Proto Language Reference

By the authors of MIT Proto

Last Updated: December 14, 2010

This is a reference to all of the currently working functions in the Proto language. For a reference of commonly used simulator and language commands, see the **Proto Quick Start**. For a tutorial on the Proto language, see the document **Thinking In Proto**. For installation instructions, see the **README**. For a user manual for the simulator, see the **Proto Simulator User Manual**. For information on how to extend the functionality of the simulator, see the **MIT Proto Developers Guide**.

This reference is intended as a "dictionary" to allow Proto programmers to look up whether the function they are looking for already exists, and how to use it. This reference guide is organized by groups of functionality. It gives only minimal explanation of the language, assuming that the programmer already understands the basics.

A note on implementation: some functions are implemented directly by the Proto kernel, others are implemented by a mixture of kernel functions and compiler pattern rewriting, and yet others are written in Proto as part of the core library (`lib/core/`). This document does not distinguish between these implementation decisions.

1 Credits for Proto

The Proto language was created in partnership by Jonathan Bachrach and Jacob Beal. As they created the language, Jonathan created the first implementation of MIT Proto, including the first compiler, kernel, simulator, and embedded device implementations. Since that time, Jake and other contributors have built on the work begun by Jonathan.

MIT Proto also includes contributions from (alphabetically): Aaron Adler, Geoffrey Bays, Anna Derbakova, Nelson Elhage, Takeshi Fujiwara, Tony Grue, Joshua Horowitz, Tom Hsu, Kanak Kshetri, Prakash Manghwani, Dustin Mitchell, Omar Mysore, Maciej Pacula, Hayes Raffle, Dany Qumsiyeh, Omari Stephens, Mark Tobenkin, Ray Tomlinson, Kyle Usbeck, Dan Vickery

The Protobo platform code in platforms/protobo/ also includes Topobo-related code from (alphabetically): Mike Fleder, Limor Fried, Josh Lifton, Laura Yip

2 Notation

Functions and special forms in this document are specified in a pattern language closely related to the quasiquote metasyntax used in LISPs.

- `.name` means *name* is a variable that matches only identifiers.
- `,name` means *name* is a variable that matches any expression.
- `,@name` means *name* is a variable that matches a list of expressions, at least one expression long.
- `...` indicates zero or more of the preceding pattern element.
- `++` indicates one or more of the preceding pattern element.

- `var|`*type* means that *var* must be of data type *type*. If there is no type specified, it means the variable can be any type.

Throughout the document Proto functions and special forms are expressed as:

`pattern`$\rightarrow$ *type*
description

where *type* is the return type.

For example, consider this definition of the `elt` function:

`(elt ,tuple|`T `,i|`S`)`$\rightarrow L$

This takes two arguments. The first is of type **Tuple** is is named, unenterprisingly, "tuple." The second is of type **Scalar** and is named "i." The function returns a **Local** value.

3 Evaluation

Proto is a purely functional language. Proto is written using s-expressions in a manner very similar to Scheme. Evaluating a Proto expression produces a program: a dataflow graph that may be evaluated against a space to produce an evolving field of values at every point on the space.

4 Data Types

All Proto expressions produce fields that map every point in space to a value. The values produced are categorized into the type system shown in Figure 1.

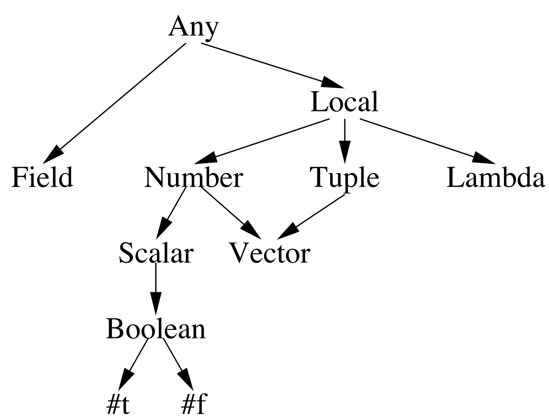

Figure 1: Proto data types; arrows indicate subset relationships.

- **Any** is the top type, encompassing all other types.
- **Field** is a function mapping a subset of the device's neighborhood to **Local** values.
- **Local** is any non-**Field** value: a **Number**, **Tuple**, or **Lambda**.
- **Lambda** is a function.
- **Tuple** is an ordered set of k **Local** values.
- **Scalar** is a floating point number (including the special values `inf/Inf`, `-inf/-Inf`, and `nan/NaN`).

- **Vector** is a **Tuple** of **Scalar** values.
- **Number** is a **Scalar** or **Vector**.
- **Boolean** is a **Scalar** interpreted as a logical value. True is any non-zero value, canonically **#t**, which is represented by 1; False, canonically **#f**, is represented by 0.

Throughout this document, types will be abbreviated by their first letter, except for **Lambda**, which will be abbreviated as λ. Tuples and vectors may be further specified by a subscript indicating length (e.g. T_3 is a tuple of 3 elements), and tuples may also have their types specified as a parenthetical list (e.g. $T(T_2, S)$ is a tuple of a tuple of 2 elements and a scalar). Fields may be further specified by a subscript indicating type of the range (e.g. F_V is a field of vectors).

Some functions require that their arguments have the "same type." The equivalence classes of type are:

- All **Scalar** values.
- All **Lambda** values. *Not working!*
- **Tuples** with equivalent values (implying the same number as well). *There are known bugs.*
- **Fields** with equivalent values.

Right now, lambdas are not first-class data types: they can only be passed around and manipulated under certain circumstances.

`(null ,expr)`$\rightarrow A$
Syntactic operator. Returns a "null" form of the value that would be returned by `expr`. `expr` is not evaluated. For scalars, this is zero, for tuples, a tuple of the same structure with zeros in place of every scalar, for fields a field of null values, and for lambdas it is undefined.

`(quote ,form)`$\rightarrow A$
Syntactic operator. A LISP quote operation. Identity on scalars, turns symbols into indexes into a symbol table, and fails on tuples. *Not working!*

5 Namespaces and Bindings

Proto is a lexically scoped language. Names are case sensitive (though this is not implemented until the neocompiler). Bindings contain values and are looked up by name. Lexical bindings are visible only within the scope in which they are bound, and shadow bindings of the same name from enclosing scopes.

When the Proto compiler encounters an unknown identifier *name*, it searches its path for a file named *name*`.proto`. If it finds such a file, then it loads the contents of the file and looks up the identifier again. Definitions in subdirectories can be accessed with identifiers of the form *dir*/*name*.

`(def .name (.arg ...) ,@body)`$\rightarrow A$
Syntactic operator. Define a function `name` in the current scope, with as many arguments as there are `arg` identifiers. The body is evaluated within an extended scope where the `arg` identifiers are bound to arguments to the function.

`(let ((.var ,value) ...) ,@body)`$\rightarrow A$
Syntactic operator. Extends scope, binding all `var` identifiers to their associated `value` in parallel. The `body` is evaluated in the extended scope.

`(let* ((.var ,value) ...) ,@body)`$\rightarrow A$
Syntactic operator. Extends scope, binding each `var` identifier to its associated `value` in sequence, so that later `value` expressions can use earlier `var` identifiers. The `body` is evaluated in the extended scope.

6 Control Flow

`(all ,@forms)`$\rightarrow A$
All `forms` are evaluated in parallel and the value of the last form returned.

`(seq ,form|`$T(L,B)$ `++)`$\rightarrow T(L,B)$
Syntactic operator. A concatenation of streams: each form returns a tuple of two elements: the first is the value of the stream, and the second is a boolean indicating whether there are more values in the stream. When the second element is false, the sequence advances to the next stream, looping back to the first after the last stream. `seq` returns a tuple where the first element is the current stream's value and the second is a boolean that is true during the first loop and false thereafter. *Capable of violating the continuous space/time abstraction.*

`(loop ,form|`$T(L,B)$ `++)`$\rightarrow L$
Syntactic operator. Identical to `seq`, except that only the value is returned.

`(mux ,test|`B `,true ,false)`$\rightarrow A$
Evaluates both `true` and `false` expressions. When `test` is true, returns the result of the `true` expression, otherwise returns the result of the `false` expression. The `true` and `false` expressions must return the same type.

`(if ,test|`B `,true ,false)`$\rightarrow A$
Syntactic operator. Restricts execution to subspaces based on `test`. Where `test` is true, the `true` expression is evaluated; where `test` is false, the `false` expression is evaluated. The `true` and `false` expressions must return the same type. The function `where` is an alias for `if`

`(select ,nth|`S `,form ++)`$\rightarrow A$
Syntactic operator. A multi-way `if`, evaluating the ith form in the subspace where `nth` $= i$ (counting from zero). Where `nth` is not non-negative integer or is greater than the number of forms, a `null` value is returned instead. All `form` expressions must return the same type.

`(cond (,test|`B `,@body) ...)`$\rightarrow A$
Syntactic operator. Another multi-way `if`, evaluating the ith body in the subspace where the ith test is true and all previous tests are false. *Not working!*

`(case ,val|`S `(,key|`S `,form) ...)`$\rightarrow A$
Syntactic operator. Another multi-way `if`. The `key` expressions must all be literal numbers, and a key's associated form is evaluated in the subspace where `val` = `key`. All `form` expressions must return the same type. *There are known bugs.*

Unimplemented: The following functions have previously been specified, but are not currently implemented: `when`, `unless`.

7 Lambdas

`(fun (.arg ...) ,@body)`$\rightarrow \lambda$
Syntactic operator. Creates an anonymous function with as many arguments as there are `arg` identifiers. The body is evaluated within an extended scope where the `arg` identifiers are bound to arguments to the function.

`(apply ,f|`λ `,args|`T`)`$\rightarrow A$
Call function `f` with arguments bound to the elements of `args`. The arity of the function and the length of `args` must be the same.

`(id ,expr)`$\rightarrow A$
The identity function: returns the value of `expr`.

8 State

Because Proto is a purely functional language, we create state using feedback loops. A state variable is initialized at some value, then evolves that value forward in time. In regions where the feedback loop is not evaluated, the state variable is reinitialized, resuming evolution when the feedback loop begins to be evaluated again.

For example, the expression:

```
(rep t 0 (+ t (dt)))
```

creates a timer that returns how long evaluation has been proceeding at each device.

`(dt)`$\rightarrow S$
Returns the time between steps in evaluating a program.

`(set-dt step|`S`)`$\rightarrow S$
Requests that the time between steps in evaluating a program be no longer than `step`. *Experimental: behavior may be flawed and may be changed without warning.*

`(letfed ((.var ,init|`L `,evolve|`L`) ...)  ,@body)`$\rightarrow L$
Syntactic operator. Creates a state variable for each `var`. `var` is initially bound to the value of expression `init`, and at each time step the state is evolved forward using expression `evolve`. The body is evaluated within an extended scope including the state variables.

In the `evolve` expression, each `var` is bound to an old value and `(dt)` is set to the time since the last step. All `init` and `evolve` expressions are evaluated in parallel, so no variable can reference another value in its `init`, but variables can use one another's old values in their `evolve` statements. *Capable of violating the continuous space/time abstraction.*.

`(rep .var ,init|`L `,evolve|`L`)`$\rightarrow L$
Syntactic operator. Create a single feedback variable and return its value. Equivalent to `(letfed ((.var ,init ,evolve)) .var)`. *Capable of violating the continuous space/time abstraction.*

`(fold-time ,f|`λ `,init|`L `,val|`L`)`$\rightarrow L$
Accumulate a value `val` across time, starting with value `init` and accumulating using function `f`. Equivalent to `(rep r ,init (,f r ,val))` *Capable of violating the continuous space/time abstraction.*

`(all-time ,expr|`B`)`$\rightarrow B$
Returns false if `expr` was ever false, true otherwise.

`(any-time ,expr|`B`)`$\rightarrow B$
Returns true if `expr` was ever true, false otherwise.

`(max-time ,expr|`S`)`$\rightarrow S$
Returns the upper limit of values for `expr` to present.

`(min-time ,expr|`S`)`$\rightarrow S$
Returns the lower limit of values for `expr` to present.

`(int-time ,expr|`S`)`$\rightarrow S$
Returns the integral of `expr` over time, starting from zero.

`(once ,expr)`$\rightarrow A$
Syntactic operator. Evaluates `expr` once, then always returns that value without evaluating `expr` again.

9 Logical

There are two types of logical operators, reflecting the difference between `if` and `mux`.

`(and ,x|`B `,y|`B`)`$\rightarrow B$
Syntactic operator. The expression `y` is only evaluated if `x` is true. Equivalent to `(if ,x ,y #f)`

`(or ,x|`B `,y|`B`)`$\rightarrow B$
Syntactic operator. The expression `y` is only evaluated if `x` is false. Equivalent to `(if ,x ,x ,y)`

`(muxand ,x|`B `,y|`B`)`$\rightarrow B$
Both expressions are always evaluated. Equivalent to `(mux ,x ,y #f)`

`(muxor ,x|`B `,y|`B`)`$\rightarrow B$
Both expressions are always evaluated. Equivalent to `(mux ,x ,x y)`

`(not ,x|`B`)`$\rightarrow B$
Returns `#t` if x is false, otherwise returns `#f`. Equivalent to `(if ,x #t #f)`.

`(xor ,x|`B`)`$\rightarrow B$
Returns `#f` if x and y have the same value, otherwise returns `#f`. Equivalent to `(if , x, (not y), y)`.

10 Numbers

Some numerical functions are generic to both vectors and scalars, others are defined for only one or the other.

Constants Note that the syntax of constants varies between the paleo- and neo-compiler. In the paleo-compiler, constants are defined by functions, and the syntax is:

`(inf)`$\rightarrow S$
Returns the floating point value for positive infinity.

`(e)`$\rightarrow S$
Returns the floating point value for the constant e.

`(pi)`$\rightarrow S$
Returns the floating point value for the constant π.

In the neo-compiler, constants are not functions and have the following syntax:

`inf`$\rightarrow S$
Returns the floating point value for positive infinity.

`e`$\rightarrow S$
Returns the floating point value for the constant e.

`pi`$\rightarrow S$
Returns the floating point value for the constant π.

Arithmetic

`(+ ,x|`N` ,y|`N` ++)`$\rightarrow N$
Adds two or more numbers of the same type. The vector version can also be called as `vadd`. If the two vectors do not have the same number of elements, the "missing elements" in the smaller vector are considered to be zero.

`(- ,x|`N` ,y|`N`)`$\rightarrow N$
Subtracts `y` from `x`. Requires numbers of the same type. The vector version can also be called as `vsub`. If the two vectors do not have the same number of elements, the "missing elements" in the smaller vector are considered to be zero.

`(neg ,x|`S`)`$\rightarrow S$
Returns the negation of `x`.

`(* ,x|`S` ++ ,y|`N`)`$\rightarrow N$
Multiplies numbers together. If the last is a vector, then it performs scalar multiplication. The vector version can also be called as `vmul`.

`(/ ,x|`S` ,y|`S`)`$\rightarrow S$
Divides `x` by `y`.

`(mod ,num|`S` ,divisor|`S`)`$\rightarrow S$
Returns the remainder when `num` is divided by `divisor`. If `num` is negative, the remainder will be negative.

`(pow ,x|`S`) ,y|`S`)`$\rightarrow S$
Returns $x^{\mathtt{y}}$.

`(exp ,x|`S`)`$\rightarrow S$
Returns $e^{\mathtt{x}}$.

`(log ,x|`S`)`$\rightarrow S$
Returns the natural log of `x`.

`(log10 ,x|`S`)`$\rightarrow S$
Returns the base-10 log of `x`.

`(logN ,x|`S`) ,n|`S`)`$\rightarrow S$
Returns the base-`n` of `x`.

`(floor ,n|`S`)`$\rightarrow S$
Returns the largest integer value not greater than `n`.

`(ceil ,n|`S`)`$\rightarrow S$
Returns the smallest integer value greater than `n`.

`(max ,x|`N` ,y|`N`)`$\rightarrow N$
Compares `x` and `y` and returns the maximum. Numbers must be of the same type. Vectors are compared lexicographically.

`(min ,x|`N` ,y|`N`)`$\rightarrow N$
Compares `x` and `y` and returns the minimum. Numbers must be of the same type. Vectors are compared lexicographically.

`(denormalize ,x|`S` ,newmin|`S` ,newmax|`S`)`$\rightarrow S$
Denormalizes (Rescales) `x`, a value between 0 and 1 to a value between `newmin` and `newmax`. This function supersedes the old "units" function.

`(denormalizeN ,x|`S` ,oldmin|`S` ,oldmax|`S` ,newmin|`S` ,newmax|`$S\rightarrow S$
More general version of denormalize, allows you to set arbitrary boundaries.

Comparison & Related Convenience Functions

`(= ,x|`S `,y|`S`)`$\rightarrow B$
Returns true iff `x` is equal to `y`. Vectors are considered to be equal iff every element is the same.

`(< ,x|`S `,y|`S`)`$\rightarrow B$
Returns true iff `x` is less than `y`. Vectors are compared lexicographically.

`(> ,x|`S `,y|`S`)`$\rightarrow B$
Returns true iff `x` is greater than `y`. Vectors are compared lexicographically.

`(<= ,x|`S `,y|`S`)`$\rightarrow B$
Returns true iff `x` is not greater than `y`. Vectors are compared lexicographically.

`(>= ,x|`S `,y|`S`)`$\rightarrow B$
Returns true iff `x` is not less than `y`. Vectors are compared lexicographically.

`(is-zero ,x|`S`)`$\rightarrow B$
Returns true if `x` is zero.

`(is-neg ,x|`S`)`$\rightarrow B$
Returns true if `x` is negative.

`(is-pos ,x|`S`)`$\rightarrow B$
Returns true if `x` is positive.

Trigonometric and Other Common Functions

`(sqrt ,n|`S`)`$\rightarrow S$
Returns the square root of `n`.

`(abs ,n|`S`)`$\rightarrow S$
Returns the absolute value of `n`.

`(sin ,n|`S`)`$\rightarrow S$
Returns the sine of `n` (in radians).

`(cos ,n|`S`)`$\rightarrow S$
Returns the cosine of `n` (in radians).

`(tan ,n|`S`)`$\rightarrow S$
Returns the tangent of `n` (in radians).

`(asin ,n|`S`)`$\rightarrow S$
Returns the arcsine of `n` (in radians).

`(acos ,n|`S`)`$\rightarrow S$
Returns the arccosine of `n` (in radians).

`(atan2 ,y|`S `,x|`S`)`$\rightarrow S$
Returns the two-argument arc-tangent of `x` and `y` (in radians). Note: Range is -

`(sinh ,n|`S`)`$\rightarrow S$
Returns the hyperbolic sine of `n`.

(cosh ,n|S)$\rightarrow S$
Returns the hyperbolic cosine of n.

(tanh ,n|S)$\rightarrow S$
Returns the hyperbolic tangent of n.

(asinh ,n|S)$\rightarrow S$
Returns the inverse hyperbolic sine of n.

(acosh ,n|S)$\rightarrow S$
Returns the inverse hyperbolic cosine of n.

(atanh ,y|S ,x|S)$\rightarrow S$
Returns the inverse hyperbolic tangent of n.

(rnd ,min|S ,max|S)$\rightarrow S$
Returns a constantly changing random number between min and max. *Capable of violating the continuous space/time abstraction.*

(rndint ,n|S)$\rightarrow S$
Returns a constantly changing random integer in the range $[0, \mathtt{n} - 1]$. *Capable of violating the continuous space/time abstraction.*

Vectors

(vdot ,a|V ,b|V)$\rightarrow S$
Returns the dot product of vectors a and b.

(vlen ,v|V)$\rightarrow S$
Returns the length of vector v, equivalent to (sqrt (vdot ,v ,v)).

(normalize ,v|V)$\rightarrow V$
Normalizes v to have the same direction, but length 1. If v is the zero vector, it remains the zero vector.

(polar-to-rect ,v|V_2)$\rightarrow V_2$
Converts a 2D vector from polar to rectangular coordinates.

(rect-to-polar ,v|V_2)$\rightarrow V_2$
Converts a 2D vector from rectangular to polar coordinates.

(rotate ,angle|S ,v|V_2)$\rightarrow V_2$
Rotates a 2D vector v by angle radians, assuming rectangular coordinates.

Missing Functions The following functions should be implemented, but currently are not, for no particular reason: ~= (not equal), rem (remainder), pos?, zero?, neg?, units (rescaling numbers).

11 Tuples

(tuple ,v|L ++)$\rightarrow T$
Creates a tuple with the set of v arguments as its elements. The function tup is an alias for tuple.

`(len ,tuple|`T`)`$\rightarrow S$
Returns the length of the `tuple`. *There are known bugs.*

`(elt ,tuple|`T `,i|`S`)`$\rightarrow L$
Returns the `i`th element of `tuple`, counting from zero.

`(nul-tup)`$\rightarrow T_0$
Creates a zero-length tuple.

`(map ,f|`λ `,tuple|`T`)`$\rightarrow T$
Return a tuple created by applying the one-argument function `f` to each element of `tuple`. *Not working!*

`(fold ,f|`λ `,base|typeL ,tuple|`T`)`$\rightarrow L$
Reduce *tuple* to a single value by folding in elements to the `base` value one at a time using function `f`. The first argument of `f` is the accumulation, the second is the tuple element.

`(slice ,tuple|`T`, start|`S `,tuple|`S`)`$\rightarrow T$
Returns a new tuple obtained by slicing the input tuple from start (inclusive) to stop (exclusive). *There are known bugs.*

`(1st ,tuple|`T`)`$\rightarrow L$
Returns the first element of `tuple`.

`(2nd ,tuple|`T`)`$\rightarrow L$
Returns the second element of `tuple`.

`(3rd ,tuple|`T`)`$\rightarrow L$
Returns the third element of `tuple`.

`(find ,value|`S `,tuple|`V`)`$\rightarrow B$
Returns true if the number `value` is an element of `tuple`.

`(position, value|`S `,tuple|`V`)`$\rightarrow S$
Returns index of `value` in `tuple`. If the item is not in the list, `-1` is returned. If it occurs more than once, the last of the match is returned. *There are known bugs.*

`(assoc ,value|`S `,tuple|`T`)`$\rightarrow T(S,S)$
Searches for `value` in the first element of each element of `tuple`. If at least one element of `tuple` matches, return the last element that matches. There must be at least one element in `tuple`, and all of its elements must be of the form $T(S,S)$. *There are known bugs.*

12 Structures

Structures are just an assignment of names to tuples.

`(defstruct .name .parent .field ++)`$\rightarrow B$
Syntactic operator. Defines a constructor and reader functions for structures of type `name`. The constructor, `new-`*name*, takes the fields as arguments and returns a tuple containing the fields. The readers, named *name*-*field*, take a tuple and return the element corresponding to `field`.

The `parent` identifier is intended to support inheritance between structures, but this is not yet implemented.

For example,

```
(defstruct foo 0 a b)
```

expands into three statements:

```
(def new-foo (a b) (tup a b))
(def foo-a (foo) (elt foo 0))
(def foo-b (foo) (elt foo 1))
```

13 Neighborhoods

There are two types of neighborhood functions: functions that create fields, and functions that summarize fields into local values. In between, any pointwise function can be applied to fields, producing a field whose values are the result of applying the pointwise operation to the values of the input fields.

Field Functions

`(nbr ,expr|`L`)`$\rightarrow F$
Returns a field mapping neighbors to their values of `expr`. *Note: it can only be called from within a "Summary Function."*

`(nbr-range)`$\rightarrow F_S$
Returns a field of distances to neighbors.

`(nbr-angle)`$\rightarrow F_S$
Returns a field of bearings to neighbors.

`(nbr-lag)`$\rightarrow F_S$
Returns a field of time lags to neighbors.

`(nbr-vec)`$\rightarrow F_V$
Returns a field of vectors to neighbors, in local coordinates.

`(is-self)`$\rightarrow F_B$
Returns a field that is true at the device and false at every other point in its neighborhood.

`(infinitesimal)`$\rightarrow F_S$
Returns a field of the density of area at each neighbor, for use in integrals.

Summary Functions

`(min-hood ,expr|`F_N`)`$\rightarrow N$
Returns the lower limit of values in the range of `expr`.

`(min-hood+ ,expr|`F_S`)`$\rightarrow S$
Returns the lower limit of values in the range of `expr`, excluding the device itself. If there are no neighbors, returns `Inf`. *Capable of violating the continuous space/time abstraction.*

`(max-hood ,expr|`F_N`)`$\rightarrow N$
Returns the upper limit of values in the range of `expr`.

`(max-hood+ ,expr|`F_S`)`$\rightarrow S$
Returns the upper limit of values in the range of `expr`, excluding the device itself. If there are no neighbors, returns `-Inf`. *Capable of violating the continuous space/time abstraction.*

`(all-hood ,expr|`F_B`)`$\rightarrow B$
Returns false if the range of `expr` includes false; otherwise returns true.

`(all-hood+ ,expr|`F_B`)`$\rightarrow B$
Returns false if the range of `expr` includes false, excluding the device itself; otherwise returns true. *Capable of violating the continuous space/time abstraction.*

`(any-hood ,expr|`F_B`)`$\rightarrow B$
Returns true if the range of `expr` includes true; otherwise returns false.

`(any-hood+ ,expr|`F_B`)`$\rightarrow B$
Returns true if the range of `expr` includes true, excluding the device itself; otherwise returns false. *Capable of violating the continuous space/time abstraction.*

`(sum-hood ,expr|`F_N`)`$\rightarrow N$
Returns the sum of `expr` over all devices in the neighborhood. *Capable of violating the continuous space/time abstraction.*

`(int-hood ,expr|`F_N`)`$\rightarrow N$
Returns the integral of `expr` over the neighborhood.

The `fold-hood` family of functions are used to implement the other summary functions. Although they are made accessible to the user, they should be used with care as they will tend to break the abstraction barrier.

`(fold-hood ,fold|`λ `,base|`L `,value|`L`)`$\rightarrow L$
Syntactic operator. Collects `value` from each of the neighbors, then folds these into a summary value, using `fold` to combine elements into `base` one at a time. *Capable of violating the continuous space/time abstraction.*

`(fold-hood* ,fold|`λ `,base|`L `,field|`F`)`$\rightarrow L$
Starting with `base`, use the accumulator function `fold` to combine all of the values in `field`. *Capable of violating the continuous space/time abstraction.*

`(fold-hood-plus ,fold|`λ `,prep|`λ `,value|`L`)`$\rightarrow L$
Syntactic operator. Collects `value` from each of the neighbors, then applies `prep` on each value, then combines the results together one at a time using `fold`. If there are is only one value, it is returned without calling `fold`. *Capable of violating the continuous space/time abstraction. There are known bugs.*

`(fold-hood-plus* ,fold|`λ `,field|`F`)`$\rightarrow L$
Use the accumulator function `fold` to combine all of the values in `field`. *Capable of violating the continuous space/time abstraction.*

The function `mix` is an alias for `fold-hood`. *Not working!*.

14 Sensor and Actuators

Actuators reset themselves to a null value whenever they are not actively being invoked. Thus, for example,

```
(if (sense 1) (mov (tup 2)) (red (tup 1)))
```

will cause devices move to the right only when `(sense 1)` is true, and to turn on their red LED only when `(sense 1)` is false.

Note that most sensors and actuators are not part of the core Proto language, but are platform-specific. For the simulator, that means they are introduced by particular layers. For a list of all of the sensor and actuator primitives bundled with the simulator, see the **Proto Simulator User Manual**.

Movement This collection of functions are actuators for moving devices and sensors for introspecting on their motion.

`(mov ,velocity|`V`)`$\rightarrow V$
Attempt to move at `velocity`. If `velocity` is not 3 elements long, missing elements will be treated as zero and extra elements will be ignored. The return echoes `velocity`.

`(speed)`$\rightarrow S$
Returns the current speed of the device.

`(bearing)`$\rightarrow S$
Returns the current 2D bearing of the device. *Not working in 2nd generation simulator!*

Geometry

`(area)`$\rightarrow S$
Returns each device's estimate of the amount of area it represents.

`(hood-radius)`$\rightarrow S$
Returns the maximum expected range at which devices can communicate. The function `radio-range` is an alias.

Other Sensors and Actuators

`(flex ,angle|`S`)`$\rightarrow S$
Attempt to flex a Topobo joint to an angle in the range $[-\pi/2, pi/2]$. Angles outside that range will be truncated. *Not working in 2nd generation simulator!*

`(mid)`$\rightarrow S$
Returns the device's ID.

15 Library Functions

These are not primitive functions, but are frequently used building blocks which have been included in Proto's distribution library, in the directory `lib/`.

`(distance-to ,source|`B`)`$\rightarrow S$
Calculates the shortest-path distance from every device to the set of devices where `source` is true. The function `gradient` is an alias.

`(broadcast ,source|`B` ,value|`L`)`$\rightarrow L$
Flow `value` outward from devices in the `source` to all other devices. Each device takes its value from the nearest `source` device. The functions `gradcast` and `grad-value` are aliases.

`(dilate ,source|`B` ,d|`S`)`$\rightarrow B$
Returns true for every device within distance `d` of the `source`.

`(distance ,region1|`B` ,region2|`B`)`$\rightarrow S$
Calculates the distance between `region1` and `region2` and broadcasts it everywhere.

`(disperse)`$\rightarrow V_3$
Devices repel from one another using spring forces.

`(dither)` $\rightarrow V_2$
Devices wander randomly in a 2D plane.

`(elect)` $\rightarrow B$
Devices choose a leader by mutual exclusion and maintain precisely one leader within a given distance.

`(flip ,p|`S `t f)` $\rightarrow A$
Continually flip a probability `p` coin: on heads evaluate `t` and on tails evaluate `f`. `t` and `f` must be of the same type. *Capable of violating the continuous space/time abstraction.*

`(timer)` $\rightarrow S$
Return the length that this device has been evaluating this expression (i.e. not going in different branches of an `if`)

.