

- `declare-bin`

  Declare a generic file to be installed as an executable. Specify file path via `:main`.
- `declare-binscript`

  Declare a janet file to be installed as an executable script. Creates a shim on windows. If `:hardcode-syspath` is true, will insert code into the script such that it will run correctly even when `JANET_PATH` is changed. If `:is-janet` is truthy, will also automatically insert a correct shebang line if jpm's configuration is set with `:auto-shebang` as truthy.
- `declare-headers`

  Declare headers for a library installation. Installed headers can be used by other native libraries. Specify paths via `:headers` and prefix via `:prefix`.
- `declare-manpage`

  Mark a manpage for installation.

Custom Trees

For per-project or per-user development (as opposed to system-wide development), you can use custom jpm trees rather than a system default by passing the `--tree=<somewhere>` argument all jpm commands or setting the `JANET_TREE` environment variable. This will set the location where jpm will install modules, headers, scripts, and other data to. For project local development in a tree `./jpm_tree`, you can use the `--local` or `-l` shorthand for this.

```
jpm --tree=/opt/jpm_tree deps
jpm --tree=/opt/jpm_tree install spork
jpm -l deps
jpm -l test
```

Versioning and Library Bundling

JPM does not do any semantic version resolution at the moment. Instead, it is recommended to make all changes to libraries as backwards-compatible as possible, and release new libraries for breaking changes in almost all cases. For creators of executable programs (versus a library author), it is recommended to use a local tree and lockfiles to pin versions for consistent builds.

As a matter of style, it is also recommended to group small libraries together into "bundles" that are updated, tested, and deployed together. Since Janet libraries are often quite small, the cost of downloading more functionality that one might need isn't particularly high, and JPM can remove unused functions and bindings from generated standalone binaries and images, so there is no runtime cost either. By avoiding a plethora of tiny libraries, users of libraries do not manage as many dependencies, and modules are more likely to work together they can be tested together.

While `jpm` may superficially resemble `npm`, it is the author's opinion that it is suited to a different style of development.

Documentation Linting
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Linting

jpm Foreign Function Interface
The `janet` binary comes with a built in linter as of version 1.16.1. This allows the compiler to emit warnings that don't always need to stop compilation. Linting is enabled by default at the command line and can be adjusted with the `-x` and `-w` flags.

Linting messages from the compiler contain a linting level, a message, and source code location. There are 3 default levels of linting messages:

- `:relaxed` - Reserved for serious messages (priority 1)
- `:normal` - The normal default level for linting messages (priority 2)
- `:strict` - The linting level reserved for unimportant or nit-picky warnings (priority 3)

Note that the levels of lints may seem backwards, as the levels of the lint are from the perspective of the programmer. For example, a "picky" lint message that was warning the user about a matter of style would only show up if the programmer had `:strict` mode enabled for linting. You can also use the priority integers directly for lint levels.

Deprecation

There is also support for deprecation messages in the compiler to trigger lint warnings if a user attempts to use a deprecated binding. Deprecation here is the process of signaling to users that a function or other binding should not longer be used for new code. Deprecation levels are the same as the linting levels.

```
(defn old-fn
 :deprecated
 "This function is deprecated"
 [x]
 (* 2 x))

(defn older-fn
 {:deprecated :relaxed}
 "This function is deprecated, and we really don't want you to use it."
 [x]
 (* 2 x))
```

Lint Warnings and Errors

Whenever Janet code is being executed by the `run-context` function, there are lint thresholds for both errors and warnings. If a lint message has a level equal to or below the error threshold, it will be converted into a compiler error. Otherwise, if the lint message has a level equal to or below the warning threshold, a warning will be printed to `stderr`. Finally, lints above both the error and warning threshold will be ignored.

The error threshold can be set with `(setdyn :lint-error error-threshold)`, and the warning threshold can similarly be set with `(setdyn :lint-warn warn-threshold)`, where threshold values are any of the lint levels (keywords) or an integer. By default, the error threshold is 0 (no lints will cause an error), and the warn threshold is 2 (relaxed and normal lints will cause a warning to print, but strict lints will be ignored). These threshold can also be set at the command line with the `-w` and `-x` flags for the warn and error thresholds respectively.

Macro Linting

Inside macros, Janet exposes an array used to collect lint messages. A function `maclintf` is provided to help construct error messages if some issue occurs inside a macro. `maclintf` can be used to check arguments and assert that certain invariants are held when invoking a macro.

```
(defmacro my-plus
 "Do addition and hint that the first argument should be a constant number."
 [x y]
 (unless (number? x)
  (maclintf :normal "my-plus should be invoked with a numeric constant as x, got %v" x))
 ~(,+ ,x ,y))
```

Use of `maclintf` over other methods of reporting warnings is preferred as the programmer has the ability to customize how to handle different levels of lint messages. For fatal errors inside the macro, use the `error` function as usual.

jpm Foreign Function Interface
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 total 0)
(var i 0)
(while (< i 10)
    (+= total i)
    (++ i))
(print total) # 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 of 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 total 0)
(for i 0 10 (+= total i))
(print total) # 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 total 0)
(loop [i :range [0 10]] (+= total i))
(print total) # 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 total 0)
(loop [i :range [0 10] :when (even? i)] (+= total i))
(print total) # prints 20
```

The `loop` macro has several verbs (e.g. `:range`) and modifiers (e.g. `:when`) that let the programmer more easily generate common looping idioms. The `loop` macro is similar to the Common Lisp `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 `eachp`, 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.

Note that the above examples can also be expressed as:

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

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

Strings, Keywords, and Symbols Macros
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

API

- array
- buffer
- bundle
- debug
- ev
- ffi
- fiber
- file
- int
- jpm
  - rules
  - cc
  - cgen
  - cli
  - commands
  - config
  - make-config
  - dagbuild
  - pm
  - scaffold
  - shutil
- math
- module
- net
- os
- peg
- parser
- spork
  - argparse
  - base64
  - cc
  - cjanet
  - crc
  - channel
  - cron
  - data
  - ev-utils
  - fmt
  - generators
  - getline
  - htmlgen
  - http
  - httpf
  - infix
  - json
  - mdz
  - math
  - misc
  - netrepl
  - pgp

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Macros

Looping Data Structures

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)))
```

Accidental Binding Capture

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 solved in many programming languages (such as Scheme) by hygienic macros. Hygienic macros prevent such accidental captures, but usually constrain the kinds of

macros that can be written. Janet's macro system does *not* use hygienic macros as presented in Kohlbecker et al.'s 1986 paper - instead, programmer diligence is required to avoid accidental captures.

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.

```
(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 Data Structures
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Modules

Error Handling Object-Oriented Programming

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 to 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")
```

By default, the following files will be searched for in order for `(import foo)`:

* `foo.jimage` * `foo.janet` * `foo/init.janet` * `foo.<native-extension>`

where <native-extension> is the file extension for shared objects / libraries for the current platform (e.g. `dll` for Windows, `dylib` for macos, and `so` for other unix-likes). This is a "first-match wins" arrangement so at most one file will be imported.

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,

because if `url` contains spaces in `load-url`, the module will not load correctly. A more robust solution would quote the URL.

```
(defn- load-url
  [url args]
  (def p (os/spawn ["curl" url "-s"] :p {:out :pipe}))
  (def res (dofile (p :out) :source url ;args))
  (:wait p)
  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 `./`. For example, suppose 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 robust because 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)

...
```

Note that relative imports are relative to the current file, not to the working directory. For that behavior, see the section on working directory imports.

Pre-loaded Modules

An easy way to bypass the import system and create custom modules is to manually add modules into `module/cache`.

```
(put module/cache "mymod" (dofile "localfile.janet"))
# The module defined by localfile.janet can now be
# imported with (import mymod)
```

This is not recommended except in a few circumstances where a module needs to be loaded at runtime but doesn't exist on disk, such as if a program were compiled into a standalone executable. A pattern to allow this in a standalone executable built with `jpm` is as follows:

```
(def mod1 (require "./localfile"))

(defn main [& args]
  (put module/cache "mod1" mod1)
  # now calls to (import mod1) at runtime will succeed.
  (import mod1))
```

Working Directory Imports

Starting in 1.14.1, Janet will allow imports relative to the current working directory. This is very useful in scripting circumstances where source code must be loaded at runtime. This can be done with the function `dofile`, but it can also be done with the default import system.

To import a file relative to the working directory, prefix the import path with a forward slash "/".

```
(import /local)
```

This will import from a file `./local.janet`, (or `./local.so`, `./local.jimage`, and so on), and prefix the imported symbols with `local/`.

@-prefixed Imports

Starting in 1.26.0, Janet allows importing modules from custom directories more easily using the `@`-prefix in a module path.

For example, if there is a dynamic binding `:custom-modules` that is a file system path to a directory of modules, import from that directory with:

```
(import @custom-modules/mymod)
```

As a special case, it is possible to import from absolute paths by prefixing an absolute path with `@`. For example, to import from `/tmp/custom-modules/mymod.janet`, express this as:

```
(import @/tmp/custom-modules/mymod)
```

Note that although using absolute paths is possible and useful for testing, it is not recommended for most production use cases.

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 via `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 keyword `:private` as metadata for 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 is 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 1000

(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
```

Error Handling Object-Oriented Programming
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Networking

Multithreading Process Management
The internet is a huge part of our daily lives in the 21st century, and the situation is no different for programmers. Janet has basic support for connecting to the internet out of the box with the `net/` module.

The `net/` module provides a socket-based networking interface that is portable, simple, and general. The net module provides both client and server abstractions for TCP sockets (streams), UDP (datagrams), IPv4, IPv6, DNS, and unix domain sockets on supported platforms. All operations on sockets are non-blocking, so servers can handle multiple clients on a single thread. For more general information on socket programming, see Beej's Guide to Network Programming, which gives a thorough overview in C. There are also example programs in the example directory of the Janet source code which show how to create simple servers and clients.

The `net/` module does not provide any protocol abstractions over sockets - this means no TLS and no HTTP out of the box. Such protocols need to be provided in third-party libraries.

Generic Stream Type

Most functions dealing with non-blocking IO in Janet will either accept or create an abstract type of `core/stream`. Streams are wrappers around file descriptors on Posix platforms, and HANDLEs on Windows. Most streams provide a number of generic methods such as `:read`, `:write`, and `:chunk` for reading and writing bytes, much like a file - there are many parallels to the `file/` module, but streams do not block and do not support buffering.

In the `net/` module, sockets and servers are both instances of `core/stream`. However, they have different capabilities - for example, you can only call `net/accept` on a server, and only call `net/read` on a socket.

Clients and Servers

In the following sections, we use "client" to mean a program that initiates communication with another program across a network, the "server", which can respond to this communication (or ignore it).

TCP (Stream sockets)

Stream sockets provide an ordered, reliable sequence of bytes across a network. However, in order to send messages a connection must be established. These are the most common kind of socket and the default if a socket type is not specified in most functions.

Client

A TCP client needs to call `net/connect` to get a socket, and then can read from and write to that socket with `net/read` and `net/write`. The third argument to `net/connect` should be `:stream`, although it is optional as stream sockets are the default - the other option is `:datagram`.

```
(with [conn (net/connect "127.0.0.1" "8000" :stream)]
  (printf "Connected to %q!" conn)
  (:write conn "Echo...")
  (print "Wrote to connection..."))
```

```
(def res (:read conn 1024))
(pp res))
```

Because streams support a `:close` method, they can be used as the target of a `with` macro for scoped resource control.

Server

The `net/` module provides a few ways to create a TCP server, but in all cases a few steps are required.

1. Create a server value with `net/listen`.
2. Repeatedly accept connections to the server with `net/accept`.
3. Handle and eventually close each connection.

There are several functions that bundle different steps together - `net/accept-loop` combines steps 2 and 3 into a single call, while `net/server` can combine steps 1, 2 and 3 into one call. However, the long form way is still very short and is no less expressive.

```
# Create a server on localhost and listen on port 8000.
(def my-server (net/listen "127.0.0.1" "8000"))

# Handle exactly 10 connections 1-by-1
(repeat 10
 (def connection (net/accept my-server))
 (defer (:close connection)
  (net/write connection "Hello from server!")))

# Close server
(:close my-server)
```

In many cases, the programmer wants to handle more than 1 connection at a time. The `ev/` module can be used to great effect here to handle each new connection in a separate fiber, so that the accepting loop can accept the next connection as quickly as possible.

```
# Create a server on localhost and listen on port 8000.
(def my-server (net/listen "127.0.0.1" "8000"))

(defn handler
  "Handle a connection in a separate fiber."
  [connection]
  (defer (:close connection)
    (def msg (ev/read connection 100))
    (ev/sleep 10)
    (net/write connection (string "Echo: " msg))))

# Handle connections as soon as they arrive
(forever
 (def connection (net/accept my-server))
 (ev/call handler connection))
```

There are many possible variations, but all servers will follow this same basic structure.

UDP (Datagram sockets)

Datagram sockets provide an out-of-order, fast way to send datagrams across a network. Datagram sockets also provide the benefit of not needing a connection - one can send packets to an address without checking if that address is listening.

Client

A UDP client will generally create a fake connection to a server with `net/connect`, where the third argument is `:datagram`. The connection is "fake" because datagrams are connection-less, but we can use this connection to encapsulate the remote address of the server. The programmer can then use `net/write` and `net/read` to send and receive datagrams from the server (or `:write` and `:read`).

```
(def conn (net/connect "127.0.0.1" "8009" :datagram))
(:write conn (string/format "%q" (os/cryptorand 16)))
(def x (:read conn 1024))
(pp x)
```

Server

A Datagram server is a bit simpler than a stream server since there is no need to manage connections. Instead, the program listens for packets with `net/recv-from` and sends packets with `net/send-to`.

```
(def server (net/listen "127.0.0.1" "8009" :datagram))
(while true
  (def buf @"")
  (def who (:recv-from server 1024 buf))
  (printf "got %q from %v, echoing!" buf who)
  (:send-to server who buf))
```

DNS

Janet uses the Posix C function `getaddrinfo` to look up IP addresses from domain names. This is currently a blocking operation which can be bad in highly concurrent applications. However, this also affords us access to various features supported by the operating system like easy and cross platform IPv4, IPv6, and unix domain socket support.

Functions that do DNS will usually take host and port arguments, where host is a host address and port further specifies the address on the same machine. In IPv4 and IPv6, host should be an IP address or domain name, and port should be a 16-bit number.

Unix domain sockets

Supporting platforms allow unix domain sockets by accepting `:unix` as the host, and a file name as the port. On linux, abstract unix domain sockets are supported if the port argument begins with "@".

```
# Connect to a unix domain socket
(net/connect :unix "/tmp/my.socket")

# Connect to an abstract unix domain socket (linux only)
(net/connect :unix "@my.socket")
```

Relationship to the `ev/` module.

All non-blocking operations in the `net/` module avoid blocking by yielding to the event loop. As such, functions in the `ev/` module for manipulating fibers, such as `ev/cancel` will also work on functions in the `net/` module. Similarly, a function that blocks the event loop will prevent any networking code from progressing.

Multithreading Process Management
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- array
- buffer
- bundle
- debug
- ev
- ffi
- fiber
- file
- int
- jpm
    - rules
    - cc
    - cgen
    - cli
    - commands
    - config
    - make-config
    - dagbuild
    - pm
    - scaffold
    - shutil
- math
- module
- net
- os
- peg
- parser
- spork
    - argparse
    - base64
    - cc
    - cjanet
    - crc
    - channel
    - cron
    - data
    - ev-utils
    - fmt
    - generators
    - getline
    - htmlgen
    - http
    - httpf
    - infix
    - json
    - mdz
    - math
    - misc
    - netrepl
    - pgp
    - build-rules

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 Comparison Operators
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 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 Comparison Operators
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs
    - Tuples
  - Destructuring
  - Fibers
    - Dynamic Bindings
    - Errors
  - Modules
  - Object-Oriented Programming
  - Parsing Expression Grammars
  - Prototypes
  - The Janet Abstract Machine
  - The Event Loop
  - Multithreading
  - Networking
  - Process Management
    - Executing
    - Spawning
  - Documentation
  - jpm
  - Linting
  - Foreign Function Interface
- API
  - array
  - buffer
  - bundle

- debug
- ev
- ffi
- fiber
- file
- int
- jpm
    - rules
    - cc
    - cgen
    - cli
    - commands
    - config
    - make-config
    - dagbuild
    - pm
    - scaffold
    - shutil
- math
- module
- net
- os
- peg
- parser
- spork
    - argparse
    - base64
    - cc
    - cjanet
    - crc
    - channel
    - cron
    - data
    - ev-utils
    - fmt
    - generators
    - getline
    - htmlgen
    - http
    - httpf
    - infix
    - json
    - mdz
    - math
    - misc
    - netrepl
    - pgp
    - build-rules
    - path
    - randgen
    - rawterm

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Object-Oriented Programming

Modules 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. As of Janet version 1.19.0, structs can have prototypes as well as tables, although in prior version structs could still be used as objects, just without any 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!"
```

To create a struct that inherits fields from another struct, we need to use the constructing function `struct/with-proto`. There is no analog to `table/setproto` for structs because they are immutable, and that goes for prototypes as well. Structs can only inherit from other structs, not tables.

```
(def Car
 {:_name "Car"
  :honk (fn [self msg] (print "Car says: " msg))})

(def my-car (struct/with-proto Car :color :silver))
(:honk my-car "beep")
```

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)
```

Modules Parsing Expression Grammars
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
    - Syntax and the Parser

- ◊ make-config
- ◊ dagbuild
- ◊ pm
- ◊ scaffold
- ◊ shutil

♦ math
♦ module
♦ net
♦ os
♦ peg
♦ parser
♦ spork

- ◊ argparse
- ◊ base64
- ◊ cc
- ◊ cjanet
- ◊ crc
- ◊ channel
- ◊ cron
- ◊ data
- ◊ ev-utils
- ◊ fmt
- ◊ generators
- ◊ getline
- ◊ htmlgen
- ◊ http
- ◊ httpf
- ◊ infix
- ◊ json
- ◊ mdz
- ◊ math
- ◊ misc
- ◊ netrepl
- ◊ pgp
- ◊ build-rules
- ◊ path
- ◊ randgen
- ◊ rawterm
- ◊ regex
- ◊ rpc
- ◊ schema
- ◊ sh
- ◊ msg
- ◊ stream
- ◊ tasker
- ◊ temple
- ◊ test
- ◊ tarray
- ◊ utf8
- ◊ zip

♦ string

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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
"cat" (a string)	The literal string.
3 (an integer)	If positive, matches a number of characters, and advances that many characters. If 0, counts as an (empty string) match. If negative, matches if not that many characters and does not advance. For example, -1 will match the end of a string.
`(range "az" "AZ")`	Matches characters in a range and advances 1 character. Multiple ranges can be combined together.
`(set "abcd")`	Match any character in the argument string. Advances 1 character.
`true`	Always matches. Does not advance any characters. Epsilon in NFA.
`false`	Never matches. Does not advance any characters. Equivalent to `(! true)`.

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 ...)`

Pattern Signature	What it matches
`(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.
`(repeat n x)`	Matches exactly 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 and defaults to 0. The peg will not advance any characters.
`(> offset ?patt)`	Alias for `(look offset ?patt)`
`(to patt)`	Match up to patt (but not including it). If the end of the input is reached and patt is not matched, the entire pattern does not match.
`(thru patt)`	Match up through patt (thus including it). If the end of the input is reached and patt is not matched, the entire pattern does not match.
`(backmatch ?tag)`	If tag is provided, matches against the tagged capture. If no tag is provided, matches against the last capture, but only if that capture is untagged. The peg advances if there was a match.
`(opt patt)`	Alias for `(between 0 1 patt)`
`(? patt)`	Alias for `(between 0 1 patt)`
`(n patt)`	Alias for `(repeat n patt)`
`(sub window-patt patt)`	Match `window-patt`, and if it succeeds match `patt` against the bytes that `window-patt` matched. `patt` cannot match more than `window-patt`; it will see end-of-input at the end of the substring matched by `window-patt`. If `patt` also succeeds, `sub` will advance to the end of what `window-patt` matched.
`(split separator-patt patt)`	Split the remaining input by `separator-patt`, and execute `patt` on each substring. `patt` will execute with its input constrained to the next instance of `separator-patt`, as if narrowed by `(sub (to separator-patt) ...)`. `split` will continue to match separators and patterns until it reaches the end of the input; if you don't want to match to the end of the input you should first narrow it with `(sub ... (split ...))`.

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 the 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
`(capture patt ?tag)`	Captures all of the text in patt if patt matches. If patt contains any captures, then those captures will be pushed on to the capture stack before the total text.
`(<- patt ?tag)`	Alias for `(capture patt ?tag)`
`(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 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)`.
`(column ?tag)`	Captures the column number of the current position in the matched text.
`(line ?tag)`	Captures the line number of the current position in the matched text.
`(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`	Duplicates the last capture with the tag `tag`. If no such capture exists then the match

Pattern Signature	What it matches
`?tag)`	fails.
`(-> tag ?tag)`	Alias for `(backref tag)`.
`(unref patt ?tag)`	Lets a user "scope" tagged captures. After the rule has matched, all captures with the given tag can no longer be referred to by their tag. However, such captures from outside rule are kept as is. If no tag is given, all tagged captures from rule are unreferenced. Note that this doesn't drop the captures, merely removes their association with the tag. This means subsequent calls to backref and backmatch will no longer "see" these tagged captures.
`(error ?patt)`	Throws a Janet error. If optional argument `patt` is provided and it matches successfully, the error thrown will be the last capture of `patt`, or a generic error if `patt` produces no captures. If no argument is provided, a generic error is thrown. If `patt` does not match, no error will be thrown.
`(drop patt)`	Ignores (drops) all captures from patt.
`(only-tags patt)`	Ignores all captures from `patt`, while making tagged captures within `patt` available for future back-referencing.
`(nth index patt ?tag)`	Capture one of the captures in `patt` at `index`. If no such capture exists, then the match fails.
`(uint num-bytes ?tag)`	Capture a little endian, unsigned, two's complement integer from `num-bytes`. Works for 1 to 8 byte integers.
`(uint-be num-bytes ?tag)`	Capture a big endian, unsigned, two's complement integer from `num-bytes`. Works for 1 to 8 byte integers.
`(int num-bytes ?tag)`	Capture a little endian, signed, two's complement integer from `num-bytes`. Works for 1 to 8 byte integers.
`(int-be num-bytes ?tag)`	Capture a big endian, signed, two's complement integer from `num-bytes`. Works for 1 to 8 byte integers.
`(lenprefix n patt)`	Matches n repetitions of a pattern, where n is supplied from other parsed input and is not a constant.
`(number patt ?base ?tag)`	Capture a number parsed from the matched pattern. This uses the same parser as Janet itself, so it supports all numeric notation allowed in Janet files. To tag the capture without forcing a particular base, use `(number patt nil :tag)`.

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 combinators 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 grammars simpler and easier to read.

Name	Expanded	Description
`:d`	`(range "09")`	Matches an ASCII digit.
`:a`	`(range "az" "AZ")`	Matches an ASCII letter.
`:w`	`(range "az" "AZ" "09")`	Matches an ASCII digit or letter.
`:s`	`(set " \t\r\n\0\f\v")`	Matches an ASCII whitespace character.
`:h`	`(range "09" "af" "AF")`	Matches a hex character.
`:D`	`(if-not :d 1)`	Matches a character that is not an ASCII digit.
`:A`	`(if-not :a 1)`	Matches a character that is not an ASCII letter.
`:W`	`(if-not :w 1)`	Matches a character that is not an ASCII digit or letter.
`:S`	`(if-not :s 1)`	Matches a character that is not ASCII whitespace.
`:H`	`(if-not :h 1)`	Matches a character that is not a hex character.
`: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.
`:h+`	`(some :h)`	Matches 1 or more hex 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.
`:h*`	`(any :h)`	Matches 0 or more hex 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. Modifiying 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 on 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))))))
```


© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Executing a Process

Process Management Spawning a Process
In the simpler approach using `os/execute`, once the program is started to create a subprocess, the function does not return or error until the subprocess has finished executing. Technically, `os/execute` "waits" on the created subprocess. Further details regarding waiting will be covered when discussing `os/spawn`. The return value for `os/execute` is the exit code of the created subprocess.

args

The only required argument, `args`, is a tuple or array of strings that represents an invocation of a program along with its arguments.

```
# prints: I drank what?
# also returns 0, the exit code
(os/execute ["janet" "-e" `(print "I drank what?")`] :p) # => 0
```

flags

If there is a second argument, `flags`, it should be a keyword. `flags` can affect program launching and subprocess execution characteristics.

Note in the example above, `flags` is `:p`, which allows searching the current `PATH` for a binary to execute. If `flags` does not contain `p`, the name of the program must be an absolute path.

If `flags` contains `x`, `os/execute` will raise an error if the subprocess' exit code is non-zero.

```
# prints: non-zero exit code
(try
  (os/execute ["janet" "-e" `(os/exit 1)`] :px)
  ([_]
    (eprint "non-zero exit code")))
```

env

If `flags` contains `e`, `os/execute` should take a third argument, `env`, a dictionary (i.e. table or struct), mapping environment variable names to values. The subprocess will be started using an environment constructed from `env`. If `flags` does not contain `e`, the current environment is inherited.

```
# prints "SITTING"
# also returns 0
(os/execute ["janet" "-e" `(pp (os/getenv "POSE"))`]
            :pe {"POSE" "SITTING"}) # => 0
```

The `env` dictionary can also contain the keys `:in`, `:out`, and `:err`, which allow redirecting standard IO in the subprocess. The associated values for these keys should be `core/file` values and these should be closed explicitly (e.g. by calling `file/close`) after the subprocess has completed.

Note that the `flags` keyword argument only needs to contain `e` if making use of environment variables. That is, for use of any combination of just `:in`, `:out`, and `:err`, the `flags` keyword does not need to contain `e`.

```
(def of (file/temp))
```

```
(os/execute ["janet" "-e" `(print "tada!")`]
            :p {:out of}) # => 0

(file/seek of :set 0)

(file/read of :all) # => @"tada!\n"

(file/close of) # => nil
```

Caveat

Although it may appear to be possible to successfully use `core/stream` values with `os/execute` in some cases, it may not work in certain situations (e.g. with another operating system, different programs, varied arguments, phase of the moon, etc.). It is not a reliable choice and is thus not recommended. The `os/spawn` function is better suited for use with `core/stream` values.

Process Management Spawning a Process
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs
    - Tuples
  - Destructuring
  - Fibers
    - Dynamic Bindings
    - Errors
  - Modules
  - Object-Oriented Programming
  - Parsing Expression Grammars
  - Prototypes
  - The Janet Abstract Machine
  - The Event Loop
  - Multithreading
  - Networking

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Process Management

Janet has support for starting, communicating with, and otherwise managing processes primarily via two approaches. One is a simpler synchronous method provided by `os/execute`. Another is a more complex way that gives finer-grained control via `os/spawn` and some other `os/` functions.

The `os/execute` function will be covered first as it is simpler but also because `os/spawn` may be easier to understand once one has a sense of `os/execute`.

Networking Executing a Process
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Spawning a Process

Executing a Process Documentation
Unlike `os/execute`, which returns an exit code (or errors) after the started subprocess completes, `os/spawn` returns a `core/process` value. This value represents a subprocess which may or may not have completed its execution. In contrast to `os/execute`, which "waits" for the subprocess, `os/spawn` does not and it is recommended to explicitly arrange for this "waiting" for resource management and other purposes ("waiting" will be covered in more detail below).

The arguments for `os/spawn` are the same as those of `os/execute`. Again, the required argument, `args`, is a tuple or array representing an invocation of a program with its arguments.

```
# prints: :salutations
# the returned value, proc, is a core/process value
(let [proc (os/spawn ["janet" "-e" "(pp :salutations)"] :p)]
  (type proc)) # => :core/process
```

`core/process` and `os/proc-wait`

Passing the `core/process` value to the `os/proc-wait` function causes Janet to "wait" for the subprocess to complete execution. This is sometimes referred to as "rejoining" the subprocess.

The main reason for "waiting" is so that the operating system can release resources. Not "waiting" appropriately can lead to resources remaining unreclaimed and this can become a problem for a running system because it may run out of usable resources.

Once "waiting" has completed, the exit code for the subprocess can be obtained via the `:return-code` key. Accessing this key before "waiting" will result in a `nil` value.

```
# prints: :relax
(def proc (os/spawn ["janet" "-e" "(pp :relax)"] :p))

(get proc :return-code) # => nil

(os/proc-wait proc) # => 0

(get proc :return-code) # => 0
```

The `os/proc-wait` function takes a single argument, `proc`, which should be a `core/process` value. The return value is the exit code of the subprocess. If `os/proc-wait` is called more than once for the same `core/process` value it will error.

```
# prints: :sit-up
(def proc (os/spawn ["janet" "-e" "(pp :sit-up)"] :p))

(os/proc-wait proc) # => 0

# prints: cannot wait twice on a process
(try
  (os/proc-wait proc)
  ([e]
    (eprint e))) # => nil
```

Note that the first call to `os/proc-wait` will cause the current fiber to pause execution until the subprocess completes.

`core/process` keys and the `env` argument

As seen above, once "waiting" for a subprocess has completed, the exit code of a subprocess becomes available via the `:return-code` key. Other information about a subprocess can also be accessed via the keys of an associated `core/process` value such as `:in`, `:out`, and `:err`. On UNIX-like platforms, there is an additional `:pid` key.

The `:in`, `:out`, and `:err` keys for a `core/process` value provide access to the standard input, output, and error of the associated subprocess provided that corresponding choices were made via the `env` dictionary (i.e. table or struct) argument to `os/spawn`. This means that, for example, if there is no key-value pair specified via `:in` in the `env` dictionary, then the standard input of the subprocess will not be programmatically accessible via the associated `core/process`'s `:in` key.

The values associated with the `:in`, `:out`, and `:err` keys of the `env` argument can be `core/file` or `core/stream` values.

```
(def out-file (file/temp))

(type out-file) # => :core/file

(def proc
  (os/spawn ["janet" "-e" `(print "again") (flush)`]
            :p {:out out-file}))

# only standard output of the subprocess is accessible
(proc :in) # => nil
(type (proc :out)) # => :core/stream
(proc :err) # => nil

(os/proc-wait proc) # => 0

(file/seek out-file :set 0)

# prints: again
(print (file/read out-file :all))

(file/close out-file)
```

Note that in the example above, the standard input and error of the subprocess were not accessible -- attempts to access them resulted in `nil` values -- because corresponding keys for the `env` argument were not specified. In contrast, because a key-value pair for `:out` and `out-file` were included in the `env` struct, the standard output of the subprocess could be accessed programmatically.

`:pipe` and `ev/gather`

For each of the `:in`, `:out`, and `:err` keys of the `env` argument to `os/spawn`, the associated value can be the keyword `:pipe`. This causes `core/stream` values to be used that can be read from and written to for appropriate standard IO of the subprocess.

In order to avoid deadlocks (aka hanging), it's important for data to be transferred between processes in a

timely manner. One example of how this can fail is if output redirected to pipes is not read from (sufficiently). In such a situation, pipe buffers might become full and this can prevent a process from completing its writing. That in turn can result in the writing process being unable to finish executing. To keep data flowing appropriately, apply the `ev/gather` macro.

In the following example, `ev/write` and `os/proc-wait` are both run in parallel via `ev/gather`. The `ev/gather` macro runs a number of fibers in parallel (in this case, two) on the event loop and returns the gathered results in an array.

```
(def proc
  (os/spawn ["janet" "-e" "(print (file/read stdin :all))"]
            :p {:in :pipe}))

(ev/gather
  (do
    # leads to printing via subprocess: know thyself
    (ev/write (proc :in) "know thyself")
    (ev/close (proc :in))
    :done)
  (os/proc-wait proc)) # => @[:done 0]
```

os/proc-close

The next example is a slightly modified version of the previous one. It involves adding a third fiber (to `ev/gather`) for reading standard output from the subprocess via `ev/read`.

```
(def proc
  (os/spawn ["janet" "-e" "(print (file/read stdin :all))"]
            :p {:in :pipe :out :pipe}))

(def buf @"")

(ev/gather
  (do
    (ev/write (proc :in) "know thyself")
    (ev/close (proc :in))
    :write-done)
  # hi! i'm new here :)
  (do
    (ev/read (proc :out) :all buf)
    :read-done)
  (os/proc-wait proc)) # => @[:write-done :read-done 0]

(os/proc-close proc) # => nil

buf # => @"know thyself\n"
```

Beware that if pipe streams are not closed soon enough, the process that created them (e.g. the `janet` executable) may run out of file descriptors or handles due to process limits enforced by an operating system.

Note the use of the function `os/proc-close` in the code above. This function takes a `core/process` value and closes all of the unclosed pipes that were created for it via `os/spawn`. In the example above, `(proc :in)` had already been closed explicitly, but `(proc :out)` had not and is thus closed by `os/proc-close`.

The `os/proc-close` function will also attempt to wait on the subprocess associated with the passed `core/process` value if it has not been waited on already. The function's return value is `nil` if waiting was not attempted and otherwise it is the exit code of the subprocess corresponding to the `core/process` value.

Effects of `:x`

If the `flag` keyword argument contains `x` for an invocation of `os/spawn` and the exit code of the subprocess is non-zero, calling a function such as `os/proc-wait`, `os/proc-close`, and `os/proc-kill` (if invoked to wait) with the subprocess' `core/process` value results in an error.

```
(def proc
  (os/spawn ["janet" "-e" "(os/exit 1)"] :px))

(defn invoke-an-error-raiser
  [proc]
  (def zero-one-or-two
    (-> (os/cryptorand 8)
        math/rng
        (math/rng-int 3)))
  # any of the following should raise an error
  (case zero-one-or-two
    0 (os/proc-wait proc)
    1 (os/proc-close proc)
    2 (os/proc-kill proc true)))

# prints: error
(try
  (invoke-an-error-raiser proc)
  ([_]
    (eprint "error")))
```

Executing a Process Documentation
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs
    - Tuples

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 2025

Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3

See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 13 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
(++ a)
(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 improved debugging, name with symbol or keyword
(fn alice [] :smile)
(fn :bob [] :sit)
```

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 when-true or when-false is evaluated, this is done in the context of a newly introduced lexical scope.

```
(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`.

Note that the `while` special form introduces a new lexical scope.

```
(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.

(upscope & body)

Similar to `do`, `upscope` evaluates a number of forms and sequence and evaluates to the result of the last form. However, `upscope` does not create a new lexical scope, which means that bindings created inside it are visible in the scope where upscope is declared. This is useful for writing macros that make several def and var declarations at once.

```
(upscope
 (def a 1)
 (def b 2)
 (def c 3)) # -> 3

(+ a b c) # -> 6
```

In general, use this macro as a last resort. There are other, often better ways to do this, including using destructuring.

```
(def [a b c] [1 2 3])
```

Syntax and the Parser Numbers and Arithmetic
© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- debug
- ev
- ffi
- fiber
- file
- int
- jpm
    - rules
    - cc
    - cgen
    - cli
    - commands
    - config
    - make-config
    - dagbuild
    - pm
    - scaffold
    - shutil
- math
- module
- net
- os
- peg
- parser
- spork
    - argparse
    - base64
    - cc
    - cjanet
    - crc
    - channel
    - cron
    - data
    - ev-utils
    - fmt
    - generators
    - getline
    - htmlgen
    - http
    - httpf
    - infix
    - json
    - mdz
    - math
    - misc
    - netrepl
    - pgp
    - build-rules
    - path
    - randgen
    - rawterm

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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 struct - 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 double 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"

# If your double-quoted string spans multiple lines, newline characters will
# be removed but other whitespace is preserved.
"Hello, this
    is al
l one line
here."   # -> "Hello, this    is all one linehere."

# 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, ð   "
```

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 5) # -> "cde"
(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 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols

- Array C API
- Buffer C API
- Table C API
- Fiber C API
- Memory Model
- Writing C Functions

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 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-function
*****
!%$^*__--__._+++===-crazy-symbol
*global-var*
ä½ å¥½
```

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)
- `\uxxxx` Escape UTF-8 codepoint with 4 hex digits.
- `\Uxxxxxx` Escape UTF-8 codepoint with 6 hex digits.
- `\\` 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.

```
# same as "Thisisastring"
"This
```

```
is
a
string"
```

Long-strings

An alternative way of representing strings in Janet is the long-string, or the backquote-delimited string. A long-string can be defined to start with a certain number of backquotes, and will end the same number of 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 containing literal newline characters, unprintable characters, or strings that would otherwise require many escape sequences.

```
# same as "This\nis\na\nstring."
``
This
is
a
string
``

# same as "This is\na string."
`
This is
a string.
`
```

Long-strings are often used for docstrings, as described in the Documentation chapter.

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, `\0`, `\f`, `\n`, `\r`, `\t`, or `\v` 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 `math/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 `"'0?\abefnrtvz`)
                       (* "x" :hex :hex)
                       (* "u" [4 :hex])
                       (* "U" [6 :hex])
                       (error (constant "bad escape"))))
    :comment (* "#" (any (if-not (+ "\n" -1) 1)))
    :symbol :token
    :keyword (* ":" (any :symchars))
    :constant (* (+ "true" "false" "nil") (not :symchars))
    :bytes (* "\"" (any (+ :escape (if-not "\"" 1))) "\"")
    :string :bytes
    :buffer (* "@" :bytes)
    :long-bytes {:delim (some "`")
                 :open (capture :delim :n)
                 :close (cmt (* (not (> -1 "`")) (-> :n) '(backmatch :n)) ,=)
                 :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 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub

- Home
- Documentation
  - Syntax and the Parser
  - Special Forms
  - Numbers and Arithmetic
  - Comparison Operators
  - Bindings (def and var)
  - Flow
  - Functions
  - Strings, Keywords, and Symbols
  - Looping
  - Macros
  - Data Structures
    - Arrays
    - Buffers
    - Tables
    - Structs
    - Tuples
  - Destructuring
  - Fibers
    - Dynamic Bindings
    - Errors
  - Modules
  - Object-Oriented Programming
  - Parsing Expression Grammars
  - Prototypes
  - The Janet Abstract Machine
  - The Event Loop
  - Multithreading
  - Networking
  - Process Management
    - Executing
    - Spawning
  - Documentation
  - jpm
  - Linting

Janet 1.38.0-73334f3 Documentation
(Other Versions: 1.37.1 1.36.0 1.35.0 1.34.0 1.31.0 1.29.1 1.28.0 1.27.0 1.26.0 1.25.1 1.24.0 1.23.0 1.22.0 1.21.0 1.20.0 1.19.0 1.18.1 1.17.1 1.16.1 1.15.0 1.13.1 1.12.2 1.11.1 1.10.1 1.9.1 1.8.1 1.7.0 1.6.0 1.5.1 1.5.0 1.4.0 1.3.1)

Multithreading

The Event Loop Networking
Multithreading is the process of running a program on multiple threads at the same time, usually to improve throughput. Threads allow your program to take full advantage of the multiple processors on modern CPUs letting you do work in the background without stopping the main program flow, or breaking up an expensive operation to run on multiple processors.

Janet's `ev/` module supports spawning native operating system threads in a way that is compatible with other `ev/` functions and will not block the event loop.

For the most part, Janet values are not shared between 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 with threaded channels.

By default, a Janet program will not exit until all threads have terminated.

Creating threads

The most primitive way to create a thread is `(ev/thread fiber &opt value flags supervisor)`. This will start and wait for a message containing a function that it will run as the main body.

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

# Create a new thread and wait for it to complete.
(ev/thread (fiber/new worker :t))
```

By itself, the above code isn't very useful because the main fiber will suspend until the new thread is complete. But it is quite useful to have threads suspend execution of the calling fiber by default - we can then easily have a thread wrapped with a fiber to be handled like other asynchronous tasks in the `ev/` module. To run the thread in the background, you can either use the `:n` flag, or wrap the call to `ev/thread` in its own fiber.

```
(ev/thread (fiber/new worker :t) nil :n)

(ev/spawn
  (ev/thread (fiber/new worker :t)))
```

To make this process easier, Janet comes with a few built-in macros, `ev/spawn-thread` to run a block of code in a new thread, return immediately, and `ev/do-thread` to run a block of code but wait for it to return.

```
(ev/spawn-thread
  (print "New thread started!"))

(ev/do-thread
  (print "New thread started!"))
```

Sending and receiving messages

Threads in Janet do not share a memory heap and must communicate via threaded channels. Threaded channels behave much like normal channels in the `ev/` module, with the only difference being that they can send values between threads by copying messages. Threaded channels are used for both communication and coordination between threads.

Threaded channels can be created with `ev/thread-chan`.

```
# Create a threaded channel with space for 10 values
(def thread-channel (ev/thread-chan 10))

(ev/spawn-thread
 (repeat 10
  (def item (ev/take thread-channel))
  (print "got " item)))

(repeat 10
 (os/sleep 0.2)
 (ev/give thread-channel :item))
```

Thread Supervisors

Threaded channels can also be used as supervisors for spawned threads. A supervisor is a channel that receives messages whenever an event, like an error, occurs in the supervised thread. Another fiber or thread can then read from this supervisor channel and handle the errors, usually by either logging the event, retrying the operation, or canceling other operations.

Thread supervisors need to be specified when creating the thread.

```
(def supervisor (ev/thread-chan 10))

(defn worker
 []
 (repeat 10
  (if (< 0.9 (math/random))
   (error "oops!")))
 (print "done!"))

# Start a worker thread that will signal events on the supervisor channel
(ev/thread worker nil :n supervisor)

# Get one event from the supervisor channel (on the initial thread here)
# It will either be (:error "oops!" nil) or (:ok nil nil).
(def event (ev/take supervisor))
(pp event)
```

Events from thread supervisors are much like events from normal fiber supervisors, but the first argument is not an entire copy of the fiber or thread, it is the event name. In the above example, depending on whether or not the "oops!" error was triggered, `event` will be either `(:error "oops!" nil)` or `(:ok nil nil)`. `:error` and `:ok` correspond to `(fiber/status f)`, while `"oops!"` and `nil` to `(fiber/last-value)` of the main fiber of the child thread. The third element of the tuple is the associated task id, if the fiber has an associated environment, or `nil` otherwise.

The Event Loop Networking

© Calvin Rose & contributors 2025
Generated on April 5, 2025 at 13:24:06 (UTC) with Janet 1.38.0-73334f3
See a problem? Source on GitHub