Jibi Scheme

version 0.1.8

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 defined in any enclosing scope.

---

integer

```
100
```

The underlying type for `integer` is `i128`. Integer overflow raises an exception.

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

---

float

```
1.5
```

The underlying type for `float` is `f64`.

Evaluation Rule: A `float` 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

```
(exception "some reason")
(error type "some reason")
```

Error values do no 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`.

---

macro

```
(macro params :expr ...)
```

Procedural macros. See Macro Definition.

Evaluation Rule: A `macro` 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)
```

Define `name` in the current local scope.

---

set!

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

Change value of `name` in the nearest enclosing scope where it is defined. Raises an error if `name` is not defined in any enclosing scope.

---

let

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

Define variables in a new local scopes.

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

---

defglobal

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

Define a global variable.

---

setglobal!

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

Change value of global variable. Raises error if the global variable is not defined.

Function Definition

defn

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

Creaet and define a lambda function as `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` conditonally 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)
```

---

foldr

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

Applies `f` to each value in list (right first) and accumulate results in `init`.

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

---

foldl

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

Applies `f` to each value in list (left first) and accumulate results in `init`.

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

---

range

```
(range from:number to:number [step:number])
```

Produce list of numbers for range [`from`, `to`], with an optional step size.

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
```

---

left-pad

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

Pad string to `width` characters.

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

Numerical Operations

If different number types are mixed, integers get promoted to floats (may raise an error if the integer is too large or small to be represented as a float).

Add: +

```
(+ :number ...)
```

Addition.

Sub: -

```
(- :number ...)
```

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

Mul: *

```
(* :number ...)
```

Multiplication.

Div: /

```
(/ :number ...)
```

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

Numeric Comparison: =, >, >=, <, <=

```
(= :number :number)
(> :number :number)
(>= :number :number)
(< :number :number)
(<= :number :number)
```

Compare numerical values.

Type Conversions

string

```
(string :expr)
```

Convert value to string.

---

integer

```
(integer :float|:string)
```

Convert value to integer.

---

float

```
(float :integer|:string)
```

Convert value to float.

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.

Importing looks for module files in the following locations in order:

- Paths in the `JIBI_PATH` environment variable (separated by `:`)
- The jibi system library path (dependent on PREFIX environment variable at build time, default: /usr/local/lib/jibi)
- The current working directory when the interpreter was launched

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.

---

paths

```
(paths)
```

Print modules import paths.

Reader Macros

Reader macros are macros that operate on lexical tokens, before parsing. They allow extending the syntax of the language.

A reader macro consists of a rule, and a transformer. The rule specifies a pattern of tokens to which the macro applies. Whenever the reader encounters a sequence of tokens that matches the pattern, the transformer is applied.

The transformer is a lambda which takes the matching token sequence as input, and returns a list of tokens to replace them.

Reader macros are applied in the order in which they were installed (with the `reader-macro!` procedure).

```
>>> ; Example
>>> ;
```

token

```
(token type [value])
```

Used to produce tokens in reader macro transformer functions.

```
>>> ; Example
>>> (token 'lparen)
#[token LPAREN]
>>> (token 'string "foo")
#[token STRING("foo")]
```

---

token-match

```
(token-match type [value])
```

A matcher for lexical tokens. A reader macro rule consists of a list of token matchers.

```
>>> ; Example
>>> (token-match 'string)
#[tokenmatcher String(#ANY)]
>>> (token-match 'any)
#[tokenmatcher #ANY]
```

---

token-value

```
(token-value :token)
```

Get value of token (or nil for tokens that have no value).

```
>>> ; Example
>>> (token-value (token 'string "foo"))
foo
```

---

token-type

```
(token-type :token)
```

Get type of token (symbol).

```
>>> ; Example
>>> (token-type (token 'lparen))
lparen
```

---

reader-macro!

```
(reader-macro! tokenmatcher [... tokenmatcher] transformer)
```

Install a new reader macro with the provided rule and transformer.

The first n arguments are token matchers to match sequences of 1 or more tokens.

The last argument is the token transfomer function to apply to (non-overlapping) sequences of tokens that match the rule.

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|:macro)`

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

---

ddm

`(ddm :macro :expr ...)`

Print code generated by a `macro` 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 :number :number)
```

Returns the least positive remainder for integer floor division. (Returns zero for floating point division, or very close to zero, because of floating point errors.)

```
>>> ; 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 raise errors due to overflow.

Importing the decimal module overloads and adds support for decimal types to the following builtin functions:

- Arithemic operators: `+`, `-`, `*`, `/`
- Comparison operators: `=`, `<`, `<=`, `>`, `>=`
- Type conversions: `string`, `float`, `integer`
- 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.
```

Procedures defined in terms of basic numerical procedures will work with decimal values once `stl/decimal` is imported - such as `range`, and all functions from `stl/math` - with the caveat that some math functions may truncate or apply floor divisions to their arguments:

```
>>> ; 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 integer part of non-whole numbers
>>> (even? (decimal "2.5"))
true
```

```
>>> ; floor division, and remainder
>>> (/ (decimal "4.5") 2)
2
>>> (remainder (decimal "4.5") 2)
.5

>>> ; true division and remainder
>>> (/ (decimal "4.5") (decimal 2))
2.25
>>> ; it's technically correct that the remainder of true division is always zero,
>>> ; but not very useful (may return not exactly zero due to rounding errors)
>>> (remainder (decimal "4.5") (decimal 2))
.000
```

Decimal numbers are represented as a an integer coefficient and an (implicitly negative) integer exponent, with base 10. The exponent encodes the number of significant digits, such that 2.5 is represented as `(25 . 1)`, meaning $25 \mathrm{x} 10^{-1}$, while 2.50 is represented as `(250 . 2)`, meaning $250 \mathrm{x} 10^{-2}$.

decimal

`(decimal :integer|:float|: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.

---

coef

```
(coef :decimal)
```

Get the coefficient of a decimal value.

---

expn

```
(expn :decimal)
```

Get the exponent of a decimal value. The exponent is implicitly negated, i.e. a return value of 3 means 10^{-3}.

---

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.