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:

```
> (define a 1)
> a
1
> (define b "hello")
> b
"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:

```
> (define (f n) (+ n 1))
> (f 3)
4
> (define (g n m) (print "hello") (print "world") (+ n m))
> (g 1 2)
"hello"
"world"
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:

```
> (define f (lambda (n m) (+ n m)))
> f
<UserDefinedFunction: f>
> (f 2 3)
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:

```
> (define a 1)
> (define b 2)
> (if (> a b) (+ a 1) (+ b 1))
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 exeption is risen. Else the symbol is bound to the new value.
Symbol:	`set!`
Arguments:	`SchemeSymbol`, `SchemeObject`
Return Value:	`SchemeVoid`

Example of usage:

```
> (set! a 2)
> a
NoBindingException: 'No binding found for symbol a.'
> (define a 1)
> a
1
> (set! a 2)
> a
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:

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

quote

Description:	Returns the unevaluated argument.
Symbol:	`and`
Arguments:	`SchemeObject`(
Return Value:	`SchemeObject`

Example of usage:

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

and

Description:	Performs a conjunction on all given arguments. Returns `SchemeTrue` if no arguments are given. If one arg)ument is false, all following arguments are not evaluated.
Symbol:	`and`
Arguments:	0+ `SchemeObjects`
Return Value:	`SchemeTrue` or `SchemeFalse`

Example of usage:

```
> (and (+ 1 2) (> 3 2) (= 2 2))
#t
> (and (+ 1 2) (< 3 2) (= 1 1))
#f
> (and (+ 1 2) (= 1 2) (
   thisWouldRaiseAnErrorButDoesNotBecauseItIsNotEvaluated))
#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:

```
> (or (= 1 2) (> 3 2) (= 2 2))
#t
> (or (> 1 2) (< 3 2) (= 1 4))
#f
> (or (= 3 2) (= 1 1) (
   thisWouldRaiseAnErrorButDoesNotBecauseItIsNotEvaluated))
#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+ `SchemeNumber`s
Return Value:	`SchemeNumber`

Example of usage:

```
> (+ 1 2)
3
> (+ 2 3 4)
9
> (+)
0
> (+ 42)
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+ `SchemeNumber`s
Return Value:	`SchemeNumber`

Example of usage:

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

```
-1.5
> (- 10 3 4)
3
> (-)
ArgumentCountException: 'function - expects at least 1 argument.'
> (- 42)
-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:

```
> (* 3.5 4)
14.0
> (* 2 3 4)
24
>(*)
1
> (* 42)
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 devided by the argument. If no argument is given an ArgumentCountException is risen.
Symbol:	`/`
Arguments:	1+ `SchemeNumber`s
Return Value:	`SchemeNumber`

Example of usage:

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

arithmetic equals

Description:	Checks the two arguments for equal value. Returns `SchemeTrue` if they are equal, otherwise `SchemeFalse`.
Symbol:	`=`
Arguments:	exactly 2 `SchemeNumber`s
Return Value:	`SchemeTrue` or `SchemeFalse`

Example of usage:

```
> (= 3 3)
#t
> (= 1 2)
#f
> (= 1)
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 `SchemeNumber`s
Return Value:	`SchemeTrue` or `SchemeFalse`

Example of usage:

```
> (> 3 3)
#f
> (> 3 2)
#t
> (> 1)
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 `SchemeNumber`s
Return Value:	`SchemeTrue` or `SchemeFalse`

Example of usage:

```
> (< 3 3)
#f
> (< 1 2)
#t
> (< 1)
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:

```
> (>= 3 3)
#t
> (>= 3 2)
#t
> (>= 1 2)
#f
> (>= 1)
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:

```
> (<= 3 3)
#t
> (<= 1 2)
#t
> (<= 3 2)
#f
> (<= 1)
```

```
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:

```
> (abs 3)
3
> (abs -2)
2
> (abs 2 3)
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 `SchemeNumber`s
Return Value:	`SchemeNumber`

Example of usage:

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

Other

exit

Description:	Closes the interpreter. Any number of arguments can be given. If the first argument is a SchemeNumber the interpreter will close with the according exit code.
Symbol:	`cons`
Arguments:	exactly two `SchemeObjects`
Return Value:	`SchemeCons`

Example of usage:

```
> (exit)
user@computer:~/Studies

> (exit 12)
user@computer:~/Studies ?12
```

print

display

cons

Description:	Creates a `SchemeCons` with the first argument as car and the second argument as cdr.
Symbol:	`cons`
Arguments:	exactly two `SchemeObjects`
Return Value:	`SchemeCons`

Example of usage:

```
> (cons 1 2)
(1 . 2)
> (cons 1 2 3)
ArgumentCountException: 'cons expects exactly 2 arguments.'
> (cons (1 (cons 2 3)))
```

```
(1 2 . 3)
> (cons 1 (cons 2 nil))
(1 2)
```

car

Description:	Returns the car of the given `SchemeCons`.
Symbol:	`car`
Arguments:	`SchemeCons`
Return Value:	`SchemeObject`

Example of usage:

```
> (car (cons 1 2))
1
> (car (list "hello" 2 3))
"hello"
> (car 1)
ArgumentTypeException: 'car expects cons as argument'
```

cdr

Description:	Returns the cdr of the given `SchemeCons`.
Symbol:	`cdr`
Arguments:	`SchemeCons`
Return Value:	`SchemeObject`

Example of usage:

```
> (cdr (cons 1 2))
2
> (car (list "hello" 2 3))
(2 3)
> (cdr 1)
ArgumentTypeException: 'cdr expects cons as argument'
```

list

Description:	Creates a regular list out of all arguments.
Symbol:	`list`
Arguments:	0+ `SchemeObjects`
Return Value:	`SchemeCons` or `SchemeNil`

Example of usage:

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

list?

Description:	Returns `SchemeTrue` if the argument is a regular list, else `SchemeFalse`.
Symbol:	`list?`
Arguments:	`SchemeObject`
Return Value:	`SchemeTrue` or `SchemeFalse`

Example of usage:

```
> (list? (list 1 2 3))
#t
> (list? (cons 1 2))
#f
> (list? 1)
#f
> (list? nil)
#t
```

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:

```
> (first (list 1 2 3))
1
> (first (list "hello" "world"))
"hello"
> (first (cons "hello" "world"))
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:

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

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:

```
> (recursion-limit)
1000
> (recursion-limit 2000)
> (recursion-limit)
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:

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

not

Description:	Returns `SchemeTrue` for `SchemeFalse`, `SchemeFalse` for everything else.
Symbol:	`not`
Arguments:	`SchemeObject`
Return Value:	`SchemeTrue` or `SchemeFalse`

Example of usage:

```
> (not #t)
#f
> (not #f)
#t
> (not 1)
#f
> (not (list 1 2 3))
#f
> (not true)
#f
> (not false)
#t
```

map

get-function-info