

Jibi Scheme

version 0.1.4

Overview

A homebrew interpreted, non-RnRS compliant dialect of Scheme.

Types

Primitive Types

string

`"some-string"`

String are immutable.

Evaluation Rule: A **string** value evaluates to itself.

symbol

`some-symbol`

All **symbol** values are interned, therefore `(eq? 'some-symbol 'some-symbol)` is true.

Evaluation Rule: **symbol** values are variable names. When evaluated, a **symbol** is replaced by the value of its binding in the nearest enclosing scope where it is defined. An error is raised if **symbol** is not bound in any enclosing scope.

integer

`100`

The underlying type for **integer** is `i128`. Integer overflow terminates the program.

Evaluation Rule: An **integer** value evaluates to itself.

bool

true
false

Only **bool** have truth values, therefore they are the only type that can be used as predicates, e.g. for **if**.

Evaluation Rule: A **bool** value evaluates to itself.

nil

nil

In Jibi Scheme, **nil** and all empty lists **()** are the same object, therefore **(eq? () ())** is true.

Evaluation Rule: **nil** evaluates to itself.

Composite Types

pair

```
(cons :expr :expr)
```

The **pair**, also known as cons cell, is the basic Scheme compound data type. It is simply a grouping of two values of any types (2-tuple); the first and second values are sometimes referred to respectively as the **car** and **cdr**.

Evaluation Rule: **pair** values are evaluated by procedure application, however, only **pair** values which are **list**'s can be properly applied; evaluating a non-list **pair** raises an error.

list

```
; code  
(:callable :expr ...)  
; data  
(  
(cons :expr (cons ()))  
(list :expr ...)
```

A **list** value is either the empty list `()`, or ordered **pair**'s terminated by `()`, where the **car** of the **pair** is an element of the list, and the **cdr** is the rest of the list.

Scheme data and code are both represented as lists, which makes meta-programming easy and fun. See Quoting and Evaluation and Macro Definition.

Evaluation Rule: The first value of the list is applied (called) with the rest of the list as arguments. If the first value of the list is not **callable**, an error is raised. Exception: the empty list `()` is not applied, it evaluates to itself. See Function Definition.

Special Types

quote

```
(quote :expr)
'expr
```

Any expression can be quoted, using either the **quote** form or a starting apostrophe '.

Evaluation Rule: A quoted expression evaluates to the expression. This is useful to prevent **symbol** binding and procedure application. See Quoting and Evaluation.

error

```
(error type "some-message")
```

Error values do not inherently do anything, until they are **raise**'d as exceptions. See Exceptions.

Evaluation Rule: An **error** value evaluates to itself.

Callable Types

lambda

`(fn params :expr ...)`

See Function Definition.

Evaluation Rule: A **lambda** value evaluates to itself. It is applied when it is the first element of a **list**.

procmacro

`(macro params :expr ...)`

Procedural macros. See Macro Definition.

Evaluation Rule: A **procmacro** value evaluates to itself. It is applied when it is the first element of a **list**.

Builtin Callable Types

function

`; not constructable`

Opaque type containing a builtin function.

Evaluation Rule: A **function** value evaluates to itself. It is applied when it is the first element of a **list**.

specialform

`; not constructable`

Opaque type containing a builtin macro.

Evaluation Rule: A **specialform** value evaluates to itself. It is applied when it is the first element of a **list**.

Forms

Binding and Assignment

def

```
(def name :expr)
```

Create and assign binding in local scope.

set!

```
(set! name :expr)
```

Change existing binding. Raises error if a binding does not already exists.

let

```
(lets ((name value:expr) ...) :expr ...)
```

Create bindings in a new local scopes.

```
>>> ; Example
>>> (let ((x 5) (y 7))
...   (display x)
...   (display y))
5
7
```

defglobal

```
(defglobal name :expr)
```

Create and assign binding in global env.

setglobal!

```
(setglobal! name :expr)
```

Change existing binding in global env. Raises error if a binding does not already exists.

Function Definition

defn

```
(defn name parameters :expr ...)
```

Create lambda function and bind it to **name**.

Variadic lambdas can be defined with formal parameters like (**x . xs**) - there must be a single parameter after **.**, which will be a list containing zero or more arguments depending on the number of arguments passed.

```
>>> ; Example
>>> (defn increment (x) (+ x 1))
>>> (increment 1)
2
>>> (defn variadic (x y . rest) rest)
>>> (variadic 1)
Unhandled ApplyError "expected at least 2 argument(s)"
>>> (variadic 1 2)
()
>>> (variadic 1 2 3 4)
(3 4)
```

fn

```
(fn parameters :expr ...)
```

Create a lambda (function). See **defn**.

Control Flow

begin

```
(begin :expr ...)
```

Evaluate expressions sequentially and return value of last expression.

if

```
(if predicate:bool then:expr else:expr)
```

Evaluates only **then** or **else** conditionally on the value of **predicate**.

cond

```
(cond (predicate:bool :expr ...) ...)
```

Evaluates body of the first clause which has a true predicate.

```
>>> ; Example
>>> (cond
...   (false (assert false))
...   ((not true) (assert false))
...   (else (print "foo") (print "bar")))
foo
bar
```

or

`(or expr expr)`

Logical or. Short-circuiting; if the first expression evaluates to true, the second expression is not evaluated.

and

`(and expr expr)`

Logical and. Short-circuiting; if the first expression evaluates to false, the second expression is not evaluated.

Comparison

eq?

`(eq? :expr :expr)`

Identity comparison. Check if two values are the same object.

equal?

`(equal? :expr :expr)`

Value comparison. Check if two values are equal.

Logical Operators

not

```
(not :bool)
```

Logical not.

any

```
(any :bool ...)
```

Returns true if any argument is true.

all

```
(all :bool ..)
```

Returns true if all arguments are true.

Pair and List Operations

cons

```
(cons left:expr right:expr)
```

Construct a pair.

car

```
(car :pair)
```

Get first item of a pair (head of list).

cdr

```
(cdr :pair)
```

Get second item of a pair (rest of list).

list

```
(list :expr ...)
```

Construct a list, which is a linked list made from pairs and terminated by `nil`.

```
>>> ; Example
>>> (equal? (list 1 2 3) (cons 1 (cons 2 (cons 3 nil))))
true
>>> (equal? (list 1 2 3) (cons 1 (list 2 3)))
true
```

lcons

`(lcons :expr ... :list)`

Prepend values to a list.

```
>>> ; Example
>>> (lcons 1 2 3 (list 4 5))
(1 2 3 4 5)
```

nth

`(nth :integer :list)`

Get nth item from a list (zero-indexed).

```
>>> ; Example
>>> (nth 3 (list 0 1 2 3 4))
3
```

empty?

`(empty? :expr)`

Check if value is the empty list (nil).

list?

`(list? :expr)`

Check if value is a nil-terminated list of ordered pairs.

map

```
(map f:procedure vals:list)
```

Applies **f** to each value in a list and return results in list.

```
>>> ; Example  
>>> (map (fn (x) (* 2 x)) (list 1 2 3))  
(2 4 6)
```

fold

```
(fold f:procedure init:expr vals:list)
```

Applies **f** to each value in a list and accumulate results in **init**.

```
>>> ; Example  
>>> (fold + 0 (list 1 2 3))  
6  
>>> (fold cons () (list 1 2 3))  
(3 2 1)
```

range

```
(range from:integer to:integer)
```

Produce list of integers for range [**from**, **to**), where **to** \geq **from**.

String Operations

len

`(len :string)`

Get length of string (number of UTF-8 scalar values).

concat

`(concat :string ...)`

Concatenate multiple strings.

```
>>> ; Example
>>> (concat "foo" "bar" "baz")
"foobarbaz"
```

replace

`(replace :string :string :string)`

Description.

```
>>> ; Example
>>> (replace "fuzzy bears are fuzzy" "fuzzy" "long")
"long bears are long"
```

substring

```
(substring :string start:integer end:integer)
```

Get a substring. Negative indices count from the end of the string. If `start > end`, the substring is reversed.

```
>>> ; Example
>>> (substring "foobar" 1 -1)
"ooba"
>>> (substring "foobar" 6 0)
"raboof"
```

split

```
(split :string separator:string)
```

Split a string by separator.

```
>>> ; Example
>>> (split "12.34.56" ".")
("12" "34" "56")
```

contains?

```
(contains? str:string substr:string)
```

Check if `str` contains `substr`.

```
>>> ; Example
>>> (contains? "foobar" "foo")
true
```

parse-integer

```
(parse-integer :string)
```

Parse an integer from a string.

```
>>> ; Example  
>>> (parse-integer "12")  
12
```

left-pad

```
(left-pad string:string char:string width:integer)
```

Pad string to width characters.

```
>>> ; Example  
>>> (left-pad "34" "0" 4)  
0034
```

Integer Operations

Add: +

`(+ :integer ...)`

Addition.

Sub: -

`(- :integer ...)`

Negation (single argument) or subtraction (multiple arguments).

Mul: *

`(* :integer ...)`

Multiplication.

Div: /

`(/ :integer ...)`

Reciprocal (single argument) or division (multiple arguments).

Comparisons: >, >=, <, <=

`(> :integer :integer)`

`(> :integer :integer)`

`(> :integer :integer)`

`(> :integer :integer)`

Comparisons.

Printing

print

```
(print :string)
```

Print a string.

repr

```
(repr :expr)
```

Get string representation of a value.

display

```
(display :expr)
```

Print string representation of a value.

Modules

`jibi` has a basic namespaced module system. A module is simply a `.jibi` file.

They provide no privacy, all variables defined in the module scope are accessible to importers.

Module files are only evaluated once, re-importing gets a reference to the existing module.

At the moment only the current working directory is searched to find modules, thus importing `"stl/unittest"` looks for the file `./stl/unittest.jibi'`.

import

```
(import module:string as name)
```

Import module and bind it to `name`.

```
>>> ; Example
>>> (import "stl/math" as math)
>>> (math::product (list 2 3 4))
24
```

use

```
(use module:symbol name ...)
```

Bind a name from a module into the global scope.

```
>>> ; Example
>>> (import "stl/math" as math)
>>> (use math product sum)
>>> (product (list 2 3 4))
24
```

import-from

```
(import-from module:string name ...)
```

Import specific names from a module.

```
>>> ; Example
>>> (import-from "stl/math" product sum)
>>> (sum (list 2 3 4))
9
>>> (product (list 2 3 4))
24
```

Type Inspection

type

```
(type :expr)
```

Inspect type of a value.

```
>>> ; Example
>>> (type "foo")
string
```

type?

```
(type? :expr type)
(string? :expr)
(symbol? :expr)
...
```

Test type of a value. There are also convenience functions for every type.

```
>>> ; Example
>>> (type? "foo" string)
true
>>> (integer? "foo")
false
```

Quoting and Evaluation

quote

`(quote :expr)`

A quoted expression evaluates to the expression.

```
>>> ; Example
>>> (def a 100)
>>> a
100
>>> (quote a)
a
>>> (+ 5 5)
10
>>> (quote (+ 5 5))
(+ 5 5)
```

eval

`(eval :expr)`

Evaluate an expression.

```
>>> ; Example
>>> (def expr (quote (+ 5 5)))
>>> expr
(+ 5 5)
>>> (eval expr)
10
```

evalfile

```
(evalfile filename:string)
```

Evaluate file in the global environment.

apply

```
(apply :procedure :list)
```

Apply a procedure to a list of arguments.

```
>>> ; Example  
>>> (apply + (list 1 2 3))  
6
```

Macro Definition

defmacro

```
(defmacro name formals :expr ...)
```

jibi macros are “procedural”; they are simply lambdas which return code.

The body of the macro is first evaluated in the macro’s lexical environment. Then the resulting expression is evaluated in the caller’s environment.

Beware of capturing variables from the macro’s environment; if you want to refer to variables in the invocation environment, use quotation.

This `add-x` macro captures the global binding for `x`:

```
>>> (defmacro add-x (y) (list + x y))
>>> (def x 100)
>>> (add-x 5)
105
>>> (set! x 200)
>>> (add-x 5)
205
>>> ((fn (x) (add-x 5)) 1000)
205
```

In this version, `x` is not captured; the value of `x` is taken from the local scope where the macro is called:

```
>>> (def x 100)
>>> (defmacro add-x (y) (list + 'x y))
>>> ((fn (x) (add-x 5)) 1000)
1005
```

macro

```
(macro formals :expr ...)
```

Create macro. See ‘defmacro’.

Exceptions

Errors can be raised to interrupt program flow, and can be caught with the `try` form.

error

```
(error type:symbol reason:string)
```

Create error with custom type.

exception

```
(exception reason:string)
```

Create error of type `Exception`.

raise

```
(raise :error)
```

Raise an error (can be any error type, not just `Exception`).

try

```
(try body:expr catch:expr)
```

Try evaluating `body`. If an error is raised, evaluate `catch`; the raised error value is bound to `err` when `catch` is evaluated.

```
>>> ; Example
>>> (defn errored ()
...   (raise (exception "oh no!"))
...   (print "never evaluated"))
>>> (errored)
Unhandled Error: oh no!
>>> (try (print "no error") (print (concat "handled " (repr err))))
no error
>>> (try (errored) (print (concat "handled " (repr err))))
handled #[error Exception "oh no!"]
```

assert

```
(assert predicate:bool)
```

Raises an exception if `predicate` is false.

Environment Procedures

env

`(env)`

Get the nearest enclosing environment (most local scope).

env-lookup

`(env-lookup :env :symbol)`

Look up symbol in the given environment.

env-def

`(env-def :env :symbol :expr)`

Define symbol in the given environment.

env-set

`(env-set! :env :symbol :expr)`

Set symbol in the given environment.

env-parent

```
(env-parent :env)
```

Get parent env, or `nil` if there is no parent env.

env-globals

```
(env-globals)
```

Get the global environment.

System Procedures

getenv

`(getenv var:string)`

Get value of environment variable. Raises exception if the variable is not set or contains non-UTF8 characters.

exit

`(exit :integer)`

Exit program with a status code.

Debugging

dd

`(dd :expr)`

Print Rust struct debug.

ddp

`(ddp :expr)`

Pretty print Rust struct debug.

dda

`(dda :expr)`

Print pointer address.

ddc

`(ddc :lambda|:procmacro)`

Print code of (non-builtin) lambda or macro.

ddm

`(ddm :procmacro :expr ...)`

Print code generated by a `procmacro` for the given arguments.

Standard Library (STL)

I hope you were not expecting a real standard library. You can have some unit testing and maths as consolation.

stl/math

Some mathematical functions.

sign

`(sign :number)`

Returns sign of number.

```
>>> ; Example
>>> (sign -12)
-1
>>> (sign 100)
1
>>> (sign 0)
0
```

abs

`(abs :number)`

Returns absolute value of number

```
>>> ; Example
>>> (abs -12)
12
```

remainder

`(remainder :integer :integer)`

Returns the least positive remainder for integer division.

```
>>> ; Example
>>> (remainder 42 5)
2
>> (remainder 42 -5)
2
```

pow

`(pow base:number exponent:number)`

Perform exponentiation.

```
>>> ; Example
>>> (pow 10 3)
1000
>>> (pow 2 10)
1024
```

sum

`(sum :list)`

Returns sum of list.

```
>>> ; Example
>>> (sum (list 1 2 3 4))
10
```

product

`(product :list)`

Returns product of list.

```
>>> ; Example
>>> (product (list 1 2 3 4))
24
```

min

`(min :number :number)`

Returns smallest of 2 values.

```
>>> ; Example
>>> (fold min INTMAX (list 21 321 421 -12))
421
```

max

`(max :number :number)`

Returns largest of 2 values.

even?

`(even? :number)`

Check if even.

odd?

```
(odd? :number)
```

Check if odd.

factorial

```
(factorial :number)
```

Compute factorial.

```
>>> ; Example  
>>> (factorial 4)  
24
```

stl/decimal

The `decimal` module implements floating point decimal arithmetic.

By default, multiplication and division produce results with a maximum precision of 10 decimal places. This can be changed with `set-precision`, but since decimals are implemented with `i128`, a high precision can cause multiplication and division to overflow, in which case an error will be raised.

Importing the `decimal` module overloads the following functions:

- Arithmetic operators: `+`, `-`, `*`, `/`
- Comparison operators: `equal?`, `<`, `<=`, `>`, `>=`
- Display: `repr`, `display`

Note that division is defined as floor division when the divisor is an integer:

```
>>> ; Example
>>> (import-from "stl/decimal" decimal)
>>> (/ (decimal 5) (decimal 2))
2.5
>>> (/ (decimal 5) 2)
2.
```

Other functions defined in terms of arithmetic operations will work with decimal values once `stl/decimal` is imported. These include `range`, and all functions from `stl/math` - with the caveat that some math functions are only defined for whole numbers:

```
>>> ; Example
>>> (import-from "stl/decimal" decimal)
>>> (import-from "stl/math" remainder pow even?)
>>> ; pow truncates the exponent (but not the base)
>>> (pow (decimal "2.5") 3)
15.625
>>> (pow (decimal "2.5") (decimal "3.5"))
15.625
>>> ; even? and odd? only check the whole number part
>>> (even? (decimal "2.5"))
true
>>> ; floor division, and remainder
>>> (/ (decimal "4.5") 2)
2
>>> (remainder (decimal "4.5") 2)
.5
```

```

>>> ; (2 * 2) + 0.5 = 4.5, makes sense
>>> ;
>>> ; true division and remainder
>>> (/ (decimal "4.5") (decimal 2))
2.25
>>> (remainder (decimal "4.5") (decimal 2))
.000
>>> ; (2.0 * 2.25) + 0.0 = 4.5, so that also makes sense, but not very useful

```

Decimal numbers are represented as a an integer coefficient and an implicitly negated integer exponent, with base 10. The exponent encodes the number of significant digits, so for example 2.5 is represented as $(25 \cdot 10^{-1})$, which is 25×10^{-1} , while 2.50 is represented as $(250 \cdot 10^{-2})$, which is 250×10^{-2} .

decimal

```
(decimal :integer|:string|:decimal)
```

Convert value to a decimal. Raises an error if an unsupported type is given.

```

>>> ; Example
>>> (import-from "stl/decimal" decimal)
>>> (+ 1 2 3)
6
>>> (+ (decimal "12.5") (decimal "0.25") 1)
13.75

```

round

```
(round :decimal n:integer)
```

Round to n decimal places. Rounds up if the next digit is ≥ 5 .

set-precision

`(set-precision :integer)`

Change maximum precision of decimals returned by multiplication and division operations.

coef

`(coef)`

Get the coefficient

expn

`(expn)`

Description.

stl/unittest

Write and run unit tests with assertions.

test

```
(test name:string :expr ...)
```

A test is simply one or more expressions. It is considered a success if no exceptions are raised when the body of the test is evaluated.

test-suite

```
(test-suite name:string (test ...) ...)
```

Execute a series of tests, print a summary of the results, and raise an error if any of the tests failed.

Set the environment variable `TEST_VERBOSE` to 1 to print more details.

```
; sometests.jibi
(import-from "stl/unittest" test test-suite assert-equal)
(test-suite "very useful tests"
  (test "correct" (assert-equal true true))
  (test "suspicious" (assert-equal true false)))

$ jibi sometests.jibi
Testing very useful tests...
Test suspicious: failed ([error Exception: true is not equal to false])
Test results: 1 ok, 1 failed
Traceback:
...
Exception: Some tests failed.
```

assert-not

```
(assert-not :bool)
```

Check if false.

assert-eq

```
(assert-eq :expr :expr)
```

Check (identity) equality.

assert-equal

```
(assert-equal :expr :expr)
```

Check (value) equality.

assert-type

```
(assert-type :expr type)
```

Check type of expression.

assert-raise

```
(assert-raise :expr)
```

Check that the given expression raises an error when evaluated.