

The Premise Language

version 3.0

by

Michael S P Miller

The Premise Language

The Premise Language

version 3.0

by

Michael S. P. Miller

Copyright © 2013 – 2025. Michael S. P. Miller.

All Rights Reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval systems) without permission in writing from the author—except in the case of a reviewer who may quote brief passages embodied in critical articles or in a review.

Trademarked names may appear throughout this book. Rather than use a trademark symbol with every occurrence of a trademarked name, names are used in an editorial fashion, with no intention of infringement of the respective owner's trademark or copyright. The same applies to quotations or synopses of copyrighted material.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused directly or indirectly by the information contained in this book. The source code examples and specifications in this book are for illustrative purposes only, and are therefore provided as is, without warranty of any kind, neither expressed nor implied that the examples work or are fit for a particular purpose. The author and publisher assume no responsibility or liability for damages or losses, neither incidental nor consequential, incurred as a result of using the provided specifications or source code.

The Premise Language™, The Premise Programming Language™ and The SubThought Logo™, the [P] logo™, are Copyright © 2013 – 2025 SubThought Corporation. All Rights Reserved.

For comments on the SubThought Premise Language contact: subthought@hotmail.com

For electronic inquiries or permissions contact: subthought@hotmail.com.

by mail: SubThought Corporation, 254 N. Lake Ave. #213, Pasadena, CA 91101 USA

This book was published in the United States of America.

Paperback ISBN-13: 979-8-849321-64-6

Hardcover ISBN-13: 979-8-849796-21-5

Paperback ISBN-10: 8-849321-64-3

Hardcover ISBN-10: 8-849796-21-8

Cover by Michael S. P. Miller

Document version: 3.120

For those who seek truth.

Contents

| | |
|-----------------------------|-----|
| Contents | 11 |
| Figures | 12 |
| Tables..... | 12 |
| Acknowledgements | 13 |
| Disclaimer | 13 |
| Introduction..... | 15 |
| 1. Getting Started | 19 |
| 2. The Interpreter..... | 21 |
| 3. Taxonomy | 25 |
| 4. Control Flow | 63 |
| 5. Code Examples | 83 |
| 6. Foundation Modules | 125 |
| 7. Quick Reference | 126 |
| 8. Function Reference | 136 |
| Bibliography..... | 689 |
| Index | 691 |

Figures

| | |
|--|----|
| Figure 1. The Premise Interpreter Architecture | 21 |
| Figure 2. Premise Taxons..... | 25 |

Tables

| | |
|--|-----|
| Table 3. Function Quick Reference..... | 135 |
|--|-----|

Acknowledgements

“Do you see over yonder, friend Sancho, thirty or forty hulking giants? I intend to do battle with them and slay them.”

— Miguel de Cervantes Saavedra, Don Quixote

This work was not possible without the efforts and advice of Dr. Sheldon Linker, whose collaboration on terminology and functionality was most welcome, and combined with his proficient coding skills led to the 2013 Java implementation of a just in time interpreter: The Premise 0.1 Executive. Suzuki Hisao's lucid and concise C# implementation of Nukata Lisp Lite was insightful, as was Peter Norvig's implementation of JScheme. Allen Holub's book Compiler Design in C was also instructive, particularly in the area regarding approaches to tokenization and parsing. Christian Queinnec's tome, Lisp In Small Pieces, also provided invaluable insights on expression evaluation. Christian's encouragement on this project was greatly appreciated. Roland Hausser's left associative grammar was pivotal in the tokenizer design.

The author would also like to thank Aaron Hosford for his suggestions on numeric similarity metrics, Ryan Hewitt for his comments on prototype construction, Brian Will for his ideas on restricted scoping, Dr. Ghassan Azar, Todd Kaufmann, Dr. Michael Schuresko, Robert Rossi, Jean-Louis Villecroze and the rest of the Premise Language group for their constructive feedback and suggestions.

Disclaimer

The source code examples and specifications in this book are for illustrative purposes only, and are therefore provided as is, without warranty of any kind, neither expressed nor implied that the examples work or are fit for a particular purpose. The author and publisher assume no responsibility or liability for damages or losses, neither incidental nor consequential, incurred as a result of using the provided specifications or source code, up to and including loss of business, injury, or death.

Introduction

The Premise Language is a functional prototype programming language which combines high level intrinsic primitives with seamless object persistence. The goal of the Premise Language is to provide a platform for artificial intelligence programming. Software agents written in Premise share a knowledge base where they can create, modify, and delete object instances amongst themselves in a stigmergic manner. Premise is influenced by Lisp, Self, JavaScript, OPS5, and CLIPS.

The Premise Language was developed by Michael S. P. Miller and Sheldon O. Linker during 2013 and 2014. In creating a platform for agent based programming for intelligent systems (JCB English and The Piagetian Modeler) they found a need for a very high level language in which to code asynchronous agent processes to perform complex pattern matching and inference. They explored the Sapir-Whorf hypothesis as it relates to computer programming—namely, that the primitive operations available in a computer language influence the way software developers frame and solve problems, and it was early determined that the primitives of the needed language should be very high level and include logical and similarity-based pattern matching, inference, messaging, stigmergy, and asynchrony. The language combines elements of declarative, functional, and imperative programming with seamless persistent object storage and retrieval. The goals of The Premise Language are simple:

- To define memory as a portable abstraction across different physical implementations. Premise uses a persistent object store as its memory.
- To facilitate the fast formation of relationships in a persistent memory. Constructing persisted records in Premise is the same as asserting facts to the working memory of other declarative language platforms such as CLIPS or OPS.
- To allow rapid interrogation of relationships in the memory.
- To provide scalability to billions of persisted records.
- To explore the Sapir Whorf hypothesis as it relates to computer programming: that the constructs available in computer languages influence the way people think about and solve problems.
- To explore the stigmergic programming paradigm, where objects constructed by one agent are later used or consumed by other agents.
- To provide a clear and concise API.

The Premise Language serves as a testbed for agent programming. Functions are the primary unit of processing. Numbers, strings, literals, prototypes, persisted records, lexicons, lists and calls provide the primary data types. The Premise Language opens up new possibilities for knowledge representation and artificial intelligence programming.

We hope you enjoy using this language.

Michael Miller

February 2019 (version 1.0)

July 2025 (version 3.0)

Document Conventions

The following are conventions used in this document.

Source Code

Premise source code will appear in Courier New font in a colored text region. The gray text region depicts a session in the Premise interpreter.

```
> (take "{a b c}")
.: {a b c}
```

A green text region depicts Premise source code snippets that appear in a file editor.

```
(relation Problem
  :Name  :Status
)
```

A blue text region depicts non-Premise source code snippets that appear in a file editor.

```
SELECT DISTINCT LENGTH(CategoryName)
  FROM Category
```

Source Code Comments

In-line comments are denoted by semicolons. Everything from the comment to the end of the line is ignored by the interpreter.

```
; this is an in-line comment
```

Multi-line comments are delimited by double quotation marks. Everything between the delimiters is treated as a single string and returned as-is by the interpreter. A free standing string, not part of a function or macro call, evaluates to itself, and thus can be used as a comment.

```
"this is a
multi-line
comment"
```

Language Style Conventions

As a matter of style, each of the following language elements is written in a specific case:

| | |
|---------------------|--|
| Modules | are written in Pascal case. |
| Types | are written in Pascal case. |
| :Slots | are written in Pascal case with a preceding colon. |
| !methods | are written in camel case with a preceding exclamation point. |
| functions | use camel case for verbs and Pascal case for nouns. |
| ?locals | (local variables) are in camel case with a preceding question mark. |
| ?Module.Name | (modular variables) are in Pascal case with a preceding question mark. |
| ?Domain.Name | (domain variables) are in Pascal case with a preceding question mark. |
| ?Global | (global variables) are in Pascal case with a preceding question mark. |
| ?CONSTANTS | are written in upper case with a preceding question mark. |

1. Getting Started

To use the Premise Language, download the Premise executable to a folder on your computer then extract the file using standard decompression software.

To run the Premise evaluator type the following:

premise

Example:

```
c:\projects\SubThought\>premise

[Premise :Version 3.0 :Build 20250801.0001 :OS Windows 11 :Edition Community]

Enter expressions followed by a blank line. Type (help), (copyright) or (license)
for information. Type (grok "file.theory") to read a file. Type (bye) to exit.

> (copyright)

.: "Copyright (c) 2014-2025 SubThought Corporation, All Rights Reserved."

>
```


2. The Interpreter

The goal of Premise is to facilitate fast persistent relationship formation in a client server environment. Clients connect to the server and execute expressions. The expressions create and modify records in the object store.

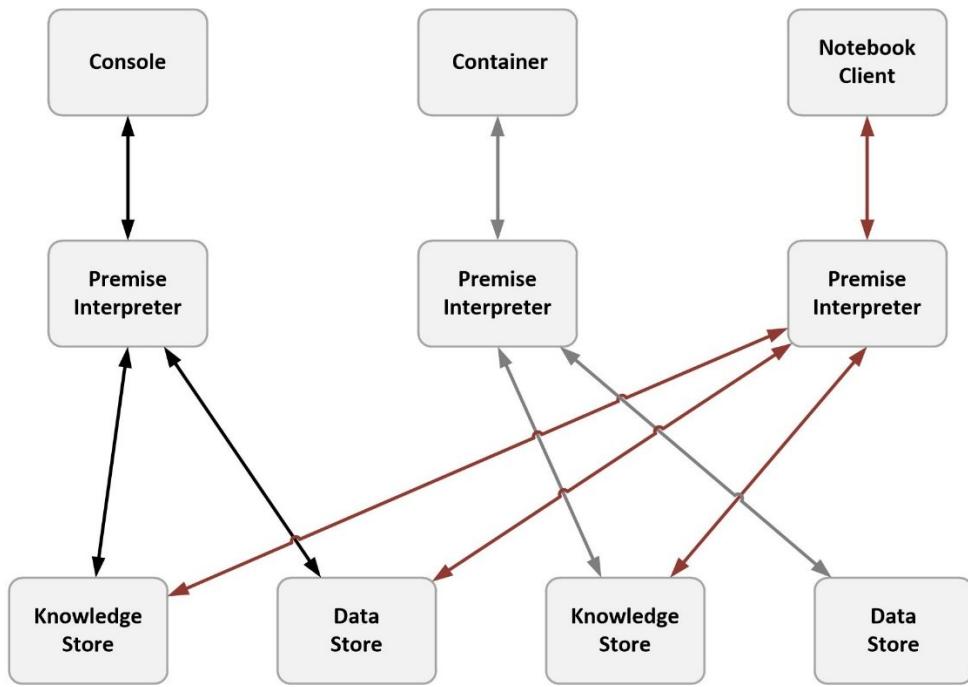


Figure 1. The Premise Interpreter Architecture

The Premise Language interpreter follows several guiding principles, among them:

1. Metacircular evaluation. Parts of Premise are written in the Premise Language itself.
2. A Lisp-1 approach to symbol and function lookup is employed. This means that variables and functions can share the same symbol environment. Because variables are prefixed by question marks and functions are not, the difference between the symbol "?foo" and the function "foo" is evident. This eliminates one source of complexity for Lisp like languages.
3. The main point of The Premise Language is to build shared structures. To that end, relations, premises, and certain containers are persisted in an object store (e.g., a RAM database) to facilitate pattern matching over persisted records, and to facilitate concurrent access to shared structures.

The Read Evaluate Emit Loop

The Premise Language interpreter, otherwise known as the Read Evaluate Emit Loop, allows users to enter expressions at the prompt (i.e., >). The interpreter evaluates the expression and returns the result. The result is printed after the ergo sign (i.e., .:). For example, when the literal true is entered at the prompt, it evaluates to itself and is returned by the interpreter as follows:

```
> true  
. : true
```

An expression that adds two numbers together would return the sum like this:

```
> (+ 1 2)  
. : 3
```

To print "Hello World" one would type the following.

```
> (tell user "Hello, World. \n")  
Hello, World.  
. : told
```

Imperative Programming

The Premise Language facilitates imperative, functional, and declarative programming. For example, adding the numbers from one to ten in an imperative style could involve defining a symbol **?total** to store the result, and looping an iteration symbol from 1 to 10.

```
> (function sum {list ?numbers}  
    (let ?total 0) ; assume ?total is 0  
    (for ?n in ?numbers  
        (set ?total (+ ?total ?n))) ; set ?total to the result of adding ?total and ?n  
        ?total) ; return ?total  
  
. : sum  
  
> (sum {1 2 3 4 5 6 7 8 9 10}) ; test the function  
. : 55
```

Functional Programming

Adding numbers in a functional style might involve generating a list of numbers, walking through the list, and adding the numbers until we reach the end of the list. This approach eliminates the need for a **?total** symbol since the result is recursively kept on the program stack.

```
> (function sum1 {list ?numbers}
  (if (void-p ?numbers) ; if the list of numbers is empty
      0 ; return zero
    else
      (+ (@ ?numbers) ; add the first number
          (sum (rest ?numbers)))) ; to the sum of the remaining numbers
  )

.: sum1

> (range 1 to 10) ; generate a list of numbers
.: {1 2 3 4 5 6 7 8 9 10}

> (sum1 (range 1 to 10)) ; test the function
.: 55
```

An even more concise approach to sum uses the function **reduce**.

```
> (function sum2 {list ?numbers}
  (reduce ?numbers +)
  )

.: sum2

> (sum2 (range 1 to 10))
.: 55
```

Declarative Programming

Declarative programming involves establishing a set facts and writing a query or rule to utilize the facts. In The Premise Language, this approach utilizes relation instances, called "thoughts."

```
> (relation Addend           ; make an Addend relation
  :Value
  :Counted false
)

.: Addend

> (step ?i from 1 to 10
  (new Addend :Value ?i))      ; make the individual addends

.: Addend_10

> (relation Total           ; make a total relation
  :Result 0
)

.: Total

> (new Total)               ; make a new total

.: Total_1

> (with
  [Total ^ as ?total]          ; match the Total
  [Addend ^ as ?addend :Value as ?value :Counted = false] ; match uncounted
  do
    (put ?addend :Counted true) ; indicate that the addend was used
    (==> ?total + :Result ?value) ; apply + to :Result and ?value, put in :Result slot
    (:Result ?total))          ; return the :Result slot value

.: 55
```

3. Taxonomy

In Premise, the data types are called **taxons** and are organized into a **taxonomy** as follows.

```
variant ; an abstract value of any type. \variant
nothing ; a concrete value representing nothing. \nothing
nil ; a concrete unspecified value. \nil
void ; a concrete value denoting emptiness \void
thing ; an abstract value representing something. \thing
container ; an abstract container. \container
assortment ; an abstract unordered grouping. \assortment
configuration ; a concrete assortment of persisted key value pairs. \configuration
enumeration ; a concrete assortment of name to number pairs. \enumeration
environment ; a concrete identifier to value mapping. \environment
domain ; a concrete rule package. \domain
module ; a concrete module. \module
context ; a concrete thought grouping \environment
problem ; a concrete thought grouping. \problem
lexicon ; a concrete assortment of key to value pairs. \lexicon
group ; an abstract unordered grouping. \group
bag ; a concrete group of possibly duplicated items. \bag
collection ; a concrete set of unique values. \collection
queue ; a concrete priority queue. \queue
state ; a concrete group of premises. \state
sequence ; an abstract ordered grouping. \sequence
call ; a concrete function call. \call
expression ; a concrete undelimited sequence of variants. \expression
list ; a concrete curly brace delimited sequence of variants. \list
series ; a concrete iterator. \series
cursor ; a concrete handle to a remote sequence. \cursor
string ; a concrete double quote a delimited character sequence. \string
vector ; a concrete pipe delimited sequence of numbers or literals. \vector
tuple ; a concrete square bracket delimited sequence of variants. \tuple
pattern ; a tuple having a literal and triples of slot, function and either value, call, or symbol. \pattern
premise ; a tuple having a literal and slot value pairs. \premise
atom ; an abstract simple thing. \atom
number; an abstract number. \number
big ; a concrete integer longer than 128 bits \big
complex ; a concrete number with real and imaginary parts \complex
imaginary ; a concrete coefficient for the square root of -1 \imaginary
integer ; a concrete 64 bit signed integer. \integer
long ; a concrete 128 bit signed integer. \long
numeric ; a concrete numeric value, one of (\infinity \negativeinfinity \undefined). \numeric
rational ; concrete number as numerator over denominator. \rational
real ; a concrete decimal number in scientific notation. \real
float ; a concrete concrete 128 bit floating point number. \float
unsigned ; concrete an unsigned 64 bit integer. \unsigned
time ; an abstract measurement. \time
instant ; an abstract single point in time. \instant
date ; a concrete UTC date. \date
epoch ; a concrete Unix epoch. \epoch
jiffy ; a concrete clock speed (0.0166667 seconds). \jiffy
moment ; a concrete nanosecond (0.0031573275 seconds). \moment
second ; a concrete second. \second
tick ; a concrete nanosecond. \tick
temporal ; a concrete time value, one of (\eternity \neternity \indefinite). \temporal
interval ; a concrete period spanning two time instants. \interval
identifier ; an abstract non-delimited case insensitive string. \identifier
symbol ; a concrete question prefixed mutable variable. \symbol
constant ; a concrete question prefixed immutable constant. \constant
reference ; a concrete pointer to a symbol. \reference
literal ; a concrete undelimited alphanumeric sequence. \literal
function ; a concrete literal that defines a function. \function
anon ; an concrete anonymous function. \anon
esonym ; an concrete internally named function. \esonym
exonym ; a concrete user named function. \exonym
generator ; a concrete function that creates a series. \generator
junction ; a concrete parallel evaluation function. \junction
macro ; a concrete user named macro. \macro
method ; a concrete exclamation prefixed function literal. \method
predicate ; a concrete monadic function returning a truth value. \predicate
procedure ; a concrete function returning nothing. \procedure
rule ; a concrete rule \rule
special ; a concrete special form \special
thunk ; a concrete evaluation delay. \thunk
reserved ; a concrete reserved set of literals. \reserved
truth ; a concrete truth value, one of (\true \false \unknown). \truth
sigil ; a concrete sigil value, one of (\optional \keyword \variadic). \sigil
slot ; a concrete colon prefixed property identifier. \slot
resource ; an abstract resource location. \resource
url ; a concrete universal resource. \url
data ; a concrete data base connection. \data
file ; a concrete file resource. \file
folder ; a concrete file resource. \file
index ; a concrete relation access resource. \index
knowledge ; a concrete knowledge base connection. \knowledge
port ; a concrete port. \port
pipe ; an concrete input output resource. \pipe
interpreter ; a concrete expression interpreter. \interpreter
reader ; a concrete expression reader. \reader
lexer ; a concrete tokenizer. \lexer
parser ; a concrete expression generator. \parser
evaluator ; a concrete expression evaluator. \evaluator
writer ; a concrete expression writer. \writer
emitter ; a concrete expression emitter. \emitter
prototype ; an abstract relationship. \prototype
relation ; a concrete persisted relationship. \relation
structure ; a concrete ephemeral relationship. \structure
record ; an abstract record. \record
thought ; a concrete persisted record. \thought
ephemeron ; a concrete ephemeral record. \ephemeron
task ; a concrete task resource. \task
uuid ; a concrete unique identifier. \uuid
```

Figure 2. Premise Taxons.

Variants

In Premise, all values are typed. Variables (also called symbols) are bound to typed values, but the symbols themselves are not typed. A variant is a value that can be any type. Variants are either nothing or something (i.e. a thing).

Nothing

The taxon **\%nothing** which evaluates to the literal **nothing** represents values that do not exist. The taxon **\%nil** represents something that is unspecified. It can be compared to the literal **nil**. Unspecified values default to nil with the **is** function. A value of **\%void** means a container has no elements. The function **null-p** can be used to determine whether a value is either unspecified or nonexistent. Containers default to empty and can be tested with the **void-p** function.

| | | |
|-----------|-------|--------|
| \%nothing | \%nil | \%void |
|-----------|-------|--------|

Things

The type **thing** represents a value that exists. The function **thing-p** can be used to detect if a value is a thing. The thing type is subdivided into **atom** and **container**.

Atoms

An atom is a number, time value , or identifier (symbol or literal).

Numbers

The number data type encapsulates several sub types: big, integer, long, unsigned, rational, real, float, imaginary, and complex.

Integers

Positive and negative decimal integers up to 64 bits are supported. Apostrophes ('') used as separators within numbers are ignored.

```
-5  
+999999999999999999
```

```
+3  
-999'999'999'999'999'999'999
```

Radix

Integers from radix 2 through radix 36 of the form \#TDDrSN... are supported, where T is a taxon indicator (n:Integer, g:Long, b:Big, r:Real, f:Float, i:Imaginary, u:Unsigned, c:Complex), D is a decimal number between zero and nine, S is an optional sign for positive (+) or negative (-) number, and N is a number from zero (0) to nine (9) or an uppercase letter from A through Z. The radix combination DD must be within the range of 2 to 36 inclusive and the digits N must be less than DD. For example #2r has digits 0 and 1 while #16r has digits 0 through F.

```
\#n02r+10110
```

```
\#n16r-778CF20
```

```
\#n36r+84QQN250MLQZ
```

Binary

Binary integers (\#n2r) are supported in the abbreviated form #\nbSN... for N equals 0 or 1.

```
\#nb+1010
```

```
\#nb-10101
```

```
\#nb0
```

Octal

Octal integers (\#n8r) are supported in the form #noSN... for N from 0 to 7.

```
\#no+1234
```

```
\#no+0
```

```
\#no+72'342
```

Decimal

Decimal integers (\#n10r) are supported by default and also in the form \#ndSN... for N from 0 to 9.

```
\#nd+9821
```

```
\#nd+0
```

```
\#nd-8'290'553
```

Hexadecimal

Hexadecimal integers (#n16r) are supported in the form #nxSN... for N from 0 to F.

```
\#nx-00FF
```

```
\#nx0
```

```
\#nx82'9CA'FF3
```

Hexatrigesimal

Hexatrigesimal integers (\#n36r) are supported in the form \e#nzSN... for N from 0 to Z.

```
\#nz-QA5N
```

```
\#nz0
```

```
\#nz+7'0M6'Z8P
```

Longs

A long number is a decimal integer that is 128 bits.

```
\#gd+10000000000000000000000000000000
```

```
\#gd-800'000'000'000'000'000'000'000
```

Big

A big number is a decimal integer that is longer than 128 bits.

```
\#bd+10000000000000000000000000000000
```

```
\#bd0
```

```
\#bd-80000000000000000000000000000000
```

```
\#bd2392837492837965821879810568725187197293872938749283749287658273648276387628376248  
73648273648726384768273684276348726384762874823648276348726384762834762876348273648726  
38476824768237648276348273648276384726834762837648768
```

Rationals

Rational numbers have a decimal integer numerator over a decimal integer denominator.

```
3/4
```

```
-8/20
```

Reals & Floats

Real numbers are 64 bit decimal floating-point numbers in scientific notation. A **float** is a sub type of real. Float numbers are 128 bits, have higher precision and a smaller range than real numbers. Floats can be used for monetary values. Float calculations must be made explicit by wrapping them in the float function , e.g. (+ (float 100) (float 200)), or prefixing #fd before the number.

```
10.0  
(real 0.001)  
\#rd+2342.243  
\#rd-54.50
```

```
-98'768'762'983.237876867749  
(float -2342.0e-23)  
\#fd+234.0e+0  
\#fd124.95
```

Imaginaries

Imaginary numbers are decimal numbers having a floating point number suffixed by the letter "i" (representing the square root of -1).

```
\#id+21
```

```
\#id-5.0
```

```
\#id2.9
```

```
\#id-9.93
```

Complexes

Complex numbers include a decimal real part plus a decimal imaginary part.

```
\#cd+55+34i  
\#cd+0-2i
```

```
\#cd-2.35+7i  
\#cd75+i
```

```
\#cd-32-14.3i  
\#cd0+i
```

Unsigned

Unsigned decimal integers up to 64 bits are supported. They are prefixed by the letter "u"

```
\#ud+0  
\#ud+18446744073709551615
```



```
\#ud+3  
\# ud+99'999'999'999'999'999'999
```

Numerics

A numeric represents any of the values for infinity, negative infinity or undefined numbers.

```
\%numeric
```

Infinity

The infinity taxon represents a positive infinite number.

```
\%infinity
```

Ninfinity

The ninfinity taxon represents a negative infinite number.

```
\%ninfinity
```

Undefined

The undefined taxon represents an impossible or undefined number.

```
\%undefined
```

Time

Time is supported in several varieties: dates, epochs, intervals, moments, seconds, and ticks.

Dates

A date represents time as a universal time coordinate (YYYY-MM-DDThh:mm:ss.sZ) in Zulu time or with an offset.

```
\@d1990-01-10T09:30:00.001Z      \@d2016-11-24T08:15:30-08:00
```

Epochs

The epoch data abstraction represents time in unix epochs as a 128 bit integer value prefixed by the letter "e". An epoch represents the number of seconds since midnight January 1, 1970.

```
\@e10          \@e592          \@e60
```

Jiffies

The jiffy data abstraction represents time as 1/60th of a second or as 16,666,666 nanoseconds.

```
\@j-12          \@j+600          \@j60
```

Intervals

The interval data abstraction represents a time coordinate as a pair of time points. The lesser time point is followed by the greater time point.

```
\@if{\@m2012001005050000 \@m2025500000000000}      \@if{\@s3 \@s10}  
\@if{\@d2012-05-19 \@d2013-07-23}                  \@if{\@u946 \@u1577}
```

Moments

The moment data abstraction represents time in nano-years (approximately 0.0031573275 seconds) as a 16 digit, 64 bit, integer value prefixed by the letter "m".

```
\@m2025500000000000          \@m-1  
\@m2012001005050000          \@m+3000      ; approximately 10 seconds
```

Seconds

The seconds data abstraction represents time in seconds as a 64 bit integer value prefixed by the letter "s".

```
\@s10           \@s592           \@s60
```

Ticks

The tick data abstraction represents time in nanoseconds as a 64 bit integer value prefixed by the letter "t".

```
\@t2342           \@t100000
```

Temporal

The temporal taxon represents eternity, negative eternity, impossible, or indefinite time.

```
\%temporal
```

Eternity

The eternity taxon represents an infinite amount of time in the future.

```
\%eternity
```

Neternity

The neternity taxon represents an infinite amount of time in the past.

```
\%neternity
```

Indefinite

The indefinite taxon represents an undefined or impossible time. Usually the default for an instant.

```
\%indefinite
```

Identifiers

An identifier is an alphanumeric sequence of characters not delimited by quotes.

Symbols

A symbol is a variable, i.e., an identifier that has both a name and an associated value. In the Premise Language, symbols are literals starting with a question mark and ending with a sequence of numeric or alphanumeric characters. When symbols are declared their value is immediately set to an initial value. Referencing an undeclared symbol will cause a failure. Symbols are typically written in lowercase. Dots in the symbol name indicate a symbol within a particular module. For example:

```
?height      ?User.width      ?length      ?2
```

Constants

A constant is a symbol whose value cannot change. By convention constants are written in uppercase.

```
?MAX-LIMIT      ?WIDTH-MIN      ?*DEFAULT-LENGTH*
```

References

A symbol reference is a pointer to a symbol. A reference allows a symbol value to be passed as a parameter. A reference is printed as `\&` and the symbol name. (e.g., `\&?me`).

```
(reference ?MAX-LIMIT)           (reference ?2)
\&?MAX-LIMIT                      \&?2
```

Symbol Scoping

Symbol scope refers to the environment a symbol belongs to. A symbol may have one of five scopes: global, module, lexical, dynamic, task, and restricted.

Global Scope

Global variables are declared using the **global** function.

```
> (do (global ?X 1)
      (function g (tell user ($$ ?X \s)) (set ?X 2))
      (function f (global ?X 3) (g)))
     (f)
     (tell user ($ ?X ($))))
3 2 .: told
```

Module Scope

A symbol may be scoped to the current module using the **modular** function.

```
> (do (global ?X 1)
      (function g (tell user ($$ ?X \s)) (set ?X 2))
      (function f (modular ?X 3) (g)))
     (f)
     (tell user ($ ?X ($))))
3 1 .: told
```

Lexical Scope

Lexical scoping in Premise is achieved via the **so**, **let**, and **var** special forms. Multiple tasks may access the same lexical symbol. The **so** special form creates a new scope, while **let** and **var** add variables to the existing scope.

```
> (do (global ?X 1)
      (function g (tell user ($$ ?X \s)) (set ?X 2))
      (function f (so {?X 3} (g)))
      (f)
      (tell user ($ ?X ($)))))

3 1 .: told

> (do (global ?X 1)
      (function g (tell user ($$ ?X \s)) (set ?X 2))
      (function f (let ?X 3) (g)))
      (f)
      (tell user ($ ?X ($)))))

3 1 .: told

> (do (global ?X 1)
      (function g (tell user ($$ ?X \s)) (set ?X 2))
      (function f (var ?X 3) (g)))
      (f)
      (tell user ($ ?X ($)))))

3 1 .: told
```

Dynamic Scope

A **dynamic** scope creates temporary global variables.

```
> (do (global ?X 1)
      (function g (tell user ($$ ?X \s)) (set ?X 2))
      (function f (dynamic {?X 3} (g)))
      (f)
      (tell user ($ ?X ($)))))

3 2 .: told
```

Task Scope

Variables **local** to a particular task are shielded from changes by other tasks.

```
> (do (global ?X 1)
      (function g (tell user ($$ ?X \s)) (set ?X 2))
      (function f (local {?X 3} (g)))
      (f)
      (tell user ($ ?X ($)))))

3 1 .: told
```

Restricted Scope

Resctricted scopes **only** shadow variables and prohibit free variables.

```
> (do (global ?X 1)
      (function g (tell user ($$ ?X \s)) (set ?X 2))
      (function f (only {?X} (g)))
      (f)
      (tell user ($ ?X ($)))))

1 1 .: told
```

Symbol Assignment

Variables may be bound to values in a variety of ways.

Fix

The **fix** special form declares taxon restricted variables in the current environment, returning true.

```
> (fix integer ?age 10 string ?name "John Doe" literal ?fruit grape)
.: true
> ?age ?name ?fruit
.: 10 "John Doe" grape
```

Let

The **let** special form declares and initializes variables in the current environment, returning true.

```
> (let ?X 1 ?Y 2 ?Z 3)
.: true
> (let {?a ?b ?c} {dog cat fish})
.: true
> ?a ?b ?c ?X ?Y ?Z
.: dog cat fish 1 2 3
```

Var

The **var** special form declares and initializes variables, returning the value of the last assignment.

```
> (var {?a ?b ?c} {dog cat fish} ?X 1 ?Y 2 ?Z 3)
.: 3
> ?a ?b ?c ?X ?Y ?Z
.: dog cat fish 1 2 3
```

Tie

The **tie** special form updates declared variables, returning the last assigned value.

```
> (tie ?X 1 ?Y (++ ?X) ?Z (++ Y))  
. : 3  
  
> (tie {?X ?Z} {2 2})  
. : 2  
  
> ?X ?Y ?Z  
. : 2 2 2
```

<- Before Tie

The **before tie** special form captures a symbol's return value before binding the symbol to an expression.

```
> (global ?X 1 ?Y 1 ?Z 1)  
. : true  
  
> (<-- ++ ?X)  
. : 1  
  
> ?X  
. : 2
```

--> After Tie

The **after tie** special form captures a symbol's return value after binding the symbol to an expression

```
.  
  
> (global ?X 1)  
. : true  
  
> (--> + ?X 1)  
. : 2  
  
> ?X  
. : 2
```

Set

The **set** special form updates declared variables, and returns **true**.

```
> (set ?X 1 ?Y (++ ?X) ?Z (++ Y))  
. : true  
> ?X ?Y ?Z  
. : 1 2 3
```

Bind

The **bind** special form declares variables in the current environment and binds them to object property values, object method values, sequence positions, or functions, and then returns **true**.

```
> (relation Counter  
    :Value 0 :Resets 0 :Reclaim false  
)  
. : Counter  
  
> (new Counter)  
. : Counter_1  
  
> (bind Counter_1 :Value ?value)  
. : true  
  
> ?value  
. : 0  
  
> (bind {a b c d e f g} 1 ?first # ?last)  
. : true  
  
> ?first ?last  
. : a g
```

Literals

Literals are non-numeric constants which evaluate to themselves. Typically, literals begin with an alphabetic or punctuation character (such as colon or exclamation point), and contain contiguous sequences of alphabetic, numeric, underscore or period characters. Literals provide names for functions, relationship prototypes, enumeration values, and so forth. Some examples of literals are as follows:

| | | | |
|-----|---------|-------------|---------------------|
| a | one | :2 | three |
| a_1 | a_12-34 | hello-world | the_quick_brown_fox |
| * | ! | ::= | -a-curious-literal |

Structures

Structures are unpersisted ephemeral relationships, created with the **structure** function.

```
> (structure Fault :What :Where :When)
.: Fault

> (Structure Message :Sender :Recipient :Content)
.: Message
```

Slots

Slots are literals preceded by a colon. Some examples are:

```
:Height
```

```
:Width
```

```
:Length
```

```
:321
```

```
^
```

The identifier slot is the caret character: ^.

Slots hold data values. In relation definitions, slots are followed by values called **defaults**. In instances, slots are followed by actual values called **terms**. When no default or referent is present for a slot, the value **nil** is implied.

The following are examples of relationship prototypes with slots and defaults.

```
(structure Red
  :Object corvette
  :Size little )

(structure Phrase
```

```
:Saying "Who goes there"
:Count 14 )
```

The following are some examples of premises with slots and terms.

```
[Red ^ Red_56 :Object corvette :Size litte]
[Event ^ Event_101 :Subject bird]
[Basket ^ Basket_12 :Items {apple banana orange}]
[Person ^ {Person_201 Person_202 Person_203} :Name "John Smith"]
```

When a call has a relation slot as its first item and a premise or thought as the second item, the call will return value of the slot within the premise or from the thought.

```
> (:Color [Fruit :Color Green])
.: Green
```

```
> (:Quantifier Fact_35)
.: Some
```

Relations

A persisted relationship is comprised of a type literal, slots, methods, and default values. A thought premise has a relation literal, slots, methods, and actual values. Relationships are defined using the **relation** function. Thoughts are created using the **new** or the **knew** function. Finally, there are three kinds of slots for a Relationship: data slots (prefixed with a colon), method slots (prefixed with an exclamation point), and identity slots (using the caret character to name the slot).

The **relation** function is used to define a relationship. The relation definition starts with a left parenthesis, followed by the literal **relation**, followed by an identifier for the type name, followed by any relation dependencies (each dependency is preceded by the **uses** keyword), then the slots and default values, methods and function names, and finally ending with a right parenthesis.

A relationship prototype must be defined before a new thought can be formed. Consider the following definitions:

```
> (relation Fruit
  :Color
  :Weight 1.0
  !eat      Fruit_eat
)
.: Fruit
```

```
> (relation Apple
  uses Fruit
  :Color red
)
.: Apple
```

```
> (relation Delicious
  uses Apple
  :Color yellow
)
.: Delicious
```

The Identity Slot

Thoughts have an identity slot (^) that contains the identifier value which uniquely demarcates the thought. Some examples of thought identifiers are

| | | |
|-------------------|--------------|--------|
| Sentence_1 | Red_34 | sov_74 |
| Vocalization_9202 | Instance_983 | |

Thought premises also contain an identity slot. For example,

```
> (premise Red_1)
.: [Red ^ Red_1 :Object corvette :Size little]
> (premise Event_29)
.: [Event ^ Event_29 :subject bird]
```

A self reference value can be used when forming thoughts using the **new** function. Self (**\^**) is equivalent to a "this" symbol in some structured languages such as C++, or the **?me** symbol in Premise.

```
(new Car :make Dodge :model RAM :year 2010 :id \^)
```

Methods

Methods are literals preceded by an exclamation point. Some examples are:

```
!action           !cleanup          !onNew           !onOld
```

Relationships have several built in methods which can be implemented by the developer: a constructor (**!onNew**), destructor (**!onOld**), copy method (**!onCopy**), and an update method (**!onSet**). The **!onNew** method initializes a premise. The **!onOld** method disposes of a premise.

Calling a method will look up the function name located in the method and invoke the function on the arguments. The methods **cloneMe** and **Concept_Add** will be retrieved when invoked.

```
[MyObject  !save   saveMe   !clone  cloneMe]  
[Concept    :Datum  0  !add   Concept_add ]
```

There are two ways of invoking a method. Methods are invoked using the **call** function or by using the method name as a function (i.e. by placing the method name first in a call followed by the instance containing the method). A method not found failure will be thrown if the instance does not contain the method.

```
(call  !start  car_255  ?myKey)      ; invoking a method using the call function.  
(!start  car_255  ?myKey)            ; invoking a method using the method name.
```

When a call has a method as its first item and a premise or identifier as the second item, the call will return the value of applying the method to the remaining arguments.

```
> (relation Foo
  :Datum 100
  !add2  Foo_addTwo
)

.: Foo

> (function Foo_addTwo {?me}
  (==> ?me + :Datum 2)
)

.: Foo_addTwo
```

```
> (new Foo :Datum 1000)

.: Foo_1

> (:Datum Foo_1)

.: 1000

> (!add2 Foo_1)

.: 1002
```

Resources

A resource is a literal which represents a tuple. Resources can be created for tasks, files, assortments, data connections, and so forth. Typically a resource will indicate the type of resource, followed by a hyphen, followed by an integer resource number.

File-45
Lexicon-87

Task-32
Environment-67

Data-988
Colors

Assortments

Assortments are resources that hold unordered elements, associations, bindings, or enumerations. Assortments such as collections, lexicons, environments, and enumerations are provided along with specific functions to manipulate them (e.g., add, cut, item, keys, put, the, tie, values, etc.).

```
> (lexicon q nil)

.: Lexicon-1

> (add Lexicon-1 "z" foo)

.: Lexicon-1

> (add Lexicon-1 84 nil)

.: Lexicon-1

> (entries Lexicon-1)

.: {{q nil}{'z' foo}{84 nil}}
```

```

> (cut Lexicon-1 "z")

.: Lexicon-1

> (entries Lexicon-1)

.: {{84 nil}{q nil}{}}

> (lexicon :Length 24 :Width 12 :Height 9 :Units inch)

.: Lexicon-2

> (@ Lexicon-2 :Length)

.: 24

> (@ Lexicon-2 :Width)

.: 12

> ?width

.: [Failure :Name UndefinedSymbol :Text "Symbol ?width is undefined."]

> ?height

.: [Failure :Name UndefinedSymbol :Text "Symbol ?height is undefined."]

> (bind Lexicon-2 :Width ?width :Height ?height)

.: true

> ?width

.: 12

> ?height

.: 9

> (keys Lexicon-2)

.: { :Height :Length :Units :Width }

> (values Lexicon-2)

.: {inch 12 24 9}

> (enumeration Colors
    red orange yellow green blue indigo violet)

.: Enumeration-1

> (@ Enumeration-1 red)

.: 1

> (keys (enum Colors))

.: {red orange yellow green blue indigo violet}

> (values Enumeration-1)

```

```
.: {1 2 3 4 5 6 7}

> (@ Enumeration-1 green)
.: 4

> (entries Enumeration-1)
.: {{red 1}{orange 2}{yellow 3}{green 4}{blue 5}{indigo 6}{violet 7}}

> (enumeration Controls
  .: clock 100
  .: timer 200
  .: start 300)

.: Enumeration-2

> (@ (enum Controls) timer)
.: 200

> (keys Enumeration-2)
.: {clock start timer}

> (collect ?key ?value per (sort (entries Enumeration-2) {< 2}) ?key)
.: {clock timer start}
```

```

> (enumeration Highways

  Maine-Miami          ; defaults to 1
  Maine-Idaho          ; defaults to 2
  Michigan-Washington 10
  NewYork-Louisiana   ;will be 11
  Massachusetts-Oregon 20
  Ohfile-Florida       ; will be 21
  NewJersey-Oregon 30
  Michigan-Alabama
  NewJersey-California 40
  Michigan-Florida
  Maryland-Nevada 50
  Wisconsin-Louisiana
  Illinois-California 60
  Minnesota-Louisiana
  NorthCarolina-Arizona 70
  JeffersonHighway
  Georgia-California 80
  NorthDakota-Texas
  Florida-Texas 90
  Montana-Nevada
  Washington-California 101)

.: Enumeration-3

> (sort (values (enum Highways)) <)
.: {1 2 10 11 20 21 30 31 40 41 50 51 60 61 70 71 80 81 90 91 101}

> (@ (enum Highways) NewJersy-California)
.: 40

```

Collections

Collections are sets that contain only keys. They are created with the **collection** function.

```
> (collection
  the quick brown fox 21  "over the" lazy dogs [MyObject ^ MyObject_32] )
.: Collection-1
```

Configuration

Configurations are key value pairs that are persisted to a file. They are created with the **configuration** function.

```
> (configuration "file://localhost.com/c$/apps/Premise/"  
  "/mind/agent/delay" \@m35  
  "/mind/activation/history/maximum" 15)  
  
. : Configuration-1
```

Environments

Environments contain identifier to value bindings. They are created with the **environment** function.

```
> (environment nil  
  (symbol x) 32  times10 (given {?x} (* ?x 10)) )  
  
. : Environment-1
```

Enumerations

Enumerations contain literal elements. Each element of the enumeration has a position. By default the first element will have position one, the second two, and so forth. If any of the arguments to the enumeration function is numeric, then that argument will set the position for the preceding literal. Enumerations are created with the **enumeration** function.

```
> (enumeration Colors red orange yellow )  
  
. : Enumeration-1  
  
> (enumeration Controls clock 100 timer 200 start 300)  
  
. : Enumeration-2
```

Ephemeros

Ephemeros are manifested structures created with the **new** function. A Fault ephemeron can be defined as follows. Because they are short lived, ephemeros are represented as premises, and do not have a unique identifier. They keys for an ephemeron are fixed and are defined by the structure the ephemeron instantiates.

```

> (new Fault :What AnError :Where RightHere :When Now)
.: [Fault :What AnError :Where RightHere :When Now]
> (new Message :Sender Me :Recipient Todd :Content "Hello There")
.: [Message :Sender Me :Recipient Todd :Content "Hello There"]

```

Thoughts

A thought is record and an instance of a relation. Thoughts have predefined slots, and are added to memory using the **new** or **knew** function. A relation is a prototype that defines a fixed record structure. Each thought instance has a literal identifier. In the case below, Apple_1 and Delicious_34 are identifiers for thoughts.

```

> (new Apple :Weight 2.0)
.: Apple_1
> (new Delicious)
.: Delicious_34

```

Lexicons

Lexicons contain value to value associations. They are created with the **lexicon** function.

```

> (lexicon
  {a b c} "123" the {"t" "th" "the" "he" "e"} :Every body )
.: Lexicon-1
> (entries Lexicon-1)
.: {{a b c} "123"}{the {"t" "th" "the" "he" "e"}}{:Every body}

```

Continuations

Continuations enable jumps into functions. They are created with the **continuation** function.

```
> (function factorial_hps {?n ?k)
  (let ?a 1)
  (continuation ?m ?a)
  (if (<= ?n 0)
    (continue ?k ?a)
  else
    (--> -- ?n)
    (continue ?m (* (++ ?n) ?a)))))

.: factorial_hps

> (function factorial {?n}
  (let ?r nil)
  (continuation ?k ?r)
  (if (nil-p ?r) (factorial_hps ?n ?k) else ?r))

.: factorial

> (factorial 5)

.: 120
```

Data

Data resources enable the use of information sources. They are created with the **data** function.

```
> (global ?Data (data driver sqlserver server "//servername" port 1433
  database mydata user "myUid" password "123456"))

.: Data-1

> (fetch ?Data "select top 2 empid, name, salary from employee")

.: {{1 "John Smith" 10.50}{2 "Jane Johnson" 12.00} }

> (disconnect ?Data)

.: disconnected
```

Files

File resources enable the reading and writing of files. They are created with the **file** function.

```
> (write (file name "quick.txt" seek 1) "The quick brown fox")
.: File-1
> (close file-1)
.: closed
```

Tasks

Tasks are expressions that can be evaluated in separate threads of execution. They are created with the **task** function.

```
> (task (/ 1000 57) (** 4 20) (** 4 20))
.: {Task-1 Task-2 Task-3}
> (complete task-1 task-2)
.: {Task-1 Task-2}
> (await task-1)
.: 17.54385964912281
> (await task-2)
.: 1099511627776
> (concurrent task-3)
.: {Task-3}
> (await task-3)
.: 1099511627776
> (free {task-1 task-2 task-3})
.: freed
```

Indices

Index resources are used to increased the speed of instance retrieval. They are created with the **index** function.

```
> (relation Car :Make :Model :Year)  
. : Car  
  
> (index Car :Make :Model)  
. : Car_Index_1  
  
> (index Car :Model)  
. : Car_Index_2  
  
> (new Car :Make A :Model M1)  
. : Car_1  
  
> (with Car ^ as ?i :Model M1) list ?i)  
. : {Car_1}  
  
> (drop Car_Index_1)  
. : dropped  
  
> (drop Car_Index_2)  
. : dropped
```

Reserved Literals

Certain literals are keywords that have a distinctive meaning. Keywords can optionally be prefixed with \% to be explicit about the use of the keyword. For example, the **Truth** reserved literals represents a possible logical states.

| | | |
|---------------------|----------------------|------------------------|
| <code>true</code> | <code>false</code> | <code>unknown</code> |
| <code>\%True</code> | <code>\%False</code> | <code>\%Unknown</code> |

Intrinsic Functions

The function literals that comprise the Premise Language are reserved and should not be redefined in their native modules since attempting to do so would redefine the behavior of the language.

Modules

Modules are name spaces. A module literal is used to avoid literal collisions and provide modularity in Premise code. Modules are delimited by dots (.) within a literal. There are several initial modules.

| | |
|--------------|----------------------------|
| Apex | Global environment |
| Base | Foundational functions |
| IO | File & Messaging functions |
| KB | Knowledge access |
| Maths | Mathematic functions |
| Tasks | Task functions |
| User | Default environment |

All modules are global, meaning that the modules cannot be concatenated or nested. Modules are declared with the **module** function, and modified with the **extend** function, and deleted with the **discard** function.

Containers

Containers are groupings of values. There are two kinds of containers, assortments and sequences. Assortments are unordered resource groupings. Sequences are ordered groupings of values.

Sequences

Sequences are ordered groupings of values. The main function of a sequence is to group values into arrays, sets, or bags of items. The Premise Language provides functions for accessing sequences as a series of elements. A sequence is a string, a list, a call, or a tuple. The language provides functions to manipulate sequences (e.g. # element, make, bind, for).

```
> (@ "ABC" 2)      ; find the second element of the string
.: "B"

> (@ {The quick brown fox jumped over the lazy dogs} 5)
.: jumped

> (copy {A B C} 1 X 3 Z)    ; copy list, replacing positions 1 and 3
.: {X B Z}

> (copy "ABC" 1 X 3 Z)   ; copy string, replacing positions 1 and 3
.: "XBZ"

> ?v1
.: [Failure :Name UndefinedSymbol :Text "Symbol ?v is undefined."]

> (let {A B C} 1 ?v1 2 ?v2)
.:      true

> ?v1
.: A

> (let "ABC" 1 ?s1 2 ?s2)
.: true

?s1
.: "A"

> ?s2
.: "B"

> (for ?c in "the quick brown fox"
  (tell user ($ ?c)))
the quick brown fox .: told
```

```
> (for ?n in {1 2 3}
  (tell user ($$ (* ?n 10) \s )))

10 20 30 :: told
```

Expressions

Expressions are sequences of zero or more variants without delimiters. Expressions are usually found inside sequences such as lists, tuples, or calls.

an expression
another expression

don't do that
He said, Sure why not?

Lists

Lists are ordered groupings of values which evaluate to themselves. Lists are malleable so they can be extended or reduced and elements can be inserted or deleted. This malleability allows lists to represent stacks or queues or orderings. The empty list, { }, may be used in code for clarity. Lists are created with the **list** function.

{a b c}

{1 2 3}

{}

Vectors

Vectors are sequences of numbers delimited by pipes (vertical bar). Numbers are not supported as a distinct data type; instead a byte is an integer written in base 16.

```
| 0 |           | 1 2 3 |           | \#xFF \#xE2 \#xC8 |           | \#FFE2C872 |
```

Strings

Strings are sequences of characters delimited by double quotes. Characters are not supported as a distinct data type, instead characters are strings of length 1.

```
"a string"          "don't do that"  
"another string"    "He said, 'Sure why not?'. "
```

Backslashes provide character escaping. Doubling the backslash character within a string provides an escape for the backslash character. For Example \\ will represent a single \. Other escape characters are as follows:

| <u>Escape Sequences</u> | <u>Character Generated</u> |
|-------------------------|----------------------------|
| \\ | \ |
| \' | ' |
| \n | newline |
| \t | tab |
| \" | " |

Character values can be generated using the backslash character. The octal values \o0 to \o177777, Unicode values \u0000 to \uFFFF, and decimal values \d0 to \d65535 can be contained in strings to select specific characters.

Tuples

Tuples are elements delimited by square brackets. For example [1 2 3] or [a b c]. Tuples are often used to represent relationships. Relationship tuples are comprised of a literal type and zero or more key-value pairs (called terms) enclosed in square brackets. For example

```
[Fault :What nil :Where nil :When nil]
```

is a tuple. Resources, ephemeros, and instances can all be represented as tuples.

Premises

Premises are tuples that represent language objects.

```
> (premise (new Fault :What "Printer Error" :Where "Line 287"
           :When "2014-003-16T21:54"))

.: [Fault :What "Printer Error" :Where "Line 287" :When "2014-003-16T21:54"]
```

For a relationship or thought, the **premise** function will retrieve a premise representation of the relationship or thought. The **uses** keyword coalesces many relations into one. Consider the following example where the **get** function retrieves the instance E_1.

```
> (relation A
   :X 0
   )

.: A

> (relation B
   :Y 1
   )

.: B

> (relation C
   :Z 0
   )

.: C

> (relation D
   uses A uses B uses C
   )

.: D
```

```
> (new D)

.: D_1

> (premise D_1)

.: [D ^ D_1 :X 0 :Y 1 :Z 0]

> (relation E
   uses D
   :Z 2
   )

.: E

> (new E)

.: E_1

> (premise E_1)

.: [E ^ E_1 :X 0 :Y 1 :Z 2]
```

Patterns

A pattern is a premise that is used to match thoughts. A pattern consists of a relation name or thought identifier list and some combination of slot bindings, slot tests, or both, enclosed within square brackets. A slot binding is a slot name, followed by **as** or **each**, followed by a symbol. A slot test is either a call containing a function that returns a truth value, or a slot name, followed by a predicate function name, followed by a value . Some examples of patterns are as follows:

```
[Prediction :What ?what :Start ?start] ; two slot bindings  
[Hypothesis ^ ?me :If ?if :Then ?then] ; three slot bindings  
[Solution as ?s :Duration > \@t5000] ; one binding one test
```

Patterns can be used in **with** function calls in order to perform rule based inference. For example, we first create a relation and some instances like this:

```
> (relation Fact :All :Are)  
. : Fact  
  
> (new Fact :All people :Are mortal)  
. : Fact_1  
  
> (new Fact :All philosophers :Are people)  
. : Fact_2
```

Then the **with** expression can infer a conclusion

```
> (with [Fact :All as ?S :Are as ?M]  
        [Fact :All = ?M :Are ?P])  
get  
  (knew Fact :All ?S :Are ?P))  
  
. : [Fact ^ Fact_3 :All philosophers :Are mortal]
```

The patterns in the **with** expression above that matched the thoughts Fact_1 and Fact_2 are

```
[Fact :All ?M :Are ?P] matches Fact_1 and Fact_2  
[Fact ^ ?f :All ?S (= (:Are ?f) ?M)] matches Fact_2 only.
```

Finally, the **knew** function attempts to find an existing thought whose slots match the values. If none are found, then the function creates a new thought.

```
> (knew Fact All ?S :Are ?P)  
. : Fact_3
```

Calls

A call is a parenthetical expression which when evaluated, will result in a value. For example, the expression `(+ 1 2 3)` is a call which when evaluated will result in the value 6. The first item in the expression is the left parenthesis, followed by a function literal followed by any actual parameters, followed by a right parenthesis. The function literal may be either a language intrinsic function or a user defined function. An empty call, `()`, returns itself.

```
(+ 1 2 3)
```

```
(trim " abc ")
```

```
()
```

Functions

Functions (also called *exonyms*) are defined using **function**.

```
> (function foo {?a ?b}
  (+ ?a ?b)
  )
.: foo
```

```
> (function bar {?a ?b}
  (* ?a ?b)
  )
.: bar
```

Functions are invoked by creating a call—wrapping the name and arguments in parentheses.

```
> (foo 10 20)
.: 30
```

```
> (bar 10 20)
.: 200
```

Parameter Lists

There are several ways define the parameters to a function

1. No parameter list.

```
(function forgetEveryone
  (each (with Person) old)
)
```

2. As a list containing zero parameters

```
(function helloWorld {}
  (tell user "Hello World")
)
```

3. As a list containing one or more required parameters

```
(function addThem {?x ?y}
  (+ ?x ?y)
)
```

4. As a list containing a variadic remainder parameter which contains all passed arguments.

```
(function addAll {& ?numbers}
  (tell user ($ the sum of ?numbers is (apply + ?numbers) \n))
)
```

5. As a list containing optional parameters.

```
(function result {+ ?y}
  (--> default ?y 0)
  (tell user ($ result = ?y \n))
)
```

6. As a list containing optional parameters with default values.

```
(function result {+ ?y = 0}
  (tell user ($ result = ?y \n)) )
```

7. As a list containing required, optional, or remainder parameters.

```
(function result {?x + ?y = 1 & ?z}
  (tell user ($ result = (apply * (& ?x ?y ?z)) \n))
)
```

8. As a list containing required and optional keyword parameters.

```
(function result {{color ?c} + % {length ?n} = 0 {width ?w} = 0
  {height ?h} = 0}
  (tell user ($ color ?c))
  (tell user ($$ , \s length \s ?n , \s width \s ?w , \s height \s ?h ))
)
```

9. Function parameters are automatically evaluated by default. Macro parameters are unevaluated by default.

```
(macro result {?x ?y ?z}
  (tell user ($$ (eval ?x( " , " (eval ?y) " , " (eval ?z) \n)))
)
```

Parameter Specifications

Any required parameters are specified first. A failure is caused if a required parameter is not supplied with an argument when a function is called.

```
(function addThem {?x ?y} ; ?x and ?y are required.
  "Add two numbers."
  (+ ?x ?y)
)
```

The optional parameter sigil, the plus sign (+), precedes any optional ordered parameters. Optional parameters are processed sequentially and are bound to **nil** if no default values are specified.

```
(function AddTwoToFive {?a ?b + ?c = 0 ?d = 0 ?e}
  "Add between two and five numbers."
  (apply + (& ?a ?b ?c ?d ?e)
)
```

Default values may immediately follow an optional parameter symbol. The default value sigil, the equal sign (=), precedes a default value to a symbol. Default values may only be literals, times, numbers, or strings. The default value must not be a symbol, resource, tuple, call, or list.

```
(function addThem {+ ?x = 0 ?y = 0}; ?x and ?y both default to zero.  
  "Add two numbers."  
  (+ ?x ?y)  
)
```

A keyword parameter is a list containing a keyword pattern and symbol pair (e.g., {literal ?symbol}). Keyword parameters may be supplied instead of a symbol. They keyword literal is matched to the arguments and the value following the literal is placed into the symbol.

```
(macro myFor {?y {as ?x} & ?body}  
  (eval (` (for , ?x in , ?y , ?body)))  
)
```

A function or macro containing required keyword parameters would be called as follows

```
(myFor {Jane Jack Joe} as ?name (tell user ($ ?name \n)))
```

Optional keywords are matched sequentially. Unordered keywords require the unordered keywords sigil, the percent sign (%). Unordered keyword arguments are matched in any order they occur in the argument list if their formal parameters follow the unordered keywords sigil.

```
(macro myStep { ' ?v {from ?a} + % {to ?z} {by ?i} = 1 & ?body}  
  (eval (` (step , ?v from , ?a to , ?z by (default , ?i 1) , ?body)))  
)
```

A variadic parameter sigil, the ampersand (&), ties the remaining arguments together into a list for the subsequent symbol. Only one symbol is permitted after the ampersand. The remainder parameter will be bound to an empty list if no actual parameters are supplied.

```
(function AddTwoOrMore {?x ?y & ?z}  
  (if (void-p ?z)  
    (+ ?x ?y)  
  else  
    (apply + (& ?x ?y ?z)))  
)
```

It would be called as follows

```
(myStep ?i to 100 by 2 from 1  (tell user ($ ?i \n)))
```

In functions, all arguments are evaluated before the function body is applied. This is not the case for macros. Macro parameters remain unevaluated until they are explicitly evaluated with the **eval** function within the body of the macro definition.

```
(macro callLength {?f}
  (ensure Call ?f)
  (# (eval ?f))
)
```

More examples...

```
(function foo
  "No parameter list."
  (do nothing))

(function foo {}
  "Zero length param list."
  (do nothing))

(function foo {?x ?y}
  "Fixed number of required parameters."
  (+ ?x ?y))

(function foo {+ ?x = 0 ?y = 0}
  "Optional parameters with default values."
  (+ ?x ?y))

(function foo {?w + ?x ?y & ?z}
  "Required, optional, and remaining parameters."
  (--> default ?x 0)
  (--> default ?y 0)
  (apply + (& ?w ?x ?y ?z)))

(function foo {+ ?w = 0 ?x = 0 ?y = 0 & ?z }
  "Optional and remaining parameters with default values."
  (apply + (& ?w ?x ?y ?z)))

(macro define {?name ?args & ?body}
  "Unevaluated parameters."
  (eval (` (function , ?name , ?args ,_ ?body))))
```

Auto-named Functions

Auto-named functions (also called *endonyms*) are created by omitting the function name in the definition. When the function name is omitted, the Premise interpreter will automatically generate a function name for the function.

```
> (function {?x ?y}
  (* ?x ?y)
  )
.: fn-24
```

```
> (function & ?args
  (apply * ?args)
  )
.: fn-25
```

```
> (function
  (with Truck ^ as ?t
    do (old ?t)))
.: fn-26
```

Anonymous Functions

Anonymous functions (i.e., also called *anonyms*) are implemented using **given**. An anonym expression can stand in for a named or auto-named function call.

```
> ((given {?x ?y} (* ?x ?y)) 2 3)           ; same effect as (fn-24 2 3)
.: 6
```

Generators

A generator is a function that creates a series. Generators are created using the **generator** function. When a generator is invoked it will produce a series.

```
> (generator generateABC {}
  (yield a)
  (yield b)
  (yield c)
  (stop))
.: generateABC

> (collect ?e in (generateABC) ?e)
.: {a b c}

> (generateABC)
.: {...}
```

Macros

A macro is a call that evaluates an expanded expression. Macros are defined using **macro**. The back quote function (`) expands an expression by substituting subexpressions that follow commas, then it evaluates the resultant expression.

```
> (macro plus {?a ?b}
  (eval (` (+ , ?a , ?b)))
)
.: plus
```

```
> (macro prod {?a ?b}
  (eval (` (* , ?a , ?b)))
)
.: prod
```

Macros are invoked by wrapping the macro name and arguments in parentheses.

```
> (plus 10 20)
.: 30
```

```
> (prod 10 20)
.: 200
```

4. Control Flow

The following expressions permit alterations to the normal sequential flow of expression evaluation.

Conditional Expressions

Conditional expressions branch the flow of evaluation to alternate paths.

if

The if special form provides branched conditional flow of evaluation.

```
> (if (= 1 2)
    (tell user ($ An impossibility.))
    (tell user ($ I'm certain. \n))
  or (= 2 1)
    (tell user ($ Another impossibility.))
    (tell user ($ Most definitely. \n))
  or (/= 1 1)
    (tell user ($ A total absurdity.))
    (tell user ($ Without a doubt. \n))
  else
    (tell user ($ The world is right as rain. \n))
  HelloWorld)

The world is right as rain.
.: HelloWorld
```

case

The case special form provides branching based on a value.

```
> (case red
  of green (tell user ($ Nature's)) (tell user ($ glory. \n))
  of blue (tell user ($ A wondrous)) (tell user ($ beauty. \n))
  in {brown black purple indigo violet} (tell user ($ A fabulous selection. \n))
  else
    (tell user ($ Hmm... I didn't consider that one.))
    (tell user ($ ($) Bravo. \n))

Hmmm... I didn't consider that one. Bravo.
.: nil
```

[on](#)

The **on** special form provides branched conditional flow of evaluation.

```
> (on (= 1 2)
      (tell user ($ A truth.))
      (tell user ($ An impossibility. \n)))

An impossibility
.: told

> (on (= 1 1)
      (tell user ($ Without a doubt. \n))
      (tell user ($ A total absurdity.)))

Without a doubt.
.: told
```

[do](#)

The **do** special form provides subroutine resumption.

```
> (do
    (escape)
    (never-happens)
  resume
    (tell user ($ Resumed execution \n))
  finally
    (tell user ($ Done. \n)))

Resumed execution
Done.
.: told
```

[try](#)

The **try** special form provides exception handling,

```
> (try
    (signal [Failure :Name MyFault :Text "Failed Miserably"])
  learn ?failure
    (tell user ($ Handled ?failure \n))
  finally
    (tell user ($ Done. \n)))

Handled [Failure :Name MyFault :Text "Failed Miserably"]
Done.
.: told
```

Looping Expressions

Certain expressions are used to loop through a sequence or iterate over a series of values.

loop

The **loop** special form allows conditional repetition of one or more expressions.

```
> (global ?I 100)  
.: true  
  
> (loop (--> -- ?I) until (= ?I 90))  
.: 90  
  
> (loop (--> ++ ?I) while (< ?I 100))  
.: 100  
  
> (loop (--> -- ?i) repeat 100)  
.: 0
```

for

The **for** special form repeatedly binds one or more variables through elements of a sequence.

```
> (for ?x in {hello world} (tell user ($$ ?x \s)))  
hello world .: told  
  
> (for ?x in "hello world" (tell user ($$ ?x \s)))  
h e l l o   w o r l d .: told  
  
> (for ?k ?v in {a 1 b 2 c 3 d 4}  
    (tell user ($ ?k ($$ ?v ,))))  
  
a 1, b 2, c 3, d 4, .: told  
  
> (for ?x ?y ?z per {{a 1 $}{b 2 @}{c 3 #}{d 4 /}}  
    (tell user ($ ?x ?y ($$ ?z ,))))  
  
a 1 $, b 2 @, c 3 #, d 4 /, .: told
```

[step](#)

The **step** special form increments or decrements a symbol over a series of numeric values.

```
> (step ?i from 5 to 9
  (tell user ($$ "?i = " ?i ", " )))

?i = 5, ?i = 6, ?i = 7, ?i = 8, ?i = 9, .: told

> (step ?i from 100 to 50 by -10
  (tell user ($$ "?i = " ?i ", " )))

?i = 100, ?i = 90, ?i = 80, ?i = 70, ?i = 60, ?i = 50, .: told

> (step ?i from 0 to 0.5 by 0.1
  (tell user ($$ "?i = " ?i ", " )))

?i = 0, ?i = 0.1, ?i = 0.2, ?i = 0.3, ?i = 0.4, ?i = 0.5, .: told
```

[repeat](#)

The **repeat** special form allows repetition of one or more expressions for a specified number of iterations.

```
> (repeat 1000 (--> ++ ?i))

.: 1100

> (repeat 5 (tell user ($$ hi \s))

hi hi hi hi hi .: told
```

[count](#)

The **count** special form repeatedly binds one or more variables to elements of a sequence and returns the count..

```
> (count ?x in {the quick brown fox} as ?total
  (tell user ($ ?x \t))

the      quick      brown      fox      .: 4
```

[while](#)

The **while** special form allows repetition of one or more expressions while a condition is true.

```
> (global ?N 0)

.: true

> (while (< ?N 5)
  (tell user ($ ?N \n))
  (--> ++ ?N))

0
1
2
3
4
.: 5
```

[until](#)

The **until** special form provides repetition of one or more expressions until a condition is true.

```
> (global ?S {a b c d 1 2 3 4} ?i 0 ?result {})

.: true

> (until (number-p (@ ?S (--> ++ ?i))) ; until S[++?i] is a number
  (--> & ?result {{(@ ?S ?i) bar}})) ; merge ?result with { S[?i] bar }

.: {{a bar}{b bar}{c bar}{d bar}}
```

Control Flow Expressions

Control flow expressions redirect the flow of control.

[confirm](#)

The **confirm** special form signals a failure if the test fails.

```
> (function multiply {?a ?b}
  (confirm (taxon-p ?a number) since ($ ?a must be a number))
  (confirm (taxon-p ?b number) since ($ ?b must be a number))
  (* ?a ?b))

.: multiply

> (multiply 10 banana)

.: [Failure :Name DataTypeFailure :Text "banana must be a number"]
```

[confute](#)

The **confute** special form signals a failure if the test succeeds.

```
> (confute (= red blue) since "colors must be different")

.: false

> (function reducing {?sequence ?function}
  (ensure sequence ?sequence function ?function)
  (confute (<= (# ?sequence) 1) since "Two or more elements are required.")
  (let ?result (@ ?sequence))
  (for ?element in (rest ?sequence) (--> ?function ?result ?element)))

.: reducing

> (reducing {1} +)

.: [Failure :Name Confutation :Text "Two or more elements are required."]
```

break

The **break** special form transfers control out of the containing loop.

```
> (do (step ?i from 1 to 10
           (if (= ?i 3) (break nil))
           (tell user ($ ?i \n)))
           (tell user ($ done \n)))

1
2
done
.: nil
```

continue

The **continue** special form transfers control to the next iteration of the containing loop.

```
> (for ?letter in "Premise"
       (if (in "meis" ?letter) (continue))
       (tell user ($ Letter is ($$ ?letter .) \n)))

Letter is P.
Letter is r.
.: nil
```

exit

The **exit** special form transfers control out of the current task. Whereas return may exit from a nested function, exit terminates the task. Exit can be thought of as a task level return.

```
> (concurrent
    (step ?x from 1 to 999999999
        (if (> ?x 100) (exit exited)))))

.: {Task-1}

> (await task-1)

.: exited

> (free task-1)

.: freed
```

signal

The **signal** special form transfers control to an encompassing try expression.

```
> (function myFun
  (try
    (signal [Failure :Name BadAttitude :Text "example"])
    learn ?f
    (case (:Name ?f)
      of BadAttitude
        (tell user ($ Caught inside myFun. \n))
        (signal ?f)))
  )

.: myFun

> (function main [& ?args]
  (try
    (myFun)
    learn ?f
    (case (:Name ?f)
      of BadAttitude
        (tell user ($ Caught inside main. \n))))
  )

.: main

> (main)

Caught inside myFun.
Caught inside main.
.: nil
```

return

The **return** special form transfers control out of the current function.

```
> (function a {?i} (if (= ?i 5) (return nil)) (tell user ($ ?i \n)))
.: a

> (function b {?i} (if (> ?i 4) (return 4 from b)) (tell user ($ ?i \n)) (return 2))
.: b

> (function c {?k ?v} (return {?k ?v} from function))

.: c

> (function d (a 30))

.: d

> (do
  (a 5)
  (a 1)
  (let ?i (b 5))
  (tell user ($ ?i \n))
  (tell user ($ (@ (c 100 90) 2) \n))
  (d))

1
4
90
30
.: nil
```

ensure

The **ensure** special form transfers control to an encompassing try expression if a type check fails.

```
> (function multiply {?a ?b}
  (ensure number ?a number ?b)
  (* ?a ?b))

.: multiply

> (multiply 10 banana)

.: [Failure :Name EnsureType :Text "The value banana must be a number"]

> (multiply 10 5)

.: 50
```

[escape](#)

The **escape** function transfers control to an encompassing **do** expression containing a **resume** tag.

```
> (do
  (escape)
  resume here
  (tell user ($ Resumed execution \n))
  finally
  (tell user ($ Done. \n)))  
  
Resumed execution  
Done.
```

go

The **go** special form transfers control to a function.

```
> (relation Queue  :Items {} :Capacity 10
    !count (given {?me} (# (:Items ?me)))
    !full-p (given {?me} (>= (!count ?me) (:Capacity ?me))))  
.  
. : Queue  
  
> (function produce {?q}
    (tell user ($ producing \n))
    (if (not (!full-p ?q))
        (repeat (random 1 to (- (:Capacity ?q) (!count ?q)))
            (enq ?q :Queue each (new Item))))
        (go consume ?q)))  
.  
. : produce  
  
> (function consume {?q}
    (tell user ($ consuming \n))
    (if (more-p (:Queue ?q))
        (repeat (random 1 to (# (:Queue ?q)))
            (deliver (pop (:Queue ?q))))))
    (either (void-p (:Queue ?q))
        done
        (go produce ?q)))  
  
> (produce (new Queue))  
  
producing  
consuming  
.  
. : done
```

Multitasking Expressions

The Premise Language facilitates asynchronous and concurrent expression evaluation.

concurrent

The **concurrent** function runs expressions asynchronously and returns immediately.

```
> (concurrent (+ 1 2 3 4) (/ 1000 57))  
.: {task-1 task-2}  
  
> (await task-1)  
.: 10  
  
> (await task-2)  
.: 17.54385964912281  
  
> (free {task-1 task-2})  
.: nil
```

complete

The **complete** function executes expressions concurrently and returns after all have completed.

```
> (complete (step ?i from 0 to 50000000 (do nothing))  
           (step ?i from 1 to 10000000 (do nada)))  
.: {Task-1 Task-2}  
  
> (await task-1)  
.: nothing  
  
> (await task-2)  
.: nada  
  
> (free {task-1 task-2})  
.: freed
```

[scatter](#)

The **scatter** function applies functions in parallel and returns a list of results.

```
> (function double {?x} (* ?x 2))  
.: double  
  
> (scatter {5} {++ -- double})  
.: {6 4 10}  
  
> (scatter {{the quick brown fox jumped over the lazy dogs}}  
{except rest top last})  
.: {{the quick brown fox jumped over the lazy }  
{quick brown fox jumped over the lazy dogs}  
{the}  
{dogs}})  
  
> (scatter {2 3 4} {+ * -})  
.: {9 24 -5}
```

5. Code Examples

The following are simple examples demonstrating the language.

Inference

Logical pattern matching and declarative inference.

```
(relation Statement
  :All
  :Are
)

(new Statement :All people :Are mortal)

(new Statement :All philosophers :Are people)

(with [Statement :All as ?s :Are as ?m]
      [Statement :All = ?m :Are as ?p]
  list
    (knew Statement :All ?s :Are ?p))
```

```
> (relation Statement
  :All
  :Are
)

.: Statement

> (new Statement :All people :Are mortal)

.: Statement_1

> (new Statement :All philosophers :Are people)

.: Statement_2

> (with [Statement :All as ?s :Are as ?m]
      [Statement :All = ?m :Are as ?p]
  list
    (knew Statement :All ?s :Are ?p))

.: {Statement_3}

> (with Statement)

.: {[Statement ^ Statement_1 :All people :Are mortal]
  [Statement ^ Statement_2 :All philosophers :Are people]
  [Statement ^ Statement_3 :All philosophers :Are mortal]})
```

Memoization

Memoization is the process of saving intermediate results for lookup rather than recomputation. Once a value has been computed, it can be memorized so that if the value is needed again, it can be easily retrieved rather than recomputed. The following example uses memoization in computing Fibonacci numbers.

```
> (relation Memoized
  :Number
  :Result
  )

. : Memoized

> (new Memoized :Number 0 :Result 0)

. : Memoized_1

> (new Memoized :Number 1 :Result 1)

. : Memoized_2

> (function fib {?n}
  (confirm (>= ?n 0) since
    ($ ?n must be zero or more.))
  (with
    [Memoized ^ :Number as ?n :Result as ?r] ; if a matching instance exists
    do
      (return ?r)) ; then return, otherwise do nothing
    (let ?result (+ (fib (- ?n 1)) (fib (- ?n 2))))
    (new Memoized :Number ?n :Result ?result)
    (return ?result)
  )
  . : fib

> (fib 7)

. : 13

> (with Memoized)

. : {[Memoized ^ Memoized_1 :Number 0 :Result 0]
  [Memoized ^ Memoized_2 :Number 1 :Result 1]
  [Memoized ^ Memoized_4 :Number 3 :Result 2]
  [Memoized ^ Memoized_6 :Number 5 :Result 5]
  [Memoized ^ Memoized_3 :Number 2 :Result 1]
  [Memoized ^ Memoized_8 :Number 7 :Result 13]
  [Memoized ^ Memoized_7 :Number 6 :Result 8]
  [Memoized ^ Memoized_5 :Number 4 :Result 3]}}
```

Programming Patterns

Event driven programming.

```
> (structure Event
  :Topic
  :Content
  )

.: Event

> (relation Fault
  :Who
  :What
  :Where
  :When
  )

.: Fault

> (function dispatcher {?message}
  (try
    (let ?taken (take ?message)
      ?event (either (takeable-p ?taken) (@ ?taken) else nil)

      (ensure Event ?event)

      (let ?event :Topic as ?topic :Content as ?content)
      (case ?topic
        of Percept (onPerceptReceived ?content)
        of Ping (onPingReceived ?content)
        else
          (signal [Failure :Name BadTopic :Text ($ Unknown topic ?topic)]))
      (tell user ($ Handled ?message))
      learn ?failure
      (new Fault :Who Dispatcher :What ?failure :When (date)
        :Where {dispatcher ?message}))
    )
  )

.: dispatcher

> (global ?URL (config "/application/eventUrlAndPort"))

.: true

> (listener ?URL dispatcher) ; creates a message loop

.: Listener-1

> (tell ?URL ($ (new Event :Topic Ping :Content {:Sent (moment)})))

.: told
Handled [Event :Topic Ping :Content {:Sent \@m201804991265124}]

> (free Listener-1 cancel yes)

.: freed
```

Similarity Based Matching

The `~` and `best` language intrinsics perform similarity based comparisons to support case-based reasoning.

```
> (~ {a b c} {a d e})  
.: 0.2  
  
> (~ "a b c" "a d e")  
.: 0.090909091  
  
> (~ 57 890)  
.: 0.00119904076738  
  
> (best 2001 (range 1 to 1000000) 5)  
.: {{2001 1} {2000 0.5} {2002 0.5} {1999 0.3333333333333333} {2003 0.3333333333333333}}  
  
> (best "this" {"that" "hat" "hit" "with" "isthmus"} 3)  
.: {"that" 0.1111111111111111} {"hat" 0.0833333333333333} {"hit" 0.0833333333333333}
```

Asynchrony and Concurrency

The **concurrent**, **complete**, and **task** functions run expressions asynchronously, concurrently, or as required. The concurrence function **complete** returns after all tasks have completed whereas the asynchronicity function **concurrent** returns immediately after invocation. The **task** function creates a task for deferred execution. The functions each return a list of task handles.

```
> (concurrent (step ?i from 1 to 30000000 (do something)) ; runs asynchronously
   (step ?j from 1 to 60000000 (do something-else)))

.: {Task-1 Task-2}

> (ready-p task-1)

.: false

> (await task-1)

.: something

> (await task-2 1000 timed-out) ; adding timeout and default args

.: timed-out
```

```

> (await task-2)

.: something-else

> (free {task-1 task-2})

.: freed

> (complete (step ?i from 1 to 30000000 (do someOtherThing)) ; runs concurrently
    (step ?j from 1 to 60000000 (do anotherThing)
    (step ?k from 1 to 100000000 (do oneLastThing))

.: {Task-3 Task-4 Task-5}

> (cancel task-5)

.: cancelled

> (await task-3 0 timed-out)

.: someOtherThing

> (await task-4 0 timed-out)

.: anotherThing

> (free {task-3 task-4})

.: freed

> (function sayHello {}
  (for ?c in "hello " (tell user ($ ?c)))))

.: sayHello

> (global ?Tasks (task (sayHello) (sayHello) (sayHello) (sayHello)))

.: true

> (free (complete ?Tasks))

hhehelelohl lello l ol o .: freed

> (function sayHello {}
  (critical printing
  (for ?c in "hello " (tell user ($ ?c)))))

.: sayHello

> (set ?Tasks (task (sayHello) (sayHello) (sayHello) (sayHello)))

.: true

> (free (complete ?Tasks))

hello hello hello hello .: freed

> (undeclare ?Tasks)

.: undeclared

```

Thoughts

Thoughts are tuples which reside in a knowledge base. Thoughts are created with **new** function and are destroyed with the **old** function. The **premise** function can be used to retrieve a thought. The **zap** function sets all slots of a thought to **nil**.

```
> (relation Foo :Bar :Baz)
.: Foo

> (new Foo :Bar 100)
.: Foo_1

> (premise Foo_1)
.: [Foo ^ Foo_1 :Bar 100 :Baz nil]

> ?bar
.: [Failure :Name UndefinedSymbol :Text "Symbol ?bar is undefined."]

> (bind (premise Foo_1) :Bar ?bar)
.: true

> ?bar
.: 100

> (put Foo_1 :Baz "The quick brown fox")
.: Foo_1

> (zap Foo_1 :Bar)
.: Foo_1

> (premise Foo_1)
.: [Foo ^ Foo_1 :Bar nil :Baz "The quick brown fox"]

> (zap Foo_1)
.: Foo_1

> (premise Foo_1)
.: [Foo ^ Foo_1 :Bar nil :Baz nil]
```

The **knew** function retrieves an existing thought, or if it does not exist, creates a new thought. The **known** function retrieves an existing thought given a premise.

```
> (premise Foo_2)
.: [Foo ^ Foo_2 :Bar 100 :Baz nil]
> (premise Foo_3)
.: [Foo ^ Foo_3 :Bar 200 :Baz nil]
> (knew Foo :Bar 100)
.: Foo_2
> (knew Foo :Bar 200)
.: Foo_3
> (knew Foo :Bar 100)
.: Foo_2
> (known [Foo :Bar 100] )
.: Foo_2
> (known [Foo :Bar 100] )
.: Foo_2
> (known [Foo :Bar 200] )
.: Foo_3
> (known [Foo :Bar 400] )
.: nil
```

The **assert** and **assume** functions create a thought containing nested thoughts. When either function encounters a nested premise, it determines whether or not the nested premise has a defined relation. If so, the assert function performs a **new** on the nested premise while the assume function performs a **knew** on the nested premise. In the example below, an Operation has a :Context property containing an anonymous function for a state that returns a list of bindings if there is a match, otherwise it returns **nil**. The :Effects property contains an anonymous function that creates and edits a successor state.

```
> (relation Operation :Name :Parameters :Context :Effects :BindingsFn :SuccessorFn)

.: Operation

> (relation Is :Type :Item)

.: Is

> (relation On :Above :Below)

.: On

> (relation Has :What :Item)

.: Has
> (assume Operation
  :Name PickUp
  :Parameters {$ob}
  :Context { [On :Above CLEAR :Below $ob]
             [On :Above $ob :Below TABLE]
             [Has :What ARM :Item EMPTY]}
  :Effects { :Add { [Has :What ARM :Item $ob] }
            :Del { [On :Above CLEAR :Below $ob]
                  [On :Above $ob :Below TABLE]
                  [Has :What ARM :Item EMPTY] } } )
.: Operation_1

> (premise Operation_1)

.: [Operation ^ Operation_1 :Name PickUp :Parameters {$ob} :Mappings {$ob ?ob}
:Context {On_1 On_2 Has_1} :Effects { :Add {Has_2} :Del On_1 On_2 Has_1}]

> (assert Operation
  :Name PickUp
  :Parameters {$ob}
  :Context { [On :Above CLEAR :Below $ob]
             [On :Above $ob :Below TABLE]
             [Has :What ARM :Item EMPTY] }
  :Effects { :Add { [Has :What ARM :Item ?ob] }
            :Del { [On :Above CLEAR :Below ?ob]
                  [On :Above ?ob :Below TABLE]
                  [Has :What ARM :Item EMPTY] } } )

.: Operation_2

> (premise Operation_2)

.: [Operation ^ Operation_2 :Name PickUp :Parameters {$ob} :Mappings {$ob ?ob}
:Context {and On_3 On_4 Has_2} :Effects { :Add {Has_2} :Del {On_5 On_6 Has_3}}]
```

To consider an operation on a state is to obtain the successor state after the edits (additions and deletions) are made to a clone of the state. We create an optimized binding function and successor function for each operation and store them within the operation for later reuse.

```

> (facility symbolify {literal ?variable}
  (reference (symbol (rest ($ ?variable)))))

.: symbolify

> (facility mapVariablesToSymbols {list ?variables}
  (coalesce ?variable in ?variables
    (given {literal ?variable}
      (expression ?variable (symbolify ?variable)))))

.: mapVariablesToSymbols

> (function bindings {State ?state Operation ?operation }
  (if (null-p (:BindingsFn ?operation))
    ; create a bindings anonymous function
    (let ?mappings (mapVariablesToSymbols (:Parameters ?operation))
      ?template (:Context ?operation)
      ?context (replace ?template ?mappings))
    (put ?operation :BindingsFn
      (` (given {State ?state ?Operation ?operation}
        (within ?state :Facts ,_ ?context
          list ,_ (coalesce ?variable ?symbol in ?mappings
            (expression ?variable (` (eval ,?symbol))))))))
    ; run the bindings function
    (eval (:BindingsFn ?operation) ?state ?operation)

.: bindings

> (function successor {State ?state Operation ?operation list ?bindings}
  (if (null-p (:SuccessorFn ?operation))
    ; create a successor anonymous function
    (` (given {State ?state Operation ?operation list ?bindings}
      (do (bind ?bindings ,_ (mapVariablesToSymbols (:Parameters ?operation)))
        (let ?successor (copy ?state :Prior ?state))
        (enq ?successor :Facts each
          ,_ (map (:Add (:Effects ?operation))
            (given {?addition} (known ?addition))))
        (deq ?successor :Facts each
          ,_ (map (:Del (:Effects ?operation))
            (given {?deletion} (known ?deletion))))
        (return ?successor))))
    ; run the successor function
    (eval (:SuccessorFn ?operation) ?state ?operation ?bindings)))

.: successor

> (function consider {State ?state Operation ?operation}
  (let ?bindings (bindings ?operation ?state))
  (if (exists-p ?bindings)
    (successor ?state ?operation ?bindings)))

.: successor

```

Theories

In the Premise language, a theory is a file with a **.th** or **.theory** extension. There are two types of theory files, **domain theories** and **module theories**.

Domain Theories

Domain theories contain problem space definitions.

```
;; Arithmetic.Theory

(domain Arithmetic

(require Basic-Arithmetic from "file:///basic-arith.theory") ; a prerequisite

(rule Basic-Addition
  salience 100
  so [Problem as ?p :State as ?state]
    [Process ^ = ?state :Facts as ?facts]
    [Summation ^ as ?sum :Result as ?result]
    [Addend ^ as ?add :Value as ?value :Status = pending]
  if (in ?sum ?facts)
    (in ?add ?facts)
  do (put ?sum :Result (+ ?value (isnull ?result 0)))
    (put ?add :Status added))

(rule Report
  salience 0
  so [Problem as ?p :State as ?state]
    [Process ^ = ?state :Facts as ?facts]
    [Summation as ?sum :Result as ?result]
    [Addend ^ as ?add :Status as ?status]
  if (in ?sum ?facts)
    (in ?add ?facts)
  no [Addend = ?add :Status = pending]
  do (put ?p
    :Result (format "For ~s, the result is ~s." ?p ?result)))

(rule Clean-up
  salience -100
  so [Addend ^ as ?a :Status = added]
  do (old ?a))

(problem
  name SimpleAddition
  domain Arithmetic
  state (assume Gap
    :Facts [Addend :Value 5]
    [Addend :Value 7]
    [Addend :Value 11]
    [Summation] )))
)
```

Module Theories

Module theories typically contain function definitions.

```
; ; SuperMath.Theory

(module SuperMath

  (facility power {?x + ?y = 1}           ; private to the module
   (** ?x ?y)
  )

  (function superpower {& ?numbers}        ; public to the module
   (if (void-p ?numbers)
       (return \:unknown))
   (let ?result (@ ?numbers))
   (for ?n in (rest ?numbers)
     (--> power ?result ?n))
   ?result
  )
)
```

Domains

A domain, also called a problem space or realm, is an environment containing relationships, rules, and references to prerequisite domains. The **domain** function creates a domain.

```
> (domain Basic-Arithmetic
    (relation Summation :Result 0)
    (relation Addend :Value 0 :Status pending))

.: Basic-Arithmetic
```

Rules

A rule is a task that operates when its preconditions are satisfied. The `rule` function creates a rule, which can be either a causal forward chaining rule or a logical backward chaining rule. Backward chaining (logical) rules will include a `to` clause to indicate the goals to be achieved.

```
(rule name
  salience 0           ; the relative priority of this rule
  domain realm          ; the rule package containing this rule
  so generator ...      ; the premises that generate symbol bindings
  to goal ...           ; the desired goals
  if fact ...           ; the extant facts in the knowledge base
  no fact ...           ; the absent facts not in the knowledge base
  do expression ...     ; the expressions to evaluate
  then next             ; the next domain to use in solving the problem.
)
```

Salience

The salience is the relative priority of the rule. It is a 64 bit integer. If omitted the rule will have a default salience of zero (0).

```
(rule Basic-Addition
  salience 100
```

Domain

The domain specifies which ruleset the rule belongs to. If omitted the ruleset will be that of the encompassing domain, or if none it will be `\:nil`.

```
(rule Basic-Addition
  domain Arithmetic
```

So

This section lists the generators for bindings for the rule. A generator is a premise containing the literal **as** followed by a symbol. The literal **as** generates candidate values for the symbol.

```
(rule Basic-Addition
  so [Problem as ?p :State as ?state]
    [Process ^ = ?state :Facts as ?facts]
    [Summation ^ as ?sum :Result as ?result]
    [Addend ^ as ?add :Value as ?value :Status = pending]
```

To

A **gap** is a spatio-temporal distance between an initial state of facts and a desired state of goals. A **plan** or set of actions is needed to specify the solution that bridges the gap between the states

xxx

If

A **gap** is a spatio-temporal distance between an initial state of facts and a desired state of goals. A **plan** or set of actions is needed to specify the solution that bridges the gap between the states

xxx

No

A **gap** is a spatio-temporal distance between an initial state of facts and a desired state of goals. A **plan** or set of actions is needed to specify the solution that bridges the gap between the states

xxx

Do

A **gap** is a spatio-temporal distance between an initial state of facts and a desired state of goals. A **plan** or set of actions is needed to specify the solution that bridges the gap between the states

xxxx

Then

A **gap** is a spatio-temporal distance between an initial state of facts and a desired state of goals. A **plan** or set of actions is needed to specify the solution that bridges the gap between the states

xxxx

Problems

A **problem** is a gap, obstacle, or novelty, or process. A problem state describes the situation to be addressed. A state may be a gap, lacuna, obstacle, novelty, or process specified using premises. The rules implementation should address how to form a solution for the situation.

```
(problem
  name      name
  memo     description
  domain    realm
  start     time
  until     time
  state     state
  status    open | idle | busy | wait | fail | done
  result    nil
)
```

Problem States

A problem state, or problem context defines the initial situation for the problem. States can be created using the assert or assume functions.

```
(relation State) ; Marker interface
```

In problem solving, there are several conventional states: gaps, lacunae, obstacles, novelties, and processes.

Gaps

A **gap** is a spatio-temporal distance between an initial state of facts and a desired state of goals. A **plan** or set of actions is needed to specify the solution that bridges the gap between the states

```
(relation Gap uses State :Facts :Goals)

(assume Gap
  :Facts premise ... ; initial facts
  :Goals premise ... ; desired goals
)
```

Lacunas

A **lacuna** is a logical distance between a set of premises and a set of conclusions (called theorems). A **proof** is needed to fill the lacuna.

```
(relation Lacuna uses State :Premises :Theorems)

(assume Lacuna
  :Premises premise ... ; initial statements
  :Theorems premise ... ; desired statements
)
```

Obstacles

An **obstacle** is a prediction failure between an expected state of affairs and an actual state of affairs. An **explanation** is required which identifies the obstacle and its cause.

```
(relation Obstacle uses State :Wanted :Actual)

(assume Obstacle
  :Wanted premise ... ; expected outcome
  :Actual premise ... ; actual outcome
)
```

Novelties

A **novelty** is a poorly understood condition which needs to be **explored** through conjecture, experimentation, and variation. Building a theory containing actions and inferences about the novelty bridges the gap in understanding.

```
(relation Novelty uses State :Facts :Theory)

(new Novelty
  :Facts premise ... ; initial facts
)
```

Processes

A **process** is a set of facts and phases through which the facts must undergo.

```
(relation Process uses State :Phases :Doing :Facts)

(new Process
  :Phases { phase ... } ; parts of the process
  :Doing phase           ; the currently executing phase
  :Facts premise ...     ; initial facts
)
```

Functions

Functions are defined using the function intrinsic. Function names may not be preceded by a question mark. Functions may have no parameter list, an empty parameter list, or a list containing, required, optional, remainder or keyword parameters.

```
> (function factorial {?n}           ; named function with a parameter list
  (reduce (range 1 to ?n) *)
)

.: factorial

> (factorial 4)

.: 24

> (function prime-p {?n}      ; named function with required parameter
  (if (< ?n 2)
    (return false)
  else
    (step ?i from 1 to (++ (ceiling (root ?n)))
      (if (and (> ?i 1) (< ?i ?n) (zero-p (% ?n ?i)))
        (return false))))
  true
)

.: prime-p

> (prime-p 2)

.: true

> (prime-p 6)

.: false

> (function sum {& ?args}      ; named function with a remaining parameter symbol
  returns number
  (apply + ?args)
)

.: sum

> (sum 1 2 3 4)

.: 10
```

```

> (function addTwoToFive {?1 ?2 + ?3 = 0 ?4 = 0 ?5 = 0} ; named function with optional
  (apply + (& ?1 ?2 ?3 ?4 ?5)) ; defaulted parameters
)

.: addTwoToFive

> (addTwoToFive 5 5)

.: 10

> (addTwoToFive 5 6 7 8)

.: 26

> (function addAll {?1 ?2 + ?3 = 0 ?4 = 0 & ?rest} ; named function with optional and
  (apply + (& ?1 ?2 ?3 ?4 ?rest)) ; remainder parameters
)

.: addAll

> (addAll 1 2 3 4 5 6 7 8 9 10)

.: 55

> (function forgetIt {?relation} ; named function
  (confirm (relation-p ?prototype) since ($ ?relation must be a relation.))
  (for ?x in (with ?relation) (old ?x))
  forgotten
)

.: forgetIt

> (with Statement)

.: {[Statement ^ Statement_1 :All people :Are mortal]
  [Statement ^ Statement_2 :All philosophers :Are people]
  [Statement ^ Statement_3 :All philosophers :Are mortal]}

> (forgetIt Statement)

.: forgotten

> (with Statement)

.: {}

> (function {& ?args} ; auto-named function with
  (apply * ?args) ; remaining parameter symbol
)

.: fn-1

```

```

> (fn-1 1 2 3 4)

.: 24

> (function transitivity ; no parameters
  (with [Statement :All as ?a :Are as ?b]
    [Statement :All = ?b :Are as ?c]
    do
      (knew Statement :All ?a :Are ?c))
  )

.: transitivity

> (macro myFor {?x {in ?y} & ?body} ; a required keyword parameter
  (eval (` (apply for (& , ?x in , ?y ,_ ?body))))
  )

.: myFor

> (myFor ?name in {Jane John Sally} (tell user ($ ?name \n)))

Jane
John
Sally

.: told

> (macro myStep {?v {from ?a} {to ?z} + {by ?i = 1} & ?body} ; optional keyword
  (eval (` (step , ?v from , ?a to , ?z by , ?i ,_ ?body))))
  )

.: myStep

> (myStep ?x from 1 to 5 by 2 (tell user ($ ?x \n)))

1
3
5

.: told

> (myStep ?x from 1 to 5 (tell user ($ ?x \n)))

1
2
3
4
5

.: told

```

Modules

Modules are literals that encapsulate function definitions. In practice, a function has a module preceding the function name, (as in “module.function”). Functions having a module in their literal are “qualified”. Functions without modules are “unqualified”. The module intrinsic creates modules. The extend intrinsic allows more function definitions to be added to a module. The using intrinsic sets or returns the current module. The require intrinsic allows functions to be visible between modules.

```
> (modules)                                ; view the modules collection
.: {Apex Base IO KB Math Tasks User}

> (use User)                               ; work in a specific module
.: User

> (module Arithmetic           ; define a new module Arithmetic
  (function sum {& ?args}
    (apply + ?args)
  )
)
.: Arithmetic

> (modules)
.: {Arithmetic Apex Base IO KB Math Tasks User}

> (sum 1 2 3 4)                         ; is sum visible from user module?
.: [Failure :Name ArgumentValue :Text "The function sum isn't found"]

> (Arithmetic.sum 1 2 3 4)                ; is fully qualified function visible?
.: 10

> (require Arithmetic as a)   ; user module requires definitions
.: Arithmetic

> (sum 1 2 3 4) ; now sum is visible
.: 10

> (a.sum 1 2 3 4) ; now a.sum is visible
.: 10

> (Arithmetic.sum 1 2 3 4) ; still accessible as Arithmetic.sum
.: 10
```

```
> (extend Arithmitic  
  (function product {& ?args}          ; add a new function  
    (apply * ?args)  
  )  
)  
. : Arithmetic
```

Delegates

Delegates are functions that are passed as parameters to other functions.

```
> (relation Product  :Name  :Price )
.: Product

> (relation Cart   :Items  {}  !total  Cart_Total )
.: Cart

> (function Cart_Total {?me ?SubTotDelegate ?TotalDelegate ?DiscountDelegate}
  (ensure Cart ?me Function ?SubTotDelegate Function ?TotalDelegate
    Function ?DiscountDelegate)
  (let ?subTotal (sum (map (:Items ?me) :Price)))
  (?SubTotDelegate ?subTotal)
  (?DiscountDelegate ($ Discounts applied.\n))
  (?TotalDelegate (:Items ?me) ?subTotal))

.: Cart_Total

> (relation Program  :Cart  !onNew (given {?me} (put ?me :Cart (new Cart))))
.: Program

> (function main [& ?args]
  (let ?me (new Program))

  (enq (:Cart ?me) :Items (new Product :Name Cheese :Price 5.50))
  (enq (:Cart ?me) :Items (new Product :Name Bread :Price 7.00))
  (enq (:Cart ?me) :Items (new Product :Name Milk :Price 4.65))
  (enq (:Cart ?me) :Items (new Product :Name Eggs :Price 5.95))

  (tell user ($ Your total is
              (!total (:Cart ?me)
                  (given {?sub} (tell user ($ The subtotal is ?sub \n)))
                  (given {?products ?sub}
                      (if (> (# ?products) 3)
                          (* ?sub 0.90)
                          else ?sub))
                  (given {?msg} (tell user ?msg)))
              \n \n)))

.: main

> (main)

The subtotal is 23.10
Discounts Applied.
Your total is 20.7

.: told
```

Poker Hand Analyzer

This function tells what kind of poker hand each player has.

```
> (enumeration Suits
  Clubs  Diamonds  Hearts  Spades
)

.: Enumeration-1

> (enumeration Ranks
  Ace   Two   Three  Four   Five   Six   Seven
  Eight Nine  Ten    Jack   Queen  King  Joker
)

.: Enumeration-2

> (relation Card
  :Rank
  :Suit
  :Dealt
)

.: Card

> (relation Player
  :Order
  :Cards {}
  :Hand Unknown
)

.: Player

> (function setup
  (repeat 2 ; use 2 decks
    (for ?suit ?i per (entries (enum Suits))
      (for ?rank ?j per (entries (enum Ranks))
        (if (/= ?rank Joker) (new Card :Rank ?rank :Suit ?suit))))))
  (let ?order 0)
  (repeat (random 2 to 8) (new Player :Order (--> ++ ?order)))
)

.: setup

> (function deal
  (with Card ^ as ?card do (bind ?card :Dealt false))
  (with Player ^ as ?player do (bind ?player :Cards {} :Hand unknown))
  (let ?players (which Player sort :Order < )
    ?cards (shuffle (which Card)))
  (repeat (config "/Game/Poker/CardsPerPlayer")
    (for ?player in ?players
      (so {?card (pop ?cards)}
        (enq ?player :Cards ?card)
        (put ?card :Dealt true)))
    )
  dealt
)

.: deal
```

```

> (enumeration Hands
  Nothing
  OnePair
  TwoPairs
  ThreeOfAKind
  Straight
  Flush
  FullHouse
  FourOfAKind
  StraightFlush
)

.Enumeration-3

> (function analyze
  (with [Player ^ as ?p :Cards as ?cards :Hand = Unknown]
    [Card ^ as ?c1 (contains ?cards ?c1) :Suit as ?s :Rank as ?r]
    (let ?n (position Ranks ?r))
    [Card ^ as ?c2 (contains ?cards ?c2) :Suit = ?s :Rank = (@ Rank (+ ?n 1))]
    [Card ^ as ?c3 (contains ?cards ?c3) :Suit = ?s :Rank = (@ Rank (+ ?n 2))]
    [Card ^ as ?c4 (contains ?cards ?c4) :Suit = ?s :Rank = (@ Rank (+ ?n 3))]
    [Card ^ as ?c5 (contains ?cards ?c5) :Suit = ?s :Rank = (@ Rank (+ ?n 4))]
    (/= ?c1 ?c2 ?c3 ?c4 ?c5)
  do
    (put ?p :Hand StraightFlush))

  (with [Player ^ as ?p :Cards as ?cards :Hand = Unknown]
    [Card ^ as ?c1 (contains ?cards ?c1) :Rank = ?r]
    [Card ^ as ?c2 (contains ?cards ?c2) :Rank = ?r]
    [Card ^ as ?c3 (contains ?cards ?c3) :Rank = ?r]
    [Card ^ as ?c4 (contains ?cards ?c4) :Rank = ?r]
    (/= ?c1 ?c2 ?c3 ?c4)
  do
    (put ?p :Hand FourOfAKind))

  (with [Player ^ as ?p :Cards as ?cards :Hand = Unknown]
    [Card ^ as ?c1 (contains ?cards ?c1) :Rank as ?m]
    [Card ^ as ?c2 (contains ?cards ?c2) :Rank = ?m]
    [Card ^ as ?c3 (contains ?cards ?c3) :Rank = ?m]
    [Card ^ as ?c4 (contains ?cards ?c4) :Rank as ?n (/= ?m ?n)]
    [Card ^ as ?c5 (contains ?cards ?c5) :Rank = ?n]
    (/= ?c1 ?c2 ?c3 ?c4 ?c5)
  do
    (put ?p :Hand FullHouse))

  (with [Player ^ as ?p :Cards as ?cards :Hand = Unknown]
    [Card ^ as ?c1 (contains ?cards ?c1) :Suit as ?s]
    [Card ^ as ?c2 (contains ?cards ?c2) :Suit = ?s]
    [Card ^ as ?c3 (contains ?cards ?c3) :Suit = ?s]
    [Card ^ as ?c4 (contains ?cards ?c4) :Suit = ?s]
    [Card ^ as ?c5 (contains ?cards ?c5) :Suit = ?s]
    (/= ?c1 ?c2 ?c3 ?c4 ?c5)
  do
    (put ?p :Hand Flush))

  (with [Player ^ as ?p Cards ?cards :Hand = Unknown]
    [Card ^ as ?c1 (contains ?cards ?c1) :Rank as ?r]
    (let ?n (position Ranks ?r))
    [Card ^ as ?c2 (contains ?cards ?c2) :Rank = (@ Ranks (+ ?n 1))]
    [Card ^ as ?c3 (contains ?cards ?c3) :Rank = (@ Ranks (+ ?n 2))]
    [Card ^ as ?c4 (contains ?cards ?c4) :Rank = (@ Ranks (+ ?n 3))]
    [Card ^ as ?c5 (contains ?cards ?c5) :Rank = (@ Ranks (+ ?n 4))]
```

```

( /= ?c1 ?c2 ?c3 ?c4 ?c5)
do
  (put ?p :Hand Straight))

(with [Player ^ as ?p :Cards as ?cards :Hand = Unknown]
  [Card ^ as ?c1 (contains ?cards ?c1) :Rank as ?n]
  [Card ^ as ?c2 (contains ?cards ?c2) :Rank = ?n]
  [Card ^ as ?c3 (contains ?cards ?c3) :Rank = ?n]
  (/= ?c1 ?c2 ?c3)
do
  (put ?p :Hand ThreeOfAKind))

(with [Player ^ as ?p :Cards as ?cards :Hand = Unknown]
  [Card ^ as ?c1 (contains ?cards ?c1) :Rank as ?m]
  [Card ^ as ?c2 (contains ?cards ?c2) :Rank = ?m]
  [Card ^ as ?c3 (contains ?cards ?c3) :Rank as ?n (/= ?m ?n)]
  [Card ^ as ?c4 (contains ?cards ?c4) :Rank = ?n]
  (/= ?c1 ?c2 ?c3 ?c4)
do
  (put ?p :Hand TwoPairs))

(with [Player ^ as ?p :Cards as ?cards :Hand = Unknown]
  [Card ^ as ?c1 (contains ?cards ?c1) :Rank as ?m]
  [Card ^ as ?c2 (contains ?cards ?c2) :Rank = ?m]
  (/= ?c1 ?c2)
do
  (put ?p :Hand OnePair))

(with [Player ^ as ?p :Cards as ?cards :Hand = Unknown]
  do
    (put ?p :Hand Nothing))
ready
)

:: analyze

> (do (setup) (deal) (analyze))

:: ready

```

OPS5 Comparison

The OPS5 programming language was developed by Charles Forgy in the late 1970s. OPS is an acronym for "Official Production System" and it is a rule based programming language primarily used for expert system development. The following is an OPS5 program that implements a model of cars.

OPS5

```
(literalize car
  age          ; new or old
  condition    ; good or junk
)

(p old-not-junk
  (car ^age old ^condition <> good)
--->
  (write (crlf) that is believable))

(p good-not-old
  (car ^age <> old ^condition good)
--->
  (write (crlf) that is possible))

(p new-and-junk
  (car ^age new ^condition junk)
--->
  (write (crlf) There are no new, junk cars))

(copy car ^age new ^condition good)
```

Premise

```
(relation Car
  :Age          ; new or old
  :Condition    ; good or junk
)

(rule  old-not-junk
  if  [Car :Age = old :Condition /= good]
  do  (tell user ($ that is believable \n)))

(rule  good-not-old
  if  [Car :Age /= old :Condition = good]
  do  (tell user ($ that is possible \n)))

(rule  new-and-junk
  if  [Car :Age = new :Condition = junk]
  do  (tell user ($ There are no new, junk cars \n)))

(new Car :Age new :Condition good)
```

CLIPS Comparison

The C Language Integrated Production System, CLIPS, was developed in 1985 by NASA and was based on Charles Forgy's OPS languages and the Automated Reasoning Tool language from Inference Corporation. The example below is derived from <https://en.wikipedia.org/wiki/CLIPS>.

CLIPS

```
(deftemplate trouble
  (slot name)
  (slot status))

(defrule rule1
  (trouble (name ignition_key) (status on))
  (trouble (name engine) (status wont_start))
  (trouble (name headlights) (status don't_work))
  =>
  (confirm (trouble (name battery) (status faulty)))

(deffacts trouble_shooting
  (trouble (name ignition_key) (status on))
  (trouble (name engine) (status wont_start))
  (trouble (name headlights) (status don't_work)))
```

Premise

```
(relation Trouble
  :Name
  :Status
)

(rule rule1
  if  [Trouble :Name = ignitionKey :Status = on]
      [Trouble :Name = engine :Status = WontStart]
      [Trouble :Name = headlights :Status = DontWork]
  do
    (knew Trouble :Name battery :Status faulty)
)

(function troubleshooting
  (new Trouble :Name ignitionKey :Status on)
  (new Trouble :Name engine :Status WontStart)
  (new Trouble :Name headlights :Status DontWork)
)

(troubleshooting)
```

SQL Comparison

Structured Query Language (SQL) is a language for programming relational database management systems (RDMS). SQL was first described by Edgar F. Codd in 1970.

Relationships

In Structured Query Language, entities and the relationships among them are central to the language. Entities (and relationships) are defined by **Tables** which have **Columns**.

SQL

```
CREATE TABLE Region (
    RegionID INTEGER NOT NULL,
    Name VARCHAR(50) NOT NULL
);

CREATE TABLE Country (
    CountryID INTEGER NOT NULL,
    Name VARCHAR(50) NOT NULL
);

CREATE TABLE Customer (
    CustomerID INTEGER NOT NULL AUTO_INCREMENT,
    RegionID INTEGER NOT NULL,
    CountryID INTEGER NOT NULL,
);
;

CREATE TABLE Order (
    OrderID INTEGER NOT NULL AUTO_INCREMENT,
    CustomerID VARCHAR(10)
);
;

CREATE TABLE Category (
    CategoryID INTEGER NOT NULL AUTO_INCREMENT,
    CategoryName VARCHAR(100) NOT NULL,
    Description MEDIUMTEXT
);
;

CREATE TABLE Product (
    ProductID INTEGER NOT NULL AUTO_INCREMENT,
    ProductName VARCHAR(100) NOT NULL,
    CategoryID INTEGER,
    UnitsInStock FLOAT,
    UnitPrice FLOAT
    Discontinued BIT
);
;

CREATE TABLE Employee (
    EmployeeID INTEGER NOT NULL AUTO_INCREMENT,
    EmployeeName VARCHAR(100) NOT NULL,
    ReportsTo INTEGER,
    Sold FLOAT
);
;
```

Premise

```
(relation Region
  :Name
)

(relation Country
  :Name
)

(relation Customer
  :Region
  :Country
)

(relation Order
  :CustomerId
)

(relation Category
  :CategoryId
  :CategoryName
  :Description
)

(relation Product
  :ProductId
  :ProductName
  :CategoryId
  :UnitsInStock
  :UnitPrice
  :Discontinued  false
)

(relation Employee
  :EmployeeId
  :EmployeeName
  :ReportsTo
  :Sold
)
```

Select All

A select all query retrieves all attributes of an entity.

SQL

```
SELECT *
  FROM Category
```

Premise

```
(with Category)
```

Selecting a single column

In SQL a single column is selected by specifying the name of the column in a select query.

SQL

```
SELECT CategoryName
  FROM Category
```

Premise

```
(with [Category as ?c]
  list (:CategoryName ?c))
```

```
(with [Category :CategoryName as ?n]
  list ?n)
```

Selecting multiple columns

In SQL multiple columns are selected by naming each column in a select query.

SQL

```
SELECT CategoryName, Description  
      FROM Category
```

Premise

```
(with [Category :CategoryName as ?n :Description as ?d]  
    list {?n ?d})
```

Selecting a calculated column

A calculated column is one in which a function is applied to one or more existing columns in a table.

SQL

```
SELECT LENGTH(CategoryName)  
      FROM Category
```

Premise

```
(with [Category :CategoryName as ?n]  
    list (# ?n))
```

Selecting distinct values

SQL

```
SELECT DISTINCT LENGTH(CategoryName)
  FROM Category
```

Premise

```
(distinct
  (with [Category :CategoryName as ?n]
    list (# ?n)))
```

Selecting a scalar value

SQL

```
SELECT MAX(LENGTH(CategoryName) )
  FROM Category
```

Premise

```
(max (with [Category :CategoryName as ?n]
      list (# ?n)))
```

Filtering results by Equality

SQL

```
SELECT *
  FROM Product
 WHERE UnitsInStock = 0
```

Premise

```
(which Product
 :UnitsInStock = 0)
```

Filtering results by Inequality

SQL

```
SELECT *
  FROM Product
 WHERE NOT (UnitPrice > 10)
```

Premise

```
(which Product
 :UnitPrice as ?p
 (not (> ?p 10)))
```

```
(which Product
 :UnitPrice <= 10 )
```

Filtering results by a range

SQL

```
SELECT *
  FROM Product
 WHERE UnitPrice BETWEEN 5 AND 9
```

Premise

```
(which Product
 :UnitPrice as ?u
 (<= 5 ?u 9))
```

Multiple filter conditions

SQL

```
SELECT *
  FROM Product
 WHERE Discontinued = 1
   AND UnitsInStock >> 0
```

Premise

```
(which Product
 :Discontinued 1
 :UnitsInStock not 0)
```

Order by value ascending

SQL

```
SELECT *  
      FROM Product  
ORDER BY UnitPrice
```

Premise

```
(which Product  
  sort :UnitPrice < )
```

Order by value descending

SQL

```
SELECT *  
      FROM Product  
ORDER BY UnitPrice DESC
```

Premise

```
(which Product  
  sort :UnitPrice > )
```

Limit number of results

SQL

```
SELECT TOP 5 *
  FROM Product
ORDER BY UnitPrice
```

Premise

```
(top (which Product
      sort :UnitPrice < )
    5)
```

Paged results

SQL

```
WITH part1 AS (
  SELECT *
  FROM Product
  ORDER BY UnitPrice)
SELECT *
FROM part1
WHERE RRN(part1) BETWEEN 5 AND 10
```

Premise

```
(mid (which Product
      sort :UnitPrice < )
    5
    10)
```

Group by value

SQL

```
SELECT TOP 100 UnitPrice
    FROM (SELECT Product.UnitPrice,
                COUNT(*) AS Count
            FROM Product
           GROUP BY Product.UnitPrice)
ORDER BY Count DESC
```

Premise

```
(with (top (with [Product as ?p :UnitPrice as ?u]
               list {:_UnitPrice ?u
                     :Count (with [Product :UnitPrice as ?u] tally)}
                     sort :Count > )
              100) each ?x
      list (:UnitPrice ?x))
```

Inner Join

SQL

```
SELECT p.*
    FROM Product p,
         Category c
   WHERE c.CategoryName = "Beverages" AND
         c.CategoryID = p.CategoryID
```

Premise

```
(with
  [Product ^ as ?p :CategoryName = "Beverages" :CategoryId as ?id]
  [Category :Categoryid = ?id]
  list (premise ?p))
```

Left Join

SQL

```
SELECT c.CustomerID, COUNT(o.OrderID)
  FROM Customer c
LEFT JOIN Order o
    ON o.CustomerID = c.CustomerID
 GROUP BY c.CustomerID
```

Premise

```
(with
  [Customer :CustomerId as ?i]
  list {:_CustomerId ?i :Count (with [Order :CustomerId = ?i] tally) }
```

```
(with
  [Customer :CustomerId as ?i]
  [Order :OrderId as ?j]
  (left ?i ?j)
  group {?i (when (any ?i) 1 0)}
  by 1 into ?group
  list {:_CustomerId (@ ?group 1)
        :Count (summation ?pair in (rest ?g) (@ ?pair 2)) } )
```

Concatenate

SQL

```
SELECT customer.CompanyName
  FROM Customer AS customer
 WHERE customer.CompanyName LIKE "A%"
UNION ALL
SELECT customer.CompanyName
  FROM Customer AS customer
 WHERE customer.CompanyName LIKE "E%"
```

Premise

```
(union
 (with [Customer :CompanyName as ?n] (like ?n "A%") list ?n)
 (with [Customer :CompanyName as ?n] (like ?n "E%") list ?n))
```

Create, Update and Delete

SQL

```
INSERT INTO Category (CategoryName, Description)
    VALUES ("Merchandising", "Cool products");

INSERT INTO Product (ProductName, CategoryID)
    SELECT TOP (1) "Red Jacket", CategoryID
        FROM Category
    WHERE CategoryName = "Merchandising";

UPDATE Product
    SET Product.ProductName = "Green Jacket"
    WHERE Product.ProductName = "Red Jacket";

DELETE FROM Product
    WHERE Product.ProductName = "Green Jacket";

DELETE FROM Category
    WHERE Category.CategoryName = "Merchandising";
```

Premise

```
(new Category :CategoryName "Merchandising"
    :Description "Cool Products")

(new Product :ProductName "Red Jacket"
    :CategoryId (each Category ^ as ?c
                    :CategoryName "Merchandising"
                    give (:CategoryId ?c)))

(with [Product ^ as ?p :ProductName = "Red Jacket"]
    do (put ?p :ProductName "Green Jacket"))

(with [Product ^ as ?p :ProductName = "Green Jacket"]
    do (old ?p))

(with [Category ^ as ?c :CategoryName = "Merchandising"]
    do (old ?c))
```

Recursive Query

SQL

```
WITH EmployeeHierarchy (EmployeeID,
                        LastName,
                        FirstName,
                        ReportsTo,
                        HierarchyLevel) AS
( SELECT EmployeeID
    , LastName
    , FirstName
    , ReportsTo
    , 1 as HierarchyLevel
  FROM Employees
 WHERE ReportsTo IS NULL

 UNION ALL

 SELECT e.EmployeeID
       , e.LastName
       , e.FirstName
       , e.ReportsTo
       , eh.HierarchyLevel + 1 AS HierarchyLevel
     FROM Employees e
INNER JOIN EmployeeHierarchy eh
      ON e.ReportsTo = eh.EmployeeID
) SELECT *
      FROM EmployeeHierarchy
ORDER BY HierarchyLevel, LastName, FirstName
```

Premise

```
(relation EmployeeHierarchy
 :EmployeeId :LastName :FirstName :ReportsTo :HierarchyLevel )

(with (union
 (with [Employee ^ as ?e :ReportsTo is nothing :EmployeeId as ?i
       :LastName as ?n :FirstName as ?f]
      list (knew EmployeeHierarchy :EmployeeId ?i :LastName ?n
            :FirstName ?f :HierarchyLevel 1))
 (with
   [EmployeeHierarchy ^ as ?eh :EmployeeId as ?i :HierarchyLevel as ?h]
   [Employee ^ as ?e :EmployeeId as ?j :LastName as ?n :FirstName as ?f
     :ReportsTo ?i]
      list (knew EmployeeHierarchy :EmployeeId ?j :LastName ?n
            :FirstName ?f :HierarchyLevel (+ ?h 1))))
 as ?result
 list (premise ?result)
 sort :HierarchyLevel < :LastName < :FirstName < )
```

6. Foundation Modules

The SubThought Premise is divided into several modules which contain built-in (i.e., intrinsic) functions. Each module has a particular area of responsibility. For example, the IO module facilitates communication while the knowledge base module (KB) declares storage and reasoning functions.

Apex

This module contains the global environment.

Base

This module provides control structures and special forms.

IO

This module contains messaging, file, and the console functions.

KB

This module provides access functions to knowledge bases.

Math

This module provides arithmetic, trigonometric, and statistical functions.

Tasks

This module has functions that manage asynchronous and concurrent tasks.

User

This module contains the default environment.

7. Quick Reference

A synopsis of each function follows.

| | | |
|----|--|---|
| 1 | (-- <i>number</i>) | Decrements a number by 1. |
| 2 | (- <i>number ...</i>) | Subtraction. |
| 3 | (@ <i>container place ...</i>) | Retrieves elements from a sequence or assortment. |
| 4 | (# <i>container</i>) | Returns the number of elements. |
| 5 | (\$ <i>value ...</i>) | Creates a new string with intervening spaces. |
| 6 | (\$\$ <i>value ...</i>) | Creates a new string by eliding arguments. |
| 7 | (% <i>command arguments</i>) | Performs an operating system command |
| 8 | (& <i>value ...</i>) | Merges arguments into a new sequence. |
| 9 | (&& <i>value ...</i>) | Merges arguments into a new sequences and removes nils. |
| 10 | (* <i>number number ...</i>) | Multiplication. |
| 11 | (** <i>base exponent</i>) | Exponentiation. |
| 12 | (/ <i>dividend divisor ...</i>) | Division. |
| 13 | (// <i>number degree</i>) | Nth Root. |
| 14 | (/~ <i>value₁ value₂</i>) | Calculates the dissimilarity between two values. |
| 15 | (/= <i>value₁ value₂ ...</i>) | Returns true if any values are not equal. |
| 16 | (~ <i>value₁ value₂</i>) | Calculates the similarity between two values. |
| 17 | (' <i>variant</i>) | Quotes a variant. |
| 18 | (` <i>variant</i>) | Expands a variant by substituting values. |
| 19 | (+ <i>number number ...</i>) | Addition. |
| 20 | (++ <i>number</i>) | Increments a number by 1. |
| 21 | (<-- <i>function symbol value ...</i>) | The value before tying a symbol to a new value. |
| 22 | (< <i>value value ...</i>) | True if each value is less than the next. |
| 23 | (<= <i>value value ...</i>) | True if each value is less or equal to the next. |
| 24 | (<== <i>container function place value ...</i>) | The value before replacement. |
| 25 | (= <i>value value ...</i>) | True if each value is equal to the next. |
| 26 | (>== <i>container function place value ...</i>) | The value after replacement. |
| 27 | (> <i>value value ...</i>) | True if each value is greater than the next. |
| 28 | (>-- <i>function symbol value ...</i>) | The value after tying a symbol to a new value. |
| 29 | (>= <i>value value ...</i>) | True if each value is greater or equal to the next. |
| 30 | (\n <i>quantity</i>) | Returns a string containing one or more new lines. |
| 31 | (\s <i>quantity</i>) | Returns a string containing one or more spaces. |
| 32 | (^ <i>thought</i>) | Returns a SubThought identifier. |
| 33 | (abolish <i>variables environment</i>) | Deletes variables from an environment. |
| 34 | (abort <i>tasks</i>) | Forcibly terminates one or more tasks |
| 35 | (about) | Provides system and version information. |
| 36 | (abs <i>number</i>) | Calculates the absolute value. |
| 37 | (absent <i>url</i>) | True if a file, folder, or url does not exist. |
| 38 | (acosecant <i>number geometry metrum</i>) | Calculates the inverse cosecant. |
| 39 | (acosine <i>number geometry metrum</i>) | Calculates the inverse cosine. |
| 40 | (acotangent <i>number geometry metrum</i>) | Calculates the inverse cotangent. |
| 41 | (actual <i>symbol</i>) | Returns the underlying symbol for a reference. |
| 42 | (add <i>assortment entry ...</i>) | Modifies an assortment by adding entries. |
| 43 | (address <i>url</i>) | Returns the address of a URL web resource. |
| 44 | (agent <i>job url handler delay</i>) | Creates an agent. |
| 45 | (align <i>sequence ordering</i>) | Returns a sorted list using the provided ordering. |
| 46 | (alike <i>value value ...</i>) | True if each value has the same taxon. |
| 47 | (all <i>condition ...</i>) | True if all relevant knowledge satisfies all of the conditions. |
| 48 | (alphabetic-p <i>value</i>) | True if the first position of a string is alphabetic. |
| 49 | (alphanumeric-p <i>value</i>) | True if the first position of a string is whitespace. |

| | | |
|-----|--|---|
| 50 | (and <i>truth truth ...</i>) | Logical conjunction. |
| 51 | (any <i>condition ...</i>) | True if some relevant knowledge satisfies all of the conditions. |
| 52 | (append <i>sequence value ...</i>) | Inserts values at the end of a sequence. |
| 53 | (apply <i>function arguments environment</i>) | Applies a function to a list of arguments. |
| 54 | (arity <i>function kind</i>) | Returns the number of variables in a function's parameter list. |
| 55 | (array <i>dimensions option</i>) | Returns a multi dimensional list. |
| 56 | (asecant <i>number geometry metrum</i>) | Calculates the inverse secant. |
| 57 | (asine <i>number geometry metrum</i>) | Calculates the inverse sine. |
| 58 | (ask <i>who message timeout default</i>) | Sends a message to a recipient and awaits a response. |
| 59 | (assert <i>relation descriptor ...</i>) | Creates a thought containing nested premises using new. |
| 60 | (assume <i>relation descriptor ...</i>) | Creates a thought containing nested premises using knew. |
| 61 | (atangent <i>number geometry metrum</i>) | Calculates the inverse tangent. |
| 62 | (attach <i>knowledge</i>) | Registers a knowledge base. |
| 63 | (average <i>symbol ... binding sequence expression ...</i>) | The arithmetic mean of a sequence. |
| 64 | (avg <i>value ...</i>) | The arithmetic mean. |
| 65 | (await <i>task timeout default</i>) | Returns the result of a task. |
| 66 | (before-p <i>interval₁ interval₂ tolerance</i>) | True if an interval finishes before a second interval. |
| 67 | (beginning <i>interval</i>) | Returns the beginning instant of an interval |
| 68 | (best <i>probe candidates option ...</i>) | Returns the best matching candidates. |
| 69 | (beyond <i>sequence position</i>) | True if a position is outside a sequence. |
| 70 | (big <i>value</i>) | Converts a value to a big number. |
| 71 | (bind <i>container assignment ...</i>) | Assigns variables to values in sequences or assortments. |
| 72 | (bindings <i>pattern</i>) | Returns a list of symbol value pairs for an environment or premise. |
| 73 | (bitwise <i>number op ...</i>) | Performs bitwise operations. |
| 74 | (bound <i>function</i>) | Returns the variables that are bound in a function. |
| 75 | (bound-p <i>symbol</i>) | True if a symbol has a value. |
| 76 | (bracket <i>truth</i>) | Returns 1 if a truth expression is true, 0 otherwise. |
| 77 | (break <i>value</i>) | Terminates a loop. |
| 78 | (busy-p <i>task</i>) | True if a task has not completed. |
| 79 | (but <i>sequence quantity</i>) | Creates a subsequence of all except the last elements. |
| 80 | (bye) | Terminates the interpreter. |
| 81 | (call <i>value ...</i>) | Creates an unevaluated call. |
| 82 | (can <i>expression result error</i>) | Evaluates an expression while suppressing failures. |
| 83 | (cancel <i>tasks option ...</i>) | Cooperatively cancels tasks. |
| 84 | (cancelled-p <i>task</i>) | True if a task has been cancelled. |
| 85 | (canonify <i>variant</i>) | Makes equal values identical. |
| 86 | (capitalize <i>string option</i>) | Capitalizes a string. |
| 87 | (case <i>probe clause ... else-clause</i>) | Branches execution based on a value. |
| 88 | (categorize <i>sequence predicate ...</i>) | Creates a list of equivalence sets. |
| 89 | (cede <i>reference</i>) | Transfers a value between variables. |
| 90 | (ceiling <i>number</i>) | Rounds a number towards positive infinity. |
| 91 | (char <i>unicode</i>) | Converts a Unicode value into a one position string. |
| 92 | (choose <i>sequence selector transform</i>) | Returns the arguments satisfying a function. |
| 93 | (clear <i>container</i>) | Eliminates all entries from an assortment or sequence. |
| 94 | (clip <i>value minimum maximum</i>) | Returns a value within a clipped range. |
| 95 | (clone <i>atom modification ...</i>) | Clones an atom with possible modifications. |
| 96 | (close <i>url</i>) | Closes a file or data url. |
| 97 | (closure <i>scope type name params expression ...</i>) | Creates a function or macro with a defined environment. |
| 98 | (coalesce <i>symbol ... gate expression ...</i>) | Returns a merged sequence. |
| 99 | (collect <i>symbol ... gate expression ...</i>) | Returns a transformed sequence. |
| 100 | (collection <i>element ...</i>) | Returns an unorderd collection of elements. |
| 101 | (combine <i>assortment entry ...</i>) | Creates a new assortment by combining entries. |
| 102 | (common <i>sequences</i>) | Creates a sequence of common elements among subsequences. |
| 103 | (comparable-p <i>value₁ value₂</i>) | Returns true if the values can be compared. |
| 104 | (compare <i>value₁ value₂</i>) | Returns < (less), = (equal), or > (greater). |

| | | |
|-----|---|--|
| 105 | (complete <i>expression ...</i>) | Runs expressions concurrently until completion. |
| 106 | (complex <i>real imaginary</i>) | Converts a value to a complex number. |
| 107 | (compose <i>functions arguments</i>) | Applies functions in reverse order using the arguments. |
| 108 | (conceive <i>knowledge</i>) | Creates a knowledge base. |
| 109 | (concurrent <i>expression ...</i>) | Evaluates expressions asynchronously. |
| 110 | (configuration <i>url association ...</i>) | Creates a configuration file. |
| 111 | (confirm <i>condition since reason</i>) | Tests that a condition is true. |
| 112 | (confute <i>condition since reason</i>) | Tests that a condition is false. |
| 113 | (connect <i>data</i>) | Connects to a data resource. |
| 114 | (constant <i>assignment ...</i>) | Creates a constant. |
| 115 | (contains <i>assortment value</i>) | True if a value is present in an assortment. |
| 116 | (continue <i>result</i>) | Continues to the next iteration of a loop. |
| 117 | (convertible-p <i>value taxon</i>) | True if the value can be converted to the taxon. |
| 118 | (copy <i>sequence count</i>) | Copies a sequence. |
| 119 | (copyright) | Displays copyright information. |
| 120 | (correlate <i>list1 list2</i>) | Finds the correlation coefficient of two lists. |
| 121 | (cosecant <i>number geometry metrum</i>) | Calculates the cosecant. |
| 122 | (cosine <i>number geometry metrum</i>) | Calculates the cosine. |
| 123 | (cotangent <i>number geometry metrum</i>) | Calculates the cotangent. |
| 124 | (count <i>symbol ... binding sequence as counter expression ...</i>) | Counts the iterations and returns the last expression. |
| 125 | (critical <i>locks option ... expression ...</i>) | Serializes evaluations across regions of code. |
| 126 | (cut <i>assortment key ...</i>) | Removes elements from assortments by key. |
| 127 | (data <i>option ...</i>) | Creates a data url. |
| 128 | (date <i>yr mth day hrs mins secs zone zmins</i>) | Creates a new date or returns the current date. |
| 129 | (declared-p <i>symbol environment</i>) | True if a symbol exists in an environment. |
| 130 | (decode <i>source format</i>) | Decodes a string. |
| 131 | (default <i>value ...</i>) | Returns the first non-nil value. |
| 132 | (definitions <i>environment</i>) | Retrieves literal value pairs in an environment. |
| 133 | (defunct <i>function</i>) | Undefines a function in a module. |
| 134 | (degrees <i>radians</i>) | Converts radians into degrees. |
| 135 | (delete <i>sequence value ...</i>) | Modifies sequence by deleting values. |
| 136 | (dependencies <i>module</i>) | Returns a module's dependencies. |
| 137 | (deq <i>container place option</i>) | Removes an element from an embedded sequence. |
| 138 | (describe <i>literal</i>) | Returns descriptions of a literal. |
| 139 | (detach <i>knowledge</i>) | Unregisters a knowledge base. |
| 140 | (difference <i>sequence1 sequence2 operation</i>) | Returns the difference of two sequences. |
| 141 | (did <i>expression</i>) | Evaluates an expression and suppresses failures. |
| 142 | (digit <i>number position</i>) | Returns the digit in the specified position of a number. |
| 143 | (digit-p <i>value</i>) | True if the first position of a string is a digit. |
| 144 | (digits <i>number</i>) | Returns the digits comprising a number. |
| 145 | (dimensions <i>sequence</i>) | Returns the lengths of each dimension in a sequence. |
| 146 | (discard <i>module</i>) | Discards a module. |
| 147 | (disjoint-p <i>sequence sequence ...</i>) | True if no elements are common among sequences. |
| 148 | (distinct <i>sequence</i>) | Creates a new sequence without duplicate elements. |
| 149 | (distribute <i>sequence function result</i>) | Applies a function to a sequence in parallel. |
| 150 | (divide <i>dividend divisor default</i>) | Performs safe division. |
| 151 | (divisible-p <i>dividend divisor</i>) | True if the divisor evenly divides the dividend. |
| 152 | (do <i>expression ... resume tag resumption ... finally cleanup ...</i>) | Evaluates expressions and handles escapes. |
| 153 | (domain <i>name element ...</i>) | Creates a package containing relations and rules. |
| 154 | (domains) | Returns a list of defined domains. |
| 155 | (done) | Terminates a generator. |
| 156 | (drop <i>index</i>) | Removes a relation index. |
| 157 | (duplicate <i>source target option</i>) | Duplicates a file or contents of a folder. |
| 158 | (during-p <i>interval₁ interval₂ tolerance</i>) | True if an interval occurs within a second interval. |

| | | |
|-----|--|---|
| 159 | (dynamic <i>assignments expression ...</i>) | Creates a dynamic environment for variables. |
| 160 | (e) | Returns Euler's number, 2.718281828459045. |
| 161 | (each <i>symbol ... binding reversal sequence ... premises option ... action</i>) | Combines for and with to iterate over sequences to match patterns against the knowledge base. |
| 162 | (encode <i>source format</i>) | Encodes a string. |
| 163 | (enq <i>container place option</i>) | Inserts an element into an embedded sequence. |
| 164 | (ensure <i>check ...</i>) | Performs type checking. |
| 165 | (entries <i>assortment</i>) | Retrieves key value pairs for an assortment. |
| 166 | (enum <i>name</i>) | Retrieves enumerated assortment. |
| 167 | (enumeration <i>name element ...</i>) | Defines an enumerated assortment. |
| 168 | (enumerations) | Returns a list of all enumerations. |
| 169 | (environment <i>parent entry ...</i>) | Creates a new context for variables and functions. |
| 170 | (epoch <i>time</i>) | Returns a Unix epoch or the current epoch. |
| 171 | (eradicate <i>knowledge</i>) | Deletes a knowledge base. |
| 172 | (erase <i>file option ...</i>) | Deletes files or folders. |
| 173 | (escape <i>tag</i>) | Escapes a do special form |
| 174 | (eval <i>expression environment</i>) | Evaluates an expression. |
| 175 | (even-p <i>number</i>) | True if the number is even. |
| 176 | (every <i>symbol ... binding sequence expression ... test</i>) | True if a predicate is true for every element in a sequence. |
| 177 | (exactly <i>quantity of sequence clause within margin</i>) | True if a number of elements satisfy a clause. |
| 178 | (exchange <i>sequence substitution ...</i>) | Creates a new sequence with values swapped. |
| 179 | (excludes <i>sequence element option ...</i>) | True if a sequence does not contain an element. |
| 180 | (exists-p <i>pattern</i>) | True if a pattern is in the knowledge base. |
| 181 | (exit <i>value</i>) | Explicitly ends a task using a return value. |
| 182 | (exponential <i>number significand</i>) | Calculates the base ten exponent of a number. |
| 183 | (expression <i>value ...</i>) | Creates an expression. |
| 184 | (extend <i>module expression ...</i>) | Adds new definitions to a module. |
| 185 | (facility <i>name parameters expression ...</i>) | Creates a private function in a module. |
| 186 | (few <i>symbol ... binding sequence expression ... test</i>) | True if a predicate is true for less than half the elements in a sequence. |
| 187 | (file <i>option ...</i>) | Opens or creates a file. |
| 188 | (files <i>option ...</i>) | Returns a list of file names. |
| 189 | (fill <i>symbol from start to end by increment with expression</i>) | Fills a list with the result of an expression. |
| 190 | (filter <i>sequence predicate</i>) | Returns a sequence of elements that satisfy a predicate. |
| 191 | (find <i>features candidates option ...</i>) | Returns the best matching candidates. |
| 192 | (finishes-p <i>interval₁ interval₂ tolerance</i>) | True if two intervals finish together. |
| 193 | (finishing <i>interval</i>) | Returns the finishing instant of an interval. |
| 194 | (fix <i>declaration ...</i>) | Adds variables to the current environment returns true. |
| 195 | (float <i>value</i>) | Converts a value to a floating number. |
| 196 | (floor <i>number</i>) | Rounds a number towards negative infinity. |
| 197 | (fold <i>symbol ... in sequence into expression ...</i>) | Transforms a sequence into a value. |
| 198 | (folder <i>option ...</i>) | Creates or locates a folder in the file system. |
| 199 | (folders <i>option ...</i>) | Returns a list of sub folders. |
| 200 | (for <i>symbol ... binding reversal sequence ... expression ...</i>) | Iterates over the elements in a sequence. |
| 201 | (forever <i>expression ...</i>) | Repeatedly evaluates expressions. |
| 202 | (forgo <i>dependency module</i>) | Removes a dependent module. |
| 203 | (format <i>template value ...</i>) | Formats a string. |
| 204 | (fractional <i>number</i>) | Returns the fractional portion of a number. |
| 205 | (free <i>resources</i>) | Releases resources. |
| 206 | (full <i>first second</i>) | True if values are equal or either value is nil. |

| | | |
|-----|--|--|
| 207 | (function scope name params returning expression ...) | Creates a public function in a module. |
| 208 | (functions module) | Returns a list of functions defined in a module. |
| 209 | (gather symbol ... binding sequence expression ... test) | Returns a sequence of elements that satisfy a predicate. |
| 210 | (generator name parameters expression ...) | Creates a series generator. |
| 211 | (get container key ...) | Retrieves a value using one or more keys. |
| 212 | (given { parameter ... } expression ...) | Creates an anonymous function. |
| 213 | (global assignment ...) | Creates global variables. |
| 214 | (go function argument ...) | Transfers control to a function. |
| 215 | (grok what) | Evaluates expressions in a url or file. |
| 216 | (group list by key ... into symbol expression ...) | Combines sublists by position or key. |
| 217 | (has assortment key) | True if a key is present in an assortment. |
| 219 | (hash value option ...) | Computes a hash code. |
| 220 | (help function) | Describes a function. |
| 221 | (hyperlink option ...) | Creates a hyperlink. |
| 223 | (id resource) | Returns a resource number. |
| 224 | (identical-p value value ...) | True if all values occupy the same memory location. |
| 225 | (identity value) | Returns the value itself. |
| 226 | (idle duration) | Pauses for a specified interval. |
| 227 | (if condition expression ... or-clause ... else-clause) | Branched conditional evaluation. |
| 228 | (imaginary value) | Converts a value to an imaginary number. |
| 229 | (in value container option ...) | True if a value is in a sequence or assortment. |
| 230 | (includes sequence element option ...) | True if a value is in a sequence. |
| 231 | (index relation slot ...) | Creates an index on slots of a relation. |
| 232 | (indices relation) | Returns a list of indices for a relation. |
| 233 | (infer domain) | Applies rules in all domains or a specific domain. |
| 234 | (infix sequence delimiter) | Creates a new sequence with interposed delimiters. |
| 235 | (inside sequence position) | Creates an expression from a sequence or assortment. |
| 236 | (insert sequence position value ...) | Creates a new sequence by inserting values. |
| 237 | (insert sequence value option ...) | Inserts a value into a sorted sequence. |
| 238 | (instantiate expression) | Creates an expression with premises in place of thoughts. |
| 239 | (integer value) | Converts a value to an integer. |
| 240 | (interior value) | Creates an expression from a sequence or assortment. |
| 241 | (interleave sequence sequence ...) | Merges arguments into a new sequence. |
| 242 | (intersection sequence sequence ...) | Creates a sequence of common elements. |
| 243 | (intersects-p sequence sequence ...) | True if any elements are common among sequences. |
| 244 | (interval start finish) | Creates a time interval. |
| 245 | (into relation thoughts) | Creates new thoughts based on existing thoughts. |
| 246 | (invoke call environment) | Invokes a call. |
| 247 | (is value predicate) | True if the value is true or if the applied predicate returns true. |
| 248 | (junction scope name params expression ...) | Creates a public function where arguments are evaluated in parallel. |
| 249 | (key assortment value) | Finds the key for a value in an assortment. |
| 250 | (keys assortment) | Creates a list of keys for an assortment. |
| 251 | (keywords) | Creates the list of Premise keywords. |
| 252 | (knew relation criterion ...) | Finds or creates a thought. |
| 253 | (knowledge option ...) | Finds or creates a knowledge base. |
| 254 | (known pattern ... option ...) | Finds or creates thoughts. |
| 255 | (lacks assortment key) | True if a key is absent from an assortment. |
| 256 | (last sequence count) | Creates a subsequence of the last elements. |
| 257 | (left first second) | True if values are equal or the second value is nil. |
| 258 | (let assignment ...) | Adds variables to the current environment returning true. |
| 259 | (lexemes string) | Creates an uppercase string with spaces between words. |
| 260 | (lexicon association ...) | Creates a lexicon. |

| | | |
|-----|---|---|
| 261 | (lexicons) | Creates a list of all lexicons. |
| 262 | (license) | Prints the software license. |
| 263 | (like sequence pattern) | Compares a pattern to a sequence. |
| 264 | (list value ...) | Creates a list. |
| 265 | (literal value) | Creates a literal. |
| 266 | (local assignments expression ...) | Creates a task level scope. |
| 267 | (location symbol) | Sets a symbol to the current location. |
| 268 | (log number base) | Logarithm. |
| 269 | (long value) | Converts a value to a long number. |
| 270 | (loop expression ... gate) | Repeatedly evaluates expressions. |
| 271 | (lowercase string) | Converts a string to lower case. |
| 272 | (macro name parameters expression ...) | Creates a public macro in a module. |
| 273 | (macros module) | Creates a list of macros defined in a module. |
| 274 | (make prototype term ...) | Creates a record by creating a relationship if it does not exist. |
| 275 | (map sequence ... function) | Applies a function to elements across sequences. |
| 276 | (match problem) | Matches rules in all open problems or a specific problem. |
| 277 | (max value ...) | Finds the maximum element. |
| 278 | (maximum symbol ... binding sequence expression ...) | Finds the maximum element. |
| 279 | (may expression) | Evaluates an expression while suppressing failures. |
| 280 | (median value ...) | Finds the middle value of a sequence. |
| 281 | (meets-p interval ₁ interval ₂ tolerance) | True if an interval finishes when a second interval starts. |
| 282 | (meron value) | Returns the meronomic prototype of a value. |
| 283 | (meron-p value meron) | True if the value is a meron or is of the specific meronomic type. |
| 284 | (meronymy value) | Returns the meronomic prototypes of a value. |
| 285 | (method scope name params expression ...) | Creates a public method in a prototype. |
| 286 | (mid sequence start stop skip) | Copies a subsequence of a sequence. |
| 287 | (min value ...) | Finds the minimum element. |
| 288 | (minimum symbol ... binding sequence expression ...) | Finds the minimum element. |
| 289 | (missing assortment value) | True if a value is absent from an assortment. |
| 290 | (mod dividend modulus) | Finds the remainder after division. |
| 291 | (modular module expression ...) | Evaluates expressions in module's scope. |
| 292 | (module name definition ...) | Creates a module. |
| 293 | (modules) | Creates a list of known modules. |
| 294 | (moment units secs mins hours days year) | Creates a moment or returns the current moment. |
| 295 | (more-p sequence) | True if a container has more than zero elements. |
| 296 | (morph arguments functions) | Applies functions in succession using the arguments. |
| 297 | (most symbol ... binding sequence expression ... test) | True if the predicate is true for more than half of the elements in a sequence. |
| 298 | (move source destination) | Moves or renames one or more files. |
| 299 | (my) | Returns the current cell resource. |
| 300 | (nall condition ...) | True if not all relevant knowledge satisfies all of the conditions. |
| 301 | (nand truth truth ...) | Negated logical conjunction. |
| 302 | (negative-p number) | True if a number is negative. |
| 303 | (nevery symbol ... binding sequence expression ... test) | True if the predicate is false for any element in the sequence. |
| 304 | (new prototype term ...) | Creates a record. |
| 305 | (next series) | Advances a series to the next element. |
| 306 | (ngrams string length) | Creates a list of substrings. |
| 307 | (nil-p value) | True if a value is nil. |
| 308 | (nix prototype) | Deletes a prototype. |
| 309 | (no condition ...) | True if no relevant knowledge satisfies all of the conditions. |
| 310 | (none symbol ... binding sequence expression ... test) | True if the predicate is false for all sequence elements. |
| 311 | (nor truth truth ...) | Negated logical disjunction. |

| | | |
|-----|---|--|
| 312 | (not <i>value predicate</i>) | True if a value is false, or if the applied predicate returns false. |
| 313 | (nothing) | Returns nothing. |
| 314 | (null-p <i>value</i>) | True if a value is nothing, nil, or void. |
| 315 | (odd-p <i>number</i>) | True if a number is odd. |
| 316 | (old <i>thought</i>) | Deletes a thought. |
| 317 | (omit <i>sequence value ...</i>) | Creates a new sequence with values removed. |
| 318 | (on <i>condition true-case false-case</i>) | Branched conditional evaluation. |
| 319 | (only <i>variables expression ...</i>) | Creates a restricted scope. |
| 320 | (open <i>url</i>) | Opens a file or data url. |
| 321 | (or <i>truth truth ...</i>) | Logical disjunction. |
| 322 | (out <i>value sequence option ...</i>) | True if a value is absent from a sequence. |
| 323 | (overlaps-p <i>interval₁ interval₂ tolerance</i>) | True if an interval finishes after a second interval starts. |
| 324 | (pad <i>sequence length value side</i>) | Creates a new padded sequence. |
| 325 | (partition <i>sequence pivot comparer</i>) | Creates a list of equivalence sets for a pivot element. |
| 326 | (path <i>folder separator file</i>) | Concatenates a folder and file name. |
| 327 | (pattern <i>expression ...</i>) | Creates a pattern containing the supplied expressions. |
| 328 | (perform <i>environment method instance ...</i>) | Invokes a method within an environment. |
| 329 | (pi) | Returns 3.141592653589793. |
| 330 | (pick <i>sequence quantity</i>) | Creates a list of randomly selected elements. |
| 331 | (pipe <i>option ...</i>) | Creates a pipe. |
| 332 | (pop <i>sequence position</i>) | Returns the element removed from a sequence. |
| 333 | (posit <i>module url</i>) | Writes module expressions to a url. |
| 334 | (position <i>sequence element option ...</i>) | Finds the position of an element or subsequence in a sequence. |
| 335 | (positions <i>sequence</i>) | Returns positions of an element in a sequence or position value pairs. |
| 336 | (positive-p <i>number</i>) | True if a number is greater than zero. |
| 337 | (premise <i>value</i>) | Creates a premise representation of a value. |
| 338 | (probability <i>events A operation B</i>) | Calculates the probability over a series. |
| 339 | (problem <i>name memo description domain realm start time until time state state status</i>) | Creates a problem to be solved by the rules engine. |
| 340 | (problems) | Returns a list of problems. |
| 341 | (procedure <i>scope name params expression ...</i>) | Creates a public function that returns %Nothing within a module. |
| 342 | (proceed <i>location set assignment ...</i>) | Transfers control to a location. |
| 343 | (product <i>symbol ... binding sequence expression ...</i>) | Multiplies an expression over a range of values. |
| 344 | (punctuation-p <i>string</i>) | True if the first position of a string is punctuation. |
| 345 | (push <i>sequence value position</i>) | Modifies a sequence by inserting a value. |
| 346 | (put <i>container entry ...</i>) | Updates values in an assortment or sequence. |
| 347 | (qualified <i>module function</i>) | Prepends a module to a function name. |
| 348 | (qualifiers <i>identifier</i>) | Creates a list of modules associated with an identifier. |
| 349 | (quantify <i>sequence predicate option ...</i>) | Returns a quantifier for elements satisfying a test. |
| 350 | (quantity <i>symbol ... binding sequence expression ... test</i>) | Returns the number of elements satisfying a test. |
| 351 | (radians <i>degrees</i>) | Converts degrees to radians. |
| 352 | (radix <i>number base</i>) | Converts a number to a base. |
| 353 | (random <i>lower to upper precision</i>) | Creates a random number. |
| 354 | (range <i>first to last by increment</i>) | Creates a list of numbers. |
| 355 | (rational <i>numerator denominator</i>) | Converts values to a rational number. |
| 356 | (read <i>file count bits</i>) | Creates a string or stream. |
| 357 | (ready-p <i>task</i>) | True if a task has completed. |
| 358 | (real <i>value</i>) | Converts a value to a real number. |
| 359 | (reclaim) | Performs garbage collection. |
| 360 | (reduce <i>sequence function</i>) | Converts a sequence into a value. |
| 361 | (reference <i>symbol</i>) | Returns a reference to a symbol. |
| 362 | (relation <i>name inclusion ... definition ...</i>) | Creates a relation. |
| 363 | (relations) | Returns a list of defined relations. |

| | | |
|-----|---|--|
| 364 | (release locks) | Releases locks on critical sections. |
| 365 | (remove assortment value ...) | Creates a new assortment by removing values. |
| 366 | (repeat count expression ...) | Loops a specified number of times. |
| 367 | (replace container entries option) | Modifies an assortment or sequence by with replacement values. |
| 368 | (require module as moniker from url ... options ...) | Retrieves an absent module or domain from a url. |
| 369 | (reset series) | Resets a series for reuse. |
| 370 | (resize sequence length default) | Modifies the length of a sequence. |
| 371 | (rest sequence skip) | Creates a subsequence of all except the first elements. |
| 372 | (retract thought ...) | Deletes thoughts using a pattern. |
| 373 | (return value from function) | Terminates a function call. |
| 374 | (reverse sequence) | Creates a reversed sequence. |
| 375 | (right first second) | True if values are equal or the first value is nil. |
| 376 | (rip assortment value ...) | Removes elements from assortments by value. |
| 377 | (round number) | Rounds a number to the nearest integer. |
| 378 | (rule name salience priority domain ruleset so bindings to goals if facts option ... action then domain) | Creates a rule. |
| 379 | (rules domain) | Creates a list of all rules in a domain. |
| 380 | (same value value ...) | True if all values are equal or contain equal elements. |
| 381 | (scale list scalar ... function) | Applies a function to a list of numbers and scalar values. |
| 382 | (scatter arguments functions) | Applies functions in parallel returning a list of results. |
| 383 | (scope keyval ...) | Finds an environment or gets the current environment. |
| 384 | (seal prototype) | Prohibits new records. |
| 385 | (secant number geometry metrum) | Calculates the secant of the number. |
| 386 | (seconds seconds) | Creates a seconds value or returns seconds since year zero. |
| 387 | (seek file position) | Repositions a file resource to a new read/write location. |
| 388 | (self) | Returns the currently executing task resource. |
| 389 | (separator kind) | Creates a separator string. |
| 390 | (series iterable) | Creates a series. |
| 391 | (service url handler) | Creates and starts a web service. |
| 392 | (set manner assignment ...) | Assigns variables to values in tandem returning true. |
| 393 | (settings url association ...) | Creates a configuration file.. |
| 394 | (sever assortment key ...) | Creates a new assortment by removing keys. |
| 395 | (shuffle sequence seed) | Creates a reordered sequence. |
| 396 | (sigma list) | Calculates the sigma of a sequence. |
| 397 | (signal condition) | Signals a condition. |
| 398 | (significand number kind) | Calculates the modified significand as zero, or a number from 1 to 10. |
| 399 | (signum number option ...) | Returns the sign of a number as a number, sigil, or word. |
| 400 | (sine number geometry metrum) | Calculates the sine of a number. |
| 401 | (so bindings expression ...) | Creates a scope with bindings. |
| 402 | (socket option ...) | Creates a TCP socket. |
| 403 | (some symbol ... binding sequence expression ... test) | True if the test is true for any element in the sequence. |
| 404 | (sort sequence ordering option ...) | Creates a sorted sequence. |
| 405 | (split sequence delimiter) | Creates a list of subsequences divided at a delimiter. |
| 406 | (starts-p interval ₁ interval ₂ tolerance) | True if two intervals start together. |
| 407 | (statistics numbers) | Finds the mean, median, sigma, and count of a list of numbers. |
| 408 | (step symbol from i limit j by k expression ...) | Loops a symbol over a numeric range to evaluate expressions. |
| 409 | (stream streamable) | Creates a stream. |
| 410 | (string value ...) | Converts a value to a string. |
| 411 | (structure name inclusion ... definition ...) | Creates a structure. |
| 412 | (structures) | Creates a list of all structures. |
| 413 | (sub sequence entry ...) | Replaces a subsequence within a sequence. |
| 414 | (subset-p subset superset) | True if all elements in a subset are in a superset. |
| 415 | (substitute sequence entries) | Modifies a sequence replacing subsequences. |

| | | |
|-----|--|--|
| 416 | (subsumes-p <i>superset subset ...</i>) | True if all elements in each subset are in the superset. |
| 417 | (sum <i>value ...</i>) | Calculates the arithmetic sum. |
| 418 | (summation <i>symbol ... binding sequence expression ...</i>) | Adds an expression over a set of values. |
| 419 | (supply <i>arguments function environment</i>) | Applies a function to an argument list. |
| 420 | (suppose <i>knowledge facts facts action goals by strategies via operators limit quantity gate condition within timeframe before timepoint options options</i>) | Performs hypothetico-deductive reasoning from facts to goals. |
| 421 | (survey <i>format relation slot</i>) | Creates a list of referents or referent-count pairs. |
| 422 | (swap <i>sequence substitution ...</i>) | Modifies a sequence by interchanging values. |
| 423 | (symbol <i>name</i>) | Creates a symbol. |
| 424 | (symbols <i>environment ancestors</i>) | Lists the symbols in an environment. |
| 425 | (take <i>readable count</i>) | Creates an expression from a readable sequence. |
| 426 | (takeable <i>readable desired</i>) | Calculates the number of expressions that can be read. |
| 427 | (tally <i>pattern</i>) | Returns the number of matches in the knowledge base. |
| 428 | (tangent <i>number geometry metrum</i>) | Calculates the tangent. |
| 429 | (tarry <i>tasks expression ...</i>) | Allows tasks to complete on their own. |
| 430 | (task <i>scope expression ...</i>) | Creates a list of tasks for deferred evaluation. |
| 431 | (tasks) | Creates a list of all tasks. |
| 432 | (taxon <i>value</i>) | Returns the datatype of a value. |
| 433 | (taxon-p <i>value taxon</i>) | True if the value is of the specific taxonomic type. |
| 434 | (taxonomy <i>value</i>) | Returns a list of datatypes for a value. |
| 435 | (tell <i>who message timeout</i>) | Sends a message to a recipient. |
| 436 | (there <i>url</i>) | True if a file, folder, or url exists. |
| 437 | (thing-p <i>value</i>) | True if a value is a thing.. |
| 438 | (think <i>problem domain</i>) | Creates a match-infer loop task to solve problems. |
| 439 | (this <i>series</i>) | The current element of a series. |
| 440 | (thought <i>relation id</i>) | Finds an extant thought. |
| 441 | (thunk <i>expression ...</i>) | Creates a thunk in a module. |
| 442 | (thunks <i>module</i>) | Returns a list of thunks defined in a module. |
| 443 | (tick <i>nanoseconds</i>) | Creates a tick or returns nanoseconds since year zero. |
| 444 | (tie <i>manner assignment ...</i>) | Assigns variables to values returning the last assigned value. |
| 445 | (time <i>unit operation time ...</i>) | Performs time operations. |
| 446 | (top <i>sequence count</i>) | Creates a subsequence of the first elements. |
| 447 | (transfer <i>source destination option ...</i>) | Transfers a value from one sequence to another. |
| 448 | (traverse <i>symbol binding sequence limit dimensions expression ...</i>) | Loops through the coordinates of a sequence. |
| 449 | (trim <i>sequence option ...</i>) | Creates a sequence without front and rear delimiters. |
| 450 | (truncate <i>number</i>) | Rounds a number towards zero. |
| 451 | (try <i>expression ... learn symbol recovery ... finally cleanup ...</i>) | Evaluates expressions and handles signals. |
| 452 | (tuple <i>value ...</i>) | Creates a tuple. |
| 453 | (uid) | Creates a unique identifier. |
| 454 | (unbound <i>function</i>) | Creates a list of free variables for a function. |
| 455 | (unbound-p <i>symbol</i>) | True if a symbol is unbound. |
| 456 | (unicode <i>string</i>) | The Unicode of a string's first position. |
| 457 | (union <i>sequence sequence ...</i>) | Concatenates sequences removing duplicates. |
| 458 | (union <i>sequence sequence ...</i>) | Concatenates sequences removing duplicates. |
| 459 | (unique <i>prefix</i>) | Creates a unique literal based on a prefix. |
| 460 | (unless <i>condition false-case true-case</i>) | Branched conditional execution. |
| 461 | (unlike <i>sequence pattern</i>) | True if a pattern does not match a sequence. |
| 462 | (unqualified <i>function</i>) | The function literal without the module prefix. |
| 463 | (unseal <i>prototype</i>) | Permits new records for a prototype. |
| 464 | (unsigned <i>value</i>) | Converts a value to an unsigned number. |
| 465 | (until <i>condition expression ...</i>) | Loops expressions until a condition is true. |

| | | |
|-----|--|---|
| 466 | (unwrap <i>module</i>) | Permits modifications to a module. |
| 469 | (uppercase <i>string</i>) | Converts a string to uppercase. |
| 470 | (uppercase-p <i>value</i>) | True if a string's first position is upper case. |
| 471 | (url <i>option ...</i>) | Creates a URL resource. |
| 472 | (use <i>knowledge expression ...</i>) | Evaluates expressions in a knowledge base scope. |
| 473 | (using <i>scope expression ...</i>) | Accesses a scope. |
| 474 | (values <i>assortment</i>) | Creates a list of values for an assortment. |
| 475 | (var <i>assignment ...</i>) | Adds variables to the current environment returning last value. |
| 476 | (vector <i>atom ...</i>) | Creates a vector of atoms. |
| 477 | (version) | Provides version information. |
| 478 | (void-p <i>sequence</i>) | True if a container has zero elements. |
| 479 | (wait <i>tasks timeout</i>) | Waits for tasks to end. |
| 480 | (which <i>pattern options ...</i>) | Creates a list of thoughts for a pattern. |
| 481 | (while <i>condition expression ...</i>) | Loops expressions while a condition is true. |
| 483 | (whitespace-p <i>value</i>) | True if the first position of a string is whitespace. |
| 484 | (with <i>premises option ... action expression</i>) | Matches patterns against the knowledge base. |
| 485 | (within <i>which slot premises option ... action</i>) | True if a position is inside a sequence. |
| 486 | (wrap <i>module</i>) | Prohibits modifications to a module. |
| 487 | (write <i>writeable value ...</i>) | Writes values to a writeable resource. |
| 488 | (xnor <i>truth truth</i>) | Logical equivalence. |
| 489 | (xor <i>truth truth</i>) | Logical exclusive disjunction. |
| 490 | (yield <i>value</i>) | Returns a result from a series generator. |
| 491 | (zap <i>container place ...</i>) | Updates associations and sequences with nil values. |
| 492 | (zero-p <i>number</i>) | True if a number is zero. |

Table 1. Function Quick Reference

8. Function Reference

This chapter outlines each function in detail.

-- *decrement*

Decrements a number by 1.

Syntax

(-- *number*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--|
| number | variant | 1 | A number or list to be decremented by 1. |

Results

| Data Type | Description |
|-----------|-------------------------|
| number | The decremented number. |

Remarks

This function decrements the argument by 1. If an argument is a list, then each element of the list is decremented.

Example

```
> (-- 1)
.: 0
> (-- -1)
.: -2
> (-- -1000+i)
.: -1001+i
> (-- {5 -8 90 32})
.: {4 -9 89 31}
```

Related

[++ increment](#)

- *subtraction*

Subtraction.

Syntax

(- *number ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|------------------------------------|
| number | variant | 1+ | A number or list to be subtracted. |

Results

| Data Type | Description |
|-----------|--|
| variant | The difference of all supplied numbers as a number or a list of differences. |

Remarks

If there is one argument, the sign is reversed. If there are two or more arguments, all arguments are subtracted from the first. If any of the arguments are **unknown**, **infinity** or **ninfinity** (i.e. negative infinity), the result shall respectively be **unknown**, **infinity**, or **ninfinity**. If an argument is a list, then each element of the list is subtracted. If a scalar and a list are subtracted, then the scalar is subtracted from each element of the list.

Example

```
> (- 1 2 3 4 5)
.: -13

> (- 1)
.: -1

> (- (- 1))
.: 1

> (- {3 4 5} 2 {1 0 2})
.: {0 2 1}
```

Related

+ addition, / division , * multiplication

@ element

Retrieves elements from a sequence or assortment.

Syntax

```
(@ container place ... )
```

place ::= integer
place ::= #
place ::= { integer ... }
place ::= key

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--|
| container | thing | 1 | A sequence, assortment, or record. |
| place | variant | 0+ | An integer position, the literal # for the last item, or a list of positions designating a coordinate, or an assortment key. |

Results

| Data Type | Description |
|------------|---|
| expression | Returns the item(s) at the position(s). |

Remarks

For sequences, this function returns the element(s) at the one-based position in the sequence. If place is not specified, the first element is returned. If place is the literal #, then the last item is returned. If place is a number or list of numbers, the item at the place within the sublist is returned. If the sequence is empty, or if a coordinate or position does not exist, the function fails. For assortments, the place is a required parameter and must correspond to a key within the assortment, otherwise the function fails.

Example

```
> (@ "abcde")
.: "a"

> (@ {{a}{b}}{c}{d}) {1 2 1}
.: b

> (@ (lexicon a 1 b 2 c 3) a c)
.: 1 3
```

Related

size, add, append, cut, put, enq, deq, has, lacks

size

Returns the number of elements.

Syntax

(# *container*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|-------------------------------------|
| container | thing | 1 | A sequence, assortment or resource. |

Results

| Data Type | Description |
|-----------|-------------------------|
| integer | The number of elements. |

Remarks

Returns the number of elements in a sequence or assortment. If a file resource is provided, then it returns the file size in bytes.

Example

```
> (# {a b c d e})
.: 5

> (# "she sells sea shells")
.: 20

> (# (lexicon a b c d))
.: 2

> (# (environment (scope) ?a 1 ?b 2 ?c 3 ?d 7))
.: 4

> (# (entries (enumeration Colors red orange yellow)))
.: 3

> (# (file (path "." "/" "premise.out")))
.: 436
```

Related

@ *element*

\$ concatenate

Creates a new string with intervening spaces .

Syntax

`($ value ...)`

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | A value |

Results

| Data Type | Description |
|-----------|--------------------------|
| string | The concatenated string. |

Remarks

Returns the concatenation of all values presented. All values are converted to strings, then concatenated. A space is preserved between each argument. Leading and trailing spaces are omitted. Any occurrence of the literal \s is replaced with a space. Any occurrence of the literal \n is replaced with a newline.

Example

```
> ($ the quick "brown" fox)
.: "the quick brown fox"

> ($ "the quick" \s "brown fox")
.: "the quick    brown fox"

> ($ {the quick} {} {brown fox})
.: "{the quick} {} {brown fox}"

> ($ atom \n molecule)
.: "atom
molecule"

> ($)
.: ""
```

Related

`$$ elide, string`

\$\$ elide

Creates a new string by eliding arguments.

Syntax

(\$\$ value ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | A value |

Results

| Data Type | Description |
|-----------|---|
| string | The concatenated string without spaces. |

Remarks

Returns the elision of all values presented. All values are converted to strings, then concatenated. Spaces are removed between arguments as are leading and trailing spaces. Any occurrence of the literal `\s` is replaced with a space. Any occurrence of the literal `\n` is replaced with a newline.

Example

```
> ($$ the quick "brown" fox)
.: "thequickbrownfox"

> ($$ the quick \s \s brown fox)
.: "thequick  brownfox"

> ($$ {the quick} {} {brown fox})
.: "{the quick}{}{brown fox}"

> ($$ atom \n molecule)
.: "atom
molecule"

> ($$)
.: ""
```

Related

`$ concatenate, string`

% shell

Executes a shell command.

Syntax

(% *command argument ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|----------------------------|
| command | literal | 1 | A shell command. |
| argument | Expression | 0+ | An argument to the command |

Results

| Data Type | Description |
|------------|---|
| expression | The output of the command as a list of strings and an integer result. |

Remarks

Returns an expression containing a list of strings from the output of the command and an integer result.

Example

```
> (% pwd)
.: {"/user/moi"} 0
> (% mkdir tmp)
.: {} 0
> (% ls)
.: {"tmp"} 0
> (% rmdir tmp)
.: {} 0
```

Related

[files, folders](#)

& merge

Merges arguments into a new sequence.

Syntax

(& value ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | A value |

Results

| Data Type | Description |
|-----------|----------------------|
| sequence | The merged sequence. |

Remarks

Returns a flattened sequence merging all subsequences of the same type. If the first element is not a sequence, then a list is returned. The sequence type shall be that of the first element.

Example

```
> (& {1 2 3} 4 5 {6} {7} {8 9} 10)
.: {1 2 3 4 5 6 7 8 9 10}

> (& [1 2 3] 4 5 [6] [7] [8 9] 10)
.: [1 2 3 4 5 6 7 8 9 10]

> (& |1 2 3| |4 5| |6| |7| |8 9| |10|)
.: |1 2 3 4 5 6 7 8 9 10| 

> (& {the quick} {} [brown fox])
.: {the quick brown fox}

> (& atom)
.: {atom}

> (&)
.: {}
```

Related

&& abridge, mid, insert, pop, push, swap, transfer

&& abridge

Merges arguments into a new sequences and removes nils.

Syntax

(&& value ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | A value |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| sequence | The merged sequence without nils. |

Remarks

The **&&** function returns a flattened sequence concatenating all values except nils.

Example

```
> (&& {} the quick brown fox {} nil nil jumped)
.: {the quick brown fox jumped}
> (&& [the quick] nil [] nil [brown [fox]])
.: [the quick brown [fox]]
> (&&)
.: {}
> (&& nil nil nil)
.: {}
```

Related

& merge, mid, insert, pop, push, swap, transfer

* *multiplication*

Multiply.

Syntax

(* *number number ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|------------------------------|
| number | variant | 2+ | A number or list of numbers. |

Results

| Data Type | Description |
|-----------|---|
| variant | The product of all factors or a list of products. |

Remarks

Returns the product of the arguments. If an argument is a list, then each element of the list is multiplied. If a scalar and a list are multiplied, then the scalar is multiplied with each element of the list. All lists must be of the same length.

Example

```
> (* 1 2 3)
.: 6
> (* {1 2 3} 5 {2 4 6})
.: {10 40 90}
```

Related

/ division, % modulo, + addition, - subtraction, ** exponent

**** exponentiation**

Raises a base number to an exponent.

Syntax

(root exponent)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------|
| root | variant | 1 | A number or list of numbers. |
| exponent | variant | 1 | A number or list of numbers. |

Results

| Data Type | Description |
|-----------|---|
| variant | The root raised to the indicated power or list of the same. |

Remarks

If **(** root exponent) = number**, and **(// 1 exponent) = degree** then **(// number degree) = root**.
If no denominator is specified, this function returns the square root of the number..

Example

```
> (** 2 3)
.: 8
> (** (e) 2)
.: 7.3890461484
> (** {2 4 6} 2)
.: {4 16 36}
> (** 2 {2 3 4})
.: {4 8 16}
> (** {2 3 4} {2 3 4})
.: {4 27 256}
> (** 4 0.5)
.: 2
```

Related

* multiplication, / division, root

/ *division*

Division.

Syntax

(/ *dividend divisor ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------|
| dividend | variant | 1 | A number or list of numbers. |
| divisor | .aogalk | 0+ | A number or list of numbers. |

Results

| Data Type | Description |
|-----------|---|
| variant | Returns the quotient of the numbers or a list of quotients. |

Remarks

Returns the quotient of the values. If there is one value presented, this function returns the reciprocal of the value. If there are two values, it divides the first value by the second. If there are more than two values, it divides the result by each successive number. Division by zero shall result in the literal **unknown** being returned. All list arguments must be the same length. If an argument is a list, then each element of the list is divided. If a scalar and a list are divided, then the scalar is divided by each element of the list.

Example

```
> (/ 5.0 2)
.: 2.5
> {/ {4 5 6} 4.0}
.: {1 1.25 1.5}
> (/ {4 5 6} {1.0 2.0 3.0})
.: {4 2.5 2}
> (/ 5 0)
.: undefined
```

Related

divide safe divide, * multiplication, + addition, - subtraction, % modulo, // nth root

// *nth root*

Returns the *n*th root of a number.

Syntax

(// *number degree*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------------------|
| number | number | 1 | A number |
| degree | number | 0-1 | A number (Default is 2) |

Results

| Data Type | Description |
|-----------|-------------|
| variant | The root. |

Remarks

If $(** \text{root} \text{ exponent}) = \text{number}$, and $(/\! 1 \text{ exponent}) = \text{degree}$ then $(// \text{number} \text{ degree}) = \text{root}$.
If no denominator is specified, this function returns the square root of the number.

Example

```
(// 866020)
.: 930.6019557254326
// 9
.: 3
// 8 3
.: 2
// 27 3
.: 3
```

Related

** *exponentiation, log*

/~ dissimilarity

Calculates the dissimilarity between two values.

Syntax

(/~/ *value₁* *value₂*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 2 | Any value |

Results

| Data Type | Description |
|-----------|--|
| real | The degree of dissimilarity between the two arguments. |

Remarks

Returns a number between 0 and 1, which identifies how dissimilar two numbers are.

Returns a number between 0 and 1. For numbers the following formula is used:

```
(- 1 (/ 1 (+ 1 (abs (- ?v1 ?v2)))))
```

If the values are of different types, they are treated as strings. For strings, lists of trigrams (ngrams of size 3) are generated and compared. For lists in general the following formula is used:

```
(- 1 (- (/ (# (intersection ?v1 ?v2)) (min (# ?v1) (# ?v2))) (* 0.1 (/ (# (difference ?v1 ?v2)) (min (# ?v1) (# ?v2))))))
```

If either list is empty, zero is returned.

Example

```
> (/~/ {a b c} {a d e})
.: 0.73333333333333334

> (/~/ "a b c" "a d e")
.: 0.21428571428571425

> (/~/ 57 890)
.: 0.9988009592326139
```

Related

~ similarity, like, unlike, = equal, /= unequal

/= unequal

Returns true if any values are not equal.

Syntax

(/= *value₁* *value₂* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | variant | 2+ | A value to be compared. |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if all values are not the same |

Remarks

Returns true if any values are not the same. Once two values are determined to be equal, the function short circuits and no further processing occurs.

For example (/= 1 2 3 5 3 6). Since 1≠2, we need to keep looking. Since 2≠3, we need to keep looking. Since 3≠5, we need to keep looking. Since 5≠3, we need to keep looking. However, since 3=3 (positions 3 and 5), false can be returned.

In the case of (/= (f ?x) (g ?x) (h ?x)), if f(?x) ≠ g(?x), h(?x) gets called. But, if f(x) = g(x), h(x) does not get called and false is returned.

Example

```
> (/= 1 2 3)
.: true
> (/= 1 1 1)
.: false
```

Related

= *equal*, ~ *similarity*, /~ *dissimilarity*

~ similarity

Calculates the similarity between two values.

Syntax

(~ *value₁* *value₂*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 2 | Any value |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| float | A similarity measure between 0 and 1. |

Remarks

Returns a number between 0 and 1. For numbers the following formula is used:

```
(/ 1 (+ 1 (abs (- ?v1 ?v2))))
```

If the values are of different types, they are treated as strings. For strings, lists of trigrams (ngrams of size 3) are generated and compared. For lists in general the following formula is used:

```
(- (/ (# (intersection ?v1 ?v2)) (min (# ?v1) (# ?v2)))
     (* 0.1 (/ (# (difference ?v1 ?v2)) (min (# ?v1) (# ?v2)))))
```

If either list is empty, zero is returned.

Example

```
> (~ {a b c} {a d e})
.: 0.26666666666666666666
> (~ "a b c" "a d e")
.: 0.21428571428571425
> (~ 57 890)
.: 0.001199040767386091
```

Related

/~ dissimilarity, = equal, /= unequal, like, unlike

' quote

Quotes a variant.

Syntax

(' *variant*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|------------------------|
| variant | variant | 1 | A variant to be quoted |

Results

| Data Type | Description |
|-----------|--------------------------|
| variant | The value after quoting. |

Remarks

The quote character signifies the quote function. All arguments are returned unevaluated.

Example

```
> (' foo)
.: foo
> (' (+ 1 2 3))
.: (+ 1 2 3)
> (' "a string")
.: "a string"
> (' (foo))
.: (foo)
> (' (+ 1 , (* 2 3)))
.: (+ 1 , (* 2 3))
```

Related

`expand, eval, function, macro, quote, unquote

` expand

Expands a variant by substituting values.

Syntax

(` *variant*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--------------------------|
| variant | variant | 1 | A variant to be expanded |

Results

| Data Type | Description |
|-----------|-------------------------------|
| variant | The argument after expansion. |

Remarks

The backquote character is the expand function. The argument is returned unevaluated unless it is preceded by or contains a comma (and underscore), in which case the value following the comma shall be evaluated in the present scope (or in which case the value following the underscore must evaluate to a list which shall be merged into the existing expression).

Example

```
> (` (+ 1 2 "3"))
.: (+ 1 2 "3")

> (so {?a 1 ?b 2 ?c {3 4 5}}  (` {, ?a , ?b , ?c}))
.: {1 2 {3 4 5}}

> (so {?a 1 ?b 2 ?c {3 4 5}}  (` {, ?a , ?b ,_ ?c}))
.: {1 2 3 4 5}

> (` (foo))
.: (foo)

> (` (+ (+ 1 1) ,(* 2 3)))
.: (+ (+ 1 1) 6)
```

Related

' quote, eval, function, macro

+ *addition*

Addition.

Syntax

(+ *number* *number*...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|------------------------------|
| number | variant | 2+ | A number or list of numbers. |

Results

| Data Type | Description |
|-----------|----------------------------|
| variant | The sum or a list of sums. |

Remarks

Calculates the sum of the values. The values must all be numbers or lists containing numbers. If an argument is a list, then each element of the list is added. If a scalar and a list are added, then the scalar is added to each element of the list. All lists must be the same length.

Example

```
> (+ 1 2 3)
.: 6
> (+ 5 {6 7 8 9 10})
.: {11 12 13 14 15}
> (+ {1 2 3} {4 5 6} 8)
.: {13 15 17}
```

Related

- minus, * multiply, / divide

++ increment

Increments a number by 1.

Syntax

(++ *number*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|------------------------------|
| number | variant | 1 | A number or list of numbers. |

Results

| Data Type | Description |
|-----------|--|
| variant | The incremented number or list of numbers. |

Remarks

This function increments the argument by 1. If the argument is a list then each number in the list is incremented by one.

Example

```
> (++ 1)
.: 2

> (++ 100)
.: 101

> (++ {99 -101 0})
.: {100 -100 1}
```

Related

-- decrement

<-- before tie

The value before tying a symbol to a new value.

Syntax

(<-- *function symbol value ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------------|
| function | literal | 1 | A function |
| symbol | symbol | 1 | A symbol |
| value | value | 0+ | A value used in the function call. |

Results

| Data Type | Description |
|-----------|---|
| variant | Returns the value bound to the symbol before the function is applied. |

Remarks

None

Example

```
> (let ?n 0)
  .: true
> (<-- + ?n 7)
  .: 0
> ?n
  .: 7
```

Related

--> *after tie, assume, assign, global, local, put, set, swap, tie*

< less

True if each value is less than the next.

Syntax

(< *value value ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | variant | 2+ | A value to be compared. |

Results

| Data Type | Description |
|-----------|--------------------------------|
| truth | True if values are increasing. |

Remarks

Returns true if each value is less than its successor from left to right, and false otherwise. All values must be of the same type (either a literal, number, or string)

Example

```
(< a b c)
.: true

(< "johnny" "ralph" "sam")
.: true

(< 3 6 2 8)
.: false

(< Amanda Samantha Zelda)
.: true
```

Related

<= less or equal, > greater, >= greater or equal , = equal , /= unequal

`<= less or equal`

True if each value is less or equal to the next.

Syntax

`(<= value value ...)`

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | variant | 2+ | A value to be compared. |

Results

| Data Type | Description |
|-----------|--------------------------------|
| truth | True if values are increasing. |

Remarks

Returns true if each value is less than its successor from left to right, and false otherwise. All values must be of the same type, and must be numbers, literals, or strings.

Example

```
> (<= a b b c)
.: true

> (<= "johnny" "ralph" "sam")
.: true

> (<= 3 6 2 8)
.: false
```

Related

`< less, > greater, >= greater or equal , = equal , /= unequal`

<== before put

The value before replacement.

Syntax

(<== *container function place value ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|------------------------------------|
| container | variant | 1 | A sequence or assortment. |
| function | function | 1 | A function |
| place | variant | 1 | A key or position. |
| value | value | 0+ | a value used in the function call. |

Results

| Data Type | Description |
|-----------|-----------------------------|
| variant | Returns the new place value |

Remarks

None

Example

```
> (relation Q :x 0)
.: Q

> (global ?Q (new Q))
.: true

> (<== ?Q + :x 1)
.: 0

> (<== ?Q ++ :x)
.: 1

> (:x ?Q)
.: 2
```

Related

==> after put

= equal

True if each value is equal to the next.

Syntax

(= *value value ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | variant | 2+ | A value to be compared. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | True if all values are the same |

Remarks

Returns true if each value is the same as its successor from left to right, and false otherwise. All values must be of the same type, and must be numbers, literals, or strings. Short circuits if false. Two items are equal if their representation is the same regardless of whether or not they occupy the same location in memory (viz. identical) or if their printed representation is the same (e.g., **true** and **\%True** are equal since the printed representation of **\%True** is **true**).

Example

```
> (= a a a a)
.: true

> (= "johnny" "ralph" "sam")
.: false

> (= 3 3)
.: true

> (= 1 1/1 1.0)
.: true

> (= true \%True)
.: true
```

Related

< less , <= less or equal, > greater, >= greater or equal, /= unequal, alike, identical

==> after put

The value after replacement.

Syntax

(==> container function place value ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|------------------------------------|
| container | variant | 1 | A sequence or assortment. |
| function | function | 1 | A function |
| place | variant | 1 | A key or position. |
| value | value | 0+ | a value used in the function call. |

Results

| Data Type | Description |
|-----------|-----------------------------|
| variant | Returns the new place value |

Remarks

None

Example

```
> (relation Q :x 0)
.: Q

> (global ?Q (new Q))
.: true

> (==> ?Q + :x 1)
.: 1

> (==> ?Q ++ :x)
.: 2

> (:x ?Q)
.: 2
```

Related

<=> before put

> greater

True if each value is greater than the next.

Syntax

(> *value value ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | value | 2+ | A value to be compared. |

Results

| Data Type | Description |
|-----------|--------------------------------|
| truth | True if values are decreasing. |

Remarks

Returns true if each value is greater than its successor from left to right, and false otherwise. All values must be of the same type, and must be numbers, literals, or strings.

Example

```
> (> c b a)
.: true

> (> "johnny" "ralph" "sam")
.: false

> (> 3 6 2 8)
.: false
```

Related

< less , <= less or equal, >= greater or equal , = equal , /= unequal

--> *after tie*

The value after tying a symbol to a new value.

Syntax

(--> *function symbol value ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------------|
| function | literal | 1 | A function |
| symbol | symbol | 1 | A symbol |
| value | value | 0+ | A value used in the function call. |

Results

| Data Type | Description |
|-----------|---|
| variant | Returns the last value bound to the symbol. |

Remarks

None

Example

```
> (so {?length 0}
      (for ?m in (modules)
        (--> + ?length (# ?m))))
.: 23
> (# (reduce (modules) $))
.: 23
```

Related

--> *before tie, constant, dynamic, global, local, put, set, swap, tie, use*

>= *greater or equal*

True if each value is greater or equal to the next.

Syntax

(>= value value ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | value | 2+ | A value to be compared. |

Results

| Data Type | Description |
|-----------|--------------------------------|
| truth | True if values are increasing. |

Remarks

Returns true if each value is greater or equal to its successor from left to right, and false otherwise.
All values must be of the same type, and must be numbers, literals, or strings.

Example

```
> (>= c b a)
.: true

> (>= "johnny" "ralph" "sam")
.: false

> (>= "sam" "ralph" "johnny")
.: true

> (>= 8 7 6 3)
.: true
```

Related

< less , <= less or equal, > greater, = equal , /= unequal

\n newline

Returns a string containing one or more new lines.

Syntax

(\n *quantity*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| quantity | number | 0-1 | The number of new lines to create. Default is 1. |

Results

| Data Type | Description |
|-----------|--|
| string | A string containing a single new line character. |

Remarks

None.

Example

```
> (\n)
:
"
> (\n 3)
:
"
"
```

Related

\$ concatenate, \$\$ elide, \s space, string

\s space

Returns a string containing one or more spaces.

Syntax

(\s *quantity*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| quantity | number | 0-1 | The number of spaces to create. Default is 1. |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| string | A string containing a single space. |

Remarks

None.

Example

```
> (\s)
.: " "
> (\s 15)
.: " "
```

Related

\$ concatenate, \$\$ elide, \n newline, string

^ thought id

Returns a SubThought identifier.

Syntax

(*^ thought*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|---|
| thought | variant | 1 | A SubThought premise or SubThought identifier |

Results

| Data Type | Description |
|-----------|------------------------------------|
| thought | Returns the SubThought identifier. |

Remarks

Returns the identifier for a thought.

Example

```
> (relation A :S1 0 :S2 1)
.: A
> (new A)
.: A_1
> (premise A_1)
.: [A ^ A_1 :S1 0 :S2 1]
> (^ (premise A_1))
.: A_1
> (^ A_1)
.: A_1
> (^ FOO)
.: nil
```

Related

[id](#), [instantiate](#), [premise](#), [type](#)

abolish

Deletes variables from an environment.

Syntax

(abolish *symbols* *environment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|---|
| variables | variant | 1 | A symbol or a list of variables. |
| environment | environment | 0-1 | The environment containing the symbol(s). |

Results

| Data Type | Description |
|-----------|--|
| literal | Returns the literal abolished . |

Remarks

Disassociates the symbol or variables from their respective values in the specified environment. If unspecified, each symbol is sought in the current scope or an ancestor environment, all the way up to the global environment. If the symbol is found, it is disassociated, then deleted. If the symbol is not found, it is ignored..

Example

```
> (var ?person DrWho)
.: DrWho
> ?person
.: DrWho
> (abolish ?person)
.: abolished
> ?person
.: [Failure :Name UnboundSymbol :Text "The symbol ?person is unbound"]
```

Related

<-- before tie, --> after tie, assign, assume, constant, dynamic, global, let, local, only , set, tie

abort

Forcibly terminates one or more tasks.

Syntax

(abort *tasks* **)**

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---|
| tasks | variant | 1 | A single task resource or a list of task resources. |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| variant | Returns the tasks or list of tasks. |

Remarks

Immediately terminates an executing task.

Example

```
> (concurrent (step ?i from 1 to 100000 (if (cancelled-p (self)) (break oops))
finished)
              (step ?i from 1 to 100000 finished))

.: {Task-57 Task-58}

> (cancel task-57)

.: task-57

> (await task-57)

.: oops

> (abort task-58)

.: task-58

> (free {task-57 task-58})

.: freed
```

Related

await, cancel, cancelled-p, complete, critical, do, done-p, reclaim , start, task, task-p, tasks

about

Provides system and version information.

Syntax

(about)

Module

Base

Parameters

None

Results

None

Remarks

Returns system and version information.

Example

```
> (about)
.: [Premise :Version 1.1.0 :Build 20200430.001 :OS Win64 :Edition Community]
```

Related

bye, eval, quote

abs

Returns the absolute value.

Syntax

(abs number)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|---------------------|
| number | variant | 1 | A number or a list. |

Results

| Data Type | Description |
|-----------|--|
| variant | A non-negative number or a list of non-negative numbers. |

Remarks

Returns a non-negative number or a list of non-negative numbers. If the number is negative it is multiplied by -1. If the number is positive no change occurs.

Example

```
> (abs 100)
.: 100

> (abs -100)
.: 100

> (abs ninfinity)
.: ninfinity

> (abs unknown)
.: unknown

> (abs {0 -2 -99 24 56 infinity})
.: {0 2 99 24 56 infinity}
```

Related

- subtraction , + addition , % modulo, / division, * multiplication

absent

True if a file, folder, or url does not exist.

Syntax

(**absent** *url*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|-------------|
| url | url | 1 | A url. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | True if the url does not exist. |

Remarks

This function checks whether or not the url exists.

Example

```
> (let ?url (ul path "quick.txt"))

.: true

> (if (absent ?url)
    (let ?file (file url ?url seek 1 mode write erase yes))
    (write ?file "The quick brown fox")
    (close ?file))

.: closed
```

Related

file, folder, there, url

acosecant

Returns the inverse cosecant.

Syntax

(**acosecant** *number geometry metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| number | The inverse cosecant of the number. |

Remarks

Example

```
> (acosecant 0.3927)
.: 1.570796326-1.58686085i

> (acosecant 0.3927 cir radians)
.: 1.570796326-1.58686085i

> (acosecant 135 cir degrees)
.: 0.438313704

> (acosecant (pi) hyp radians)
.: 0.31316588045

> (acosecant 45 hyp degrees)
.: 1.06202877524
```

Related

cosine, tangent

acosine

Returns the inverse cosine.

Syntax

(**acosine** *number geometry metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| number | The inverse cosine of the number. |

Remarks

None.

Example

```
> (acosine 1.570796326794)
.: 0+1.0322747855i

> (acosine cir 1.570796326794 radians)
.: 0+1.0322747855i

> (acosine cir 22.5 degrees)
.: 1.1672317198

> (acosine hyp 4.712388980384 radians)
.: 2.23188925305

> (acosine hyp 90 degrees)
.: 0.9136433572987
```

Related

[cosine](#), [tangent](#)

acotangent

Returns the inverse cotangent.

Syntax

(acotangent *number geometry metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| number | The inverse cotangent of the number. |

Remarks

Example

```
> (acotangent 0.785398163397)
.: 0.905022576

> (acotangent cir 0.785398163397 radians)
.: 0.905022576

> (acotangent cir 180 degrees)
.: 0.3081690711

> (acotangent hyp 2.356194490192 radians)
.: 0.453062565662

> (acotangent hyp 120 degrees)
.: 0.519695325409
```

Related

cosine, tangent

actual

Returns a the underlying symbol for a reference.

Syntax

(actual *reference*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--------------------|
| Reference | reference | 1 | A symbol reference |

Results

| Data Type | Description |
|-----------|--------------------------------|
| symbol | Returns the symbol referenced. |

Remarks

The convention for denoting reference variables is to use two question marks: ??name.

Example

```
> (structure ListNode :Val :Next)
  .: ListNode

> (function IsPalindrome2 {ListNode ?tail reference ??head}
  (ensure ListNode (actual ??head))
  (if (null-p ?tail)
    (return true)
  or (and (IsPalindrome2 (:Next ?tail)) (= (:Val ?tail) (:Val ??head)))
    (set ??head (:Next ??head))
    (return true)
  else (return false)))

.: IsPalindrome2

> (function IsPalindrome {?head}
  (ensure ListNode ?head)
  (return (IsPalindrome2 ?head (reference ?head)))))

.: IsPalindrome

> (IsPalindrome
  (new ListNode :Val a :Next (new ListNode :Val b :Next (new ListNode :Val a))))
  .: true
```

Related

assume, dynamic, given, global, let, reference, scope, set, tie

add

Modifies an assortment by adding entries.

Syntax

(**add** *assortment entry ...*)

entry ::= *key value*

entry ::= *key*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---------------------------|
| assortment | assortment | 1 | An assortment or sequence |
| entry | expression | 1+ | A key, or key value pair. |

Results

| Data Type | Description |
|------------|--------------------------|
| assortment | The modified assortment. |

Remarks

Collections require only a key for each entry. Assortments require a key and value for each entry.

Example

```
> (entries (add (lexicon) a 1 b 2 c 3))  
.: {{a 1}{b 2}{c 3}}  
  
> (entries (add (collection) 1 2 3 4 5))  
.: {{1 1}{2 2}{3 3}{4 4}{5 5}}
```

Related

size, cut, deq, enq, get, insert, key, keys, push, put, rip, values, zap

address

Returns the address of a URL web resource.

Syntax

(address *url*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|---------------------|
| url | URL | 1 | A url web resource. |

Results

| Data Type | Description |
|-----------|--------------------------|
| string | A valid URL in a string. |

Remarks

If no arguments are provided, then this function returns the universal resource locator for the SubThought Premise interpreter running on the current cell.

Example

```
> (address (url))
.: "http://localhost"

> (address (url scheme https))
.: "https://localhost"

> (url scheme https host "www.youtube.com" path "watch" query "v=ja76cnLKSL4")
.: Url-3

> (address Url-3)
.: "https://www.youtube.com/watch?v=ja76cnLKSL4"
```

Related

[url](#), [url-p](#)

agent

Creates an agent.

Syntax

(agent job url handler delay)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|---|
| job | function | 1 | A niladic function for the agent to repeatedly perform. |
| url | url | 1 | A Universal Resource Locator.. |
| handler | function | 1 | An message handler function. |
| delay | time | 0-1 | The delay between job executions. Default is 1 second. |

Results

| Data Type | Description |
|-----------|-----------------|
| agent | The agent task. |

Remarks

This function creates an asynchronous agent task that repeatedly performs a job and also processes incoming messages. The message handler function should receive a sender url and a message string.

Example

```
> (let ?url (url scheme tcp port 5001))

.: true

> (function reply {url ?sender string ?message} (tell ?sender "OK"))

.: reply

> (function job {} (tell user ($ 1)))

.: job

> (agent job ?url reply \@s0.5)

.: Agent-1

> (do (start Agent-1) (wait 3) (cancel Agent-1) (free Agent-1))

111111.. freed
```

Related

ask, cancel, service, pause, perform, start, tell

align

Returns a sorted list using the provided ordering.

Syntax

(align sequence ordering)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-----------------------------------|
| sequence | sequence | 1 | A sequence of unordered elements. |
| ordering | sequence | 1 | An ordered sequence. |

Results

| Data Type | Description |
|-----------|----------------------|
| sequence | An ordered sequence. |

Remarks

Places items in the sequence in the specified ordering. If items in the sequence do not appear in the ordering, they are appended to the result in the order of appearance.

Example

```
> (align {b d c f} {a b c d})  
. : {b c d f}  
  
> (align "72qaj8x" "zyx123a")  
. : "x2a7qj8"
```

Related

[insort](#), [sort](#)

alike

True if each value has the same taxon.

Syntax

(alike *value value ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | variant | 2+ | A value to be compared. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | True if all values are the same |

Remarks

Returns true if each value has the same taxon as its successor from left to right, and false otherwise.

Example

```
> (alike a a a a)  
. : true  
  
> (alike 1 1/1 1.0)  
. : false  
  
> (alike "johnny" "ralph" "sam")  
. : true  
  
> (alike true \%True)  
. : false  
  
> (alike 1 five "three")  
. : false
```

Related

< less , <= less or equal, > greater, >= greater or equal, /= unequal, identical

all

True if all relevant knowledge satisfies all the premises.

Syntax

```
(all premise ... )  
  
premises ::= premise  
premises ::= premise premises  
premises ::= premise premises  
premises ::= premise restriction ... premises  
restriction ::= (predicate value ...)  
premise ::= [category descriptor ... ]  
category ::= type  
category ::= list  
descriptor ::= slot binding  
descriptor ::= slot condition  
binding ::= as symbol  
binding ::= each symbol  
condition ::= slot = value  
condition ::= slot is unary-predicate  
condition ::= slot not value  
condition ::= slot binary-predicate value  
condition ::= slot n-ary-predicate value-list  
condition ::= (predicate value ...)
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|----------------------------------|
| premise | variant | 1+ | A pattern, call, or truth value. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | True if all values are not nil. |

Remarks

True if all values are not nil, false otherwise. Evaluates values one at a time. If a value is nil then it returns immediately, ignoring remaining values. . If the value is an iterator, then a cursor is created over which the remaining values are tested until truth or falsity is determined.

Example

```
> (all one two three)  
. : true  
> (all a b nil)  
. : false
```

Related

[any](#), [no](#), [nall](#)

alphabetic-p

True if the first position of a string is alphabetic.

Syntax

(alphabetic-p *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the value is a string and the first position is alphabetic. |

Remarks

Can be upper case or lower case.

Example

```
> (alphabetic-p {a b c})  
. : false  
  
> (alphabetic-p "a b c")  
. : true  
  
> (alphabetic-p " a b c")  
. : false  
  
> (alphabetic-p 450)  
. : false  
  
> (alphabetic-p "450")  
. : false  
  
> (alphabetic-p "Hello World.")  
. : true
```

Related

capitalized-p, letter-p, lowercase-p, uppercase-p

alphanumeric-p

True if the first position of a string is a number or a letter.

Syntax

(alphanumeric-p *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the value is a string and the first position is alphanumeric. |

Remarks

True if the first position of the value is an upper or lower case alphabetic character or a digit.

Example

```
> (alphanumeric-p {a b c})  
. : false  
  
> (alphanumeric-p "a b c")  
. : true  
  
> (alphanumeric-p " a b c")  
. : false  
  
> (alphanumeric-p 450)  
. : false  
  
> (alphanumeric-p "450")  
. : true
```

Related

capitalized-p, letter-p, lowercase-p, uppercase-p

and

Logical conjunction.

Syntax

(and *truth* *truth* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------|
| truth | truth | 2+ | A truth value |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | True if all arguments are true. |

Remarks

Evaluates the values presented one at a time. If any of the values is false, this function returns false immediately, ignoring any remaining values. This function fails if any argument is not truth.

Example

```
> (and (is 234 (given {?x} (= (taxon ?x) number)))
       (is abc (given {?x} (= (taxon ?x) literal))))
 
.: true

> (and true true true true false)

.: false
```

Related

exists, nand, nil, nor, not, or, xor

any

True if any relevant knowledge satisfies all the premises.

Syntax

```
(any premise ... )  
  
premises ::= premise  
premises ::= premise premises  
premises ::= premise premises  
premises ::= premise restriction ... premises  
restriction ::= (predicate value ...)  
premise ::= [category descriptor ... ]  
category ::= type  
category ::= list  
descriptor ::= slot binding  
descriptor ::= slot condition  
binding ::= as symbol  
binding ::= each symbol  
condition ::= slot = value  
condition ::= slot is unary-predicate  
condition ::= slot not value  
condition ::= slot binary-predicate value  
condition ::= slot n-ary-predicate value-list  
condition ::= (predicate value ...)
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|----------------------------------|
| premise | variant | 1+ | A pattern, call, or truth value. |

Results

| Data Type | Description |
|-----------|-------------------------------|
| truth | True if any value is not nil. |

Remarks

True if any value is not nil. Returns immediately upon encountering the first non-nil value. If the value is an iterator, then a cursor is created over which the remaining values are tested until truth or falsity is determined.

Example

```
> (any nil nil apple)
.: true

> (any one two three four)
.: true

> (any nil nil)
.: false
```

Related

[all](#), [default](#), [is](#), [nall](#), [no](#)

append

Inserts values at the end of a sequence.

Syntax

(append *sequence value ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence |
| value | variant | 1+ | A value. |

Results

| Data Type | Description |
|-----------|------------------------|
| sequence | The modified sequence. |

Remarks

Inserts the value into the end of a sequence.

Example

```
> (let ?s {})
.: true

> (append ?s a b c)
.: {a b c}

> (append ?s d e f)
.: {a b c d e f}

> ?s
.: {a b c d e f}
```

Related

@, add, fix, insert, pop, push, swap

apply

Applies a function to a list of arguments.

Syntax

(apply *function arguments environment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|--------------------------------------|
| function | function | 1 | The name of a function to invoke |
| arguments | list | 1 | A list containing actual parameters. |
| environment | environment | 0-1 | An optional environment. |

Results

| Data Type | Description |
|-----------|--|
| value | The result from applying the function. |

Remarks

Applies the function to the list using the supplied environment, or the current scope if no environment is provided. The function is applied to the entire list and the result is returned.

Example

```
> (apply + {1 2 3})  
. : 6  
  
> (so { ?x 9  
        ?s (scope) }  
(so { ?x 3 }  
  
    (apply + {1 2 ?x} ?s)))  
. : 12
```

Related

count, gather, invoke, map, morph, reduce, supply

arity

Returns the number of variables in a function's parameter list.

Syntax

(arity *function kind*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| function | function | 1 | A function or function name . |
| kind | literal | 0-1 | One of required , optional , variadic , or all (default). |

Results

| Data Type | Description |
|-----------|--|
| integer | Returns the number of variables matching the parameter kind. |

Remarks

None

Example

```
> (function foo {?a + ?b ?c & ?d}
   (tell user ($ (&& ?a ?b ?c ?d))))  

.: foo  

> (arity foo)  

.: 4  

> (arity foo required)  

.: 1  

> (arity foo optional)  

.: 2  

> (arity foo variadic)  

.: 1
```

Related

has, in, member, within

array

Returns a multi dimensional list.

Syntax

(array *dimensions* *option*)

dimensions ::= *size*
dimensions ::= { *size* ... }

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| dimensions | variant | 1 | A number or list of numbers for each dimension |
| option | expression | 1 | A key value pair, either of expression (default), or each function. |

Results

| Data Type | Description |
|-----------|-------------|
| list | A list. |

Remarks

If option is **of** then all elements of the list are initialized to the same value. If option is **each**, during initialization, the function receives the current coordinate and the value is used for the position.

Example

```
> (array 5)
.: {nil nil nil nil nil}

> (array 5 of 0)
.: {0 0 0 0 0}

> (array 5 each (given {?x}(random 0 to 9)))
.: {7 5 8 2 4}

> (array {2 2} each (given {?coordinate}(reduce ?coordinate +)))
.: {{2 3}{3 4}}
```

Related

@, bind, edit, make

asecant

Returns the inverse secant.

Syntax

(asecant *number geometry metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| number | The inverse secant of the number. |

Remarks

Example

```
> (asecant 2.356194490192)
.: 1.13248262224

> (asecant cir 2.356194490192 radians)
.: 1.13248262224

> (asecant cir 120 degrees)
.: 1.0730291831

> (asecant hyp 0.3927 radians)
.: 1.58686

> (asecant hyp 45 degrees)
.: 0.72336752453
```

Related

cosecant, secant

asine

Returns the inverse sine.

Syntax

(**asine** *number geometry metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| number | The inverse sine of the number. |

Remarks

Example

```
> (asine 0.3927)
.: 0.403566

> (asine cir 0.3927 radians)
.: 0.403566

> (asine cir 95 degrees)
.: 1.570796326+1.092133189i

> (asine hyp 3.141592653589 radians)
.: 1.862295743311

> (asine hyp 120 degrees)
.: 1.4850704719
```

Related

cosine, sine

ask

Sends a message to a recipient and awaits a response.

Syntax

(**ask** *who message option ...*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|---|
| who | literal | 1 | A URL or the literal user for the console. |
| message | string | 0-1 | A message or query. |
| option | expression | 0+ | Key value pairs timeout <i>instant</i> – delay before terminating request, default <i>value</i> – value to return upon timeout, columns <i>list</i> – a column list is the first record, if omitted then no columns are listed. (default). skip <i>number</i> – skips a number of records limit <i>number</i> – number of records or #. (default is 1). meta <i>where</i> – either before or within . (default is before). |

Results

| Data Type | Description |
|-----------|--|
| variant | The response. The value nil is returned if no response was given. |

Remarks

Sends a message to the recipient and waits for a response. If **user** is specified for who, then the input is taken from the console. If the url is a data url, then this function retrieves query results from the data url. If the option **columns** is provided along with a list of columns, and the option meta is provided along with a location (**before** or **within**) then each column name is inserted either in the list element before the data, or within each data list.

Example

```
> (ask user "What is your name?")  
  
What is your name? Michael  
. : "Michael"  
  
> (service (url scheme tcp port 5950)  
           (function reply {?sender ?message}  
             (tell ?sender ($ received ?message))))  
  
. : Service-1  
  
> (start Service-1)  
  
. : Service-1
```

```

> (ask (url scheme tcp port 5950) "secret" timeout \@s5 default NobodyHome)

.: "received secret"

> (var ?data (data driver sqlserver server "//servername" port 1433
                     database mydata user "myUid" password "123456"))

.: Data-1

> (open ?data)

.: Data-1

> (ask Data-1 "select top 10 empid, name, salary from employee" record values)

.: {{1 "John Smith" 10.50}{2 "Jane Johnson" 12.00} ...}

> (ask ?data
      "select empid, name, wage from employee limit 2"
      columns {empid name wage}
      limit 2)

.: {{empid name wage} {1 "John Smith" 10.50} {2 "Jane Johnson" 12.00} ...}

> (ask ?data "select empid, name, wage from employee" limit 2)

.: {{1 "John Smith" 10.50}{2 "Jane Johnson" 12.00} ...}

> (ask ?data "select top 2 empid, name, wage from employee" meta within
              columns {:Id :Name :Rate})

.: {{:Id 1 :Name "John Smith" :Rate 10.50}
    {:Id 2 :Name "Jane Johnson" :Rate 12.00} ...}

> (ask ?data "select top 2 empid, name, wage from employee"
           columns {:Id :Name :Rate} meta before)

.: {{:Id :Name :Rate} {1 "John Smith" 10.50} {2 "Jane Johnson" 12.00} ...}

> (close ?data)

.: Data-1

```

Related

[cancel](#), [free](#), [Service](#), [start](#), [tell](#)

assert

Creates a new thought containing nested premises using new.

Syntax

(assert *relation descriptor ...* **)**

Module

KB

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--------------------------|
| relation | relation | 1 | A relationship. |
| descriptor | expression | 0+ | A slot expression pair.. |

Results

| Data Type | Description |
|-----------|--|
| thought | A new Context or .meron if :Meron is a descriptor. |

Remarks

Creates a new thought using the provided relation. If the thought has nested premises, for those premises with defined relationships, the premises will be replaced by a thought identifier using the **new** function.

Example

```
> (relation Operation :Name :Parameters :Context :Effects)
  (relation Is :Type :Item) (relation On :Above :Below) (relation Has :What :Item)

.: Operation Is On Has

> (assert Operation
  :Name PickUp :Parameters {$ob}
  :Context { and [On :Above CLEAR :Below $ob]
             [On :Above $ob :Below TABLE]
             [Has :What ARM :Item EMPTY] }
  :Effects { do {affirm [Has :What ARM :Item $ob] }
            {remove [On :Above CLEAR :Below $ob]
             [On :Above $ob :Below TABLE]
             [Has :What ARM :Item EMPTY] }})
.: Operation_1

> (premise Operation_1)

.: [Operation ^ Operation_1 :Name PickUp :Parameters {$ob} :Context {and On_3 On_4
Has_2} :Effects {do {affirm Has_2} {remove On_5 On_6 Has_3}}
```

Related

assume , clone, knew, new, relation

assume

Creates a new thought containing nested premises using knew.

Syntax

(assume *relation descriptor ...* **)**

Module

KB

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--------------------------|
| relation | relation | 1 | A relationship. |
| descriptor | expression | 0+ | A slot expression pair.. |

Results

| Data Type | Description |
|-----------|--|
| thought | A new Context or .meron if :Meron is a descriptor. |

Remarks

Creates a new thought using the provided relation. If the thought has nested premises, for those premises with defined relationships, the premises will be replaced by a thought identifier using the **new** function.

Example

```
> (relation Operation :Name :Parameters :Context :Effects)
  (relation Is :Type :Item) (relation On :Above :Below) (relation Has :What :Item)

.: Operation Is On Has

> (assume Operation
  :Name PickUp :Parameters {$ob}
  :Context { and [On :Above CLEAR :Below $ob]
             [On :Above $ob :Below TABLE]
             [Has :What ARM :Item EMPTY] }
  :Effects { do {affirm [Has :What ARM :Item $ob] }
            {remove [On :Above CLEAR :Below $ob]
             [On :Above $ob :Below TABLE]
             [Has :What ARM :Item EMPTY] }})
.: Operation_2

> (premise Operation_2)

.: [Operation ^ Operation_2 :Name PickUp :Parameters {$ob} :Context {and On_1 On_2
Has_1} :Effects {do {affirm Has_2} {remove On_1 On_2 Has_1}}
```

Related

assert, clone, knew, new, relation

atangent

Returns the inverse tangent.

Syntax

(atangent *number geometry metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|------------------------------------|
| number | The inverse tangent of the number. |

Remarks

None.

Example

```
> (atangent 2.356194490192)
.: 1.4691228248

> (atangent 2.356194490192 cir radians)
.: 1.4691228248

> (atangent 120 cir degrees)
.: 1.12533883288

> (atangent 0.785398163397 hyp radians)
.: 1.05930617082

> (atangent 85 hyp degrees)
.: 0.8181615399-1.5707963267i
```

Related

[tangent](#)

attach

Registers a knowledge base.

Syntax

(attach knowledge)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|------------------|
| knowledge | knowledge | 1 | A knowledge base |

Results

| Data Type | Description |
|-----------|----------------------------|
| knowledge | The opened knowledge base. |

Remarks

Enables lookup and modification of a knowledge base.

Example

```
> (var ?kb (knowledge url "file:///c:/premise/kb/defaultKb.mdb"))

.: Knowledge-1

> (attach ?kb)

.: Knowledge-1

> (relation Fact :All :Are)

.: Fact

> (detach ?kb)

.: Knowledge-1

> (free (cede (reference ?kb)))

.: freed
```

Related

conceive, detach, each, eradicate, get, knowledge, knowing, put, using, which, with

average

The arithmetic mean of a sequence.

Syntax

(average symbol ... binding sequence expression ...)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| symbol | symbol | 1+ | A symbol. |
| binding | literal | 1 | The literal in or per . |
| sequence | sequence | 1 | A sequence. |
| expression | expression | 1+ | any expression involving variables from the pattern(s) |

Results

| Data Type | Description |
|-----------|---|
| variant | The result of applying the plus function to the expression for the indicated range. |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to subelements of each subsequence. If there are insufficient elements to bind to the variables then the function fails. The last expression shall be evaluated and the average function shall be applied to both that expression and any prior result. The final value returned is the sum of all expressions divided by the length of the sequence.

Example

```
> (average ?x in (range 1 to 4) ?x)
.: 2.5

> (average ?fruit ?count per {{apple 1}{pear 2}{banana 3}{coconut 4}} ?count)
.: 2.5
```

Related

[fold](#), [product](#), [summation](#)

avg

The arithmetic mean.

Syntax

(avg value ...)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------------|
| value | variant | 1+ | An number or list of numbers. |

Results

| Data Type | Description |
|-----------|--|
| number | The arithmetic mean of all elements in the sequence. |

Remarks

Returns the arithmetic mean. If the list or expression is empty, it returns zero. If the only value is a list then all elements within the list are compared, otherwise, all subsequent values are compared

Example

```
> (avg {1 2 3 4 5})  
. : 3  
  
> (avg {})  
. : 0  
  
> (avg 1 2 3 4 5)  
. : 3  
  
> (avg {1 2 3} 4 {6 3 2})  
. : {3.66667 3 3}
```

Related

[max](#), [median](#), [min](#), [statistics](#), [sigma](#), [sum](#)

await

Returns the result of a task.

Syntax

(await *task* *timeout* *default*)

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--|
| task | task | 1 | A task resource |
| timeout | instant | 0-1 | The amount of time to wait for the task. The default is eternity which implies to wait indefinitely |
| default | variant | 0-1 | The value to return once the timeout is reached. If unspecified, then nil is returned. |

Results

| Data Type | Description |
|-----------|--------------------------|
| value | The task result, or nil. |

Remarks

The await function waits for either the timeout duration or the task to complete.

Example

```
> (concurrent (repeat 10000000 foo))  
.: Task-2  
> (await Task-2 \@t100000 timed-out)  
.: timed-out  
> (await Task-2)  
.: foo
```

Related

[wait](#), [busy-p](#), [complete](#), [do](#), [done-p](#), [free](#), [task](#), [tarry](#)

before-p

True if an interval finishes before a second interval.

Syntax

(before-p *interval₁* *interval₂* *tolerance*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------------------|-----------|-----|--|
| interval ₁ | interval | 1 | An interval |
| interval ₂ | interval | 1 | An interval |
| tolerance | instant | 0-1 | Amount of variation. (Defaults to zero ticks). |

Results

| Data Type | Description |
|-----------|---|
| truth | True if interval ₁ finishes some time before interval ₂ . |

Remarks

None.

Example

```
> (before-p  \@i{\@s1 \@s2}  \@i{\@s2 \@s4})  
.: false  
  
> (before-p  \@i{\@s1 \@s5}  \@i{\@s6 \@s8})  
.: true  
  
> (before-p  \@i{\@s1 \@s9}  \@i{\@s4 \@s6})  
.: false  
  
> (before-p  \@i{\@s4 \@s6}  \@i{\@s8 \@s9})  
.: true  
  
> (before-p  \@i{\@s4 \@s6}  \@i{\@s6 \@s9} \@s1)  
.: false
```

Related

during-p, finishes-p, meets-p, overlaps-p, starts-p

beginning

Returns the instant an interval begins.

Syntax

(beginning *interval*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| interval | interval | 1 | An interval |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| instant | The beginning value of the interval. |

Remarks

None.

Example

```
> (beginning \@i{\@s1 \@s2})  
. : \@s1  
> (beginning \@i{\@s1 \@s5})  
. : \@s1  
> (beginning \@i{\@s4 \@s6})  
. : \@s4
```

Related

[finishing](#)

best

Returns the best matching candidates.

Syntax

(best *probe* *candidates* *option ...* **)**

option ::= key value

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| probe | variant | 1 | The value to compare to the candidates |
| candidates | list | 1 | The candidates to be compared |
| option | expression | 0+ | <p>Key value pairs , where a key is one of:</p> <p>limit - The number of matches to return. (default is 1)</p> <p>minscore - value below which candidates shall not be considered. Default is ninfinity</p> <p>maxscore – Value above which candidates are acceptable. Default is 1.</p> <p>transform – The literal name of a value transformation function.</p> <p>comparer - The literal name of a comparison function</p> |

Results

| Data Type | Description |
|-----------|--|
| list | A sorted list of lists containing candidate - score pairs. |

Remarks

Best compares a probe to candidates and returns the best matching candidate(s) with match score(s). It uses the similarity function (\sim) as the default comparer. If provided, transforms are applied before comparison. Finally, a comparer can accept two values and returns a score representing the degree of similarity where higher means more similar.

Example

```
> (best "this" {"there" "that" "though" "thin"})  
. : {{"thin" 0.142857}}  
  
> (best 345 (range 1 to 1000000) limit 5 minscore -1 maxscore 1 comparer ~)  
. : {{345 1}{344 0.5}{346 0.5}{343 0.333333}{347 0.333333}}
```

Related

\sim similarity, $/\sim$ dissimilarity

beyond

True if a position is outside a sequence.

Syntax

(beyond *sequence position*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| sequence | sequence | 1 | A sequence. |
| position | variant | 1 | A number or coordinate (i.e. a list of numbers). |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if the position is not in the sequence, false otherwise. |

Remarks

None.

Example

```
> (beyond "not empty" 5)
.: false

> (beyond {not empty} 1)
.: false

> (beyond "" 1)
.: true

> (beyond {{a b}{c d}{e f}} {3 2})
.: false

> (beyond {{a b}{c d}{e f}} {7 4})
.: true
```

Related

@ *element, length, void, more, within*

big

Converts a value to a big number.

Syntax

(big *value* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 1 | A number value or the literal minimum or maximum |

Results

| Data Type | Description |
|-----------|--------------------|
| monadic | An monadic number. |

Remarks

A big number is bounded by memory. if the value cannot be converted to a big number, the function fails. The fractional part of the number is truncated. A suffix **b** denotes a big number.

Example

```
> (big {a b c})
.: [Failure :Name ArgumentValue :Text "{a b c} is not a number."]

> (big "a b c")
.: [Failure :Name ArgumentValue :Text "'a b c' is not a number."]

> (big 500.3)
.: 500b

> (big "23.4")
.: 23b

> (big
"2982039842304982039842187591823741209250293823978298234187235897234234237429387903
984203948203984209813719875209856087197129871")

.:
2982039842304982039842187591823741209250293823978298234187235897234234237429387903
84203948203984209813719875209856087197129871b
```

Related

[ceiling](#), [complex](#), [floor](#), [integer](#), [long](#), [radix](#), [rational](#), [real](#), [round](#), [truncate](#), [unsigned](#)

bind

Assigns variables to values in sequences or assortments.

Syntax

(bind *container assignment ...*)

assignment ::= slot symbol

assignment ::= function symbol

assignment ::= position symbol

assignment ::= {slot ...} as symbol

assignment ::= {function ...} as symbol

assignment ::= {position ...} as symbol

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| container | variant | 1 | A sequence or assortment |
| assignment | expression | 1+ | A key symbol, or position symbol, or list symbol pair. |

Results

| Data Type | Description |
|-----------|-------------------------|
| truth | Returns the value true. |

Remarks

The key may be a position, function, slot, method, list of functions, or other value.

Example

```
> (relation Node :Data :Next)
.: Node

> (new Node :Data C :Next (new Node :Data B :Next (new Node :Data A)))
.: Node_3

> (bind Node_3 :Data ?last {:Next :Next :Data} ?first)
.: true

> (bind {a b {c {d e}}} 1 ?a {3 2 2} ?e (given {?x} (uppercase ($ (@ ?x)))) ?a)
.: true
```

Related

dynamic, given, global, scope, set, tie

bindings

Returns a list of symbol value pairs for a pattern.

Syntax

(bindings *pattern option ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|---|
| pattern | variant | 1 | A premise or pattern |
| option | expression | 0+ | An option for retrieving the bindings. limit n - the number of results to return. |

Results

| Data Type | Description |
|-----------|----------------------------------|
| list | A list of all symbol value pairs |

Remarks

Returns a series containing binding lists that match the pattern. If no variables are in the pattern, then a premise is returned for each match and **nil** is specified for the symbol.

Example

```
> (relation Poll :Response)
.: Poll

> (step ?i from 1 to 1000
  (new Poll :Response (on (> (random 0 to 1) 0) Yes No)))

.: Poll_1000

> (for ?bindings in (bindings [Poll :Responses as ?response] limit 5)
  (tell user ($ ?bindings))

{?response yes} {?response no} {?response yes} {?response no} {?response no}
.: told

> (for ?bindings in (bindings [Poll :Responses yes] limit 4)
  (tell user ($ ?bindings))

{nil [Poll ^ Poll_1 :Response yes]} {nil [Poll ^ Poll_3 :Response yes]}
{nil [Poll ^ Poll_8 :Response yes]} {nil [Poll ^ Poll_9 :Response yes]}
.: told
```

Related

tally, using, which, with

bitwise

Performs bitwise operations.

Syntax

(**bitwise** *number* *op* ...)

op ::= *operation number*

op ::= **not**

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|---|
| number | number | 1 | A number |
| op | expression | 1+ | <p>an operation or operation value pair where the operation is either and, or, not, nand, nor, xor, <<, >>, or count and where the value is a number <i>y</i> such that:</p> <ul style="list-style-type: none">and <i>y</i> – performs logical andor <i>y</i> – performs logical ornot – performs complementnand <i>y</i> – performs logical nandnor <i>y</i> – performs logical norxor <i>y</i> – performs exclusive orxnor <i>y</i> – performs exclusive or<< <i>y</i> – performs left shift of <i>x</i> for <i>y</i> iterations>> <i>y</i> – performs right shift of <i>x</i> for <i>y</i> iterationscount <i>y</i> – returns the number of bits that = <i>y</i> (where <i>y</i> is either 1 or 0) |

Results

| Data Type | Description |
|-----------|----------------------------------|
| number | Result depends on the operation. |

Remarks

Each operation is performed in succession and the final result is returned.

Example

```
> (bitwise 2 and 3)
.: 2

> (bitwise 31 or 14)
.: 31

> (unsigned (bitwise 31 or 14 and 2))
.: 2

> (function countOneBits {?a}
  (bitwise ?a count 1))

.: countOneBits

> (countOneBits 255)
.: 8

> (bitwise 31 xor 14 count 1)
.: 2

> (function countConversionBits {?a ?b}
  (bitwise ?a xor ?b count 1))

.: countConversionBits
```

Related

+ *plus*, - *minus*, *unsigned*

bound

Returns the variables that are bound in a function.

Syntax

(bound *function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| function | variant | 1 | A function. |

Results

| Data Type | Description |
|-----------|----------------------|
| list | A list of variables. |

Remarks

None.

Example

```
> (function foo (?x ?y + ?z)
  (bar ?x ?y)
  (baz ?z ?a))

.: foo

> (bound foo)

.: {?x ?y ?z}

> (unbound foo)

.: {?a}

> (bound (' (given {?s} (@ ?s ?p)))))

.: {?s}

> (unbound (' (given {?s} (@ ?s ?p)))))

.: {?p}
```

Related

[unbound](#)

bound-p

True if a symbol has a value.

Syntax

(bound-p *symbol*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| symbol | symbol | 1 | A symbol. |

Results

| Data Type | Description |
|-----------|------------------------------|
| bool | True if the symbol is bound. |

Remarks

None.

Example

```
> (let ?a 1 ?b 2)
.: true
> (bound-p ?a)
.: true
> (bound-p ?b)
.: true
> (bound-p ?c)
.: false
```

Related

[unbound-p](#)

bracket

Returns 1 if a truth expression is true, 0 if false, and -1 if unknown.

Syntax

(bracket *truth*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------|
| truth | truth | 1 | A truth value |

Results

| Data Type | Description |
|-----------|---|
| integer | Returns 1 if the truth value is true, 0 if false, and -1 if unknown.. |

Remarks

None.

Example

```
> (bracket true)
.: 1
> (bracket false)
.: 0
> (bracket banana)
.: [Failure :Name InvalidType :Text "A truth was expected instead of banana."]
> (bracket unknown)
.: -1
```

Related

all, any, convert, every, nall, nevery, no, none, some,

break

Terminates a loop.

Syntax

(**break** *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---|
| value | variant | 0-1 | Any value to be returned. Default is nil. |

Results

| Data Type | Description |
|-----------|--|
| variant | Returns the value if provided, otherwise returns nil |

Remarks

Exits a loop with a return value or nil if no return value is specified. Can be used for a **do** loop.

Example

```
> (for ?item in (range 1 to 1000000)
    (if (= ?item 123456) (break))
    done)

.: nil

> (for ?item in (range 1 to 1000000)
    (if (= ?item 123456) (break foo))
    done)

.: foo
```

Related

[fail](#), [resume](#), [return](#), [try](#)

busy-p

True if a task has not completed.

Syntax

(**busy-p** *task*)

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|------------------|
| task | task | 1 | A task resource. |

Results

| Data Type | Description |
|-----------|------------------------------------|
| truth | True if the task has not completed |

Remarks

Returns true if the task has not completed, false otherwise. The opposite of this function is **ready-p**.

Example

```
> (let ?task (@ (concurrent (idle \@s100))))  
.: true  
  
> (do (idle \@s20)  
      (if (busy-p ?task)  
          running  
          else  
            (free ?task)))  
  
.: running
```

Related

cancel, cancelled-p, wait, await, complete, ready-p, reclaim, tarry, task

but

Creates a subsequence of all except the last elements.

Syntax

(**but** *sequence quantity*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| sequence | sequence | 1 | A list or string |
| quantity | number | 0-1 | The number of elements to omit. Defaults to 1. |

Results

| Data Type | Description |
|-----------|---------------------------|
| sequence | The selected sub sequence |

Remarks

If quantity is omitted, then all but the last item is returned. This is the opposite of **rest**.

Example

```
> (but {a b c d e f})  
.: {a b c d e}  
  
> (but {a b c d e f} 3)  
.: {a b c}  
  
> (but "abcde" 2)  
.: "abc"
```

Related

@ *element*, **last**, **rest**, **top**

bye

Terminates the interpreter.

Syntax

(bye)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|--|
| code | integer | 0-1 | An integer exit code. (Default is zero.) |

Results

None

Remarks

Terminates the read-eval-print loop.

Example

```
> (bye)
Bye!
```

Related

exit

call

Creates a call.

Syntax

(call *value* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | A value |

Results

| Data Type | Description |
|-----------|------------------------|
| call | The concatenated call. |

Remarks

Returns an unevaluated call. Note the call must have at least one value, preferably a literal. Note that an empty call cannot be invoked.

Example

```
> (call foo {})
.: (foo {})

> (call tell me "hello world")
.: (tell user "hello world"))

> (call)
.: [Failure :Name ArgumentRequired :Text "A required argument was not supplied."]

> (call given {?x} (+ ?x ?x))
.: (given {?x} (+ ?x ?x))
```

Related

\$, &, closure, eval, invoke, list, tuple, quote

can

Evaluates an expression while suppressing failures.

Syntax

(can expression result error)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|----------------------------|
| expression | variant | 1 | An expression to evaluate. |
| result | reference | 0-1 | A symbol reference. |
| error | reference | 0-1 | A symbol reference |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if no failure occurred; result and error shall contain respective values |

Remarks

If the expression succeeds, the result symbol shall contain the result the evaluated expression and the error symbol shall be **nil**. If the expression fails, the result shall be **nil** and the error symbol shall contain the failure.

Example

```
> (let ?result nil ?error nil)
.: true

> (can (string x) (reference ?result) (reference ?error))

.: true

> ?result ?error

.: "x" nil

.: (can (integer x) (reference ?result) (reference ?error))

.: false

> ?result ?error

.: nil [Failure :Name TypeConversion :Text "Cannot convert x to Integer."]
```

Related

assert, did, ensure, fail, finally, may, try

cancel

Cooperatively cancels tasks.

Syntax

(cancel *tasks* **)**

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---|
| tasks | variant | 1 | A single task resource or a list of task resources. |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| variant | Returns the tasks or list of tasks. |

Remarks

Cancels an executing task. The task shall be signaled to terminate. The task must call **cancelled-p** to determine whether to terminate, and if so, call **exit** or **return**.

Example

```
> (concurrent (step ?i from 1 to 100000 (if (cancelled-p (self)) (break oops))
finished)
              (step ?i from 1 to 100000 finished))

.: {Task-57 Task-58}

> (cancel task-57)

.: task-57

> (await task-57)

.: oops

> (abort task-58)

.: task-58

> (free {task-57 task-58})

.: freed
```

Related

[wait](#), [abort](#), [await](#), [cancelled-p](#), [complete](#), [critical](#), [do](#), [done-p](#), [reclaim](#) , [start](#), [task](#), [task-p](#), [tasks](#)

cancelled-p

True if a task has been cancelled.

Syntax

(**cancelled-p** *task*)

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|-----------------|
| task | task | 1 | A task resource |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| truth | True if the task has been cancelled. |

Remarks

The task may be cancelled using the **cancel** function. The cancel function sets a flag which must be recognized by the running task so that it can cooperatively terminate itself. Therefore all tasks should routinely invoke the **cancelled-p** function to determine whether they must terminate.

Example

```
> (concurrent (step ?i from 1 to infinity
    (if (cancelled-p (self)) (break quitting))
    finished))

.: {Task-57}

> (idle \@s5)

.: ready

> (cancel task-57)

.: cancelled

> (await task-57)

.: quitting
```

Related

wait, abort, await, cancel, cancelled-p, complete, do, reclaim, self, task

canonify

Makes equal values identical.

Syntax

(**canonify** *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0-1 | A thing.. |

Results

| Data Type | Description |
|-----------|-----------------------|
| variant | The canonified value. |

Remarks

Calling canonify with no arguments shall clear the canon of all stored values.

Example

```
> (global ?Original "The quick brown fox" ?Clone (clone ?Original))  
. : true  
> (= ?Original ?Clone)  
. : true  
> (identical-p ?Original ?Clone)  
. : false  
> (set ?Original (canonify ?Original) ?Clone (canonify ?Clone))  
. : true  
> (identical-p ?Original ?Clone)  
. : true
```

Related

=, clone, copy, make, identical-p, identity, same

capitalize

Capitalizes a string.

Syntax

(capitalize *string option* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| string | string | 1 | A string to capitalize |
| option | expression | 0-1 | A key value pair words <i>which-</i> where <i>which</i> is all or first (default). |

Results

| Data Type | Description |
|-----------|------------------------|
| string | The capitalized string |

Remarks

Capitalizes the string.

Example

```
> (capitalize "the quick brown fox")
.: "The quick brown fox"

> (capitalize "the quick brown fox" words all)
.: "The Quick Brown Fox"

> (capitalize "the quick brown fox" words first)
.: "The quick brown fox"
```

Related

[lowercase, uppercase](#)

case

Branches execution based on a value.

Syntax

(**case** *probe clause ... else-clause*)

clause ::= of value expression ...
clause ::= in {value ...} expression ...
else-clause ::= else expression ...

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|------------|-----|--|
| probe | variant | 1 | a value to compare |
| clause | expression | 1+ | Test value and expressions to be evaluated |
| else-clause | expression | 0-1 | A default case and expressions to be evaluated |

Results

| Data Type | Description |
|-----------|---|
| value | The last value in the successful clause |

Remarks

The first expression is evaluated and its result is matched against the **of** or **in** values. The comparison stops as soon as the first match is made. When that happens, the various expressions following the value are evaluated, one at a time. If no **of** or **in** expression is matched, then if an **else** clause exists, the expressions are evaluated, otherwise **nil** is returned.

Example

```
> (global ?Value a)
.: true

> (case ?Value
  of b nope
  in {c d} nope
  else not-found)

.: not-found
```

Related

and, if, not, or, unless

categorize

Returns a list of equivalence sets.

Syntax

(categorize sequence predicate ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|-------------------------|
| sequence | sequence | 1 | A sequence. |
| predicates | function | 1+ | Unary truth predicates. |

Results

| Data Type | Description |
|-----------|--|
| list | A list of sublists each containing elements partitioned by a predicate.. |

Remarks

A result sublist is returned for each predicate which determines if each item belongs in the respective equivalence set. If the predicate is true for the item, it is added to the result sublist.

Example

```
> (categorize (3 5 2 0 5 8 9) even odd zero)
.: {{even 2 0 8}{odd 3 5 5 9}{zero 0}}
> (categorize "abcdefghijklk" (given {?x} (< ?x "e")))
.: {{(given {?x} (< ?x "e")) "a" "b" "c" "d")}}
```

Related

[partition](#)

cede

Transfers a value between variables.

Syntax

(cede *reference* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--------------------|
| reference | reference | 1 | A symbol reference |

Results

| Data Type | Description |
|-----------|---|
| variant | the value formerly held by the symbol . |

Remarks

Transfers a value from one symbol to another, ensuring only one symbol is pointing to a value.

Example

```
> (function matrix+ {??a ??b}
  (ensure reference ??a reference ??b)
  (let ?r (cede ??a))
  (traverse ?i into ??b limit (dimensions ?r)
    (==> ?r + ?i (@ ??b ?i)) )
  (set ??b nil)
  (return ?r))

.: matrix+

> (let ?x (array {9000 9000} each (given {?c}(random 0 to 1)))
  ?y (array {9000 9000} each (given {?c}(random 0 to 1)))
  ?z (matrix+ (reference ?x) (reference ?y)))

.: true

> (is ?x null-p) (is ?y null-p) (is ?z null-p)
.: true true false
```

Related

copy, new, relation

ceiling

Rounds a number towards positive infinity.

Syntax

(ceiling number)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| number | The next higher positive integer. |

Remarks

Returns the next highest positive integer.

Example

```
> (ceiling 100.5)
.: 101
> (ceiling -100.5)
.: -100
```

Related

[floor](#), [round](#), [truncate](#)

char

Converts a Unicode value into a one position string.

Syntax

(char unicode)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|---|
| unicode | variant | 1 | A unicode value between 0 and 65535 or the literal maximum or minimum . |

Results

| Data Type | Description |
|-----------|------------------------------------|
| string | A string containing the character. |

Remarks

None.

Example

```
> (char 65)
.: "A"
```

Related

unicode

choose

Returns the elements of a sequence that satisfies a function.

Syntax

(choose sequence selector transform)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---|
| sequence | sequence | 1 | A sequence of values. |
| selector | function | 1 | The function to be applied. |
| transform | function | 0-1 | A function that returns a key given a sequence element prior to computing the selector. |

Results

| Data Type | Description |
|-----------|---|
| list | A list of elements satisfying the function. |

Remarks

Returns the elements in the sequence that satisfy the function. If the sequence is empty, it returns an empty list. The transform function takes elements of the sequence and returns numbers before computing the function of the transformed sequence.

Example

```
> (choose {1 2 3 4 5} max)
.: {5}

> (choose {a b c d e} max)
.: {e}

> (choose {} max)
.: {}

> (choose {{a 0}{b 1}{c 2}{d 0}} min (given {?item} (@ ?item 2)))
.: {{a 0}{d 0}}
```

Related

max, min, partition

clear

Eliminates all entries from an assortment or sequence.

Syntax

(**clear** *container*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|----------------------------|
| container | thing | 1 | An assortment or sequence. |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| thing | The modified assortment or sequence. |

Remarks

This function clears dynamic assortments and sequences. The **clear** function shall fail on fixed assortments (i.e. enumerations, records, and prototypes). For fixed assortments use **zap** to set slots values to **nil**.

Example

```
> (var ?s (lexicon a 1 b 2 c 3 d 4) ?list {a b c})  
.: {a b c}  
> (entries ?s)  
.: {{a 1}{b 2}{c 3}{d 4}}  
> (entries (clear ?s))  
.: {}  
> (clear ?list)  
.: {}  
> ?list  
.: {}
```

Related

[add](#), [includes](#), [item](#), [items](#), [keys](#), [reclaim](#), <#>, [tie](#), [values](#)

clip

Returns a value within a clipped range.

Syntax

(clip *number minimum maximum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|---|
| number | number | 1 | A number |
| minimum | number | 1 | A number (default is <code>ninfinity</code>) |
| maximum | number | 1 | A number (default is <code>infinity</code>) |

Results

| Data Type | Description |
|-----------|--|
| number | Returns the number constrained to the range. |

Remarks

If the value is greater than the maximum, the maximum is returned. If the value is less than the minimum, the minimum is returned, otherwise the value is returned.

Example

```
> (clip 3 0 5)
.: 3

> (clip 7 0 5)
.: 5

> (clip -4 0 5)
.: 0

> (clip 84 ninfinity 100)
.: 84

>(clip 84 0 infinity)
.: 84
```

Related

[max](#), [min](#)

clone

Clones an atom with possible modifications.

Syntax

(clone *atom modification ...* **)**

modification ::= key value

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------------|------------|-----|-------------------|
| atom | atom | 1 | An atomic value.. |
| modification | expression | 0+ | A key value pair. |

Results

| Data Type | Description |
|-----------|-------------|
| thing | The copy. |

Remarks

A duplicate or shallow copy of the original value is made. If the atom is a thought, a deep copy is made of all the slots. Zero or more modifications may be specified and applied after the value is cloned. If the value is a SubThought and the relation has an !onNew method defined, that method is first run on the clone. If an !onClone event method is defined, then the onClone method is invoked thereafter. The clone event (!onClone) requires two parameters {?clone ?original}. To clone sequences use the **copy** function.

Example

```
> (relation Car
    :Make      :Model      :Year      :Version 1
    !onClone  (function Car_clone {?m ?o} (==> ?m ++ :Version (:Version ?o)))
  
  .: Car

> (premise (new Car :Make  oole :Model civic :Year 2000))

.: [Car ^ Car_1 :Make  oole :Model civic :Version 1 :Year 2000 !onClone Car_clone]

> (premise (clone Vehicle_1 :Year 2016))

.: [Car ^ Car_2 :Make  oole :Model civic :Version 2 :Year 2016 !onClone Car_clone]
```

Related

[copy](#), [new](#), [relation](#)

close

Closes a file or data url.

Syntax

(**close** *url*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|---------------------|
| url | url | 1 | A file or data url. |

Results

| Data Type | Description |
|-----------|-------------|
| url | The url. |

Remarks

Closes an open url.

Example

```
> (var ?file (file name "kwikfox.txt"))

.: File-1

> (open ?file)

.: File-1

> (let ?contents (read ?file #))

.: true

> ?contents

.: "The quick brown fox jumped over the lazy dogs."

> (close ?file)

.: File-1
```

Related

[close](#), [data](#), [file](#), [open](#)

closure

Creates a public function in a module.

Syntax

(closure *scope type name parameters expression ...*)

```
parameters ::=  
parameters ::= {}  
parameters ::= {required ...}  
parameters ::= {required ... + optional ...}  
parameters ::= { + optional ...}  
parameters ::= {required ... + optional ... & remaining}  
parameters ::= { & remaining}
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-------------|-----|---|
| scope | environment | 0-1 | An environment or module name. |
| type | literal | 1 | The literal function or macro |
| name | literal | 0-1 | The name of the function. If not supplied then the name shall follow the pattern fn-<number> . |
| parameters | list | 0-1 | A list containing variables, keywords, sigils, and keyword symbol associations. |
| expression | variant | 1+ | One or more expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|--------------------------|
| literal | The name of the function |

Remarks

The closure intrinsic defines a function or macro with an explicit scope. A new scope shall be constructed for the function (or macro) using variables and function definitions contained in the ancestral scopes. If the name is supplied, then the function is a user named function (also called an **exonym**), otherwise it is an automatically named function (called an **esonym**). If a parameter list is supplied then it shall bind the formal parameter variables to any actual parameter expressions provided. The first variables in the parameter list are required. If a plus sign is provided, then those variables following the plus sign are optional. Each optional parameter may have a default value (specified with an equal sign and the value). If no default value is specified, the default value shall be **nil**. If an ampersand is provided, then the symbol following it is bound to a list of remaining (variadic) actual parameter values. Keyword parameters (literal symbol pairs) may be provided as required or optional. The entire parameter list may be omitted. The function shall be accessible from other modules by prefixing it with the module name in which it was defined.

Example

```
> (closure (scope) function {?n} (+ 1 ?n))  
. : fn-1  
  
> (closure (scope) function plusN {?n + ?i = 1} (+ ?n ?i))  
. : plusN  
  
> (fn-1 100)  
. : 101  
  
> (plusN 100)  
. : 101  
  
> (closure (scope go -1)  
  function tryParseNum {?s}  
  (let ?convertible (convertible-p ?s number))  
  {?convertible (if ?convertible (@ (take ?s))))})  
  
. : tryParseNum  
  
> (tryParseNum "foo")  
. : {false nil}  
  
> (tryParseNum "123")  
. : {true 123}  
  
> (closure (scope of User)  
  macro set2 {?value ?var1 ?var2}  
  "puts value into two variables in parallel"  
  (eval (` (set , ?var1 , ?value , ?var2 , ?value))))  
  
. : set2  
  
> (let ?x1 nil ?x2 nil) ?x1 ?x2  
. : true nil nil  
  
(set2 (+ 2 2) ?x1 ?x2)  
  
> ?x1 ?x2  
. : 4 4
```

Related

arguments, call, extend, function, macro, module, modules, parameters

coalesce

Returns a merged sequence.

Syntax

(coalesce symbol ... gate expression ...)

gate ::= in sequence

gate ::= per sequence

gate ::= from expression ... until predicate

gate ::= from expression ... while predicate

gate ::= from i limit j by k

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| symbol | symbol | 1+ | An iteration symbol |
| gate | expression | 1 | One of the gates expressions listed above. where i is the initial value of the symbol, limit is any of to , above , below or go . where j is the final value of the symbol, and k is the optional increment. |
| expression | variant | 1+ | An expression that potentially transforms the iteration symbol |

Results

| Data Type | Description |
|-----------|---|
| list | A new merged list after the transformation is applied |

Remarks

In the case of **in** or **per** For each element in the sequence, the element is stored in the symbol, and then the expressions are evaluated. If **from** is specified, each symbol is bound to the initially evaluated expression. If **to** is specified, the symbol is assumed to be a number, and the last number is included in the iteration. If **below** or **above** is specified the last number is excluded from the iteration. If **go** is specified, then the j value is the number of iterations. On each iteration the expressions are evaluated, and the last expression is collected into a merged result list.

Example

```
> (collect ?n in {1 2 3 4} {(* ?n 3) (* ?n 2)}))

.: {{3 2}{6 4}{9 6}{12 8}}


> (coalesce ?n from 1 while (<= ?n 4) {(* ?n 3) (* ?n 2)}))

.: {3 2 6 4 9 6 12 8}

> (collect ?x ?y per {{1 1}{1 2}{2 5}} {(/ ?x ?y) (* ?x y)})

.: {{1 1}{0.5 2}{0.4 10}}


> (coalesce ?x ?y per {{1 1}{1 2}{2 5}} {(/ ?x ?y) (* ?x y)})

.: {1 1 0.5 2 0.4 10}

> (coalesce ?n in {1 2 3 4} {(* ?n 3)}))

.: {3 6 9 12}

> (coalesce ?i from 1 to 9 by 2 {?i})

.: {1 3 5 7 9}

> (coalesce ?n from 1 below 5 {(* ?n 3)})

.: {3 6 9 12}

> (coalesce ?i from 1 go 4 (pick {a b c d e f g h i j} ?i))

.: {c h b e a c d j c f e b h j c}
```

Related

collect, filter, map , reduce, scale

collect

Returns a transformed sequence.

Syntax

(**collect** *symbol* ... *gate expression* ...)

gate ::= in sequence

gate ::= per sequence

gate ::= from expression ... until predicate

gate ::= from expression ... while predicate

gate ::= from i limit j by k

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| symbol | symbol | 1+ | An iteration symbol |
| gate | expression | 1 | One of the gates expressions listed above. where i is the initial value of the symbol, limit is any of to , above , below or go . where j is the final value of the symbol, and k is the optional increment. |
| expression | variant | 1+ | An expression that potentially transforms the iteration symbol |

Results

| Data Type | Description |
|-----------|--|
| list | A new list after the transformation is applied |

Remarks

In the case of **in** or **per** For each element in the sequence, the element is stored in the symbol, and then the expressions are evaluated. If **from** is specified, each symbol is bound to the initially evaluated expression. If **to** is specified, the symbol is assumed to be a number, and the last number is included in the iteration. If **below** or **above** is specified the last number is excluded from the iteration. If **go** is specified, then the j value is the number of iterations. On each iteration the expressions are evaluated, and the last expression is collected into a result list.

Example

```
> (collect ?n in {1 2 3 4} (* ?n 3))
.: {3 6 9 12}

> (collect ?x ?y per {{1 1}{1 2}{2 5}} (/ ?x ?y))
.: {1 0.5 0.4}

> (collect ?i from 1 to 9 by 2 ?i)
.: {1 3 5 7 9}

> (collect ?n from 1 below 5 (* ?n 3))
.: {3 6 9 12}

> (collect ?i from 1 go 4 (pick {a b c d e f g h i j} ?i))
.: {{c} {h b} {e a c} {d j c f} {e b h j c}}
```

Related

[coalesce](#), [filter](#), [map](#), [reduce](#), [scale](#)

collection

Returns an unorderd collection of elements.

Syntax

(collection *element ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|-------------|
| element | variant | 0+ | A key. |

Results

| Data Type | Description |
|------------|---------------|
| collection | A collection. |

Remarks

A collection is an unordered set of elements.

Example

```
> (collection
    {a b c} "foo" bar 500 [Fruit :Name Apple])
.: Collection-1

> (has Collection-1 foo)
.: false

> (has Collection-1 "foo")
.: true
```

Related

enumeration, environment, lexicon

combine

Creates a new assortment by combining entries.

Syntax

(combine *assortment entry ...* **)**

entry ::= key value

entry ::= key

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---------------------------|
| assortment | assortment | 1 | An assortment or sequence |
| entry | expression | 1+ | A key, or key value pair. |

Results

| Data Type | Description |
|------------|--------------------------|
| assortment | The modified assortment. |

Remarks

Collections require only a key for each entry. Assortments require a key and value for each entry. Use the **add** function to modify an assortment.

Example

```
> (entries (add (lexicon) a 1 b 2 c 3))  
.: {{a 1}{b 2}{c 3}}  
> (entries (add (collection) 1 2 3 4 5))  
.: {{1 1}{2 2}{3 3}{4 4}{5 5}}
```

Related

[add](#)

common

Creates a sequence of common elements among subsequences.

Syntax

(**common** *sequences*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---------------------------------|
| sequences | sequence | 1 | A sequence containing sequences |

Results

| Data Type | Description |
|-----------|------------------------|
| sequence | A list of common items |

Remarks

Returns the intersection of the subsequences, i.e., the elements common to all subsequences.

Example

```
> (common {{1 2 3 4} {2 3 4 5}})  
.: {2 3 4}  
  
> (common {{2 3 4 5} {4 5 6 7}})  
.: {4 5}  
  
> (common (map {[Data :A {1 2 3 4}] [Data :A {2 3 4 5}]} (given {?d} (:A ?d))))  
.: {4}  
  
> (common {"abc" "bcd" "cde"})  
.: "c"
```

Related

[difference](#), [intersection](#), [union](#)

comparable-p

Returns true if the values can be compared.

Syntax

(**comparable-p** *value₁* *value₂*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 2 | A value |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if the values can be compared. |

Remarks

Returns **true** if the values are of the same type and their elements are of the same type, false otherwise.

Example

```
(comparable-p 1 2)
.: true

(comparable-p 2 x)
.: false

(comparable-p {1 2 3 4}{2 3 4 5})
.: true

(comparable-p {1 2 3 4}{a b c d})
.: false

(comparable-p b a)
.: true

(comparable-p z {a b c})
.: false
```

Related

=equal, /=unequal

compare

Returns less, equal, or greater.

Syntax

(compare *value₁* *value₂*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 2 | A value |

Results

| Data Type | Description |
|-----------|---|
| literal | Returns < (less than), = (equals), or > (greater than). |

Remarks

Returns the less than literal if the first value is less than the second value, the equal literal if the values are equal, and the greater than literal if the first value is greater than the second value. The function fails if the values are not comparable (i.e., of the same type).

Example

```
(compare 1 2)
.: <
(compare 2 1)
.: >
(compare {1 2 3 4}{2 3 4 5})
.: <
(compare a a)
.: =
(compare b a)
.: >
```

Related

=equal, /=unequal

complete

Runs expressions concurrently.

Syntax

(complete *expression* ... **)**

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--------------------------------|
| expression | variant | 1+ | An expression to be evaluated. |

Results

| Data Type | Description |
|-----------|---------------------------|
| list | A list of task resources. |

Remarks

Executes the expressions concurrently, and returns after the last expression completes, returning a list of task handles. If expression is not a task, then the expression is converted to a task prior to execution.

Example

```
> (complete
    (step ?i from 0 to 50000000 (do nothing))
    (step ?i from 1 to 10000000 (do something)))))

.: {task-1 task-2}

> (free (wait {task-1 task-2}))

.: freed
```

Related

abort, await, concurrent, do, busy-p, free, ready-p, start, tarry, task, tasks, wait

complex

Converts a value to a complex number.

Syntax

(complex *real* *imaginary*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--|
| real | real | 1 | A number value or the literal minimum or maximum |
| imaginary | imaginary | 1 | A number value or the literal minimum or maximum |

Results

| Data Type | Description |
|-----------|-------------------|
| complex | A complex number. |

Remarks

If the values cannot be converted to a complex number, the function fails.

Example

```
> (complex {a b c})  
. : [Failure :Name ArgumentValue :Text "{a b c} is not a number."]  
  
> (complex "a b c")  
. : [Failure :Name ArgumentValue :Text "'a b c' is not a number."]  
  
> (complex 500 8i)  
. : 500+8i  
  
> (complex 500.3 0)  
. : 500.3+0i  
  
> (complex "23.4" "0i")  
. : 23.4+0i
```

Related

[ceiling](#), [complex](#), [floor](#), [long](#), [rational](#), [real](#), [round](#), [truncate](#),

compose

Applies functions in reverse order using the arguments.

Syntax

(**compose** *functions arguments*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---|
| functions | list | 1 | A list of functions to be applied in reverse order. |
| arguments | variant | 0-1 | The arguments to the last function in the functions list. |

Results

| Data Type | Description |
|-----------|-----------------------------|
| value | The result of the last call |

Remarks

If the arguments parameter is an empty list, the rightmost function is assumed to be niladic (requiring no arguments). The rightmost function is applied to the arguments parameter, then the second from rightmost is applied to the result, and so on, until all functions are applied. The final result is returned. The opposite function is **morph**. The **scatter** function applies functions in parallel.

Example

```
> (function double {?x} (* ?x 2))

.: double

> (compose {double ++ ++ ++} {0})

.: 6

> (compose {rest but rest but}
            {{the quick brown fox jumped over the lazy dogs}})

.: {brown fox jumped over the}

> (compose {double ++ ++ +} {2 3 4})

.: 22
```

Related

apply, map, morph, scatter

conceive

Creates a knowledge base.

Syntax

(conceive *knowledge*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|------------------|
| knowledge | knowledge | 1 | A knowledge base |

Results

| Data Type | Description |
|-----------|----------------------------|
| knowledge | The opened knowledge base. |

Remarks

Constructs a new knowledge base if one does not already exist.

Example

```
> (conceive (var ?kb (knowledge name defaultKB path "c://premise/kb/")))

.: defaultKB

> (attach ?kb)

.: defaultKB

> (relation Fact :All :Are)

.: Fact

> (detach ?kb)

.: defaultKB

> (free (cede (reference ?kb)))

.: freed
```

Related

attach, detach, each, eradicate, get, knowing, knowledge, put, using, which, with

concurrent

Evaluates expressions asynchronously.

Syntax

(concurrent *expression ...* **)**

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--------------------------------|
| expression | variant | 1+ | An expression to be evaluated. |

Results

| Data Type | Description |
|-----------|-------------------------------|
| list | The executing task resources. |

Remarks

If expression is not a task, then the expression is converted to a task. Each task is then started asynchronous and a list of the executing tasks is returned.

Example

```
> (concurrent (+ 1 2 3 4) (/ 1000 57) (** 4 20))
.: {Task-1 Task-2 Task-3}

> (await task-1)
.: 10

> (await task-2)
.: 17.54385964912281

> (await task-3)
.: 1099511627776

> (free task-1 task-2 task-3)
.: nil

> (free (tarry (concurrent (+ 1 2 3 4) (/ 1000 57) (** 4 20))))
.: freed
```

Related

await, busy-p, cancel, cancelled-p, conclude, critical, is, free , ready-p, self, tarry, task, tasks, wait

configuration

Creates a configuration file.

Syntax

(configuration url association ...)

association ::= section settings

settings ::= { setting setting }

setting ::= key value

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|------------|-----|--|
| url | url | 1 | A url. |
| association | expression | 0+ | A section literal and list of key value pairs. |

Results

| Data Type | Description |
|-----------|------------------------------------|
| value | The value associated with the key. |

Remarks

This function creates a configuration file at the specified URL. The file constis of sections that have key value pairs.

Example

```
> (configuration "c$/apps/premise/myApp.cfg" security {User John password "foo"})
.: :Configuration-1
> (get Configuration-1 security password)
.: "foo"
> (put (get Configuration-1 security) password "bar")
.: Lexicon-248
> (get Configuration-1 security password)
.: "bar"
```

Related

[add](#), [cut](#), [erase](#), [get](#), [put](#)

confirm

Tests that a condition is true.

Syntax

(confirm *condition* **since** *reason*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--|
| condition | truth | 1 | The test. If it evaluates to false the function fails |
| since | literal | 0-1 | The literal since. |
| reason | string | 0-1 | The failure text to display. Defaults to "Assertion failure." if not supplied. |

Results

| Data Type | Description |
|-----------|-------------------------------|
| truth | Returns true if test is true. |

Remarks

If the since keyword and reason are omitted, the failure text shall be "Failed condition." The failure name is always "Confirmation".

Example

```
> (confirm (= red blue) since "colors must be the same")
.: [Failure :Name Assertion :Text "colors must be the same"]

> (function reducing {?sequence ?function}
  (ensure sequence ?sequence function ?function)
  (confirm (> (# ?sequence) 1) since "Two or more elements are required.")
  (let ?result (@ ?sequence))
  (for ?element in (rest ?sequence) (--> ?function ?result ?element)))

.: reducing

> (reducing {1} +)

.: [Failure :Name Confirmation :Text "Two or more elements are required."]
```

Related

confute, ensure, fail, signal, try

confute

Tests that a condition is false.

Syntax

(confute condition since reason)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--|
| condition | truth | 1 | The test. If it evaluates to false the function fails |
| since | literal | 0-1 | The literal since . |
| reason | string | 0-1 | The failure text to display. Defaults to "Rejection failure." if not supplied. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | Returns false if test is false. |

Remarks

If the since keyword and reason are omitted, the failure text shall be "Failed condition." The failure name is always "Confutation."

Example

```
> (confute (= red blue) since "colors must be different")
.: false

> (function reducing {?sequence ?function}
  (ensure sequence ?sequence function ?function)
  (confute (<= (# ?sequence) 1) since "Two or more elements are required.")
  (let ?result (@ ?sequence))
  (for ?element in (rest ?sequence) (--> ?function ?result ?element)))

.: reducing

> (reducing {1} +)
.: [Failure :Name Confutation :Text "Two or more elements are required."]
```

Related

confirm, ensure, fail, signal, try

constant

Creates a constant.

Syntax

(constant *assignment* ... **)**

assignment ::= *symbol* *value*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|-------------------------|
| assignment | expression | 1+ | Symbol and value pairs. |

Results

| Data Type | Description |
|-----------|-------------------------|
| truth | Returns the value true. |

Remarks

Creates constant variables in the current environment. Any attempts to put a different value into the symbol shall create a failure. Calling **constant** again on the variables is allowed.

Example

```
> (constant ?SECRET "mysecret")
.: true
> ?SECRET
.: "mysecret"
> (constant ?MIN_ENTRIES 300
            ?MAX_ENTRIES 400)
.: true
> ?MAX_ENTRIES
.: 400
```

Related

<-- before tie, --> after tie, global, dynamic, modular, local

contains

True if a value is present in an assortment.

Syntax

(**contains** *assortment value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--------------------------|
| assortment | assortment | 1 | A sequence or assortment |
| value | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|--|
| truth | Returns true if the key is present. or false if the key is absent. |

Remarks

None

Example

```
> (lexicon :A 1 :B 2 !m X_foo :C 3)
.: Lexicon-1

> (contains Lexicon-1 3)
.: true

> (contains Lexicon-1 X_foo)
.: true

> (contains Lexicon-1 elephant)
.: false
```

Related

has, lacks, missing

continue

Continues to the next iteration of a loop.

Syntax

(**continue** *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 0-1 | A return value for the next iteration. |

Results

| Data Type | Description |
|-----------|--|
| value | Returns the value if provided, otherwise returns nil |

Remarks

If this is executed within a loop, then the next iteration resumes until the loop is completed. If this is executed outside of a loop, then the value is returned. If no value is supplied, nil is returned.

Example

```
> (for ?item in (range 1 to 1000000)
   (if (< ?item 123456)
       (continue)
     else
       (break)))

.: nil

> (for ?item in (range 1 to 1000000)
   (if (< ?item 123456)
       (continue)
     else
       (break foo)))

.: foo
```

Related

[break](#), [fail](#), [return](#), [try](#)

convertible-p

True if the value can be converted to the datatype.

Syntax

(convertible-p *value taxon*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|----------------------------------|
| value | variant | 1 | The value to be converted |
| taxon | literal | 1 | The name of the target data type |

Results

| Data Type | Description |
|-----------|--|
| truth | Returns true if the value can be converted to the data type. |

Remarks

If the value can be converted converted to the given data type, then this function returns true, otherwise it returns false.

Example

```
> (convertible-p {a b c} string)
.: true

> (convertible-p "500" number)
.: true

> (convertible-p "500" literal)
.: false

> (convertible-p "{wxy}" number)
.: false
```

Related

[datatype](#), [datais](#), [relation-p](#), [type](#), [is](#)

copy

Copies a sequence.

Syntax

(copy *sequence count*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | sequence | 1 | A sequence |
| count | integer | 0-1 | The number of times to repeat the sequence. Default is 1. |

Results

| Data Type | Description |
|-----------|----------------------|
| sequence | The copied sequence. |

Remarks

A shallow copy of the original sequence is made and replicated. All elements of the sequence are repeated the specified number of times.

Example

```
> (copy "a" 5)
.: "aaaaa"
> (copy {the quick brown fox} 3)
.: {the quick brown fox the quick brown fox the quick brown fox}
> (copy "hello")
.: "hello"
> (copy ($ hello world \s) 2)
.: "hello world  hello world"
```

Related

[copy](#), [new](#), [relation](#)

copyright

Displays copyright information.

Syntax

(copyright)

Module

Base

Parameters

None

Results

| Data Type | Description |
|-----------|----------------|
| nil | The value nil. |

Remarks

Prints copyright information.

Example

```
> (copyright)
(c) 2014-2020 Michael S P Miller  All Rights Reserved
.: nil
```

Related

[bye](#), [help](#), [license](#)

correlate

Finds the correlation coefficient of two lists.

Syntax

(correlate *list₁* *list₂*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|-------------|
| list | list | 2 | Lists. |

Results

| Data Type | Description |
|-----------|------------------------------|
| number | The correlation coefficient. |

Remarks

Compares only the number of elements in the shorter of the two lists.

Example

```
(correlate {1 2 3 4 5} {4 3 2 1 0})  
. : -1  
  
(correlate {1 2 3 4 5} {6 7 8 9 10})  
. : 1  
  
(correlate {1 2 3 4 5} {6 3 8 2 10})  
. : 0.33071891
```

Related

probability, statistics, survey

cosecant

Returns the cosecant.

Syntax

(**cosecant** *number geometry metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|-----------------------------|
| number | The cosecant of the number. |

Remarks

Example

```
> (cosecant 0.785398)
.: 1.414213793

> (cosecant cir 0.785398 radians)
.: 1.414213793

> (cosecant cir 45 degrees)
.: 1.4142135623

> (cosecant hyp 1.5708 radians)
.: 0.434535467

> (cosecant hyp 90 degrees)
.: 0.434537208
```

Related

asecant, secant

cosine

Returns the cosine.

Syntax

(cosine *number geometry metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|---------------------------|
| number | The cosine of the number. |

Remarks

None.

Example

```
> (cosine 0.785398)
.: 0.707106896

> (cosine cir 0.785398 radians)
.: 0.707106896

> (cosine cir 45 degrees)
.: 0.707106896

> (cosine hyp 1.5708 radians)
.: 2.50918693

> (cosine hyp 90 degrees)
.: 2.509178478
```

Related

asine, sine

cotangent

Returns the cotangent.

Syntax

(cotangent *number* *geometry* *metrum*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|------------------------------|
| number | The cotangent of the number. |

Remarks

Example

```
> (cotangent 0)
.: infinity

> (cotangent cir 1.5708 radians)
.: 0

> (cotangent cir 45 degrees)
.: 1

> (cotangent hyp 1.5708 radians)
.: 1.09033

> (cotangent hyp 90 degrees)
.: 1.0903314107
```

Related

[atangent](#), [tangent](#)

count

Counts the iterations and returns the last expression.

Syntax

(**count** *symbol* ... *binding sequence as counter expression ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , or over . |
| sequence | sequence | 1 | A sequence (string or list) |
| as | literal | 1 | The literal as |
| counter | symbol | 1 | A counter symbol |
| expression | expression | 0+ | An expression. |

Results

| Data Type | Description |
|-----------|--|
| variant | Returns the last expression in the sequence. |

Remarks

None.

Example

```
> (count ?word in {the quick brown fox} as ?n
    (tell user ($ ?n ?word \n)) ?n)
1 the
2 quick
3 brown
4 fox
.: 4
```

Related

collect, exactly, filter, gather, quantify, quantity

critical

Serializes evaluations across regions of code.

Syntax

(critical *locks* *option ... expression ...*)

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| locks | list | 0-1 | A list of literals representing critical section locks |
| option | expression | 0+ | A key value pair: retries <i>integer</i> - number of times to retry any lock. (default is 0) lockwait <i>time</i> - how long to wait for a lock. (default is 2,000,000 ticks) within <i>time</i> - the overall time to obtain all locks (default is 1 second) |
| expression | variant | 1+ | An expression to be evaluated. |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| variant | The value of the last expression. |

Remarks

The critical section is used to serialize the execution of the contained expressions. Any other sections with the same tags shall be blocked. Generates failure if the locks cannot be obtained on the tags under the default or indicated conditions, or if a lock was forcibly released during execution.

Example

```
> (function sayHello {}
  (critical {printing}
    (for ?c in "hello " (tell user ?c)))

.: sayHello

> (free (wait (concurrent (sayHello) (sayHello) (sayHello)))))

hello hello hello .: freed
```

Related

[release](#)

cut

Removes elements from assortments by key.

Syntax

(**cut** *assortment* *key* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|----------------|
| container | thing | 1 | An assortment. |
| key | variant | 0+ | A key. |

Results

| Data Type | Description |
|------------|--------------------------------------|
| assortment | The modified assortment or sequence. |

Remarks

Assortments are modified destructively. To delete nondestructively use the **remove** function.

Example

```
> (entries (collection a b c d))
.: {{a a} {b b} {c c} {d d}}
> (entries (cut (lexicon a 1 b 2 c 3 d 4) b d))
.: {{a 1}{c 3}}
```

Related

[rip](#), [remove](#)

data

Creates a data url.

Syntax

(data option ...)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|---|
| option | expression | 1+ | <p>Any key value pair of the following:</p> <p>connect <i>string</i> - use the specified connection string. poolszie <i>poolszie</i> - maximum # of connections provider <i>platform</i> - either mssql, odbc, maria, pgsql</p> <p>or build the connection string from the following elements.</p> <p>driver <i>name</i> - the driver server <i>address</i> - server address version <i>number</i> - server version class <i>class</i> - class name of driver host <i>address</i> - host address port <i>port</i> - server port for database service instance <i>instance</i> - server instance name integratedSecurity <i>truth</i> - true or false database <i>name</i> - database name user <i>username</i> - user name password <i>password</i> - user password trusted <i>yesorno</i> - trusted connection yes or no. source <i>name</i> - data source name protocol <i>protocol</i> - database protocol subprotocol <i>subprotocol</i> - database sub protocol subname <i>subname</i> - sub name</p> |

Results

| Data Type | Description |
|-----------|-------------|
| url | A data url. |

Remarks

None

Example

```
> (open (data driver sqlserver server "//servername" port 1433
           database mydata user "myUid" password "123456"))

.: Data-1

> (ask Data-1 "select top 2 empid, name, salary from employee"
           format row columns {name wage})

.: {{1 "John Smith" 10.50}{2 "Jane Johnson" 12.00} }

> (close Data-1)

.: Data-1

> (open (data host "localhost" database "sample" user "username"
           password "secret" connections 3))

.: Data-2

> (tell Data-2 "update employee set wage = wage + 1.00;")

.: executed

> (close Data-2)

.: Data-2

> (global ?data
      (open (data class "com.microsoft.jdbc.sqlserver.SQLServerDriver"
              subprotocol "sqlserver"
              subname "//serverName:port;database=db;user=uid;password=pwd")))

.: true

> (function valueslist {?rows}
      ($ (interior(infix (map (collect ?row in ?rows (interior(infix ?row ,)))
                                call)
                  , ))))

.: valuelist

> (tell ?data ($ "insert Employee (name, hours, wage) values "
      (valueslist {"Jane Johnson" 40 10.50}
                  {"John Smith" 5 10.50})))

.: 2

> (close ?data)

.: Data-3
```

Related

[disconnect, tell, fetch, store](#)

date

Creates a new date or returns the current date.

Syntax

(date yr mth day hrs mins secs zone zmins)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| yr | integer | 0-1 | Four digits: The year (e.g., 2018, 2024) |
| mth | integer | 0-1 | A number between 1 and 12 |
| day | integer | 0-1 | A number between 1 and 31 |
| hrs | integer | 0-1 | A number between 0 and 23 |
| mins | integer | 0-1 | A number between 0 and 59 |
| secs | real | 0-1 | A real number between 0 and 59 |
| zone | integer | 0-1 | A number between -12 and 12. |
| zmins | integer | 0-1 | A number between -12 and 12. |

Results

| Data Type | Description |
|-----------|---------------|
| date | The UTC date. |

Remarks

If no arguments are provided, this function returns the current date in Coordinated Universal Time. Otherwise, it attempts to construct a date from the supplied arguments.

Example

```
> (date)
.: \@d2014-09-16T02:15:51.97152+00:00
> (date 2018 5 18 0 0 0 0 0)
.: \@d2018-05-18T00:00:00.00000+00:00
```

Related

epoch, moment, tick

declared-p

True if a symbol exists in an environment.

Syntax

(**declared-p** *symbol environment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|-----------------|
| symbol | symbol | 1 | A symbol |
| environment | environment | 0-1 | An environment. |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| truth | True if the symbol has been declared. |

Remarks

If location is omitted, the symbol is sought in the current environment.

Example

```
> (extend Example
  (modular ?hello  World))

.: bar

> (declared-p ?User.hello)

.: false

> (declared-p ?Example.hello)

.: true
```

Related

function-p, symbol-p,

decode

Decodes a string.

Syntax

(decode *source format*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--|
| source | string | 1 | A string to decode. |
| format | literal | 1 | One of { base16 , base32 , base64 , utf8 , none } |

Results

| Data Type | Description |
|-----------|--------------------|
| string | The encoded string |

Remarks

Each entry in the elements list may be a literal or a list containing a literal and an index.

Example

```
> (encode "The quick brown fox" base16)
.: "54686520717569636B2062726F776E20666F78"

> (encode "The quick brown fox" base64)
.: "VGhlIHF1aWNrIGJyb3duIGZveA=="

> (encode "The quick brown fox" none)
.: "The quick brown fox"

> (decode "54686520717569636B2062726F776E20666F78" base16)
.: "The quick brown fox"

> (decode "VGhlIHF1aWNrIGJyb3duIGZveA==" base64)
.: "The quick brown fox"

> (decode "The quick brown fox" none)
.: "The quick brown fox"
```

Related

[encode](#)

default

Returns the first non-nil value.

Syntax

(default *value* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | variant | 1+ | A value to be returned. |

Results

| Data Type | Description |
|-----------|----------------------------------|
| value | The first value that is not nil. |

Remarks

Returns the first value which is not nil. If all values are nil, then nil is returned.

Example

```
> (default nil 1 2 3)
.: 1

> (default nil a b c)
.: a

> (default nil)
.: nil

> (default nil nil nil nil 1)
.: 1
```

Related

any, is , no

definitions

Retrieves literal value pairs in an environment.

Syntax

(definitions *environment* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|----------------|
| environment | environment | 1 | An environment |

Results

| Data Type | Description |
|-----------|----------------------------------|
| list | A list of all symbol value pairs |

Remarks

Returns a list containing all literal value pairs.

Example

```
> (using User
    (definitions (scope)))
.: {}

> (extend User
    (function foo {} hello)
    (function bar {} world))

.: User

> (using User
    (definitions (scope)))
.: {{foo "(function foo {} hello)"}
 {bar "(function bar {} world)"}}
```

Related

[variables](#)

defunct

Undefines a function in a module.

Syntax

(**defunct** *function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|----------------------------------|
| function | literal | 1 | A fully qualified function name. |

Results

| Data Type | Description |
|-----------|---------------------------|
| literal | The name of the function. |

Remarks

Undefines a function or causes a failure if the function does not exist. Returns the name of the function if successful. Looks for the function in the current module if module is omitted.

Example

```
> (function foo bar)
.: foo

> (describe foo)
.: {"(function User.foo {} bar)"}

> (defunct foo)
.: foo

> (describe foo)
.: nil
```

Related

[defined](#), [describe](#), [function](#), [macro](#), [nix](#), [relation](#)

degrees

Converts radians into degrees.

Syntax

(degrees radians)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|----------------------|
| radians | real | 1 | a number in radians. |

Results

| Data Type | Description |
|-----------|------------------------|
| number | The number of degrees. |

Remarks

The radians are multiplied by $180 / \pi$, or roughly 57.29578049044297, for the conversion.

Example

```
> (round (degrees (* (/ 4 9) (pi))))  
. : 80  
> (round (degrees 1.4))  
. : 80  
> (round (degrees 0.7864815))  
. : 45
```

Related

cosine, radians, sine, tangent

delete

Modifies a sequence by deleting values.

Syntax

(**delete** *sequence value ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence. |
| value | variant | 0+ | A value. |

Results

| Data Type | Description |
|------------|-------------------------|
| assortment | The modified assortment |

Remarks

Destructively modifies a sequence by deleting the values. To create a new sequence with values omitted use the **omit** function.

Example

```
> (delete {a b c d e f g} a b)
.: {c d e f g}

> (delete {a 1 nil b 2 c 3 nil nil d  nil 4} nil)
.: {a 1 b 2 c 3 d 4}
```

Related

[pop](#), [push](#),

dependencies

Returns a module's dependencies.

Syntax

(dependencies *module*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| module | literal | 1 | A module |

Results

| Data Type | Description |
|-----------|--|
| list | A list of modules upon which the argument depends. |

Remarks

Returns dependencies of a module.

Example

```
> (module a
  (function foo bar))

.: a

> (module b
  (function bar baz))

.: b

> (module c
  (require a)
  (require b)
  (function baz {}
    {(foo) (bar)}))

.: c

> (dependencies c)
.: {a b}
```

Related

discard, extend, forgo, module, require, grok, using

deq

Removes an element from an embedded sequence.

Syntax

(**deq** *container place option*)

option ::= all value(s)

option ::= at position

option ::= the value(s)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|--|
| container | thing | 1 | An assortment or sequence. |
| place | variant | 1 | A key or position. |
| option | expression | 0-1 | One of all value(s) - Removes all occurrences of the remaining value. at position - A number or the literal #. Default is 1. each value(s) - Removes the first occurrence of each value. |

Results

| Data Type | Description |
|-----------|------------------------|
| variant | Returns the container. |

Remarks

If no options are provided, the last item of the sequence is removed.

Example

```
> (deq (lexicon 1 {a a b} 2 {d e f}) each 2)
.: Lexicon-1
> (get Lexicon-1 2)
.: {d e}
> (get (deq Lexicon-1 1 all a x y) 1)
.: {b}
```

Related

add, cut, enq, get, has, key, keys, peek, put, values

describe

Returns descriptions of a literal.

Syntax

(describe *literal*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--|
| literal | literal | 1 | A function, module, relation, or other defined item. |

Results

| Data Type | Description |
|-----------|--|
| list | A possibly empty list of descriptions. |

Remarks

Returns a list of string descriptions.

Example

```
> (describe foo)
.: {}

> (relation Foo :Bar :Baz)
.: Foo

> (function foo {} bar)
.: foo

> (describe foo)
.: {"(function User.foo {} bar)" "(relation Foo :Bar nil :Baz nil)"}  

```

Related

[defined](#), [defunct](#), [function](#), [macro](#), [nix](#), [relation](#)

detach

Unregisters a knowledge base.

Syntax

(**detach** *knowledge*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|-------------------|
| knowledge | knowledge | 1 | A knowledge base. |

Results

| Data Type | Description |
|-----------|---------------------|
| knowledge | The knowledge base. |

Remarks

Unregisters an attached knowledge base.

Example

```
> (var ?kb (knowledge url "file:///c:/premise/kb/esoteric.mdb"))

.: Knowledge-1

> (attach ?kb)

.: Knowledge-1

> (use ?kb)

.: Knowledge-1

> (relation Fact :All :Are)

.: Fact

> (detach ?kb)

.: Knowledge-1

> (free (cede (reference ?kb)))

.: freed
```

Related

attach, conceive, each, eradicate, knowledge, knowing, get, put, using, which, with

difference

Returns the difference of two sequences.

Syntax

(difference sequence₁ sequence₂ operation)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---|
| sequence | sequence | 2 | A list or string |
| operation | literal | 0-1 | One of : set - returns the set difference. (default) sym - returns symmetric difference |

Results

| Data Type | Description |
|-----------|---|
| list | Set difference of sequence1 and sequence2 |

Remarks

If the operation is **set**, this function returns a sequence containing elements from the first sequence not in the second. If the operation is **symmetric**, it returns items that are not in the intersection of the provided sequences. Sequences must be of the same data type, e.g., either all lists or all strings.

Example

```
> (difference "abcd" "abef" set)
.: "cd"
> (difference {1 2 3 4} {2 3 4 5} sym)
.: {1 5}
> (difference {1 2 3} {5 4 3 2})
.: {1}
```

Related

common, intersection, union

did

Evaluates an expression and suppresses failures.

Syntax

(did *expression* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|----------------------------|
| expression | variant | 1 | An expression to evaluate. |

Results

| Data Type | Description |
|------------|--|
| expression | Returns a truth success value, the result , and a possible failure |

Remarks

If the expression fails, success shall be false, the result shall be nil, and the failure shall contain the generated failure.

Example

```
> (did (string x))
.: true "x" nil

> (did (integer x))
.: false nil [Failure :Name TypeConversion :Text "Cannot convert x to Integer."]
> (bind ?success ?result ?error (did (string "hello world")))
.: true
> ?success ?result ?error
.: true "hello world" nil
```

Related

[assert](#), [can](#), [ensure](#), [fail](#), [finally](#), [may](#), [try](#)

digit

Returns the digit in the specified position of a number.

Syntax

(digit *number position*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number. |
| position | number | 1 | The position of the digit within the number |

Results

| Data Type | Description |
|-----------|---|
| variant | The indicated digit (a number between 0 an 9) or nil. |

Remarks

Returns the digit in the position of the fractional normed significand (0.nnnnn) of a number. If the position is out of range, nil is returned.

Example

```
> (digit 500 1)
.: 5
> (digit 500 2)
.: 0
> (digit 500 3)
.: 0
> (digit 500 4)
.: nil
> (digit (pi) 7)
.: 2
```

Related

digits, exponential, fractional, sign, significand,

digit-p

True if the first position of a string is a digit.

Syntax

(digit-p *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A variant. |

Results

| Data Type | Description |
|-----------|--|
| truth | Returns true if the first position is a digit. |

Remarks

None

Example

```
> (digit-p "The Quick BROWN fox")
.: false
> (digit-p "100")
.: true
> (digit-p foo)
.: false
> (digit-p 100)
.: false
```

Related

[alphanumeric-p](#), [capitalized-p](#), [lowercase-p](#), [uppercase-p](#)

digits

Returns the digits comprising a number.

Syntax

(digits *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number. |

Results

| Data Type | Description |
|-----------|-------------|
| string | The digits. |

Remarks

Returns the fractional digits in the normed significand (0.nnnnn) of a number as a string. Trailing zeros are deleted while leading fractional zeros are preserved.

Example

```
> (digits 500)
.: "500"

> (digits 0.00465)
.: "00465"

> (digits -1050.064)
.: "1050064"

> (digits -2302.023900)
.: "23020239"

> (digits 000.003500)
.: "0035"
```

Related

digit, exponential, fractional, sign, significand

dimensions

Returns the lengths of each dimension in a sequence.

Syntax

(dimensions *sequence*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence. |

Results

| Data Type | Description |
|-----------|---|
| list | The length of each dimension in a sequence. |

Remarks

This function returns the length of each nested sequence as a list. The function assumes that the entire sequence has a uniform number of elements in each dimension.

Example

```
> (dimensions {{{a}{b}{c}}{ {d}{e}{f}}})  
.: {2 3 1}  
  
> (dimensions {a b c d})  
.: {4}  
  
> (dimensions {{a b c}{d e f}})  
.: {2 3}  
  
> (dimensions "abc")  
.: {3}
```

Related

[array](#)

discard

Discards a module.

Syntax

(discard *module*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| module | literal | 1 | A module |

Results

| Data Type | Description |
|-----------|------------------------------|
| literal | The literal discarded |

Remarks

Removes an unwrapped module from the modules collection if no functions are defined in the module, causes a failure otherwise.

Example

```
> (module module1
  (function foo {} bar))

.: module1

> (for ?fn in (functions module1)
  (defunct (qualified module1 ?fn)))

.: nil

> (discard module1)

.: discarded
```

Related

dependencies, discard, extend, forgo, module, require, grok, use

disjoint-p

True if no elements are common among sequences.

Syntax

(disjoint-p sequence sequence ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 2+ | A sequence |

Results

| Data Type | Description |
|-----------|--|
| truth | True if no elements are common to any sequences. |

Remarks

Returns true if, for the provided sequences, no elements are common to any sequences.

Example

```
> (disjoint-p {1 2 3 4}{2 3 4 5)
.: false
> (disjoint-p {2 3 4 5}{1 6 7 8})
.: true
> (disjoint-p {1 2 3 4}{2 3 4 5}{3 4 5 6}{4 5 6 7})
.: false
> (disjoint-p "abc" "def" "ghi")
.: true
```

Related

intersects-p, subset-p

distinct

Creates a new sequence without duplicate elements.

Syntax

(distinct *sequence*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence. |

Results

| Data Type | Description |
|-----------|--|
| sequence | A new sequence with duplicates removed |

Remarks

Creates a new sequence containing unique items.

Example

```
> (distinct {1 2 2 2 3 4 2 3 4 5})  
. : {1 2 3 4 5}  
> (distinct "abcabcdcde")  
. : "abcde"
```

Related

[difference](#), [intersection](#), [union](#)

distribute

Applies a function in parallel to a sequence.

Syntax

(**distribute** *sequence* *function* *result*)

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | list | 1 | A list containing the arguments to each function. |
| function | function | 1 | The function to be distributed and processed in parallel. |
| result | literal | 0-1 | The literal tasks or values (default). |

Results

| Data Type | Description |
|-----------|--|
| list | If tasks is specified, a list of tasks is returned immediately, otherwise a list of values are returned upon completion of all the tasks. |

Remarks

Returns a list of tasks for each element of the sequence by applying the function to each element.

Example

```
> (distribute {5 6 7} (given {?x} (* ?x 2)))
.: {10 12 14}

> (distribute {5 6 7} (given {?x} (* ?x 2)) tasks)
.: {task-1 task-2 task-3}

> (map {task-1 task-2 task-3} await)
.: {10 12 14}

> (distribute {2 3 4} (given {?x} (morph {?x} {++ ++ ++})) values)
.: {5 6 7}
```

Related

apply, concurrent, complete, compose, map, mete, morph, scatter

divide

Safe division.

Syntax

(divide dividend divisor default)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| dividend | variant | 1 | A number.or a list. |
| divisor | variant | 1 | A number.or a list. |
| default | variant | 0-1 | A number or a list. If not supplied, unknown is returned. |

Results

| Data Type | Description |
|-----------|--|
| number | Returns the quotient of the numerator and denominators or the default. |

Remarks

Returns the quotient of the values or the default. It divides the dividend by the divisor. If the default is not supplied, **unknown** is returned. All list arguments must be the same length. If an argument is a list, then each element of the list is divided. If a scalar and a list are divided, then the scalar is divided by each element of the list.

Example

```
> (divide 1.0 2)
.: 0.5

> (divide 5 0 0)
.: 0

> {divide {4 5 6} 0}
.: {undefined undefined undefined}

> {divide {4 5 6} {1 2 0} -1}
.: {4 2.5 -1}
```

Related

/ division, * multiplication, + addition, - subtraction, % modulo , // nth root

divisible-p

True if the divisor evenly divides the dividend.

Syntax

(divisible-p *dividend divisor*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------------------|
| dividend | number | 1 | The value to be divided |
| divisor | number | 1 | The value to divide by |

Results

| Data Type | Description |
|-----------|--|
| truth | True if the modulus of the dividend and divisor equals zero. |

Remarks

Causes a failure if the arguments are not numbers, or if the arguments cannot be converted to the same type for comparison (for example, an imaginary compared to an integer).

Example

```
> (divisible-p 501 10)
.: false
> (divisible-p 500 10)
.: true
> (divisible-p 28 7)
.: true
```

Related

datatype, type, is

do

Evaluates expressions and handles escapes.

Syntax

(**do** *expression ... resume tag recovery ... finally cleanup ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------------|------------|-----|--|
| expression | expression | 1+ | Expressions to do some processing |
| resume | literal | 1 | The literal resume |
| tag | literal | 0-1 | A literal location name (often here) |
| resumption | expression | 0+ | Expressions for resuming after the escape |
| finally | literal | 0-1 | The literal finally |
| cleanup | expression | 0+ | Expressions to do some clean up |

Results

| Data Type | Description |
|-----------|--|
| variant | Returns the value of the last executed expression. |

Remarks

The **do** special form performs escape handling. The **escape** function shall return to the **resume** tag and check if the location matches the intended location of the escape. If the location is correct, the **do** will resume, otherwise it will throw to an encompassing resume. If no location is provided, it will simply resume at the encompassing **do** form.

Example

```
> (do foo)
.: foo

> (do foo bar baz)
.: baz

> (do (step ?x from 1 to 500      nothing)
       (step ?y from 501 to 1000    nothing)
       (step ?z from 1000 to 2000  nothing)
       done)
.: done
```

```

(function main {& ?args}
  (tell user ($ \n Attempting dangerous operation... \n))
  (do
    (do
      (tell user "Escaping...")
      (escape outer)
      (tell user ($ \n Never gonna happen!! \n))
      resume here

      (tell user ($ \n Wrong Location.)))
    resume outer

    (tell user ($ \n Resumed.))
    finally
    (tell user ($ \n Goodbye \n))
  )
)

.: main

> (main)

Escaping...

Resumed.
Goodbye
.: told

```

Related

[wait](#), [abort](#), [await](#), [complete](#), [done-p](#), [reclaim](#), [use](#), [for](#), [iterate](#), [loop](#), [step](#), [task](#), [try](#), [until](#), [while](#)

domain

Creates a package containing relations and rules.

Syntax

(domain *name element*)

Module

KB

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|---|
| name | literal | 0-1 | The name of the domain. (If omitted a name is created). |
| element | variant | 0+ | A relation or rule definition or domain requirement. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| literal | Returns the name of the domain. |

Remarks

Relation definitions and rules may be included. A **require** call may also be included to retrieve additional relations and rules. .

Example

```
> (domain FixIt
    (relation Car :Age :Condition)
    (rule
      if
        [Car :Age /= old :Condition = good]
      do
        (tell user ($ ($$ \n that) is possible. \n)
      then done)))

.: FixIt

> (problem state [Car :Age new :Condition good])

.: Problem_1

> (start Problem_1)

.: {task-1}

that is possible.

>
```

Related

cancel, domain, domains, problem, problems, rule, rules, solve

domains

Returns a list of defined domains.

Syntax

(domains)

Module

KB

Parameters

None.

Results

| Data Type | Description |
|-----------|---|
| variant | Returns a list of all domains or the element void if none exist.. |

Remarks

Relation definitions and rules may be included. A **require** call may also be included to retrieve additional relations and rules. .

Example

```
> (domain FixIt
    (relation Car :Age :Condition)
    (rule
      if
        [Car :Age /= old :Condition = good]
      do
        (tell user ($ ($$ \n that) is possible. \n)
        then done)))
.

.: FixIt

> (domains)

.: FixIt
```

Related

cancel, domain, domains, problem, problems, rule, rules, solve

done

Terminates a generator.

Syntax

(done)

Module

Base

Parameters

None

Results

| Data Type | Description |
|-----------|--------------------|
| escape | Returns an escape. |

Remarks

None.

Example

```
> (generator myGenerator {?interrupt}
  (yield 1)
  (yield 2)
  (if ?interrupt (done))
  (yield 3)
  (yield 4)
  (done))

.: myGenerator

> (collect ?x in (myGenerator true) ?x)

.: {1 2}

> (collect ?x in (myGenerator false) ?x)

.: {1 2 3 4}
```

Related

[generator](#), [generator-p](#), [yield](#)

drop

Removes a relation index.

Syntax

(drop *index* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------------------|
| index | index | 1+ | A relation index or list of indices |

Results

| Data Type | Description |
|------------|---|
| expression | The literal dropped and the number of indices dropped. |

Remarks

Removes an index from a relation if one exists, otherwise causes a failure.

Example

```
> (relation Car :Make :Model :Year)
.: Car

> (index Car :Make :Model)

.: Car_Index_1

> (index Car :Year)

.: Car_Index_2

> (drop Car_Index_1)

.: dropped 1

> (drop Car_Index_2)

.: dropped 1
```

Related

[relation](#), [index](#), [indices](#)

duplicate

Duplicates a file or contents of a folder.

Syntax

(duplicate source target option)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| source | url | 1 | A source url. |
| target | url | 1 | A target url. |
| option | expression | 0-1 | A key value pair, one of overwrite yes no – (default is no). |

Results

| Data Type | Description |
|-----------|---------------------------------|
| literal | The literal duplicated . |

Remarks

Duplicates files on a file system.

Example

```
> (let ?f (file name "foo.txt"))

.: true

> (write ?f "The quick brown fox")

.: File-1

> (close ?f)

.: closed

> (duplicate (url ?f) (url path "bar.txt"))

.: duplicated

> (duplicate (url ?f) (url path "baz.txt") overwrite yes)

.: duplicated
```

Related

[erase](#), [move](#)

during-p

True if an interval occurs within a second interval.

Syntax

(during-p *interval₁* *interval₂* *tolerance*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------------------|-----------|-----|--|
| interval ₁ | interval | 1 | An interval |
| interval ₂ | interval | 1 | An interval |
| tolerance | instant | 0-1 | Amount of variation. (Defaults to zero ticks). |

Results

| Data Type | Description |
|-----------|---|
| truth | True if interval ₁ occurs within interval ₂ . |

Remarks

None.

Example

```
> (during-p  \@i{\@s1 \@s2}  \@i{\@s3 \@s4})  
.: false  
  
> (during-p  \@i{\@s1 \@s5}  \@i{\@s1 \@s8})  
.: true  
  
> (during-p  \@i{\@s1 \@s9}  \@i{\@s4 \@s6})  
.: false  
  
> (during-p  \@i{\@s4 \@s6}  \@i{\@s1 \@s9})  
.: true  
  
> (during-p  \@i{\@s7 \@s9}  \@i{\@s3 \@s8} \@s1)  
.: true
```

Related

before-p, finishes-p, meets-p, overlaps-p, starts-p

dynamic

Creates a dynamic environment for variables.

Syntax

(dynamic *assignments expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-----------|-----|------------------------------|
| assignments | list | 1 | A list of symbol value pairs |
| expression | variant | 0+ | An expression |

Results

| Data Type | Description |
|-----------|-------------------------|
| truth | Returns the value true. |

Remarks

Creates temporary global variables existing only for the duration of the scope.

Example

```
> (function appendToFile {& ?messages}
  (dynamic {?file nil}
    (try
      (set ?file (file name "myfile.out"))
      (seek ?file #)
      (for ?s in ?messages (write ?file ?s))
      (close ?file)
      appended
      learn ?f
        ?f)
    )
  .: appendToFile
```

Related

<-- before tie, --> after tie, assume, assign, constant, global, local, modular

e

Returns Euler's number, 2.718281828459045.

Syntax

(e)

Module

Math

Parameters

None.

Results

| Data Type | Description |
|-----------|--|
| real | Returns the constant 2.718281828459045 . |

Remarks

None.

Example

```
> (e)
.: 2.7182818284590451
```

Related

pi

each

Combines for and with to iterate over sequences matching patterns to the knowledge base.

Syntax

```
(each symbol ... binding reversal sequence ... premises option ... action )  
premises ::= premise  
premises ::= premise premises  
premises ::= premise restriction ... premises  
restriction ::= (predicate value ...)  
premise ::= [category descriptor ...]  
category ::= type  
category ::= list  
descriptor ::= slot binding  
descriptor ::= slot condition  
binding ::= in  
binding ::= per  
binding ::= over  
binding ::= across  
reversal ::= across  
condition ::= slot = value  
condition ::= slot /= value  
condition ::= slot is unary-predicate  
condition ::= slot not unary-predicate  
condition ::= slot binary-predicate value  
condition ::= slot n-ary-predicate value-list  
condition ::= (predicate value ...)  
option ::= scan number  
option ::= limit number  
option ::= group slot ... by slot ... into symbol  
option ::= merge value ...  
option ::= sort ordering ...  
action ::= deem value1 else value2  
action ::= do expression ...  
action ::= give expression  
action ::= list value ...  
action ::= tally  
action ::= yield expression  
ordering ::= term comparator
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , over , or across . |
| reversal | literal | 0-1 | The literal reverse . |
| sequence | sequence | 1+ | A list or string |
| premises | expression | 1+ | One or more premises |
| option | expression | 0-2 | Either scan , limit , group , merge , sort , or any combination. |
| action | literal | 0-1 | Either deem , do , give , list , tally , yield . |
| expression | expression | 0+ | An expression |

Results

| Data Type | Description |
|-----------|--|
| value | The last value returned on the last iteration. |

Remarks

Variables and sequences are required for **in**, **per**, and **over**. If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If the binding **across** is specified, each symbol is bound to elements of the corresponding sequence (i.e, first symbol to the first sequence, second to the second, etc.). If there are insufficient elements to bind to the variables then the function fails. If the literal **reverse** is present, the sequence is traversed in reverse. On each iteration, all the premises are evaluated. Note that a predicate is a truth function, **scan** tells the number of matches to attempt (default is **infinity**), and **limit** tells how many results to return (default is **infinity**). If the action is **deem**, then the subsequent value is returned if the final descriptor is matched, otherwise the else value is returned. If the action is **do**, then for each match the expressions are run, and for the last match, the last value of the last expression is returned, or nil is returned if all descriptors were false or had no matches. If the action is **give**, then the expression is immediately returned from the each call on the first match, akin to **return from each**. If the action is **yield**, then the expression is immediately returned from the each call, but may be resumed with a **next** function call. When all matches are completed, the **done** function is implicitly called to terminate the generator. If the action is **group**, followed by a list of slots and values from which the group is picked, then the literal **by** and a set of slots and values that determines the grouping inclusion , and finally the literal **into** followed by a symbol to which the group shall be bound on each iteration. If the action is either **list** (or **merge**), then a (merged) list is returned or an empty list if nothing matched. If the action is **sort** then the list of slots and comparators should follow. (Note that **list** is required if the sort action is selected.) If the action is **tally**, then the number of matched premises is returned, or zero if no matches.

(**each** symbol ... binding reversal sequence ... premises option ... action expression ...))

Is equivalent to

(**for** symbol ... binding reversal sequence ... (**with** premises option ... action expression ...))

The difference between **for** and **each** is that in a for expression, on every iteration, all the expressions following the sequence(s) are evaluated and the return value is the result of the last expression evaluated on the last iteration. In the **each** expression, the premises following the sequence(s) are evaluated, and if any premise is false, the next iteration occurs, otherwise, if all the premises are true, the action is taken, and the action expressions are evaluated.

Example

```
> (relation Sport
  :Name
  :PlayersPerTeam)

.: Sport

> (relation Team
  :Name
  :Sport
  :Players {})

.: Team

> (relation Person
  :Name
  :Skills
  :Teams {})

.: Person

> (relation Skill
  :Sport
  :Level)

.: Skill

> (repeat (random 1 to 10)
  (let ?sport (new Sport :Name (unique "sport"))
    :PlayersPerTeam (random 1 to 12)))
  (repeat (* (random 1 to 10) 2) ; even number of teams
    (let ?team (new Team :Name (unique "team") :Sport ?sport)))
  Leagues)

.: Leagues

> (so {?sports      (which Sport)
        ?sportCount (# ?sports)
        ?teams       (which Team)}
  (do
    (let ?abilities (pick ?sports (random 1 to ?sportCount))
      ?player     (new Person :Name (unique "person")
                        :Skills (collect ?sport in ?abilities
                                      (new Skill
                                         :Sport ?sport
                                         :Level (random 1 to 10)))))

    (each ?sport in ?abilities
      (let ?team (@ (sort (which ?teams :Sport = ?sport)
                           (given {?a ?b} (<= (# (:Players ?a))
                                         (# (:Players ?b)))))))
        do
          (enq ?team :Players ?player)
          (enq ?player :Teams ?team)))))

.:
```

```
until (every ?team in ?teams
      (>= (# (:Players ?team)) (# (:PlayersPerTeam (:Sport ?team))))))
Players)

.: Players

> (collect ?sport in (which Sport)
  {?sport (avg (which Skill :Sport = ?sport :Level as ?level list ?level))})
.: {{sport3 4.9} {sport1 7.898} {sport2 6.382} {sport4 8.452}}
```

Related

which, with

encode

Encodes a string.

Syntax

(**encode** *source format*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|---|
| source | string | 1 | A string to encode. |
| format | literal | 1 | One of { base16 , base64 , utf8 , utf16 , none } |

Results

| Data Type | Description |
|-----------|--------------------|
| string | The encoded string |

Remarks

Converts a string to the specified format.

Example

```
> (encode "The quick brown fox" base16)
.: "54686520717569636B2062726F776E20666F78"

> (encode "The quick brown fox" base64)
.: "VGhlIHF1aWNrIGJyb3duIGZveA=="

> (encode "The quick brown fox" none)
.: "The quick brown fox"

> (decode "54686520717569636B2062726F776E20666F78" base16)
.: "The quick brown fox"

> (decode "VGhlIHF1aWNrIGJyb3duIGZveA==" base64)
.: "The quick brown fox"

> (decode "The quick brown fox" none)
.: "The quick brown fox"
```

Related

[decode](#)

enq

Inserts an element into an embedded sequence.

Syntax

(**enq** *container place option*)

option ::= **at** *position value*
option ::= **in** *coordinate value*
option ::= **unique** *value*
option ::= **each** *value(s)*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|--|
| container | assortment | 1 | An assortment or tuple |
| place | variant | 1 | A key or position. |
| option | expression | 1+ | Any of the following: each - appends remaining value(s) to the sequence. at <i>value</i> - inserts the value at the position. in <i>coordinate value</i> - inserts all values at the coordinate. unique <i>value</i> - appends value if it is not already present. |

Results

| Data Type | Description |
|-----------|-------------------------|
| variant | The modified container. |

Remarks

If only a single value is provided for *entry*, then the value is appended to the sequence. If an integer position and value is provided, then the value is inserted at the position. If the keyword **unique** is provided, then the value is appended to the sequence if it does not already exist

Example

```
> (let ?a {{}{1 2 3}{}})  
.: true  
> (enq ?a in {2 2} each 8)  
.: {{}{1 8 2 3}{}}
```

Related

add, cut, deq, get, has, key, keys, peek, put, values

ensure

Performs type checking.

Syntax

(**ensure** *check* ...)

check ::= *kind value*
check ::= {*kind* ...} *value*
kind ::= *taxon*
kind ::= *meron*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|------------|-----|---|
| check | expression | 1+ | A datatype, prototype, or list containing datatypes and prototypes. |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if all values correspond to their indicated types. |

Remarks

If any of the values do not correspond to their designated types, then the function fails; otherwise the function returns **true**.

Example

```
> (function multiply {?a ?b + ?c}
  (ensure number ?a number ?b {number nil} ?c)
  (* ?a ?b (default ?c 1)))

.: multiply

> (multiply 10 banana)

.: [Failure :Name InvalidArg :Text "Expected a number, encountered banana."]

> (multiply 10 5)

.: 50
```

Related

assert, can, did, fail, may, try

entries

Retrieves key value pairs for an assortment.

Syntax

(entries *assortment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------|
| assortment | assortment | 1 | An assortment. |

Results

| Data Type | Description |
|-----------|-------------------------------|
| list | A list of all key value pairs |

Remarks

Returns a list containing key value pairs for an association.

Example

```
> (entries (lexicon a 1 b 2 c 3 d 4))
.: {{a 1} {b 2} {c 3} {d 4}}
> (entries (structure Foo :X :Y :Z apple))
.: {{relation Foo} {:X nil} {:Y nil} {:Z apple}}
```

Related

bindings, definitions, positions

enum

Retrieves an enumerated assortment.

Syntax

(enum name)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|-----------------------------------|
| name | literal | 1 | The name of a defined enumeration |

Results

| Data Type | Description |
|-----------|-----------------------------|
| literal | The name of the enumeration |

Remarks

Each entry in the elements list may be a literal or a list containing a literal and a number. If a number is present, the number shall be assigned to the preceding literal as the index of that literal, and the next literal shall have an index of the number plus one, unless another index is present. For example, the arguments a, b 7, c: a shall be assigned the value 1, b shall be assigned the value 7, and c shall be assigned the value 8. An enumeration is an immutable assortment so it cannot be modified.

Example

```
> (enumeration MySimpleType a b 7 c)
.: Enumeration-1
> (keys Enumeration-1)
.: {a b c}
> (entries (enum MySimpleType))
.: {{a 1}{b 7}{c 8}}
> (values (enum MySimpleType))
.: {1 7 8}
```

Related

[add](#), [cut](#), [elem](#), [enumerations](#), [elems](#), [entries](#), [size](#)

enumeration

Defines an enumerated assortment.

Syntax

(enumeration name element ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|-----------------------------------|
| name | literal | 1 | The name of an enumeration |
| element | expression | 1+ | A literal or literal number pair. |

Results

| Data Type | Description |
|-----------|-----------------------------|
| literal | The name of the enumeration |

Remarks

Each entry in the elements list may be a literal or a list containing a literal and a number. If a number is present, the number shall be assigned to the preceding literal as the index of that literal, and the next literal shall have an index of the number plus one, unless another index is present. For example, the arguments a, b 7, c: a shall be assigned the value 1, b shall be assigned the value 7, and c shall be assigned the value 8. An enumeration is an immutable assortment so it cannot be modified.

Example

```
> (enumeration MySimpleType a b 7 c)
.: Enumeration-1
> (keys Enumeration-1)
.: {a b c}
> (entries Enumeration-1)
.: {{a 1}{b 7}{c 8}}
> (values Enumeration-1)
.: {1 7 8}
```

Related

[add](#), [cut](#), [elem](#), [enum](#), [enumerations](#), [elems](#), [entries](#), [size](#)

enumerations

Returns a list of all enumerations.

Syntax

(enumerations)

Module

Base

Parameters

None.

Results

| Data Type | Description |
|-----------|---------------------------------|
| list | Returns a list of enumerations. |

Remarks

None.

Example

```
> (enumerations)
.: {}

> (enumeration MySimpleType {a b c})
.: MySimpleType
> (enumerations)
.: {MySimpleType}
```

Related

[enumeration](#)

environment

Creates a new context for variables and functions.

Syntax

(environment *parent* *entry* ...)

entry ::= *symbol value*

entry ::= *function definition*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-------------|-----|--|
| parent | environment | 0-1 | An environment or the value nil. |
| entry | expression | 0+ | A symbol and value pair or function and definition pair. |

Results

| Data Type | Description |
|-----------|-------------------------------|
| literal | A resource to the environment |

Remarks

Entries with literal keys are retrieved via the **definitions** function, while entries with symbol keys are retrieved with the **bindings** function. All entries may be retrieved using the **entries** function.

Example

```
> (environment nil (symbol A) "hello world")
.: Environment-1

> (add Environment-1 foo (given {?x} (+ ?x 1)))
.: Environment-1

> (add Environment-1 bar (function bar {?x} (* ?x 2)))
.: Environment-1

> (entries Environment-1)
.: {{?A "hello world"} {foo (given {?x} (+ ?x 1))} {bar (fn bar {?x} (* ?x 2))}}
```

Related

@, add, includes, cut, entries, environment-p, keys, reclaim, #, values

epoch

Returns a Unix epoch or the current epoch.

Syntax

(epoch *time*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|--|
| time | time | 0-1 | A date, moment, seconds or tick value. |

Results

| Data Type | Description |
|-----------|---|
| epoch | The Unix time in seconds since January 1, 1970 (UTC). |

Remarks

Converts the supplied time to a Unix epoch.

Example

```
> (epoch 2014-11-29T23:39:29.741)
.: \@e1417333169
> (epoch \@m2014939313515537)
.: \@e1481565910
> (epoch \@t63573798191026000)
.: \@e1481565910
> (epoch (date))
.: \@e1533235279
> (epoch)
.: \@e1533235284
> (epoch (tick))
.: \@e1533235295
```

Related

date, moment, tick

eradicate

Deletes a knowledge base.

Syntax

(**eradicate** *knowledge*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|------------------|
| knowledge | knowledge | 1 | A knowledge base |

Results

| Data Type | Description |
|-----------|----------------------------|
| knowledge | The opened knowledge base. |

Remarks

Deletes an existing knowledge base.

Example

```
> (conceive (var ?kb (knowledge name defaultKB path "c://premise/kb/")))

.: defaultKB

> (attach ?kb)

.: defaultKB

> (relation Fact :All :Are)

.: Fact

> (detach ?kb)

.: defaultKB

> (eradicate ?kb)

.: defaultKB

> (free (cede (reference ?kb)))

.: freed
```

Related

attach, conceive, detach, each, get, knowledge, knowing, put, using, which, with

erase

Deletes files or folders.

Syntax

(**erase** *file option ...*)

option ::= key value

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| file | file | 1 | A file resource. |
| option | expression | 0-1 | Key value pairs where a key is one of force yes no - forcibly erase the file. Default is no . type {file or folder} - what to erase. Default is file . |

Results

| Data Type | Description |
|-----------|-----------------------------|
| literal | The literal erased . |

Remarks

This function attempts to delete a file. If successful it returns **erased**, otherwise it causes a failure.

Example

```
> (let ?path (file path (path "quick.txt")))

.: true

> (erase ?path force yes)

.: erased

> (erase (url scheme folder path (folder "c:/foo")) type folder)

.: erased
```

Related

[close](#), [file-p](#), [files](#), [folders](#), [file](#), [read](#), [seek](#), [write](#)

escape

Escapes to the next resume location in the call graph, matching the location if provided.

Syntax

(escape tag)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|------------------|
| tag | literal | 0-1 | A location name. |

Results

| Data Type | Description |
|-----------|-------------|
| Failure | A failure. |

Remarks

Returns control to superordinate do .. resume special form. If a tag is provided, it will compare the tag and throw again if the tag does not match.

Example

```
> (do
  (do
    (escape proceed)
    (try
      (signal [Failure :Name User :Text "A big problem"])
      learn ?e
      (tell user ($ Learned ?e \n))
      (tell user ($ Rethrowing... \n)))
    resume
    (tell user ($ This is ignored)))
  resume proceed
  (tell user ($ Resuming \n))
  finally
  done)

Resuming
.: done
```

Related

confirm, confute, ensure, fail, signal, try

eval

Evaluates an expression.

Syntax

(eval *expression environment* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|-------------------------------|
| expression | expression | 1 | An expression to be evaluated |
| environment | environment | 0-1 | An environment |

Results

| Data Type | Description |
|-----------|---|
| value | The result of evaluating the expression |

Remarks

Evaluates the expression and returns the resulting value. If no environment is provided, the current scope is used. The environment parameter is always evaluated in the current scope while the expression is evaluated in the provided scope, or the current scope if argument 2 is omitted.

Example

```
> (eval (+ 1 2 3 4))  
.: 10  
  
> (' (+ 1 2 3 4))  
.: (+ 1 2 3 4)  
  
> (eval (' (' (+ 1 2 3 4))) (scope))  
.: (+ 1 2 3 4)  
  
> (eval (eval (' (' (+ 1 2 3 4)))))  
.: 10
```

Related

'*quote*, `*expand*, *apply*, *call*, *invoke*

even-p

True if the number is even.

Syntax

(even-p *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number |

Results

| Data Type | Description |
|-----------|-----------------------------|
| truth | True if the number is even. |

Remarks

None.

Example

```
> (even-p 500)
.: true
> (even-p 0)
.: true
> (even-p -23)
.: false
```

Related

odd-p, negative-p, positive-p, zero-p

every

True if a predicate is true for every element in a sequence.

Syntax

(every symbol ... binding sequence expression ... test)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , or over . |
| sequence | sequence | 1 | A list or string |
| expression | expression | 0+ | An expression. |
| test | truth | 1 | A truth expression |

Results

| Data Type | Description |
|-----------|--|
| truth | True if every element satisfies the predicate, else false. |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If there are insufficient elements to bind to the variables then the function fails. The expressions and test are evaluated. If the sequence is empty the function fails.

Example

```
> (every ?n in {1 2 3 4} (= number (taxon ?n)))
.: true

> (every ?m ?n per {{1 2} {3 4}} (< ?m ?n))
.: true

> (every ?m ?n over {4 3 2 1} (divisible-p ?m ?n))
.: false
```

Related

few, most, nevery, none, some

exactly

True if a number of elements satisfy a clause.

Syntax

(**exactly** *quantity* **of** *sequence* **clause** **within** *margin*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| quantity | number | 1 | A number |
| of | literal | 1 | The literal of |
| sequence | sequence | 1 | A sequence |
| clause | expression | 1 | A key value pair. One of is <i>value</i> - a sequence element, are <i>value</i> - a sequence element, satisfies <i>predicate</i> - a unary truth function. satisfy <i>predicate</i> - a unary truth function. |
| within | literal | 0-1 | The literal within |
| margin | number | 0-1 | A number. (default is zero). |

Results

| Data Type | Description |
|-----------|---|
| truth | True if exactly quantity elements satisfies the predicate |

Remarks

The predicate must return true or false.

Example

```
(exactly 4 of {1 2 3 4} satisfies number-p)
.: true

(exactly 1 of {1 a 3 4} is a)
.: true

(exactly 3 of {1 2 3 4} satisfy odd-p within 1)
.: true
```

Related

= *equal*,

exchange

Creates a new a sequence with values swapped.

Syntax

(exchange *sequence* *substitution* ...)

substitution ::= *prior-position* *later-position*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------------|------------|-----|---|
| sequence | sequence | 1 | A list or a string. |
| substitution | expression | 1+ | A prior and later position pair. A position may be a number or a list of numbers.. |

Results

| Data Type | Description |
|-----------|-----------------------|
| sequence | The modified sequence |

Remarks

Returns a copy of the original sequence with substitutions made. For destructive modification of a sequence use the **swap** function.

Example

```
> (exchange {A B C} 1 3)
.:{C B A}

> (exchange "ABCD" 1 3)
.: 'CBAD'

> (exchange {{1 2 3}{4 5 6}} {2 1} {1 3})
.: {{1 2 4}{3 5 7}}
```

Related

@ *element*, append, fix, push, pop, swap

excludes

True if a sequence does not contain an element.

Syntax

(excludes *sequence* *element* *option ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| sequence | sequence | 1 | A sequence. |
| element | variant | 1 | A value |
| option | expression | 0-2 | A key value pair. Any of depth <i>number</i> - Number or # (for any depth). Default is 1. transform <i>function</i> - A function to transform the element.. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the element is not in the sequence. |

Remarks

If the element and sequence are strings, then a string search is performed.

Example

```
> (excludes {a b c d e f g} x)
.: true

> (excludes "cbazyx" "a")
.: false
```

Related

includes, excludes, in, occurrences, out, pick, position

exists-p

True if a value is a thing.

Syntax

(exists-p *value* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1+ | A value |

Results

| Data Type | Description |
|-----------|--------------------------------|
| truth | True if the values is a thing. |

Remarks

None.

Example

```
> (exists-p yes)
.: true
> (exists-p nil)
.: false
> (exists-p (nothing))
.: false
> (exists-p 3.1415926)
.: true
```

Related

null-p

exit

Explicitly ends a task using a return value.

Syntax

(exit *value*)

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 0-1 | A return value. Defaults to nil if not specified. |

Results

| Data Type | Description |
|-----------|--------------------|
| value | The argument value |

Remarks

Forcibly ends a task. Calling **exit** from the interpreter shall not exit the interpreter. The **bye** function must be called to exit the interpreter.

Example

```
> (concurrent (step ?x from 1 to 999999999
           (if (> ?x 1000000) (exit exited))) ;
.: {Task-343}

> (await task-343)

.: exited

> (free task-343)

.: freed

> (exit done)

.: done
```

Related

[wait](#), [abort](#), [await](#), [complete](#), [do](#), [task](#)

exponential

Returns the base ten exponent of a number.

Syntax

(exponential *number significand*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-------------|-----------|-----|--|
| number | number | 1 | A number. |
| significand | literal | 0-1 | One of normed , scientific , integer . Default is scientific. |

Results

| Data Type | Description |
|-----------|---------------|
| number | The exponent. |

Remarks

The normed significand represents the number as a fraction (0.nnnn). The scientific significand represents the number as a decimal between 1.0 and 10.0 (n.nnnn). The integer significand represents the number as an integer (nnnn). The exponential represents the power of 10 to which the significand must be raised to accurately reflect the number.

Example

```
> (exponential 500)
.: 2

> (exponential 0)
.: 1

> (exponential -1000 normed)
.: 4

> (exponential -2302.023900 integer)
.: -4

> (exponential 0.003500 scientific)
.: -3
```

Related

digit, digits, fractional, sign, significand,

expression

Creates an expression.

Syntax

(expression *value* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | Any value |

Results

| Data Type | Description |
|------------|---|
| expression | Returns a expression containing the values. |

Remarks

Expressions are lists without delimiters. An empty expression has no printed representation.

Example

```
> (expression a b c {d})  
.: a b c {d}  
  
> (expression "a b c")  
.: "a b c"  
  
> (let ?a 1 ?b 2)  
.: true  
  
> (bind ?a ?b (expression ?b ?a))  
.: true  
  
> ?a ?b  
.: 2 1  
  
> (void-p (expression))  
.: true
```

Related

inside, is, void-p

extend

Adds new definitions to a module.

Syntax

(extend *module expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| module | literal | 1 | The module containing the extended definitions |
| expression | expression | 0+ | An expression. |

Results

| Data Type | Description |
|-----------|---------------------|
| literal | The module literal. |

Remarks

Allows new definitions to occur in the target module. The module must be unwrapped.

Example

```
> (use User)
.: User
> (extend MyModule (function foo {} 100))
.: MyModule
> (MyModule.foo)
.: 100
```

Related

dependencies, discard, forgo, module, modules, require, grok, use

facility

Creates a private function in a module.

Syntax

(facility *name* *parameters* *expression* ...)

parameters ::=
parameters ::= {}
parameters ::= {*required* ... }
parameters ::= {*required* ... + *optional* ... }
parameters ::= { + *optional* ... }
parameters ::= {*required* ... + *optional* ... & *remaining* }
parameters ::= { & *remaining* }

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| name | literal | 0-1 | The name of the function. If not supplied then the name shall follow the pattern fn-<number> . |
| parameters | list | 0-1 | A list containing variables, keywords, sigils, and keyword symbol associations. |
| expression | variant | 0+ | One or more expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|--------------------------|
| literal | The name of the function |

Remarks

Facilities are module level functions which are not accessible outside of the module in which they are created. If the name is supplied then the facility is a named facility, otherwise it is automatically named (auto-named). If a parameter list is supplied then it shall bind the formal parameter variables to any actual parameter expressions provided. The first variables in the parameter list are required. If a plus sign is provided, then those variables following are optional. If an ampersand is provided, then the symbol following is bound to a list of remaining actual parameter values. Keyword parameters (literal symbol pairs) may be provided as required or optional. The entire parameter list may be omitted.

Example

```
> (module SuperMath

  (facility power {?x + ?y = 1}           ; private to the module
    (** ?x ?y)
  )

  (function superpower {& ?numbers}        ; public to the module
    (if (void-p ?numbers)
      (return unknown))
    (let ?result (@ ?numbers))
    (for ?n in (rest ?numbers)
      (--> power ?result ?n))
    ?result
  )
)

.: SuperMath

> (use User)

.: User

> (SuperMath.superpower 2 3 4)

.: 4096

> (SuperMath.power 2 3)

.: [Failure :Name UndefinedFunction :Text "The function SuperMath.power is not
defined"]

> (facility-p power SuperMath)

.: true

> (function-p power SuperMath)

.: false
```

Related

call, extend, functions, macro, module, modules

few

True if a predicate is true for less than half the elements in a sequence.

Syntax

(**few** *symbol* ... *binding* *sequence* *expression* ... *test*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , or over . |
| sequence | sequence | 1 | A list or string |
| expression | expression | 0+ | An expression. |
| test | truth | 1 | A truth expression |

Results

| Data Type | Description |
|-----------|---|
| truth | True if less than half the elements satisfies the predicate |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If there are insufficient elements to bind to the variables then the function fails. The expressions and test are evaluated.

Example

```
> (few ?n in {1 2 3 4} (= (taxon ?n) number))
.: false

> (few ?m ?n per {{2 1} {3 4} {7 9}} (< ?m ?n))
.: false

> (few ?m ?n over {4 2 3 7} (divisible-p ?m ?n))
.: true
```

Related

every, nevery, most, none, some

file

Opens or creates a file.

Syntax

(**file** *url option ...*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| url | url | 1 | A url to the file. |
| option | expression | 0+ | A key value pair. Any of mode <i>mode</i> - append , create , open (default), truncate . access <i>access</i> - read , write , or readwrite (default). sharing <i>share</i> - none , read (default), write , or readwrite . seek <i>location</i> - a number from 1 (default) to # (eof). content type - binary (default) or text . from <i>place</i> - memory or network (default). buffer size - minimum buffer size (default is 1). |

Results

| Data Type | Description |
|-----------|-----------------|
| file | A file literal. |

Remarks

Opens a file for reading or writing and returns a file resource. If the file does not exist then it is created. If **from** memory is specified, the contents of the file is first cached locally to memory then accessed. If **from** network is specified, the contents of the file is accessed over the network.

Example

```
> (let ?url (url "file://localhost/C:/temp/quick.txt"))

.: true

> (if (there ?url)
    (let ?file (file ?url mode append))
    (write ?file "The quick brown fox")
    (close ?file)
    (free ?file))

.: closed
```

Related

[close](#), [erase](#), [free](#), [read](#), [seek](#), [there](#), [write](#)

files

Returns a list of file names.

Syntax

(files *folder* **)**

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| folder | folder | 1 | A folder. |

Results

| Data Type | Description |
|-----------|---------------------------|
| list | A list file name strings. |

Remarks

If the path is invalid, the function fails. Files or symbolic links to files are returned.

Example

```
> (files path (path "c" (separator volume)))
.: {"homework.txt" "temp.txt" "mypath.lnk"}
```

Related

folder, folders, links, there, path, separator

fill

Fills a list with the result of an expression.

Syntax

(**fill** *symbol from start to end by increment with expression*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------------|-----------|-----|---|
| symbol | symbol | 1 | A symbol iterator |
| from | literal | 0-1 | The literal from . |
| start | number | 0-1 | A number. If from is given this is required. Defaults to 1. |
| to | literal | 1 | The literal to . |
| finish | number | 1 | A number. |
| by | literal | 0-1 | The literal by . |
| increment | number | 0-1 | A number. If by is supplied this is required. |
| with | literal | 1 | the literal with . |
| expression | variant | 1 | a value to populate the list |

Results

| Data Type | Description |
|-------------|--|
| list | A list of the indicated length containing the value of the expression. |

Remarks

The value of the symbol iterator is available to the expression when building the list.

Example

```
> (fill ?i to 3 with ($ "foo-" ?i))
.: {"foo-1" "foo-2" "foo-3"}
> (fill ?j to 5 with (* ?j 2))
.: {2 4 6 8 10}
> (fill ?x from -4 to 6 by 2 with {?x})
.: {{-4}{-2}{0}{2}{4}{6}}
```

Related

[fill](#), [pad](#), [range](#)

filter

Returns a sequence of elements that satisfy a predicate.

Syntax

(filter *sequence predicate* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|-------------------|
| sequence | sequence | 1 | A sequence. |
| predicate | function | 1 | A truth function. |

Results

| Data Type | Description |
|-----------|--|
| list | A new list containing elements satisfying the predicate. |

Remarks

For each element in the sequence, if the predicate is true for the element, the element is concatenated into a result list, otherwise the item is excluded from the result list. Finally, the result list is returned.

Example

```
> (filter {1 2 3 4} even-p)
.: {2 4}
> (filter {1 2 3 4 5 6} (given {?n} (divisible-p ?n 3)))
.: {3 6}
```

Related

apply, coalesce, collect, gather, reduce, sort, map, quantity

find

Returns the best matching candidates.

Syntax

(find *features candidates option ...* **)**

option ::= key value

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| features | List | 1 | The features to compare to the candidates |
| candidates | list | 1 | The candidates to be compared |
| option | expression | 0+ | <p>Key value pairs , where a key is one of:</p> <p>limit - The number of matches to return. (default is 1)</p> <p>minscore - value above which candidates shall be considered. Default is ninfinity</p> <p>maxscore – Value below which candidates are acceptable. Default is 1.</p> <p>transform – The literal name of a value transformation function.</p> <p>comparer - The literal name of a comparison function</p> |

Results

| Data Type | Description |
|-----------|--|
| list | A sorted list of lists containing candidate - score pairs. |

Remarks

Find compares a set of features to those of candidates and returns the best matching candidate(s) with match score(s). It uses the similarity function (\sim) as the default comparer. If provided, transforms are applied before comparison. Finally, a comparer can accept two values and returns a score representing the degree of similarity where higher means more similar.

Example

```
> (find {"t" "th" "thi" "his" "is" "s"}  
        {"there" "that" "though" "thin"})  
        comparer (given {?x} (ngrams ?x 3)))  
  
. : {{"thin" 0.142857}}
```

Related

\sim similarity, $/\sim$ dissimilarity

finishes-p

True if two intervals finish together.

Syntax

(finishes-p *interval₁* *interval₂* *tolerance*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------------------|-----------|-----|--|
| interval ₁ | interval | 1 | An interval |
| interval ₂ | interval | 1 | An interval |
| tolerance | instant | 0-1 | Amount of variation. (Defaults to zero ticks). |

Results

| Data Type | Description |
|-----------|---|
| truth | True if interval ₁ finishes with interval ₂ . |

Remarks

None.

Example

```
> (finishes-p  \@i{\@s1 \@s2}  \@i{\@s3 \@s4})  
.: false  
  
> (finishes-p  \@i{\@s1 \@s5}  \@i{\@s4 \@s5})  
.: true  
  
> (finishes-p  \@i{\@s1 \@s9}  \@i{\@s4 \@s6})  
.: false  
  
> (finishes-p  \@i{\@s4 \@s9}  \@i{\@s1 \@s9})  
.: true  
  
> (finishes-p  \@i{\@s7 \@s9}  \@i{\@s3 \@s8} \@s1)  
.: true
```

Related

before-p, during-p, meets-p, overlaps-p, starts-p

finishing

Returns the instant an interval finishes.

Syntax

(finishing *interval*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| interval | interval | 1 | An interval |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| instant | The finishing value of the interval. |

Remarks

None.

Example

```
> (finishing \@i{\@s1 \@s2})  
. : \@s2  
> (finishing \@i{\@s1 \@s5})  
. : \@s5  
> (finishing \@i{\@s4 \@s6})  
. : \@s6
```

Related

[beginning](#)

fix

Adds variables to the current environment.

Syntax

(fix *declaration ...* **)**

declaration ::= *taxon symbol value*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|------------|-----|-----------------------------------|
| declaration | expression | 1+ | A taxon, symbol and value triple. |

Results

| Data Type | Description |
|-----------|---------------|
| Truth | Returns true. |

Remarks

Sequentially creates variables in the current environment. Each symbol shall replace any extant symbol with the same name in the environment. The taxon shall restrict the type of the symbol. If the default does not match the taxon, a failure shall be triggered.

Example

```
> (fix literal ?person DrWho)
.: true
> ?person
.: DrWho
> (fix integer ?x 1
    float   ?y 2.0
    long    ?z 3n)
.: true
```

Related

-- before tie, --> after tie, constant, dynamic, fix, global, local, set, tie

float

Converts a value to a floating number.

Syntax

(float *value* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 1 | A number or the literal minimum or maximum . |

Results

| Data Type | Description |
|-----------|-----------------|
| float | A float number. |

Remarks

If the value cannot be converted to a float, the function fails.

Example

```
> (float {a b c})  
. : [Failure :Name ArgumentValue :Text "{a b c} is not a float"]  
  
> (float "a b c")  
. : [Failure :Name ArgumentValue :Text "'a b c' is not a float"]  
  
> (float 500)  
. : 500.0f  
  
> 500.0f  
. : 500.0f  
  
> (float "23.4")  
. : 23.4f
```

Related

[complex](#), [imaginary](#), [integer](#), [long](#), [rational](#), [real](#)

floor

Rounds a number towards negative infinity.

Syntax

(floor *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number |

Results

| Data Type | Description |
|-----------|-------------------------|
| number | The next lower integer. |

Remarks

Returns the next lower positive integer.

Example

```
> (floor 100.5)
.: 100
> (floor -100.5)
.: -101
```

Related

[ceiling](#), [round](#), [truncate](#)

fold

Transforms a sequence into a value.

Syntax

(fold symbol ... in sequence into expression ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--|
| symbol | symbol | 2+ | Iteration variables |
| in | literal | 1 | The literal in |
| sequence | sequence | 1 | A sequence (string or list) |
| into | literal | 1 | The literal into |
| expression | variant | 1+ | A expression that transforms the iteration variables |

Results

| Data Type | Description |
|-----------|--|
| list | A new list after the transformation is applied |

Remarks

The first element of the list is stored in the first variables and the second element is stored in the second symbol (and so forth), then the expressions are evaluated. The result of the last expression is stored in the first symbol while the next element(s) in the sequence is (or are) stored in the second symbol (and so on). This is repeated until all elements are processed

Example

```
> (fold ?x ?y in {1 2 3 4 5} into (* ?x ?y))  
. : 120  
  
> (generator OneToFive (yield 1)(yield 2)(yield 3)(yield 4)(yield 5)(done))  
. : OneToFive  
  
> (fold ?i ?j ?k in (OneToFive) into (+ ?i ?j ?k))  
. : 15
```

Related

[reduce](#)

folder

Creates or locates a folder in the file system.

Syntax

(folder url)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|--------------------|
| url | url | 1 | A url to the file. |

Results

| Data Type | Description |
|-----------|--------------------|
| folder | A folder resource. |

Remarks

Returns a folder resource.

Example

```
> (let ?path (path "." (separator) "temp"))

.: true

> (there path ?path)

.: false

> (folder path ?path)

.: folder-1

> (there path ?path)

.: true
```

Related

[file](#), [file-p](#), [folder-p](#), [path](#), [read](#), [separator](#), [write](#)

folders

Returns a list of sub folders.

Syntax

(folders option ...)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|---|
| option | expression | 1+ | A key value pair. One of url <i>url</i> - full url to the file. host <i>host</i> - name of the host computer volume <i>volume</i> - name of the disk volume path <i>path</i> - name of the path to the folder name <i>name</i> - name of the folder |

Results

| Data Type | Description |
|-----------|--|
| list | A list of folders strings in the specified path. |

Remarks

If the path is invalid, the function fails. Otherwise, a list of sub folders is returned for the path.

Example

```
>(folders path (path "."))
.: {"build" "META-INF" "WebContent"}
```

Related

erase, file, files, folder, there, path, separator

for

Iterates over the elements in a sequence.

Syntax

(**for** *symbol* ... *binding reversal sequence* ... *expression* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , over , or across . |
| reversal | literal | 0-1 | The literal reverse . |
| sequence | sequence | 1+ | A list or string |
| expression | variant | 0+ | The expression(s) to evaluate |

Results

| Data Type | Description |
|-----------|--|
| value | The last value returned on the last iteration. |

Remarks

Variables and sequences are required for **in**, **per**, and **over**. If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If the binding **across** is specified, each symbol is bound to elements of the corresponding sequence (i.e, first symbol to the first sequence, second to the second, etc.). If there are insufficient elements to bind to the variables then the function fails. If the literal **reverse** is present, the sequence is traversed in reverse. On each iteration, all the expressions are executed. The return value is the result of the last call evaluated on the last iteration.

Example

```
> (for ?c in reverse "the quick brown fox"  (tell user ?c))

xof nworb xciuq eht.: told

> (for ?key ?value per {{k1 v1}{k2 v2}{k3 v3}}
   (tell user ($ ?key ($$ ?value , (\s)))))

k1 v1, k2 v2, k3 v3, .: told
```

Related

count, do, each, loop, repeat, step, until, while

forever

Repeatedly evaluates expressions until a failure is encountered.

Syntax

(forever *expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , over , or across . |
| reversal | literal | 0-1 | The literal reverse . |
| sequence | sequence | 1+ | A list or string |
| expression | variant | 0+ | The expression(s) to evaluate |

Results

| Data Type | Description |
|-----------|--|
| value | The last value returned on the last iteration. |

Remarks

On each iteration, all the expressions are executed. The return value is the result of the last call evaluated on the last iteration.

Example

```
> (forever
  (try  (print ">> " (eval (ask user "*")))
    learn ?error (print ">> " ?error)))

* hello
>> hello

*
```

Related

count, do, each, loop, repeat, step, until, while

forgo

Removes a dependent module.

Syntax

(forgo *dependency module*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--|
| dependency | literal | 1 | A dependent module. |
| module | literal | 0-1 | A module. Default to the module obtained by (use) . |

Results

| Data Type | Description |
|-----------|---------------------|
| module | Returns the module. |

Remarks

Removes a module from the current module's dependencies list.

Example

```
> (dependencies User)
.: {Premise}

> (extend User (require (module Algebra)))
.: Algebra

> (dependencies User)
.: {Premise Algebra}

> (forgo Algebra User)
.: User

> (dependencies User)
.: {Premise }
```

Related

dependencies, discard, extend, forgo, module, require, grok, use

format

Formats a string.

Syntax

(format template value ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| template | string | 1 | A format template. which uses the following directives: ~c - formats a number as currency ~e - formats a number as an exponential ~f - formats a floating point number ~g - formats a long number ~n - formats an integer number ~s - formats a string. ~v - formats a generic variant ~~ - formats a single tilde character |
| value | variant | 0+ | A value to be formatted. |

Results

| Data Type | Description |
|-----------|--------------------|
| string | A formatted string |

Remarks

The formatted result is returned.

Example

```
> (format "~v ~v" hello world)
.: "hello world"
```

Related

\$ concatenate, \$\$ elide, ask, capitalize, char, lowercase, tell, trim, uppercase

fractional

Returns the fractional portion of a number.

Syntax

(fractional *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number. |

Results

| Data Type | Description |
|-----------|-------------------------|
| number | The fractional portion. |

Remarks

The fractional portion is the number minus the floor of the number.

Example

```
> (fractional 300)
.: 0.0

> (fractional 0.564)
.: 0.564

> (fractional -2302.023900)
.: -0.0239

> (fractional 0.003500)
.: 0.0035
```

Related

digit, digits, exponential, sign, significand

free

Releases resources.

Syntax

(free *resources* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---|
| resources | variant | 1 | A single resource or list of resources. |

Results

| Data Type | Description |
|-----------|---|
| literal | Returns the literal freed if successful. |

Remarks

None.

Example

```
> (free
  (tarry (complete
            (do (idle \@s5) something)
            (do (idle \@s5) something-else)
            (do (idle \@s5) one-more-thing)))))

.: freed

> (free
  (cancel (complete
            (do (idle \@s5) something)
            (do (idle \@s5) something-else)
            (do (idle \@s5) one-more-thing)
            mode immediate))

.: freed
```

Related

concurrent, cancel, complete, reclaim, tarry, task, wait

full

True if values are equal or either value is nil.

Syntax

(full *first second* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------------------|
| first | variant | 1 | A value to be compared. |
| second | variant | 1 | A value to be compared. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if all values are the same or if either is nil |

Remarks

Returns true if each value is the same or if either value is **nil**. Both values must be of the same type, and must be numbers, literals, or strings. Short circuits if false. Two items are equivalent if their representation is the same regardless of whether or not they occupy the same location in memory (viz. identical). Return false otherwise.

Example

```
> (full a a)
.: true

> (full "johnny" "ralph")
.: false

> (full 3 nil)
.: true

> (full nil 4)
.: true
```

Related

< less , <= less or equal, > greater, >= greater or equal, /= unequal, identical, left, right

function

Creates a public function in a module.

Syntax

(function *scope name parameters returning expression ...* **)**

parameters ::=
parameters ::= {}
parameters ::= {required ...}
parameters ::= {required ... + optional ...}
parameters ::= {required ... + optional ... % keyword}
parameters ::= {required ... + optional ... % keyword ... & remaining}
parameters ::= {required ... % keyword ... & remaining}
parameters ::= {required ... & remaining}
parameters ::= { + optional ...}
parameters ::= { + optional ... % keyword}
parameters ::= { + optional ... % keyword ... & remaining}
parameters ::= { + optional ... & remaining}
parameters ::= { % keyword ...}
parameters ::= { % keyword ... & remaining}
parameters ::= { & remaining}

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-------------|-----|---|
| scope | environment | 0-1 | An environment. If not supplied, a new one is made. |
| name | literal | 0-1 | The name of the function. If not supplied then the name shall follow the pattern fn-<number> . |
| parameters | list | 0-1 | A list containing variables, keywords, sigils, and keyword symbol associations. |
| returning | expression | 0-1 | The expression “ returns taxon ”, where taxon is a taxon or meron. |
| expression | variant | 1+ | One or more expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|--------------------------|
| literal | The name of the function |

Remarks

Functions are defined using the **function** intrinsic function. If the name is supplied then the function is a user named function (also called an **exonym**), otherwise it is an automatically named function (called an **esonym**). If a parameter list is supplied then it shall bind the formal parameter variables to any actual parameter expressions provided. The first variables in the parameter list are required. If a plus sign is provided, then those variables following the plus sign are optional. Each optional parameter may have a default value (specified with an equal sign and the value). If no default value is specified, the default value shall be **nil**. If a percent sign is supplied the next variables are keyword variables where the supplied slot shall have the same name as the symbol parameter excepting the first character of the name which shall be a colon for the slot and a question mark for the symbol. If an ampersand is provided, then the symbol following it is bound to a list of remaining (variadic) actual parameter values. The entire parameter list may be omitted.

If the literal **returns** follows the (non)extant parameter list then the next literal is either a taxon or a meron, which restricts the result of the function. The function shall be accessible from other modules by prefixing it with the module name in which it was defined.

Example

```
> (function {?n} (+ 1 ?n))  
. : fn-1  
  
> (function plusN {?n + ?i = 1} (+ ?n ?i))  
. : plusN  
  
> (function (scope) addAll {& ?nums} (apply + ?nums))  
. : addAll  
  
> (fn-1 100)  
. : 101  
  
> (plusN 100)  
. : 101  
  
> (function tryParseNum {String ?s}  
    (let ?convertible (convertible-p ?s number))  
    {?convertible (if ?convertible (@ (take ?s))))})  
  
> (tryParseNum "foo")  
. : {false nil}  
  
> (tryParseNum "123")  
. : {true 123}
```

```
> (function dont {{try ?what} + {at ?where = nil}}
  (apply (given {?s}(tell user ?s))
         ($ don't try ?what (unless (any ?where) "" ($ at ?where)))))

.: dont

> (dont try "skydiving into a volcano")

don't try skydiving into a volcano
.: told

> (dont try this at home)

don't try this at home
.: told
```

Related

arguments, call, extend, functions, junction, macro, module, modules, parameters, procedure

functions

Returns a list of functions defined in a module.

Syntax

(functions *module*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|---|
| module | literal | 0-1 | A module. If not provided defaults to the current module. |

Results

| Data Type | Description |
|-----------|--|
| list | A list of functions defined in the module. |

Remarks

The module must be a literal in the list obtained by calling the (**modules**) function. If the module is not provided this function uses the current module (obtained via the (**use**) function).

Example

```
> (module Mine
    (function foo {} bar))

.: Mine

> (functions Mine)

.: {foo}

> (use User)

.: User

> (function hello {} world)

.: hello

> (functions)

.: {hello}
```

Related

function, module, modules, variables

gather

Returns a sequence of elements that satisfy a predicate.

Syntax

(**gather** *symbol* ... *binding* *sequence* *expression* ... *test*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , or over . |
| sequence | sequence | 1 | A sequence (string or list) |
| expression | variant | 0+ | An expression. |
| test | truth | 1 | A truth expression |

Results

| Data Type | Description |
|-----------|---|
| sequence | A sequence containing the elements that satisfy the test. |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If there are insufficient elements to bind to the variables then the function fails. The expressions and test are evaluated. If the last value returned is a true truth value, the item is concatenated into a result list, otherwise the item is excluded from the result list. Finally, the result list is returned.

Example

```
> (gather ?n in {-3 -2 -1 0 1 2 3 4} (even-p ?n))  
.: {-2 0 2 4}  
  
> (gather ?x ?y per {{4 2}{7 3}{1 2}{15 5}} (divisible-p ?x ?y))  
.: {{4 2}{15 5}}
```

Related

coalesce, collect, filter

generator

Creates a series generator.

Syntax

(generator *name* *parameters* *expression* ... **)**

```
parameters ::=  
parameters ::= {}  
parameters ::= {required ...}  
parameters ::= {required ... + optional ...}  
parameters ::= { + optional ...}  
parameters ::= {required ... + optional ... & remaining}  
parameters ::= { & remaining}
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| name | literal | 0-1 | The name of the function. If not supplied then the name shall follow the pattern gn-<number> . |
| parameters | list | 0-1 | A list containing variables, keywords, sigils, and keyword symbol associations. |
| expression | variant | 1+ | One or more expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| generator | A function that returns a series. |

Remarks

If the name is not supplied then the generator is automatically named (auto-named). If a parameter list is supplied then it shall bind the formal parameter variables to any actual parameter expressions provided in the call to the generator. The first variables in the parameter list are required. If a plus sign is provided, then those variables following are optional. If an asterisk is provided, then keyword and values are expected. Keyword parameters (literal symbol pairs) may be provided as required or optional. If an ampersand is provided, then the symbol following is bound to a list of remaining actual parameter values. The entire parameter list may be omitted.

Example

```
> (generator primeNumbers
  (so {?primes {}}
    (step ?i from 2 to infinity
      (if (none ?p in ?primes (divisible-p ?i ?p))
        (add ?primes ?i)
        (yield ?i)))))

.: primeNumbers

> (collect ?p in (primeNumbers)
  (if (> ?p 20) (break ?p) else ?p))

.: {2 3 5 7 11 13 17 19 23}

> (let ?series (series (primeNumbers)))

.: true

> (if (next ?series) (this ?series))

.: 2

> (if (next ?series) (this ?series))

.: 3

> (if (next ?series) (this ?series))

.: 5

> (generator {?interrupt}
  (yield 1)
  (yield 2)
  (if ?interrupt (done))
  (yield 3)
  (yield 4)
  (done))

.: gen-1

> (collect ?x in (gen-1 true) ?x)

.: {1 2}

> (collect ?x in (gen-1 false) ?x)

.: {1 2 3 4}

> (gen-1 true)

.: {...}
```

Related

[done](#), [is](#), [next](#), [reset](#), [this](#), [yield](#)

get

Retrieves a value using one or more keys.

Syntax

(**get** *container key ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---------------------------|
| container | thing | 1 | A sequence or assortment. |
| key | variant | 1+ | A key or function. |

Results

| Data Type | Description |
|-----------|---|
| value | The value resulting from the retrieval. |

Remarks

Consecutively applies the next key to each resultant value. Terminates when all keys are exhausted or when the function fails.

Example

```
> (get (new (relation Box :Length 24 :Width 12 :Height 9)) :Width ++ ++)
.: 14

> (relation Person :Name :Friend)

.: Person

> (so {?person nil}
      (for ?name in {Jane John Sally Chuck Martha Shane Todd}
          (tie ?person (new Person :Name ?name :Friend ?person))))
.: Person_7

> (get Person_7 :Friend :Name)

.: Shane

> (get Person_7 :Friend :Friend :Name)

.: Martha
```

Related

@ *element, deq, enq, get, let, the, with*

given

Creates an anonymous function.

Syntax

(given { *parameter* ... } *expression* ...)

parameter ::= *required* ...
parameters ::= *required* ... + *optional* ...
parameters ::= + *optional* ...
parameters ::= *required* ... + *optional* ... & *remaining*
parameters ::= & *remaining*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--|
| parameters | list | 1 | A list containing zero or more parameters and sigils |
| expression | variant | 0+ | Expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|---|
| anonym | Returns the anonym the expression represents. |

Remarks

An anonym is an anonymous function. An anonym is equivalent to **lambda** in the Lisp programming language, and can be used in most places a function name is required. Anonyms cannot be recursive, since they are unnamed. Continuations may be taken within an anonym. Neither do anonyms use tags that can be reference by a **return**. If no parameters are specified, then each expression is executed in succession and the value of the last expression is returned. If parameters are specified, then arguments must be provided in the invocation. Anonyms are ephemeral because function definitions are not maintained for them.

Example

```
> ((given {?s} (tell user ($ ?s \n))) "Hello, world.")

Hello, world.
.: told

> (map {One Two Three Four Five}
  (given {?x} (tell user ($$ ?x "... \n)))

One...
Two...
Three...
Four...
Five...
.: told

> (filter (range 1 to 40) (given {?n} (and (even-p ?n) (divisible-p ?n 7))))
.: {14 28}
```

Related

[wait](#), [abort](#), [concurrent](#), [await](#), [complete](#), [done-p](#), [reclaim](#), [task](#), [use](#)

global

Creates global variables.

Syntax

(global *assignment* ... **)**

assignment ::= *symbol* *value*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------------|
| assignment | expression | 1+ | A symbol value pair. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | Returns the value true . |

Remarks

Creates global variables.

Example

```
> (global ?START_OF_TASK 1)
.: true

> (global ?FIRST_NAME "Jane"
      ?LAST_NAME "Smith"
)
.: true
```

Related

<-- before tie, --> after tie, constant, dynamic, local, let, modular, put, suppose, swap, undeclare

go

Transfers control to a function.

Syntax

(**go** *function argument ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------|
| function | function | 1 | A function. |
| argument | variant | 0+ | An argument to the function. |

Results

| Data Type | Description |
|-----------|---------------------|
| variant | Returns the result. |

Remarks

Transfers control to the specified function by unwinding the current function from the call graph then jumping to the specified function.

Example

```
> (relation Queue   :Items {} :Capacity 100
    !count (given {?me} (# (:Items ?me)))
    !full-p (given {?me} (>= (!count ?me) (:Capacity ?me)))

.: Queue

> (function produce {?q}
  (while (not (!full-p ?q))
    (repeat (random 1 to (- (:Capacity ?q) (!count ?q)))
      (enq ?q :Queue each (new Item))))
    (go consume ?q))

.: produce

> (function consume {?q}
  (while (more-p (:Queue ?q))
    (repeat (random 1 to (# (:Queue ?q)))
      (deliver (pop (:Queue ?q))))
    (go produce ?q))

> (produce (new Queue))
```

Related

continuation, continue

grok

Evaluates expressions in a url or file.

Syntax

(**grok** *what*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|--|
| what | variant | 1 | A uniform resource locator or file resource. |

Results

| Data Type | Description |
|-----------|--|
| variant | Returns the last expression evaluated. |

Remarks

This function opens the url or file, and evaluates each expression therein.

Example

```
> (grok (file name "Module1.th"))
.: Module1
> (grok (url "file://MyCodeSite.com/Module2.th"))
.: Module2
```

Related

eval, file, posit, url

group

Combines sublists by position or key.

Syntax

(group *list* **by** *key* ... **into** *symbol* *expression* ...))

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| list | list | 1 | A list containing sublists or tuples. |
| by | literal | 1 | The literal by . |
| keys | list | 1 | A list containing either slots, integer positions, or transform functions that return a value given an element of the list. |
| into | literal | 1 | The literal into . |
| symbol | symbol | 1 | A symbol storing each grouping. |
| expression | variant | 0+ | An expression to be evaluated. |

Results

| Data Type | Description |
|-----------|--------------------------------|
| list | A list of the last expression. |

Remarks

Each key is either a numeric position, slot or method name, or a transform function which when applied to a list element shall return a value. If **as** is not provided, then a list of the intermediate groupings is returned. If keyword **as** is provided, then the subsequent expressions shall have the grouping symbol available for evaluation and the final expression is collected into a list.

Example

```
> (group {{1 a b c}{2 b c a}{3 c a b}{4 a b c}{5 b c a}{6 c a b}} by {2 3} into ?g)
.: {{1 a b c}{4 a b c}}{{2 b c a}{5 b c a}}{{3 c a b}{6 c a b}}
> (group
  [[Fruit :Color red :Name cherry][Fruit :Color red :Name apple]
   [Fruit :Color red :Name tomato][Fruit :Color yellow :Name banana]]
  by {:Color} into ?g
  [R :Color (:Color (@ ?g)) :Count (# ?g)])
.: {[R :Color red :Count 3] [R :Color yellow :Count 1]}
```

Related

coalesce, collect, count, gather, given, separate, sort, with

has

True if a key is present in an assortment.

Syntax

(**has** *assortment key*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--------------------------|
| assortment | assortment | 1 | A sequence or assortment |
| key | variant | 1 | A key. |

Results

| Data Type | Description |
|-----------|--|
| truth | Returns true if the key is present. or false if the key is absent. |

Remarks

None

Example

```
> (lexicon :A 1 :B 2 !m X_foo :C 3)
.: Lexicon-1

> (has Lexicon-1 :C)
.: true

> (has Lexicon-1 !m)
.: true

> (has Lexicon-1 !q)
.: false
```

Related

beyond, lacks, within

hash

Computes a hash code.

Syntax

(hash *value option ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| value | variant | 1 | A value |
| option | expression | 0+ | A key value pair size number - a hash table size (default 1000) seed number - a prime number seed. |

Results

| Data Type | Description |
|-----------|----------------|
| integer | The hash value |

Remarks

Returns a hash value between zero and the given size, based on the given prime number seed. The size and seed can be omitted.

Example

```
> (hash {a b c})  
. : 822826764062  
  
> (hash {a b c} size 100)  
. : 62  
  
> (hash {a b c} size 1000 prime 3)  
. : 186
```

Related

array, collection, lexicon

help

Describes a function.

Syntax

(help *function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------------------|
| function | literal | 0-1 | The name of a function. |

Results

| Data Type | Description |
|-----------|--------------|
| nil | Returns nil. |

Remarks

Prints the function documentation and returns nil.

Example

```
> (help)

Enter expressions followed by a blank line. Type (help), (copyright) or (license)
for information. Type (grok "file.theory") to read a file. Type (bye) to exit.

.: nil

> (help help)

(intrinsic Base.help  {+ ?function} returns NIL ...)

  Returns documentation about a function. If no function is provided, general help
is displayed.

.: nil
```

Related

[about](#), [bye](#), [copyright](#), [license](#)

here

Adds variables in parallel to the current environment.

Syntax

(**here** *assignment* ...)

assignment ::= *symbol* *value*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--------------------------|
| assignment | expression | 1+ | A symbol and value pair. |

Results

| Data Type | Description |
|-----------|--------------|
| truth | Returns true |

Remarks

Concurrently creates variables in the current environment. Each symbol shall replace any extant symbol with the same name in the current environment. Each symbol assignment occurs either in parallel with, or in isolation to, the other symbol assignments.

Example

```
> (let ?start 1 ?end 100 ?range (range ?start to ?end))  
.: true  
  
> (undeclare {?start ?end ?range})  
.: undeclared  
  
> (here ?start 1 ?end 100 ?range (range ?start to ?end))  
.: [Failure :Name UnboundSymbol :Text "The symbol ?start is unbound"]  
  
> (do (here ?start 1 ?end 100)  
      (step ?i from ?start to ?end (do nothing)))  
.: nothing
```

Related

-- before tie, --> after tie, assume, constant, dynamic, given, global, local, put

hyperlink

Creates a hyperlink.

Syntax

(**hyperlink** *option ...*)
(**hyperlink** *address*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|--|
| address | string | 0-1 | The url address. |
| option | expression | 0+ | A key value pair. One of user <i>credentials</i> -user name and password host <i>host</i> - host computer name (default is localhost) port <i>number</i> – a port number. (default is 2001) path <i>path</i> – a path query <i>query</i> – a query element fragment <i>fragment</i> – a fragment element |

Results

| Data Type | Description |
|-----------|-----------------|
| service | A file resource |

Remarks

Returns a hyperlink.

Example

```
> (function reply {?target ?request} (tell ?target "goodbye."))

.: reply

> (service (var ?service (hyperlink "http://localhost:4500/foxyy")) reply)

.: Service-1

> (ask ?service "hello")

.: "goodbye."

> (cancel Service-1) (free ?service) (free Service-1)

.: cancelled freed freed
```

Related

close, erase, path, pipe, read, socket, write

id

Returns a resource number.

Syntax

(id *resource* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| resource | resource | 1 | A resource. |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| variant | The resource number or nil . |

Remarks

None

Example

```
> (new (relation Q))

.: Q_1

> (id Q_1)

.: 1

> (id [Q ^ Q_2])

.: 2

> (id file-27)

.: 27

> (id foo)

.: nil
```

Related

^ thought, id, uid, unique

identical-p

True if all values occupies the same memory location.

Syntax

(identical-p value value ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|------------------------|
| value | value | 2+ | Values to be compared. |

Results

| Data Type | Description |
|-----------|----------------------------------|
| truth | True if all values are identical |

Remarks

Returns true if each value is identical to its successor from left to right, and false otherwise. All values must be of the same type, and must occupy the same address in memory. This function short circuits if false.

Example

```
> (identical-p a a a a)  
.: true  
  
> (identical-p a (clone a) a a)  
.: false  
  
> (identical-p "johnny" "ralph" "sam")  
.: false  
  
> (identical-p 3 3)  
.: true  
  
> (identical-p 3 (clone 3))  
.: false
```

Related

< less , <= less or equal, > greater, >= greater or equal, /= unequal, canonify, identity

identity

Returns the value itself.

Syntax

(identity *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | Any value |

Results

| Data Type | Description |
|-----------|--------------------|
| variant | Returns the value. |

Remarks

None.

Example

```
> (identity a)
.: a
> (identity {a b c})
.: {a b c}
> (identity (++ 0))
.: 1
```

Related

canonify, identical-p

idle

Pauses for a specified interval.

Syntax

(idle duration)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--------------------------------|
| duration | instant | 1 | The quantity of time to pause. |

Results

| Data Type | Description |
|-----------|----------------------|
| literal | Returns ready |

Remarks

None

Example

```
> (idle \@s10)
.: ready
```

Related

moment, seconds, tick

if

Branched conditional evaluation.

Syntax

(if condition expression ... or-clause ... else-clause)

or-clause ::= **or** condition expression ...

else-clause ::= **else** expression ...

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|------------|-----|---|
| condition | truth | 1 | A truth expression |
| expression | expression | 1+ | Any expression |
| or-clause | expression | 0+ | The literal or followed by a condition and zero or more expressions. |
| else-clause | expression | 0-1 | The literal else followed by one or more expressions |

Results

| Data Type | Description |
|-----------|---|
| variant | The last value of the executed clause or nil if no conditions were true and no else clause was provided. |

Remarks

None

Example

```
> (global ?N 100)

.: true

> (if (> ?N 100) over
     or (< ?N 100) under
     else exactly)

.: exactly
```

Related

and, case,either, not, or, unless

imaginary

Converts a value to an imaginary number.

Syntax

(imaginary *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 1 | A number or the literal minimum or maximum . |

Results

| Data Type | Description |
|-----------|--------------------|
| integer | An integer number. |

Remarks

If the number cannot be converted to an imaginary number, the function fails. If the value is complex, then the imaginary portion is selected.

Example

```
> (imaginary {a b c})  
.: [Failure :Name ArgumentValue :Text "{a b c} is not a number."]  
  
> (imaginary "a b c")  
.: [Failure :Name ArgumentValue :Text "{a b c} is not a number."]  
  
> (imaginary 500)  
.: 500i  
  
> (imaginary 500.1)  
.: 500.1i  
  
> (imaginary "-23.4")  
.: -23.4i
```

Related

[complex](#), [integer](#), [long](#), [rational](#), [real](#)

in

True if a value is in a sequence.

Syntax

(**in** *value sequence option ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| value | variant | 1 | A value. |
| sequence | sequence | 1 | A sequence. |
| option | expression | 0-2 | A key value pair. Any of depth <i>number</i> - Number or # (for any depth). Default is 1. transform <i>function</i> - A function to transform the element.. |

Results

| Data Type | Description |
|-----------|--|
| truth | Returns true if the value is in the sequence, false otherwise. |

Remarks

None.

Example

```
> (in d {a b c d e f g})  
.: true  
  
> (in "a" "cbazyx")  
.: true
```

Related

beyond, includes, excludes, has, out, within

includes

True if a value is in a sequence.

Syntax

(includes *sequence element option ...* **)**

Module

Base

Parameters

None

| Name | Data Type | Qty | Description |
|----------|------------|-----|--|
| sequence | sequence | 1 | A sequence. |
| element | variant | 1 | A value. |
| option | expression | 0-2 | A key value pair. Any of depth <i>number</i> - Number or # (for any depth). Default is 1. transform <i>function</i> - A function to transform the element. |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| truth | True if the item is in the container. |

Remarks

If value and sequence are strings, then a string search is performed. If sequence is a list and depth is greater than one, then subsequences shall be checked for the value. A depth of # implies search to any depth.

Example

```
> (includes {a b c d e f g} d)
.: true

> (includes "cbazyx" "w")
.: false

> (includes "cbazyx" 25 transform (given {?c} (++ (- (unicode ?c) (unicode "a")))))
.: true
```

Related

[excludes](#), [in](#), [out](#)

index

Creates an index on slots of a relation.

Syntax

(index *relation slot ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------------|
| relation | literal | 1 | A meron. |
| slot | literal | 1+ | The name of a slot in the relation |

Results

| Data Type | Description |
|-----------|-----------------------|
| literal | The name of the index |

Remarks

Creates an index for the specified slots to expedite value retrieval.

Example

```
> (relation X :A :B :C)
.: X

> (index X :B)
.: X_Index_1

> (index X :A :B)
.: X_Index_2

> (new X :A 1 :B 2)
.: X_1

> (which X ^ as ?x :B 1 list ?x)
.: {X_1}
```

Related

[drop](#), [indices](#), [relation](#)

indices

Returns a list of indices for a relation.

Syntax

(indices relation)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| relation | relation | 1 | A relation. |

Results

| Data Type | Description |
|-----------|---|
| list | A list of indices associated with the relation. |

Remarks

None.

Example

```
> (relation X :A :B :C)
.: X
> (indices X)
.: {}
> (index X :B)
.: X_Index_1
> (index X :A :B)
.: X_Index_2
> (indices X)
.: {X_Index_1 X_Index_2}
> (drop (indices X))
.: 1
```

Related

[drop](#), [index](#), [relation](#)

infer

Applies the effects of matched rules.

Syntax

(infer *problem*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--|
| problem | problem | 0-1 | A specific problem identifier Default is all if not supplied. |

Results

| Data Type | Description |
|-----------|---|
| integer | Returns the number of rules processed . |

Remarks

If no problem is provided, the (**problems**) function is called and the matched rules in each open problem are processed. Returns the total rules processed.

Example

```
> (domain FixIt
  (relation Car :Age :Condition)
  (rule
    if
      [Car :Age /= old :Condition = good]
    do
      (tell user ($ ($$ \n that) is possible. \n)
      then done)))
  .: FixIt

> (problem state [Car :Age new :Condition good])
.: Problem_1

> (loop (match Problem_1) while (> (infer Problem_1) 0))
that is possible.

.: true
```

Related

cancel, domain, domains, infer, match, problem, problems, rule, rules

infix

Creates a new sequence with interposed delimiters.

Syntax

(infix sequence delimiter)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence. |
| delimiter | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|---|
| list | A list with delimiters between the elements |

Remarks

A delimiter is placed between each element. If the sequence is a string, the delimiter shall be converted into a string. If the sequence has less than two elements, no changes are made.

Example

```
> (infix {a b c d} * )
.: {a * b * c * d}

> (infix "abcd" " ")
.: "a, b, c, d"

> (infix {} ",")
.: {}

> (infix "" ",")
.: ""
```

Related

\$concatenate, split

inside

True if a position is inside a sequence.

Syntax

(inside *sequence position* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| sequence | sequence | 1 | A sequence. |
| position | variant | 1+ | A number or coordinate (i.e. a list of numbers). |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if the position is in the sequence, false otherwise. |

Remarks

None.

Example

```
> (inside "not empty" 5)
.: true

> (inside {not empty} 1)
.: true

> (inside "" 1)
.: false

> (inside {} 5)
.: false

> (inside {{a b}{c d}{e f}} {3 2})
.: true

> (inside {{a b}{c d}{e f}} {7 4})
.: false
```

Related

@ *element*, beyond, length, empty, more

insert

Creates a new sequence by inserting values.

Syntax

(insert *sequence position value ...* **)**

entry ::= value position

entry ::= 388value

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | sequence | 1 | A sequence |
| position | integer | 1 | The position to begin inserting the values. |
| value | variant | 1+ | A value. |

Results

| Data Type | Description |
|-----------|-----------------|
| sequence | A new sequence. |

Remarks

This function creates a new sequence. To modify a sequence use the **push** function.

Example

```
> (insert "abcdef" 1 g)
.: "gabcdef"
> (insert "abcdef" # g)
.: "abcdefg"
> (insert {a b c} 1 d e f)
.: {d e f a b c}
> (insert "abc" 1 d e f)
.: "defabc"
```

Related

size, add, cut, deq, enq, get, key, keys, push, put, values, zap

insort

Inserts a value into a sorted sequence.

Syntax

(**insort** *sequence* *value* *option ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| sequence | sequence | 1 | A sequence |
| value | value | 1 | A value |
| option | expression | 0+ | A key value pair where the key is comparer <i>function</i> - a function returning <, =, > unique yes no - Allows duplicate values. Default yes. |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| sequence | The new sequence after insertion. |

Remarks

Inserts the item in sorted order in the sequence. If the sequence is a list and first element is a number then the list is kept numerically sorted, otherwise it is alphabetically sorted.

Example

```
> (inserir {a b c d e x} q)
.: {a b c d e q x}

> (inserir "abcde" "z")
.: "abcdez"

> (function desc {?x ?y}
  (if (> ?x ?y) < or (= ?x ?y) = else >))

.: desc

> (inserir {108 79 33} 84 comparar desc)
.: {108 84 79 33}
```

Related

[insert](#)

instantiate

Creates an expression with premises in place of thoughts.

Syntax

(instantiate *expression* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| expression | variant | 1 | A SubThought or a list containing thoughts. |

Results

| Data Type | Description |
|-----------|----------------------------|
| list | A list containing premises |

Remarks

Replaces thoughts with corresponding premises within the supplied expression.

Example

```
> (relation E :Value)
.: E
> (new E)
.: E_1
> (relation B :Slot)
.: B
> (new B)
.: B_1
> (instantiate {E_1 {:Foo {B_1 x}}})
.: {[E ^ E_1 :Value nil] {:Foo {[B ^ B_1 :Slot nil] x}}}
```

Related

^ thought, get, id, premise

integer

Converts a value to an integer.

Syntax

(integer *value* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 1 | A number value or the literal minimum or maximum |

Results

| Data Type | Description |
|-----------|--------------------|
| integer | An integer number. |

Remarks

If the value is **minimum** (or **maximum**) the minimum (or maximum) integer is returned; otherwise, if the value cannot be converted to an integer, the function fails. The fractional part of the number is truncated. An integer is a signed 64 bits number.

Example

```
> (integer {a b c})  
.: [Failure :Name ArgumentValue :Text "{a b c} is not a number."]  
  
> (integer "a b c")  
.: [Failure :Name ArgumentValue :Text "'a b c' is not a number."]  
  
> (integer 500.3)  
.: 500  
  
> (integer "23.4")  
.: 23
```

Related

ceiling, complex, floor, is, long, radix, rational, real, round, truncate, unsigned

interior

Creates an expression from a sequence or assortment.

Syntax

(interior value)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 1 | A sequence, assortment, or expression. |

Results

| Data Type | Description |
|------------|---|
| expression | returns the contents of the value as an expression. |

Remarks

If value is a sequence or assortment then the contents are returned as an expression, otherwise, an empty expression is returned.

Example

```
> (interior {the quick brown fox})  
.: the quick brown fox  
  
> (interior {})  
.:  
  
> (void-p (interior {}))  
.: true  
  
> (interior (lexicon 1 a 2 b 3 c 4 d))  
.: 1 a 2 b 3 c 4 d  
  
> (interior (collection 1 2 3 a b c))  
.: 1 2 3 a b c
```

Related

& merge, && abridge, expression, is, list

interleave

Interleaves arguments into a new sequence.

Syntax

(intersection *sequence sequence ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 2+ | A sequence |

Results

| Data Type | Description |
|-----------|---------------------------|
| sequence | The interleaved sequence. |

Remarks

Returns a single sequence interleaving all sequences. If there are insufficient elements, NOTHING is supplied as the element. If the first element is not a sequence, then a list is returned. The sequence type shall be that of the first element.

Example

```
> (interleave {1 2 3} {4 5 6} {7 8 9} {10})  
.: {1 4 7 10 2 5 8 3 6 9}  
  
> (interleave [1 2 3] [4 5 6] [7 8 9] [10])  
.: [1 4 7 10 2 5 8 3 6 9]  
  
> (interleave |1 2 3| |4 5| |6| |7 8 9| |10|)  
.: |1 4 7 10 2 5 8 3 6 9|  
  
> (interleave "abc" "def" "ghi" "j")  
.: "adgjbehcfi"
```

Related

&& *abridge*, *mid*, *insert*, *pop*, *push*, *swap*, *transfer*

intersection

Creates a sequence of common elements.

Syntax

(intersection *sequence sequence ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------|
| sequence | sequence | 2+ | A list or string |

Results

| Data Type | Description |
|-----------|------------------------|
| sequence | A list of common items |

Remarks

Returns the intersection of the provided sequences, i.e., the elements common to all sequences.

Example

```
> (intersection {1 2 3 4} {2 3 4 5})
.: {2 3 4}

> (intersection {2 3 4 5} {4 5 6 7})
.: {4 5}

> (intersection {1 2 3 4} {2 3 4 5} {3 4 5 6} {4 5 6 7})
.: {4}

> (intersection "abc" "bcd" "cde")
.: "c"
```

Related

[common](#), [difference](#), [union](#)

intersects-p

True if any elements intersect.

Syntax

(intersects-p *set1* *set2*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|-------------|
| set1 | sequence | 1 | A sequence. |
| set2 | sequence | 1 | A sequence |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the sets have any elements in common. |

Remarks

None.

Example

```
> (intersects-p {a b c d e f g} {a b x})  
.: true  
  
> (intersects-p {a b c d e f g} {a b d})  
.: true  
  
> (intersects-p "the quick brown" "ick bro")  
.: true  
  
> (intersects-p {a b c} {d e f})  
.: false  
  
> (intersects-p {1 2 3} {4 5 6})  
.: false
```

Related

[disjoint-p](#), [subset-p](#)

interval

Creates a time interval.

Syntax

(interval start finish)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--|
| start | time | 0-1 | A time value. (Default is neternity) |
| finish | time | 0-1 | A time value. (Default is eternity) |

Results

| Data Type | Description |
|-----------|--------------|
| interval | An interval. |

Remarks

If no parameters are provided, start shall default to negative eternity (**neternity**), and finish will default to positive **eternity**. If only start is provided, finish shall default to positive **eternity**. If finish is less than start, the positions shall be swapped so that start is always less than or equal to finish.

Example

```
> (interval 2014-01-01 2015-12-31)
.: \@i{\@d2014-01-01T00:00:00.000 \@d2015-12-31T00:00:00.000}

> (interval \@m2015000000000000)
.: \@i{\@m2015000000000000 eternity}

> (interval \@t63573798191026000 neternity)
.: \@i{neternity \@t63573798191026000}

> (interval)
.: \@i{neternity eternity}

> (interval \@e600  \@e300)
.: \@i{\@e300  \@e600}
```

Related

[date](#), [epoch](#), [moment](#), [seconds](#), [tick](#)

into

Creates new thoughts based on existing thoughts.

Syntax

(into relation thoughts)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--------------------|
| relation | relation | 1 | A relation . |
| thoughts | list | 1 | A list of records. |

Results

| Data Type | Description |
|-----------|--|
| literal | The name of the relation or structure. |

Remarks

If the thoughts contain slots that are in the relation, the slot values are copied to new thoughts.

Example

```
> (relation X :A :B :C)
.: X

> (repeat 10
    (new X :A (random 1 to 100) :B (random 50 to 500) :C (random 0 to 1)))
.: X_10

> (into (relation Y uses {X}) (with X))
.: Y

> (which Y)
.: {Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7 Y_8 Y_9 Y_10}
```

Related

[knew](#), [new](#)

invoke

Invokes a call.

Syntax

(**invoke** *call environment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|--------------------------------------|
| call | call | 1 | A call. |
| environment | environment | 0-1 | An environment (defaults to (scope)) |

Results

| Data Type | Description |
|-----------|-------------------------------|
| variant | A value returned by the call. |

Remarks

Invokes the call using the provided (or the current) environment.

Example

```
> (relation A
  :Datum 0
  !incr (function {?me}
    (bind ?me :Datum ?n)
    (put ?me :Datum (++ n))))  

.: A  

> (new A)  

.: A_1  

> (invoke (call !incr A_1) (scope))  

.: 1  

> (invoke (call !incr A_1) (scope))  

.: 2  

> (invoke (call @ "hello"))  

.: "h"
```

Related

apply, call, closure, eval, function, macro, scope, supply

is

Applies the predicate to the value returning true or false..

Syntax

(is *value predicate*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---|
| value | variant | 1 | A value |
| predicate | literal | 0-1 | a unary function returning true or false. |

Results

| Data Type | Description |
|-----------|--|
| truth | True if the value is true, or if the applied unary predicate returns true. |

Remarks

None.

Example

```
> (is true)
.: true

> (is false)
.: false

> (is 41 even-p)
.: false

> (is 9 (given {?n} (divisible-p ?n 3)))
.: true

> (is foo null-p)
.: false

> (is nil null-p)
.: true
```

Related

convert, datatype, known, the, type

junction

Creates a public function in a module with arguments evaluated in parallel.

Syntax

(junction scope name parameters expression ...)

parameters ::=
parameters ::= {}
parameters ::= {required ... }
parameters ::= {required ... + optional ... }
parameters ::= { + optional ... }
parameters ::= {required ... + optional ... & remaining }
parameters ::= { & remaining }

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-------------|-----|---|
| scope | environment | 0-1 | An environment. If not supplied then a new one is made. |
| name | literal | 0-1 | The name of the function. If not supplied then the name shall follow the pattern fn-<number> . |
| parameters | list | 0-1 | A list containing variables, keywords, sigils, and keyword symbol associations. |
| expression | variant | 1+ | One or more expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|--------------------------|
| literal | The name of the function |

Remarks

Junctions are defined using the **junction** intrinsic function. A junction is merely a function that evaluates all its arguments in parallel. A parallel argument evaluation the name is supplied then the function is a user named function (also called an **exonym**), otherwise it is an automatically named function (called an **esonym**). If a parameter list is supplied then it shall bind the formal parameter variables to any actual parameter expressions provided. The first variables in the parameter list are required. If a plus sign is provided, then those variables following the plus sign are optional. Each optional parameter may have a default value (specified with an equal sign and the value). If no default value is specified, the default value shall be **nil**. If an ampersand is provided, then the symbol following it is bound to a list of remaining (variadic) actual parameter values. Keyword parameters (literal symbol pairs) may be provided as required or optional. The entire parameter list may be omitted. The function shall be accessible from other modules by prefixing it with the module name in which it was defined.

Example

```
> (junction add_it {integer ?a integer ?b integer ?c} (+ ?a ?b ?c))
.: add_it
> (add_it 1 2 3)
.: 6
> (function fib {?n} (on (<= ?n 1) 1 (add_it (fib (- ?n 1)) (fib (- ?n 2)))))
.: fib
> (function fib2 {?n} (on (<= ?n 1) 1 (+ (fib (- ?n 1)) (fib (- ?n 2)))))
.: 101
> (macro duration {?x} (so {?now (jiffy) ?value (eval ?x) ?done (jiffy)}
  (- ?ticks ?now)))
.: duration
> (duration (fib2 10))
.: 50
> (duration (fib 10))
.: 5
```

Related

[arguments](#), [call](#), [extend](#), [functions](#), [macro](#), [module](#), [modules](#), [parameters](#)

key

Finds the key for a value in an assortment.

Syntax

(key *assortment value* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------|
| assortment | assortment | 1 | An assortment. |
| value | variant | 1 | A value |

Results

| Data Type | Description |
|-----------|------------------------|
| variant | The key for the value. |

Remarks

If a key does not exist for the value, the function fails. It is advised to test whether the value exists in the assortment by first using the **values** function.

Example

```
> (var ?s (lexicon a 1 b 2 c 3 d 4))
.: Lexicon-1

> (values ?s)
.: {1 2 3 4}

> (keys ?s)
.: {a b c d}

> (key ?s 4)
.: d

> (key ?s 5)
.: [Failure :Name NotFound "The value 5 was not found in the assortment."]
```

Related

@ *element*, position, values

keys

Returns a list of keys for an assortment.

Syntax

(keys *assortment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---------------|
| assortment | assortment | 1 | An assortment |

Results

| Data Type | Description |
|-----------|------------------|
| list | A list of values |

Remarks

All keys from the assortment are returned in a list.

Example

```
> (var ?s (lexicon a 1 b 2 c 3 d 4))  
.: Lexicon-1  
> (values ?s)  
.: {1 2 3 4}  
> (keys ?s)  
.: {a b c d}  
> (key ?s 4)  
.: d  
> (key ?s 5)  
. : [Failure :Name NotFound :Text "The value 5 was not in the assortment."]
```

Related

[key](#), [values](#)

keywords

Returns the list of Premise keywords.

Syntax

(keywords)

Module

Base

Parameters

None.

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | True if the value is a keyword. |

Remarks

None.

Example

```
> (keywords)
.: {\%eternity \%false \%impossible \%indefinite \%infinity \%neternity \%ninfinity
\%true \%undefined \%unknown}
```

Related

is

knew

Finds or creates a thought.

Syntax

(**knew** *relation criterion ...*)

criterion ::= slot referent

criterion ::= method function

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|---|
| relation | relation | 1 | The name of the relation to find |
| criterion | expression | 0+ | A slot referent pair or method function pair. |

Results

| Data Type | Description |
|-----------|-------------------------|
| thought | A known or new thought. |

Remarks

Looks for an existing thought that matches the criteria and returns the first match. If no thought matches the criteria, a new thought is created using the criteria.

Example

```
> (relation Pairs :A :B)  
.: Idea  
> (knew Pairs :A 1 :B 2)  
.: Pairs_1  
> (knew Pairs :A 1)  
.: Pairs_1  
> (knew Pairs :B 3)  
.: Pairs_2
```

Related

assert, assume, known, new, relation

knowledge

Finds or creates a knowledge base.

Syntax

(knowledge *option ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|---|
| option | expression | 0+ | <p>Key value pair. Any of the following</p> <p>name <i>name</i> - A literal to identify the knowledge base.</p> <p><u>Options for constructing a local knowledge base</u></p> <p>path <i>string</i> - the local path (required if constructing).</p> <p>unit <i>literal</i> - either TB, GB, MB (default)</p> <p>size <i>real</i> - the initial knowledge base (default is 5).</p> <p>max <i>real</i> - maximum kb size (default is nil)</p> <p>log <i>real</i> - initial log size. (default is 2).</p> <p>growth <i>real</i> - Resize percentage (default is 10).</p> <p><u>Options for connecting to an existing knowledge base</u></p> <p>connect <i>string</i> - use the specified connection string.</p> <p>poolszie <i>integer</i> - maximum # of connections</p> <p>provider <i>literal</i> - either mssql, odbc, maria, pgsql</p> <p><u>Options for building the connection string from elements.</u></p> <p>driver <i>string</i> - the driver</p> <p>server <i>string</i> - server address</p> <p>version <i>number</i> - server version</p> <p>class <i>class</i> - class name of driver</p> <p>host <i>address</i> - host address</p> <p>port <i>port</i> - server port for database service</p> <p>instance <i>instance</i> - server instance name</p> <p>integratedSecurity <i>truth</i> - true or false</p> <p>database <i>name</i> - database name</p> <p>user <i>username</i> - user name</p> <p>password <i>password</i> - user password</p> <p>trusted <i>yesorno</i> - trusted connection yes or no.</p> <p>source <i>name</i> - data source name</p> <p>protocol <i>protocol</i> - database protocol</p> <p>subprotocol <i>subprotocol</i> - database sub protocol</p> <p>subname <i>subname</i> - sub name</p> |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| literal | The name of the new knowledge base. |

Remarks

If **path** is provided, then **name** is required. If **data** is provided then a knowledge base must already exist at the specified data url.

Example

```
> (knowledge provider mssql connect "Data Source=MYHOST\SERVER; Initial Catalog=MyKB; Integrated Security=SSPI;")

.: Knowledge-1

> (knowledge name Mathematics path "c:/premise/kb/" units MB size 2 max 10
   log 1 growth 10)

.: Mathematics
```

Related

attach, conceive, detach, each, eradicate, knew, knowing, new, relation, which, with

known

Finds or creates a thought.

Syntax

(**known** *pattern* ... *option* ...)

pattern ::= [relation clause ...]
clause ::= slot referent
clause ::= slot **is** unary-predicate
clause ::= slot **not** referent
clause ::= slot binary-predicate referent
clause ::= slot n-ary-predicate referent-list
clause ::= method function

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|---|
| pattern | expression | 0+ | A relation clause pattern enclosed in square brackets. |
| option | expression | 0-2 | Key value pairs: limit number - maximum number of thoughts to return. scan number - maximum number of thoughts to search. |

Results

| Data Type | Description |
|------------|-------------------------------|
| expression | Returns the matched thoughts. |

Remarks

Looks for existing thoughts matching the criteria.

Example

```
> (relation Pairs :A :B)
.: Pairs

> (new Pairs :A 1 :B 2) (new Pairs :A 3 :B 4) (new Pairs :A 5 :B 2)
.: Pairs_1 Pairs_2 Pairs_3

> (known [Pairs :B 2] limit 1)
.: Pairs_1
```

Related

assert, assume, knew, new, relation, the, with

lacks

True if a key is absent from an assortment.

Syntax

(**lacks** *assortment* *key*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------|
| assortment | assortment | 1 | An assortment. |
| key | variant | 1 | A key. |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if the key is absent or false if the key is present. |

Remarks

None

Example

```
> (lexicon :A 1 :B 2 !m X_foo :C 3)
.: Lexicon-1

> (lacks Lexicon-1 :C)
.: false

> (lacks Lexicon-1 !m)
.: false

> (lacks Lexicon-1 !q)
.: true
```

Related

beyond, has, within

last

Creates a subsequence of the last elements.

Syntax

(last sequence count)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | sequence | 1 | A list or string |
| count | number | 0-1 | The number of elements to return. Default is 1. |

Results

| Data Type | Description |
|-----------|---------------------------|
| sequence | The selected sub sequence |

Remarks

None

Example

```
> (last {a b c})  
. : {c}  
  
> (last {a b c d e f} 3)  
. : {d e f}  
  
> (last "abcde" 2)  
. : "de"
```

Related

but, rest, top

left

True if values are equal or the second value is nil.

Syntax

(left *first second* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------------------|
| first | variant | 1 | A value to be compared. |
| second | variant | 1 | A value to be compared. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if all values are the same or if second is nil |

Remarks

Returns true if each value is the same or if the second value is **nil**, and false otherwise. Both values must be of the same type, and must be numbers, literals, or strings. Short circuits if false. Two items are equal if their representation is the same regardless of whether or not they occupy the same location in memory (viz. identical). Returns false if both values are false.

Example

```
> (left a a)
.: true

> (left "johnny" "ralph")
.: false

> (left 3 nil)
.: true

> (left nil 4)
.: false
```

Related

< less , <= less or equal, > greater, >= greater or equal, /= unequal, full, identical, right

let

Adds variables to the current environment.

Syntax

(**let** *manner* *assignment* ...)

assignment ::= *symbol value*

assignment ::= {*symbol* ... } *list*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------------|------------|-----|--|
| manner | literal | 0-1 | The manner in which the assignments are performed either the literal parallel or tandem (default if omitted) |
| assignment | expression | 1+ | A symbol and value pair. |

Results

| Data Type | Description |
|--------------|---------------|
| Truth | Returns true. |

Remarks

Sequentially creates variables in the current environment. Each symbol shall replace any extant symbol with the same name in the environment. If **parallel** is specified as the manner, the variables are defined in parallel, otherwise the variables are assigned in **tandem** (by default).

Example

```
> (let ?person DrWho)
  .: true
> ?person
  .: DrWho
> (let parallel ?x 1 ?y 2 ?z 3)
  .: true
```

Related

-- before tie, --> after tie, constant, dynamic, fix, global, local, set, tie

lexemes

Creates an uppercase string with spaces between words.

Syntax

(lexemes *string*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--------------------------|
| string | string | 1 | A string to be converted |

Results

| Data Type | Description |
|-----------|---------------------|
| string | A converted string. |

Remarks

Converts a string to uppercase with a single space between alphanumeric words. Removes punctuation.

Example

```
> (lexemes "The quick brown fox!! ")  
. : "THE QUICK BROWN FOX"
```

Related

char, format, lexemes, like, lowercase, ngrams, split, trim, unlike, unicode, uppercase

lexicon

Creates a lexicon.

Syntax

(lexicon *association* ... **)**

association ::= *key value*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|------------|-----|--------------------|
| association | expression | 0+ | A key and a value. |

Results

| Data Type | Description |
|-----------|----------------------------|
| lexicon | The newly created lexicon. |

Remarks

None.

Example

```
> (lexicon {a b c} "foo 123" bar "bar 456" baz (lexicon))
.: Lexicon-2

> (keys Lexicon-2)
.: {{a b c} bar baz}

> (values Lexicon-2)
.: {"foo 123" "bar 456" Lexicon-1}

> (@ Lexicon-2 {a b c})
.: "foo 123"

> (key Lexicon-2 Lexicon-1)
.: baz
```

Related

[lexicon](#)

lexicons

Creates a list of all lexicons.

Syntax

(lexicons)

Module

Base

Parameters

None.

Results

| Data Type | Description |
|-----------|-----------------------------|
| list | Returns a list of lexicons. |

Remarks

None.

Example

```
> (lexicons)
.: {}

> (lexicon k1 v1 k2 v2)
.: Lexicon-1

> (lexicons)
.: {Lexicon-1}
```

Related

[lexicon](#)

license

Prints the software license.

Syntax

(license)

Module

Base

Parameters

None

Results

| Data Type | Description |
|-----------|-------------|
| nil | nil |

Remarks

None.

Example

```
> (license)

Premise Community Edition

Copyright (c) 2013-2025 SubThought Corporation. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of the
Community edition of the software and associated documentation files (the
"Software"), to deal in the Software without restriction, including without
limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE
OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

.: nil
```

Related

about, bye, copyright

like

Compares a pattern to a sequence.

Syntax

`(like sequence pattern)`

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | sequence | 1 | A string or list of strings to be compared. |
| pattern | string | 1 | A regex pattern. |

Results

| Data Type | Description |
|-----------|---|
| variant | If sequence is a string, then a truth is returned. If sequence is a list, then a list of matches is returned |

Remarks

If the sequence is a list, then this function Creates a new list containing only those elements matching the pattern. If the sequence is a string, then this function returns true if the pattern matches the candidate, and false otherwise. Standard regular expression matching is used.

Example

```
> (like "abc" "bc")
.: true

> (like "abc" "[xy]")
.: false

> (like {"abc" "def" "xyz" "wxy"} "xy")
.: {"xyz" "wxy"}
```

Related

[unlike](#)

list

Creates a list.

Syntax

(list *value* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | Any value |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| list | Returns a list containing the values. |

Remarks

A list can also be created by simply enclosing the elements with curly braces, {}.

Example

```
> (list a b c {d})  
. : {a b c {d}}  
  
> (list "a b c")  
. : {"a b c"}  
  
> (list 500)  
. : {500}  
  
> (list foo)  
. : {foo}  
  
> {a b c {d}}  
. : {a b c {d}}  
  
> {"a b c"}  
. : {"a b c"}
```

Related

& merge, && abridge, inside, vector

literal

Creates a literal.

Syntax

(literal value)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--------------------------------------|
| value | variant | 1 | A string or literal to be converted. |

Results

| Data Type | Description |
|-----------|-------------|
| literal | A literal. |

Remarks

Only strings or literals can be converted to a literal. sequences, numbers and variables cannot be converted to literals. This function fails if the value cannot be converted.

Example

```
> (literal abc)
.: abc

> (literal 500)
.: [Failure :Name ArgumentValue :Text "Cannot convert '500' to a literal."]

> (literal "500")
.: [Failure :Name ArgumentValue :Text "Cannot convert '500' to a literal."]

> (literal "foo")
.: foo

> (literal (symbol bar))
.: [Failure :Name ArgumentValue :Text "Cannot convert '?bar' to a literal."]
```

Related

is

local

Creates a task level scope.

Syntax

(local *assignments expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-----------|-----|------------------------------|
| assignments | list | 1 | A list of symbol value pairs |
| expression | variant | 0+ | An expression |

Results

| Data Type | Description |
|-----------|---------------|
| truth | Returns true. |

Remarks

The new variables shall shadow any existing variables with the same name.

Example

```
> (function appendToFile {& ?messages}
  (local {?file (file name "myfile.out")})
  (try
    (seek ?file #)
    (for ?s in ?messages (write ?file ?s))
    (close ?file)
    appended)
  learn ?e
  (may (close ?file)) ; suppress further failure
  ?e)))
.: appendToFile
```

Related

<-- before tie, --> after tie, constant, dynamic, given, global, only, set

location

Sets a symbol to the current location.

Syntax

(**location** *symbol*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-----------------------------------|
| symbol | symbol | 1 | A symbol to contain the location. |

Results

| Data Type | Description |
|-----------|-------------------|
| variant | The result value. |

Remarks

This function saves the present program location in a symbol. A subsequent **resume** call shall reset the location to the one in the symbol for the program control to resume at the location.

Example

```
> (function makeSandwich {?where}
  (tell user "Made a sandwich.\n")
  (proceed ?where set ?sandwich made))

.: makeSandwich

> (function eatSandwich
  (tell user "Ate the sandwich.\n"))

.: eatSandwich

> (do
  (let ?sandwich none)
  (location ?kitchen)
  (if (= ?sandwich made)
    (eatSandwich)
    else
    (makeSandwich ?kitchen)))

Made a sandwich.
Ate the sandwich.
.: told
```

Related

[resume](#), [go](#)

log

Logarithm.

Syntax

(log number base)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number |
| base | number | 1 | The base. |

Results

| Data Type | Description |
|-----------|---------------|
| number | The logarithm |

Remarks

Returns the logarithm of the number for a base.

Example

```
> (log 100 10)
.: 2

> (log 27 3)
.: 3

> (log 22026.31763368418 (e))
.: 10
```

Related

**** exponentiation**

long

Converts a value to a long number.

Syntax

(long *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 1 | A number value or the literal minimum or maximum |

Results

| Data Type | Description |
|-----------|-----------------|
| long | An long number. |

Remarks

A long is a signed 128 bit number. If the value is **minimum** (or **maximum**) the minimum (or maximum) monadic is returned; otherwise, if the value cannot be converted to a monadic, the function fails. The fractional part of the number is truncated. The suffix **n** denotes a long number.

Example

```
> (long {a b c})  
.: [Failure :Name ArgumentValue :Text "{a b c} is not a number."]  
  
> (long "a b c")  
.: [Failure :Name ArgumentValue :Text "'a b c' is not a number."]  
  
> (long 500)  
.: 500n  
  
> (long 500.0)  
.: 500n  
  
> (long "23.4")  
.: 23n
```

Related

[big, integer](#)

loop

Repeatedly evaluates expressions.

Syntax

(loop *expression* ... *gate* **)**

gate ::= **repeat** *integer*

gate ::= **until** *truth*

gate ::= **while** *truth*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| expression | variant | 0+ | Expressions to be evaluated. |
| gate | expression | 1 | the literal while or until followed by a truth expression, or the literal repeat followed by an integer. |

Results

| Data Type | Description |
|-----------|--|
| variant | Returns the value of the last executed expression before the gate. |

Remarks

Each expression is executed in succession and the value of the last expression before the gate is returned. This function repeats the expression(s) according to the gate. If **until** is specified, the iteration fails when the condition becomes true. If **while** is specified, the iteration fails when the condition becomes false. If **repeat** and an integer value is specified as the gate, then the iteration repeats for the number of times in the integer value. The gate may be omitted to iterate only once.

Example

```
> (loop (--> ++ ?i) while (< ?i 10))
.: 10

> (loop (--> -- ?i) until (< ?i 1))
.: 0

> (loop (--> ++ ?i) repeat 10)
.: 10
```

Related

[wait](#), [abort](#), [await](#), [complete](#), [done-p](#), [reclaim](#), [use](#), [for](#), [iterate](#), [loop](#), [step](#), [task](#), [until](#), [while](#)

lowercase

Converts a string to lower case.

Syntax

(lowercase *string*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| string | string | 1 | A string |

Results

| Data Type | Description |
|-----------|------------------------|
| string | The lower cased string |

Remarks

None

Example

```
> (lowercase "The Quick BROWN fox")
.: "the quick brown fox"
```

Related

capitalize, uppercase

lowercase-p

Returns true if a string's first position is lower case.

Syntax

(**lowercase-p** *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|--|
| truth | Returns true if the first position is lowercase. |

Remarks

None

Example

```
> (lowercase-p "the quick brown fox")
.: true
> (lowercase-p "This and that")
.: false
```

Related

letter-p, lowercase-p, uppercase-p, whitespace-p

macro

Creates a macro.

Syntax

(macro *name* *parameters* *expression* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|-----------------------|
| name | literal | 1 | A macro name |
| parameters | list | 1 | A parameter list |
| expression | variant | 1+ | Forms to be evaluated |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| value | The last value in the list of frames |

Remarks

A macro is a function whose parameters are unevaluated. To evaluate any of the parameters, the **eval** function must explicitly be called on that parameter.

Example

```
> (macro set2 {?value ?var1 ?var2}
    "puts value into two variables in parallel"
    (eval (` (set , ?var1 , ?value , ?var2 , ?value)))))

.: set2

> (let ?x1 nil ?x2 nil) ?x1 ?x2

.: true nil nil

(set2 (+ 2 2) ?x1 ?x2)

> ?x1 ?x2

.: 4 4
```

Related

[evaluate](#), [function](#), [` quote](#), [` expand](#)

macros

Creates a list of macros defined in a module.

Syntax

(macros *module*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--|
| module | literal | 0-1 | A valid module name. Defaults to the current module. |

Results

| Data Type | Description |
|-----------|---|
| list | A list of macros defined in the module. |

Remarks

None.

Example

```
> (macro set2 {?s1 ?s2 ?value}
    (eval (` (set , ?s1 , ?value , ?s2 , ?value))))  
.: set2  
> (macros)  
. : {set2}
```

Related

`expand, 'quote, eval, functions, is, macro

make

Creates a record by creating a relationship if it does not already exist.

Syntax

(**make** *prototype term ...*)

term ::= slot referent

term ::= method function

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|--|
| prototype | prototype | 1 | A relation or structure. |
| term | expression | 0+ | A slot and a referent, or method and function. |

Results

| Data Type | Description |
|------------|-------------------------------|
| identifier | The identifier of the premise |

Remarks

If the prototype is a relation which has an !onNew method defined, the method shall be invoked once during SubThought creation, before the SubThought identifier is returned to the calling function. If the relationship is new, the referents and functions shall serve as defaults for the relationship.

Example

```
> (make Employee :Name "John Smith" :Salary 225000
    !onNew (function Employee_new {?me}
              (tell user ($ Emp - ($$ (:Name ?me),)
                           Pay - (:Salary ?me) \n \n)))))

Emp - John Smith, Pay - 225000
.: Employee_1
```

Related

nix, old, relation, seal, unseal

map

Applies a function to elements across sequences.

Syntax

(**map** *sequence ... function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------------|
| sequence | sequence | 1+ | A sequence of values |
| function | variant | 1 | The name of the function to invoke |

Results

| Data Type | Description |
|-----------|---|
| sequence | The sequence resulting from applying the function to all sequences. |

Remarks

Applies the argument to a list containing values and returns the results.

Example

```
> (map {1 2 3} {4 5 6} {7 8 9} +)
.: {12 15 18}

> (map {1 2 3} even-p)
.: {false true false}

> (map {10 20 30} {3 4 5} *)
.: {30 80 150}
```

Related

apply, collect, count, each, filter, fold, gather, group, merge, reduce , scale , separate, sort

match

Matches rules for all open problems or a specific problem.

Syntax

(**match** *problem*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--|
| problem | problem | 0-1 | A specific problem identifier Default is all if not supplied. |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| integer | Returns the number of rules matched . |

Remarks

If no problem is provided, the (**problems**) function is called and the applicability and relevance of candidate rules in each open problem are determined. Returns the total rules matched.

Example

```
> (domain FixIt
  (relation Car :Age :Condition)
  (rule
    if
      [Car :Age /= old :Condition = good]
    do
      (tell user ($ ($$ \n that) is possible. \n)
      then done)))
  .: FixIt

> (problem state [Car :Age new :Condition good])
  .: Problem_1

> (loop (match Problem_1) while (> (infer Problem_1) 0))
that is possible.

.: true
```

Related

cancel, domain, domains, infer, match, problem, problems, rule, rules

max

Finds the maximum element.

Syntax

(max *value* ... **)**

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------------------------|
| value | variant | 1+ | A sequence, or an atomic value. |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| variant | The largest element in the sequence. |

Remarks

Returns the maximum element in a list or expression. If the list or expression is empty, it returns nil. If the only value is a list then all elements within the list are compared, otherwise, all subsequent values are compared.

Example

```
> (max {1 2 3 4 5})  
.: 5  
  
> (max {a b c d e})  
.: e  
  
> (max {})  
.: nil  
  
> (max 1 2 3 4 5)  
.: 5  
  
> (max {1 2 3} 4 {6 3 2})  
.: {6 4 4}
```

Related

choose, min

maximum

Finds the maximum element.

Syntax

(**maximum** *symbol* ... *binding* *sequence* *expression* ...)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| symbol | symbol | 1+ | A symbol. |
| binding | literal | 0-1 | The literal in or per . |
| sequence | sequence | 0-1 | A sequence. |
| expression | expression | 1+ | any expression involving variables from the pattern(s) |

Results

| Data Type | Description |
|-----------|--|
| variant | The result of applying the max function to the expression for the indicated range. |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to subelements of each subsequence. If there are insufficient elements to bind to the variables then the function fails. The last expression shall be evaluated and the maximum function shall be applied to both that expression and any prior result. The final value is returned.

Example

```
> (maximum ?x in (range 1 to 10) ?x)
.: 10

> (maximum ?x in {1 2 3 4} (** ?x 2))
.: 16

> (maximum ?x ?y per {{a 1}{b 2}{c 3}} ?x)
.: c
```

Related

fold, minimum, product, summation

may

Evaluates an expression while suppressing failures.

Syntax

(**may** *expression*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|----------------------------|
| expression | variant | 1 | An expression to evaluate. |

Results

| Data Type | Description |
|-----------|--|
| list | Returns a list containing a truth success value, the result, and possible failure. |

Remarks

If the expression fails, the result shall be nil.

Example

```
> (may (string x))
.: {true "x" nil}
> (may (integer x))
.: {false nil [Failure :Name TypeConversion :Text "Cannot convert x to Integer"]}
> (let {?success ?result ?error} (may ($ hello world)))
.: true
> ?success ?result ?error
.: true "hello world" nil
```

Related

assert, can, did, ensure, fail, finally, try

median

Finds the middle value of a sequence.

Syntax

(**median** *value* ...)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|------------------------------|
| value | variant | 1+ | A number or list of numbers. |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| number | The median value of all elements. |

Remarks

Returns the median. If the list or expression is empty, it returns nil. If the only value is a list then all elements within the list are compared, otherwise, all subsequent values are compared.

Example

```
> (median {1 2 3 4 5})  
. : 3  
  
> (median {})  
. : nil  
  
> (median 1 2 3 4 5)  
. : 3  
  
> (median {1 2 3} 4 {6 3 2})  
. : {4 3 3}
```

Related

[avg](#), [max](#), [median](#), [min](#), [statistics](#), [sigma](#), [sum](#)

meets-p

True if an interval finishes when a second interval starts.

Syntax

(meets-p *interval₁* *interval₂* *tolerance*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------------------|-----------|-----|--|
| interval ₁ | interval | 1 | An interval |
| interval ₂ | interval | 1 | An interval |
| tolerance | instant | 0-1 | Amount of variation. (Defaults to zero ticks). |

Results

| Data Type | Description |
|-----------|---|
| truth | True if interval ₁ finishes when interval ₂ starts. |

Remarks

None.

Example

```
> (meets-p  \@i{\@s1 \@s2}  \@i{\@s3 \@s4})  
.: false  
  
> (meets-p  \@i{\@s1 \@s5}  \@i{\@s5 \@s8})  
.: true  
  
> (meets-p  \@i{\@s1 \@s9}  \@i{\@s4 \@s6})  
.: false  
  
> (meets-p  \@i{\@s4 \@s6}  \@i{\@s6 \@s9})  
.: true  
  
> (meets-p  \@i{\@s5 \@s6}  \@i{\@s8 \@s9} \@s2)  
.: true
```

Related

before-p, during-p, finishes-p, overlaps-p, starts-p

meron

Returns the meronomic (i.e., proto) type of a value.

Syntax

(**meron** *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-----------------------------------|
| value | variant | 1 | A structure, relation, or premise |

Results

| Data Type | Description |
|-----------|---|
| literal | The meronomic prototype of the relation |

Remarks

None

Example

```
> (meron (relation A))
.: A

> (meron (new (structure B :Slot1 :Slot2)))
.: B

> (meron 100)
.: nil

> (meron foo)
.: nil

> (meron (new (relation C uses {A B})))
> C

> (meron [D :Foo 1 :Bar 2])
> D
```

Related

id, is, meronomy,, taxon, taxonomy

meron-p

True if the value is a meron or is of a specific meronomic type.

Syntax

(**meron-p** *value* *meron*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------------|
| value | variant | 1 | A value. |
| meron | literal | 0-1 | The name of a meronomic type. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the value is a meron or is of a specific meronomic type.. |

Remarks

If the meron parameter is omitted, the function returns true if the value is a meron. Otherwise, the function returns true if the value is of the specified meronomic type. .

Example

```
> (meron-p 500)
.: false

> (relation A :S1 :S2 0) (relation B uses {A} :S3 100)
.: A B

> (meron-p (new A))
.: true

> (meron-p (new B) A)
.: true
```

Related

meron, meronomy, taxon, taxon-p, taxonomy

meronomy

Returns a list of meronomic (i.e., proto) types for a value.

Syntax

(**meronomy** *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A value |

Results

| Data Type | Description |
|-----------|--------------------------|
| list | The meronomic types used |

Remarks

The value must be a premise, prototype, or record , otherwise an empty list is returned

Example

```
> (meronomy 100)
.: {}

> (meronomy donut)
.: {}

> (relation C uses {(relation A) (relation B)})
.: C

> (meronomy (new C))
.: {C A B}

> (meronomy A)
.: {A}
```

Related

taxon,taxonomy, id, is, meron, relation, structure

method

Creates a function within a prototype.

Syntax

(method scope name parameters expression ...)

parameters ::=
parameters ::= {}
parameters ::= {required ... }
parameters ::= {required ... + optional ... }
parameters ::= { + optional ... }
parameters ::= {required ... + optional ... & remaining }
parameters ::= { & remaining }

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-------------|-----|---|
| scope | environment | 0-1 | An environment. If not supplied, a new one is made. |
| name | literal | 0-1 | The name of the function. If not supplied then the name shall follow the pattern fn-<number> . |
| parameters | list | 0-1 | A list containing variables, keywords, sigils, and keyword symbol associations. |
| expression | variant | 1+ | One or more expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|--------------------------|
| literal | The name of the function |

Remarks

Methods are defined using the **method** intrinsic function. If the name is supplied then the method is a user named function (also called an **exonym**), otherwise it is an automatically named function (called an **esonym**). If a parameter list is supplied then it shall bind the formal parameter variables to any actual parameter expressions provided. The first variables in the parameter list are required. If a plus sign is provided, then those variables following the plus sign are optional. Each optional parameter may have a default value (specified with an equal sign and the value). If no default value is specified, the default value shall be **nil**. If an ampersand is provided, then the symbol following it is bound to a list of remaining (variadic) actual parameter values. Keyword parameters (literal symbol pairs) may be provided as required or optional. The entire parameter list may be omitted. The function shall be accessible from other modules by prefixing it with the module name in which it was defined. A hidden variant parameter **?me** is supplied as the first parameter to the method function and is available within the method for reading. It may not be upadted or changed in the scope of the method.

Example

```
> (relation Context_Items
  :Lexicon
  :Item )

.: Context_Item

> (relation MyLexicon
  :Name
  :Context {}
  !refreshContext
  (method {}
    (let ?me :Name ?lexicon :Context ?context)
    (with Context_Item :Lexicon = ?lexicon :Item as ?Item
      (out ?item ?context)
      do (zap Context_Item :Lexicon ?lexicon :Item ?item)
      (for ?item in ?context
        (knew Context_Item :Lexicon ?lexicon :Item ?item)))))

.: MyLexicon

> (structure IMonsterTask
  :Name
  !setInfo
  !setRoomTask
  :RoomTask
  !kill )

.: IMonstertask

> (structure MonsterTask
  uses {IMonsterTask}

  :MonsterInfo (new MonsterInfo)

  !onActivate (method {truth ?cancelled}
    (put ?me :Id (!getPrimaryKey ?me))
    (registerTimer move nil (interval \@s150
\@s9000)))
  )

  !setInfo (method {MonsterInfo ?info}
    (put ?me :MonsterInfo ?info)
    (return done))

  !name (method {} (get ?me :MonsterInfo :Name ))

  !setRoomTask (method {RoomTask ?room}
    (if (exists-p (:RoomTask ?me))
        (await (!exit (:RoomTask ?me))))
    (put ?me :RoomTask ?room)
    (!enter (:RoomTask (:MonsterInfo ?me)))))

  !move (method {}
    (if (exist-p (:RoomTask ?me))
        (let ?directions {north south west east}
          ?random (random 0 to 4)
          ?nextRoom (await (!exitTo (:RoomTask ?me)
            (@ ?directions ?random))))
        (if (null-p ?nextRoom) (return nil))
        (await (!exit (:RoomTask ?me) (:MonsterInfo ?me)))
        (await (!enter ?nextRoom (:MonsterInfo ?me))))
```

```
(put ?me :RoomTask ?nextRoom))

!kill (method {RoomTask ?room}
      (if (exists-p (:RoomTask ?me))
          (return (on (/= (get ?me :RoomTask :PrimaryKey)
                           (get ?room :PrimaryKey))
                     (tell User ($ (!name (:MonsterInfo ?me)) snuck away.
                                  You were too slow!)))
                     (do (!exit (:RoomTask ?me))
                         (tell User ($ (!name (:MonsterInfo ?me)) is dead.))))
          (return (tell User ($ (!name (:MonsterInfo ?me)) is already dead. \n
                                  You were too slow and someone else got to
                                  him.)))))

.. : MonsterTask
```

Related

[arguments](#), [call](#), [extend](#), [functions](#), [junction](#), [macro](#), [module](#), [modules](#), [parameters](#)

mid

Copies a subsequence of a sequence.

Syntax

(**mid** *sequence start stop skip*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| sequence | sequence | 1 | sequence |
| start | number | 1 | Starting position |
| stop | expression | 1 | One of to <i>finish</i> or go <i>count</i> |
| skip | expression | 0-1 | Key value pairs by <i>count</i> – skips a number of elements The expression by <i>count</i> default is 1. |

Results

| Data Type | Description |
|-----------|------------------|
| sequence | The sub-sequence |

Remarks

The *end* value must be greater than the *start* value.

Example

```
> (mid {a b c d} 2 to 3)
.: {b c}

> (mid "abcdefg" 4 go 4)
.: "defg"

> (mid "abcdefg" 2 to 4)
.: "bcd"

> (mid "abcdefg" 2 go 3 by 2)
.: "bd"
```

Related

[but](#), [last](#), [rest](#), [top](#)

min

Finds the minimum element.

Syntax

(min *value* ... **)**

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------------------|
| value | variant | 1+ | An atom or list of atoms. |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| variant | The smallest element in the sequence. |

Remarks

Returns the minimum element in a list or expression. If the list or expression is empty, it returns nil. If the only value is a list then all elements within the list are compared, otherwise, all subsequent values are compared.

Example

```
> (min {1 2 3 4 5})  
.: 1  
  
> (min {a b c d e})  
.: a  
  
> (min {})  
.: nil  
  
> (min 1 2 3 4 5)  
.: 1  
  
> (min {1 2 3} 4 {6 3 2})  
.: {1 2 2}
```

Related

[choose](#), [max](#)

minimum

Finds the minimum element.

Syntax

(**minimum** *symbol ... binding sequence expression ...*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| symbol | symbol | 1+ | A symbol. |
| binding | literal | 0-1 | The literal in or per . |
| sequence | sequence | 0-1 | A sequence. |
| expression | expression | 1+ | any expression involving variables from the pattern(s) |

Results

| Data Type | Description |
|-----------|---|
| variant | The result of applying the plus function to the expression for the indicated range. |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to subelements of each subsequence. If there are insufficient elements to bind to the variables then the function fails. The last expression shall be evaluated and the minimum function shall be applied to both that expression and any prior result. The final value is returned.

Example

```
> (minimum ?x in (range 1 to 10) ?x)
.: 1

> (minimum ?x in {1 2 3 4} (** ?x 2))
.: 1

> (maximum ?x ?y per {{a 1}{b 2}{c 3}} ?x)
.: a
```

Related

[fold](#), [maximum](#), [minimum](#), [product](#)

missing

True if a value is absent from an assortment.

Syntax

(missing *assortment key*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------|
| assortment | assortment | 1 | An assortment. |
| value | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if the value is absent or false if the value is present. |

Remarks

None

Example

```
> (lexicon :A 1 :B 2 !m X_foo :C 3)
.: Lexicon-1

> (missing Lexicon-1 3)
.: false

> (missing Lexicon-1 X_foo)
.: false

> (missing Lexicon-1 elephant)
.: true
```

Related

beyond, has, within

mod

Finds the remainder after division.

Syntax

(mod *dividend modulus*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------|
| dividend | variant | 1 | A number or list of numbers. |
| modulus | variant | 1 | A number or list of numbers. |

Results

| Data Type | Description |
|-----------|---|
| variant | the remainder after dividing dividend by divisor as a number or list of remainders. |

Remarks

Returns the remainder of the dividend divided by the modulus. Both values must be numbers or lists. If each argument is a list, they must be the same length.

Example

```
> (mod 28 3)
.: 1

> (mod {6 8 9 2} 2)
.: {0 0 1 0}

> (mod 17 {5 6 3 4})
.: {2 5 2 1}

> (mod {12 45 8 17} {5 5 3 7})
.: {2 0 2 3}
```

Related

/ division, * multiplication

modular

Evaluates expressions within a module's scope.

Syntax

(modular *module expression ...* **)**

assignment ::= symbol value expression

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---------------|
| module | Literal | 1 | A module name |
| expression | variant | 1+ | An expression |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | Returns the value true . |

Remarks

Creates module level variables.

Example

```
> (modular User (let ?START_OF_TASK 1))

.: true

> (modular Business
    (let ?FIRST_NAME Jane
        ?LAST_NAME Smith))

.: true
```

Related

<-- before tie, --> after tie, constant, dynamic, local, put, suppose, swap, undefine, use

module

Creates a module.

Syntax

(module *name definition ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| name | literal | 1 | A module name |
| definition | expression | 0+ | A function, macro, or relation definition call |

Results

| Data Type | Description |
|-----------|-------------------------|
| literal | The name of the module. |

Remarks

Shall create a new module or redefine an existing module. To add definitions to an existing module use the **extend** function. To prohibit new definitions in a module use the **wrap** function. Each module is global, this means that nested module definitions do not concatenate.

Example

```
> (module Algebra
    (function sum ?args (apply + ?args)))

.: Algebra

> (module Outer
    (module Inner
        (function Which {} One) ; the function shall be Inner.Which
    )
)
.: Outer

> (Inner.Which)

.: One
```

Related

dependencies, discard, extend, forgo, modules, require, grok, unwrap, use, wrap

modules

Creates a list of known modules.

Syntax

(modules)

Module

Base

Parameters

None.

Results

| Data Type | Description |
|-----------|-----------------------|
| list | A list of all modules |

Remarks

Returns the list of known modules.

Example

```
> (modules)
.: {Apex Base IO KB Math Tasks User}
```

Related

dependencies, discard, extend, forgo, module, require, grok, unwrap, use, wrap

moment

Creates a moment or returns the current moment.

Syntax

(moment *units secs mins hours days year* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| units | number | 0-1 | Units. Up to three digits for milliseconds. |
| secs | number | 0-1 | Two digits: e.g., (seconds / 60) * 100 |
| min | number | 0-1 | Two digits: e.g., (minutes / 60) * 100 |
| hour | number | 0-1 | Two digits: e.g., (hour / 24) * 100 |
| days | number | 0-1 | Three digits e.g. (day of year / 365.25) x 1000 |
| year | variant | 0-1 | Four digits for year (e.g. 2015, 2017, 2018, 2025) |

Results

| Data Type | Description |
|-----------|----------------------------------|
| moment | The current moment in nano years |

Remarks

If no arguments are supplied, moment returns the current time as a moment, otherwise it attempts to construct a moment from the supplied arguments.

Example

```
> (moment)
.: \@m2018270038501744

> (moment 3000)
.: \@m3000

> (moment 0 0 0 0 0 2020)
.: \@m2020000000000000

> (moment 0 0 0 (time hour part 18) (time days part 182) 2027)
.: \@m2027499030000000
```

Related

date, epoch, tick

more-p

True if a container contains more than zero elements.

Syntax

(more-p *sequence*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the container contains more than zero elements. |

Remarks

None.

Example

```
> (more-p "not empty")
.: true

> (more-p (call pi))
.: true

> (more-p "")
.: false

> (more-p {})
.: false

> (more-p [A])
.: true

> (more-p (expression foo))
.: true
```

Related

[void-p](#)

morph

Applies functions in succession using the arguments.

Syntax

(**morph** *arguments functions*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--|
| arguments | list | 1 | The intial arguments to the last function in the functions list. |
| functions | list | 1 | An ordered list of functions to be applied. |

Results

| Data Type | Description |
|-----------|---|
| value | The result of the application of the last function. |

Remarks

If the arguments parameter is an empty list, the leftmost function is assumed to be niladic (requiring no arguments). The leftmost function is applied to the arguments parameter list, then the second from leftmost is applied to the result, and so on, until all functions are applied. The final result is returned. The opposite function is **compose**. The **scatter** function applies functions in parallel.

Example

```
> (morph {0} {++ ++ ++ (given {?n} (* ?n 2)))}
.: 6

> (morph {{the quick brown fox jumped over the lazy dogs}}
           {but rest but rest})
.: {brown fox jumped over the}

> (morph {2 3 4} {+ ++ ++ (given {?n} (* ?n 2)))}
.: 22
```

Related

apply, compose, map, scatter

most

True if the predicate is true for more than half of the elements in a sequence.

Syntax

(most *symbol* ... *binding sequence expression* ... *test*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , or over . |
| sequence | sequence | 1 | A list or string |
| expression | expression | 0+ | An expression. |
| test | truth | 1 | A truth expression |

Results

| Data Type | Description |
|-----------|---|
| truth | True if more than half the elements satisfies the predicate |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If there are insufficient elements to bind to the variables then the function fails. The expressions and test are evaluated.

Example

```
> (most ?n in {1 2 3 4} (= (taxon ?n) number))
.: true

> (most ?m ?n per {{2 1} {3 4} {7 9}} (< ?m ?n))
.: true

> (most ?m ?n over {4 3 2 7} (divisible-p ?m ?n))
.: false
```

Related

every, few, nevery, none, some

move

Moves or renames one or more files.

Syntax

(move *source destination*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-------------|-----------|-----|---------------------|
| source | url | 1 | The source url |
| destination | url | 1 | The destination url |

Results

| Data Type | Description |
|-----------|---|
| list | A possibly empty list of file path strings for each file moved. |

Remarks

If the path is invalid, the function fails.

Example

```
> (move (url path "foo.txt") (url path "bar.txt"))
.: {"bar.txt"}
```

Related

[close](#), [mkdir](#), [file](#), [path](#), [write](#)

my

Returns the current cell resource.

Syntax

(my)

Module

Base

Parameters

None.

Results

| Data Type | Description |
|-----------|---|
| cell | The cell resource for the performing functions. |

Remarks

None.

Example

```
> (my)
.: Cell-0
> (get (my) :Self)
.: Task-0
```

Related

premise, self

nall

True if not all relevant knowledge satisfies all the premises.

Syntax

```
(nall premise ... )  
  
premises ::= premise  
premises ::= premise premises  
premises ::= premise premises  
premises ::= premise restriction ... premises  
restriction ::= (predicate value ...)  
premise ::= [category descriptor ... ]  
category ::= type  
category ::= list  
descriptor ::= slot binding  
descriptor ::= slot condition  
binding ::= as symbol  
binding ::= each symbol  
condition ::= slot = value  
condition ::= slot is unary-predicate  
condition ::= slot not value  
condition ::= slot binary-predicate value  
condition ::= slot n-ary-predicate value-list  
condition ::= (predicate value ...)
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|----------------------------------|
| premise | variant | 1+ | A pattern, call, or truth value. |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| truth | True if not all values return true. |

Remarks

True if any value is nil. Terminates on the first nil value. If the value is an iterator, then a cursor is created over which the remaining values are tested until truth or falsity is determined.

Example

```
> (nall grapefruit nil apple)  
. : true  
  
> (nall nil nil)  
. : true  
  
> (nall nil)  
. : true  
  
> (nall foo)  
. : false
```

Related

[all](#), [any](#), [no](#)

nand

Negated logical conjunction.

Syntax

(nand *truth* *truth* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------|
| value | truth | 2+ | A truth value |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| truth | True if any argument value is false |

Remarks

Equivalent to (not (and truth ...)).

Example

```
> (nand true true true false)
.: true
> (nand true true true true)
.: false
```

Related

and, is, nor, not, or, xor

negative-p

True if a number is negative.

Syntax

(negative-p *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| truth | True if the number is less than zero. |

Remarks

None.

Example

```
> (negative-p 500)
.: false
> (negative-p 0)
.: false
> (negative-p -25)
.: true
```

Related

[even-p](#), [odd-p](#), [positive-p](#), [zero-p](#)

nevery

True if the predicate is false for any element in the sequence.

Syntax

(nevery symbol ... binding sequence expression ... test)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , or over . |
| sequence | sequence | 1 | A list or string |
| expression | expression | 0+ | An expression. |
| test | truth | 1 | A truth expression |

Results

| Data Type | Description |
|-----------|--|
| truth | True if any element does not satisfy the predicate |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If there are insufficient elements to bind to the variables then the function fails. The expressions and test are evaluated. If the sequence is empty the function fails.

Example

```
> (nevery ?n in {1 a 3 4} (= (taxon ?n) number))
.: true

> (nevery ?m ?n per {{1 2} {3 4}} (> ?m ?n))
.: true

> (nevery ?m ?n over {4 2 1} (divisible-p ?m ?n))
.: false
```

Related

every, few, most, none, some

new

Creates a record.

Syntax

(**new** *prototype term ...*)

term ::= slot referent

term ::= method function

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|--|
| prototype | prototype | 1 | A relation or structure. |
| term | expression | 0+ | A slot and a referent, or method and function. |

Results

| Data Type | Description |
|------------|-------------------------------|
| identifier | The identifier of the premise |

Remarks

If the prototype is a relation which has an !onNew method defined, the method shall be invoked once during SubThought creation, before the SubThought identifier is returned to the calling function.

Example

```
> (relation Employee :Name :Salary
    !onNew  (function Employee_new {?me}
              (tell user ($ Emp - ($$ (:Name ?me) ,
                                         Pay - (:Salary ?me) \n \n)))))

.: Employee

> (new Employee :Name "John Smith" :Salary 225000)

Emp - John Smith, Pay - 225000

.: Employee_1
```

Related

nix, old, relation, seal, unseal

next

Advances a series to the next element.

Syntax

(**next** *series*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| series | series | 1 | A series. |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if the next item in the iteration can be accessed. |

Remarks

None.

Example

```
> (function walk {?iterable}
  (let ?series (series ?iterable))
  (while (next ?series)
    (tell user ($ Found ($$ (this ?series) .) \n)))
  walked)

.: walk

> (walk {a b c})
Found a.
Found b.
Found c.
.: walked

> (walk (lexicon a 1 b 2 c 3))
Found {a 1}.
Found {b 2}.
Found {c 3}.
.: walked
```

Related

[reset](#), [series](#), [this](#)

ngrams

Creates a list of substrings.

Syntax

(ngrams string length)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|---|
| string | string | 1 | A target string |
| length | integer | 0-1 | The length of substrings to find. (Default is 3.) |

Results

| Data Type | Description |
|-----------|---|
| list | A list of strings of the specified length |

Remarks

Affixes spaces before and after the string, then divides the string into substrings of the specified length.

Example

```
> (ngrams "fox" 3)
.: {" f" " fo" "fox" "ox" "x" }
> (ngrams "fox")
.: {" f" " fo" "fox" "ox" "x" }
> (ngrams "fox" 5)
.: {" f" " fo" " fox" " fox" "fox" "ox" "x" }
```

Related

[lexemes](#), [split](#), [trim](#)

nil-p

True if a value is nil.

Syntax

(**nil-p** *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1+ | A value |

Results

| Data Type | Description |
|-----------|----------------------------|
| truth | True if the values is nil. |

Remarks

None.

Example

```
> (nil-p yes)
.: false
> (nil-p \%nil)
.: true
> (nil-p (nothing))
.: false
> (nil-p \%void)
.: false
> (nil-p nil)
.: true
```

Related

null-p, thing-p, void-p

nix

Deletes a prototype.

Syntax

(**nix** *prototype ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|------------------------------------|
| prototype | prototype | 1+ | A prototype or list of prototypes. |

Results

| Data Type | Description |
|------------|--|
| expression | Returns the literal nixed and the quantity deleted. |

Remarks

Undeclares a prototype. All thoughts of the relation must be reclaimed with the **old** function first, otherwise the function fails.

Example

```
> (new (relation Q))

.: Q_1

> (nix Q)

.: [Failure :Name InvalidOperation :Text "Cannot nix relations having thoughts."]

> (do (with Q ^ as ?q do (old ?q)) (nix Q))

.: nixed 1

> (is Q (given {?r} (more-p (meronomy ?r)))

.: false

> (new (relation P))

.: P_1

> (do (old (with P)) (nix P))

.: nixed 1
```

Related

structure, structures, old, relation, relations, seal, unseal

no

True if no relevant knowledge satisfies all of the premises.

Syntax

```
(no premise ...)

premises ::= premise
premises ::= premise premises
premises ::= premise premises
premises ::= premise restriction ... premises
restriction ::= (predicate value ...)
premise ::= [category descriptor ... ]
category ::= type
category ::= list
descriptor ::= slot binding
descriptor ::= slot condition
binding ::= as symbol
binding ::= each symbol
condition ::= slot = value
condition ::= slot is unary-predicate
condition ::= slot not value
condition ::= slot binary-predicate value
condition ::= slot n-ary-predicate value-list
condition ::= (predicate value ...)
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|----------------------------------|
| value | variant | 1+ | A pattern, call, or truth value. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | True if all the values are nil. |

Remarks

Equivalent to (not (any premise ...)). If the value is a pattern or a relationship, then a cursor is created over which the remaining values are tested until truth or falsity is determined. If the value is an iterator, then a cursor is created over which the remaining values are tested until truth or falsity is determined.

Example

```
> (no one two three)
.: false

> (no a b nil)
.: false

> (no nil nil nil)
.: true
```

Related

[all](#), [any](#), [default](#), [nall](#)

none

True if the predicate is false for all sequence elements.

Syntax

(none symbol ... binding sequence expression ... test)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , or over . |
| sequence | sequence | 1 | A list or string |
| expression | variant | 0+ | An expression. |
| test | truth | 1 | A truth expression |

Results

| Data Type | Description |
|-----------|--|
| truth | True if every element does not satisfy the predicate |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If there are insufficient elements to bind to the variables then the function fails. The expressions and test are evaluated. If the sequence is empty the function fails.

Example

```
> (none ?n in {1 2 3 4} (alphabetic-p ?n))  
.: true  
  
> (none ?m ?n per {{1 2} {3 4}} (< ?m ?n))  
.: true  
  
> (none ?m ?n over {4 2 1} (divisible-p ?m ?n))  
.: false
```

Related

every, few, most, nevery, some

nor

Negated logical disjunction.

Syntax

(nor *truth* *truth* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------|
| truth | truth | 1+ | A truth value |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| truth | True if all argument values are false |

Remarks

Equivalent to (not (or value value ...)).

Example

```
> (nor true true true false)
.: false
> (nor true true true true)
.: false
> (nor false false false false)
.: true
```

Related

and, nand, not, or, xnor, xor

not

True if the value is false, or if the applied unary predicate returns false.

Syntax

(not *value* *predicate*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|-------------------|
| value | variant | 1 | A value |
| predicate | literal | 0-1 | A unary predicate |

Results

| Data Type | Description |
|-----------|--|
| truth | True if value is false, or if the applied unary predicate returns false. |

Remarks

None.

Example

```
> (not false)
.: true

> (not true)
.: false

> (not 3 even-p)
.: true

> (not 7 (given {?n} (divisible-p ?n 3)))
.: true

> (not foo null-p)
.: true

> (not nil null-p)
.: false
```

Related

convert, datatype, is, known, the, type

nothing

Returns nothing.

Syntax

(nothing)

Module

Base

Parameters

None.

Results

| Data Type | Description |
|-----------|------------------|
| nothing | A nothing value. |

Remarks

None.

Example

```
> (nothing)
.:
> {a (nothing) b (nothing) c}
.: {a b c}
```

Related

ceiling, complex, floor, is, long, radix, rational, real, round, truncate, unsigned

novelty

Creates a problem state of type novelty.

Syntax

(novelty *option ...*)

Module

KB

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| option | expression | 0-1 | Any combination of facts premise ... |

Results

| Data Type | Description |
|-----------|---|
| literal | Returns a thought identifier for the gap state containing the premises. |

Remarks

A novelty represents a situation having unknown consequences and implications to be explored. .
A **Novelty** thought is created having slots that correspond to each key (i.e.,:Facts).

Example

```
> (problem
    domain Robotics
    state (novelty
        facts
            [Is :Type Room :Item RoomA] [Is :Type Room :Item RoomB]
            [Is :Type Ball :Item Ball1] [Is :Type Ball :Item Ball2]
            [Is :Type Ball :Item Ball3] [Is :Type Ball :Item Ball4]
            [Is :Type Arm :Item Left] [Is :Type Arm :Item Right]
    )
    .: Problem_3

> (get Problem_3 :State :Facts)
.: {Is_1 Is_2 Is_3 Is_4 Is_5 Is_6 Is_7 Is_8}

> (get Problem_3 :State)
.: Novelty_1
```

Related

domain, domains, gap, lacuna, obstacle, problem, problems,

null-p

True if a value is nothing or nil.

Syntax

(null-p *value* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1+ | A value |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| truth | True if the values is nothing or nil. |

Remarks

None.

Example

```
> (null-p yes)
.: false
> (null-p \%nil)
.: true
> (null-p (nothing))
.: true
> (null-p \%void)
.: true
```

Related

nil-p, thing-p, void-p

odd-p

True if a number is odd.

Syntax

(odd-p *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number |

Results

| Data Type | Description |
|-----------|----------------------------|
| truth | True if the number is odd. |

Remarks

Returns true if the number is not evenly divisible by two.

Example

```
> (odd-p 500)
.: false
> (odd-p 1)
.: true
> (odd-p -33)
.: true
```

Related

even-p, negative-p, positive-p, zero-p

old

Deletes a thought.

Syntax

(**old** *thought*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--|
| thought | variant | 1 | A premise, thought, relation, or list of thoughts. |

Results

| Data Type | Description |
|------------|---|
| expression | Returns the literal reaped and the number of thoughts deleted. |

Remarks

Returns the number of thought identifiers deleted. If the relation has an **!onOld** method defined, then the method is invoked prior to the thought being deleted.

Example

```
> (relation Q)
.: Q

> (new Q)
.: Q_1

> (new Q)
.: Q_2

> (old Q_1)
.: 1

> (old Q)
.: 1

> (nix Q)
.: nixed
```

Related

[structure](#), [new](#), [nix](#), [relation](#)

omit

Creates a new sequence with values omitted.

Syntax

(omit *sequence* *value* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence. |
| value | variant | 0+ | A value. |

Results

| Data Type | Description |
|------------|-------------------------|
| assortment | The modified assortment |

Remarks

A new sequence is constructed. To modify the sequence use the **rip** function.

Example

```
> (omit {a b c d e f g} a b)
.: {c d e f g}

> (omit {a 1 nil b 2 c 3 nil nil d  nil 4} nil)
.: {a 1 b 2 c 3 d 4}
```

Related

[Insert](#), [pop](#), [push](#)

on

Branched conditional evaluation.

Syntax

(on condition true-case false-case)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---------------------------|
| condition | truth | 1 | A truth expression |
| true-case | variant | 1 | An expression to evaluate |
| false-case | variant | 1 | An expression to evaluate |

Results

| Data Type | Description |
|-----------|--|
| variant | The last value of the executed expression. |

Remarks

Evaluates the condition and either the true case or the false case.

Example

```
> (on (> 101 100)  (tell user "yes")  (tell user "no"))
yes .: told

> (on (< 101 100)  (tell user "yes")  (tell user "no"))
no .: told
```

Related

and, case, if, not, or, unless

only

Creates a restricted scope.

Syntax

(only *variables expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--|
| variables | list | 1 | A list of variables from the enclosing scope |
| expression | variant | 1+ | An expression to be evaluated. |

Results

| Data Type | Description |
|-----------|-------------------------|
| value | Returns the last value. |

Remarks

Creates a new environment shadowing the designated variables in the encompassing environment and executes each of the expressions, one at a time, returning the result of the last expression. No free variables (i.e., variables not defined inside the scope or not in the symbol list) are permitted within the scope. **only** is equivalent to an in-line anonymous function. Changes to the variables shall not affect the enclosing scope.

Example

```
> (function sample {?b ?c}
  (let ?a 5)
  (tell user ($ given a+b= (+ ?a ?b) \n))
  (tell user ($ given a+b+c= (+ ?a ?b ?c) \n))
  (only {?a ?b}
    (tell user ($ only a+b= (+ ?a ?b) \n))
    (try (tell user ($ only a+b+c= (+ ?a ?b ?c) \n))
      learn ?f (tell user ($ "?c" is not in scope \n))))))
  .: sample

> (sample 15 30)

given a+b= 20
given a+b+c= 50
only a+b= 20
?c is not in scope
.: told
```

Related

given, dynamic, local

open

Opens a file or data url.

Syntax

(**open** *url*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|---------------------|
| url | url | 1 | A file or data url. |

Results

| Data Type | Description |
|-----------|-----------------|
| url | The opened url. |

Remarks

Opens an file.

Example

```
> (var ?file (file name "kwikfox.txt"))

.: File-1

> (open ?file)

.: File-1

> (let ?contents (read ?file #))

.: true

> ?contents

.: "The quick brown fox jumped over the lazy dogs."

> (close ?file)

.: closed
```

Related

[close](#), [data](#), [file](#)

or

Logical disjunction.

Syntax

(**or** *truth* *truth* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------|
| truth | truth | 1+ | A truth value |

Results

| Data Type | Description |
|-----------|------------------------------------|
| truth | True if any argument value is true |

Remarks

Short circuits evaluation upon reaching the first true argument.

Example

```
(or false true false false)
.: true

(or true true true true)
: true

(or false false false false)
.: false
```

Related

and, is, nand, nor, not, xnor, xor

out

True if a value is absent from a sequence.

Syntax

(out *value sequence option ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| value | variant | 1 | A value |
| sequence | sequence | 1 | A sequence. |
| option | expression | 0-2 | A key value pair. Any of depth <i>number</i> - Number or # (for any depth). Default is 1. transform <i>function</i> - A function to transform the element.. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the value is not in the sequence. |

Remarks

If value and sequence are strings, then a string search is performed.

Example

```
> (out x {a b c d e f g})  
.: true  
  
> (out "a" "cbazyx")  
.: false
```

Related

includes, excludes, in, occurrences, pick, position

overlaps-p

True if an interval finishes after a second interval starts.

Syntax

(overlaps-p *interval₁* *interval₂* *tolerance*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------------------|-----------|-----|--|
| interval ₁ | interval | 1 | An interval |
| interval ₂ | interval | 1 | An interval |
| tolerance | instant | 0-1 | Amount of variation. (Defaults to zero ticks). |

Results

| Data Type | Description |
|-----------|--|
| truth | True if interval ₁ finishes after interval ₂ starts. |

Remarks

None.

Example

```
> (overlaps-p  \@i{\@s1 \@s2}  \@i{\@s3 \@s4})  
.: false  
  
> (overlaps-p  \@i{\@s1 \@s5}  \@i{\@s4 \@s8})  
.: true  
  
> (overlaps-p  \@i{\@s1 \@s9}  \@i{\@s4 \@s6})  
.: false  
  
> (overlaps-p  \@i{\@s4 \@s6}  \@i{\@s5 \@s9})  
.: true  
  
> (overlaps-p  \@i{\@s3 \@s4}  \@i{\@s5 \@s9 \@s2})  
.: true
```

Related

before-p, during-p, finishes-p, meets-p, starts-p

pad

Creates a new padded sequence.

Syntax

(pad sequence length value side)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| sequence | sequence | 1 | A string or list |
| length | number | 1 | The length of the final sequence |
| value | variant | 1 | A value to insert into the sequence |
| side | literal | 0-1 | the literal left or right (default). |

Results

| Data Type | Description |
|-----------|------------------------|
| sequence | Creates a new sequence |

Remarks

If the sequence is a string, the value must also be a one position string. If the sequence is a list, the value may be any value. To modify a string destructively, use **resize**.

Example

```
> (pad {1 2 3} 10 0)
.: {1 2 3 0 0 0 0 0 0 0}

> (pad "123" 10 "0")
.: "1230000000"

> (pad "123" 10 "0" left)
.: "0000000123"

> (pad {1 2 3} 10 0 left)
.: {0 0 0 0 0 0 1 2 3}
```

Related

[fill](#) , [resize](#)

partition

Creates a list of equivalence sets for a pivot element.

Syntax

(**partition** *sequence pivot comparer*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | sequence | 1 | A sequence. |
| pivot | variant | 1 | An element in the list. |
| comparer | function | 0-1 | A comparison function.(Defaults to compare .) |

Results

| Data Type | Description |
|-----------|--|
| list | A list of sublists each containing elements partitioned by a predicate.. |

Remarks

If no comparer is provided, the function **compare** is used. The element argument is compared to each item in the sequence and three lists are returned, elements that are less, elements that are the same, and elements that are greater. If a comparer function is provided, it must return the literals <, =, > for elements that are less than, equal to, or greater than the pivot element.

Example

```
> (partition (3 5 2 0 5 8 9) 5)
.: {{3 2 0}{5 5}{8 9}}
> (partition "abcdefg" "e")
.: {{"a" "b" "c" "d"} {"e"} {"f" "g"}}
> ((function qksort {?list}
  (if (void-p ?list) {}
    else (let {?less ?same ?more} (partition ?list (pick ?list)))
      (& (qksort ?less) ?same (qksort ?more))))
  (3 5 2 7 5 6 0 8 9))
.: {0 2 3 5 5 6 7 8 9}
```

Related

categorize

path

Concatenates a folder and file name.

Syntax

(**path** *component* ...)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|------------------------------------|
| component | string | 1 | A folder, separator, or file name. |

Results

| Data Type | Description |
|-----------|---|
| string | Returns the concatenated folder and file as a string. |

Remarks

Returns a properly formatted path combining the folder and file name. The folder may or may not contain a terminating directory separator. The file may or may not contain a leading directory separator.

Example

```
> (path "C:/projects/project1/" (separator) "plan.pdf")
.: "C:/projects/project1/plan.pdf"
> (path "C:/projects/project1/" (file name "plan.pdf"))
.: "C:/projects/project1/plan.pdf"
> (path "C:/projects/project1/" (separator) "plan" (separator file) "pdf")
.: "C:/projects/project1/plan.pdf"
```

Related

file, folder, separator, url

pattern

Creates a pattern containing the supplied expressions.

Syntax

(pattern *expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------|
| expression | expression | 1+ | An expression. |

Results

| Data Type | Description |
|-----------|-------------|
| premise | A premise |

Remarks

Returns an unevaluated pattern.

Example

```
> (pattern Prediction ^ as ?p :Hypothesis as ?h)
.: [Prediction ^ as ?p :Hypothesis as ?h]

> (pattern Hypothesis as ?h :Before as ?b :After as ?a (< ?b ?a))
.: [Hypothesis as ?h :Before as ?b :After as ?a (< ?b ?a)]

> (pattern {Solution_1 Solution_2} as ?s :Timeframe >= ?t)
.: [Solution ^ {Solution_1 Solution_2} for ?s :Timeframe >= ?t]
```

Related

thought, id, premise

pause

Pauses an agent.

Syntax

(**pause** *agent*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| agent | agent | 1 | An agent |

Results

| Data Type | Description |
|-----------|-----------------|
| agent | The agent task. |

Remarks

This function pauses an asynchronous agent task that processes incoming messages and performs a default job.

Example

```
> (let ?url (url scheme tcp port 5001))

.: true

> (function reply {?sender ?message} (tell ?sender "OK"))

.: reply

> (function job {& ?args} (tell user ($ 1)) (wait 0.5))

.: job

> (agent ?url reply job) (start Agent-1)

.: Agent-1 Agent-1

> (pause Agent-1) (wait 3) (start Agent-1) (wait 3) (cancel Agent-1) (free Agent-1)
111111..: cancelled freed
```

Related

ask, cancel, listen, pause, perform, start, tell

perform

Invokes a call.

Syntax

(**perform** *environment method instance expression ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|--------------------------------------|
| environment | environment | 0-1 | An environment (defaults to (scope)) |
| method | method | 1 | The method to be invoked |
| instance | instance | 1 | A prototype instance |
| expression | variant | 0+ | Additional arguments |

Results

| Data Type | Description |
|-----------|-------------------------------|
| variant | A value returned by the call. |

Remarks

Invokes the method using the provided (or the current) environment.

Example

```
> (relation A
  :Datum 0
  !incr  (method {}
            (let ?me :Datum ?n)
            (put ?me :Datum (++ ?n))))  

.: A  

> (new A)  

.: A_1  

> (perform (scope) !incr A_1)  

.: 1  

> (perform !incr A_1)  

.: 2
```

Related

apply, call, closure, eval, function, macro, method, scope, supply

pi

Returns 3.141592653589793.

Syntax

(**pi**)

Module

Math

Parameters

None

Results

| Data Type | Description |
|-----------|------------------------------------|
| real | The value of pi, 3.141592653589793 |

Remarks

Returns the constant 3.141592653589793

Example

```
> (pi)  
. : 3.141592653589793
```

Related

e

pick

Creates a list of randomly selected elements.

Syntax

(**pick** *sequence quantity*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| sequence | sequence | 1 | A sequence of elements |
| quantity | integer | 0-1 | The number of items to select. Default is 1. |

Results

| Data Type | Description |
|-----------|--|
| sequence | A sequence containing the selected elements. |

Remarks

Returns the specified quantity of randomly-chosen elements from the sequence. The quantity must be a number between 1 and the sequence size.

Example

```
> (function chunk (+ 7 (random -2 to +2)))
.: chunk

> (pick {a b c d e f g h i j k l m n o p q r s t u v w x y z} (chunk))
.: {k j z h q f}

> (pick "abcdefghijklmnopqrstuvwxyz" (chunk))
.: "cfilnsw"
```

Related

random, shuffle

pipe

Creates a pipe.

Syntax

(**pipe** *option ...*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| option | expression | 1+ | A key value pair. One of host <i>host</i> - name of the host computer port <i>number</i> – a port number. path <i>name</i> - name of the pipe mode <i>mode</i> - either read , write , or readwrite (default). content type - either binary or text (default). address url - url string.. |

Results

| Data Type | Description |
|-------------|-----------------|
| io resource | A file resource |

Remarks

Creates a pipe for reading or writing.

Example

```
> (let ?pipe (pipe address "pipe://localhost:4500/MyPipe"))

.: true

> (function reply {?target ?message} (tell ?target "goodbye."))
   .: reply

> (service ?pipe reply)
   .: Service-1

> (ask ?pipe "hello")

.: "goodbye."

> (cancel Service-1)  (free ?pipe)  (free Service-1)
   .: cancelled freed freed
```

Related

close, erase, path, read, service, socket, write

pop

Returns the element removed from a sequence.

Syntax

(pop sequence position)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-----------------------------|
| sequence | sequence | 1 | An assortment or sequence. |
| position | integer | 0-1 | A position. (defaults to 1) |

Results

| Data Type | Description |
|-----------|---------------------|
| variant | The removed element |

Remarks

If position is omitted the first element of the sequence shall be deleted. The sequences is modified.

Example

```
> (pop {a b c d})  
. : a  
  
> (pop {a b c d} 2)  
. : b  
  
> (pop {a b c d e f g} 7)  
. : g
```

Related

@ *element*, add,bind, cut, deq, enq, get, key, keys, put, #, values, zap

posit

Writes module expressions to a url.

Syntax

(posit *module url*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| module | module | 1 | A module. |
| url | url | 0-1 | A url. |

Results

| Data Type | Description |
|-----------|---|
| literal | Returns the literal posited if successful. |

Remarks

This function writes all expression in the module to a file named "<module>.th".

Example

```
> (posit Geometry (folder path "c:/projects/premise/"))

.: posited

> (posit MyModule (url path "//MyServer/Misc/"))

.: posited
```

Related

dependencies, discard, file, folder, forgo, module, modules, require, grok, use, url

position

Finds the position of an element or subsequence within a sequence.

Syntax

(**position** *sequence element option ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|--|
| sequence | sequence | 1 | The sequence to search |
| element | value | 1 | The item to be located |
| option | expression | 0+ | Key value pairs containing any of the following keys: from integer – the beginning position, (default is 1). to integer – a stop position, or # (default) for end of list go direction – forward (default) or backward . |

Results

| Data Type | Description |
|-----------|---|
| variant | The position of the item, or nil if not found. |

Remarks

None

Example

```
> (position {a b c d e f g h} f)
.: 6

> (position {a b c d e f g h} {f g h})
.: 6

> (position {a b a} a go backward)
.: 3

> (position {a b a} a go backward from 2)
.: 1

> (position "abcdefg" "cde" from 4)
.: nil
```

Related

@ *element*, gather, in, occurs, pick

positions

Returns positions of an element in a sequence or position value pairs.

Syntax

(positions *sequence* *element* *option ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|--|
| sequence | sequence | 1 | The sequence to search |
| element | value | 0 | The item to be located |
| option | expression | 0+ | Key value pairs containing any of the following keys: from integer – the beginning position, (default is 1). to integer – a stop position, or # (default) for end of list go direction – forward (default) or backward . |

Results

| Data Type | Description |
|-----------|--|
| list | A possibly empty list containing the positions of the element. |

Remarks

If element is omitted, the function returns a list containing position value pairs for the sequence.

Example

```
> (positions {a b c d} b)
.: {2}
> (positions {a b c d} x)
.: {}
> (positions {a b c b d b} x)
.: {2 4 6}
> (positions (tuple :X nil :Y nil :Z apple))
.: {{1 :X}{2 nil}{3 :Y}{4 nil}{5 :Z}{6 apple}}
> (positions "abcd")
.: {{1 "a"}{2 "b"}{3 "c"}{4 "d"}}
```

Related

bindings, definitions, entries, position

positive-p

True if a number is greater than zero.

Syntax

(**positive-p** *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the number is greater than zero |

Remarks

None.

Example

```
> (positive-p 500)
.: true
> (positive-p 0)
.: false
> (positive-p -25)
.: false
```

Related

even-p, negative-p, odd-p, zero-p

premise

Creates a premise representation of a value.

Syntax

(premise *value* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A variant. |

Results

| Data Type | Description |
|-----------|-------------|
| premise | A premise |

Remarks

None.

Example

```
> (premise 1)
.: [Integer :Value 1]
> (premise "The quick brown fox")
.: [String :Value "The quick brown fox" :Size 19]
> (relation Foo :Item)
.: Foo
> (premise (new Foo :Item bar))
.: [Foo ^ Foo_1 :Item bar]
```

Related

id, position, known, new, old, relation, structure, the, with

probability

Calculates the probability over a series.

Syntax

(**probability** *events A operation B*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|---|
| events | list | 1 | A list of elements representing events |
| A | expression | 1 | A key value pair. One of of event - a list element that predicate - a unary truth function. |
| operation | literal | 0-1 | An operation. One of { and , or , given } |
| B | expression | 0-1 | A key value pair. One of of event - a list element that predicate - a unary truth function. |

Results

| Data Type | Description |
|-----------|---|
| number | The probability of the supplied expression. |

Remarks

Calculates the probability of A in a list, or a conditional probability if B is also provided. The operator may be one of {and, or, given}.

Example

```
> (probability {head tail head tail} of tail)
.: 0.5

> (probability {1 2 3 4 5 6} that even-p and that (given {?n} (< ?n 4)))
.: 0.166666666667

> (probability {1 2 3 4 5 6} that even-p given that (given {?n} (< ?n 4)))
.: 0.083333333333
```

Related

statistics

problem

Creates a problem.

Syntax

(**problem** *name element*)

Module

KB

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|--|
| name | literal | 0-1 | The name of the domain. (If omitted a name is created). |
| element | expression | 0+ | A key and value expression. Any of domain the name of a domain state a problem state premise start a start time until an ending time |

Results

| Data Type | Description |
|-----------|----------------------------------|
| literal | Returns the name of the problem. |

Remarks

None.

Example

```
> (domain FixIt
  (relation Car :Age :Condition)
  (rule
    (if
      [Car :Age /= old :Condition = good]
      do
        (tell user ($ ($$ \n that) is possible. \n)
        then done)))

.. FixIt

> (problem domain FixIt state [Car :Age new :Condition good])
.. Problem_1

> (complete (think Problem_1))

.. {Problem_1}

that is possible.
```

Related

[cancel](#), [domain](#), [domains](#), [infer](#), [matchproblem](#), [problems](#), [rule](#), [rules](#), [solve](#)

problems

Returns a list of defined problems.

Syntax

(problems)

Module

KB

Parameters

None.

Results

| Data Type | Description |
|-----------|--|
| variant. | Returns a list of all problems or the element void if none exist.. |

Remarks

Problems are defined with the **problem** function..

Example

```
> (domain FixIt
    (relation Car :Age :Condition)
    (rule
      if
        [Car :Age /= old :Condition = good]
      do
        (tell user ($ ($$ \n that) is possible. \n)
      then done)))

.: FixIt

> (problem state [Car :Age new :Condition good])
.: Problem_1

> (problems)
.: {Problem_1}
```

Related

cancel, domain, domains, infer, match, problem, problems, rule, rules, solve

procedure

Creates a public function that returns `\%void` within a module.

Syntax

`(procedure scope name parameters expression ...)`

parameters ::=
parameters ::= {}
parameters ::= {required ... }
parameters ::= {required ... + optional ... }
parameters ::= {required ... + optional ... % keyword }
parameters ::= {required ... + optional ... % keyword ... & remaining }
parameters ::= {required ... % keyword ... & remaining }
parameters ::= {required ... & remaining }
parameters ::= { + optional ... }
parameters ::= { + optional ... % keyword }
parameters ::= { + optional ... % keyword ... & remaining }
parameters ::= { + optional ... & remaining }
parameters ::= { % keyword ... }
parameters ::= { % keyword ... & remaining }
parameters ::= { & remaining }

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-------------|-----|---|
| scope | environment | 0-1 | An environment. If not supplied, a new one is made. |
| name | literal | 0-1 | The name of the function. If not supplied then the name shall follow the pattern <code>fn-<number></code> . |
| parameters | list | 0-1 | A list containing variables, keywords, sigils, and keyword symbol associations. |
| expression | variant | 1+ | One or more expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|--------------------------|
| literal | The name of the function |

Remarks

Procedures are functions defined using the **procedure** intrinsic function. If the name is supplied then the procedure is a user named function (also called an **exonym**), otherwise it is an automatically named function (called an **esonym**). If a parameter list is supplied then it shall bind the formal parameter variables to any actual parameter expressions provided. The first variables in the parameter list are required. If a plus sign is provided, then those variables following the plus sign are optional. Each optional parameter may have a default value (specified with an equal sign and the value). If no default value is specified, the default value shall be **nil**. If a percent sign is supplied the next variables are keyword variables where the supplied slot shall have the same name as the symbol parameter excepting the first character of the name which shall be a colon for the slot and a question mark for the symbol. If an ampersand is provided, then the symbol following it is bound to a list of remaining (variadic) actual parameter values. The entire parameter list may be omitted.

A procedure returns the taxon **\%void**.

Example

```
> (procedure {reference ??n} (++ ??n))
.: fn-1

> (so {?x 100} (fn-1 ?x))

.: \%void

> (so {?x 100} (fn-1 ?x) ?x)

.: 101

> (procedure plusN {reference ??n + integer ?i = 1} (++ ?n ?i))

.: plusN

> (so {?y 100} (plusN ?y))

.: \%void

> (so {?y 100} (plusN ?y) ?y)

.: 101
```

Related

arguments, call, extend, function, functions, junction, macro, module, modules, parameters

proceed

Transfers control to a location.

Syntax

(proceed *location* **set** *assignment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| location | location | 1 | A location resource. |
| set | literal | 0-1 | The literal set . |
| assignment | expression | 0+ | Symbol value pairs for extant variables at the location. |

Results

| Data Type | Description |
|-----------|---------------|
| truth | Returns true. |

Remarks

Transfers control to the specified location; then sets the indicated variables to new values. If the variables are not defined at the location, an unbound symbol failure occurs.

Example

```
> (function factorial_hps {?n ?k}
  (let ?a 1)
  (location ?m)
  (if (<= ?n 0) (proceed ?k set ?r ?a)
    else (proceed ?m set ?n (* (<-- -- ?n) ?a)))))

.: factorial_hps

> (function factorial {?n}
  (let ?r nil)
  (location ?k)
  (if (any ?r) ?r
    else (factorial_hps ?n ?k)))))

.: factorial

> (factorial 5)

.: 120
```

Related

location, go

product

Multiplies an expression over a range of values.

Syntax

(**product** *symbol*... *binding* *sequence* *expression*...)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--|
| symbol | symbol | 1+ | A symbol. |
| binding | literal | 0-1 | The literal in or per . |
| sequence | sequence | 0-1 | A sequence. |
| expression | variant | 1+ | any expression involving variables from the pattern(s) |

Results

| Data Type | Description |
|-----------|---|
| variant | The result of applying multiplication to the last expression for the indicated range. |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to subelements of each subsequence. If there are insufficient elements to bind to the variables then the function fails. The last expression shall be evaluated and the plus function shall be applied to both that expression and any prior result. The final value is returned.

Example

```
> (product ?x in (range 1 to 10) ?x)
.: 55

> (generator OneToTen {}
  (step ?i from 1 to 10 (yield ?i))
  (done))

.: OneToTen

> (product ?x in (OneToTen) ?x)
.: 55
```

Related

fold, summation

punctuation-p

True if the first position of a string is punctuation.

Syntax

(**punctuation-p** *string*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| string | string | 1 | A value.. |

Results

| Data Type | Description |
|-----------|--|
| truth | Returns true if the first position is punctuation. |

Remarks

None

Example

```
> (punctuation-p " the quick brown fox ")
.: false

> (punctuation-p "This and that")
.: false

> (punctuation-p "400 years")
.: false

> (punctuation-p "....")
.: true
```

Related

alphanumeric-p, lowercase-p, uppercase-p, whitespace-p

push

Modifies a sequence by inserting a value.

Syntax

(push *sequence* *value* *position* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | sequence | 1 | A sequence |
| value | variant | 1 | The value to be inserted. |
| position | integer | 0-1 | The position to insert the value. (Default is 1.) |

Results

| Data Type | Description |
|-----------|------------------------|
| sequence | The modified sequence. |

Remarks

If position is omitted the value is prepended to the sequence. If # is used for the position, the value is appended to the sequence. Otherwise, the value is inserted at the indicated position. The sequence itself is modified.

Example

```
> (push {a b c} d)
.: {d a b c}
> (push {a b c} d 2)
.: {a d b c}
> (push "abcdef" g #)
.: "abcdefg"
```

Related

size, add, cut, deq, enq, get, insert, key, keys, put, values, zap

put

Modifies an assortment or sequence by updating values.

Syntax

(**put** *container entry ...*)

entry ::= key value

entry ::= position value

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|----------------------------|
| container | thing | 1 | An assortment or sequence. |
| entry | expression | 1+ | A key and value pair. |

Results

| Data Type | Description |
|-----------|--|
| thing | Returns the modified assortment or sequence. |

Remarks

For assortments, the key(s) to be updated must be present, for sequences, the positions to update must be within the sequence, otherwise the function fails. Keys and positions are added via the **add** function. Positions may be also added using **append**. Use **copy** or **clone** to return a new container. A coordinate is a position list. A list of keys may be provided for an entry as well, in which case the keys are traversed and the value is placed at the last key. To create a new container use **replace**.

Example

```
> (let ?a (lexicon x 1 y 2 z 3) ?s {a b c d e})  
.: true  
> (entries (put ?a y 12 z 13))  
.: {{x 1}{y 12}{z 13}}  
> (put ?s 2 hello 3 goodbye)  
.: {a hello goodbye d e}
```

Related

add, cut, get, has, key, keys, lacks, let, values

qualified

Prepends a module to a function name.

Syntax

(qualified *module* *function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------|
| module | literal | 0-1 | A module literal. |
| function | literal | 1 | A function or macro literal. |

Results

| Data Type | Description |
|-----------|--|
| literal | The fully qualified literal with the module prepended to the function. |

Remarks

Returns a fully qualified function literal. If omitted, the first function matching the literal is returned.

Example

```
> (module Math
    (function sum {& ?args} (apply + ?args)))

.: Math

> (qualified sum)

.: User.sum

> (qualified Math sum)

.: Math.sum

> (require Math)

.: Math

> (qualified sum)

.: Math.sum
```

Related

functions, modules, qualifier, unqualified

qualifiers

Creates a list of modules associated with an identifier.

Syntax

(qualifiers *identifier*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---------------|
| identifier | identifier | 1 | An identifier |

Results

| Data Type | Description |
|-----------|--|
| list | The modules associated with the literal. |

Remarks

Checks the function against known modules and returns the modules in which the function is defined.

Example

```
> (qualifiers sum)
.: {}

> (module Math
    (function sum ?args (apply + ?args)))
.: Math

> (use User)
.: User

> (require Math)
.: Math

> (qualifiers sum)
.: {Math}
```

Related

functions, modules, qualified, unqualified

quantify

Returns a quantifier for elements satisfying a test.

Syntax

(**quantify** *sequence predicate option ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|--|
| sequence | sequence | 1 | A sequence |
| predicate | function | 1 | A truth expression |
| option | expression | 0+ | A key value pair. One of as <i>format</i> - either the literal word (default) or percent . |

Results

| Data Type | Description |
|-----------|---|
| variant | Returns a percentage or the words none , few , half , most , all . |

Remarks

None.

Example

```
> (quantify {1 2 3 -4 5 6 7 8 9 10} (given {?n} (< ?n 0)))  
.: few  
  
> (quantify {1 2 3 4 5 6 7 8 9 10} number-p as percent)  
.: 1.0  
  
> (quantify {1 2 3 4 5 6 7 8 9 10} odd-p)  
.: half  
  
> (quantify {1 2 3 4 5 6 7 8 9 10} negative-p)  
.: none  
  
> (quantify {1 2 3 5 6 7 8 9} odd-p)  
.: most
```

Related

coalesce, collect, exactly, filter, gather, quantity

quantity

Returns the number of elements satisfying a test.

Syntax

(quantity symbol ... binding sequence expression ... test)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--------------------------------------|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per . |
| sequence | sequence | 1 | A sequence (string or list) |
| expression | expression | 0+ | An expression. |
| test | truth | 1 | A truth expression |

Results

| Data Type | Description |
|-----------|---|
| number | Returns the number of elements satisfying the test. |

Remarks

None.

Example

```
> (quantity ?n in {1 2 3 4 5 6 7 8 9 10}
   (is ?n (given {?x} (= (taxon ?x) number))))  

.: 10  

> (summation ?n in {1 2 3 4 5 6 7 8 9 10} (bracket (odd-p ?n)))  

.: 5  

> (quantity ?n in {1 2 3 4 5 6 7 8 9 10} (odd-p ?n))  

.: 5  

> (quantity ?n in {1 2 3 4 5 6 7 8 9 10} (negative-p ?n))  

.: 0
```

Related

coalesce, collect, exactly, filter, gather, quantify

radians

Converts degrees to radians.

Syntax

(radians *degrees*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|----------------------|
| degrees | number | 1 | a number in degrees. |

Results

| Data Type | Description |
|-----------|--------------------|
| number | Number of radians. |

Remarks

For the conversion the degrees are multiplied by $\pi / 180$, or roughly 0.0174532922222222.

Example

```
> (radians 200)
.: 3.490658444444444
> (radians 120)
.: 2.0943950666666667
> (radians 45)
.: 0.7864815
```

Related

cosine, degrees, sine, tangent

radix

Converts a number to a base.

Syntax

(**radix** *number* *base*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--|
| number | number | 1 | An integer or unsigned number to convert |
| base | integer | 0-1 | A number between 2 and 36. (default is 10) |

Results

| Data Type | Description |
|-----------|--|
| number | The number converted to the base as as an integer or long. |

Remarks

None.

Example

```
> (radix 15 8)
.: \#n08r17

> (radix (unsigned 89975) 16)
.: \#u16r15F77

> (radix \#nx15F77)
.: \#n10r89975

> (radix #gd10 10)
.: \#g10r10

> (radix (integer 2.003) 2)
.: \#n02r10
```

Related

big, complex, float, imaginary, integer, long, real, unsigned

random

Creates a random number.

Syntax

(random lower to upper precision)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--|
| lower | number | 1 | A lower bound |
| to | literal | 1 | The literal to |
| upper | number | 1 | An upper bound |
| precision | literal | 0-1 | Either integer (default), real or float |

Results

| Data Type | Description |
|-----------|------------------|
| number | A random number. |

Remarks

If the precision is omitted the default precision is **integer**. If precision is omitted and the bounds are zero and one, then the default precision is **real**. If the precision is supplied, then it is used.

Example

```
> (random 0 to 1)
.: 0.325377881526947

> (random 0 to 1 float)
.: 0.8128264546394348216582

> (random 0 to 100)
.: 63

> (random 1000 to 2000 real)
.: 1194.230922829
```

Related

[pick](#), [shuffle](#)

range

Creates a list of numbers.

Syntax

(range *first* to *last* by *increment*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---|
| first | number | 1 | The initial value of the symbol |
| limit | literal | 1 | The literal to , above , below or go . (default is to). |
| last | number | 1 | The final value of the symbol |
| by | literal | 0-1 | by |
| increment | number | 0-1 | The positive or negative increment for the symbol |

Results

| Data Type | Description |
|-----------|---|
| list | A list containing the numbers in the range. |

Remarks

If **to** is specified, the last number is included in the iteration. If **below** is specified the *last* value is excluded from the iteration. If **above** is specified the next number above the *last* value is included in the iteration. If **go** is specified, then the *last* value specifies the number of iterations.

Example

```
> (range 1 to 5)
.: {1 2 3 4 5}

> (range 1 to 10 by 2)
.: {1 3 5 7 9}

> > (range 100 go 5)
.: {100 101 102 103 104 105}
```

Related

[for](#), [step](#)

rational

Converts values to a rational number.

Syntax

(rational numerator denominator)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-----------|-----|--|
| numerator | variant | 1 | An integer or the literal minimum or maximum . |
| denominator | variant | 0-1 | A non-zero integer.(Default is 1.) |

Results

| Data Type | Description |
|-----------|--------------------|
| number | A rational number. |

Remarks

All integers are rational numbers with a denominator of 1. If a non-zero denominator is supplied, it must be an integer. If a denominator is not supplied, then an attempt is made to convert the value to a rational number.

Example

```
> (rational {a b c})  
. : [Failure :Name ArgumentValue :Text "{a b c} is not number"]  
  
> (rational "a b c")  
. : [Failure :Name ArgumentValue :Text "'a b c' is not number"]  
  
> (rational 500)  
. : 500/1  
  
> (rational 500.0)  
. : 500/1  
  
> (rational "23.4")  
. : 234/10
```

Related

[complex](#), [float](#), [imaginary](#), [integer](#), [is](#), [real](#)

read

Creates a string or stream.

Syntax

(read *file count bits*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---|
| file | resource | 1 | An file, or pipe. |
| count | variant | 0-1 | The number of bytes to read, the literal all , or the literal line . (Default is all.) |
| bits | integer | 0-1 | If count is provided, then this represents the size of the character in bits. Either 7, 8, 16,o,r 32. (Default is 8.) |

Results

| Data Type | Description |
|-----------|---|
| variant | A string, stream, or the value nothing (if at end of file). |

Remarks

If the literal **all** is provided for count, the entire file (or stream) is read. If **line** is provided, one line is read (up to the newline character). If count is omitted, then the entire file read.

Example

```
> (global ?File (file name "test.txt" seek 1 erase yes)) ; open at BOF
.: true
> (seek ?File 1)
.: File-1
> (read ?File all)
.: "Hello, world."
> (close ?File)
.: closed
```

Related

bytes, close, take, peek, peep, file, seek, write

ready-p

True if a task has completed.

Syntax

(ready-p task)

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|------------------|
| task | task | 1 | A task resource. |

Results

| Data Type | Description |
|-----------|--------------------------------|
| truth | True if the task has completed |

Remarks

Returns true if the task has completed, false otherwise.

Example

```
> (let ?task (concurrent (@ (concurrent (idle \@s100)))))

.: true

> (do (idle \@s20)
      (if (ready-p ?task)
          (free ?task)
          else
          running))

.: running
```

Related

[wait](#), [abort](#), [await](#), [busy-p](#), [cancel](#), [cancelled-p](#), [complete](#), [reclaim](#), [tarry](#), [task](#)

real

Converts a value to a real number.

Syntax

(real *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 1 | A number or the literal minimum or maximum . |

Results

| Data Type | Description |
|-----------|----------------|
| number | A real number. |

Remarks

If the value cannot be converted to a real number, the function fails.

Example

```
> (real {a b c})  
. : [Failure :Name ArgumentValue :Text "{a b c} is not number"]  
  
> (real "a b c")  
. : [Failure :Name ArgumentValue :Text "'a b c' is not number"]  
  
> (real 500)  
. : 500.0  
  
> (real 500.0)  
. : 500.0  
  
> (real "23.4")  
. : 23.4
```

Related

[complex](#), [float](#), [imaginary](#), [integer](#), [long](#), [rational](#)

reclaim

Performs garbage collection.

Syntax

(reclaim)

Module

Base

Parameters

None.

Results

| Data Type | Description |
|-----------|--|
| literal | Returns the literal reclaimed when completed. |

Remarks

None.

Example

```
> (reclaim)
.: reclaimed
```

Related

[free](#)

reduce

Converts a sequence into a value.

Syntax

(reduce *sequence function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence |
| function | function | 1 | A function. |

Results

| Data Type | Description |
|-----------|--|
| value | The return value of the function applied to all the items in succession. |

Remarks

Applies the binary function to each item of a list and returns the final value.

Example

```
(reduce {1 2 3 4 5} *)
.: 120

(reduce {1 2 3 4 5} +)
.: 15

(reduce {3 2 1} **)
.: 9
```

Related

[fold](#)

reference

Returns a reference to a symbol.

Syntax

(reference *symbol*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| symbol | symbol | 1 | A symbol |

Results

| Data Type | Description |
|-----------|-----------------------------|
| reference | Returns a symbol reference. |

Remarks

The convention for denoting reference variables is to use two question marks: ??name.

Example

```
> (structure ListNode :Val :Next)
  .: ListNode

> (function IsPalindrome2 {ListNode ?tail reference ??head}
  (ensure ListNode (actual ??head))
  (if (null-p ?tail)
    (return true)
  or (and (IsPalindrome2 (:Next ?tail)) (= (:Val ?tail) (:Val ??head)))
    (set ??head (:Next ??head))
    (return true)
  else (return false)))

.: IsPalindrome2

> (function IsPalindrome {?head}
  (ensure ListNode ?head)
  (return (IsPalindrome2 ?head (reference ?head)))))

.: IsPalindrome

> (IsPalindrome
  (new ListNode :Val a :Next (new ListNode :Val b :Next (new ListNode :Val a))))
  .: true
```

Related

actual, assume, dynamic, given, global, let, scope, set, tie

relation

Creates a relation.

Syntax

(relation *name inclusion ... definition ...* **)**

inclusion ::= uses {prototype ... }

definition ::= slot

definition ::= slot default-value

definition ::= method function

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|------------------------------|
| name | literal | 1 | The name of the relation |
| inclusion | expression | 0+ | A inclusion expression |
| definition | expression | 0+ | A slot definition expression |

Results

| Data Type | Description |
|-----------|----------------------------------|
| literal | The name of the defined relation |

Remarks

Relations are assortment prototypes that represent persistent relationships among one or more values. A relation is comprised of a name, attributes (collectively called slots) with optional default values, and methods. Relations are defined using the **relation** function. A SubThought is a relation record created using the **new** function. A relation must be defined before a SubThought can be formed. Any number of slots or methods can be specified, and default values may also be defined. A sealed relation cannot create new thoughts.

Example

```
> (relation Employee
  :Salary
  :Name
  :Supervisor
  :Reclaim    false
  !SalaryYTD  Employee_salaryYTD
)

.: Employee

> (new Employee :Name "Jane Dough" :Salary 2000000.00)

.: Employee_1
```

```
> (premise Employee_1)
. : [Employee ^ Employee_1 :Salary 2000000.0 :Name "Jane Dough" :Supervisor nil
     :Recalim false !SalaryYTD Employee_salaryYTD]
```

Related

[new](#), [nix](#), [old](#), [relation-p](#), [relations](#), [seal](#), [structure](#), [structures](#), [unseal](#)

relations

Returns a list of defined relations.

Syntax

(relations)

Module

Base

Parameters

None

Results

| Data Type | Description |
|-----------|--|
| list | Returns a list of all defined relations. |

Remarks

None

Example

```
> (relation Animal
  :Phylum :Class :Order :Family :Genus :Species :Color :Weight :Height
  )

.: Animal

> (relations)

.: {Animal}

> (nix Animal)

.: nixed

> (relations)

.: {}
```

Related

structures, structure, new, nix, old, relation,

release

Releases locks on critical sections.

Syntax

(**release** *locks*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---|
| locks | list | 1 | A list of literals denoting critical section locks. |

Results

| Data Type | Description |
|-----------|--|
| literal | Returns the literal released if successful. |

Remarks

Forces the release of critical section locks.

Example

```
> (function pp {+ ?a ?b ?c ?d}
  (critical {A B} (tell user ($ ?a ?b)))
  (critical {C D} (tell user ($ \s ?c ?d \n))))  
  
. : pp  
  
> (free      (complete (critical {A B C D} (idle \@s30)))
  (do (idle 5)
      (release {A B C D})
      (tarry   (complete
                  (pp the quick brown fox)
                  (pp jumped over lazy dogs)
                  (pp four score and seven)))))  
  
the quick jumped over brown fox
four score lazy dogs
and seven
.: freed
```

Related

[critical](#)

remove

Creates a new assortment by removing values.

Syntax

(**remove** *assortment value ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------|
| assortment | assortment | 1 | An assortment. |
| value | variant | 1+ | A value. |

Results

| Data Type | Description |
|------------|---------------------|
| assortment | The new assortment. |

Remarks

To remove elements destructively use the **rip** function.

Example

```
> (entries (collection a b c d))
.: {{a a} {b b} {c c} {d d}}
> (entries (remove (lexicon a 1 b 2 c 3 d 4) 2 4))
.: {{a 1}{c 3}}
```

Related

[add](#), [combine](#), [cut](#), [rip](#)

repeat

Loops a specified number of times.

Syntax

(repeat count expression ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---------------------------|
| count | variant | 1 | The number of iterations. |
| expression | expression | 0+ | A call to be evaluated. |

Results

| Data Type | Description |
|-----------|--|
| variant | returns the last expression on the last iteration. |

Remarks

Repeats the expression(s) the specified number of times. An implicit counter from 0 to the designated count. Count may be an integer or long value.

Example

```
> (global ?i 0)
.: true
> (repeat 10 (--> ++ ?i))
.: 10
> (repeat 10 (--> -- ?i))
.: 0
> (repeat 1000 (--> ++ ?i))
.: 1000
> ?i
.: 1000
```

Related

do, for, loop, step, until, while

replace

Creates a new assortment or sequence by updating values.

Syntax

(replace *container entries option*)

entry ::= key value

entry ::= position value

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|---|
| container | thing | 1 | An assortment or sequence. |
| entries | variant | 1+ | A list of key value pairs or index value pairs, or a lexicon. |
| options | expression | 0-1 | |

Results

| Data Type | Description |
|-----------|--|
| thing | Returns the modified assortment or sequence. |

Remarks

For assortments, the key(s) to be updated must be present, for sequences, the indices to update must be within the sequence, otherwise the function fails. Keys and positions are added via the **add** function. Positions may be also added using **append**. Use **copy** or **clone** to return a new container. A coordinate is a position list. A list of keys may be provided for an entry as well, in which case the keys are traversed and the value is placed at the last key.

Example

```
> (let ?a (lexicon x 1 y 2 z 3) ?s {a b c d e})  
. : true  
  
> (entries (replace ?a {{y 12}{z 13}}))  
. : {{x 1}{y 12}{z 13}}  
  
> (replace ?s {{2 hello}{3 goodbye}} by index)  
. : {a hello goodbye d e}
```

Related

add, cut, get, has, key, keys, lacks, let, values

require

Retrieves an absent module.

Syntax

(**require** *module* **as** *moniker* **from** *url* ... *options* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------------|------------|-----|--|
| module | literal | 1 | The name of the required module |
| as | literal | 0-1 | The literal as . |
| moniker | literal | 0-1 | A literal. |
| from | literal | 0-1 | The literal from . |
| url | url | 0+ | URLs from which to retrieve the module. |
| options | expression | 0+ | Any of the following update choice where choice is yes or no . |

Results

| Data Type | Description |
|-----------|-------------|
| literal | The module |

Remarks

Allows the current module to access functions in the specified module using a moniker. The **from** URLs shall be evaluated if the module is either not present or if the update option is **yes**. Grok is applied to the urls in succession until one url is available to be read.

Example

```
> (mySum 1 2 3)
.: [Failure :Name NotFound :Text "The function mySum isn't found"]
> (require MyMath as m from (file "lib/mymath.th"))
.: MyMath
> (mySum 1 2 3) (m.mySum 1 2 3)
.: 6 6
```

Related

dependencies, discard, extend, forgo, module, modules, run, use

reset

Resets a series for reuse.

Syntax

(reset *series*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| series | series | 1 | A series. |

Results

| Data Type | Description |
|-----------|---------------------|
| series | Returns the series. |

Remarks

After the series is reset, the **next** function must be used to move the series to the first element.

Example

```
> (global ?Series (series {a b c}))  
. : true  
  
> (while (next ?Series) (tell user ($$ (this ?Series) ,) ($)))  
a, b, c, . : told  
  
> (reset ?Series)  
. : Series-1  
  
> (this ?Series)  
. : [Failure :Name OutOfRange :Text "Must advance series to the first element."]  
  
> (next ?Series)  
. : true  
  
> (this ?Series)  
. : a
```

Related

[next](#), [series](#), [this](#)

resize

Modifies the length of a sequence.

Syntax

(resize *sequence length default*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| sequence | sequence | 1 | A sequence. |
| length | integer | 1 | The length of the final sequence |
| default | variant | 0-1 | A value to fill the sequence (default is nil or space). |

Results

| Data Type | Description |
|-----------|------------------------------|
| list | Returns a the modified list. |

Remarks

If the sequence is a string, the default value must be a single position string (space if unspecified). If the sequence is a list, the default value may be any value (**nil** if unspecified).

Example

```
> (let ?s (array 3 of 0))  
.: true  
  
> (fix ?s 1 1 2 2 3 3)  
.: {1 2 3}  
  
> (resize ?s 10 0)  
.: {1 2 3 0 0 0 0 0 0 0}  
  
> (resize ?s 2 0)  
.: {1 2}
```

Related

[array](#), [fill](#) , [pad](#)

rest

Creates a subsequence of all except the first elements.

Syntax

(**rest** *sequence* *skip*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| sequence | sequence | 1 | A list or string |
| skip | number | 0-1 | The number of elements to skip. Default is 1 |

Results

| Data Type | Description |
|-----------|--|
| sequence | The sequence without the skipped elements. |

Remarks

None

Example

```
> (rest {a b c d e})
.: {b c d e}

> (rest "abcdefg")
.: "bcdefg"

> (rest "abcdefg" 2)
.: "cdefg"

> (rest {a b c d e} 3)
.: {d e}

> (rest {a b c d e} 5)
.: {}
```

Related

but, last, top

retract

Deletes thoughts using a pattern.

Syntax

(**retract** *pattern* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|-------------|
| pattern | pattern | 1+ | A pattern. |

Results

| Data Type | Description |
|-----------|------------------------------|
| literal | The literal retracted |

Remarks

None.

Example

```
> (domain Counting
  (relation Num :Number)

  (rule
    so [Num :Number as ?n]
    if (< ?n 100)
    no [Num ^ as ?f :Number = (+ ?n 1)]
    do (knew Num :Number (+ ?n 1)))

  (rule CleanUp
    salience -1
    so [Num ^ as ?thought]
    do (retract [Num ^ = ?thought])))

.: Counting

> (rules)

.: {CleanUp Rule-1}

> (retract (rules))

.:retracted

> (rules)

.: {}
```

Related

wait, abort, await, task, complete, busy-p, ready,-p, reclaim

return

Terminates a function call.

Syntax

(**return** *value option ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| value | variant | 0-1 | Any value to be returned. (Default is nil .) |
| option | expression | 0-1 | A key value pair. One of: from <i>function</i> - returns from a function on the call graph. unwind <i>count</i> - unwinds the call graph a number of times.. |

Results

| Data Type | Description |
|-----------|---------------------|
| variant | The returned value. |

Remarks

Return exits from an encompassing function. If the value is not specified, **nil** is returned. When **return** is evaluated, the encompassing function immediately returns the value. If **from** is specified along with the name of a function, the first encompassing function having the same name is exited. If **unwind** is specified, then the call graph is unwound the indicated number of times.

Example

```
> (function foo
    (step ?item from 1 to 20
        (if (= ?item 10) (return found-it))
        not-found))

.: foo

> (foo)

.: found-it

> (function f1 {}
    (do (tell user ($ inside do. \n))
        (return 0))
    (tell user ($ within f1. \n)))

.: f1
```

```

> (function main
  (fn1)
  (tell user ($ within main. \n)))

.: main

> (main)

inside do.
within f1.
within main.
.: nil

> (function f1 {}
  (do (tell user ($ inside do. \n))
      (break 0))
  (tell user ($ within f1. \n)))

.: f1

> (main)

inside do.
within f1.
within main.
.: told

> (function f1 {}
  (do (tell user ($ inside do. \n))
      (break 0))
  (tell user ($ within f1. \n)))

.: f1

> (main)

inside do.
within f1.
within main.
.: told

> (function f1 {}
  (do (tell user ($ inside do. \n))
      (return 0 from f1))
  (tell user ($ within f1. \n)))

.: f1

> (main)

inside do.
within main.
.: told

```

```

> (function f1 {}
  (do (tell user ($ inside do. \n))
      (return 0 from f1))
  (tell user ($ within f1. \n)))

.: f1

> (main)

inside do.
within main.
.: told

> (function f1 {}
  (do (tell user ($ inside do. \n))
      (return 0 from f1))
  (tell user ($ within f1. \n)))

.: f1

> (main)

inside do.
within main.
.: told

> (function f1 {}
  (do (tell user ($ inside do. \n))
      (return 0 from main))
  (tell user ($ within f1. \n)))

.: f1

> (main)

inside do.
.: 0

> (let ?a (so {?y "hello"}
  (if (< (@ ?name) "M") (return ($ Goodbye ?name) unwind 2)
      else (tell user ($ Hello ?name)))
  (ask user ($ What is your favorite color?))
  ($ You're nice)))
.: true

> ?a

.: "You're nice"

```

Related

[continue](#), [do](#), [break](#), [fail](#), [function](#), [go](#), [location](#), [resume](#), [try](#)

reverse

Creates a reversed sequence.

Syntax

(reverse *sequence*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------|
| sequence | sequence | 1 | A list or string |

Results

| Data Type | Description |
|-----------|-----------------------|
| sequence | The reversed sequence |

Remarks

None.

Example

```
> (reverse "abcde")
.: "edcba"
> (reverse {a b c d e})
.: {e d c b a}
> (reverse [A :B C :D E])
.: [E :D C :B A]
> (reverse (' (a b c d e)))
.: (e d c b a)
```

Related

but, first, last, pick, rest, mid, top

right

True if values are equal or the first value is nil.

Syntax

(right *first second*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------------------|
| first | variant | 1 | A value to be compared. |
| second | variant | 1 | A value to be compared. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if all values are the same or if second is nil |

Remarks

Returns true if each value is the same or if the first value is **nil**, and false otherwise. Both values must be of the same type, and must be numbers, literals, or strings. Short circuits if false. Two items are equal if their representation is the same regardless of whether or not they occupy the same location in memory (viz. identical). Returns false if both values are false.

Example

```
> (right a a)
.: true

> (right "johnny" "ralph")
.: false

> (right nil 3)
.: true

> (right nil nil)
.: false
```

Related

< less , <= less or equal, > greater, >= greater or equal, /= unequal, full, identical, left

rip

Removes elements from assortments by value.

Syntax

(**rip** *assortment* *value* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------|
| assortment | assortment | 1 | An assortment. |
| value | variant | 1+ | A value. |

Results

| Data Type | Description |
|------------|--------------------------|
| assortment | The modified assortment. |

Remarks

Assortments are modified destructively. To delete nondestructively use the **omit** function.

Example

```
> (entries (collection a b c d))
.: {{a a} {b b} {c c} {d d}}
> (entries (rip (lexicon a 1 b 2 c 3 d 4) 2 4))
.: {{a 1}{c 3}}
```

Related

[cut](#), [omit](#)

round

Rounds a number to the nearest integer.

Syntax

(round number)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|----------------|
| number | number | 1 | A real number. |

Results

| Data Type | Description |
|-----------|---------------------|
| number | The rounded integer |

Remarks

If the number is zero, zero is returned. If the number is negative then if the tenths digit is five or less, it shall be rounded down towards negative infinity, otherwise it shall be rounded towards positive infinity. If the number is positive, then if the tenths digit is five or greater, it shall be rounded towards positive infinity, otherwise it shall be rounded towards negative infinity.

Example

```
> (round 0.5)
.: 1

> (round 0.4)
.: 0

> (round 0.49)
.: 0

> (round -0.5)
.: -1

> (round -0.4)
.: 0
```

Related

% modulus, ceiling, floor, truncate,

rule

Creates a rule.

Syntax

```
(rule name salience priority domain ruleset
    so bindings to goals if premises no premises options action then domain)
```

bindings ::= *assignment*
bindings ::= *assignment* *bindings*
bindings ::= *premise* *restriction* ... *bindings*
goals ::= *premise*
goals ::= *premise* *goals*
goals ::= *premise* *restriction* ... *goals*
facts ::= *premise*
facts ::= *premise* *facts*
facts ::= *premise* *restriction* ... *facts*
restriction ::= (*predicate* *value* ...)
assignment ::= [*category definition* ...]
assignment ::= [*category definition* ... *descriptor* ...]
premise ::= [*category descriptor* ...]
category ::= *type*
category ::= *list*
definition ::= *slot binding*
descriptor ::= *slot condition*
binding ::= **as** *symbol*
binding ::= **each** *symbol*
condition ::= *slot* = *value*
condition ::= *slot* **is** *unary-predicate*
condition ::= *slot* **not** *value*
condition ::= *slot* *binary-predicate* *value*
condition ::= *slot* *n-ary-predicate* *value-list*
condition ::= (*predicate* *value* ...)
option ::= **scan** *number*
option ::= **limit** *number*
option ::= **group** *slot* ... **by** *slot* ... **into** *symbol*
option ::= **merge** *value* ...
option ::= **sort** *ordering* ...
action ::= **deem** *value1* **else** *value2*
action ::= **do** *expression* ...
action ::= **give** *expression*
action ::= **list** *value* ...
action ::= **tally**
action ::= **yield** *expression*
ordering ::= *term comparator*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------------------|------------|-----|---|
| name | literal | 0-1 | The name of the rule. If omitted, the rule is auto-named. |
| properties | list | 0-1 | A list containing any of the following properties: :Salience <i>integer</i> – indicates the rule priority :Stage <i>literal</i> – indicate the rule stage |
| with | literal | 1 | The literal with . |
| clauses / premises | expression | 1+ | Either one or more clauses, or, one or more premises |
| option | expression | 0-2 | Either scan, limit, group, merge, sort, or any combination. |
| action | literal | 0-1 | Either deem, do, give, list, tally, yield. |
| expression | expression | 0+ | An expression |

Results

| Data Type | Description |
|-----------|-----------------|
| rule | Returns a rule. |

Remarks

The rule macro creates a rule to monitor premises. The rule is run spontaneously when premises match the preconditions. There is no guarantee as to when or whether or not the rule shall execute. The rule may be cancelled and deleted using the **reclaim** command.

Note that a predicate is a truth function, **scan** tells the number of matches to attempt (default is **infinity**), and **limit** tells how many results to return (default is **infinity**). If the action is **deem**, then the subsequent value is returned if the final descriptor is matched, otherwise the **else** value is returned. If the action is **do**, then for each match the expressions are run, and for the last match, the last value of the last expression is returned, or **nil** is returned if all descriptors were false or had no matches. If the action is **give**, then the expression is immediately returned from the **with** call on the first match, akin to **return from with**. If the action is **yield**, then the expression is immediately returned from the **with** call, but may be resumed with a **next** function call. When all matches are completed, the **done** function is implicitly called to terminate the generator. If the action is **group**, followed by a list of slots and values from which the group is picked, then the literal **by** and a set of slots and values that determines the grouping inclusion, and finally the literal **into** followed by a symbol to which the group shall be bound on each iteration. If the action is either **list** (or **merge**), then a (merged) list is returned or an empty list if nothing matched. If the action is **sort** then the list of slots and comparators should follow. (Note that **list** is required if the sort action is selected.) If the action is **tally**, then the number of matched premises is returned, or zero if no matches.

Example

```
> (domain Syllogisms

  (relation Fact :All :Are)

  (rule ExcludedMiddle
    so   [Fact :All as ?S :Are as ?M]
        [Fact :All = ?M :Are as ?P]
    do
      (knew Fact :All ?S :Are ?P))
  )

.: Syllogisms

> (problem
  facts [Fact :All people :Are mortal]
  [Fact :All philosophers :Are people])

.: Problem_1

> (start Problem_1 Syllogisms)

.: Problem_1

> (:Facts Problem_1)

.: {Fact_1 Fact_2 Fact_3}

> (get Fact_3)

.: [Fact ^ Fact_3 :All philosophers :Are mortal]

> (using Syllogisms (rules))

.: {ExcludedMiddle}

> (using Syllogisms (free ExcludedMiddle))

.: freed

> (using Syllogisms
  (rule {:Salience 100 :Stage 0}
    so   [Fact :All as ?S :Are as ?M]
        [Fact :All = ?M :Are as ?P]
    do
      (knew Fact :All ?S :Are ?P)))

.: Rule-1

> (using Syllogisms (rules))

.: {Rule-1}

> (using Syllogisms (free (rules)))

.: freed

> (using Problem_1 (which Fact ^ as ?a :Are mortal list ?a))

.: {Fact_1 Fact_3}
```

```
> (using Problem_1 (which Fact :Are mortal))  
. : {[Fact ^ Fact_1 :All people :Are mortal]  
     [Fact ^ Fact_3 :All philosophers :Are mortal]}
```

Related

reasoning, reclaim, rules, using, with

rules

Returns a list of rules defined in a domain.

Syntax

(rules domain)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|---|
| domain | literal | 0-1 | A module. If not provided defaults to the current module. |

Results

| Data Type | Description |
|-----------|--|
| list | A list of rules defined in the domain. |

Remarks

The domain must be a literal in the list obtained by calling the (**domains**) function. If the domain is not provided this function uses the current domain (obtained via the (**use**) function).

Example

```
> (domain Mine
  (rule foo  salience 0 if [A ^ as ?a] list ?a))

.: Mine

> (rules Mine)

.: {foo}

> (domain Affable
  (rule hello if [A ^ as ?a] list ?a))

.: Affable

>

.: hello

> (rules)

.: {hello}
```

Related

function, module, modules, variables

same

True if all values are equal or contain equal elements.

Syntax

(same *value* *value* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------------------|
| value | variant | 2+ | A value to be compared. |

Results

| Data Type | Description |
|-----------|----------------------------------|
| truth | True if all values are identical |

Remarks

Returns true if each value is the equal to the next value from left to right, sequences are the same if they are equal length and contain the same elements regardless of order.

Example

```
> (same a a a)
.: true

> (same "johnny" "ralph" "sam")
.: false

> (same 3 3 3)
.: true

> (same {a b c} {c b a} {c a b})
.: true

> (same "abc" "cba" "cab")
.: true

> (same {"abc"} {"cba"} {"cab"})
.: false
```

Related

= *equal*, < *less* , <= *less or equal*, > *greater*, >= *greater or equal*, /= *unequal*, identical

scale

Applies a function to a list of numbers and scalar values.

Syntax

(scale *list scalar ... function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------------------------|
| list | list | 1 | A list of numbers. |
| scalar | number | 1+ | A number. |
| function | function | 1 | The name of the function to invoke |

Results

| Data Type | Description |
|-----------|-------------------------|
| sequence | The resulting sequence. |

Remarks

None.

Example

```
> (scale {1 2 3} 5 +)
.: {6 7 8}
> (scale {1 2 3} 5 10 *)
.: {50 100 150}
```

Related

apply, collect, count, filter, gather, group, map, merge, reduce , separate, sort

scatter

Applies functions in parallel returning a list of results.

Syntax

(scatter arguments functions)

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---|
| arguments | list | 0-1 | The intial arguments to each function. |
| functions | list | 1 | An unordered list of functions to be applied. |

Results

| Data Type | Description |
|-----------|-----------------|
| list | Non nil results |

Remarks

This function supplies the arguments to each function and returns a list of results from all the functions evaluated in parallel. Sequential forms of this function are **compose** and **morph**.

Example

```
> (scatter {5} {++ -- (given {?x} (* ?x 2)))}
.: {6 4 10}

> (scatter {{the quick brown fox jumped over the lazy dogs}}
           {but rest top last})
.: {{the quick brown fox jumped over the lazy }
   {quick brown fox jumped over the lazy dogs}
   {the} {dogs}}

> (scatter {2 3 4} {+ * -})
.: {9 24 -5}
```

Related

apply, concurrent, complete, compose, map, morph

scope

Finds an environment or gets the current environment.

Syntax

(**scope** *keyval* ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|---|
| keyval | expression | 0-2 | A key value pair where key is either: at <i>name</i> where name of an environment or module, of <i>what</i> where what is a symbol or function, to <i>offset</i> search until negative integer ancestors back, go <i>offset</i> jumps to negative integer ancestors back. |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| variant | The environment or nil if not found. |

Remarks

If no argument are provided, scope returns the current environment. If offset is provided, it indicates the number of ancestors to search backward. Kewords **of** and **to** are used together to locate a function or symbol, while **go** is used to pick a specific environment relative to a starting environment.

Example

```
> (function foo {}
  (let ?func (scope) ?symbol (symbol func))
  (so {?inner (scope)}
    (tell user ($ function scope ?func \n))
    (tell user ($ inner scope ?inner \n)))
    (tell user ($ parent of inner scope (scope at ?inner go -1) \n)))
    (tell user ($ location of ?symbol (scope of ?symbol to -10) \n)))))

.: foo

> (foo)

function scope Environment-2
inner scope Environment-3
parent of inner scope Environment-2
location of ?func scope Environment-1
.: told
```

Related

do, function, given, global, invoke, local, modular

seal

Prohibits new records.

Syntax

(seal *prototype*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|-------------------------|
| prototype | literal | 1 | The name of a relation. |

Results

| Data Type | Description |
|-----------|-----------------------|
| literal | Returns sealed |

Remarks

Calling the **new** function on a sealed relation shall fail. Sealed relations must be unsealed before the **new** function shall work again.

Example

```
> (relation Q)
.: Q

> (new Q)
.: Q_1

> (seal Q)
.: sealed

> (new Q)
.: [Failure :Name Permission :Text "Attempt to modify sealed relation Q"]

> (unseal Q)
.: unsealed

> (new Q)
.: Q_2
```

Related

[relation](#), [structure](#), [unseal](#)

secant

Calculates the secant of the number.

Syntax

(secant number geometry metrum)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|---------------------------|
| number | The secant of the number. |

Remarks

None.

Example

```
> (secant 0.785398)
.: 1.41421333

> (secant cir 0.785398 radians)
.: 1.41421333

> (secant cir 75 degrees)
.: 3.863703305

> (secant hyp 10 degrees)
.: 0.98496007966

> (secant hyp 0.0872665 radians)
.: 0.9962043239686199
```

Related

acosecant, asecant, cosecant

seconds

Creates a seconds value or returns seconds since year zero.

Syntax

(seconds *seconds*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--|
| seconds | variant | 0-1 | The number of seconds as an integer or time. |

Results

| Data Type | Description |
|-----------|--|
| second | The argument converted to seconds, or seconds since the year zero. |

Remarks

If no argument is supplied, this function returns the current date and time since the year zero in seconds, otherwise it attempts to construct a second atom from the supplied argument.

Example

```
> (seconds)
.: \@s63698773193

> (seconds 546)
.: \@s546

> (if (< (seconds) eternity)
        (tell user ($ Now is less than the infinite future. \n)))
Now is less than the infinite future.
.: told

> (if (> (seconds) neternity)
        (tell user ($ Now is greater than the infinite past. \n)))
Now is greater than the infinite past.
.: told
```

Related

date, moment, paused, tick

securelink

Creates a secure link.

Syntax

```
(securelink option ...)
(securelink address)
```

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|--|
| address | string | 0-1 | The url address. |
| option | expression | 0+ | A key value pair. One of user <i>credentials</i> -user name and password host <i>host</i> - host computer name (default is localhost) port <i>number</i> – a port number. (default is 2001) path <i>path</i> – a path query <i>query</i> – a query element fragment <i>fragment</i> – a fragment element |

Results

| Data Type | Description |
|-----------|-----------------|
| service | A file resource |

Remarks

Returns a secure link.

Example

```
> (function reply {?target ?request} (tell ?target "goodbye."))
.: reply

> (service (var ?service (securelink "https://localhost:4500/foxyy")) reply)
.: Service-1

> (ask ?service "hello")

.: "goodbye."

> (cancel Service-1) (free ?service) (free Service-1)
.: cancelled freed freed
```

Related

[close](#), [erase](#), [path](#), [pipe](#), [read](#), [socket](#), [write](#)

seek

Repositions a file resource to a new read/write location.

Syntax

(seek *file position*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| file | file | 1 | An file resource. |
| position | variant | 1 | A byte position in the file, or 1 (beginning), or # (end). |

Results

| Data Type | Description |
|-----------|----------------------------|
| file | Returns the file resource. |

Remarks

None.

Example

```
> (set ?file (file name "test.txt"))

.: File-1

> (read ?file #)

.: "Hello"

> (seek ?file 1)

.: File-1

> (write ?file "Goodbye")

.: File-1

> (close ?file)

.: closed
```

Related

[bytes](#), [close](#), [file](#), [read](#), [write](#)

self

Returns the currently executing task resource.

Syntax

(self)

Module

Tasks

Parameters

None.

Results

| Data Type | Description |
|-----------|---|
| task | The task resource for the executing task. |

Remarks

None.

Example

```
> (self)
.: task-0
> (concurrent (tell user ($ My task is (self) \n)))
My task is task-4
.: {task-4}
```

Related

wait, abort, concurrent, await, complete, reclaim, task, tasks

separator

Creates a separator string.

Syntax

(separator kind)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|--|
| kind | literal | 0-1 | One of scheme , volume , folder (default), file , path , port , unhost , uncpath . |

Results

| Data Type | Description |
|-----------|---|
| string | A folder or volume separator character. |

Remarks

None.

Example

```
> (separator)
.: "/"

> (separator path)
.: "/"

> (separator scheme)
.: "//"

> (separator volume)
.: ":/"

> (separator port)
.: ":""

> (separator file)
.: "."

```

Related

[file](#), [folder](#), [path](#), [url](#)

series

Creates a serial iterator.

Syntax

(series iterable)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------------------------------|
| iterable | variant | 1 | A sequence, assortment, or stream,. |

Results

| Data Type | Description |
|-----------|------------------------------|
| variant | returns the last expression. |

Remarks

The iterable must be either a sequence, assortment, or stream..

Example

```
> (function walk {?iterable}
  (ensure {sequence assortment stream} ?iterable)
  (let ?series (series ?iterable))
  (map (infix (so {?e nil ?result {}}
    (reset ?series)
    (while (next ?series)
      (set ?e (this ?series))
      (add ?result (do ($ ?e))))
    ?result)
    ",")
    (given {?x} (tell user ?x)))
  walked)

.: walk

> (walk {a b c})
a, b, c.: walked

> (walk (lexicon a 1 b 2 c 3))
{a 1}, {b 2}, {c 3}, .: walked
```

Related

next, reset, this

service

Creates a service.

Syntax

(service *url handler*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--------------------------------|
| url | url | 1 | A Universal Resource Locator.. |
| handler | function | 1 | An message handler function. |

Results

| Data Type | Description |
|-----------|-------------------|
| Service | The Service task. |

Remarks

This function creates an asynchronous service task that processes messages. The handler function shall receive the sender's url, and an optional message string so that it may return a response.

Example

```
> (let ?url (url scheme tcp port 5001))  
.: true  
  
> (function reply {?sender ?message}  
    (tell ?sender "OK"))  
  
.: reply  
  
> (start (service ?url reply))  
.: Service-3  
  
> (ask ?url "How are you?")  
.: "OK"  
  
> (free (cancel Service-3))  
.: freed
```

Related

agent, ask, cancel, free, tell, start

set

Assigns variables to values..

Syntax

(**set** *manner* *assignment* ...)

assignment ::= *symbol value*

assignment ::= {*symbol* ... } *list*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------------|------------|-----|--|
| manner | literal | 0-1 | The manner in which the assignments are performed either the literal parallel or tandem (default if omitted) |
| assignment | expression | 1+ | A symbol and value pair. |

Results

| Data Type | Description |
|--------------|---------------|
| Truth | Returns true. |

Remarks

If a list of variables is provided for an assignment, then the value must be a list with the corresponding number of elements. The symbol must have previously been defined otherwise a symbol not found failure shall occur. If **parallel** is specified as the manner, the variables are defined in parallel, otherwise the variables are assigned in **tandem** (by default).

Example

```
> (set ?person DrWho)
.: true
> ?person
.: DrWho
> (set parallel ?x 1 ?y 2 ?z 3)
.: true
```

Related

-- before tie , --> after tie, dynamic, global, let, local, modular, only, scope, tie

sever

Creates a new assortments by removing keys.

Syntax

(**sever** *assortment key ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|----------------|
| assortment | assortment | 1 | An assortment. |
| key | variant | 1+ | A key. |

Results

| Data Type | Description |
|------------|---------------------|
| assortment | The new assortment. |

Remarks

Assortments are modified destructively. To delete nondestructively use the **remove** function.

Example

```
> (entries (collection a b c d))
.: {{a a} {b b} {c c} {d d}}
> (entries (sever (lexicon a 1 b 2 c 3 d 4) b d))
.: {{a 1}{c 3}}
```

Related

[cut](#), [rip](#), [remove](#)

shuffle

Creates a reordered sequence.

Syntax

(shuffle *sequence* *seed*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-----------------------|
| sequence | sequence | 1 | A list or string. |
| seed | number | 0-1 | A randomization seed. |

Results

| Data Type | Description |
|-----------|---------------------|
| list | The list reordered. |

Remarks

None..

Example

```
> (global ?Alphabet {a b c d e f g h i j k l m n o p q r s t u v w x y z})  
. : true  
  
> (shuffle ?Alphabet)  
. : {v z e g s y c b t h x o p I q a d r k l n u m j f w}  
  
> (shuffle ?Alphabet 123)  
. : {y g l r x b k u t a j q f c d i s h e m v p n z o w}  
  
> (shuffle ?Alphabet 123)  
. : {y g l r x b k u t a j q f c d i s h e m v p n z o w}  
  
> (shuffle "abcdefghijklmnopqrstuvwxyz" (tick))  
. : "pnitdvjbqyuxkfafazorecmwgslh"
```

Related

[pick](#), [random](#)

sigma

Calculates the sigma of a sequence.

Syntax

(sigma *list*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|---|
| list | list | 1 | A list. |
| of | literal | 0-1 | The literal of . |
| kind | literal | 0-1 | Either population or sample . |

Results

| Data Type | Description |
|-----------|------------------------|
| number | The standard deviation |

Remarks

Returns the amount of variation or dispersion of the list. Low deviation indicates the values are close to the mean while high deviation indicates values are more spread out.

Example

```
> (sigma {1 2 3 4} of population)
.: 1.118033988749895

> (sigma (map {{apple 1}{pear 2}{banana 3}{coconut 4}} (given {?e} (@ ?e 2)))
          of population)

.: 1.118033988749895

> (sigma (range 1000 to 3000000 by 10000) of sample)
.: 866020.5925188307

> (sigma (range 100 to 3000 by 100) of sample)
.: 865.544144839919
```

Related

probability, range, statistics

signal

Signals a condition.

Syntax

(signal *condition* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--------------------------------|
| condition | premise | 1 | The condition to be signalled. |

Results

| Data Type | Description |
|-----------|---|
| premise | A premise indicating the signalled information. |

Remarks

A premise is required. If no arguments are supplied, a generic failure premise is constructed. The premise is thrown to an encompassing **try** form and the learn symbol is set to the premise. If no encompassing **try** form exists, the interpreter is halted with the sign as the error.

Example

```
> (try
  (try
    (signal [Failure :Name User :Text "A big problem"])
    learn ?e
    (tell user ($ Learned ?e \n))
    (tell user ($ Signaling... \n))
    (signal ?e))
  learn ?f
  (tell user ($ Learned ?f again. \n))
  finally
  done)

Learned [Failure :Name User :Text "A big problem"]
Signaling...
Learned [Failure :Name User :Text "A big problem"] again
.: done
```

Related

confirm, confute, ensure, escape, signal, try

significand

Calculates the modified significand as zero, or a number from 1 to 10.

Syntax

(significand *number kind*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|---|
| number | number | 1 | A number. |
| kind | literal | 0-1 | One of normed , scientific (default), or integer . |

Results

| Data Type | Description |
|-----------|------------------------------------|
| number | A number between 1 and 10, or 0.0. |

Remarks

The normed significand represents the number as a fraction (0.nnnn). The scientific significand represents the number as a decimal between 1.0 and 10.0 (n.nnnn). The integer significand represents the number as an integer (nnnn).

Example

```
> (significand 500.645 integer)
.: 500645

> (significand 0)
.: 0.0

> (significand -12345 normed)
.: 0.12345

> (significand -2302.023900 scientific)
.: 2.3020239
```

Related

digit, digits, exponential, fractional

signum

Returns the sign of a number as a number, sigil, or word.

Syntax

(**signum** *number option ...*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|--|
| number | number | 1 | A real number. |
| option | expression | 0+ | One of part <i>which</i> - where which is real (default)or imaginary format <i>output</i> - either number (default) , glyph or word . |

Results

| Data Type | Description |
|-----------|--|
| variant | Returns the literal + if the number is positive or zero, or the literal - otherwise. |

Remarks

By default , this function returns the number 1 if positive, the number 0 if zero, or -1 if negative. If the format is glyph, then If the number is positive this function returns the plus character "+", if zero it returns space character " ", otherwise it returns the minus character "-". When the **format** is word then it returns **positive** or **zero**, or **negative**,

Example

```
> (signum 500)
.: 1
> (signum 0 format number)
.: 0
> (signum -1000+32i part real format word)
.: negative
> (signum 500 format glyph)
.: "+"
```

Related

[digit](#), [digits](#), [exponential](#), [fractional](#), [significand](#)

sine

Calculates the sine of a number.

Syntax

(**sine** number geometry metrum)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|-------------------------|
| number | The sine of the number. |

Remarks

None.

Example

```
> (sine 0.785398)
.: 0.7071066656

> (sine cir 0.785398 radians)
.: 0.7071066656

> (sine cir 45 degrees)
.: 0.7071067811

> (sine hyp 1.5708 radians)
.: 2.30130811905

> (sine hyp 90 degrees)
.: 2.30129890230
```

Related

asine, cosine

so

Creates a lexical scope.

Syntax

(**so** *bindings expression ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| bindings | list | 0+ | A list of symbol value pairs or taxon symbol value triples. |
| expression | variant | 0+ | An expression. |

Results

| Data Type | Description |
|-----------|------------------------------|
| variant | Returns the last expression. |

Remarks

Any new variables declared shall shadow any prior variables with the same name. The **so** special form is a shortcut for (do (let ?s1 v1 ...) ...).

Example

```
> (so {?file (file name "myfile.out") }
      (seek ?file #)
      (for ?s in ?message (write ?file ($ ?s)))
      (close ?file))

.: closed

> (so {long ?n 1'000'000'000
      big ?b 1}
      (+ ?n ?b))

.: \#bd1000000001
```

Related

do, let, scope

socket

Creates a TCP socket.

Syntax

(socket *option ...*)

odule

IO

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|---|
| option | expression | 0+ | A key value pair. One of scheme <i>scheme</i> - either tcp (default) or udp user <i>username</i> - user name password <i>password</i> - password host <i>host</i> - host computer name (default is localhost) port <i>number</i> – a port number. (default is 2001) address <i>url</i> - the uniform resource locator as string |

Results

| Data Type | Description |
|-------------|-----------------|
| io resource | A file resource |

Remarks

Opens a port for reading or writing and returns a socket.

Example

```
> (function reply {?target ?request} (tell ?target "goodbye."))
.: reply

> (service (var ?socket (socket "http://localhost:4500/foxyy")) reply)
.: Service-1

> (ask ?socket "hello")
.: "goodbye."

> (cancel Service-1) (free ?socket) (free Service-1)
.: cancelled freed freed
```

Related

close, erase, path, pipe, read, service, write

some

True if the test is true for any element in the sequence.

Syntax

(some symbol ... binding sequence expression ... test)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| symbol | symbol | 1+ | An iteration symbol |
| binding | literal | 1 | The literal in , per , or over . |
| sequence | sequence | 1 | A list or string |
| expression | expression | 0+ | An expression. |
| test | truth | 1 | A truth expression |

Results

| Data Type | Description |
|-----------|---|
| truth | True if any element satisfies the predicate |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to sub-elements of each subsequence. If the binding **over** is specified, each symbol is bound to overlapping elements of the sequence. If there are insufficient elements to bind to the variables then the function fails. The expressions and test are evaluated. If the sequence is empty the function fails.

Example

```
> (some ?x in {a b c d} (= ?x c))  
.: true  
  
> (some ?m ?n per {{2 1} {3 4}} (< ?m ?n))  
.: true  
  
> (some ?m ?n over {4 3 2 7} (divisible-p ?m ?n))  
.: false
```

Related

every, few, nevery, none, most

sort

Creates a sorted sequence.

Syntax

(sort *sequence ordering option ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| sequence | sequence | 1 | The string or list to sort |
| ordering | variant | 1 | <p>an ordering is either:</p> <p>a direction, i.e. < (ascending) or > (descending) for a sequence of atoms, or</p> <p>an ordered list of slot direction pairs (e.g. , { slot dir slot dir ...}) if the elements are premises or tuples, or</p> <p>an ordered list of position direction pairs (e.g., {position dir position dir ...}) if the elements are sequences.</p> <p>an ordered list of key direction pairs (e.g., {key dir key dir ...}) if the elements are assortments.</p> |
| option | expression | 0+ | A key value pair containing any of the following keys: comparer - a comparison function (uses the compare function if this key value pair is not supplied). transform - function to retrieve a value given an item key – a alist key (if sequence is a alist) index – a list index (if sequence is a list of lists) minval – a minimum value to consider in card sort maxval – a maximum value to consider in card sort limit - a number of items to return after sorting algorithm - value one of { tree , card } default is tree |

Results

| Data Type | Description |
|-----------|-----------------|
| list | The sorted list |

Remarks

The typical ordering is < for ascending and > for descending.

Example

```
> (sort {1 2 8 9 3 6 2} <)
.: {1 2 2 3 6 8 9}

> (sort {1 2 8 9 3 6 2} >)
.: {9 8 6 3 2 2 1}

> (Relation N :Value 0)
.: N

> (for ?value in (range 1 to 10) (new N :Value (random 1 to 100)))
.: 55

> (sort (with N :Value as ?v list ?v) <)
.: {4 27 38 41 55 68 83 94 97}

> (sort {{A 1}{C 3}{F 6}{B 2}{D 4}{E 5}} {2 >})
.: {{F 6}{E 5}{D 4}{C 3}{B 2}{A 1}}

> (sort {{Letter A Number 1}{Letter C Number 3}{Letter B Number 2}}
       {Letter >})
.: {{Letter C Number 3}{Letter B Number 2}{Letter A Number 1}}
```

Related

align, apply, best, count, gather, map

split

Creates a list of subsequences divided at a delimiter.

Syntax

(split sequence delimiter)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---|
| sequence | sequence | 1 | A list or a string. |
| delimiters | sequence | 0-1 | A sequence of delimiters.(Default is space character) |

Results

| Data Type | Description |
|-----------|-------------------------|
| list | A list of subsequences. |

Remarks

Returns a list of sequences split at a delimiter. The delimiter is not included in the result.

Example

```
(split "the quick brown fox")
.: {"the" "quick" "brown" "fox"}

(split {the quick brown fox jumped over the lazy dogs} {the})
.: {{quick brown fox jumped over} {lazy dogs}}
```

Related

\$concatenate , capitalize, char, infix, lexemes, lowercase, ngrams, take, trim, unicode, uppercase

start

Starts an agent service, problem, or task.

Syntax

(**start** *compute arugment ...*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-----------------------------|
| compute | compute | 1 | An agent, service, or task. |
| argument | variant | 0+ | An argument to the compute |

Results

| Data Type | Description |
|-----------|-----------------------|
| compute | Returns the argument. |

Remarks

This function starts an asynchronous task.

Example

```
> (let ?url (url scheme tcp port 5001))

.: true

> (function reply {?sender ?message} (tell ?sender "OK"))

.: reply

> (function job {& ?args} (tell user ($ 1)) (wait 0.5))

.: job

> (agent ?url reply job)

.: Agent-1

> (start Agent-1) (wait 3) (cancel Agent-1) (free Agent-1)
111111.: Agent-1 cancelled freed
```

Related

ask, cancel, listen, pause, perform, start, tell

starts-p

True if two intervals start together.

Syntax

(starts-p *interval₁* *interval₂* *tolerance*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------------------|-----------|-----|--|
| interval ₁ | interval | 1 | An interval |
| interval ₂ | interval | 1 | An interval |
| tolerance | instant | 0-1 | Amount of variation. (Defaults to zero ticks). |

Results

| Data Type | Description |
|-----------|---|
| truth | True if interval ₁ starts with interval ₂ . |

Remarks

None.

Example

```
> (starts-p  \@i{\@s1 \@s2}  \@i{\@s3 \@s4})  
.: false  
  
> (starts-p  \@i{\@s1 \@s5}  \@i{\@s1 \@s8})  
.: true  
  
> (starts-p  \@i{\@s1 \@s9}  \@i{\@s4 \@s6})  
.: false  
  
> (starts-p  \@i{\@s4 \@s6}  \@i{\@s4 \@s9})  
.: true  
  
> (starts-p  \@i{\@s5 \@s9}  \@i{\@s3 \@s9} \@s2)  
.: true
```

Related

before-p, during-p, finishes-p, meets-p, overlaps-p

statistics

Finds the mean, median, sigma, and count of a list of numbers.

Syntax

(statistics *numbers*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--------------------|
| numbers | list | 1 | A list of numbers. |

Results

| Data Type | Description |
|-----------|--|
| list | Returns an association list containing the mean, median, sigma, min, max. and count. |

Remarks

None.

Example

```
> (statistics (array (random 0 to 1000) each (random 0 to 10000)))
.: {mean 5601.26666 median 5701 sigma 3692.856193 min 43 max 9444 count 368}
> (statistics {6 3 1 1 5 1 0 5 5 0 2})
.: {mean 2.636363 median 2 sigma 2.1436047 min 0 max 6 count 11}
```

Related

[probability](#), [survey](#)

step

Loops a symbol over a numeric range to evaluate expressions.

Syntax

(step *symbol from i limit j by k expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|------------|-----|---|
| symbol | symbol | 1 | An index symbol to be incremented |
| from | literal | 1 | from |
| i | number | 1 | The initial value of the symbol |
| limit | literal | 1 | The literal to , above , below or go . (default is to). |
| j | number | 1 | The final value of the symbol |
| by | literal | 0-1 | by |
| k | number | 0-1 | The positive or negative increment for the symbol |
| expression | expression | 0+ | The expression(s) to be evaluated |

Results

| Data Type | Description |
|-----------|---|
| value | The last expression on the last iteration of the loop |

Remarks

If **to** is specified, the *j* value is included in the iteration. If **below** is specified the *j* value is excluded from the iteration. If **above** is specified the next number above the *j* value is included in the iteration. If **go** is specified, then the *j* value specifies the number of iterations.

Example

```
> (step ?n from 1 below 5 (tell user ($$ ?n , (\s))))
1, 2, 3, 4, .: nil

> (step ?i from 1 to 9 by 2 (tell user ($$(( ?i \s)))
1 3 5 7 9 .: told

> (step ?i from 100 go 5 (tell user ($$ ?i \s)))
100 101 102 103 104 .: told
```

Related

[do](#), [for](#), [loop](#), [repeat](#), [until](#), [while](#)

stream

Creates a stream.

Syntax

(stream *streamable* **)**

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--------------------------------|
| streamable | variant | 0+ | A file, pipe, string, or url.. |

Results

| Data Type | Description |
|-----------|--------------------|
| stream | A stream resource. |

Remarks

A stream is a potentially infinite byte array containing serialized expressions. Serializations can be prepended to, appended to, or taken from the stream.

Example

```
> (function unstream {?s}
  (decode (trim (string ?s) cut "|") base16))

.: unstream

> (var ?s (stream "The quick brown fox jumped over the lazy dogs."))
.

|540068006500200071007500690063006b002000620072006f0077006e00200066006f00780020006a
0075006d0070006500640020006f00760065007200200074006800650020006c0061007a00790020006
4006f00670073002e00|


> (unstream ?s)

.: "The quick brown fox jumped over the lazy dogs."
```

Related

append, takeable, take, peek, peep, prepend, stream-p

string

Converts a value to a string.

Syntax

(string *value* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | A value. |

Results

| Data Type | Description |
|-----------|----------------------|
| string | The converted value. |

Remarks

None.

Example

```
> (string {a b c})
.: "{a b c}"

> (string "a b c" (\s) "d e f")
.: "\"a b c\"    \"d e f\""

> (string 500 50 5)
.: "500 50 5"

> (string [A :B :C])
.: "[A :B :C]"

> (string the quick "brown" fox)
.: "the quick \"brown\" fox"

> (string)
.: ""
```

Related

\$ concatenate, \$\$ elide, is

structure

Creates a structure.

Syntax

(structure *name* *inclusion* ... *definition* ... **)**

inclusion ::= **uses** { *structure* ... }

definition ::= *slot*

definition ::= *slot default*

definition ::= *method function*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------------|------------|-----|--|
| <i>name</i> | literal | 1 | The name of the structure. |
| <i>inclusion</i> | expression | 0+ | The literal uses followed by a list of structure names. |
| <i>definition</i> | expression | 0+ | A slot definition or method definition expression |

Results

| Data Type | Description |
|-----------|----------------------------|
| literal | The name of the structure. |

Remarks

Structures are assortment prototypes which represent ephemeral relationships among one or more values. A structure record is called an ephemerons and is comprised of a name, slots with optional default values, and methods. Like the name implies, ephemerons are immediate unpersisted objects. This stands in contrast to thoughts which are persisted in the object store. Both structure and relation definitions are persisted in the object store. Structure ephemerons are created using the **new** function. The **Failure** prototype is an example of a structure. A prototype is a relation or structure.

Example

```
> (structure Fault
  :What
  :When
  :Where
)

.: Fault

> (new Fault :What "An error" :When (moment) :Where "Over Here")

.: [Fault :What "An error" :When \@m20180561212150036 :Where "Over Here"]

> (structure Error
  uses {Fault}
  :Who )

.: Error

> (new Error :Who MyFunction :What "Punted." :When (moment) :Where "Here")

.: [Error :Who MyFunction :What "Punted." :When \@m20180561212489526 :Where "Here"]
```

Related

is, new, nix, relation, seal, structures, unseal

structures

Creates a list of all structures.

Syntax

(structures)

Module

Base

Parameters

None

Results

| Data Type | Description |
|-----------|---------------------------|
| list | A list of all structures. |

Remarks

None

Example

```
> (structure Foo :Bar :Baz)  
.: Foo  
> (structures)  
. : {Failure Foo}
```

Related

[new](#), [nix](#), [old](#), [relation](#), [relations](#), [seal](#), [structure](#), [unseal](#)

sub

Replaces a subsequence within a sequence.

Syntax

(**sub** *sequence entry ...*)

entry ::= start to end replacement

entry ::= start go length replacement

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|--|
| sequence | sequence | 1 | An assortment or sequence. |
| entry | expression | 1+ | A 4-tuple of the start position, either to position or go length, and the replacement sequence |

Results

| Data Type | Description |
|-----------|--------------------------------|
| sequence | Returns the modified sequence. |

Remarks

Each replacement is inserted into the sequence at the indicated positions. .

Example

```
> (sub {the quick red fox jumped} 2 to 4 {cute pink rabbit} )
.: {the cute pink rabbit jumped}

> (sub "abcdefghijkl" 4 go 3 "456")
.: "abc456ghijkl"

> (sub "the quick brown fox jumped" 11 go 5 "green" 17 to 19 "dog")
> "the quick green dog jumped"
.: (sub {the quick brown fox jumped} 2 to 4 {cute rabbit})
.: {the cute rabbit jumped}
```

Related

fix, put, replace, sub

subset

True if all elements in each subset are in the superset.

Syntax

(subset-p *subset superset* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| subset | sequence | 1 | A sequence |
| superset | sequence | 1+ | A sequence |

Results

| Data Type | Description |
|-----------|---|
| truth | True if all elements of the subset are in the superset. |

Remarks

If superset and subset are strings and ordered is true, then a string search is performed.

Example

```
> (subset-p {a b x} {a b c d e f g})  
.: false  
  
> (subset-p {a b d} {a b c d e f g})  
.: true  
  
> (subset-p "ick bro" "the quick brown")  
.: true  
  
> (subset-p {g h i} {a b c d e f g})  
.: false  
  
> (subset-p {e f d} {a b c d e f g})  
.: true
```

Related

[disjoint-p](#), [intersects-p](#), [position](#), [in](#), [pick](#)

substitute

Modifies a sequence replacing subsequences.

Syntax

(substitute *sequence entries* **)**

entry ::= {{*start end replacement*} ... }

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|--|
| sequence | sequence | 1 | An assortment or sequence. |
| entry | expression | 1 | A list containing lists of triples: the start position, end position and replacement subsequence |

Results

| Data Type | Description |
|-----------|--------------------------------|
| sequence | Returns the modified sequence. |

Remarks

Each replacement is inserted into the sequence at the indicated positions..

Example

```
> (substitute {the quick red fox jumped} {{2 4 {cute pink rabbit}}})  
. : {the cute pink rabbit jumped}  
  
> (substitute "abcdefghijklm" {{4 7 "456"}})  
. : "abc456ghijklm"  
  
> (substitute "the quick brown fox jumped" {{11 16 "green"}{17 19 "dog"}})  
. : "the quick green dog jumped"  
  
> (substitute {the quick brown fox jumped} {{2 4 {cute rabbit}}})  
. : {the cute rabbit jumped}
```

Related

fix, put, replace, sub

subsumes-p

True if all elements in each subset are in the superset.

Syntax

(subsumes-p superset subset ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|------------------|
| superset | sequence | 1 | A list or string |
| subset | sequence | 1+ | A list or string |

Results

| Data Type | Description |
|-----------|--|
| truth | True if superset contains all elements of each subset. |

Remarks

None

Example

```
> (subsumes-p {a b c d e f g} {a b x})
.: false

> (subsumes-p {a b c d e f g} {a b x} {a b d})
.: true

> (subsumes-p "the quick brown" "ick bro")
.: true

> (subsumes-p {a b c d e f g} {d e f} {g h i})
.: false

> (subsumes-p {a b c d e f g} {e f d})
.: false
```

Related

disjoint-p, intersects-p, position, in, pick, subset-p

sum

Calculates the arithmetic sum.

Syntax

(sum *value* ...)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------------------|
| value | variant | 1+ | An atom or list of atoms. |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| number | The arithmetic sum of all elements. |

Remarks

Returns the arithmetic sum. If the list or expression is empty, it returns zero. If the only value is a list then all elements within the list are compared, otherwise, all subsequent values are compared.

Example

```
> (sum {1 2 3 4 5})  
. : 15  
  
> (sum {})  
. : 0  
  
> (sum 1 2 3 4 5)  
. : 15  
  
> (sum {1 2 3} 4 {6 3 2})  
. : {11 9 9}
```

Related

[avg](#), [max](#), [median](#), [min](#), [statistics](#), [sigma](#)

summation

Adds an expression over a range of values.

Syntax

(**summation** *symbol* ... *binding* *sequence* *expression* ...)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| symbol | symbol | 1+ | A symbol. |
| binding | literal | 0-1 | The literal in or per . |
| sequence | sequence | 0-1 | A sequence. |
| expression | expression | 1+ | any expression involving variables from the pattern(s) |

Results

| Data Type | Description |
|-----------|---|
| variant | The result of applying the plus function to the expression for the indicated range. |

Remarks

If the binding **in** is specified, each symbol is bound to successive elements of the sequence. If the binding **per** is specified, each element is expected to be a sequence, so that each symbol is bound to subelements of each subsequence. If there are insufficient elements to bind to the variables then the function fails. The last expression shall be evaluated and the plus function shall be applied to both that expression and any prior result. The final value is returned.

Example

```
> (summation ?x in (range 1 to 10) ?x)
.: 55

> (summation ?x in {1 2 3 4} (** ?x 2))
.: 25

> (summation ?n in {1 2 3} (+ ?n 5))
.: 21
```

Related

[fold](#), [maximum](#), [minimum](#), [product](#)

supply

Applies a function to an argument list.

Syntax

(*supply arguments function environment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|--------------------------------------|
| arguments | list | 1 | A list containing actual parameters. |
| function | function | 1 | The name of a function to invoke |
| environment | environment | 0-1 | An environment. |

Results

| Data Type | Description |
|-----------|--|
| variant | The result from applying the function. |

Remarks

Applies the function to the list using the supplied environment, or the current scope if no environment is provided. The function is applied to the entire list and the result is returned.

Example

```
> (supply {1 2 3} +)
.: 6
> (condvier {?x 9} (let ?env (scope)) (so {?x 3} (supply {1 2 ?x} + ?env)))
.: 12
```

Related

apply, call, count, gather, map, reduce

suppose

Performs hypothetico-deductive reasoning.

Syntax

```
(suppose facts action goals by strategies via operators limit quantity  
gate condition within timeframe before timepoint options options)
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------------|-----------|-----|---|
| <i>facts</i> | variant | 1 | a list of expressions, or the literal “known” |
| <i>action</i> | literal | 1 | one of prove , refute , solve , infer , speculate , explain |
| <i>goals</i> | list | 1 | a list of goal expression |
| <i>by</i> | literal | 1 | the literal “by” |
| <i>strategies</i> | list | 1 | a list of user defined search functions. |
| <i>via</i> | literal | 0-1 | the literal “via” |
| <i>operators</i> | list | 0-1 | a list of operators |
| <i>limit</i> | literal | 0-1 | the literal “limit” |
| <i>quantity</i> | number | 0-1 | the number of solutions/proofs to generate (default is 1) |
| <i>gate</i> | literal | 0-1 | one of while or until |
| <i>condition</i> | truth | 0-1 | an expression which follows while or until (default is true) |
| <i>within</i> | literal | 0-1 | the literal “within” |
| <i>timeframe</i> | number | 0-1 | a time duration (default is eternity) |
| <i>before</i> | literal | 0-1 | the literal “within” |
| <i>timepoint</i> | number | 0-1 | an exact time (default is now + 60 seconds) |
| <i>options</i> | literal | 0-1 | the literal “options” |
| <i>options</i> | lexicon | 0-1 | a lexicon or list. |

Results

| Data Type | Description |
|-----------|---|
| variant | Returns a solution or nil . If the action is solve , each solution is a list of operators. If the action is prove , each solution is a list of conclusion-operator pairs. If the action is refute , each solution is a list of conclusion-operator pairs. If the action is infer , a truth result tells if the goals are inferable from the facts If the action is speculate , each solution is a conclusion-operator-probability triple. If the action is appraise , each solution is a list of solution-cost pairs. if the action is explain , each solution is a list of reasons. |

Remarks

Returns solutions (i.e. lists of operators) which transform the facts into the goals, or proofs (lists of operators as inference steps) showing the goals follow from the facts, or a truth indicating the goals are provable from the facts, by creating a monitor task and worker tasks which search for operator chains from facts to goals. The using funciton invokes all the supplied strategies (i.e. search functions) concurrently, canceling them if a condition dictates or a timeframe is passed. One or more result lists (up to the limit) is returned.

1. If the action is **solve**, each solution is a list of operators.
2. If the action is **prove**, each proof is a list of conclusion-operator pairs.
3. If the action is **refute**, each proof is a list of conclusion-operator pairs.
4. If the action is **infer**, a truth result determines if the goals are inferable from the facts.
5. If the action is **speculate**, each proof is a list of conclusion-operator-probability triples.
6. If the action is **explain**, each explanation lists the inferences linking goals to facts.

Each strategy (search function) needs to define keyword parameters which accept arguments from the using call. All the following keyword parameters are optional: **solve**, **prove**, **infer**, **via**, **within**, **options**. The strategy shall need to periodically check its own task identifier (by calling **(cancelled-p (self))**) to determine if the search task has been interrupted by the monitor task.

Example

```
> (structure Pegs
  :Left
  :Temp
  :Right
)

.: Pegs

> (function LeftToTemp {?state} ; a state is a list containing one Pegs structure.
  (let (@ ?state) :Left ?left :Temp ?temp :Right ?right)
  (if (and (more-p ?left) (or (void-p ?temp) (< (@ ?left) (@ ?temp))))
    {(new Pegs :Left (rest ?left) :Temp (& (@ ?left) ?temp) :Right ?right)}))

.: LeftToTemp

> (function LeftToRight {?state}
  (let (@ ?state) :Left ?left :Temp ?temp :Right ?right)
  (if (and (more-p ?left) (or (void-p ?right) (< (@ ?left) (@ ?right))))
    {(new Pegs :Left (rest ?left) :Temp ?temp :Right (& (@ ?left) ?right)}))

.: LeftToRight

> (function TempToRight {?state}
  (let (@ ?state) :Left ?left :Temp ?temp :Right ?right)
  (if (and (more-p ?temp) (or (void-p ?right) (< (@ ?temp) (@ ?right))))
    {(new Pegs :Left ?left :Temp (rest ?temp) :Right (& (@ ?left) ?right)}))

.: TempToRight
```

```

> (function TempToLeft {?state}
  (let (@ ?state) :Left ?left :Temp ?temp :Right ?right)
  (if (and (more-p ?temp) (or (void-p ?left) (< (@ ?temp) (@ ?left))))
    {(new Pegs :Left(& (@ ?temp) ?left) :Temp (rest ?temp) :Right ?right)}))

.: TempToLeft

> (function RightToTemp {?state}
  (let (@ ?state) :Left ?left :Temp ?temp :Right ?right)
  (if (and (more-p ?right) (or (void-p ?temp) (< (@ ?right) (@ ?temp))))
    {(new Pegs :Left ?left :Temp (& (@ ?right) ?temp) :Right (rest ?right))}))

.: RightToTemp

> (function RightToLeft {?state}
  (let (@ ?state) :Left ?left :Temp ?temp :Right ?right)
  (if (and (more-p ?right) (or (void-p ?left) (< (@ ?right) (@ ?left))))
    {(new Pegs :Left (& (@ ?right) ?left) :Temp ?temp :Right (rest ?right))}))

.: RightToLeft

> (relation Node
  :Parent
  :Operator
  :State
  :Children {}
  :Tried {}
  :Task)

.: Node

> (function depthFirstSearch {{facts ?facts}{solve ?goals}{using ?operators}
  & ?rest}

  (let ?root      (new Node :State ?facts)
    ?visited   {}
    ?useless   {}
    ?current   ?root
    ?stack     {}
    ?solution  nil)

  (repeat
    (let ?current :State ?state :Children ?children :Tried ?tried)

    (if (and (void-p ?tried)
              (void-p ?children))
        ;;; Expand current node.
        (for ?operator in ?operators
          (push ?children (new Node :Parent ?current :Operator ?operator
                                :Task (self))))
        (put ?current :Children ?children)

      or (more-p ?children)
        ;;; Expand a child
        (set ?child (pick ?children))
        (enq ?current :Tried each ?child)
        (deq ?current :Children ?child)
        (so {?state ((:Operator ?child) (:State ?current))}
            (put ?child :State ?state)
            (if (null-p ?state)
                (push ?useless ?child)
            or (= ?goals ?state)

```

```

        (set ?solution (map (rest ?stack) :Operator))
        or (in ?visited ?state transform :State)
            (push ?useless ?child)
        else
            (push ?stack ?current)
            (push ?visited ?child)
            (set ?current ?child)))
    else
        ;;; Backtrack
        (if (more-p ?stack)
            (set ?current (pop ?stack))))
until (or ?solution
        (void-p ?stack)
        (cancelled-p (self))))
(with [Node ^ ?n] (= (:Task ?n) (self)) do (old ?p)))
?solution
)

.: depthFirstSearch

> (function breadthFirstSearch{{facts ?facts}{solve ?goals}{using ?operators}
    & ?rest}

    (let ?root      (new Node :State ?facts)
        ?visited   {}
        ?useless   {}
        ?current   ?root
        ?solution  nil)

    (repeat
        (let ?current :State ?state :Children ?children :Tried ?tried)

        (if (and (void-p ?tried)
                  (void-p ?children))
            ;;; Expand current node.
            (for ?operator in ?operators
                (push ?children (new Node :Parent ?current :Operator ?operator)
                      :Task (self))))
            (put ?current :Children ?children)

        or (more-p ?children)
            ;;; Expand a child
            (set ?child (pick ?children))
            (enq ?current :Tried each ?child)
            (deq ?current :Children ?child)
            (so {?state ((:Operator ?child) (:State ?current))}

                (put ?child :State ?state)
                (if (null-p ?state)
                    (push ?useless ?child)
                or (= ?goal ?state)
                    (set ?solution (only {?child}
                        (so {?result {}}
                            (repeat (--> & ?result (:Operator ?child))
                                (set ?child (:Parent ?child))
                                until (null-p ?child))
                            (reverse (but ?result))))))

                or (in ?visited ?state transform :State)
                    (push ?useless ?child)
                else
                    (push ?frontier ?child)))
            else
                ;;; Backtrack

```

```

(if (more-p ?frontier)
    (set ?current (pop ?frontier)))))

until (or ?solution
        (void-p ?frontier)
        (cancelled-p (self)))
(with Node ^ ?n (= (:Task ?n) (self)) do (old ?p))
?solution
)

.: breadthFirstSearch

> (function TowersOfHanoi {%
  % ?facts ?goals}
  (suppose {(new Pegs :Left (@ ?facts) :Temp (@ ?facts 2) :Right (@ ?facts 3))}
  solve {(new Pegs :Left (@ ?goals) :Temp (@ ?goals 2) :Right (@ ?goals 3))}
  by {depthFirstSearch breathFirstSearch}
  via {LeftToTemp LeftToRight TempToLeft TempToRight RightToTemp
  RightToLeft})
)
.: TowersOfHanoi

> (TowersOfHanoi facts {{1 2 3} {} {}} goals {{}}{}{1 2 3}})

.: {{LeftToRight LeftToTemp RightToTemp LeftToLeft TempToLeft TempToRight
LeftToRight}}

> (TowersOfHanoi facts {{3}{1 2}{}} goals {{}}{}{1 2 3}})

.: {{LeftToRight TempToLeft TempToRight LeftToRight}}
> (TowersOfHanoi facts {{1 2} {} {}} goals {{}}{}{1 2}})

.: {{LeftToTemp LeftToRight TempToRight}}

```

Related

wait, abort, concurrent, await, cancelled-p, complete, rule, self, tarry, with

survey

Creates a list of referents or referent-count pairs.

Syntax

(survey *format relation slot*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--|
| format | literal | 1 | One of values , counts . |
| relation | relation | 1 | A relation |
| slot | slot | 1 | The slot of interest |

Results

| Data Type | Description |
|-----------|--|
| list | An association list or list of all values contained in the slot. |

Remarks

Returns a list values or an association list of counts (i.e., observations) for a relation slot

Example

```
> (relation Fruit :Color :Type)
.: Fruit

> (new Fruit :Color red :Type apple)  (new Fruit :Color red :Type cherry)
.: Fruit_1 Fruit_2

> (new Fruit :Color red :Type strawberry)
.: Fruit_3

> (survey values Fruit :Color)
.: {red red red}

> (survey counts Fruit :Type)
.: {{apple 1}{cherry 1}{strawberry 1}}
```

Related

probability, statistics

symbol

Creates a symbol.

Syntax

(**symbol** *name environment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-------------|-----|---|
| name | variant | 0-1 | A string, literal or integer. |
| environment | environment | 0-1 | An environment (defaults to the current scope). |

Results

| Data Type | Description |
|-----------|-------------|
| symbol | A symbol |

Remarks

The current environment is available via **scope**. If no arguments are supplied, then a new symbol is returned for the current context based on the value "v".

Example

```
> (symbol)
.: ?v-1

> (use User)

.: User

> (modular (symbol 500) nil (symbol) 350)

.: true

> (entries (scope (use)))
.: {{?500 nil} {?s-2 350}}
```

Related

<-- before tie, --> after tie, add, assign, assume, is, scope, set, tie, var, variables

symbols

Lists the variables in an environment.

Syntax

(**symbols** *scope option*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|------------|-----|---|
| scope | variant | 0-1 | A environment. or module. |
| option | expression | 0-1 | A key value pair, one of: ancestors truth - Search ancestors. (Default is false.) |

Results

| Data Type | Description |
|-----------|--|
| list | A list of symbols defined in the environment, and parent environments. |

Remarks

If **ancestors** is true, then the symbols from parent environments are gathered recursively.

Example

```
> (let ?x 1 ?y 2 ?z 3) (scope)
.: true {?x ?y ?z}

> (module Mine
    (modular ?foo bar))

.: Mine

> (variables (scope at Mine))

.: {?foo}

> (so {?g 7 ?h 8 ?i 9}
      (so {?d 4 ?e 5 ?f 6}
          (so {?a 1 ?b 2 ?c 3}
              (variables (scope) ancestors true)))))

.: {?a ?b ?c ?d ?e ?f ?g ?h ?i}
```

Related

environment, functions, module, modules, scope, var, symbol

swap

Modifies a sequence by interchanging values.

Syntax

(**swap** *sequence substitution ...*)

substitution ::= prior-position later-position

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------------|------------|-----|--|
| sequence | sequence | 1 | A list or a string. |
| substitution | expression | 1+ | A prior and later position pair. A position may be a number o a list of numbers.. |

Results

| Data Type | Description |
|-----------|-----------------------|
| sequence | The modified sequence |

Remarks

Destructively modifies the sequence. No changes occur if prior and later are the same. To create a new sequence use the **exchange** function.

Example

```
> (swap {A B C} 1 3)
.:{C B A}

> (swap "ABCD" 1 3)
.: 'CBAD'

> (swap {{1 2 3}{4 5 6}} {2 1} {1 3})
.: {{1 2 4}{3 5 7}}
```

Related

@ *element*, append, exchange, fix, push, pop

take

Creates an expression from a readable sequence.

Syntax

(**take** *readable options ...*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|--|
| readable | variant | 1 | An file, stream, or string. |
| options | expression | 0-2 | A key value pair, one of: start position - Place to setart parrings. (Default is 1). limit quantity - Number of parses to make (Default is 1.) |

Results

| Data Type | Description |
|------------|---|
| expression | The parsed expression and the next position to parse in the readable. |

Remarks

Returns an expression containing a list of the extracted expression(s) according to SubThought Premise syntax, and the last parsed position in the readable. If limit is not supplied, the limit becomes one, the next expression is returned. If limit is greater than one then as many expressions are returned as can be extracted up to the limit. If # is provided as the limit, then all remaining expressions are returned. This function returns an empty expression if no expression could be extracted. It's recommended to test the argument prior to parsing using the **takeable** function.

Example

```
> (take (stream ""))
.: 0

> (take "hello world")
.: hello 7

> (take (stream "hello world" #) start 7)
.: world 12

> (take "{the quick brown fox} jumped over the lazy dogs" start 23 limit 3)
.: jumped over the 39
```

Related

append, peek, prepend, skip, stream, stream-p, takeable, whitespace-p

takeable

Calculates the number of expressions that can be read.

Syntax

(takeable *readable options ...*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| readable | variant | 1 | A stream, or file resource or a string containing zero or more expressions. |
| options | expression | 0-2 | A key value pair, one of: start position - Place to setart parrsing. (Default is 1). want quantity - Number of parses to make (Default is #.) |

Results

| Data Type | Description |
|-----------|--|
| number | Calculates the number of actual expression that can be taken.. |

Remarks

If want is greater than one then as many expressions are tested for completeness as can be extracted up to the count. If # is provided, then all remaining expressions are tested for completeness. After testing, the resource shall be unchanged. Returns a number up to the number of desired expressions that can be successfully taken (because they are complete).

Example

```
> (global ?Stream (stream "The quick brown fox {jumped} "))

.: true

> (takeable ?Stream)

.: 4

> (takeable ?Stream #)

.: 4

> (takeable ?Stream 2)

.: 2
```

Related

[skip](#), [append](#), [take](#), [peek](#), [prepend](#), [stream](#), [stream-p](#)

tally

Returns the number of matches in the knowledge base.

Syntax

(tally pattern option)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|---|
| pattern | pattern | 1 | A grounded premise |
| option | expression | 0+ | An option for retrieving the bindings. limit n - the number of results to return. |

Results

| Data Type | Description |
|-----------|----------------------------------|
| list | A list of all symbol value pairs |

Remarks

Returns an anonymous list generator containing lists of symbol value lists.

Example

```
> (relation Poll :Response)
.: Poll

> (step ?i from 1 to 1000
    (new Poll :Response (on (> (random 0 to 1) 0) Yes No)))

.: Poll_1000

> (tally [Poll :Responses Yes])
.: 672
```

Related

[tally](#), [using](#), [with](#)

tangent

Calculates the tangent.

Syntax

(tangent number geometry metrum)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| number | number | 1 | A number |
| geometry | literal | 0-1 | One of: cir (cle), hyp (erbola). Default is circle. |
| metrum | literal | 0-1 | One of: degrees , radians . Default is radians. |

Results

| Data Type | Description |
|-----------|----------------------------|
| number | The tangent of the number. |

Remarks

None.

Example

```
> (tangent 5)
.: -3.38051500625

> (tangent 5 cir radians)
.: -3.38051500625

> (tangent 120 degrees)
.: 1.7320508075

> (tangent 45 hyp degrees)
.: 0.655794202632

> (tangent (pi) hyp radians)
.: 0
```

Related

acotangent, atangent, cotangent

tarry

Allows tasks to complete on their own.

Syntax

(tarry *tasks expression ...*))

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--|
| tasks | variant | 1 | A task or list of task resources. |
| expression | variant | 0+ | An expression to evaluate after the tasks are completed. |

Results

| Data Type | Description |
|-----------|---------------------|
| list | The task resources. |

Remarks

This function allows tasks to complete in a non-blocking manner. It is equivalent to

```
(given {?tasks}
  (& ?tasks
    (concurrent (do (until (every ?t in ?tasks (ready-p ?t)) (idle \@m1))
  ready))))
```

Example

```
> (var ?tasks (tarry (concurrent (repeat 1000000 foo)
                                (repeat 2000000 bar)
                                (repeat 3000000 baz)))))

.: {task-1 task-2 task-3 task-4}

> (map ?tasks ({?t} (await ?t \@s0)))

.: {foo bar baz done}

> (free (tarry (complete
                  (repeat 1000 foo)
                  (repeat 2000 bar)
                  (repeat 3000 baz)))))

.: freed
```

Related

wait, abort, concurrent, await, complete, do, done-p, free, task

task

Creates a list of tasks for deferred evaluation.

Syntax

(task *cell scope expression ...* **)**

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|------------|-------------|-----|---|
| cell | cell | 0-1 | A cell. |
| scope | environment | 0-1 | An environment. A child scope is created if not provided. |
| expression | expression | 1+ | Any expression to be evaluated. |

Results

| Data Type | Description |
|-----------|---|
| list | A list of task resources, one resource for each expression. |

Remarks

The tasks need to be initiated with the **concurrent** or **complete** function.

Example

```
> (task (/ 1000 57) (** 4 20) (** 4 20))
.: {Task-1 Task-2 Task-3}

> (concurrent task-1)
.: {Task-1}

> (await task-1)
.: 17.54385964912281

> (concurrent task-2 task-3)

> (await task-2)
.: 1099511627776

> (await task-3)
.: 1099511627776

> (free {task-1 task-2 task-3})
.: nil
```

Related

wait, abort, concurrent, await, busy-p, cancel, cancelled-p, complete, critical, do, is, ready-p, reclaim , task

tasks

Creates a list of all tasks.

Syntax

(tasks)

Module

Tasks

Parameters

None.

Results

| Data Type | Description |
|-----------|--------------------------------|
| list | Returns a list of task handles |

Remarks

This list does not include the interpreter task. To obtain the interpreter task, use the **self** function.

Example

```
> (task (apply + {1 2 3}))  
.: {Task-1}  
  
> (complete (apply * {1 2 3}))  
.: {Task-2}  
  
> (tasks)  
.: {Task-2 Task-1}  
  
> (free (tasks))  
.: freed
```

Related

[wait](#), [abort](#), [concurrent](#), [await](#), [task](#), [await](#), [complete](#), [reclaim](#), [self](#), [task](#)

taxon

Returns the taxonomic (i.e., data) type of a value.

Syntax

(taxon *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|-----------------------|
| literal | A taxonomic type name |

Remarks

None

Example

```
> (taxon true)
.: truth
> (taxon nil)
.: nothing
> (taxon 12345)
.: number
> (taxon #)
.: literal
```

Related

convert, convertible, taxonomy, identity, is, meron, meronomy, relation, structure

taxon-p

True if the value is of the specific taxonomic type.

Syntax

(taxon-p *value* *taxon* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|----------------------|
| value | variant | 1 | A value. |
| taxon | literal | 1 | The name of a taxon. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| truth | True if the value is the taxon. |

Remarks

None.

Example

```
> (taxon-p 500 literal)
.: false

> (taxon-p \@t300 time)
.: true

> (taxon-p "the quick brown fox" string)
.: true
```

Related

meron, meron-p, meronomy, taxon, taxonomy

taxonomy

Returns a list of taxonomic types for a value.

Syntax

(taxonomy *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A value |

Results

| Data Type | Description |
|-----------|--------------------------|
| list | The taxonomic types used |

Remarks

None

Example

```
> (taxonomy A)
.: {literal atom thing}

> (taxonomy 100)
.: {number atom thing}

> (taxonomy {a b c})
.: {list sequence thing}

> (taxonomy [A :B 1 :C 2])
.: {premise tuple sequence thing}

> (taxonomy \@t522)
.: {tick time atom thing}
```

Related

[taxon](#), [is](#), [id](#), [meron](#), [meronomy](#), [relation](#), [structure](#)

tell

Sends a message to a recipient.

Syntax

(**tell** *who message option ...*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|---|
| who | literal | 1 | A URL web resource or the literal user . |
| message | string | 1 | A message or command.. |
| option | expression | 0+ | Key value pairs timeout <i>instant</i> – delay before terminating request, default <i>value</i> – value to return upon timeout, |

Results

| Data Type | Description |
|-----------|--|
| literal | Returns told if successful. If a default is provided then the default is returned in case of a timeout. |

Remarks

If **user** is specified for who, then the output is displayed to the console.

Example

```
> (tell user "Hello World!\n")

Hello World
.: told

> (function reply {?url ?message}
  (tell ?url ($ received ?message)))

.: reply

> (service (url scheme tcp port 5950) reply)

.: Service-1

> (ask (url scheme tcp port 5950) "secret")

.: "received secret"

> (free Service-1)

.: freed

> (global ?Data (data source (get Settings-1 "/db/dsn")))
```

```
.: true
> ?Data

.: Data-1
> (open ?Data)
.: Data-1
> (tell ?Data "update product set price = price * 1.1025;")
.: told
> (close ?Data)
.: closed
```

Related

ask, listen,free

there

True if a file, folder, or url actually exists.

Syntax

(there *url* **)**

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|-------------|
| url | url | 1 | A url. |

Results

| Data Type | Description |
|-----------|--------------------------|
| truth | True if the url exists.. |

Remarks

This function checks whether or not the url exists.

Example

```
> (let ?url (ul path "quick.txt"))
  .: true

> (if (there ?url)
    (let ?file (file url ?url seek 1 mode write erase yes))
    (write ?file "The quick brown fox")
    (close ?file))

  .: closed
```

Related

[absent](#), [file](#), [folder](#), [url](#)

thing-p

True if a value is a thing.

Syntax

(thing-p *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1+ | A value |

Results

| Data Type | Description |
|-----------|--------------------------------|
| truth | True if the values is a thing. |

Remarks

None.

Example

```
> (thing-p yes)
.: true
> (thing-p nil)
.: false
> (thing-p (nothing))
.: false
> (thing-p 3.1415926)
.: true
```

Related

[nil-p](#), [null-p](#) , [void-p](#)

think

Creates a match-infer loop task to solve problems.

Syntax

(think problem domain)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|---|
| problem | literal | 0-1 | A problem identifier Default is all if not supplied. |
| domain | literal | 0-1 | A start domain for the problem. |

Results

| Data Type | Description |
|------------|---|
| expression | Returns a list of task ids for each match infer loop created or void if none created. |

Remarks

If no problem is provided, the (**problems**) function is called and a task is created for each problem that has an **open** status. The value **void** is returned if there are no problems to solve.

Example

```
> (domain FixIt
    (relation Car :Age :Condition)
    (rule
      if
        [Car :Age /= old :Condition = good]
      do
        (tell user ($ ($$ \n that) is possible. \n)
        then done)))
.

.: FixIt

> (problem domain FixIt facts [Car :Age new :Condition good])
.: Problem_1

> (complete (think))
.: {Problem_1}

that is possible.
```

Related

cancel, domain, domains, infer, match, problem, problems, rule, rules

this

The current element of a series.

Syntax

(this *series*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| series | series | 1 | A series. |

Results

| Data Type | Description |
|-----------|--|
| variant | Returns the current value of the series. |

Remarks

The **next** function is used to advance a series to the first through last elements of an iterable thing. If **this** is called immediately after the creation of the iterable, or immediately after a **reset** of the iterable, or before the first **next** is called, then an out of range failure shall occur since the series is pointing to an unknown item before the first element. If **this** is called after next returns false, an out of range failure shall also occur because the series is pointing after the last element.

Example

```
> (function walk {?iterable}
  (confirm (?iterable-p ?iterable) since
          "Argument to walk must be iterable")
  (let ?series (series ?iterable))
  (while (next ?series)
    (tell user ($$ (this ?series) ,)))
  walked)

.: walk

> (walk {a b c})
a, b, c.: walked

> (walk (lexicon a 1 b 2 c 3))
{a 1}, {b 2}, {c 3}, .: walked
```

Related

[next](#), [reset](#), [series](#)

thought

Finds an extant thought.

Syntax

(**thought** *relation id*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--------------------|
| relation | literal | 1 | A relation |
| id | number | 1 | An instance number |

Results

| Data Type | Description |
|-----------|-------------|
| thought | A thought. |

Remarks

Returns an existing thought, otherwise signals a failure If the SubThought does not exist.

Example

```
> (relation E :Value)
.: E

> (new E :Value 100)
.: E_1

> (new E :Value 200)
.: E_2

> (thought E 1)
.: E_1

> (thought E 2)
.: E_2

> (thought E 3)
.: [Failure :Name NotFound :Text "Thought E_3 was referenced but not found."]
```

Related

^ SubThought *id*, *id*, is, premise

thunk

Creates a thunk in a module.

Syntax

(thunk expression ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|------------------------------|
| expression | variant | 0+ | Expressions to be evaluated. |

Results

| Data Type | Description |
|-----------|---|
| thunk | Returns the thunk containing the expressions. |

Remarks

A thunk is a parameterless anonymous function. A thunk is equivalent to **lambda** expression without parameters in the Lisp programming language, and can be used in most places a function name is required. Anonyms cannot be recursive, since they are unnamed. Continuations may be taken within an anonym. Neither do anonyms use tags that can be reference by a **return**. Since no parameters are specified, each expression is executed in succession and the value of the last expression is returned. The thunk is ephemeral because function definitions are not maintained for them.

Example

```
> (thunk (tell user "Hello, world."))
.: (thunk (tell user "Hello, world."))
> (invoke (thunk (tell user "Hello, world.")))
Hello, world.
.: told
```

Related

[wait](#), [abort](#), [concurrent](#), [await](#), [complete](#), [done-p](#), [reclaim](#), [task](#), [use](#)

thunks

Returns a list of thunks defined in a module.

Syntax

(thunks *module*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|---|
| module | literal | 0-1 | A module. If not provided defaults to the current module. |

Results

| Data Type | Description |
|-------------|--|
| list or nil | A list of functions defined in the module. |

Remarks

The module must be a literal in the list obtained by calling the (**modules**) function. If the module is not provided this function uses the current module (obtained via the (**use**) function).

Example

```
> (module Mine      (thunk bar))  
.: Mine  
  
> (thunks Mine)  
.: {thunk-1}  
  
> (use User)  
.: User  
  
> (thunk "hello world")  
.: thunk-2  
  
> (thunks)  
.: {thunk-2}
```

Related

function, module, modules, rule, rules, variables

tick

Creates a tick or returns nanoseconds since year zero.

Syntax

(`tick` *nanoseconds*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|-----------|-----|--------------------------|
| nanoseconds | number | 0-1 | A number of nanoseconds. |

Results

| Data Type | Description |
|-----------|----------------------------------|
| tick | nanoseconds since the year zero. |

Remarks

If no argument is supplied, this function returns the current date and time in nanoseconds, otherwise it attempts to construct a tick from the supplied argument.

Example

```
> (tick)
.: \@t63552721198807000

> (tick 5465464)
.: \@t5465464
```

Related

[date](#), [moment](#), [idle](#)

tie

Assigns variables to values..

Syntax

(tie *manner* *assignment* ...)

assignment ::= *symbol value*

assignment ::= {*symbol* ... } *list*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| manner | literal | 0-1 | The manner in which the assignments are performed either the literal parallel or tandem (default if omitted) |
| assignment | expression | 1+ | A symbol and value pair. |

Results

| Data Type | Description |
|-----------|----------------------------------|
| variant | Returns the last assigned value. |

Remarks

If a list of variables is provided for an assignment, then the value must be a list with the corresponding number of elements. The symbol must have previously been defined otherwise a symbol not found failure shall occur. Returns the value of the last assignment. If **parallel** is specified as the manner, the variables are defined in parallel, otherwise the variables are assigned in **tandem** (by default).

Example

```
> (tie ?person DrWho)
.: DrWho
> ?person
.: DrWho
> (tie parallel {?x ?y ?z} {1 2 3})
.: 3
```

Related

-- before tie, --> after tie, assume, constant, dynamic, given, global, local, set

time

Performs time operations.

Syntax

(time unit operation value ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--|
| unit | literal | 1 | one of { year , month , week , weekday , days , day , hour , minute , second , millisec , jiffy , moment , microsec , nanosec , tick , zone , zmin }. |
| operation | literal | 1 | One of {+ (addition), - (difference), of , in , part , ytd } |
| value | time | 1+ | A time value |

Results

| Data Type | Description |
|-----------|---|
| number | The operation performed on the time value(s). |

Remarks

Returns the value of the operation in the specified units. If **+** is specified as the operation, the times are added together and the total is returned in the unit. If **-** is specified as the operation, the difference of the times are returned in the specified unit. If **of** is specified for the operation, then the specified unit is extracted for the specified time value. If **in** is specified for the operation, then the total time value is converted to the units. If **part** is specified for the operation, and if the time is converted to a moment segment (day part takes a number between 1 and 366, returning a number between 1 and 999; hour part takes a number between 0 and 23, returning a number between 0 and 99; minute part takes a number between 0 and 59, returning a number between 0 and 99; seconds part takes a number between 0 and 59, returning a number between 0 and 99). If **ytd** is specified then the units are computed for the year to date for the date specified as the time. Otherwise the part of the time value is returned.

(imExample

```
> (time seconds - \@s5 \@s10)
.: -5

> (time hour part 18)
.: 75

> (time seconds in \@s26226)
.: 26226

> (time day ytd 2014-04-09T14:24:19.488Z)
.: 9

> (time month of (tick))
.: 12

> (time month of (moment))
.: 12

> (time month of (date))
.: 12

> (@ {mon tue wed thu fri sat sun} (time weekday part (date 2019 8 31)))
.: sat
```

Related

[date](#), [epoch](#), [interval](#), [is](#), [moment](#), [second](#), [tick](#)

top

Creates a subsequence of the first elements.

Syntax

(top sequence count)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | sequence | 1 | The sequence to search |
| count | number | 0-1 | The number of elements.(Default is 1.) |

Results

| Data Type | Description |
|-----------|--|
| sequence | A sub sequence containing the desired quantity of elements |

Remarks

None.

Example

```
> (top {a b c d e f g} 3)
.: {a b c}

> (top {a b c d e f g})
.: {a}
```

Related

@ *element*, but, last, rest

transfer

Transfers a value from one sequence to another.

Syntax

(transfer *source destination option ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------------|------------|-----|---|
| source | symbol | 1 | A symbol bound to a sequence |
| destination | symbol | 1 | A symbol bound to a sequence |
| option | expression | 0+ | A key value pair where the key is from position - The position in the source (defaults to 1) to position - The position in the destination (defaults to 1) unique prohibited (Boolean) - indicates if duplicate values are allowed. (Default is false.) |

Results

| Data Type | Description |
|-----------|---|
| value | Returns the value that was transferred. |

Remarks

Modifies two sequences by removing a value from the source and adding a value to the destination.

Example

```
> (let ?x  {a b c d e f} ?y {})

.: true

> (transfer ?x ?y)

.: a

> (transfer ?x ?y unique true from 2 to 2)

.: c

> ?x ?y

.: {b d e f} {a c}
```

Related

but, exchange, last, pop, push, rest, reverse, swap, top

traverse

Loops through the coordinates of a sequence.

Syntax

(traverse symbol binding sequence limit dimensions expression ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| symbol | symbol | 1 | A symbol holding the index or coordinates of the current element of the sequence. |
| binding | literal | 1 | One of the following: into - varies the inner (rightmost) coordinates first across - varies the outer (leftmost) coordinates first. |
| sequence | sequence | 1 | A possibly nested sequence. |
| limit | literal | 1 | the keyword literal limit |
| dimensions | list | 1 | if the keyword limit is supplied, then this is a list of indicies indicating the maximum dimensions that shall be traversed. |
| expression | expression | 0+ | An expression to be evaluated. |

Results

| Data Type | Description |
|-----------|--|
| variant | This form returns the last expression. |

Remarks

This function returns last expression of the last iteration. The function assumes that the entire sequence has a uniform number of elements in each dimension. On each iteration the symbol is updated with the coordinates of the current element of the sequence. The **into** binding provides depth first traversal, while the **across** binding provides breadth first traversal.

Example

```
> (traverse ?i across {{a b}{c d}{e f}{g h}{i j}} limit {5 2} (tell user ($$ ?i \s)))  
{1 1} {2 1} {3 1} {4 1} {5 1} {1 2} {2 2} {3 2} {4 2} {5 2} .: told  
> (traverse ?i into {{a b}{c d}{e f}{g h}{i j}} limit {5 2} (tell user ($$ ?i \s)))  
{1 1} {1 2} {2 1} {2 2} {3 1} {3 2} {4 1} {4 2} {5 1} {5 2} .: told
```

Related

add, includes, cut, for, item, items, keys, reclaim, values

trim

Creates a sequence without leading and trailing delimiters.

Syntax

(trim *sequence option ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|------------|-----|---|
| sequence | sequence | 1 | The target sequence |
| option | expression | 0+ | A key value pair: cut delimiters - a string or list containing the delimiters. on location - the literal left , right , or both (default). |

Results

| Data Type | Description |
|-----------|-----------------------|
| sequence | The modified sequence |

Remarks

Removes delimiter characters from the front or rear of a sequence. If the location is **left**, then only the left portion is trimmed. If the location is **right**, then only the right portion of the string is trimmed. If the location is **both** or if the location is omitted, then both the left and right portion of the sequence is trimmed.

Example

```
> (trim " the quick brown fox   ")
.: "the quick brown fox"

> (trim " the_quick_brown_fox __" cut "__")
.: " the_quick_brown_fox   "

> (trim " the quick brown fox   " on left)
.: "the quick brown fox   "

> (trim {the quick brown fox} cut {the fox} on both)
.: {quick brown}
```

Related

[split](#)

truncate

Rounds a number towards zero.

Syntax

(truncate *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number. |

Results

| Data Type | Description |
|-----------|-------------|
| number | An integer |

Remarks

None.

Example

```
> (truncate 10.5)
.: 10
> (truncate 10.4)
.: 10
> (truncate 10.49)
.: 10
> (truncate -10.5)
.: -10
> (truncate -10.4)
.: -10
> (truncate -10.49)
.: -10
```

Related

%, ceiling, floor, round,

try

Evaluates expressions and handles signs.

Syntax

(try expression ... learn symbol recovery ... finally cleanup ...)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------------|------------|-----|---|
| expression | expression | 1+ | Expressions to do some processing |
| learn | literal | 1 | The literal learn |
| symbol | symbol | 0-1 | A symbol |
| recovery | expression | 0+ | Expressions for failure recovery and learning |
| finally | literal | 0-1 | The literal finally |
| cleanup | expression | 0+ | Expressions to do some clean up |

Results

| Data Type | Description |
|-----------|---|
| value | Returns last call in normal operation, or last failure call if a failure was thrown |

Remarks

The **try** special form performs signal handling. The **signal** function shall return to the **learn** tag and bind the thrown premise to the symbol.

Example

```
> (function main {& ?args}
  (tell user ($ \n Attempting dangerous operation... \n))
  (try
    (dangerousOperation)
    (tell user ($ \n Goodbye \n))
    learn ?condition
    (case (:Name ?condition )
      of ResourceFailure
        (tell user ($ Resource failure occurred \n))
      else
        (tell user ($ \n (:Text ?condition ) \n)))
    (tell user ($ \n Goodbye \n))
    (signal ?condition ) ; rethrow failure
  )
)
.: main
)
```

```

> (main)

Attempting dangerous operation...

The function dangerousOperation was not found.

Goodbye
.: [Failure :Name User :Text "The function dangerousOperation was not found."]

> (so {?file nil}
  (try
    (set ?file (file name "quick.txt" seek 1 mode read type text))
    (tell user ($ (read ?file) \n))
    learn ?e
    (signal ?e)
  finally
    (close ?file)))

The quick brown fox jumped over the lazy dogs.
.: told

> ; implement a REPL.

  (forever
    (tell user ($ "\n=>" (try (eval (ask user "\n*")) learn ?failure)))))

* HELLO WORLD

=> HELLO WORLD

*

```

Related

can, confirm, confute, did, do, ensure, escape, fail, may, signal

tuple

Creates a tuple.

Syntax

(tuple *value ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 0+ | Any value |

Results

| Data Type | Description |
|-----------|--|
| tuple | Returns a tuple containing the values. |

Remarks

A tuple can also be created by simply enclosing the elements with square brackets, [].

Example

```
> (tuple a b c {d})  
. : [a b c {d}]  
  
> (tuple "a b c")  
. : ["a b c"]  
  
> (tuple 500)  
. : [500]  
  
> (tuple)  
. : []  
  
> [a b c {d}]  
. : [a b c {d}]  
  
> ["a b c"]  
. : ["a b c"]
```

Related

&, inside, list-p, sequence-p

uuid

Creates a unique universal identifier.

Syntax

(uid)

Module

Base

Parameters

None

Results

| Data Type | Description |
|-----------|--------------------------|
| uid | A new unique identifier. |

Remarks

None.

Example

```
> (uid)
.: 90A836B0-63E8-4049-B411-908A6B30F69F
> (uid)
.: CE79CB00-F400-429A-BFC3-8F9801B7B017
```

Related

id, unique

unbound

Creates a list of free variables in a function.

Syntax

(**unbound** *function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| function | function | 1 | A function |

Results

| Data Type | Description |
|-----------|--|
| list | A list of variables not declared in the parameter list (aka free variables). |

Remarks

None.

Example

```
> (function foo {?x ?y + ?z}
  (bar ?x ?y)
  (baz ?z ?a))

.: foo

> (bound foo)

.: {?x ?y ?z}

> (unbound foo)

.: {?a}

> (bound (' (given {?s} (@ ?s ?p)))))

.: {?s}

> (unbound (' (given {?s} (@ ?s ?p)))))

.: {?p}
```

Related

[bound](#)

unbound-p

True if a symbol is unbound.

Syntax

(**unbound-p** *symbol*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| symbol | symbol | 1 | A symbol. |

Results

| Data Type | Description |
|-----------|--------------------------------|
| bool | True if the symbol is unbound. |

Remarks

None.

Example

```
> (let ?a 1 ?b 2)
.: true
> (unbound-p ?a)
.: false
> (unbound-p ?b)
.: false
> (unbound-p ?c)
.: true
```

Related

[unbound-p](#)

unicode

The Unicode of a string's first position.

Syntax

(unicode *string*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--------------|
| string | string | 1 | input string |

Results

| Data Type | Description |
|-----------|-----------------------------------|
| number | The Unicode of the first position |

Remarks

None

Example

```
(unicode "A")
.: 65
(Unicode "apple")
.: 97
```

Related

char, format, lexemes, lowercase, ngrams, split, trim, uppercase

union

Concatenates sequences removing duplicates.

Syntax

(union *sequence sequence ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-----------------|
| sequence | sequence | 2+ | input sequences |

Results

| Data Type | Description |
|-----------|-----------------------------|
| sequence | A single de-duped sequence. |

Remarks

None

Example

```
> (union {a b c} {b c d} {d e f})  
. : {a b c d e f}  
> (union "abc" "bcd" "def")  
. : "abcdef"
```

Related

[difference](#), [disjoint](#), [distinct](#), [intersection](#)

unique

Creates a unique literal based on a prefix.

Syntax

(unique *prefix*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--|
| prefix | variant | 0-1 | A literal or string prefix for the id. |

Results

| Data Type | Description |
|-----------|---------------|
| literal | A new literal |

Remarks

If prefix is not provided the default prefix "G" is used. Hyphens and underscores are removed from the prefix. This function increments a counter to generate a new literal.

Example

```
(unique)

.: G1

(unique)

.: G2

(unique "hello")

.: hello1

(unique My-Object)

.: MyObject1

(unique "my_Object_")

.: MyObject2

(unique "my-object-")

.: MyObject3
```

Related

[Id, uid](#)

unless

Branched conditional evaluation.

Syntax

(unless *condition* *false-case* *true-case* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|---------------------------|
| condition | truth | 1 | A truth expression |
| false-case | variant | 1 | An expression to evaluate |
| true-case | variant | 1 | An expression to evaluate |

Results

| Data Type | Description |
|-----------|--|
| variant | The last value of the executed expression. |

Remarks

Evaluates the condition and either the true case or the false case.

Example

```
> (unless (> 101 100)  (tell user "no")  (tell user "yes"))
yes .: told

> (unless (< 101 100)  (tell user "no")  (tell user "yes"))
no .: told
```

Related

and, case, if, not, on, or

unlike

True if a pattern does not match a sequence.

Syntax

(unlike *sequence* *pattern*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|---|
| sequence | sequence | 1 | A string or list of strings to be compared. |
| pattern | string | 1 | A regex pattern. |

Results

| Data Type | Description |
|-----------|---|
| variant | If sequence is a string, then a truth is returned. If sequence is a list, then a list of non-matches is returned |

Remarks

If the candidate argument is a list, then this function creates a new list containing only those elements not matching the pattern. If the candidate is a string, then this function returns true if the pattern does not match the candidate, and false otherwise. Standard RegEx matching is used.

Example

```
> (unlike "abc" "bc")
.: false

> (unlike "abc" "xy")
.: true

> (unlike {"abc" "bcd" "def" "ghi"} "bc")
.: {"def" "ghi"}
```

Related

like

unqualified

The function literal without the module prefix.

Syntax

(unqualified *function*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|--------------------|
| function | function | 1 | A function literal |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| literal | The function literal without a module |

Remarks

None

Example

```
> (unqualified Math.sum)  
. : sum
```

Related

[qualified](#), [qualifiers](#)

unseal

Permits new records for a prototype.

Syntax

(**unseal** *prototype*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|--------------------------|
| prototype | literal | 1 | The name of a prototype. |

Results

| Data Type | Description |
|-----------|-------------------------|
| literal | Returns unsealed |

Remarks

Calling the **new** function on a sealed relation shall fail. Sealed relations must be unsealed before the **new** function shall work again.

Example

```
> (relation Q)
.: Q

> (new Q)
.: Q_1

> (seal Q)
.: sealed

> (new Q)
.: [Failure :Name Permission :Text "Attempt to modify sealed relation Q"]

> (unseal Q)
.: unsealed

> (new Q)
.: Q_2
```

Related

[new](#), [relation](#), [seal](#), [structure](#)

unsigned

Converts a value to an unsigned number.

Syntax

(unsigned *value* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| value | variant | 1 | A number value or the literal minimum or maximum |

Results

| Data Type | Description |
|-----------|--------------------|
| integer | An integer number. |

Remarks

If the value is **minimum** (or **maximum**) the minimum (or maximum) unsigned number is returned; otherwise, if the value cannot be converted to an unsigned number, the function fails. The fractional part of the number shall be truncated.

Example

```
> (unsigned {a b c})  
.: [Failure :Name ArgumentValue :Text "{a b c} is not a number."]  
  
> (unsigned "a b c")  
.: [Failure :Name ArgumentValue :Text "'a b c' is not a number."]  
  
> (unsigned 500.3)  
.: 500u  
  
> (unsigned "23.4")  
.: 23u  
  
> (unsigned (* 2 2))  
.: 4u
```

Related

ceiling, complex, floor, is, long, rational, real, round, truncate

until

Loops expressions until a condition is true.

Syntax

(until *condition expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|-------------------------------|
| condition | truth | 1 | A truth expression |
| expression | expression | 0+ | An expression to be evaluated |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| value | The last value of the last iteration |

Remarks

None

Example

```
> (modular ?i 0)
.: true
> (until (>= ?i 10) (idle \@s1) (--> ++ ?i))
.: 10
```

Related

for, iterate, loop, repeat, step, while

unwrap

Permits modifications to a module.

Syntax

(unwrap *module*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| module | literal | 1 | A module |

Results

| Data Type | Description |
|-----------|--------------------------|
| literal | Returns unwrapped |

Remarks

This function has no effect on already unwrapped modules.

Example

```
> (module Math
    (function sum ?args (apply + ?args)))

.: Math

> (wrap Math)

.: wrapped

> (extend Math
    (function prod ?args (apply * ?args)))

.: [Failure :Name Permission :Text "Can't modify a wrapped module"]

> (unwrap Math)

.: unwrapped

> (extend Math
    (function prod ?args (apply * ?args)))

.: Math
```

Related

[extend](#), [module](#), [wrap](#)

uppercase

Converts a string to uppercase.

Syntax

(uppercase *string*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|--------------------|
| string | string | 1 | A value to convert |

Results

| Data Type | Description |
|-----------|----------------------|
| string | The converted string |

Remarks

None

Example

```
> (uppercase "the quick brown fox")
.: "THE QUICK BROWN FOX"
```

Related

capitalize, lexemes, lowercase

uppercase-p

True if a string's first position is upper case.

Syntax

(uppercase-p *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|---|
| truth | Returns true if the value is a string and its first position is upper case. |

Remarks

None

Example

```
> (uppercase-p "the quick brown fox")
.: false
> (uppercase-p "This and that")
.: true
```

Related

letter-p, lowercase-p

url

Creates a URL resource.

Syntax

(**url** *option ...*)

(**url** *address*)

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|---------|------------|-----|---|
| address | string | 0-1 | The url address. |
| option | expression | 0+ | A key value pair. One of scheme <i>scheme</i> - the url scheme (e.g., file, http, https) user <i>user</i> - the name of the user password <i>password</i> - the password of the user host <i>host</i> - hostname or the literal ip for this ip address port <i>port</i> - port number or the literal new path <i>path</i> - file path query <i>query</i> - query key value pairs. fragment <i>fragment</i> - section specification.. address <i>url</i> - A full url address. |

Results

| Data Type | Description |
|-----------|-----------------|
| url | A url resource. |

Remarks

If no arguments are provided, then this function returns the universal resource locator for the SubThought Premise interpreter running on the current machine. If **new** is specified as the port, an unused port number is utilized.

Example

```
> (url scheme https host "www.youtube.com" path "watch" query "v=kqicbyONxO8")
.: Url-1
> (address Url-1)
.: "https://www.youtube.com/watch?v=kqicbyONxO8"
```

Related

address, ask, is, listen, tell

use

Creates a scope for knowledge based operations.

Syntax

(use *knowledge expression ...* **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--|
| knowledge | variant | 1 | A knowledge base or url to a knowledge base. |
| expression | variant | 0+ | An expression. |

Results

| Data Type | Description |
|-----------|------------------------------|
| variant | Returns the last expression. |

Remarks

The use function creates a scope by attaching to the knowledge base, evaluating the expressions, then detaching from the knowledge base. The last expression is returned.

Example

```
> (use (knowledge url "file:///c:/premise/kb/defaultKb.mdb")
  (relation Fact :All :Are)
  (new Fact :All people :Are mortal)
  (new Fact :All philosophers :Are people)

  (with [Fact :All as ?x :Are as ?y]
    [Fact :All = ?y :Are as ?z]
    list (knew [Fact :All ?x :Are ?z])))

)
.: Fact_3

> (use
  (knowledge name Mathematics path "c:/premise/kb/" units MB size 2 max 10
  log 1 growth 10)

  (relation Circle :Radius)
  )

.: Circle
```

Related

attach, detach, each, get, knowledge, put, using, which, with

using

Accesses a scope.

Syntax

(**using** *scope expression ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|-----------|-----|--|
| scope | variant | 1 | An environment or module name or binding list. |
| expression | variant | 0+ | An expression. |

Results

| Data Type | Description |
|-----------|------------------------------|
| variant | Returns the last expression. |

Remarks

The using function accesses a scope by attaching to scope and then evaluating the expressions, then detaching from the scope. The last expression is returned.

Example

```
> (using User
    (definitions (scope)))

.: {}

> (extend User
    (function foo {} hello)
    (function bar {} world))

.: User

> (using User
    (definitions (scope)))

.: {{foo "(function foo {} hello)"}
   {bar "(function bar {} world)"}}

> (using (' ?x 1 ?y 2 ?z 3)
    (list ?x ?y ?z))

.: {1 2 3}
```

Related

attach, detach, each, get, knowledge, put, using, which, with

values

Creates a list of values for an assortment.

Syntax

(**values** *assortment*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|---|
| assortment | assortment | 1 | A environment, enumeration, or lexicon. |

Results

| Data Type | Description |
|-----------|------------------|
| list | A list of values |

Remarks

For collections this function returns the same result as the **keys** function.

Example

```
> (global ?M nil)
.: true

> (set ?M (lexicon :X 1 !m noop :T nil A 2))
.: true

> (values ?M)
.: {1 noop nil 2}

> (filter (keys ?M) (given {?v} (term-p ?v)))
.: {:_ !m :T}

> (filter (values ?M) (given {?v} (not (term-p ?v))))
.: {1 noop nil 2}

> (values (collection the quick brown fox))
.: {the quick brown fox}
```

Related

@ *element*, add, bindings, cut, free, get, in, key, keys, size

var

Adds variables to the current environment returning the last assigned value.

Syntax

(**var** *manner* *assignment* ...)

assignment ::= *symbol* *value*

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|--|
| manner | literal | 0-1 | The manner in which the assignments are performed either the literal parallel or tandem (default if omitted) |
| assignment | expression | 1+ | A symbol and value pair. |

Results

| Data Type | Description |
|-----------|----------------------------------|
| variant | Returns the last assigned value. |

Remarks

Sequentially creates variables in the current environment. Each symbol shall replace any extant symbol with the same name in the environment. Returns the value of the last assignment.

Example

```
> (var ?person DrWho)
.: DrWho
> ?person
.: DrWho
```

Related

<-- before tie, --> after tie, assume, constant, dynamic, given, global, let, local, set, symbol, variables

vector

Creates a vector of atoms.

Syntax

(vector *atom* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------|-----------|-----|---|
| atom | atom | 0+ | An atomic value such as a number, time, or literal. |

Results

| Data Type | Description |
|-----------|--|
| vector | Returns a vector containing the atoms. |

Remarks

A vector is an immutable sequence of atoms. Vectors can also be created by simply enclosing the elements with vertical bars called pipes (|). Whenever vectors are manipulated, a new vector is created. Vectors cannot be embedded, and cannot contain sequences.

Example

```
> (vector a b c d)
.: |a b c d|
> (vector 500)
.: |500|
> (vector foo)
.: |foo|
> |a b c d|
.: |a b c d|
```

Related

& merge, && abridge, inside, list

version

Provides version information.

Syntax

(version)

Module

Base

Parameters

None

Results

| Data Type | Description |
|-----------|-------------|
| nil | Returns nil |

Remarks

Prints system and version information.

Example

```
> (version)
[Premise :Version 0.2.0 20180201.001 :OS Windows 10 :Edition Community]
.: nil
```

Related

[bye](#), [help](#), [copyright](#)

void-p

True if a container has zero elements.

Syntax

(void-p sequence)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|----------|-----------|-----|-------------|
| sequence | sequence | 1 | A sequence. |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| truth | True if the sequence has zero items. |

Remarks

True if the container has zero items.

Example

```
> (void-p "not empty")
.: false

> (void-p (expression))
.: true

> (void-p "")
.: true

> (void-p {})
.: true

> (void-p (call))
.: true

> (void-p [])
.: true
```

Related

[more-p](#)

wait

Waits for tasks to end.

Syntax

(wait *tasks* *timeout* **)**

Module

Tasks

Parameters

| Name | Data Type | Qty | Description |
|---------|-----------|-----|--|
| tasks | list | 1 | A list of task resources |
| timeout | instant | 0-1 | The amount of time to wait for the task. The default is eternity which implies to wait indefinitely |

Results

| Data Type | Description |
|-----------|---------------------|
| list | The task resources. |

Remarks

The wait function waits for either the timeout duration or for the task to complete. This function allows tasks to complete in a blocking manner.

Example

```
> (var ?tasks (wait (concurrent (repeat 1000000 foo)
                                (repeat 2000000 bar)
                                (repeat 3000000 baz))
                           \@s10))

.: {task-1 task-2 task-3}

> (map ?tasks (given {?t} (await ?t \@s0)))

.: {foo bar baz}

> (free (wait (complete
                  (repeat 1000 foo)
                  (repeat 2000 bar)
                  (repeat 3000 baz)))))

.: freed
```

Related

wait, abort, concurrent, await, complete, do, done-p, free, task, tarry

which

Creates a list of thoughts for a pattern..

Syntax

```
(which category criterion ... option ...)  
  
category ::= relation  
category ::= list  
criterion ::= slot binding  
criterion ::= slot condition  
criterion ::= slot condition  
binding ::= as symbol  
iteration ::= each symbol  
condition ::= slot = value  
condition ::= slot /= value  
condition ::= is unary-predicate  
condition ::= not unary-predicate  
condition ::= binary-predicate value  
condition ::= slot n-ary-predicate value-list  
condition ::= (predicate value ...)  
option ::= scan number  
option ::= limit number  
option ::= group slot ... by slot ... into symbol  
option ::= merge value ...  
option ::= sort ordering ...  
ordering ::= term comparator  
action ::= give expression  
action ::= list value ...  
action ::= tally
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|------------|-----|---|
| relation | variant | 1 | A relation name or list of thoughts. |
| criterion | expression | 0+ | A slot referent pair or method function pair. |
| option | expression | 0-2 | Either scan , limit , group , merge , sort , or any combination. |

Results

| Data Type | Description |
|-----------|--|
| list | Returns a list of premises matching the pattern. |

Remarks

Looks for existing thoughts matching the criteria. If no variables are specified, the relation name is used, e.g. the Employee shall use an **?Employee** symbol for each instance found. If the action is **give**, then the expression is immediately returned from the which call on the first match. If the action is either **list**, then a list containing each expression is returned or an empty list is returned if nothing matched. If the action is **tally**, then the number of matched premises is returned, or zero if no matches.

Example

```
(relation Car :Make :Model :Year)

(which Car :Make Toyota)

(which (which Car :Make Toyota) each ?thought
  [Dealer :Has (:Make ?tuple)]
  deem true)

> (relation Pairs :A :B) (new Pairs :A 1 :B 2) (new Pairs :A 3 :B 4)

..: Pairs Pairs_1 Pairs_2

> (which Pairs :B 2)

..: Iterator-1

> (relation Fact :All :Are)

..: Fact

> (new Fact :All people :Are mortal) (new Fact :All philosophers :Are people)

..: Fact_1 Fact_2

> (function ExcludedMiddle {}
  (with [Fact :All as ?s :Are as ?m]
    [Fact :All ?m :Are as ?p]
    (no [Fact :All ?s :Are ?p])
    list (premise (knew A :All ?s :Are ?p)))))

..: ExcludedMiddle

> (ExcludedMiddle)

..: {[Fact ^ Fact_3 :All philosophers :Are mortal]}

> (which Fact ^ as ?a :Are mortal)

..: Iterator-2

> (with Fact :Are mortal)

..: {[Fact ^ Fact_1 :All people :Are mortal]
  [Fact ^ Fact_3 :All philosophers :Are mortal]}

> (relation Fruit :Name :Size :Shape :Color)

..: Fruit
```

```

> (step ?i from 1 to (random 1 to 1000)
  (new Fruit :Size (pick {small medium large})
    :Shape (pick {ellipsoid spheroid bananoid}))
    :Color (pick {red orange yellow green})))

.: Fruit_387

> (with [Fruit ^ as ?f :Name as ?name :Size as ?size :Shape spheroid :Color orange]
  (null-p ?name)
  (in {small medium} ?size)
  deem Orange else Other)

.: Orange

> (with
  [Fruit as ?f]
  (let ?f :Size ?size :Shape ?shape :Color ?color)
  (in {small medium} ?size)
  (= ?shape bananoid)
  (= ?color yellow)
  deem true else false)

.: true

> (relation Customer :CustomerId :FirstName :LastName)

.: Employee

> (relation Order :OrderId :CustomerId :Quantity :Product)

.: Order

> (step ?i from 1 to (random 1 to 10)
  (let ?c (new Customer))
  (step ?j from 1 to (random 0 to 10)
    (new Order :CustomerId ?c)))

.: Order_347

> (with [Customer as ?c]
  [(with Order as ?o (= (:CustomerId ?o) (:CustomerId ?c))) each ?orders]
  tally)

.: 2

> (with
  [Customer as ?c :CustomerId ?x]
  [(with Order as :CustomerId ?y (= ?x ?y)) as ?orders]
  scan 30
  limit 5
  list
  [R :Customer ?c :Orders (# ?orders)])

.: {[R :Customer Customer_1 :Orders 3]
[R :Customer Customer_2 :Orders 7]
[R :Customer Customer_4 :Orders 4]
[R :Customer Customer_3 :Orders 9]}

> (relation Student
  :First
  :Last
  :StudentNo
  :Year

```

```

:Tests {})

> (do
(new Student :First John :Last Jones :No 20 :Year 2 :Tests {100 56 87 86})
(new Student :First Lexie :Last Bates :No 16 :Year 3 :Tests {67 68 98 87})
(new Student :First Yong :Last Lee :No 17 :Year 1 :Tests {78 79 90 95})
(new Student :First Paul :Last Gates :No 14 :Year 4 :Tests {86 89 88 89})
(new Student :First Jenny :Last Luria :No 15 :Year 3 :Tests {99 98 97 99})
(new Student :First Phillipa :Last Sanchez :No 18 :Year 2 :Tests {85 87 89 97})
(new Student :First Patrice :Last Jones :No 13 :Year 1 :Tests {99 95 94 72})
(new Student :First Jesus :Last Castaneda :No 12 :Year 4 :Tests {98 89 95 94})
(new Student :First Charlie :Last Rose :No 11 :Year 2 :Tests {97 98 98 97})
(new Student :First Boris :Last Pushkin :No 19 :Year 3 :Tests {100 85 100 100})
(new Student :First Alex :Last Sartre :No 22 :Year 1 :Tests {85 97 86 95})
(new Student :First Elizabeth :Last Wilson :No 21 :Year 4 :Tests {100 100 85 95}))

.: Student_12

> (which Student ^ ?student
  (let ?exam 3 ?score 90)
  (> (@ (:Tests ?student) ?exam) ?score)
  list {:Name (:Last ?student) :Score (@ (:Tests ?student) ?exam)}))

.: {{:Name Bates :Score 98}{:Name Luria :Score 97}{:Name Parker :Score 94}
{:Name Castaneda :Score 95}{:Name Rose :Score 98}{:Name Pushkin :Score 100} }

> (which Student as ?s
  group ?s
  by :Year into ?g
  list ?g
  sort 1 <)

.: {{1 [Student :First Yong :Last Lee :No 17 :Year 1 :Tests {78 79 90 95}]
[Student :First Patrice :Last Jones :No 13 :Year 1 :Tests {99 95 94 72}]
[Student :First Alex :Last Sartre :No 22 :Year 1 :Tests {85 97 86 95}]}
{2 [Student :First John :Last Jones :No 20 :Year 2 :Tests {100 56 87 86}]
[Student :First Phillipa :Last Sanchez :No 18 :Year 2 :Tests {85 87 89 97}]
[Student :First Charlie :Last Rose :No 11 :Year 2 :Tests {97 98 98 97}]}
{3 [Student :First Lexie :Last Bates :No 16 :Year 3 :Tests {67 68 98 87}]
[Student :First Jenny :Last Luria :No 15 :Year 3 :Tests {99 98 97 99}]
[Student :First Boris :Last Pushkin :No 19 :Year 3 :Tests {100 85 100 100}]}
{4 [Student :First Paul :Last Gates :No 14 :Year 4 :Tests {86 89 88 89}]
[Student :First Jesus :Last Castaneda :No 12 :Year 4 :Tests {98 89 95 94}]
[Student :First Elizabeth :Last Wilson :No 021 :Year 4 :Tests {100 100 85 95})]}}

> (which Student as ?s :Id ?id
  group ?id
  by (given {?s} (@ ($ (:Last ?s)) 1)) into ?g
  list ?g)

.: {{"B" 16}{"C" 12}{"G" 14}{"P" 19} {"R" 11} {"L" 17 15} {"W" 21} {"S" 18 22} {"J" 20 13} }

> (which Student as ?s :Tests ?t
  (let ?percentile (/ (avg ?t) 10))
  group ($ (:First ?s) (:Last ?s))
  by ?percentile into ?g
  list ?g
  sort 1 < )

.: {{8 "John Jones" "Lexie Bates" "Yong Lee" "Paul Gates" "Phillipa Sanchez"}
{9 "Jenny Luria" "Patrice Jones" "Jesus Castaneda" "Charlie Rose" "Boris Pushkin" "Alex Sartre" "Elizabeth Wilson"}}


```

```

> (with Student as ?s
  group [R :FirstLetter (@ ($ (:Last ?s)) 1)
  :Score (:Score ?s)]
  by (given {?s} (> (@ (:Tests ?s) 1) 85)))
  into ?g
  list ?g
  sort (given {?x} (:FirstLetter ?x)) <)

.: {{"C" [R :FirstLetter "C" :Score 98]} {"J" [R :FirstLetter "J" :Score 100]
[R :FirstLetter "J" :Score 99]} {"L" [R :FirstLetter "L" :Score 99]}
{"G" [R :FirstLetter "G" :Score 86]} {"P" [R :FirstLetter "P" :Score 100]}
 {"R" [R :FirstLetter "R" :Score 97]} {"W" [R :FirstLetter "W" :Score 100]}}

```

Related

each, knew, known, new, relation, such, suppose, unless, using, with

while

Loops expressions while a condition is true.

Syntax

(while *condition* *expression* ... **)**

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|------------|------------|-----|-------------------------|
| condition | truth | 1 | A truth expression |
| expression | expression | 0+ | A call to be evaluated. |

Results

| Data Type | Description |
|-----------|--------------------------------------|
| value | The last value of the last iteration |

Remarks

None

Example

```
> (modular ?i 0)
.: true

> (while (< ?i 1000)
    (--> ++ ?i))      ; assign ?i to the increment of ?i.

.: 1000
```

Related

for, iterate, loop, repeat, step, until

whitespace-p

True if the first position of a string is whitespace.

Syntax

(whitespace-p *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|-------------|
| value | variant | 1 | A value. |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the value is a string and the first position is whitespace. |

Remarks

Whitespace is a space, newline, or tab.

Example

```
> (whitespace-p {a b c})  
. : false  
  
> (whitespace-p "a b c")  
. : false  
  
> (whitespace-p " a b c")  
. : true  
  
> (whitespace-p 450)  
. : false  
  
> (whitespace-p " foo")  
. : true  
  
> (whitespace-p "hello world")  
. : false
```

Related

[capitalized-p](#), [letter-p](#), [lowercase-p](#), [uppercase-p](#)

with

Matches patterns against the knowledge base.

Syntax

```
(with premises option ... action )  
premises ::= premise  
premises ::= premise premises  
premises ::= premise premises  
premises ::= premise restriction ... premises  
restriction ::= (predicate value ...)  
premise ::= [category descriptor ... ]  
category ::= type  
category ::= list  
descriptor ::= slot binding  
descriptor ::= slot condition  
binding ::= as symbol  
binding ::= each symbol  
condition ::= slot = value  
condition ::= slot /= value  
condition ::= slot is unary-predicate  
condition ::= slot not unary-predicate  
condition ::= slot binary-predicate value  
condition ::= slot n-ary-predicate value-list  
condition ::= (predicate value ...)  
option ::= scan number  
option ::= limit number  
option ::= group slot ... by slot ... into symbol  
option ::= merge value ...  
option ::= sort ordering ...  
action ::= deem value1 else value2  
action ::= do expression ...  
action ::= give expression  
action ::= list value ...  
action ::= tally  
action ::= yield expression  
ordering ::= term comparator
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------------------|------------|-----|---|
| clauses / premises | expression | 1+ | Either one or more clauses, or, one or more premises |
| option | expression | 0-2 | Either scan , limit , group , merge , sort , or any combination. |
| action | literal | 0-1 | Either deem , do , give , list , tally , yield . |
| expression | expression | 0+ | An expression |

Results

| Data Type | Description |
|------------|---|
| expression | if the action is deem , then either the success or failure value is returned; if the action is do then the last expression value on the last iteration or nil; If the action is list , then a list of premises or an empty list; if the action is tally , then the number of matches or zero, if the action is give , then the expression is returned. |

Remarks

Note that a predicate is a truth function, **scan** tells the number of matches to attempt (default is **infinity**), and **limit** tells how many results to return (default is **infinity**). If the action is **deem**, then the subsequent value is returned if the final descriptor is matched, otherwise the **else** value is returned. If the action is **do**, then for each match the expressions are run, and for the last match, the last value of the last expression is returned, or nil is returned if all descriptors were false or had no matches. If the action is **give**, then the expression is immediately returned from the with call on the first match, akin to **return from with**. If the action is **yield**, then the expression is immediately returned from the with call, but may be resumed with a **next** function call. When all matches are completed, the **done** function is implicitly called to terminate the generator. If the action is **group**, followed by a list of slots and values from which the group is picked, then the literal **by** and a set of slots and values that determines the grouping inclusion , and finally the literal **into** followed by a symbol to which the group shall be bound on each iteration. If the action is either **list** (or **merge**), then a (merged) list is returned or an empty list if nothing matched. If the action is **sort** then the list of slots and comparators should follow. (Note that **list** is required if the sort action is selected.) If the action is **tally**, then the number of matched premises is returned, or zero if no matches.

Example

```
> (relation Fact :All :Are)

.: Fact

> (new Fact :All people :Are mortal) (new Fact :All philosophers :Are people)

.: Fact_1 Fact_2

> (function ExcludedMiddle {}
  (with [Fact :All as ?s :Are as ?m]
    [Fact :All = ?m :Are as ?p]
    (no [Fact :All = ?s :Are = ?p])
    list (premise (knew A :All ?s :Are ?p)))))

.: ExcludedMiddle

> (ExcludedMiddle)

.: {[Fact ^ Fact_3 :All philosophers :Are mortal]}

> (which Fact ^ as ?a :Are = mortal list ?a)

.: {Fact_1 Fact_3}

> (which Fact :Are mortal)

.: {[Fact ^ Fact_1 :All people :Are mortal]
  [Fact ^ Fact_3 :All philosophers :Are mortal]}

> (relation Fruit :Name :Size :Shape :Color)

.: Fruit

> (step ?i from 1 to (random 1 to 1000)
  (new Fruit :Size (pick {small medium large})
    :Shape (pick {ellipsoid spheroid bananoid}))
    :Color (pick {red orange yellow green})))

.: Fruit_387

> (with [Fruit ^ as ?f :Name as ?name :Size as ?size :Shape = spheroid
  :Color = orange]
  (null-p ?name)
  (in {small medium} ?size)
  deem Orange else Other)

.: Orange

> (with
  [Fruit as ?f]
  (let ?f :Size ?size :Shape ?shape :Color ?color)
  (in {small medium} ?size)
  (= ?shape bananoid)
  (= ?color yellow)
  deem true else false)

.: true
```

```

> (relation Customer :CustomerId :FirstName :LastName)

.: Employee

> (relation Order :OrderId :CustomerId :Quantity :Product)

.: Order

> (step ?i from 1 to (random 1 to 10)
  (let ?c (new Customer))
  (step ?j from 1 to (random 0 to 10)
    (new Order :CustomerId ?c)))

.: Order_347

> (with [Customer as ?c]
  [(which Order as ?o (= (:CustomerId ?o) (:CustomerId ?c))) each ?orders]
 tally)

.: 2

> (with
  [Customer as ?c :CustomerId ?x]
  [(with Order :CustomerId = ?x) as ?orders]
 scan 30
 limit 5
 list
  [R :Customer ?c :Orders (# ?orders)])

.: {[R :Customer Customer_1 :Orders 3]
[R :Customer Customer_2 :Orders 7]
[R :Customer Customer_4 :Orders 4]
[R :Customer Customer_3 :Orders 9]}

> (relation Student
  :First
  :Last
  :StudentNo
  :Year
  :Tests {})

> (do
(new Student :First John :Last Jones :No 20 :Year 2 :Tests {100 56 87 86})
(new Student :First Lexie :Last Bates :No 16 :Year 3 :Tests {67 68 98 87})
(new Student :First Yong :Last Lee :No 17 :Year 1 :Tests {78 79 90 95})
(new Student :First Paul :Last Gates :No 14 :Year 4 :Tests {86 89 88 89})
(new Student :First Jenny :Last Luria :No 15 :Year 3 :Tests {99 98 97 99})
(new Student :First Phillipa :Last Sanchez :No 18 :Year 2 :Tests {85 87 89 97})
(new Student :First Patrice :Last Jones :No 13 :Year 1 :Tests {99 95 94 72})
(new Student :First Jesus :Last Castaneda :No 12 :Year 4 :Tests {98 89 95 94})
(new Student :First Charlie :Last Rose :No 11 :Year 2 :Tests {97 98 98 97})
(new Student :First Boris :Last Pushkin :No 19 :Year 3 :Tests {100 85 100 100})
(new Student :First Alex :Last Sartre :No 22 :Year 1 :Tests {85 97 86 95})
(new Student :First Elizabeth :Last Wilson :No 21 :Year 4 :Tests {100 100 85 95}))

.: Student_12

```

```

> (with Student ^ ?student
  (let ?exam 3 ?score 90)
  (> (@ (:Tests ?student) ?exam) ?score)
  list {:Name (:Last ?student) :Score (@ (:Tests ?student) ?exam)})}

.: {{:Name Bates :Score 98}{:Name Luria :Score 97}{:Name Parker :Score 94}
{:Name Castaneda :Score 95}{:Name Rose :Score 98}{:Name Pushkin :Score 100}}


> (with [Student as ?s]
  group ?s
  by :Year into ?g
  list ?g
  sort 1 <)

.: {{1 [Student :First Yong :Last Lee :No 17 :Year 1 :Tests {78 79 90 95}]
[Student :First Patrice :Last Jones :No 13 :Year 1 :Tests {99 95 94 72}]
[Student :First Alex :Last Sartre :No 22 :Year 1 :Tests {85 97 86 95}]}
{2 [Student :First John :Last Jones :No 20 :Year 2 :Tests {100 56 87 86}]
[Student :First Phillipa :Last Sanchez :No 18 :Year 2 :Tests {85 87 89 97}]
[Student :First Charlie :Last Rose :No 11 :Year 2 :Tests {97 98 98 97}]}
{3 [Student :First Lexie :Last Bates :No 16 :Year 3 :Tests {67 68 98 87}]
[Student :First Jenny :Last Luria :No 15] :Year 3 :Tests {99 98 97 99}]
[Student :First Boris :Last Pushkin :No 19 :Year 3 :Tests {100 85 100 100}]}
{4 [Student :First Paul :Last Gates :No 14 :Year 4 :Tests {86 89 88 89}]
[Student :First Jesus :Last Castaneda :No 12 :Year 4 :Tests {98 89 95 94}]
[Student :First Elizabeth :Last Wilson :No 021 :Year 4 :Tests {100 100 85 95})]}]}}}

> (which Student as ?s :Id as ?id
  group ?id
  by (given {?s} (@ ($ (:Last ?s)) 1)) into ?g
  list ?g)

.: {{"B" 16} {"C" 12} {"G" 14} {"P" 19} {"R" 11} {"L" 17 15} {"W" 21} {"S" 18 22} {"J" 20
13} }

> (with [Student as ?s :Tests as ?t]
  (let ?percentile (/ (avg ?t) 10))
  group ($ (:First ?s) (:Last ?s))
  by ?percentile into ?g
  list ?g
  sort 1 < )

.: {{8 "John Jones" "Lexie Bates" "Yong Lee" "Paul Gates" "Phillipa Sanchez"}
{9 "Jenny Luria" "Patrice Jones" "Jesus Castaneda" "Charlie Rose" "Boris
Pushkin" "Alex Sartre" "Elizabeth Wilson"}}

> (with [Student as ?s]
  group [R :FirstLetter (@ ($ (:Last ?s)) 1)
  :Score (:Score ?s)]
  by (given {?s} (> (@ (:Tests ?s) 1) 85)))
  into ?g
  list ?g
  sort (given {?x} (:FirstLetter ?x)) <)

```

```
.: [{"C" [R :FirstLetter "C" :Score 98]},{"J" [R :FirstLetter "J" :Score 100]
[R :FirstLetter "J" :Score 99]}, {"L" [R :FirstLetter "L" :Score 99]}
{"G" [R :FirstLetter "G" :Score 86]}, {"P" [R :FirstLetter "P" :Score 100]}
 {"R" [R :FirstLetter "R" :Score 97]}, {"W" [R :FirstLetter "W" :Score 100]}]}
```

Related

each, which

within

Matches patterns against the knowledge base.

Syntax

```
(within which slot premises option ... action)  
premises ::= premise  
premises ::= premise premises  
premises ::= premise premises  
premises ::= premise restriction ... premises  
restriction ::= (predicate value ...)  
premise ::= [category descriptor ... ]  
category ::= type  
category ::= list  
descriptor ::= slot binding  
descriptor ::= slot condition  
binding ::= as symbol  
binding ::= each symbol  
condition ::= slot = value  
condition ::= slot /= value  
condition ::= slot is unary-predicate  
condition ::= slot not unary-predicate  
condition ::= slot binary-predicate value  
condition ::= slot n-ary-predicate value-list  
condition ::= (predicate value ...)  
option ::= scan number  
option ::= limit number  
option ::= group slot ... by slot ... into symbol  
option ::= merge value ...  
option ::= sort ordering ...  
action ::= deem value1 else value2  
action ::= do expression ...  
action ::= give expression  
action ::= list value ...  
action ::= tally  
action ::= yield expression  
ordering ::= term comparator
```

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------------------|------------|-----|---|
| which | literal | 1 | A thought or relation. |
| slot | slot | 1 | The slot in which to match. |
| clauses / premises | expression | 1+ | Either one or more clauses, or, one or more premises |
| option | expression | 0-2 | Either scan , limit , group , merge , sort , or any combination. |
| action | literal | 0-1 | Either deem , do , give , list , tally , yield . |
| expression | expression | 0+ | An expression |

Results

| Data Type | Description |
|------------|---|
| expression | if the action is deem , then either the success or failure value is returned; if the action is do then the last expression value on the last iteration or nil; If the action is list , then a list of premises or an empty list; if the action is tally , then the number of matches or zero, if the action is give , then the expression is returned. |

Remarks

Note that a predicate is a truth function, **scan** tells the number of matches to attempt (default is **infinity**), and **limit** tells how many results to return (default is **infinity**). If the action is **deem**, then the subsequent value is returned if the final descriptor is matched, otherwise the **else** value is returned. If the action is **do**, then for each match the expressions are run, and for the last match, the last value of the last expression is returned, or nil is returned if all descriptors were false or had no matches. If the action is **give**, then the expression is immediately returned from the `with` call on the first match, akin to **return from with**. If the action is **yield**, then the expression is immediately returned from the `with` call, but may be resumed with a **next** function call. When all matches are completed, the **done** function is implicitly called to terminate the generator. If the action is **group**, followed by a list of slots and values from which the group is picked, then the literal **by** and a set of slots and values that determines the grouping inclusion, and finally the literal **into** followed by a symbol to which the group shall be bound on each iteration. If the action is either **list** (or **merge**), then a (merged) list is returned or an empty list if nothing matched. If the action is **sort** then the list of slots and comparators should follow. (Note that **list** is required if the sort action is selected.) If the action is **tally**, then the number of matched premises is returned, or zero if no matches.

Example

```
> (relation Node :Facts)

.: Node

.: Fact

> (assume Node
  :Facts  [Fact :All people :Are mortal]
           [Fact :All philosophers :Are people])

.: Node_1

> (function ExcludedMiddle {}
  (within Node_1 :Facts
    [Fact :All as ?s :Are as ?m]
    [Fact :All = ?m :Are as ?p]
    (no [Fact :All = ?s :Are = ?p])
    list (premise (knew A :All ?s :Are ?p)))))

.: ExcludedMiddle

> (ExcludedMiddle)

.: {[Fact ^ Fact_3 :All philosophers :Are mortal]}

> (which Fact ^ as ?a :Are = mortal list ?a)

.: {Fact_1 Fact_3}

> (which Fact :Are mortal)

.: {[Fact ^ Fact_1 :All people :Are mortal]
  [Fact ^ Fact_3 :All philosophers :Are mortal]}

> (relation Fruit :Name :Size :Shape :Color)

.: Fruit

> (step ?i from 1 to (random 1 to 1000)
  (new Fruit :Size (pick {small medium large})
    :Shape (pick {ellipsoid spheroid bananoid}))
    :Color (pick {red orange yellow green})))

.: Fruit_387

> (with [Fruit ^ as ?f :Name as ?name :Size as ?size :Shape = spheroid
  :Color = orange]
  (null-p ?name)
  (in {small medium} ?size)
  deem Orange else Other)

.: Orange
```

```

> (with
  [Fruit as ?f]
  (let ?f :Size ?size :Shape ?shape :Color ?color)
  (in {small medium} ?size)
  (= ?shape bananoid)
  (= ?color yellow)
  deem true else false)

..: true
> (relation Customer :CustomerId :FirstName :LastName)

..: Employee

> (relation Order :OrderId :CustomerId :Quantity :Product)

..: Order

> (step ?i from 1 to (random 1 to 10)
  (let ?c (new Customer))
  (step ?j from 1 to (random 0 to 10)
    (new Order :CustomerId ?c)))

..: Order 347

> (with [Customer as ?c]
  [(which Order as ?o (= (:CustomerId ?c) (:CustomerId ?c))) each ?orders]
  tally)

..: 2

> (with
  [Customer as ?c :CustomerId ?x]
  [(with Order :CustomerId = ?x) as ?orders]
  scan 30
  limit 5
  list
  [R :Customer ?c :Orders (# ?orders)])

..: {[R :Customer Customer_1 :Orders 3]
[R :Customer Customer_2 :Orders 7]
[R :Customer Customer_4 :Orders 4]
[R :Customer Customer_3 :Orders 9]}

> (relation Student
  :First
  :Last
  :StudentNo
  :Year
  :Tests {})

> (do
  (new Student :First John :Last Jones :No 20 :Year 2 :Tests {100 56 87 86})
  (new Student :First Lexie :Last Bates :No 16 :Year 3 :Tests {67 68 98 87})
  (new Student :First Yong :Last Lee :No 17 :Year 1 :Tests {78 79 90 95})
  (new Student :First Paul :Last Gates :No 14 :Year 4 :Tests {86 89 88 89})
  (new Student :First Jenny :Last Luria :No 15 :Year 3 :Tests {99 98 97 99})
  (new Student :First Phillipa :Last Sanchez :No 18 :Year 2 :Tests {85 87 89 97})
  (new Student :First Patrice :Last Jones :No 13 :Year 1 :Tests {99 95 94 72})
  (new Student :First Jesus :Last Castaneda :No 12 :Year 4 :Tests {98 89 95 94})
  (new Student :First Charlie :Last Rose :No 11 :Year 2 :Tests {97 98 98 97})
  (new Student :First Boris :Last Pushkin :No 19 :Year 3 :Tests {100 85 100 100})
  (new Student :First Alex :Last Sartre :No 22 :Year 1 :Tests {85 97 86 95})
```

```
(new Student :First Elizabeth :Last Wilson :No 21 :Year 4 :Tests {100 100 85 95}))
```

```
..: Student_12
```

```

> (with Student ^ ?student
  (let ?exam 3 ?score 90)
  (> (@ (:Tests ?student) ?exam) ?score)
  list {:Name (:Last ?student) :Score (@ (:Tests ?student) ?exam)})

..: {{"Name": "Bates", "Score": 98}, {"Name": "Luria", "Score": 97}, {"Name": "Parker", "Score": 94},
    {"Name": "Castaneda", "Score": 95}, {"Name": "Rose", "Score": 98}, {"Name": "Pushkin", "Score": 100}]

> (with [Student as ?s]
  group ?s
  by :Year into ?g
  list ?g
  sort 1 <)

..: [{1: {"First": "Yong", "Last": "Lee", "No": 17, "Year": 1, "Tests": [78, 79, 90, 95]}, "id": 1}, {"2: {"First": "Patrice", "Last": "Jones", "No": 13, "Year": 1, "Tests": [99, 95, 94, 72]}, "id": 2}, {"3: {"First": "Alex", "Last": "Sartre", "No": 22, "Year": 1, "Tests": [85, 97, 86, 95]}, "id": 3}, {"4: {"First": "John", "Last": "Jones", "No": 20, "Year": 2, "Tests": [100, 56, 87, 86]}, "id": 4}, {"5: {"First": "Phillipa", "Last": "Sanchez", "No": 18, "Year": 2, "Tests": [85, 87, 89, 97]}, "id": 5}, {"6: {"First": "Charlie", "Last": "Rose", "No": 11, "Year": 2, "Tests": [97, 98, 98, 97]}, "id": 6}, {"7: {"First": "Lexie", "Last": "Bates", "No": 16, "Year": 3, "Tests": [67, 68, 98, 87]}, "id": 7}, {"8: {"First": "Jenny", "Last": "Luria", "No": 15, "Year": 3, "Tests": [99, 98, 97, 99]}, "id": 8}, {"9: {"First": "Boris", "Last": "Pushkin", "No": 19, "Year": 3, "Tests": [100, 85, 100, 100]}, "id": 9}, {"10: {"First": "Paul", "Last": "Gates", "No": 14, "Year": 4, "Tests": [86, 89, 88, 89]}, "id": 10}, {"11: {"First": "Jesus", "Last": "Castaneda", "No": 12, "Year": 4, "Tests": [98, 89, 95, 94]}, "id": 11}, {"12: {"First": "Elizabeth", "Last": "Wilson", "No": 021, "Year": 4, "Tests": [100, 100, 85, 95]}, "id": 12}]

> (which Student as ?s :Id as ?id
  group ?id
  by (given (?s) (@ (:Last ?s)) 1) into ?g
  list ?g)

..: [{"id": 1, "First": "John", "Last": "Jones", "No": 16}, {"id": 2, "First": "Patrice", "Last": "Jones", "No": 12}, {"id": 3, "First": "Alex", "Last": "Sartre", "No": 14}, {"id": 4, "First": "Charlie", "Last": "Rose", "No": 19}, {"id": 5, "First": "Phillipa", "Last": "Sanchez", "No": 11}, {"id": 6, "First": "Yong", "Last": "Lee", "No": 17}, {"id": 7, "First": "Lexie", "Last": "Bates", "No": 15}, {"id": 8, "First": "Jenny", "Last": "Luria", "No": 13}, {"id": 9, "First": "Boris", "Last": "Pushkin", "No": 21}, {"id": 10, "First": "Jesus", "Last": "Castaneda", "No": 18}, {"id": 11, "First": "Paul", "Last": "Gates", "No": 22}, {"id": 12, "First": "Elizabeth", "Last": "Wilson", "No": 20}]

> (with [Student as ?s :Tests as ?t]
  (let ?percentile (/ (avg ?t) 10))
  group ($ (:First ?s) (:Last ?s))
  by ?percentile into ?g
  list ?g
  sort 1 <)

..: [{"First": "John", "Last": "Jones", "Percentile": 8}, {"First": "Lexie", "Last": "Bates", "Percentile": 12}, {"First": "Yong", "Last": "Lee", "Percentile": 14}, {"First": "Charlie", "Last": "Rose", "Percentile": 19}, {"First": "Phillipa", "Last": "Sanchez", "Percentile": 11}, {"First": "Patrice", "Last": "Jones", "Percentile": 17}, {"First": "Jesus", "Last": "Castaneda", "Percentile": 15}, {"First": "Boris", "Last": "Pushkin", "Percentile": 21}, {"First": "Alex", "Last": "Sartre", "Percentile": 18}, {"First": "Elizabeth", "Last": "Wilson", "Percentile": 22}]

> (with [Student as ?s]
  group [R :FirstLetter (@ ($ (:Last ?s)) 1)
        :Score (:Score ?s)] into ?g
  by (given (?s) (> (@ (:Tests ?s) 1) 85)) into ?g
  list ?g
  sort (given (?x) (:FirstLetter ?x)) <)

```

```
..: {"C" [R :FirstLetter "C" :Score 98]} {"J" [R :FirstLetter "J" :Score 100] [R :FirstLetter "J" :Score 99]} {"L" [R :FirstLetter "L" :Score 99]} {"G" [R :FirstLetter "G" :Score 86]} {"P" [R :FirstLetter "P" :Score 100]} {"R" [R :FirstLetter "R" :Score 97]} {"W" [R :FirstLetter "W" :Score 100]}  
~~~
```

Related

each, which

wrap

Prohibits modifications to a module.

Syntax

(**wrap** *module*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| module | module | 1 | A module |

Results

| Data Type | Description |
|-----------|----------------------------|
| literal | The literal wrapped |

Remarks

None

Example

```
> (module Math
    (function sum ?args (apply + ?args)))

.: Math

> (wrap Math)

.: wrapped

> (extend Math
    (function prod ?args (apply * ?args)))

.: [Failure :Name Permission :Text "Can't modify a wrapped module"]

> (unwrap Math)

.: unwrapped

> (extend Math
    (function prod ?args (apply * ?args)))

.: Math
```

Related

[unwrap](#)

write

Writes values to a writeable resource.

Syntax

(write *file value* **)**

Module

IO

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|--|
| file | resource | 1 | A file or pipe. |
| value | variant | 1 | A value to write |
| bits | integer | 0-1 | The size of the byte or character in bits. Either 7, 8, 16, or 32. (Default is 8.) |

Results

| Data Type | Description |
|-----------|----------------------|
| resource | Returns the resource |

Remarks

None.

Example

```
> (let ?file (file name "test.txt"))

.: true

> (read ?file)

.: "Hello"

> (seek ?file 1)

.: File-1

> (write ?file ($ Goodbye my friends.))

.: File-1

> (close ?file)

.: closed
```

Related

[bytes](#), [close](#), [file](#), [read](#), [reposition](#), [write](#)

xnor

Logical equivalence.

Syntax

(xnor *truth* *truth*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------|
| truth | truth | 2 | A truth value |

Results

| Data Type | Description |
|-----------|---------------------------------------|
| truth | True if the argument values are equal |

Remarks

Implements logical equivalence as the result is true when both arguments are the same.

Example

```
> (xnor true false)  
. : false  
  
> (xnor false true)  
. : false  
  
> (xnor false false)  
. : true  
  
> (xnor true true)  
. : true
```

Related

and, nand, void, nor, not, or, is, xor

xor

Logical exclusive disjunction.

Syntax

(xor *truth* *truth*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---------------|
| truth | truth | 2 | A truth value |

Results

| Data Type | Description |
|-----------|---|
| truth | True if the argument values are not equal |

Remarks

None.

Example

```
> (xor true false)
.: true
> (xor false true)
.: true
> (xor false false)
.: false
> (xor true true)
.: false
```

Related

and, nand, void, nor, not, or, is, xnor

yield

Returns a result from a series generator.

Syntax

(yield *value*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-------|-----------|-----|---|
| value | variant | 0-1 | Any value to be returned. Default is nil. |

Results

| Data Type | Description |
|-----------|---------------------|
| variant | The returned value. |

Remarks

Yield exits from an encompassing generator. If the value is not specified, **nil** is returned. When yield is executed the encompassing generator immediately returns the value. To termine use **done**.

Example

```
> (generator myGenerator {?interrupt}
  (yield 1)
  (yield 2)
  (if ?interrupt (done))
  (yield 3)
  (yield 4)
  (done))

.: myGenerator

> (collect ?x in (myGenerator true) ?x)

.: {1 2}

> (collect ?x in (myGenerator false) ?x)

.: {1 2 3 4}
```

Related

[done](#), [generator-p](#), [series](#)

zap

Updates associations and sequences with nil values.

Syntax

(**zap** *container place ...*)

Module

Base

Parameters

| Name | Data Type | Qty | Description |
|-----------|-----------|-----|---------------------------|
| container | variant | 1 | A sequence or assortment |
| place | variant | 0+ | A slot, key, or position. |

Results

| Data Type | Description |
|-----------|---------------------------------|
| container | Returns the modified container. |

Remarks

If no places are provided, then all keys, positions, or slots are set to **nil**.

Example

```
> (relation A :x 1 :y 2 :z 3)
.: a

> (get (new A))
.: [A ^ A_1 :x 1 :y 2 :z 3]

> (zap A_1 :y)
.: A_1

> (get A_1)
.: [A ^ A_1 :x 1 :y nil :z 3]

> (zap A_1)
.: A_1

> (get A_1)
.: [A ^ A_1 :x nil :y nil :z nil]
```

Related

let, new, nix

zero-p

True if a number is zero.

Syntax

(zero-p *number*)

Module

Math

Parameters

| Name | Data Type | Qty | Description |
|--------|-----------|-----|-------------|
| number | number | 1 | A number |

Results

| Data Type | Description |
|-----------|-------------------------------------|
| truth | Returns true if the number is zero. |

Remarks

None.

Example

```
> (zero-p 100)
.: false
> (zero-p 0)
.: true
> (zero-p -0)
.: true
> (zero-p -30)
: false
```

Related

even-p, negative-p, odd-p, positive-p

Bibliography

1. Chambers, C. 1992
The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages
2. Forgy, Charles 1981
OPS5 User's Manual, Technical Report CMU-CS-81-135 (Carnegie Mellon University)
3. Krishnamurthi, Sriram 2017
Programming Languages: Application and Interpretation, Second Edition
4. Linker, Sheldon O. 2005
A knowledge base and question answering system based on loglan and english
5. Linker, Sheldon O., Miller, M. S. P. 2014
Association for the Advancement of Artificial Intelligence
Spring Symposium Series 2014 (AAAI SSS 14)
Implementing Selves with Safe Motivational Systems and Self-Improvement Invited Talk
"Premise: A Language for Cognitive Systems"
6. Miller, Michael S. P. 2013
The Construction of Reality in a Cognitive System
7. Miller, Michael S. P. 2018
Building Minds with Patterns, ISBN 978-0-692-54140-1
8. Queinnec, Christian 1994
Lisp in Small Pieces, Cambridge University Press, ISBN 0-521-54566-8
9. Steele Jr , Guy L. 1981
Common LISP: The Language, 2nd Ed., Digital Press, ISBN 978-1-55558-041-4.

Index

| | | | |
|---------------------------|---------|----------------------------|---------|
| -- decrement | 125 | append | 178 |
| ' quote | 141 | apply | 179 |
| - subtraction | 126 | arity | 180 |
| # size | 128 | array | 181 |
| \$ concatenate | 129 | asecant | 182 |
| \$\$ elide | 130 | asine | 183 |
| % shell | 131 | ask | 184 |
| & merge | 132 | atan | 186 |
| && abridge | 133 | atom | 26 |
| * multiplication | 134 | attach | 187 |
| ** exponentiation | 135 | Auto-named Functions | 62 |
| / division | 136 | average | 188 |
| // nth root | 137 | avg | 189 |
| /~ dissimilarity | 138 | await | 190 |
| /= unequal | 139 | before-p | 191 |
| @ element | 127 | beginning | 192 |
| \n newline | 154 | best | 193 |
| \s space | 155 | beyond | 195 |
| ^ thought id | 156 | big | 28, 196 |
| ` expand | 142 | binary | 27 |
| ~ similarity | 140 | bind | 40, 197 |
| + addition | 143 | bindings | 198 |
| ++ increment | 144 | bion | 199 |
| < less | 146 | bitwise | 200 |
| <- before tie | 39, 145 | bound | 202 |
| <= less or equal | 147 | bound-p | 203 |
| <== before put | 148 | bracket | 204 |
| = equal | 149 | break | 70, 205 |
| ==> after put | 150 | busy-p | 206 |
| --> after tie | 39, 152 | but | 207 |
| > greater | 151 | bye | 208 |
| >= greater or equal | 153 | call | 57, 209 |
| abolish | 157 | can | 210 |
| abort | 158 | cancel | 211 |
| about | 159 | cancelled-p | 212 |
| abs | 160 | canonify | 213 |
| absent | 161 | capitalize | 214 |
| acosecant | 162 | case | 64, 215 |
| acosine | 163 | categorize | 216 |
| acotangent | 164 | cede | 217 |
| actual | 165 | ceiling | 218 |
| add | 166 | cell | 219 |
| address | 167 | char | 220 |
| agent | 168 | choose | 221 |
| align | 169 | clear | 222 |
| alike | 170 | clip | 223 |
| all | 171 | clone | 224 |
| alphabetic-p | 173 | close | 225 |
| phanumeric-p | 174 | closure | 226 |
| and | 175 | coalesce | 228 |
| anonym | 62 | collect | 230 |
| Anonymous Functions | 62 | collection | 45, 232 |
| any | 176 | combine | 233 |

| | |
|-------------------------------|-------------|
| common..... | 234 |
| comparable-p..... | 235 |
| compare..... | 236 |
| complete..... | 75, 237 |
| complex | 29, 238 |
| Complex numbers..... | 29 |
| compose | 239 |
| conceive | 240 |
| concurrent | 75, 241 |
| configuration | 46, 242 |
| confirm | 69, 243 |
| confute..... | 69, 244 |
| constant..... | 34, 245 |
| container..... | 52 |
| contains | 246 |
| continuation | 48 |
| continue | 70, 247 |
| convertible-p..... | 248 |
| copy | 249 |
| copyright..... | 250 |
| correlate | 251 |
| cosecant..... | 252 |
| cosine..... | 253 |
| cotangent..... | 254 |
| count..... | 67, 255 |
| critical | 256 |
| cut..... | 257 |
| data..... | 48, 50, 258 |
| date..... | 31, 260 |
| decimal | 28 |
| Declarative Programming | 24 |
| declared-p..... | 261 |
| decode | 262 |
| default..... | 263 |
| definitions..... | 264 |
| defunct..... | 265 |
| degrees | 266 |
| delete..... | 267 |
| dependencies..... | 268 |
| depends | 270 |
| deq..... | 269 |
| detach..... | 271 |
| did | 273 |
| difference..... | 272 |
| digit | 274 |
| digit-p..... | 275 |
| digits | 276 |
| dimensions..... | 277 |
| discard | 278 |
| disjoint-p..... | 279 |
| distinct | 280 |
| distribute | 281 |
| divide | 282 |
| divisible-p..... | 283 |
| do | 65, 284 |
| done..... | 286 |
| drop | 287 |
| duplicate | 288 |
| during-p..... | 289 |
| dynamic..... | 290 |
| Dynamic Scope..... | 36 |
| e | 291 |
| each..... | 292 |
| encode..... | 296 |
| endoym | 62 |
| enq | 297 |
| ensure | 72, 298 |
| entries | 299 |
| enum | 300 |
| enumeration..... | 46, 301 |
| enumerations | 302 |
| environment..... | 46, 303 |
| ephemerons | 46 |
| epoch..... | 304 |
| eradicate | 305 |
| erase..... | 306 |
| escape | 73, 307 |
| eternity..... | 33 |
| eval..... | 308 |
| even-p | 309 |
| every..... | 310 |
| exactly | 311 |
| exchange | 312 |
| excludes..... | 313 |
| exists-p | 26, 314 |
| exit..... | 70 |
| exit..... | 315 |
| exonym..... | 57 |
| exponential | 316 |
| expression | 52, 317 |
| extend | 318 |
| facility..... | 319 |
| few | 321 |
| file | 49, 322 |
| files..... | 323 |
| fill | 324 |
| filter..... | 325 |
| find | 326 |
| finishes-p | 328 |
| finishing | 329 |
| fix | 38, 330 |
| float | 29, 331 |
| floor | 332 |
| fold | 333 |
| folder..... | 334 |
| folders | 335 |
| for..... | 66, 336 |
| forever | 337 |
| forgo..... | 338 |
| format | 339 |
| fractional | 340 |
| free | 341 |
| full | 342 |
| function | 343 |
| Functional Programming | 23 |
| functions | 346 |

| | |
|------------------------------|---------|
| gather | 347 |
| generator | 348 |
| get | 350 |
| given | 351 |
| global | 353 |
| Global Scope | 35 |
| go | 74, 354 |
| grok | 355 |
| group | 356 |
| halt | 357 |
| has | 358 |
| hash | 359 |
| help | 360 |
| here | 361 |
| hexadecimal | 28 |
| hexatrigesimal | 28 |
| hyperlink | 362 |
| id | 363 |
| identical-p | 364 |
| identifier | 34 |
| identity | 43, 365 |
| idle | 366 |
| if | 64, 367 |
| imaginary | 29, 368 |
| Imaginary numbers | 29 |
| Imperative Programming | 22 |
| in | 514 |
| includes | 369 |
| indefinite | 33 |
| index | 370 |
| indices | 371 |
| <i>infinity</i> | 30 |
| infix | 372 |
| insert | 374 |
| inside | 373 |
| insort | 375 |
| instance | 54 |
| instantiate | 376 |
| integer | 27, 377 |
| interleave | 378 |
| intersection | 379 |
| intersects-p | 380 |
| interval | 31, 381 |
| into | 382 |
| intrinsics | 51 |
| invoke | 383 |
| is | 384 |
| jiffy | 31 |
| junction | 385 |
| key | 387 |
| keys | 388 |
| keywords | 389 |
| knew | 390 |
| knowledge | 391 |
| known | 393 |
| lacks | 394 |
| last | 395 |
| left | 396 |
| let | 38, 397 |
| lexemes | 398 |
| Lexical Scope | 36 |
| lexicon | 47, 399 |
| lexicons | 400 |
| license | 401 |
| like | 402 |
| list | 52, 403 |
| literal | 41, 404 |
| local | 405 |
| location | 406 |
| log | 407 |
| long | 28, 408 |
| long number | 28 |
| loop | 66, 409 |
| lowercase | 410 |
| lowercase-p | 411 |
| macro | 412 |
| macros | 413 |
| make | 414 |
| map | 415 |
| max | 416 |
| maximum | 417 |
| may | 418 |
| median | 419 |
| meets-p | 420 |
| meron | 421 |
| meronymy | 423 |
| meron-p | 422 |
| method | 424 |
| methods | 44 |
| mid | 427 |
| min | 428 |
| minimum | 429 |
| missing | 430 |
| mod | 431 |
| modular | 432 |
| module | 433 |
| Module Scope | 35 |
| modules | 51, 434 |
| moment | 32, 435 |
| more-p | 436 |
| morph | 437 |
| most | 438 |
| move | 439 |
| my | 440 |
| nall | 441 |
| nand | 443 |
| negative-p | 444 |
| neternity | 33 |
| nevery | 445 |
| new | 446 |
| next | 447 |
| ngrams | 448 |
| nil-p | 449 |
| ninfinity | 30 |
| nix | 450 |
| no | 451 |

| | |
|-------------------------------|-------------|
| none..... | 453 |
| nor | 454 |
| not | 455 |
| nothing..... | 26, 456 |
| null-p..... | 26, 457 |
| number | 27 |
| numerics | 30 |
| octal | 27 |
| odd-p | 458 |
| old..... | 459 |
| omit | 460 |
| on | 65, 461 |
| only | 462 |
| open..... | 463 |
| or | 464 |
| out | 465 |
| overlaps-p | 466 |
| pad..... | 467 |
| parameters | 58 |
| partition | 468 |
| path..... | 469 |
| pattern..... | 55, 470 |
| pause | 471 |
| perform..... | 472 |
| pi | 473 |
| pick..... | 474 |
| pipe | 475 |
| pop..... | 476 |
| posit | 477 |
| position | 478 |
| positions | 479 |
| positive-p | 480 |
| premise..... | 481 |
| probability..... | 482 |
| procedure | 483 |
| proceed..... | 485 |
| product | 486 |
| prototype..... | 43 |
| punctuation-p | 487 |
| push | 488 |
| put | 489 |
| qualified | 490 |
| qualifiers | 491 |
| quantify..... | 492 |
| quantity | 493 |
| radians | 494 |
| radix..... | 27, 495 |
| random | 496 |
| range | 497 |
| rational | 29, 498 |
| read..... | 499 |
| Read Evaluate Print Loop..... | 22 |
| ready-p..... | 500 |
| real..... | 29, 501 |
| Real numbers | 29 |
| reasoning | 502 |
| reclaim | 503 |
| reduce..... | 504 |
| reference..... | 505 |
| references | 35 |
| relation | 506 |
| relations | 508 |
| relatum..... | 42 |
| release..... | 509 |
| remove | 510 |
| repeat..... | 67, 511 |
| replace..... | 512 |
| require..... | 513 |
| reserved | 51 |
| reset | 515 |
| resize | 516 |
| resource | 45 |
| rest | 517 |
| Restricted Scope..... | 37 |
| retract | 518 |
| return | 72, 519 |
| reverse | 522 |
| right | 523 |
| rip | 524 |
| round | 525 |
| rule | 526 |
| rules..... | 530 |
| same | 531 |
| scale | 532 |
| scatter | 76, 533 |
| scope | 534 |
| seal | 535 |
| secant | 536 |
| seconds | 31, 32, 537 |
| securelink | 538 |
| seek | 539 |
| self | 540 |
| separator | 541 |
| sequence | 52 |
| series | 542 |
| service | 543 |
| set..... | 40, 544 |
| sever | 545 |
| shuffle | 546 |
| sigma | 547 |
| signal | 71, 548 |
| significand | 549 |
| signum..... | 550 |
| sine | 551 |
| so | 552 |
| socket | 553 |
| some | 554 |
| sort | 555 |
| split | 557 |
| start | 558 |
| starts-p | 559 |
| statistics | 560 |
| step..... | 67, 561 |
| stream | 562 |
| string | 53, 563 |
| structure..... | 564 |

| | | | |
|---------------------|---------|------------------------------|---------|
| structures..... | 41, 566 | Tuples | 54 |
| sub | 567 | unbound | 615 |
| subset-p | 568 | unbound-p..... | 616 |
| substitute..... | 569 | undefined | 30 |
| subsumes-p..... | 570 | unicode..... | 617 |
| sum | 571 | union | 618 |
| summation..... | 572 | unique | 619 |
| supply | 573 | unless | 620 |
| suppose..... | 574 | unlike..... | 621 |
| survey | 579 | unqualified | 622 |
| swap..... | 582 | unseal | 623 |
| symbol | 580 | unsigned..... | 30, 624 |
| Symbol Scoping..... | 35 | until | 68, 625 |
| symbols..... | 34, 581 | unwrap | 626 |
| take | 583 | uppercase..... | 627 |
| takeable | 584 | uppercase-p..... | 628 |
| tally | 585 | url..... | 629 |
| tangent | 586 | use | 630 |
| tarry | 587 | User Defined Functions | 57 |
| task | 49, 588 | User Defined Macros..... | 63 |
| Task Scope | 37 | using | 631 |
| tasks..... | 590 | uuid | 614 |
| taxon | 591 | values | 632 |
| taxonomy..... | 593 | var | 38, 633 |
| taxon-p..... | 592 | variant | 26 |
| tell..... | 594 | vector | 634 |
| temporal | 33 | vectors..... | 53 |
| there | 596 | version..... | 635 |
| thing..... | 26 | void-p | 636 |
| this | 597 | wait | 637 |
| thought | 47, 598 | which | 638 |
| thunk..... | 599 | while | 68, 643 |
| thunks..... | 601 | whitespace-p | 644 |
| tick | 32, 602 | with | 645 |
| tie | 39, 603 | within | 651 |
| time..... | 31, 604 | wrap | 652 |
| top | 606 | write | 653 |
| transfer | 607 | xnor | 654 |
| traverse..... | 608 | xor | 655 |
| trim | 609 | yield | 656 |
| truncate | 610 | zap | 657 |
| try | 65, 611 | zero-p | 658 |
| tuple..... | 613 | | |

