

Lisp Interpreter in Python 3.4

Concepts of Modern Programming Languages

Maria Floruß

August 24, 2014

Overall Structure

Lisp Objects

TODO

Functionality

Builtin Syntax

define

Description: Adds a binding from the first argument to the second to the current environment.

Symbol: `define`

Arguments: `SchemeSymbol`, `SchemeObject`

Return Value: `SchemeVoid`

Example of usage:

```
1 > (define a 1)
2 > a
3 1
4 > (define b "hello")
5 > b
6 "hello"
```

Lambda short hand syntax:

Description: The lambda short hand syntax takes the first element of the first argument and uses it as name. The following elements of the first argument are the arguments of the resulting user defined function. The following arguments define the function body of the user defined function.

Arguments: `SchemeCons`, `SchemeObject`

Return Value: `SchemeVoid`

Example of usage:

```

1 > (define (f n) (+ n 1))
2 > (f 3)
3 4
4 > (define (g n m) (print "hello") (print "world") (+ n m))
5 > (g 1 2)
6 "hello"
7 "world"
8 3

```

lambda

Description: Creates a user defined function. The first argument is a regular list of arguments, the second is a `SchemeCons` defining the body of the function.

Symbol: `lambda`

Arguments: `SchemeCons`, `SchemeCons`

Return Value: `SchemeUserDefinedFunction`

Example of usage:

```

1 > (define f (lambda (n m) (+ n m)))
2 > f
3 <UserDefinedFunction: f>
4 > (f 2 3)
5 5

```

if

Description: Checks if the condition in the first argument is true. If it is true, the second argument is evaluated, otherwise the third one is evaluated.

Symbol: `if`

Arguments: Condition (everything except `SchemeFals` evaluates to `SchemeTrue`), Then-Part, Else-Part.

Return Value: `SchemeObject`

Example of usage:

```

1 > (define a 1)
2 > (define b 2)
3 > (if (> a b) (+ a 1) (+ b 1))
4 3

```

set!

Description: Checks if a binding is found for the first argument, which has to be a symbol. If the binding does not exist a exception is risen. Else the symbol is bound to the new value.

Symbol: `set!`

Arguments: `SchemeSymbol`, `SchemeObject`

Return Value: `SchemeVoid`

Example of usage:

```

1 > (set! a 2)
2 > a
3 NoBindingException: 'No binding found for symbol a.'
4 > (define a 1)
5 > a
6 1
7 > (set! a 2)
8 > a
9 2

```

let

begin

Description: Evaluates one argument after another and returns the return value of the last argument. If no argument is given begin returns `SchemeVoid`.

Symbol: `begin`

Arguments: `0+ SchemeObjects`

Return Value: `SchemeObject`

Example of usage:

```
1 > (begin (print 3) (+ 1 2) (print 4) (+ 2 3))
2 3
3 4
4 5
```

quote

Description: Returns the unevaluated argument.

Symbol: and

Arguments: SchemeObject(

Return Value: SchemeObject

Example of usage:

```
1 > (quote (+ 1 2))
2 (+ 1 2)
3 > (type? (quote (+ 1 2)))
4 "schemeCons"
5 > (quote 1 2 3)
6 ArgumentCountException: 'quote expects exactly 1 argument.'
```

and

Description: Performs a conjunction on all given arguments. Returns `SchemeTrue` if no arguments are given. If one argument is false, all following arguments are not evaluated.

Symbol: and

Arguments: 0+ SchemeObjects

Return Value: SchemeTrue or SchemeFalse

Example of usage:

```

1 > (and (+ 1 2) (> 3 2) (= 2 2))
2 #t
3 > (and (+ 1 2) (< 3 2) (= 1 1))
4 #f
5 > (and (+ 1 2) (= 1 2) (
      thisWouldRaiseAnErrorButDoesNotBecauseItIsNotEvaluated))
6 #t

```

or

Description: Performs a disjunction on all given arguments. Returns `SchemeFalse` if no arguments are given. If one argument is true, all following arguments are not evaluated.

Symbol: `or`

Arguments: `0+ SchemeObjects`

Return Value: `SchemeTrue` or `SchemeFalse`

Example of usage:

```

1 > (or (= 1 2) (> 3 2) (= 2 2))
2 #t
3 > (or (> 1 2) (< 3 2) (= 1 4))
4 #f
5 > (or (= 3 2) (= 1 1) (
      thisWouldRaiseAnErrorButDoesNotBecauseItIsNotEvaluated))
6 #t

```

Builtin Functions

Arithmetic

add

Description: Adds an arbitrary amount of numbers and returns the accumulated value as `SchemeNumber`. If only one argument is given, the arguments value is returned as `SchemeNumber`. If no argument is given the return value is 0.

Symbol: `+`

Arguments: `0+ SchemeNumbers`

Return Value: `SchemeNumber`

Example of usage:

```
1 > (+ 1 2)
2 3
3 > (+ 2 3 4)
4 9
5 > (+)
6 0
7 > (+ 42)
8 42
```

subtract

Description: Subtracts an arbitrary amount of numbers from the first number and returns the accumulated value as `SchemeNumber`. If only one argument is given, the arguments value is negated and returned as `SchemeNumber`. If no argument is given an `ArgumentCountException` is risen.

Symbol: `-`

Arguments: `1+ SchemeNumbers`

Return Value: `SchemeNumber`

Example of usage:

```
1 > (- 0.5 2)
```

```

2  -1.5
3  > (- 10 3 4)
4  3
5  > (-)
6  ArgumentCountException: 'function - expects at least 1 argument.'
7  > (- 42)
8  -42

```

multiply

Description: Multiplies an arbitrary amount of numbers and returns the resulting value as `SchemeNumber`. If only one argument is given, the arguments value is returned as `SchemeNumber`. If no argument is given the return value is 1.

Symbol: `*`

Arguments: `0+ SchemeNumbers`

Return Value: `SchemeNumber`

Example of usage:

```

1  > (* 3.5 4)
2  14.0
3  > (* 2 3 4)
4  24
5  >(*)
6  1
7  > (* 42)
8  42

```


divide

Description: Divides the first argument by the second, the result by the third and so on. If only one argument is given, the result is 1 divided by the argument. If no argument is given an `ArgumentCountException` is risen.

Symbol: `/`

Arguments: `1+ SchemeNumbers`

Return Value: `SchemeNumber`

Example of usage:

```
1 > (- 0.5 2)
2 -1.5
3 > (/ 12 3 2)
4 2.0
5 > (/)
6 ArgumentCountException: 'function / expects at least 1 argument.'
7 > (/ 3)
8 0.3333333333333333
```

arithmetic equals

Description: Checks the two arguments for equal value. Returns `SchemeTrue` if they are equal, otherwise `SchemeFalse`.

Symbol: `=`

Arguments: `exactly 2 SchemeNumbers`

Return Value: `SchemeTrue` or `SchemeFalse`

Example of usage:

```
1 > (= 3 3)
2 #t
3 > (= 1 2)
4 #f
5 > (= 1)
6 ArgumentCountException: 'function = expects exactly 2 arguments.'
```

greater than

Description: Returns `SchemeTrue` if the first argument is greater than the second one, otherwise `SchemeFalse`.

Symbol: `>`

Arguments: exactly 2 `SchemeNumbers`

Return Value: `SchemeTrue` or `SchemeFalse`

Example of usage:

```
1 > (> 3 3)
2 #f
3 > (> 3 2)
4 #t
5 > (> 1)
6 ArgumentCountException: 'function > expects exactly 2 arguments.'
```

less than

Description: Returns `SchemeTrue` if the first argument is less than the second one, otherwise `SchemeFalse`.

Symbol: `<`

Arguments: exactly 2 `SchemeNumbers`

Return Value: `SchemeTrue` or `SchemeFalse`

Example of usage:

```
1 > (< 3 3)
2 #f
3 > (< 1 2)
4 #t
5 > (< 1)
6 ArgumentCountException: 'function < expects exactly 2 arguments.'
```

greater or equal

Description: Returns `SchemeTrue` if the first argument is greater than or equals the second one, otherwise `SchemeFalse`.

Symbol: `>=`

Arguments: exactly 2 `SchemeNumbers`

Return Value: `SchemeTrue` or `SchemeFalse`

Example of usage:

```
1 > (>= 3 3)
2 #t
3 > (>= 3 2)
4 #t
5 > (>= 1 2)
6 #f
7 > (>= 1)
8 ArgumentCountException: 'function >= expects exactly 2 arguments.'
```

less or equal

Description: Returns `SchemeTrue` if the first argument is less than or equals the second one, otherwise `SchemeFalse`.

Symbol: `<=`

Arguments: exactly 2 `SchemeNumbers`

Return Value: `SchemeTrue` or `SchemeFalse`

Example of usage:

```
1 > (<= 3 3)
2 #t
3 > (<= 1 2)
4 #t
5 > (<= 3 2)
6 #f
7 > (<= 1)
```

```
8  ArgumentCountException: 'function <= expects exactly 2 arguments.'
```

absolute value

Description: Returns the absolute value of the given argument.

Symbol: `abs`

Arguments: exactly 1 SchemeNumber

Return Value: SchemeNumber

Example of usage:

```
1  > (abs 3)
2  3
3  > (abs -2)
4  2
5  > (abs 2 3)
6  ArgumentCountException: 'function abs expects exactly 1 argument.'
```

modulo

Description: Does the modulo operation for the two given arguments, i.e. finds the remainder of division of the first argument by the second.

Symbol: `%`

Arguments: exactly 2 SchemeNumbers

Return Value: SchemeNumber

Example of usage:

```
1  > (% 5 3)
2  2
3  > (% 4)
4  ArgumentCountException: 'function % expects exactly 2 arguments.'
5  > (% 3.4 1.1)
6  ArgumentTypeException: '3.4 is no valid operand for procedure %.
    Expects integer.'
```

Other

print

display

cons

car

cdr

first

Description: Returns the first element of the given list. A regular list is expected.

Symbol: `first`

Arguments: `SchemeCons` - has to be a regular list.

Return Value: `SchemeObject`

Example of usage:

```
1 > (first (list 1 2 3))
2 1
3 > (first (list "hello" "world"))
4 "hello"
5 > (first (cons "hello" "world"))
6 ArgumentTypeException: 'rest expects a not empty list as argument.'
```

rest

Description: Returns the rest list after the first argument of the given list. A regular list is expected.

Symbol: `rest`

Arguments: `SchemeCons` - has to be a regular list.

Return Value: `SchemeObject`

Example of usage:

```
1 > (rest (list 1 2 3))
2 (2 3)
```

```

3 > (rest (cons 1 2))
4 ArgumentTypeException: 'rest expects a not empty list as argument.'
5 > (rest (list "hello" "world"))
6 ("world")

```

list

time

recursion-limit

Description: If no argument is given the current recursion limit is returned. Per default this is 1000. If a `SchemeNumber` is given, the recursion limit is set to this number.

Symbol: `recursion-limit`

Arguments: nothing or `SchemeNumber`

Return Value: `SchemeVoid` or `SchemeNumber`

Example of usage:

```

1 > (recursion-limit)
2 1000
3 > (recursion-limit 2000)
4 > (recursion-limit)
5 2000

```

type?

Description: Evaluates the given `SchemeObject` and returns the type of the return value as `SchemeString`.

Symbol: `type?`

Arguments: `SchemeObject`

Return Value: `SchemeString`

Example of usage:

```
1 > (type? 1)
2 "schemeNumber"
3 > (type? (define a 1))
4 "schemeVoid"
5 > (type? (quote (+ 1 2)))
6 "schemeCons"
```

not

map

get-function-info