

JDice Language Reference Manual

Robert Bossy

2008-09-03

Contents

1	Introduction	1
1.1	What is and Why JDice?	1
1.2	Notations	2
1.3	Data types	3
1.3.1	JDice lists	3
1.3.2	Conversions	4
1.4	Namespaces	4
2	Grammar specification	4
2.1	BNF	4
2.2	Token precedence	7
2.3	Dice token	7
3	Dice expressions	8
3.1	Syntax	8
3.1.1	Comments	8
3.1.2	Primitives	8
3.1.3	Expression sequence	8
3.1.4	Lists	8
3.2	Semantics	8
3.2.1	Truth values	8
3.2.2	Automatic lambda	9
3.2.3	Loop conditions	9
3.2.4	Dice types	9
3.3	Expression reference	10
3.3.1	And	10
3.3.2	Arithmetic	10
3.3.3	Assign	11
3.3.4	Attribute	11
3.3.5	Call	12
3.3.6	Comparison	12
3.3.7	Concat	13
3.3.8	Conditional	13

3.3.9	Dice	14
3.3.10	Foreach	14
3.3.11	Globals	15
3.3.12	IntConstant	15
3.3.13	Lambda	16
3.3.14	Length	16
3.3.15	ListConstructor	16
3.3.16	Locals	17
3.3.17	Nil	17
3.3.18	Not	17
3.3.19	Or	18
3.3.20	Range	18
3.3.21	Reroll	18
3.3.22	Revert	19
3.3.23	Select	19
3.3.24	SelectSingle	20
3.3.25	Self	20
3.3.26	SingleDie	21
3.3.27	Sort	21
3.3.28	StringConstant	21
3.3.29	Subscript	22
3.3.30	Sum	22
3.3.31	Variable	22

4	Modules	23
----------	----------------	-----------

	About this document	23
--	----------------------------	-----------

1 Introduction

1.1 What is and Why JDice?

JDice is a specific domain language designed to express and simulate dice rolls.

A lot board games, wargames and role-playing games include dice rolls in their rules. The number and the type of dice rolled can be different from one game to another. Moreover the different rules specify various things to do with the rolls (add, take highest, reroll, etc).

JDice is a language that allows to express these rules. It is also an implementation of this language written in Java that can parse this language and simulate the rolls in two modes:

1. roll and display the result;
2. roll a lot of times and display the distribution of the results obtained.

We hope JDice will be useful for:

- players and game masters who do not care rolling actual dice and are fed up with repetitive rolls;
- dice roll server developpers who want to provide something more elaborated than 2d6 or d20;
- hardcore gamist players who want to optimize their characters or their army;
- game designers who want to put up balanced, fun and original resolution systems.

The JDice language was designed as a *functional* language: this means that one writes expressions (called *dice expressions*) evaluated by the JDice implementation. The evaluation of dice expressions produces *values*. An expression may contain subexpressions, thus the evaluation of the containing expression depends on the value produced by the evaluation of the contained expression: in this way expressions are evaluated from the innermost to the outermost. We did not see yet the JDice syntax but it is enough to say that `2d6 + 3` is a valid JDice expression. In order to evaluate it, the subexpression `2d6` must be evaluated, then 3 is added to the roll.

What JDice is not

- Strict Functional Language: in JDice, functions are first-class values, the functional part of JDice stops here since it does not allow curriification, polymorphic types, tail recursion optimization; moreover JDice has side-effects and assignation as expressions and many more things that would give rabies to functional language blowhards.
- Do-Everything Scripting Language: JDice lacks a lot of built-ins necessary to be an all-purpose language (string operations, floating points, etc.); if you want to manipulate strings, then use Perl; if you want to make a prototype application, then use Python; if you want to develop megawebsites, then use Java(Script).
- Extremely efficient: JDice is efficient enough to roll complicated dice schemes a couple of thousand times in a second; if you want to simulate millions of rolls and don't want to wait a minute, feel free to either contribute to the JDice implementation, or implement the language in C.

JDice language idiosynchrasies Experienced programmers who know several langages will notice that the JDice language has some oddities. Let's warn about them right now:

- the letter `d` is parsed in a special way, it can be either part of an identifier or an operator;
- identifiers cannot contain digits;

- there is a single structured data type described below used to implement lists, mappings and namespaces;
- there is no *while* loop.

1.2 Notations

In this document the reader will recognize: **reserved words & symbols**, *operands and variable parts of expressions* and full example expressions.

In the expression reference part, the evaluation result of an expression *expr*, will be noted *eval(expr)*. The evaluation of *expr* converted into an integer or list will be noted *int(expr)* and *list(expr)* respectively. The evaluation of *expr* as a function will be noted *fun(expr)*.

1.3 Data types

The values produced by the evaluation of a JDice expression may be one of the following five types:

- *nil*: this is a type with a single value with the same name usually meaning there is no appropriate value, or if an error has occurred somewhere.
- integer: 1, 2, 3, 4, etc, but also 0 (zero) and -1, -2, -3, etc.
- string: arbitrary sequence of characters.
- list: lists are ordered sequences of values, more on this data structure later.
- function: these are stored expressions that can be evaluated in several different context with several different sets of parameters.

These are the only possible data types in JDice, this limited set keeps it easy but is rich enough to express a wide range of things.

1.3.1 JDice lists

JDice lists (or simply *lists*) are very important in JDice since they are the only way to express structured data. A list, like lists or arrays in other languages, is an ordered sequence of values. Each value may be of one of the five JDice types, so a list can include other lists or even functions.

A value in a list can be accessed by its index. In JDice indexes start at 0 (zero).

Moreover a value in a list may have a name (which is a string), a named element can be accessed either by its index, either by its name. A name is unique within a list: there can be only one value with a given name in a list.

List merging This document will sometimes refer to an operation on lists called *merging*. The merging of a list L into a list M is performed as follows:

```
for each  $L_i$  element of  $L$ :
  if  $L_i$  has name ( $n$ )
  then
    if  $M$  has element named  $n$  ( $M_n$ )
    then
       $M_n \leftarrow L_i$ 
    else
      append  $L_i$  with name  $n$  to  $M$ 
  else
    append  $L_i$  to  $M$ 
```

1.3.2 Conversions

Some expressions expect a particular type of value for the result of the evaluation of an operand. If the sub-expression produces a value from a different type, JDice will try to convert the value to the required type. The conversion rules are:

Integer to list The integer i is converted into a list with a single element i (not named).

String to list The string s is converted into a list with a single element s (not named).

Integer to string The integer i is converted into a string with the digit characters representing i in decimal notation.

String to integer The string s is converted into an integer by assuming that the characters are digits representing an integer in decimal notation.

List to integer Each element of the list is converted into integers then summed.

Nil to integer nil is converted to 0 (zero).

Nil to string nil is converted to an empty string.

Nil to list nil is converted to an empty list (no elements).

Other conversions Any other conversion is considered impossible and cause an error to the JDice implementation. Nothing can be converted into nil, and functions cannot be converted from and to any other type. Also there is no meaningful way in JDice to convert a list into a string.

1.4 Namespaces

2 Grammar specification

2.1 BNF

```
expression_sequence ::=
  strict_expression_sequence
| strict_expression_sequence ';'
;

strict_expression_sequence ::=
  expression
| strict_expression_sequence ';' expression
;

expression ::=
  '(' expression_sequence ')'
| INT
| 'fun' '(' list ')' '{' expression_sequence '}'
| '[' list ']'
| STRING
| 'nil'
| IDENTIFIER
| expression '=' expression
| expression '.' IDENTIFIER
| expression '.' INT
| expression '[' expression:idx ']'
| expression '(' list ')'
| 'if' expression 'then' expression
| 'if' expression 'then' expression 'else' expression_
| expression 'foreach' expression
| expression 'foreach' expression 'if' loop_condition
| expression 'foreach' expression 'while' loop_condition
| 'foreach' expression 'if' loop_condition
| 'foreach' expression 'while' loop_condition
| 'reroll' expression 'if' loop_condition
| 'reroll' expression 'while' loop_condition
| expression 'and' expression
| expression 'or' expression
| 'not' expression
| expression comparison_operator expression
| expression arithmetic_operator expression
| D expression
| D
| expression D expression
```

```

| expression D
| expression '..' expression
| expression '+=' expression
| 'length' expression
| 'revert' expression
| 'sort' expression
| 'sum' expression
| select_operator 'of' expression
| select_operator expression 'of' expression
| '@'
| '@' INT
| '@' IDENTIFIER
| '$'
| '$' INT
| '$' IDENTIFIER
| '~'
| '~' INT
| '~' IDENTIFIER
| 'import' module_name
| 'import' module_name '[' '*' ']',
| 'import' module_name '[' name_list ']'
;

```

```

comparison_operator ::=
    '=='
| '!='
| '<'
| '>'
| '<='
| '>='
;

```

```

select_operator ::=
    'first'
| 'last'
| 'highest'
| 'lowest'
;

```

```

arithmetic_operator ::=
    '+'
| '-'
| '*'
| '/'
| '%'
;

```

```

list ::=
    <EMPTY>
  | non_empty_list
  ;

non_empty_list ::=
    list_element
  | non_empty_list ',' list_element
  ;

list_element ::=
    expression
  | IDENTIFIER ':' expression
  ;

loop_condition ::=
    expression
  | comparison_operator expression
  ;

module_name ::=
    IDENTIFIER
  | module_name '.' IDENTIFIER
  ;

name_list ::=
    IDENTIFIER
  | name_list ',' IDENTIFIER
  ;

```

2.2 Token precedence

All operators in JDice are left-associative, except for the assignment operator (=) which is right-associative. The following table gives the tokens by order of precedence.

```

=
foreach
if then while
else
+ = length revert sort sum
..
and or
== != < > <= >=
+ -
/ %
d
( [ { .

```

2.3 Dice token

The letter `d` is parsed as the dice operator in the following conditions:

Preceded by	Followed by	Parsed as
non-alpha	lower or <code>_</code>	part of an identifier
non-alpha	upper	dice followed by an identifier
non-alpha	non-alpha	dice
lower or <code>_</code>	non-alpha	part of an identifier
single upper	non-alpha	dice preceded by a single letter identifier
single upper	upper	dice preceded by a single letter identifier and followed by an identifier

This set of rules allows to use the letter `d` in classical role-playing notations: `2d6`, `d20` or `NdX`. It still allows the letter `d` in identifiers in the most common cases.

3 Dice expressions

3.1 Syntax

This section describes the syntax of JDice.

3.1.1 Comments

From character `#` to the end of the line.

3.1.2 Primitives

Nil: the `nil` keyword.

Integer litteral: sequence of digits.

String litteral: a sequence of arbitrary characters enclosed between double quotes (`"`).

Identifier: a sequence of alphabetic or underscore characters, or a sequence of arbitrary characters enclosed between simple quotes (').

3.1.3 Expression sequence

An expression sequence is a sequence of expressions separated by semicolons (;). The last expression of an expression sequence can be followed by a final semicolon.

3.1.4 Lists

An expression list is a sequence of expressions separated by commas (.). Each expression may be preceded by an identifier followed by a colon (:). When an expression in a list is preceded by an identifier, then this element is said to be named; its name is the identifier string.

3.2 Semantics

3.2.1 Truth values

Some expression constructs (`if`, `foreach`, `reroll`, etc.) affect the control flow according to a condition. The truth of a value depends on its type:

- `nil` is always false;
- an integer is true if it is different from zero, and false if it is equal to zero;
- a string is false if it is the empty string, otherwise it is true;
- a list is false if it is empty, it is true if it contains at least one element, even if all elements are false;
- a function is always true.

3.2.2 Automatic lambda

When JDice needs to evaluate an expression as a function, its behaviour depends on the type of the expression:

- if the expression is a lambda expression, then it will be evaluated normally;
- if the expression necessarily evaluates to a non-function value, then it will be evaluated as a function with no defaults and the expression as the body;
- if the expression could evaluate to any type, including function, then it will be evaluated normally; if the result is a function, then it will be used as is, otherwise a function will be built as described above.

3.2.3 Loop conditions

Loop conditions benefit from several syntactic sugars.

If the condition consists on a comparison operator followed by an expression, that is a second half of a comparison, then the last value considered in the loop will be compared as if it was placed on the left-hand side of the comparison operator. The last element of the loop is the current list element in a `foreach` loop, or the result of the last evaluation in a `reroll` loop. For instance `reroll d if > 3` is strictly equivalent to `reroll d if $0 > 3`.

If the condition is an integer constant, then JDice will test the equality between this constant and the last value considered in the loop. In this way `reroll d if 1` is strictly equivalent to `reroll d if $0 == 1`.

If the condition is a negation, and the negated expression is an integer constant, then then JDice will test the non-equality between this constant and the last value considered in the loop. In this way `reroll d while not 1` is strictly equivalent to `reroll d while $0 != 1`.

3.2.4 Dice types

The expression on the right-hand side of a dice (`d`) operator is the type of the rolled dice. JDice behaviour will depend on the type of the dice type:

- if the dice type is an integer, then JDice will produce a random integer between 1 and the dice type inclusive;
- if the dice type is a list, then JDice will pick one member randomly, the list is left untouched (random pick with replacement);
- if the dice type is a function, there are two different cases:
 - if the function has a default named `N`, then the function will be called with the number of dice assigned to `n` and the result is returned as is;
 - otherwise the function is called as many times as the required number of dice and the list of the successive results is returned.

3.3 Expression reference

3.3.1 And

Synopsis	Logical 'and'
Syntax	<i>left and right</i>
Operands	<i>left</i> expression evaluated <i>right</i> expression evaluated
Type	undefined

Description If $eval(left)$ is false then this value is returned. Otherwise returns $eval(right)$.

$right$ is evaluated iff $eval(left)$ is true.

3.3.2 Arithmetic

Synopsis Integer arithmetic operations

Syntax $left + right$
 $left - right$
 $left * right$
 $left / right$
 $left \% right$

Operands $left$ expression evaluated as an integer
 $right$ expression evaluated as an integer

Type integer

Description Performs the integer arithmetic operation on $int(left)$ and $int(right)$.
 $/$ denotes integer division and $\%$ division remainder.

3.3.3 Assign

Synopsis

Syntax Value assignation $lvalue = rvalue$

Operands $lvalue$ expression evaluated
 $rvalue$ expression evaluated

Type undefined

Description $eval(rvalue)$ is assigned to $lvalue$. Depending on the expression type of $lvalue$, the assignation has different effects:

Variable Binds the value to the variable name in the current namespace.

Attribute Sets the value to the corresponding list element.

Subscript Sets the value to the corresponding list element.

ListConstructor Converts $eval(rvalue)$ to a list and merges it to $lvalue$.

$eval(rvalue)$ is returned.

3.3.4 Attribute

Synopsis Get an element from a list by index or name

Syntax *list* :: *idx*
list :: *name*

Operands *list* expression evaluated as a list
idx integer constant
name name

Type undefined

Description Returns the *idx* th element of *list(list)*, or the element named *name* in list. If the element is a function, then the containing list is set to *list(list)* (see Self).

3.3.5 Call

Synopsis Function call

Syntax *fun* (*args*)

Operands *fun* expression evaluated as a function
args named list

Type undefined

Description *args* is merged to the *fun(fun)* defaults. The merged list is used as the current namespace when evaluating *fun(fun)* body. The result of the body evaluation is returned.

3.3.6 Comparison

Synopsis	Compare two values
Syntax	<i>left</i> == <i>right</i> <i>left</i> != <i>right</i> <i>left</i> < <i>right</i> <i>left</i> > <i>right</i> <i>left</i> <= <i>right</i> <i>left</i> >= <i>right</i>
Operands	<i>left</i> expression evaluated <i>right</i> expression evaluated
Type	integer

Description Applies the comparison operator to *eval(left)* and *eval(right)* and returns an integer indicating if the comparison is true.

Order and equality of dice values is defined as follows:

- Two values of different types are different.
- The order of types is: function > list > string > int > nil.
- nil equals to nil .
- An integer compares to another like integers always do.
- A string compares to another by asciibetical order.
- A list compares to another by comparing the pairs of elements with the same index. The pair of non equal elements with the lowest index determines the order between the two lists. If all the pairs are equal, then the shortest list is considered lower. If all the pairs are equal and both lists are of the same size, then they are equal. Note that element names are never considered in the list comparison.
- There is no particular definition of the comparison between two functions. Implementations are required to provide a consistent comparison method.

3.3.7 Concat

Synopsis	Concatenates two lists
Syntax	<i>target += source</i>
Operands	<i>target</i> expression evaluated as a list <i>source</i> expression evaluated as a list
Type	list

Description Each element of *list(source)* is considered in order. If the element is not named, then it is appended at the end of *list(target)*. If the element is named, then it is set in *list(target)* with the same name. *list(target)* is returned after the concatenation.

3.3.8 Conditional

Synopsis	Evaluates expressions conditionally
Syntax	<i>if condition then true else false</i> <i>if condition then true</i>
Operands	<i>condition</i> expression evaluated <i>true</i> expression evaluated <i>false</i> expression evaluated
Type	undefined

Description If *eval(condition)* is true, returns *eval(true)* (*false* is not evaluated), otherwise returns *eval(false)* (*true* is not evaluated). In the form without else, if *eval(condition)* is not true, then `nil` is returned.

3.3.9 Dice

Synopsis	Roll dice
Syntax	<i>n d</i> <i>n d dice</i>
Operands	<i>n</i> expression evaluated as an integer <i>dice</i> expression evaluated as a dice type
Type	list

Description Rolls $int(n)$ dice of type `dice`. This expression returns a list with $int(n)$ unnamed elements. If the dice type is omitted, then the default dice type of the current namespace is used. If dice is an integer, then each element of the result is a random integer between 1 and $int(dice)$, inclusive. If dice is a function with a default named "n", then $fun(dice)$ is called with an argument named "n" set to $int(n)$. If dice is a function without a default named "n", then $fun(dice)$ is called $int(n)$ times. If dice is a list, then $int(n)$ elements from $list(dice)$ are randomly chosen to form the result list.

3.3.10 Foreach

Synopsis Iterates over the elements of a list

Syntax `fun foreach list`
`fun foreach list if condition`
`fun foreach list while condition`
`foreach list if condition`
`foreach list while condition`

Operands `fun` expression evaluated as a function
`list` expression evaluated as a list
`condition` loop condition

Type list

Description In the first form, calls $fun(fun)$ as many times there are elements in $list(list)$. There are four arguments passed to $fun(fun)$ in this order:

- the element value;
- the index of the element;
- the name of the element, or `nil` if the element is not named;
- the list being build.

This expression returns a list containing the results of consecutive calls to $fun(fun)$. Elements have the same name than the corresponding input element from $list(list)$. The result list is passed to $fun(fun)$ as the fourth argument, modification of this list is possible, though quite risky.

In the `if` form, $fun(condition)$ is called for each element with the same arguments than $fun(fun)$. If the result is true then $fun(fun)$ is called and its result appended to the list, otherwise $fun(fun)$ is not called and the returned list is not grown.

In the `while` form, if the result of the call of $fun(condition)$ is false, then the iteration is stopped and the result list is returned as is.

In the forms where fun is omitted, then the result list is filled with $list(list)$ values unchanged.

3.3.11 Globals

Synopsis Access to the current module namespace

Syntax @

Operands
Type list

Description Returns the variable bindings in the module global namespace in the form of a list. Modifications on this list will affect the namespace.

3.3.12 IntConstant

Synopsis Integer constant

Syntax *i*

Operands *i* integer constant

Type integer

Description Returns the integer denoted by *i*.

3.3.13 Lambda

Synopsis Build a function

Syntax `fun (defaults) body`

Operands *defaults* named list
body expression evaluated

Type function

Description Returns a function with *defaults* as defaults, and *body* as body. Each *defaults* is evaluated when the `lambda` expression is evaluated, they are not reevaluated when the function is called.

3.3.14 Length

Synopsis	List size
Syntax	<code>length list</code>
Operands	<i>list</i> expression evaluated as a list
Type	integer

Description Returns the number of elements in *list(list)*.

3.3.15 ListConstructor

Synopsis	Build a list by specifying each element
Syntax	<code>[elements]</code>
Operands	<i>elements</i> named list
Type	list

Description Evaluates each element of *elements* and returns a list containing the result values.

3.3.16 Locals

Synopsis	Current namespace access
Syntax	<code>\$</code>
Operands	
Type	list

Description Returns the variable bindings in the current namespace in the form of a list. Modifications on this list will affect the namespace. In a function body `$` will return the list of arguments.

3.3.17 Nil

Synopsis Nil value

Syntax nil

Operands
Type nil

Description Returns the nil value.

3.3.18 Not

Synopsis Logical negation

Syntax not *negated*

Operands *negated* expression evaluated

Type integer

Description If *eval(negated)* is true, returns zero, otherwise returns 1.

3.3.19 Or

Synopsis Logical or

Syntax *left* or *right*

Operands *left* expression evaluated
right expression evaluated

Type undefined

Description If *eval(left)* is true then this value is returned. Otherwise returns *eval(right)*.

right is evaluated iff *eval(left)* is false.

3.3.20 Range

Synopsis	Integers between two boundaries
Syntax	<i>from</i> .. <i>to</i>
Operands	<i>from</i> expression evaluated as an integer <i>to</i> expression evaluated as an integer
Type	list

Description Returns a list including all integers between *int(from)* (inclusive) and *int(to)* (exclusive). If *int(from)* is lower than *int(to)*, then the integers are in ascending order. If *int(from)* is greater than *int(to)*, then the integers are in descending order. If *int(from)* and *int(to)* are equal, then an empty list is returned.

3.3.21 Reroll

Synopsis	Recalls conditionally a function
Syntax	<code>reroll fun if condition</code> <code>reroll fun while condition</code>
Operands	<i>fun</i> expression evaluated as a function <i>condition</i> loop condition
Type	list

Description In the `if` form, *fun(fun)* and *fun(condition)* are called successively, if the result of the call of *fun(condition)* is true the re-calls *fun(fun)*. A list containing the first and possibly the second results of call(s) to *fun(fun)* is returned.

Both functions are called with four arguments:

- the result of the last call to *fun(fun)*;
- the number of times *fun(fun)* has been called before;
- `nil` (this argument has been kept to keep the same form as `foreach` loops);
- the list being build.

In the `while` form, *fun(condition)* is always called after *fun(fun)* (not just the first time) and *fun(fun)* is always re-called if the result of the call of *fun(condition)* is true.

3.3.22 Revert

Synopsis	List inversion
Syntax	<code>revert list</code>
Operands	<i>list</i> expression evaluated as a list
Type	list

Description Reverts the elements of *list(list)* in place and returns *list(list)*. Element names are kept.

3.3.23 Select

Synopsis	Pick elements from a list
Syntax	<code>first n of list</code> <code>last n of list</code> <code>lowest n of list</code> <code>highest n of list</code>
Operands	<i>n</i> expression evaluated as an integer <i>list</i> expression evaluated as a list
Type	list

Description Returns a list containing *int(n)* elements taken from *list(list)* according to the first keyword:

first the first elements;

last the last elements;

lowest the lowest elements in comparison order (see Comparison);

highest the highest elements in comparison order.

list(list) is left untouched.

3.3.24 SelectSingle

Synopsis	Pick a single element from a list
Syntax	<code>first</code> of <i>list</i> <code>last</code> of <i>list</i> <code>lowest</code> of <i>list</i> <code>highest</code> of <i>list</i>
Operands	<i>list</i> expression evaluated as a list
Type	undefined

Description Returns a the element taken from *list(list)* according to the first keyword:

`first` the first element;

`last` the last element;

`lowest` the lowest element in comparison order (see Comparison);

`highest` the highest element in comparison order.

list(list) is left untouched.

3.3.25 Self

Synopsis	Containing list access
Syntax	<code>~</code>
Operands	
Type	list

Description Returns the list containing the function currently called. If the evaluation is not in a function body or if this function was not obtained from the evaluation of an Attribute expression, then `~` acts like `$$`

3.3.26 SingleDie

Synopsis	Single die roll
Syntax	<code>d</code> <code>d <i>die</i></code>
Operands	<i>die</i> expression evaluated as a dice type
Type	undefined

Description Rolls a die of type *die*. If the dice type is omitted, then the default dice type of the current namespace is used. If *die* is an integer, then returns a random integer between 1 and $\text{int}(\text{dice})$, inclusive. If *die* is a function, then $\text{fun}(\text{die})$ is called with an argument named "n" set to 0. If *die* is a list, then a random element from this list is returned.

3.3.27 Sort

Synopsis	List reordering
Syntax	<code>sort <i>list</i></code>
Operands	<i>list</i> expression evaluated as a list
Type	list

Description Sorts $\text{list}(\text{list})$ in place according to the comparison order (see Comparison). $\text{list}(\text{list})$ is returned after sorting.

3.3.28 StringConstant

Synopsis	String constant
Syntax	<code><i>s</i></code>
Operands	<i>s</i> string constant
Type	string

Description Returns a string denoted by *s* .

3.3.29 Subscript

Synopsis	List element
Syntax	<i>list</i> [<i>subscript</i>]
Operands	<i>list</i> expression evaluated as a list <i>subscript</i> expression evaluated as a string or an integer
Type	integer

Description Returns the element of *list(list)* designated by *eval(subscript)*. If *eval(subscript)* is a string, then the element named *subscript* is returned, otherwise the element at index *int(subscript)* is returned.

3.3.30 Sum

Synopsis	List tally
Syntax	sum <i>list</i>
Operands	<i>list</i> expression evaluated as a list
Type	integer

Description Converts each element of *list(list)* to integers and returns the sum.

3.3.31 Variable

Synopsis	Variable name
Syntax	<i>var</i>
Operands	<i>var</i> name
Type	undefined

Description Returns the last value assigned to the variable named *var* in the current namespace. If *var* was not defined in the current namespace, then it is looked for in the containing namespaces (from innermost to outermost). If *var* was not defined in any namespace, then `nil` is returned.

4 Modules

About this document