JB Scheme Manual

Contents

JB Scheme Manual

Language

JB Scheme is a homebrew, non-RnRS compliant dialect of Scheme.

Primitive Types

string

symbol

integer

bool

Composite Types

pair

list

Special Types

quote

error

Callable Types

lambda

macro

Builtin Callable Types

function

specialform

Binding & 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

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

Create a binding in a new local scope.

```
>>> (let x 12 (display x))
12
```

lets

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

Create multiple bindings in a new local scope.

```
>>> (lets ((x 5) (y 7))
...    (display x)
...    (display y))
5
7
```

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.

```
>>> (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

if

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

Evaluates only `then` or `else` conditonally on the value of `predicate`.

begin

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

Evaluate expressions sequentially and return value of last expression.

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)
```

Pair & 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 termninated by `nil`.

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

nil?

```
(nil? :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.

```
>>> (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`.

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

String Operations

concat

```
(concat :string ...)
```

Concatenate multiple strings.

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

Integer Operations

add (+)

```
(+ :integer ...)
```

mul (*)

```
(* :integer ...)
```

Printing

print

```
(print :string)
```

repr

```
(repr :expr)
```

Get string representation of a value.

display

```
(display :expr)
```

Print string representation of a value.

Type Inspection

type

```
(type :expr)
```

Inspect type of a value.

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

type?

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

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

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

Quoting & Evaluation

quote

```
(quote :expr)
```

A quoted expression evaluates to the expression.

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

eval

```
(eval :expr)
```

Evaluate an expression.

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

apply

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

Apply a procedure to a list of arguments.

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

evalfile

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

Evaluate file in the global environment.

Macro Definition

defmacro

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

jbscheme 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 calling 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, it is looked up in the calling environment:

```
>>> (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 :string)`

raise

`(raise :error)`

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.

```
>>> (defn errored () (begin (raise (error "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.

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|:macro)
```

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

Standard Library

math

unittest