Beads Reference Manual

updated May, 2025 version 058 © 2020-2025 edj

Introduction

Overview

`Beads` is a new language designed to build graphical interactive software that will last decades without modification. It is a new category of language, which has the following characteristics:

Reversible - `Beads` can run both forward and in reverse at an event level, which means you can back up the state and see how the screen looked previously. This makes is much easier to find errors in logic, especially with animations. `Beads` achieves reversibility by a careful internal design that makes use of a tracked mutable state.

Deductive - `Beads` adopts some of the deductive features of PROLOG, a highly innovative language invented in the 70's, whose breakthroughs have been forgotten. Deduction lets the computer carry the burden of the most error-prone aspect of programming which is determining the proper order of calculations. In Beads it uses deduction to figure out what needs to be redrawn, based on state changes.

Closed arithmetic - `Beads` has a completely defined arithmetic that eliminates many common errors. It also offers physical units of measurement, a major convenience for engineering and scientific work.

Portable - `Beads` is designed to be independent of hardware and operating systems, and by means of automatic forward conversion and backwards compatibility, is designed to allow programs to run without having to touch the source code for decades.

Simple - `Beads` is simpler than most other languages and their associated frameworks and API's. The notation is concise, and straightforward with no tricky abstract concepts.

Basic Structure

Each Beads program consists of various modules. During compilation time, the main module and the main entry points are specified. Every program has a main initialization entry point called `main_init` which is run exactly once at the start, and a regular drawing entry point called `main_draw`, which is called on each refresh cycle.

In smaller projects you will have only one source code module. Each module has the same basic layout:

- Beads language marker
- Global definitions of enumerated values, constants, records, units, and variables
- Chunks of code representing functions for calculation and drawing

Global variables are by default tracked, which means their changes are recorded, which allows Beads programs to run both backwards in time.

The compiler is a two-pass system, where variables and global level declarations are processed before the function bodies. In the declaration area, constants must be declared before they are referenced, as there is only one pass through the declaration section.

Each indented block of code (some people call them functions or subroutines), has a prefix in front that identifies the general purpose of that chunk. For calculations you use a prefix of `calc`. For plain drawing you prefix with `draw`. For slicing part of the screen into horizontal or vertical slices you use the `slice` prefix. For drawing a two-dimensional grid you use the `grid` prefix. There are also other prefixes like `machine`, `report`, etc., see the drawing section for more details. Inside a `calc` block, you cannot draw on the screen. Inside a `draw` block you have an implied variable `b : a_block` which holds the information relating to the drawing context. There is a shortcut for the `b.box` field, `bb`, which is used so frequently we offer an abbreviation. Inside a `track` block you can refer to the block variable `b`, and there is also an implied variable `e : a_event`, which holds information about the event that was sent to your block. The implied variables save repetitive typing, and make all programs more uniform.

The first line

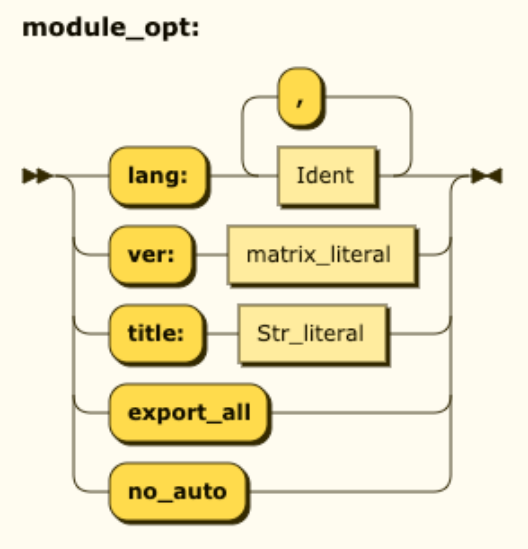
The first line must declare the level of language the file is written for, and the kind of module you are building:

```
beads 1 program my_first_game
```

By labeling each source file with the language version number, it is possible to detect and

automatically upgrade source files from earlier versions. In this way source code can last for many decades. There are four kinds of modules you can build:

a) A `program` (the most common kind, a program you wish to run)
b) A `library`, which is a collection of functions that comprise a library
c) A `monitor` program that runs on top of another program, like a debugger
d) A `system` interface module that defines the runtime standard library functions



The `lang` keyword lets you specify which internationalizations are in use. If your program is written in English, and has Spanish as a second language, you would specify `lang:LANG_ENG, LANG_ESP`. This allows the compiler to know which localizations to bundle into the final product. The string tables are built into the final file.

The `ver` keyword allows you to store the version number of the module. The version number is a matrix literal like `[1 2 3]` which would mean version `1.2.3`.

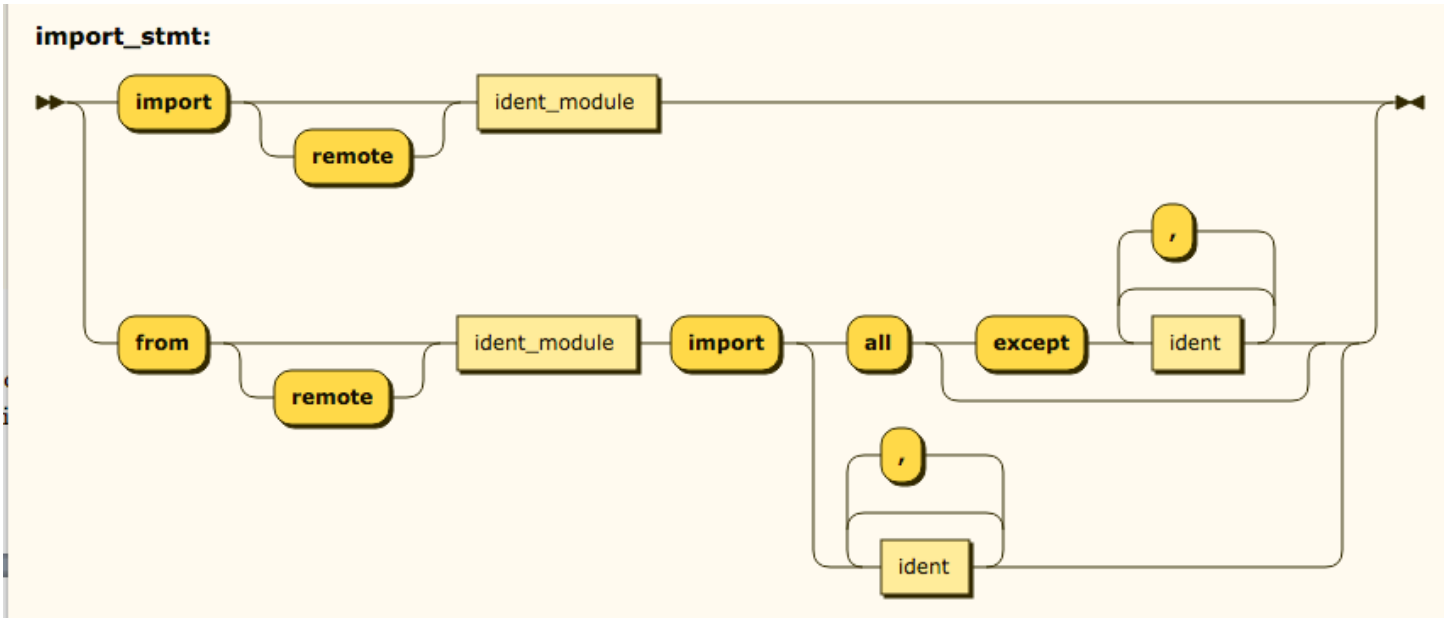
The `title` option is used for naming the top window of a web app, or the main window of a Windows program.

The `export_all` option is useful for library modules that routinely export all the top-level symbols.

The `no_auto` option is used for system modules, which sometimes wish to suppress automatic import of the `std` and `str` libraries.

Module structure

A program consists of a set of modules; one module per file. To reference constants, variables, and functions from other modules, you use the `import` statement to make those names known to the local program. The `import` statements must immediately follow the first line containing the `beads` statement, so that scanning programs can quickly build cross-references.



You can import all the exported systems of a module, or you can select specific symbols to import, or you can import all except a few symbols. The `std` and `str` system modules are automatically imported. Since Beads programs can subscribe to a `remote` server module, you use that keyword when the module going to be called will be on a remote computer. Importing a remote module will give you remote function calls, and the ability to subscribe to subsets of a server's database.

Drawing

The Beads drawing model

Each refresh cycle the Beads runtime system redraws whatever part of the screen is affected by the most recent changes to the state, by identifying which parts of the screen need redrawing. It uses deduction to determine which blocks are dirty and need refreshing, and which blocks are unaffected. This deduction is based on what state variables the code in the draw block used; if a quantity `nmonsters` changes in the global state, then any draw block that referenced `nmonsters`

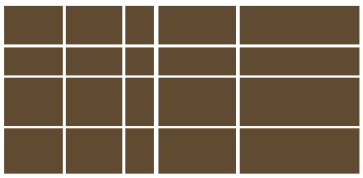
is redrawn, and any children blocks.

There are the kinds of drawing blocks:

`draw` - a simple draw function for a rectangular area
`slice` - subdivide the existing space vertically or horizontally into slices

`scroll` - a scrolling stack of items either horizontal or vertical. Very similar to slice, but instead of taking the screen size and subdividing, the horizontal or vertical height is unlimited, and a scrollbar will be drawn to allow the user to scroll the content.

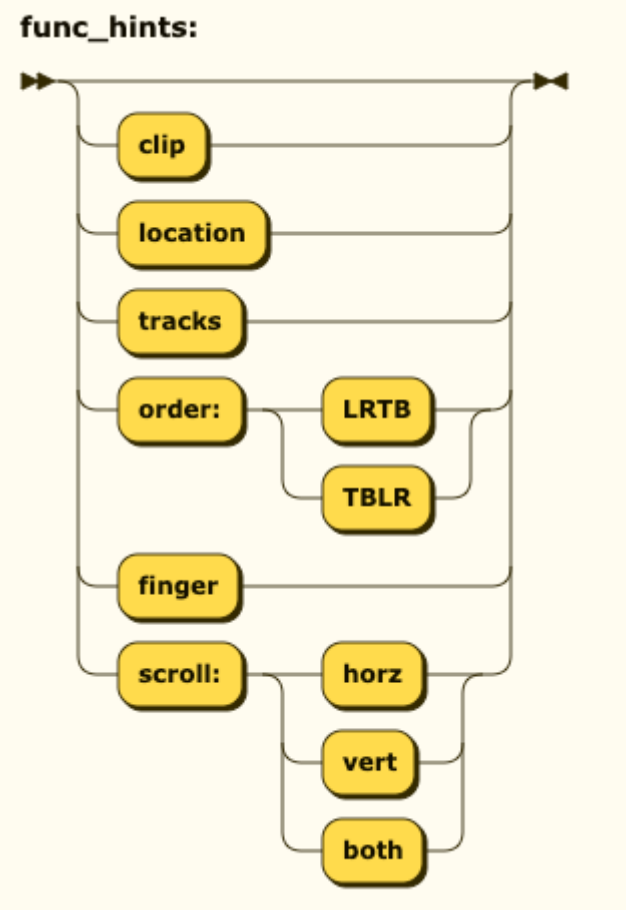
`grid` - subdivide the space horizontally and vertically into columns and rows. A grid can be sliced or scrolled in each of the two directions:

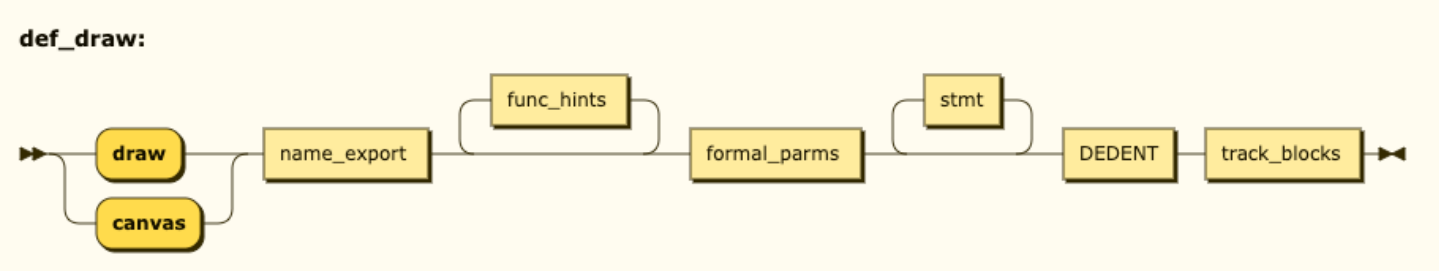


`report` - draw a complex multi-column scrolling list with an underlying horizontal alignment grid, very common in business accounting reports:

A drawing block has an optional event-tracking chunk of code prefixed by `track`. A `track` block handles events associated with the prior draw block, like taps and mouse clicks. Because blocks are laid out hierarchically, events are passed from the topmost, most recently drawn, tracking block first. A `track` block absorbs the passed event by returning `Y`. Mouse clicks and taps are specifically tied to the area of the screen that the block controls, and are passed through only to blocks that cover that area of the screen. Since keyboard events are non-specific in terms of coordinates, they are passed to all blocks starting with most recently drawn first. Once a block absorbs an event, it stops propagating to other blocks.

Draw blocks



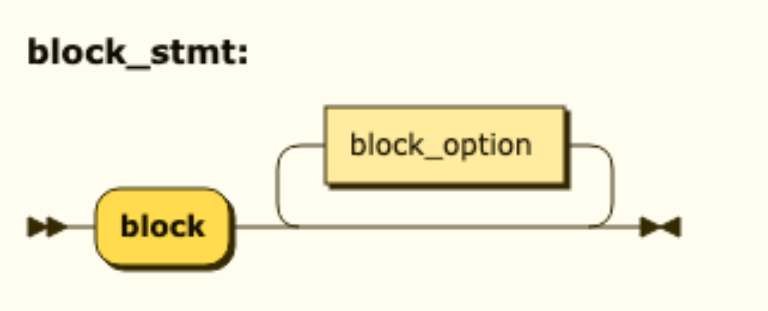


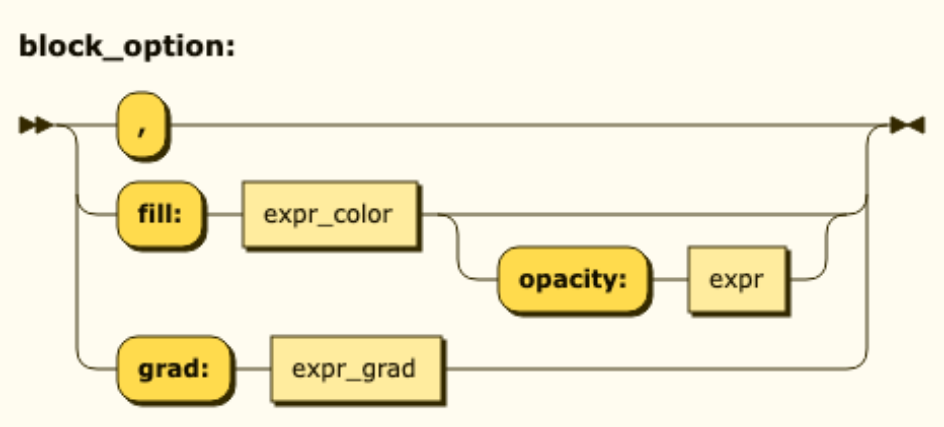
A regular `draw` block can call another draw function, and possibly add a subset layer into the block. It is the final leaf node of the screen's block structure, and doesn't further subdivide. Inside each draw block there is an implied block variable `b` that holds various fields describing the current block such as:

`bb` - a shortcut for `b.box`, the rectangle of the whole block

There is a special statement available in draw blocks that allows you to set the CSS style of the

block. You can only have one of these per block, because it is really not an executable operation, but a means of setting the CSS style of the underlying DIV block. At present it is limited to specifying a fill color, opacity, or gradient. This is the most efficient and recommended way to fill a block with a solid color or gradient.





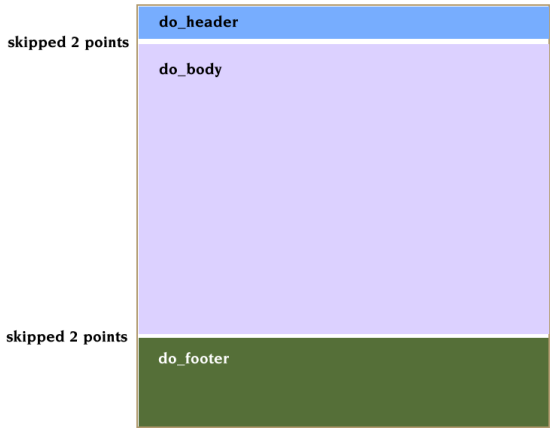
Slice/scroll blocks

A `slice` or `scroll` block takes the space allocated by the parent, and subdivides it into slices. The only difference between the `slice` and `scroll` is whether the existing space is subdivided (`slice`) or the space is considered unlimited in the direction of motion, and a scrollbar will be drawn on the right or below. Inside a `slice` block you specify whether it is to be split horizontally, or vertically. If you want to split in both directions at once, you use a `grid` block instead. The slivers when totaled fill the available space. You can either create a spacing sliver with the `skip` command, or create a sliver of content with the `add` statement, which specifies the function that will draw that sliver. The `add` and `skip` statements specify a width in pixels (`px`), points (`pt`), picas (`pc`) or aliquots (`al`). The aliquot system is similar to the method used in chemistry formulas, where you give the amounts in terms of proportions relative to each other, and they can add up to any total number. You can think of them of shares of space. For example, assume that a block called `big_block` was given a size of 400 by 400 pixels, the following block definition subdivides that vertical space and splits the area into 5 bands which add up to cover

the entire 400 pixel vertical space:

```
vert slice big_block
  add 10 al do_header
  skip 2 pt
  add 80 al do_body
  skip 2 pt
  add  20 al do_footer
```

This block would draw like:



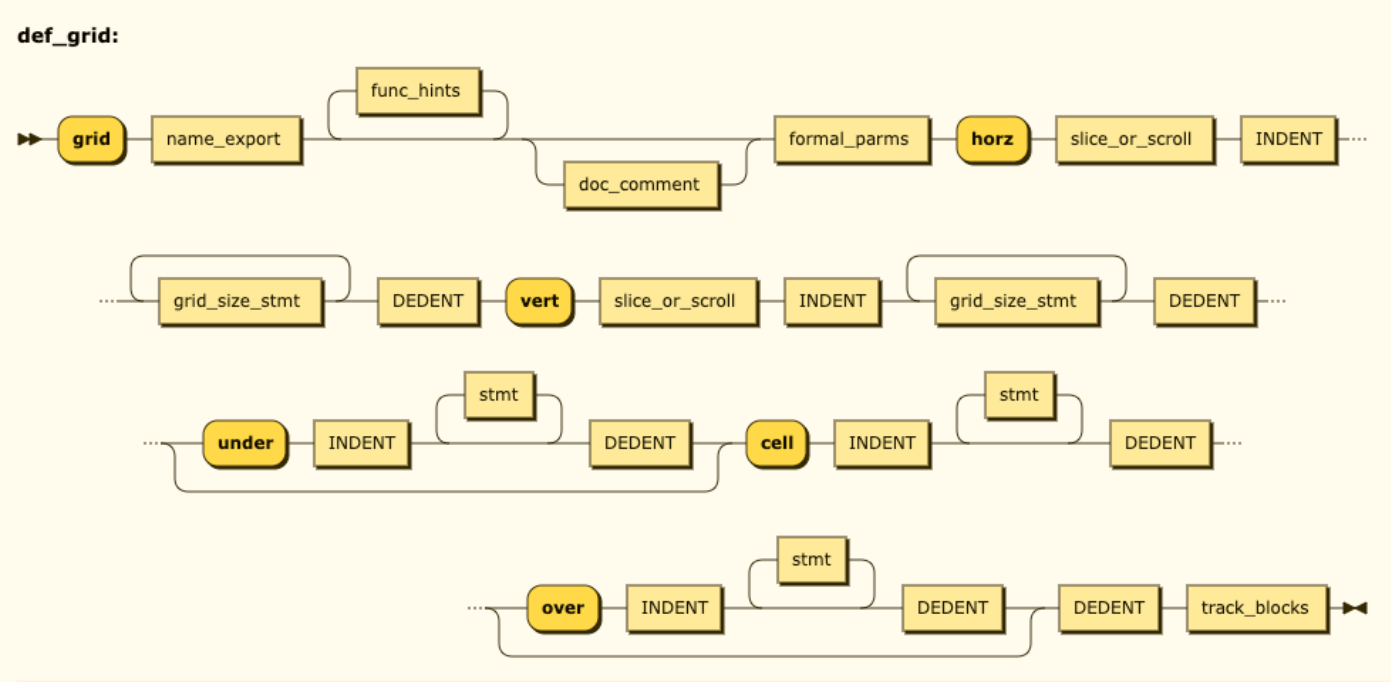
In this `big_block` definition, the system would first subtract all the fixed-size space from the total, using the current DPI to convert any point measurements into pixels. Let's assume that our device is 144 dpi, so given there are 72 points per inch, that means each point uses 2 pixels. There are 2 `skip` statements of 2 points each, totaling 8 pixels of reserved space. The available space would then be 400 - 8 = 392 pixels leftover. Then the aliquots are summed, which are 10 + 80 + 20 = 110 aliquots. The final block sizes can then be calculated using 392/110 (3.56) pixels per aliquot. The aliquot method is particularly good for allowing responsive layouts, which reflect window resizing on desktop, or the wide variety of mobile devices a mobile application will be run on. This proportional system is based on the renaissance proportional drawing system, which created many of architecture and painting masterpieces. In conjunction with the rectangle solver, it greatly eliminates the tedious and error prone space calculation issues employed in other drawing models.

Note that the `big_block` function didn't actually draw anything on the screen, but merely subdivided the space into 3 active areas, and the lower level functions `do_header`, `do_body`, and `do_footer`, will do the actual work of calling drawing primitives.

Note that the `add` statements can be inside `if` statements, so you can make a layout conditional on a flag. That makes the proportional system of extreme usefulness, as you don't have to make percentages add up or worry about unused space.

Note that the functions in the `add` statement can have parameters sent to them. The drawing system automatically calls drawing functions that are 'dirty', because they use data that has been changed. This is part of the deductive features of Beads, and thus each of these calls can happen or not in any given refresh cycle. Beads does its best to eliminate unnecessary redrawing of the screen, which can be quite slow even on today's fast computers, as rendering text and other drawing functions consume millions of CPU cycles.

Grid blocks



A grid function draws a two-dimensional grid. You first specify how the screen is to be divided in the horizontal, then the vertical. After defining the horizontal and vertical subdivisions of the grid, there is an optional `under` section, where you can draw underneath the calls, then you define the `cell` section, where each of the cells is drawn, and then an optional `over` section, where you can draw something on top of the grid.

Thus each `grid` function consists of the following sections:

```
horz slice
   … add or skip to define the columns of the grid
vert slice
   … add or skip to define the rows of the grid
under
      … optional drawing under the grid
cell
```

…. this section is called repeatedly to draw all the cells of the grid

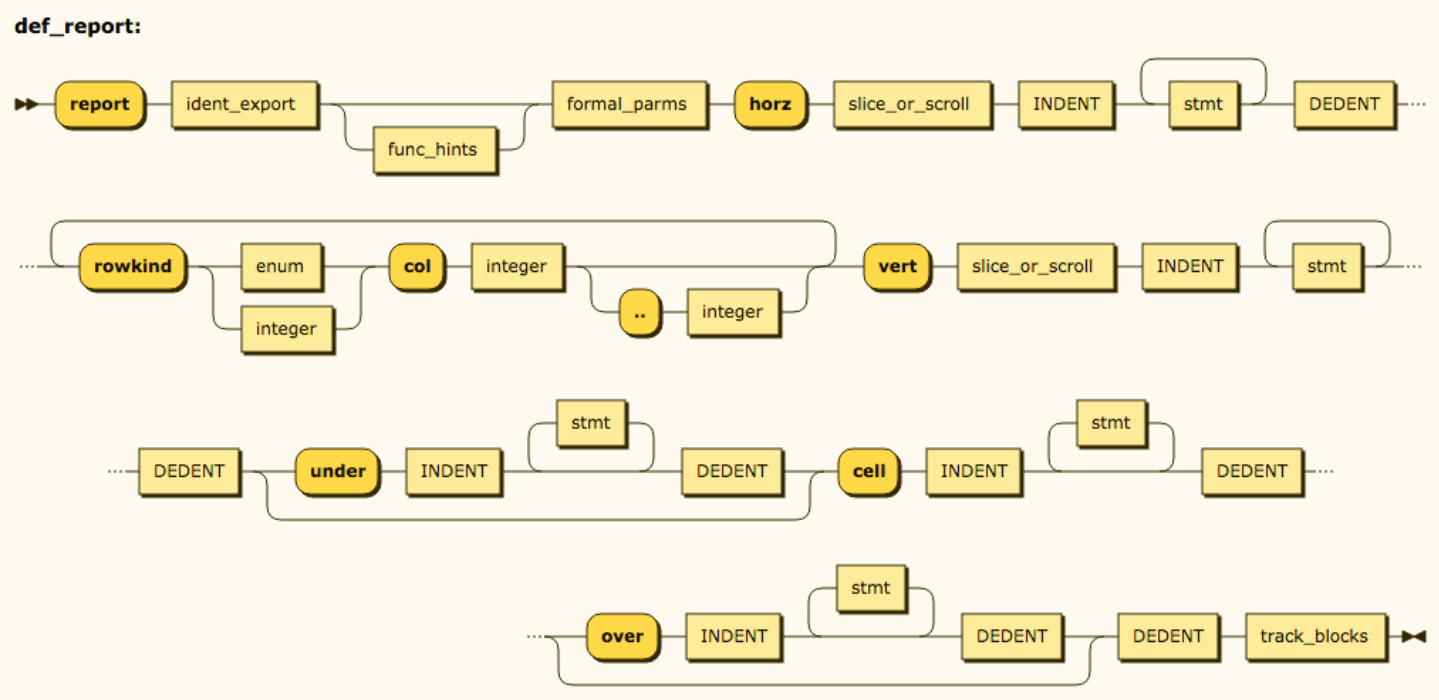
`over`

… optional section to draw on top of the grid

The `cell` section is called to draw each cell in the resulting grid. Inside the `cell` function, there is an implied block variable `b` that contains the following information as each cell is drawn:

`b.box` -- the bounding box of the cell, abbreviated by `bb`
`b.cell_seq` -- the sequential cell number, where 1 is the first cell, top down left to right
`b.cell.x` -- the column of the cell, where 1 is the first column
`b.cell.y` -- the row of the cell, where 1 is the first row
`b.cell_id.x` -- the cell columnar ID
`b.cell_id.y` -- the cell row ID

Report blocks



The report drawing block is the most complex and most powerful drawing block kind. In a `report` block you first specify a micro-grid, which sets up the vertical lines that constitute that all the columns will be aligned to. After defining a micro-grid then you specify all the row kinds, and specify which micro-grid columns each logical column of this row kind spans. After defining all the columns each row kind, then you add rows into the report. Implicitly all reports are scrolling vertical lists.

Graphic blocks

`Graphic` blocks are calculation blocks that carry the implied drawing context information variable `b`. They are occasionally needed, as some calculation routines need to be able to access block information from the caller, for example the ability to convert points to pixels, but are not actually drawing.

Sub-layers

`layer` - inside a block you may wish to overlay on top of the previous layer, change the coordinate transform, etc., so you create a sub-block inside a draw block by using the `layer` command, which allows you to create a new coordinate transformation in a sub-area of the block.

Inside drawing blocks

A function must have a `draw` prefix to be permitted to draw on the screen. Most of the code in a typical program relates to drawing. It is often 70-80% of the total lines of code, and there are special syntactical features available in a draw block. The most important one is automatic units conversion to the device space. You can specify measurements such as `pt` for points (1/72nd of an inch), or `mm` for millimeters after a size, and the program will do the conversion automatically at runtime, based on the dots per inch of the display, printed page, or perhaps print preview.

```
y += 10 pt
```

The above statement will convert 10 points into screen dots, and that value will be added to `y`. No rounding is done; many drawing systems support fractional pixels.

Inside a `draw` function there are predefined variables accessible that permit you to interrogate the state of the block you are drawing:

```
this.bounds : rect      -- the bounds of the block (rectangle)
this.bid : num          -- the unique block ID of this block
```

There are standard library functions for drawing the most common things:

```
draw_str
draw_image
draw_line
draw_circle
draw_oval
draw_rect
```

```
draw_polygon
draw_polyline
```

The runtime system has a display list for each logical window on a connected display. In each program there is one master function associated with a window that is responsible for drawing the window in its entirety. The screen is gradually subdivided into sub-rectangles, and each sub-function is responsible for drawing one sub-rectangle. If the screen needs to be redrawn due to some external event, the computer can examine the list of affected rectangles and redraw on the necessary sections. The top level drawing block is called `root`.

Example:

```
draw statusbar
  fill_solid_rectangle (this.bounds, COLOR_RED)
```

Inside a draw function, there is meta-variable called `this`, which allows you to access read-only values related to the block you are inside, most importantly the `bounds` of the block, which is a rectangle describing the total area of the block.

Drawing functions make use of the drawing primitives. Draw functions should not modify global state; they should be pure functions that only have read access to external variables. There is a draw function for each rectangular area on a screen. Screens are built from the outside in, which means the containing rectangle's size is first set, then subdivided. Tracking functions interact with the event model, so that events associated with an area are sent to the affected rectangles, and if not consumed gradually work their way upwards in the display list hierarchy. Note that the resulting display list forms a tree.

Drawing modules are identified by a pattern match of names. The list of names on the `draw` line gives you a partial path to the block in question. A block name of `empty` means leave that block transparent. A question mark character indicates a wildcard match, where any block:

```
horz slice root
   add 40 al main_left
   add 30 al main_center
   add  5 pt
   add 10 pt main_right
```

in this case, the main_left block get 40/80ths of the horizontal space allocated to it, the main_center block gets 30/80ths, there is a margin of 5 points, and the main_right block gets 10 points (fixed size).

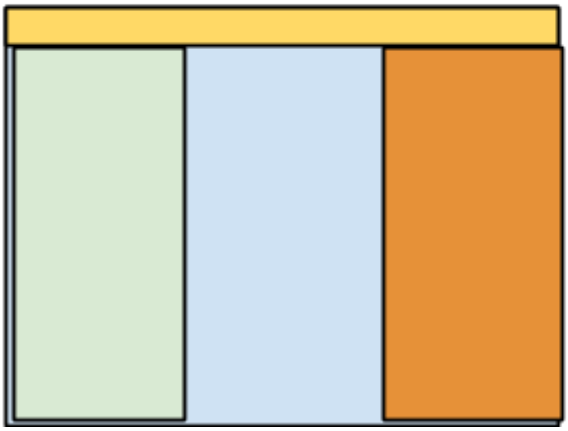
```
vert slice main_left
   add 80 pt promo_upper order 99
```

```
add 20 pt promo_middle
add 1 al promo_lower
```

in the above block, `promo_upper` got 80 points of space, `promo_middle` got 20 points, and whatever was leftover was given to `promo_lower`. There is an optional draw order to a split block, the promo_upper was given explicitly a draw order of 99 which is higher than the default draw order of 0, so the promo_upper which would normally be drawn first, is drawn last.

Step 1) Undivided rectangle.

Step 2) Divide the vertical area into menu bar, content



Step 3) Divide the content area into 3 vertical strips

Inside each draw function, the code either draws the rectangular area passed to it, or subdivides the rectangle further. The functions to subdivide are:

For example this code takes the main drawing block (`main_draw`), and splits it into two parts vertically one for the menu bar and one for the remaining area (`area1`):

```
vert slice main_draw
  add 20 pt menubar  // reserve a fixed amount of space
```

```
add 10 al area1  //  take all remaining space
```

Then we take `area1` and split it into 3 pieces horizontally:

```
horz slice area1
    add 10 al partA
    add 12 al partB
    add 10 al partC

draw partA
  //  fill and stroke with green
  fill_stroke_rect (b.box, border:2 mm border_color: black, color:green)

draw partB
  //  fill with blue
  fill_rect (b.box, border:2 mm border_color: black, color:blue)

draw partC
  //  fill with orange
  fill_rect (b.box, border:2 mm border_color: black, color:orange)
```

The block subdivision primitives consist of:

- `horz slice` -- divides a block into horizontal slices
- `horz scroll` -- divides a block into horizontal slices, but can overflow and scroll
- `vert slice`
- `vert scroll`
- `layer` -- shrinks the current drawing area on the sides
- `grid` -- creates a 2D grid of cells (similar to HTML table)
- `report` -- the classic computer report with columns and indenting

Primitive drawing functions:

- `draw_oval`
- `draw_rect`
- `draw_circle`
- `draw_line`

To draw static text on the screen:

- `draw_str`

To create a text entry field:

- `draw_input`

Tracking events for a draw block

After each drawing block, you can specify a `track` block, which absorbs events. The tracking block must appear immediately after the drawing block to be recognized. In this way the reader can know exactly which events each subsection could possibly absorb.

Inside a tracking block, if you return `Y`, then the event is absorbed, and no further blocks will see the event. Normally a draw block does not modify the mutable state of your program. The tracking block is where you cannot draw, but can update the state in response to an event. Once you update the state variables, the runtime system will deduce which sections of the screen need to be recomputed/redrawn.

All events are stored in single global array tree named `loom`. The `loom` holds the past and future events of the system. As input events occur, they are placed into the loom. The loom is a tree structure with some special internal properties. Each time you add an event to the loom, it assigns a unique integer slot id that can be used to reference the event later. Events can only be changed in the future, once the event happens and moves into the past, that part of the loom cannot be modified, and eventually is discarded. The amount of the time stored by the run time is implementation defined.

Each event record in the `loom` has at a minimum a field called `EVENT_TYPE`, and `EVENT_SCHEDULED_TIME` (which is when it is supposed to be executed), and `EVENT_ACTUAL_TIME` (when it was actually executed or U if not yet executed). There are predefined event types for keystrokes (`EV_CHAR`), mouse events (`EVT_HOVER`), and touch gestures (`EV_TAP`, etc.).

The user can insert program-generated events of type `EVT_FUNC` (delayed function call).

The events in the `loom`, which originate from the input devices attached to the computer, or from the program itself (like timer events), are sent to the block handlers, where the front-most blocks receive the events first, and the events bubble up the display chain until the master block receives the event if no sub-block absorbs it. Pointing device events are dispatched to the block they cover, unless a block has captured the mouse pointer, in which case it will get first chance to absorb the event.

The following example will call `myfunc(1,5)` 10 seconds from the moment the scheduling line is executed, and then call it 2 more times (for a total count of 3 repetitions) with 20 seconds between each call:

```
id1 = loom_timer (myfunc(1,5), delay=10 sec, reps=3, interval=20 sec)
```

The id value returned by `loom_add` is a unique ID that can be used to clear the event, or chain to it. Event chaining lets you schedule an event to occur a certain amount of time after

some other event has finished. If you wanted to schedule an event after `myfunc` finishes, you can append an event to another event. In this example, we use the `loom_add` function to schedule a call to `myfunc2` 10 seconds after the previous event with id of `id1` is finished:

```
id2 = loom_timer (myfunc2, delay=10 sec, after=id1)
```

It is extremely common to wish to execute a movement of an object across the screen. In this case you are asking for a continuous stream of calls to a function over a time period. To create an iterated event, use the `loom_add` call with the optional duration parameter. When a duration is used, the called function's first parameter is a progress indicator that will go from 0 to 1.0 , starting at the future time you specify:

```
id = loom_timer (myfunc3, duration=10 sec, curve=ease_out)
```

The above call will call `myfunc3` repeatedly over a 10 second period of time. The first parameter to `myfunc3` will be the progress value, which varies from 0 to 1.0. The `ease_out` function maps an input value from 0 to 1 representing the elapsed time, to a resulting value in the range 0 to 1. The default mapping function is linear. The number of callbacks will depend on the speed of the computer, and how long it takes the callback function to execute, as well as the run-time system. If the system runs 60 frames a second, then `myfunc3` might be called as many as 600 times, but since other processes may be processor intensive, the calls may occur at irregular intervals. You are guaranteed to get a call with the 0 time and the 100 time, so that initialization can be performed at 0 time and cleanup at 1.0 time.

This example starts an event 5 seconds after event `id1` finishes, with a duration of 10 seconds:

```
id2 = loom_timer (myfunc, after=id1, delay=5 sec, duration=10 sec,
curve=ease_out)
```

It is also very common to wish to call a timer for a certain number of repetitions, at an interval, with a starting delay. This call sets up a callback that will happen 5 times, starting 4 seconds from now, with a 10 second interval between each call:

```
id = loom_timer (myfunc, delay=4 sec, interval=10 sec, reps=5)
```

Variables used in the function call parameters for future execution are carried as pointers, so the future value of the parameter may change during the time the event is waiting to execute. For example:

```
loom_timer (myfunc (table[5], table[6]), delay=10)
```

In 10 seconds from now, the future value of `table[5]` is passed to the function. Some other

process might be incrementing `table[5]`.

NOT YET IMPLEMENTED // When you are going to execute a function and want to pass a tree structure, but don't want to use the future version of the tree, but the current version only, use parentheses around the tree expression, which will have the effect of cloning the current value of the tree and storing it into loom for future use, and ignoring the future value. This has significant performance implications, and of course should be avoided.

```
var mytree : tree = { 11, 12, 13 }
loom_timer (myfunc((mytree)), delay=10 sec)
```

Any attempt to store an event into the past will be ignored. The minimum delay is implementation dependent, but the first version has a minimum of 10 milliseconds. Execution of the loom's events is on a best-effort basis, and it is quite easy to schedule more events in the future that can be executed on time, so the actual execution time will likely lag the scheduled time.

Each event in the loom has a unique ID. For example:

```
id1 = loom_timer (myfunc (11), delay=10 sec)
id2 = loom_timer (myfunc (22), delay=10 sec)
```

When you create the event, you get back a unique id value that can be used to clear the event if desired. You can flush out an event by truncating the loom (which is a tree) using the standard truncation operation:

```
delete=> loom[id2]
```

This has the effect of deleting the future event located at internal time slot `id2`. The loom is after all a `tree` structure with integer subscripts, but the subscript does not give any clue as when the event is scheduled. There are special internal data structures that allow the loom to be efficient.

To peek at the pending events, you can iterate over the loom using a form of the `loop` statement:

```
loop across:loom index:id from:now to:now+10
```

….`id` will iterate across events scheduled for execution in the next 10 sec, in ascending order of event time…

You can traverse the events in reverse order (from most future back to now) using the `rev:` flag to go backwards through the loop over the next 10 second period:

```
loop across:loom index:id from:now to:now+10 rev:T
```

You can also find the first event that scheduled at or after a specific time:

```
id = loom_first_id (mytime)
```

You can find the next event ID by time given an existing reference number:

```
id2 = loom_next_id (id)
```

Or go backwards:

```
id3 = loom_prev_id (id)
```

If there is no next or previous event in the loom, then the return value will be `U`

Basic concepts

State variables

One of the most powerful features in Beads is the ability of the run time system to automatically determine when a section of code needs to be executed. When you declare a mutable state variable at the top level, draw blocks containing code that refer to that state variable will be automatically refreshed when that value changes. If for example, a part of the screen consists of a thermometer drawing, and the state variable `temperature` changes value, then that draw block will be automatically refreshed.

User interface buttons, sliders, widgets in general are commonly tracked with state variables, because they control the user interface that is drawn. This is the heart of the deductive model of Beads.

The state variable system is invisible in the syntax, but any variable declared at top-level scope is tracked for state changes unless it is suffixed with `no_track` after the type declaration. The tracking system allows code to be executed in reverse, and also controls automatic refresh. It operates inside the runtime system.

Built-In constants and enumerated values

There are hundreds of predefined constants in the standard library, covering the following types of values:

Numeric constants:

```
T           -- value of yes/true/on
F           -- value of no/false/off
U           -- undefined value
ERR         -- error value
INFINITY    -- positive infinity
PI
```

There are the HTML color names:

```
ALICE_BLUE ⇐ #F0F8FF
ANTIQUE_WHITE ⇐ #FAEBD7
AQUA ⇐ #00FFFF
AQUAMARINE ⇐ #7FFFD4
```

```
AZURE ⇐ #F0FFFF
```

There are key codes for keyboard events:

```
KEYCODE_LEFT    = 37
KEYCODE_RIGHT   = 39
KEYCODE_UP      = 38
KEYCODE_DOWN    = 40
```

There are lists of currencies:

```
CURRENCY_USD  "$" -- "USD"
CURRENCY_EUR  "€" -- "EUR"
CURRENCY_JPY  "¥" -- "JPY"
CURRENCY_GBP  "£" -- "GBP"
```

There are lists of languages:

```
LANG_AMH "AMH"    --  amharic
LANG_ARA "ARA"    --  arabic
LANG_BEL "BEL"    --  belarusian
LANG_BEN "BEN"    --  bengali
LANG_BUL "BUL"    --  bulgarian
```

When a variable is created, it is given the initial value of `U`. Referencing elements of an array that are not yet initialized return `U`. Calling a function pointer that is undefined has no effect (but will be trapped if checks are on). Storing a value at a subscript of `U` or `ERR` has no effect, although when run time checks are enabled this is trapped.

Identifiers

Identifiers start with an alphabetic character, and can include Unicode characters. They are case sensitive. The first character must be alphabetic. The second and later characters can be alphabetic, numeric, underscore, and include most Unicode characters. Examples of identifiers are:

```
myname      -- valid
MyName      -- identifiers are case sensitive
總          -- Unicode letters are valid
func_22     -- underscores are allowed
$name       --  $ is legal, but reserved for special system names
```

Invalid identifiers:

```
22f          -- must not start with a number
_myname      -- cannot start with underscore, reserved for system names
```

Data types

These are the main data types used in Beads:

`func`	a pointer to a function
`meas`	a physical measurement, a record with fields `mag` and `unit`
`num`	a numerical quantity, or an enumerated value, or a unique index
`str`	a Unicode character string
`tree`	a tree/graph structure, where each node contains a value (of any type but `tree`)
`bool`	a subset of the `num` type, has values `T, F, U`, or `ERR`

There are some predefined data types for commonly used audio/visual resources:

`color`	a color
`font`	a font resource
`image`	a bitmap image
`sound`	a sound resource
`video`	a video resource

There is a flexible open type of value that has a dynamic type:

`any`	a flexible type of data that can store any kind of data

You can create compound types by using the pointer or array prefixes:

```
array of num     -- an array of numbers
array^3 of str   -- a 3-dimensional array of strings
ptr to a_rect    -- pointer to a rectangle
ptr to array of num   -- pointer to an array of numbers
```

There are some types related to find/replace operations:

`changelist`	a list of find/replace operations to be applied as a set (similar to SED)
`regexp`	a regular expression pattern

There are low-level types that are used for interfacing with external systems:

`object` an internal system object that holds OS dependent data

`bits` a variable length bit string (used for TCP/IP packet formation)
`bytes` a variable length byte string (used for C record interfacing)

There are commonly used record definitions in the standard library:

`a_date` - a date/time record
`a_gradient` - a color gradient resource
`a_xy` - a 2D point (fields `x`, and `y`)
`a_xyz` - a 3D point (fields `x`, `y` and `z`)
`a_rect` - a rectangle (fields: `left, top, width, height`)

There is no separate data type for a single character; it is considered a string of length 1. To convert between `num` and a single character, use the `to_char()` and `from_char()` built-in functions.

There are various built-in functions available to convert between data types, such as the very frequently used `to_str()` or `to_num()` functions. See the arithmetic appendix for examples of arithmetic operations and conversions.

The Beads arithmetic space

Yes/No literals:

A `bool` value (an evolution of the normal 2-value Boolean type common in other languages) has 4 possible values:

`T` -- the value is true (on)
`F` -- the value is false (off)
`U` -- the value is undefined
`ERR` -- the value is the error value

In the `if` statement, the `if` clause will only be executed if the conditional expression resolves to `T`

Note that per Douglas Crockford's suggestion, if you multiply a `bool` variable with a numeric value, it has the effect of only adding that term if the value is true, so one can avoid `if` statements by using a `bool` variable to control factor inclusion, for example:

```
const FLAG1 = T
const FLAG2 = F
var sum = FLAG1*factor1 + FLAG2*factor2
```

The above statement is equivalent to:

```
var sum = 0
if FLAG1
  sum = sum + factor1
if FLAG2
  sum = sum + factor2
```

Number literals:

A number literal can be in the following forms:

`12`	
`12.45`	
`0.45`	a number must start with a 0 before the decimal point.
`12.45e2`	scientific notation; equivalent to 1245.000
`-12.45e-2`	negative exponent; equivalent to -0.1245
`12_456_890`	underscores are allowed in numbers (but not commas)
`__0__`	a very elaborate way to write 0
`U`	the undefined value
`ERR`	the error value
`INFINITY`	infinity

Invalid numbers:

```
.45      -- a number cannot start with decimal point
__       -- a number must have at least one digit
```

Arithmetic Operators

```
+    -- Addition
-    -- Subtraction
*    -- Multiplication
/    -- Division
/.   -- Integer division; equivalent to round_down(a/b)
^    -- Exponentiation
|    -- Rational exponent marker
```

See the tables in the appendix that show the results of various combinations of values. Note that Beads supports a reasonable implementation of transfinite arithmetic, where an expression like `3` divided by `INFINITY` is `0`. The rules concerning the propagation of the error value `ERR` are very helpful at limiting the impact of an invalid value inside a long calculation; in many languages you would need extensive input value checking, but in Beads you can just let the long calculation execute completely, and if at any point a value involved in the calculation involves an error value, then the expression will result in `ERR`, which can be detected at the end with a single guard expression.

Examples:

```
3 + INFINITY     evaluates to INFINITY
INFINITY + -INFINITY  evaluates to 0
3 / 0            evaluates to INFINITY
-3 / 0           evaluates to -INFINITY
5 /. 3           evaluates to 1.0 (integer division)
3 + ERR          evaluates to ERR
U + 4            evaluates to U
U + ERR          evaluates to ERR
```

Exponentiation is restricted to rational integer constant powers, and uses the vertical bar character to mark the exponent ratio rather than the slash, so that expressions can be parsed unambiguously. To perform exponentiation to a real number power, use the `power()` standard library function.

```
a^3       -- equal to a to the third power, a*a*a
a^1|2     -- equal to a to the one half power, or sqrt(a)
a^1|2/2   -- equal to sqrt(a)/2
a^3|2     -- equal to a to the three-halves power, or sqrt (a*a*a)
a^1.5     -- illegal construction, use power(a, 1.5) to calculate this
```

Exponentiation can be used with physical measurement values, and the units are multiplied:

```
var a : Length = 2 meter
var b : meas = a^3            -- b is now 8 meters^3
```

Arithmetic comparison operators are very similar to arithmetic operators, and are extended in Beads, where the result of a comparison has 4 possible values: true, false undefined, or error (`T`, `F`, `U` or `ERR`):

```
3 < 4       evaluates to T
3 > 4       evaluates to F
3 > U       evaluates to U
3 > ERR     evaluates to ERR
```

Boolean operations use extended arithmetic rules:

```
T and F     evaluates to  F
T and U     evaluates to  U
not ERR     evaluates to ERR
```

Operator Precedence

The operators are mostly left-to-right, with a few sensible exceptions:

Highest priority to lowest priority:

```
( )
not
^, |
*, /
&, +, -
==, <, <=, >=, >, <>
and, or, xor
```

```
const a = 3 + 1 * 10     -- has the value 30, as times is higher priority than plus

if a < 3 and b > 5       -- same as if (a < 3) and (b > 5)

y = 2*x+3*2              // evaluated as (2*x) + (3*2)
y = 3+4*x^2              // evaluated as 3 + (4*(x^2))
```

note that a `0x` prefix is the hex encoding prefix, and does not mean `0` times `x`.

```
z = 2 + 0x002            // evaluated as 2 + (hex 002), which is 4
c = not a and b          // evaluated as (not a) and b
d = not a and not b      //  evaluated as (not a) and (not b)
```

Units and Unit Families:

Variables of type `meas` store the unit of the quantity as well as the numeric value. The unit and

quantity form are carried around at execution time, and enables the Beads system to catch errors at compile time and at runtime. Expressions like `12 lb + 3 kg` are perfectly valid, but mixing length and mass like `12 ft + 3 kg` are nonsensical, and will be flagged by the compiler as an error. Since any two measurements can be combined at runtime, the runtime system will convert an erroneous expression into the `ERR` value. During compilation and execution time automatic conversion is performed, as the terms in the expression `12 ft + 3 m + 2 in + 5 mm` will be converted to SI internal units.

Units of a similar type like foot, meter, inch, mile, are members of a predefined unit family: `Length`. When defining a function you can require that the parameter passed be of a particular unit family. This allows the function to be generic across all compatible units.

To indicate compound physical units, like energy, you use the bullet • (U+2022) character as a unit separator, although an asterisk is permitted if you can't conveniently enter the bullet character. Positive and negative exponents are allowed, but the exponents must be either whole numbers or ratios of whole numbers. To convert a value explicitly, you can use the assignment statement where you specify the target unit:

```
var len1 = 30 meter          -- define variable of type meas
var len2 = len1 as inch      -- len1 is converted to inches and stored in len2
```

See the appendix for a list of the full set of units and their abbreviations.

`meas` literals:

To describe a number + unit, just follow the quantity with the unit name or abbreviation:

```
8 percent            -- percent is a kind of unit
32 euro              -- currencies are a kind of unit
12 pt                -- printer's points are 1/72nd of an inch, common in layout work
340 kilogram
340 kg               -- many units have abbreviations
18 kg•m^2/sec^2      -- a compound unit expression
3 m3                 -- 3 cubic meters
2 kg•m^2/sec^2
2 kg•m^2•sec^-2      -- same as above, but using negative exponent instead of /
5 m^1|2/kg           -- meter to the ½ power per kilogram
```

(note that the vertical bar character is used to denote rational exponents, so as to not conflict with the slash which means arithmetic division.

A `meas` variable is treated as a record with two fields: `val` (for the value) and `unit`.

```
var mydog = 18 kilogram          -- mydog implied to be meas type
var myscalar = mydog.val         -- myscalar is now 18
var myunit = mydog.unit          -- myunit is now kilogram
```

Note that any number is equivalent to a `meas` of type `scalar`. So `3 %` is the same as the value `0.03`. Arithmetic between a scalar value and a measurement of type unit is permitted without the use of conversion functions. Since most of the practical unit families are defined already in the standard library, one normally just adds an additional unit to a family that already exists. You specify a unit by giving the conversion formula between a previously defined unit and the new unit:

```
family Scalar base: each
unit of Scalar percent abbrev: '%' ratio: 100 percent = 1 each
unit of Scalar dozen   abbrev: 'doz' ratio: 1 dozen = 12 each
unit of Scalar bakers_dozen ratio: 1 bakers_dozen = 13 each
unit of Scalar gross  ratio: 1 gross = 144 each
unit of Scalar tenth  ratio: 10 tenth = 1 each
```

The compiler will automatically convert compatible terms in an expression:

```
var mylen = 12 foot + 22 meter + 1 inch + 2 mm
```

When you perform arithmetic on quantities with units, the units combine as expected:

```
var myarea = 12 ft * 22 meter     -- myarea now has dimensions of length²
```

The compiler and runtime system enforce that compatible units are being used in expressions. For example, this expression would get a compiler error, as the units are not compatible:

```
var mylen = 12 ft + 2 hr      --  compiler error: distance and time cannot be added
```

The convenience of units is particularly evident when using built-in functions. For example to obtain the sine of 30 degrees, since the sine library function defines its argument as a measurement of the family `Angle`, you cannot send a scalar value to the function but must always specify the units of the Angle, which could be degrees, radians or gradians:

```
myval = sin(30 deg)        -- equal to ½; not the same as  sin(30 rad)
```

Use the `meas_to_num` library function to convert a physical measurement into the magnitude of any unit in the same family. In this case the `arcsin` function by default returns an angle with internal SI value of `radians`, but if you want the result as a scalar in degrees:

```
myangle = meas_to_num(arcsin(0.50), deg)
```
-- returns the magnitude of angle in degrees

Note that you cannot request a unit that doesn't belong to the underlying family. This would not compile, as you can't convert a length into a mass unit:

```
myweight = meas_to_num(20 ft, kilogram)    // compiler error
```

The unit system is a 2-level hierarchy. Each unit belongs to a unit family. User defined units are created by using the `unit` keyword, and then specifying a conversion in terms of other members of the family. The conversion can either be a static scaling factor (the most common case), or a pair of functions to convert to and from the base unit of the family if the relationship is not a simple ratio (like temperature). For example, let's create our own private unit of length, called a `smoot`, named after a fraternity prank at MIT where some students painted lines on the on the Charles River Bridge walkway by using Oliver Smoot's body as a ruler:

```
//  define our new unit supply the ratio relative to a known unit
unit of Length Smoot abbrev: "Smt" ratio: 1 Smoot = 1.7018 m
```

You can then refer to a `smoot` (or its abbreviation `smt`) in formulas and conversion routines, as it is freely convertible to any other unit of type length.

```
//  compute 12 feet plus 3 meters plus 4 Smoot:
var mylength = 12 Ft + 3 Meter + 4 Smt
```

Money is an interesting type of unit, as the exchange rates fluctuate daily. In this special case you will typically use a plug-in module that will obtain the current rates. But for some models, the rates are frozen, and you can specify the rates as follows:

```
//  update the conversion rates
unit of Money JPY ratio: 1 JPY = 110 USD
unit of Money GBP ratio1 GBP = 1.50 USD
```

You can add your own unit families. In this example we will add a new unit family called `resonance` that has no relationship to other fundamental physical units, with the units `throb`, `kilothrob`, and `megathrob`:

```
family Resonance base: Throb
unit of Resonance kiloThrob  abbrev:"kT" ratio: 1 kiloThrob = 1000 Throb
unit of Resonance megaThrob ratio: 1 megaThrob = 1000000 throb
```

```
unit of Resonance microThrob ratio: 10000000 microThrob = 1 throb
```

Now in the source code you can use the physical units in expressions:

```
var myval : Resonance = 3.5 Throb + 1 kTb + 2 megaThrob
```

You can mark a unit family as nonlinear, which will prevent the user from doing any arithmetic on that unit. For example, decibels cannot be added, as they relate to logarithms:

```
family SIGNAL_STRENGTH base: decibel abbrev: db nonlinear
```

Then the expression:

```
3 db + 4 db
```
-- will generate a compiler error, as the units are nonlinear.

You can create unit types of arbitrary complexity; the only limitation is that the exponents of the base types can be expressed as ratio of integers. By creating a new unit family and sets of related units, you can eliminate almost every explicit unit conversion, which may reduce the amount of arithmetic in your code by a significant margin. Each part of a physical unit can be any rational exponent of the fundamental units. In this example we define the family of energy with the base unit called joule, using fundamental physical dimensions length^{1}•mass^{2}•time^{-2}

```
family Energy base: joule dimensions: length • mass^2 • time^-2
```

Advanced usage: In physics there is a unit of electric charge discovered by Maxwell, with the units length$^{3/2}$ • mass$^{1/2}$ • time^{-1}. To define such a unit we name the family charge, and then specify the generic formula as a series of terms referring to the internal names of existing unit categories (the abbreviation of the category actually) and the ratio of the numerator and denominator of the exponents of those units. A negative power means division, and ½ power means square root. The power 3/2 means the square root of the cube.

```
family Charge base:mx dimensions: length^3|2 • mass^1|2 / time
```

Units can go beyond simple ratios to a base unit, by specifying a pair of functions for conversion. For example the units of temperature are stored in the SI units of degrees Kelvin, and the other temperature scales are related by means of conversion functions:

```
calc Celsius_to_Kelvin (Celsius:num) : num
    return Celsius+273.15
```

```
calc Kelvin_to_Celsius (Kelvin:num) : num
    return Kelvin-273.15

family Temperature base: DegK
unit of Temperature DegC toBase:Celsius_to_Kelvin fromBase:Kelvin_to_Celsius
```

There is a standard plug-in library input control that offers 2 related boxes; one for the value, and one for the unit type, which can be constrained to a particular unit family, such as weight. This allows convenient inclusion of user-entered unit values:

Inside a draw block, there are special automatic conversions available for physical units of length for screen dimensions. The unit of points (`pt`) will be converted at run-time using the current dots per inch value to a scalar number of dots. The units of point are not a variable of type `meas` like a regular measurement would be, but are a way to specify a scalar number of pixels in a more convenient form.

String literals:

A string literal is enclosed with matching pairs of double quote characters.

```
"This can't be bad!"  -- note the single quote inside is valid
"You can also include \"double\" quotes"  -- backslash escapes double quote characters
```

The ampersand `&` is the string concatenation operator. The plus symbol is reserved for numeric arithmetic.

```
"hello" & "--" & "world"    -- results in "hello--world"
```

Standard escape sequences:

Sequence	Purpose	Unicode name	Hex Code
`\\`	Single backslash	REVERSE SOLIDUS	U+005c
`\'`	Single quote	APOSTROPHE	U+0027
`\"`	Double quote	QUOTATION MARK	U+0022
`\t`	Tab character	CHARACTER TABULATION	U+0009
`\r`	Carriage return	CARRIAGE RETURN	U+000D
`\n`	Line feed	LINE FEED	U+000A

\b	Non breaking space (prevents word wrap from splitting apart)	NO-BREAK SPACE	U+00A0
\-	A wide dash —	EM DASH	U+2014
_	A space character (used in translated strings to put spaces at beginning or end)	SPACE	U+0020
\!	Inverted exclamation ¡	INVERTED EXCLAMATION MARK	U+00A1
\?	Inverted question mark ¿	INVERTED QUESTION MARK	U+0191
\<	Left guillemot «	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK	U+0171
\>	Right guillemot »	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK	U+0187
\.	Bullet •	BULLET	U+2022
\u1234	4 character Unicode hex code (Astral 5 digit codes not supported yet)		
\c[unicodename]	Unicode character by name, see appendix for a list of supported names		
\{	Embed a brace without it being interpreted as an embedded expression		
\}	Embed a brace without it being interpreted as an embedded expression		
{expression}	Embed an arithmetic expression		

Multiline literals are very handy for inserting boilerplate text into a string literal. You start with triple double quotes, end with triple double quotes:

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

To allow inline comments that are entirely ignored by the compiler, useful for computer generated code, a literal in backquotes is skipped over by the compiler:

```
`this is ignored`
```

The above would generate a string `"this is\na multi-line\nmessage"`. This example took the default options, which are to map any line break characters in the source file into `\n` (linefeed) characters.

You can follow the string with a pair of brackets with options, in the form `[opt:xxxx]`

The options are as follows:

`inset` - skip the first and last line breaks (for easy pasting)
`break_to_cr` - map line breaks to \r (Classic Mac style)
`break_to_lf` - map line breaks to \n (default, Unix, Mac style)

`break_to_crlf` - map line breaks to \r\n (Windows style)
`break_to_sp` - map line breaks in the line into spaces
`tab_to_sp` - convert any tabs to spaces
`tab_to_semi` - convert any tabs to semicolons (;)
`tab_to_bar` - convert any tabs to vertical bar (|)
`skip_low` - strip out any low characters other than line breaks
`skip_break` - strip out any line breaks

```
const a = """this has
a bunch of
words""" [opt:break_to_sp]
```

The above code would create a string `"this has a bunch of words"`, because each line break was mapped to a space.

```
const b = """
the rain
in spain
""" [opt:inset]
```

The above would map to the string `"the rain\nin spain"`; note that the break after the first triple quote was stripped, and the line break before the ending triple quote was stripped, but the interior line break was preserved. The `inset` feature is very handy for making it easy to re-paste blocks of text from outside the program code, so that you don't have to put the triple quotes on the same line.

String substitutions

Similar to Python, Beads has string comprehensions, which is a way of performing number to string conversions and string substitutions in one step, which is an extremely common task in programming. However, unlike Python, which has a fixed encoding system, the Beads string comprehension system is extensible; the standard modules can be augmented, and new conversion types added by the programmer. To do a string substitution, you put in an expression enclosed in braces {}.
Example of a substitution using a simple number conversion:

```
nwidgets = 3
"the number of widgets is {to_str(nwidgets)}"
```

The typing of the same conversion function `to_str` gets tedious, so there is a set of

automatically implied conversion functions, so one merely needs to put in an expression without a conversion function:

```
"the number of widgets is {nwidgets}"
```

The variable `nwidgets` is a `num` type, so the default conversion is to convert it to a string with automatic decimal digits. Each inline substitution calls either an implied or explicitly specified conversion function. The above escape sequence is mapped to a call to `convert_num(nwidgets)`

Depending on the type of the expression, the following conversion functions are implicitly called:

Data type	Conversion function implicitly called
`str`	
`num`	`to_str()`
`meas`	`to_str()` using the base unit of the family
`func`	Converted to function name
`tree`	`to_str()`
`path`	Converted to string like `mytree[1,22,33]`
`photo`	Converted to string like `myicon 33 x 22`
`enum`	String representation in default language

```
"The amount is {total+100}"
```

In this case the expression `total+100` is embedded into the string. If you wish to have more control over how the substitution is performed, you can explicitly call the conversion function:

```
"the current total is {to_str(total, precision:3)}"

person = "Fred"
adjective = "red"
noun = "car"
print "{person} has a {adjective} {noun}"

        -- would print "Fred has a red car"
```

To include braces in a string, you will need to escape them, otherwise it would be mistaken for an embedded expression:

```
"I just wanted \{ some braces \}"
```

Example #2:

```
person = "Fred"
```

```
amount = 12
item = "apples"
"{person} is going to need {amount} {item}"
```

Note that in most European languages, the noun changes it spelling depending on whether the quantity is 1 or more (some languages have even more complex rules). See a later section on translating string plurals.

String manipulation

The & operator is used for string concatenation. This is unlike JavaScript, where + means both arithmetical addition and string concatenation. In Beads the plus sign (+) is reserved for numeric addition.

Strings are treated like an array of characters. The array is densely packed and is not sparse like the tree structure; all existing slots have some character assigned. The first character of a string is at index 1. This follows the example of Pascal, Modula-2 and other languages, but differs from C, JavaScript and others, which start at 0.

There is a collection of string functions, see the standard library section for more information. The most commonly used string function is str_subset() which allows you to extract a portion of a string.

Bit string manipulation

(note: not yet implemented in compiler)

Bit strings are treated like an array of bits. The array is densely packed, and when padding is needed, 0 bits are inserted. The bits are numbered from the high order bit, starting at 1. You manipulate them as if they were character strings, but each letter instead of being a Unicode character is either a 0 or 1 bit. The same subset functions apply to bit strings, but use single bits as their unit of measurement. The & operator means concatenation.

```
var ss : bits
ss = 0b101          ss now holds 3 bits
ss[FIRST] = 0b0     Sets the first bit of the string to 0
ss[200] = 0b1       The bits from 4..199 are padded with 0 bits
ss = 0b10 & 0b111   ss is 2 bits concatenated with 3 bits, 10111 binary
ss[1::2] = 0b01     ss first two bits are changed, now contains 01111
ss[1::5] = 0b1      ss now holds 00001  (automatically padded on left with zeros)
ss[5] = 0b0         ss now holds 00000 (set bit at offset #5 to zero)
```

```
ss[4..5] = 0b11      Bits 4 though 5 inclusive are set, now holds 00011
len(ss)              Returns 5, the length of the bit string in bits
```

Byte strings are treated like an array of 8-bit bytes. The bytes are numbered starting at 1. You manipulate them as if they were unsigned numbers from 0 through 255. The same subset syntax applies to byte arrays as character strings, and bit strings. The `&` operator performs concatenation. You can include ASCII character, or embed Unicode strings using the `str_to_utf8()` function. There is an additional syntax to specify endian-ness of byte fields. The default is native CPU order (usually little endian for Intel and ARM CPU's), but you can specify "big" or "little" preceded by vertical bar to force a specific byte order:

```
var ss : bytes
ss = 0x22 & 0x33 & 0x00      ss is now 3 bytes long: 0x223300
ss[1::3|big] = 0x22          ss is 3 bytes long, big endian layout, 0x00, 0x00, 0x22
ss[1::3|little] = 0x22       ss is 3 bytes long, little endian format, 0x22, 0x00, 0x00

var n : num
n = ss[1::3|big]             extract 3 consecutive bytes, big endian,  convert to num
```

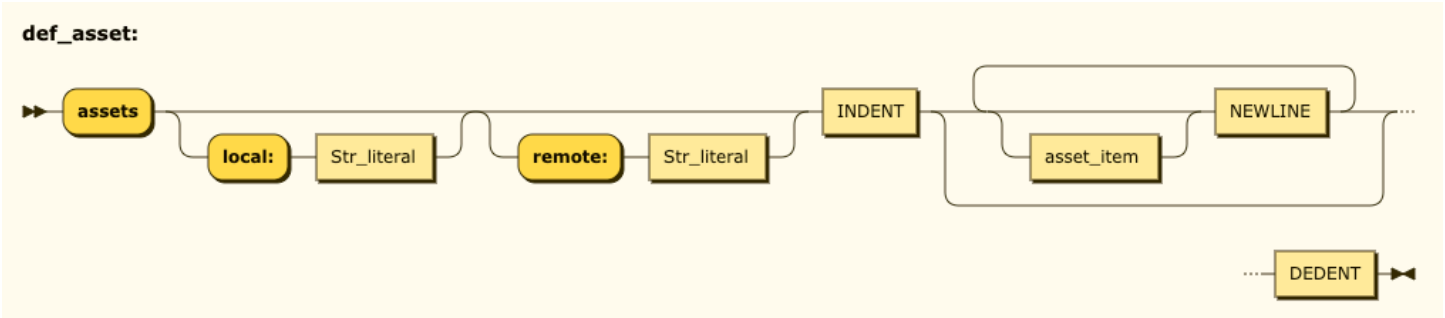
(note: in first implementation `num` type is only 52 bits of integer precision, so any byte field larger than 6 bytes cannot be converted to a number.

Art Assets

Almost every program will need to include various art assets. To include a resource, you use the `assets` directive. The current kind of `asset` file suffixes:

`image` resources:	`.gif, .jpeg, .jpg, .png, .svg`
`sound` resources:	`.aac, .mp3, .wav`
`video` resources:	`.fv4, .mp4, .m4a, .mov, .mp4v, .3gp, .3g2`
`font` resources:	`.ttf, .ttc, .otf, .woff`

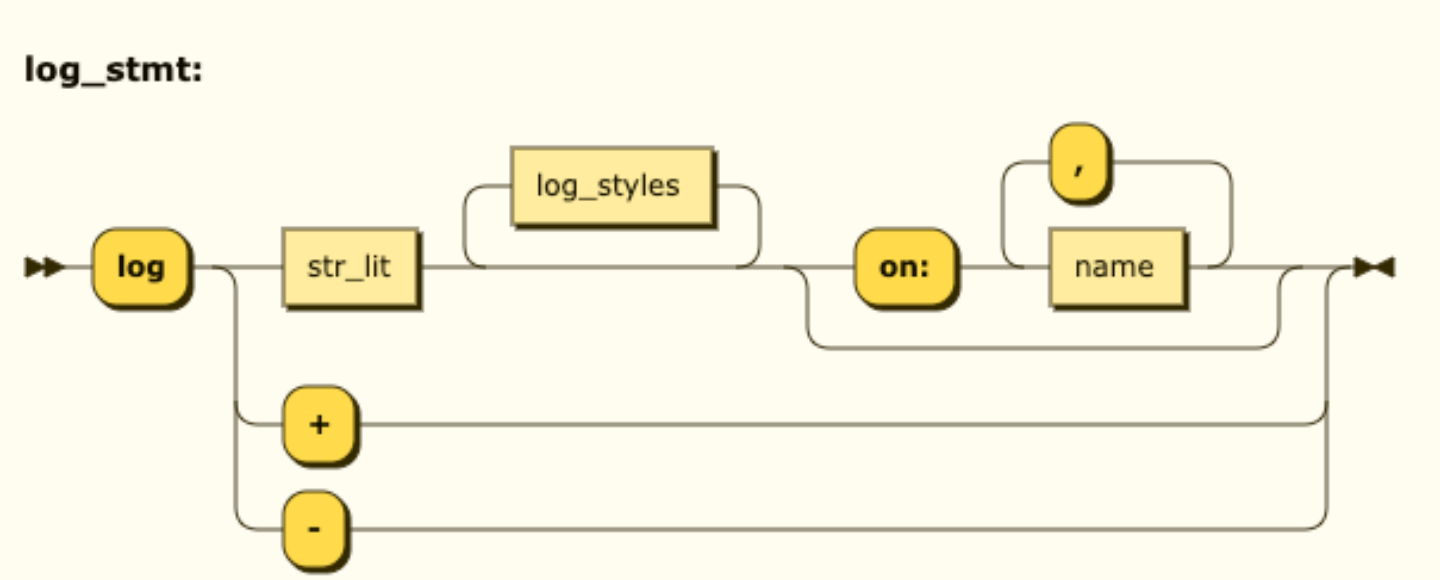
You give the imported resource a name; it will be treated as a constant with that name, and of data type `image`, `sound`, `video`, or `font`. For bulk import of resources, you can specify a folder, and which part of the name should create the key. For example if you have a folder of art resources with names and a numeric suffix like `picture_002.jpg`, and use the `tail` keyword to specify that the tail of the file name creates the key for the index into the array, then that picture would be stored at index 2. The `seq` keyword means insert the resources sequentially starting at 1. If you use the `name` keyword, then the name becomes the key. When making products for both desktop and web use, you will typically specify the local file location of the resources, and the folder where the web resource will be located, so the program knows where to access the resource at execution time. In web apps, the art resources are downloaded from some image cache.

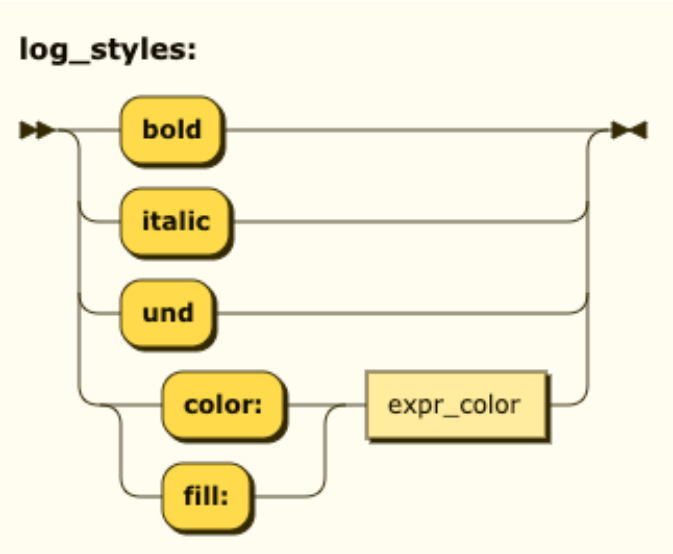


Examples of some resource statements:

```
assets local:"art/"
   file:"patek_cleaned_750v.jpg" label:watch_background preload
    file:"hand_day.png" label:day_hand preload
    file:"hand_min.png" label:minute_hand preload
    file:"hand_hour.png" label:hour_hand preload
    file:"hand_sec.png" label:second_hand preload
```

Logging





In order to facilitate logging, there is a built-in command to the console or a log file. A simple example would be:

```
log "the value of x={x}, y={y}" on:FLAG1
```

The optional `on:` keyword specifies the flag that determines if the tracing is to be enabled, this saves the effort of doing IF statements around the log statement which is very common. There are also options to increase/decrease the indenting (which uses tabs)

```
log +    // this increases the indenting level of the log tracing
```

```
log -    // this decreases the indenting level of the log tracing
```

Note that any `log` statements that you include in a server, can be echoed back to the client's console if the client specifies the log option in the `subscribe_start` call. This is a very handy feature when debugging client/server programs.

Inside a local browser you can customize the style by changing the text color, the background color, or making the console text bold or italic or underlined.

Modules

The first line of each file contains a module declaration. You must declare the version of the Beads language to be used during compilation. You can specify the language of the source code, because Beads has its keywords translated into many languages besides English. A project can mix source code using different versions of the language, as the compiler makes an effort to be backwards compatible.

```
#beads level 3 lang LANG_ENG module mymodule      // module definition
```

At run time when the main module and all dependent modules are loaded, each module that has a function named `prolog`, that function will be called to perform any module-level initialization which might be required. `Prolog` functions are called in reverse topological order so that the lowest level prolog function is called first.

For the `import` statement see Module structure

Enumerated constants

An enumerated constant is a symbol you use as a guaranteed to be unique number throughout the program, without caring about its internal value. For example, let's say you are implementing a traffic light model, and the light has 3 possible states:

```
enum
  RED
  YELLOW
  GREEN

var
  mycolor = GREEN
```

In this code above, the numeric variable `mycolor` is assigned the value `GREEN`. Any attempt

to use arithmetic on `mycolor` when it holds an enumerated constant value will result in `ERR`.

```
mycolor  = mycolor  + 1   -- will result in  ERR
```

In many languages you would define a constant `RED` to be 1, `YELLOW` to be 2, etc., the Beads approach has the advantage of protecting against accidental arithmetic on an enumerated value, which can cause baffling program malfunctions.

You can use either the single line form to declare a single `enum`:

```
enum ITEM_NAME
```

Or you can declare a block of enumerated constants:

```
enum
  ITEM_NAME "Item Name"
  ITEM_COST
  ITEM_QTY
```

When you specify an `enum`, you can optionally provide a custom string representation, which is used automatically during string conversion. If you don't declare a custom string representation, the default string form is the `enum` name. Enumerated types form a convenient way to implement a simple string table.

To obtain the string form of an `enum` use the built in function `enum_to_str(val)`

`enum_to_str(ITEM_NAME)` would evaluate to `"Item Name"`
`enum_to_str(ITEM_COST)` would be `"ITEM_COST"`

An enumerated type is stored internally a `num` type. The internal value is calculated using a hashing algorithm, so it is consistent between compilations, and guaranteed to be unique among all other numerical values. The internal values have no predictable ordering, so one cannot tell by looking whether `ITEM_COST` would sort before or after `ITEM_NAME`. You can use built-in functions to determine if a number is numeric (`is_numeric`) or is an `enum` (`is_enum`).

Because all `enum` constants are globally unique, run-time debuggers can show the enumerated string form rather than the internal numeric value. It also eliminates a subtle kind of problem. In Microsoft Windows for example, there are tens of thousands of numeric constants declared in header files by various modules, and many of the constants are defined as low value integers, and the values are therefore ambiguous. In Windows, if you assign a `PRINTER_A4_PAPER` value to a variable, and later compare that variable's value with some other system constant, which

happens to have the same value, but an entirely different meaning like `KEYBOARD_ERROR,` it will match and you have created an extremely subtle error. It is also very difficult to use debuggers with enumerated constants in languages like C, because the debugger doesn't know which of the thousands of matches to the value "1" should be displayed. The use of a guaranteed-to-be-unique internal value has many advantages.

Number, String, and Tree Constants

You can declare a single constant:

```
const  maximum_tries = 12
```

Or use the indented block form to declare a group of constants:

```
const
  max_tries = 12
  filename  = "mydata.doc"
```

In most cases the compiler can infer the type of the constant from the value string, but you can specify the type if you wish:

```
const
  max_tries : num = 12
  filename  : str = "mydata.doc"
```

Examples of numerical constants:

```
const num pi = 3.14159         //  constant has an explicit type
const float128 sqrt_9 = 3.0    //  constant has an explicit type
const pi2 = 3.14159            //  implied type of num
const pi_squared = 3.14159*3.14159 //  a simple expression
const pi_squared2 = pi*pi      //  can use previously defined constants
const pi_root = sqrt(pi)       //  resolved at compile time
```

Only functions that are pure functions with no external dependencies can be called at compile time and used in a constant expression. If the function is not pure, you will have to use a variable instead of a constant. If you precede a constant name with `$`, then that constant is exported outside the module, and can be used into another module. In C this would be like using the "extern" keyword.

String constants

```
const
   $myss = "hello"
   myss2 = myss & " world" //  myss2 is "hello world"
```

The `&` operator performs string concatenation. The `+` symbol is reserved for arithmetic.

Matrix literals:

Here is an example of a one-dimensional matrix:

```
const primes = [ 2, 3, 5, 7, 11, 13, 17, 23 ]
```

Note that the default starting subscript of an array is 1 as in Pascal, not 0 as in C and many other languages. Also note that commas in an array literal are optional. It is up to the preference of the programmer. Multidimensional matrices can be declared using the semicolon as a marker for the next dimension. You insert a semicolon for each new dimension you are initializing. For example, a two-dimensional array of two rows and three columns could be defined:

```
const matrix2d = [ 11  12  13  ;  14  15  16 ]
```

In mathematical notation as: $\begin{bmatrix} 11 & 12 & 13 \\ 14 & 15 & 16 \end{bmatrix}$

2d Matrices are indexed [row, column], as the larger unit comes first. So in the above declaration, `matrix2d [2, 3]` has a value of `16`. And `matrix2d [1, 3]` has a value of `13`.

You can specify as many dimensions as you wish, although when you create an array constant you must fully populate the matrix. This is an example of a 3-dimensional 2 x 2 x 3 matrix:

```
const matrix3d = [1  2  3 ; 4 5 6 ;; 11 12 13 ; 14 15 16]
```

The above declaration creates a 3d matrix, which consists of two 2d matrices:

`matrix3d[1]` = $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

`matrix3d[2]` = $\begin{bmatrix} 11 & 12 & 13 \\ 14 & 15 & 16 \end{bmatrix}$

A 3D matrix is indexed [section, row, column]. The expression `matrix3d[2 2 1]` has a value of `14`.

A one-dimensional matrix is usually 1 row by many columns, but you can make a 2-dimensional matrix with one column and many rows:

```
matrix_vert = [ 1; 2; 3; 4 ]
```

This would be drawn in mathematics as: $\begin{bmatrix}1\\2\\3\\4\end{bmatrix}$

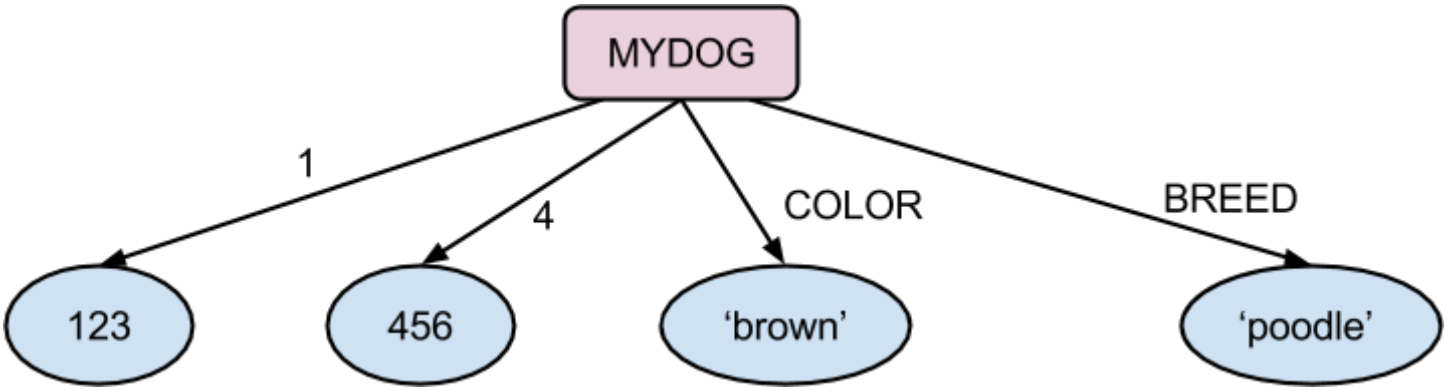
There are functions available in the standard library to multiply, invert, etc. matrices.

Tree literals:

The previous section showed array literals. The actual storage format for arrays is what we call the Supertree (hereafter referred to as `tree`), which is more flexible than a simple array. The `tree` structure is the most commonly used data type in Beads. It consists of a root node with as many children (and children of children) as you wish, up to the compiler's limit. The first implementation supports more than 4 billion nodes across, and at least 12 levels deep). The indices of the nodes are numbers (rounded to the nearest integer), enumerated values, or strings. Each node of the tree can store a single value.

Note that tree constants are marked by braces `{}`, while array constants use square brackets `[]`.

```
enum
      COLOR
      BREED
const MYDOG : tree = { 1:123, 4:456, COLOR:"brown", BREED:"poodle" }
```



Note that the commas in the above declaration were optional.
The following expressions would evaluate as follows:

`MYDOG[1]`	is the numeric value 123
`MYDOG[4]`	is the numeric value 456
`MYDOG[4.2]`	is the same as `MYDOG[4]` because we round to nearest integer
`MYDOG[5]`	is `U`, undefined
`MYDOG[COLOR]`	is the string `"brown"`

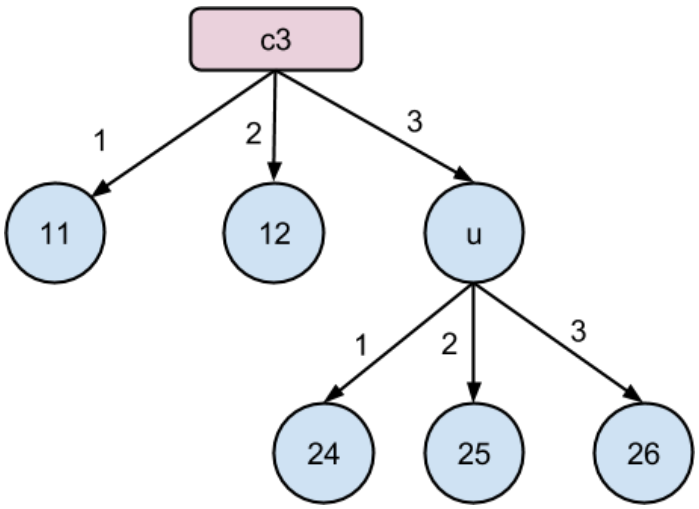
`MYDOG[BREED]` is the string `"poodle"`
`MYDOG["BREED"]` is undefined, because an enum is distinct from a string
`MYDOG` is the string `"MYDOG"`, roots of trees have by default a value of a string containing the name of the variable. If a tree is copied the name is not copied over.

The subscript is either an enum, a number rounded to the nearest integer value, or a string. String subscripts are case sensitive. A null length string is not a valid subscript. You cannot store a value at a subscript with the special meta values `U` or `ERR`. Assignment to an element at a null string, or `U` subscript will result in a run time error if checks are on.

If you don't know the value type of a tree node, you can call the built-in function `type_of_val()` to get the data type of a single node in a tree. It returns the enum value such as `TYPE_NUM`, `TYPE_MEAS`, `TYPE_STR`, `etc.`, or `U` if the node doesn't exist yet.

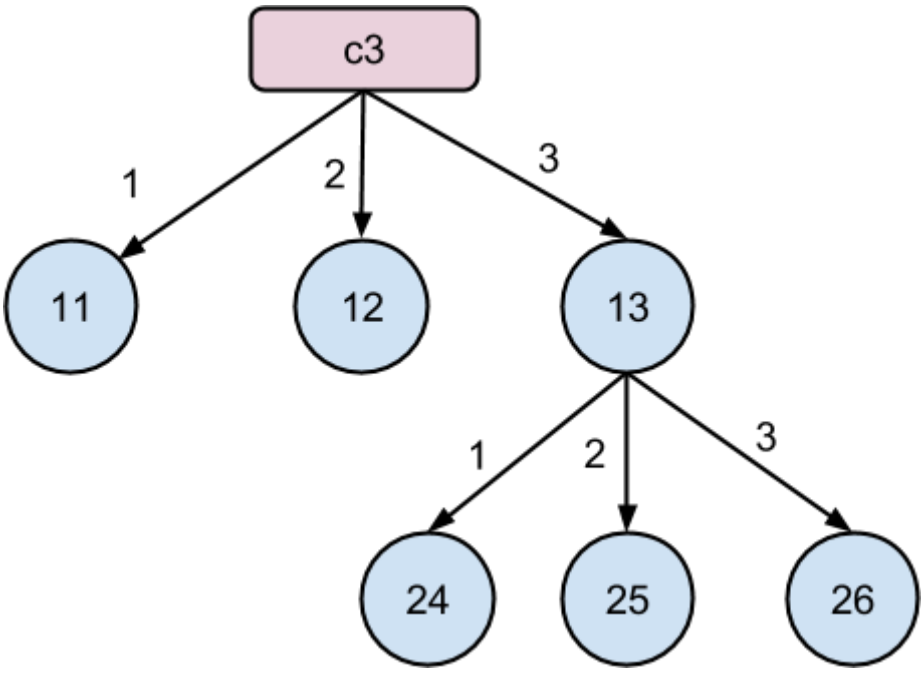
You can specify tree literals using nesting inside the constant declaration:

```
const tree c3  = { 1:11 2:12 3:{1:24 2:25 3:26}} }
```



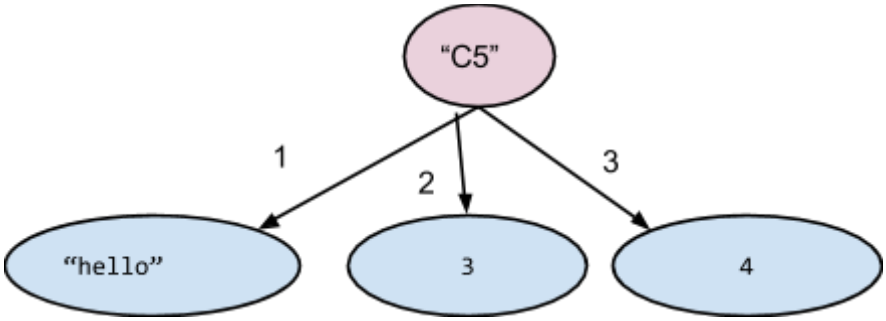
In the tree above, the 3rd child of C3 was not given a value. The following syntax assigns a value to node 3, and also specifies the children nodes of 24, 25, 26 and a subarray:

```
const tree c3  = { 1:11 2:12 3:13 [24 25 26]} }
```



As each node can store either a `str`, `num` or `photo` or `func` value, when you define a constant, the compiler can easily tell which field to assign. In this example below, a tree constant is defined that stores both strings and numbers:

```
const tree c5 = [ "hello" 3 4 ]
```



In the above examples, the subscripts at which the values were place were not explicitly specified. To specify the subscript at which a value is stored, use the explicit subscript syntax:

```
    subscript : val
```

alternative 1 / colon:

```
   const tree c6 = { PROD_NAME:"my prod", PROD_NOTES:"great", PROD_COST:99.01 }
```

Note that each node of a tree has a unique address, similar to the coordinate in a multidimensional array. To refer to an element of a tree, use the `tree[a, b, c]` notation,

where a, b, and c are the subscripts at each successive level of the tree.

Variables

There is a single line form of variable declaration:

```
var a
var x, y, z, q, r, s, t    --  defines variables x, y, etc.
var aa bb cc : str         --  commas are optional
```

There is a multi-line form where you start a var block, indent and declare a list of variables

```
var
  x : tree
  y : num
  z : str
```

If you precede the variable name with a $ character, it signifies that the identifier is exported and usable by other modules, otherwise it is private to the module. Therefore most symbols in a module are private (local) to the module. This is very similar to the syntax of Oberon (the sequel to Modula-2 and Pascal by the late great Prof Wirth).

Tree and Array subsets (slicing)

You can specify array subsets (or array slicing) using two alternative notations:

1) Spanning range -- specify a start index and end index (inclusive) using double periods:

`start .. end` -- example: `3..5` means `3, 4, 5`

(if end is less than start, empty list)

2) Segmented range -- specify the range as start index, and a length, using double colons:

`start :: length` -- example: `3::4` means `3, 4, 5, 6`

(a length of 0 or less means empty list)

All array subsets are listed from lowest value to highest subscript value, there is no way to specify the subset in reverse order:

Examples:

`a[5..3]` refers to no elements, will generate a compiler error

`a[3..3]` refers to a single element `a[3]`

`a[2..5]` refers to `a[2], a[3], a[4], a[5]`

`a[3::4]` refers to `a[3], a[4], a[5], a[6]`

`a[-4::5]` refers to `a[-4], a[-3], a[-2], a[-1], a[0]`

`a[4..0]` refers to no elements, will generate a compiler error

`a[4::-5]` refers to no elements, will generate a compiler error

Subscripts can be expressions, and expressions cannot be checked at compile time, so it is possible to refer to a null range at run time:

```
x = 1
y = 5
```
`a[x..y]` - refers to a[1], a[2], … a[5]
`a[y..x]` - empty list

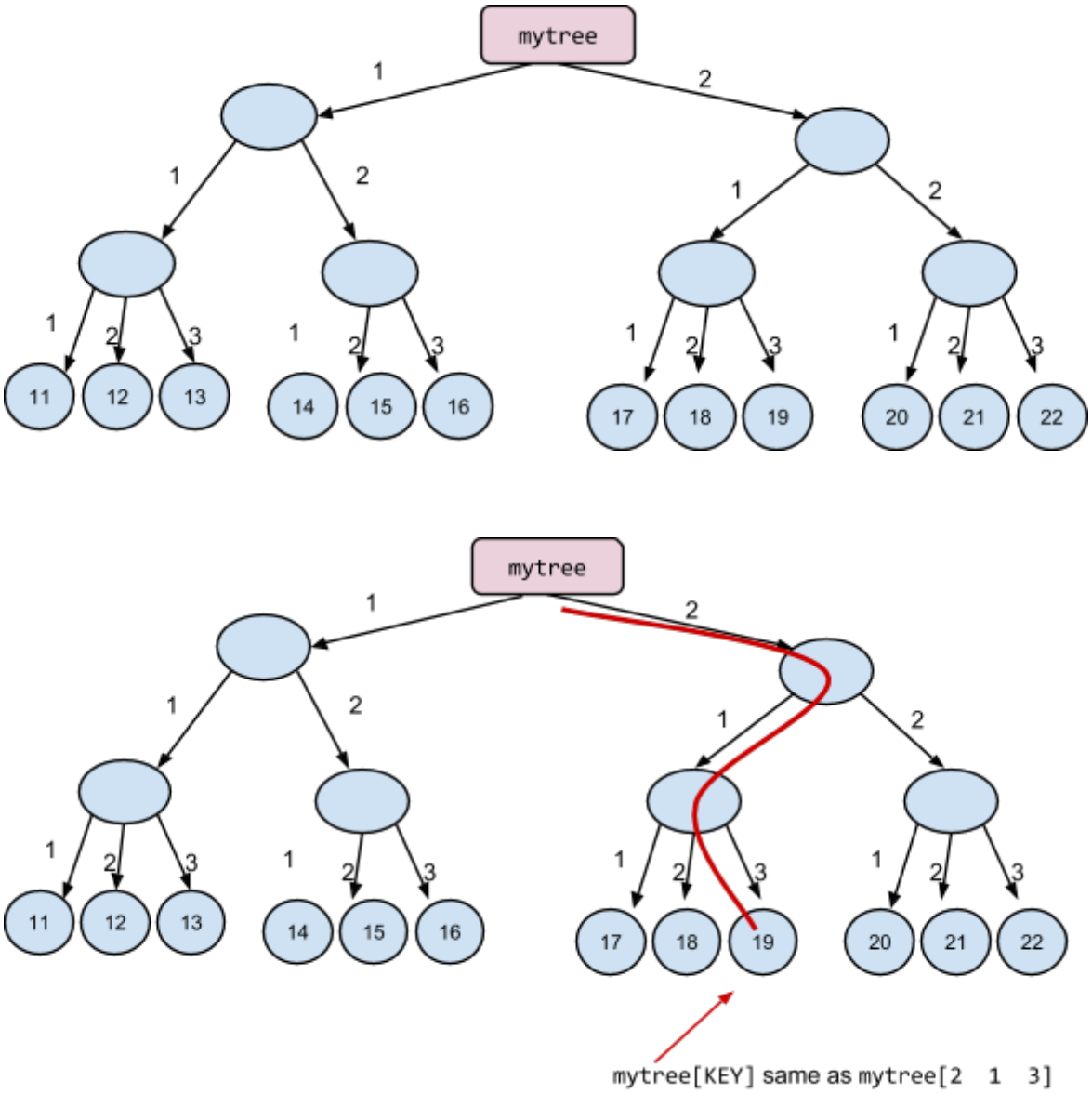
When the subscript argument is a tree value, then the tree is considered to be a 1-dimensional list of subscripts:

```
mykey = [2, 3, 5, 7, 11]
is_prime [mykey] = T
```

This will set the tree `is_prime` to have values of `T` at the indices of 2, 3, 5, 7, and 11.

Here is an example of a tree that will be indexed using a key, which navigates to a specific node in the tree.

```
var mytree = { ⟨[11 12 13],[14 15 16]⟩  ⟨[17 18 19],[20 21 22]⟩ }
const KEY = [2  1  3]
```



A specific node in the tree can be referred to with explicit subscripts (`mytree[2 1 3]`), or you can use a one dimensional array as a key into the tree, so `mytree[KEY]` refers to that same node. If you set the value of a node in a tree, that node is created, along with any nodes in the path that need to be created. This ability to specify an address before a value is created, is one of the greatest advantages of the system. In almost every other conventional language, you cannot manipulate a pointer to a location before the object exists, which creates an incredibly difficult to guarantee precondition in a program that has complex flow. In C if you refer to the value of a pointer that has not yet been allocated you may get an address protection violation or a garbage value, and Java is bedeviled by null pointers caused by usage of a pointer before it has been defined, or after it has been de-allocated.

In the example above, these two statements have the same effect:

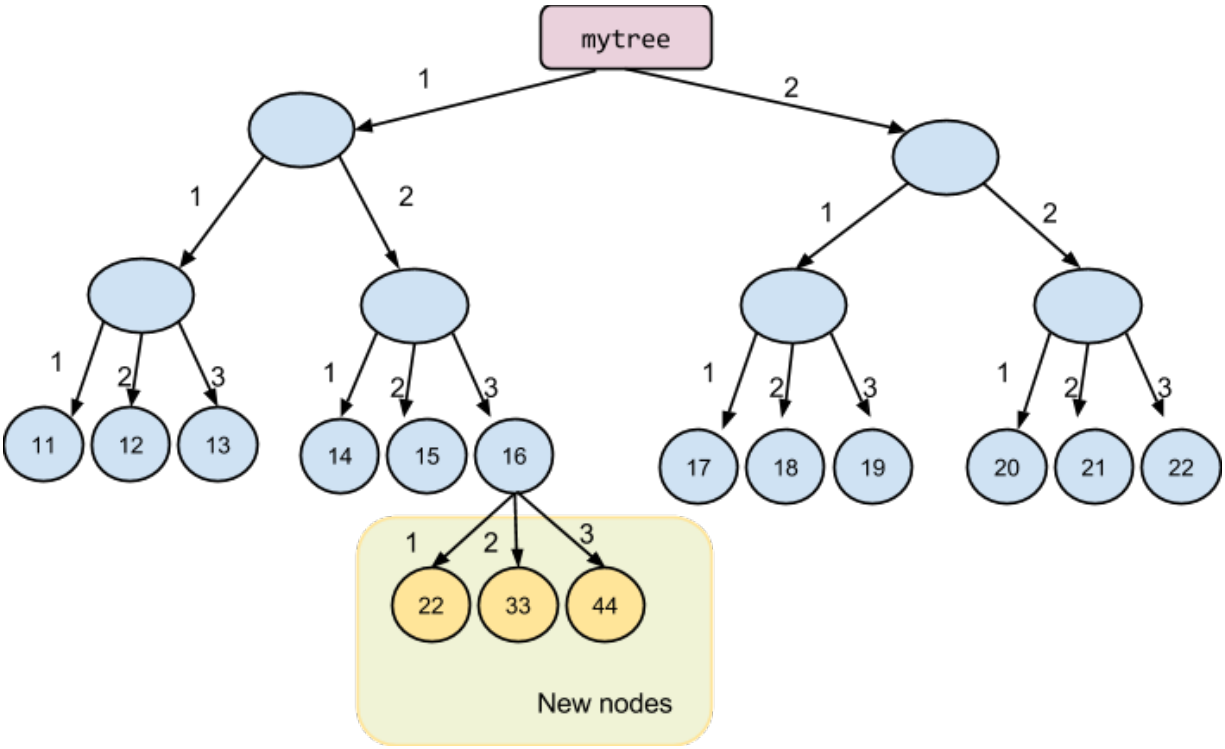
```
mytree[2 1 3] = 999
```

```
mytree[KEY] = 999    -- same as mytree[2 1 3]
```

Using array subscripts allows you to store pointers to specific nodes, without fear that the program will crash if the node is deleted; a reference to a non-existent node location will return `U`. If you set the value of a non-existent node, you bring that node into existence. The following statement:

```
mytree[1 2 3] <=== [22 33 44]
```

will copy the subtree `[22 33 44]` into the `mytree` structure at subscript `[1 2 3]`:



There are a variety of built-in functions to manipulate trees, permitting flexible addition, subtraction, and enumeration of subsets of the tree. The tree is best thought of as a database. See later sections on tree operations.

`mytree [path sub sub sub]` is valid
`mytree [path]` is valid
`mytree [sub sub sub]` is valid
`mytree [path sub sub path]` is invalid

Conditional statements

```
if boolean_expression
  <statements>
elif boolean_expression
  <statements>
else
  <statements>
```

The comparison operators are as follows:

```
a == b      -- is a equal to b?
a <> b      -- is a not equal to b?
a <= b      -- is a less than or equal to b?
a < b       -- is a less than b?
a >= b      -- is a greater than or equal to b?
a > b       -- is a greater than b?
```

The logical operators are as follows:

```
a and b     -- AND operator
a or b      -- OR operator
a xor b     -- XOR operator
```

There is also the `not` operator for negation. The `not` operator has low priority. The `and` and `or` operators are short circuit operators, so in an `and` operation once it hits the first non-true item it stops evaluation. And in an `or` operation once it hits the first true term it stops.

The precedence of `and` and `or` are lower than comparison operators, so no parentheses are needed in a cascaded set of comparisons:

```
if a < 10 or b > 10 or c < 100
  …statements…
```

There are some special considerations in a conditional expression. Boolean expressions have 4 possible values: true (`T`), false (`F`), undefined (`U`) or error (`ERR`). The condition of an IF statement is only executed if the expression is true (`T`). It will not execute if it is undefined (`U`) or an error (`ERR`). This solves one of the great ambiguity issues in prior languages, which did not define clearly which branch would be taken for an undefined or erroneous result. If run time checks are on, an `if` statement that evaluates to `U` or `ERR` will produce a run time error. If run-time checks are off, it will be treated as if it was false. The fact that an `if` clause will not be executed on an undefined or error value can be a subtle but important point:

```
if a <= 10
  statement1
elif a > 10
  statement2
else
  --  if a is U or ERR this statement will be executed
  statement3

if a == 10
  statement1
elif a <> 10
  statement2
else
  --  if a is U or ERR this statement will be executed
  statement3
```

WARNING: because a Boolean value in Beads has 4 possible values, the old trick used in most Algol family languages like C, JavaScript, etc., where you can shortcut evaluation of an expression by stopping on the first true condition in an expression doesn't apply. The expression:

```
result = flag_a or flag_b
```

could have 4 possible resulting values, and both terms need to be evaluated to determine the final result. This means functions that return Boolean values will always be called in a Boolean statement:

```
result = func1() or func2()
```

Beads does however support shortcutting in `if` statements. If the first term in an or expression is true, the condition will be considered true immediately, and the further term is not evaluated. This is quite different than the assignment statement, but is necessary because functions that perform side effects are often cascaded.

```
if myfunc(123) or myfunc(456)
 ….some code…
```

in the above case, if `myfunc(123)` evaluates to true, then `myfunc(456)` is not called.

However, if this were the assignment statement:

```
myval = myfunc(123) or myfunc(456)
```

then both function calls would be evaluated, and the net result would be computed and stored into `myval`.

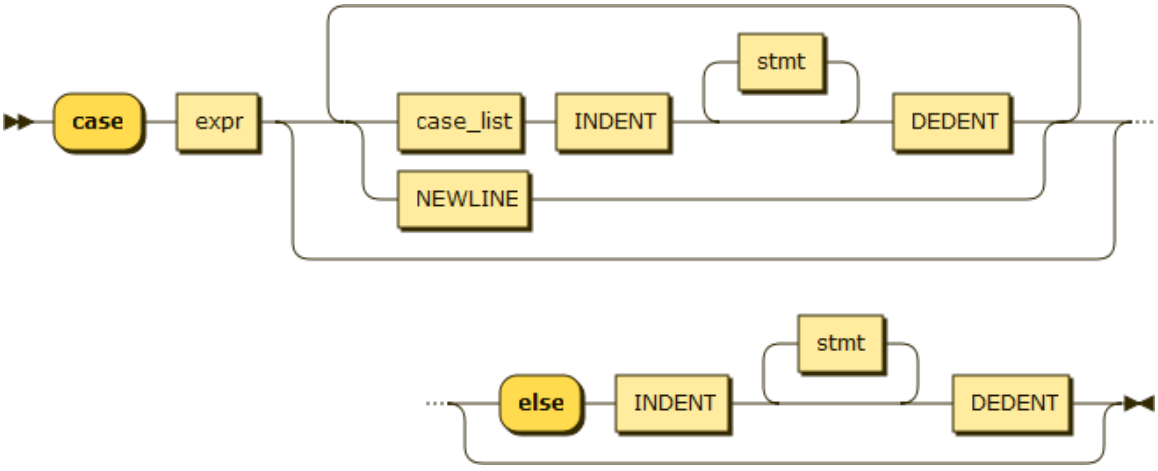
Although there is a `not` operator, to invert the state, but it is hardly ever used. The reason is that undefined values `U` when not is applied still are U. So the correct way to test against non truth is to compare not equal to `T`.

```
if not my_func()      -- incorrect way
if my_func() <> T     -- the correct way
```

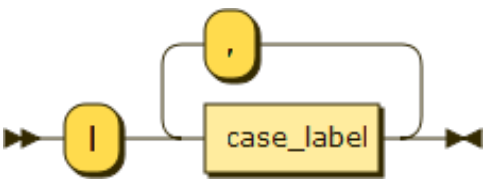
Undefined variables permeate the runtime environment, and to prevent execution against undefined values, to make your code conservative, you should also compare not equal to `T`, so that when it is either `F`, `U`, or `ERR` then the code will be executed.

A `case` statement is basically an abbreviated `if` statement. Note that unlike in C, `case` sections need no `break` statement, as execution does not fall into later cases. Also unlike C, which can only have single values, the sections of a case statement in Beads can have expressions or value ranges:

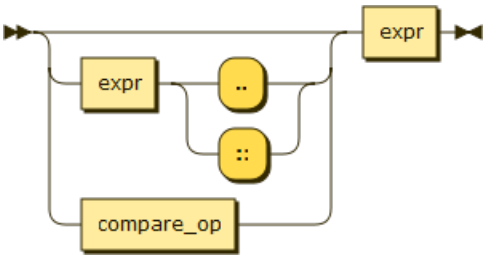
case_statement:



case_list:



case_label:




```
case myval
  | 1
    statements A…
  | 2..10  // the range of values from 2 to 10 inclusive
    statements B…
  | 11, 21, 31  //  multiple values
    statements C…
  | 30::5        //  starting at 30 for 5 consecutive values
    statements D…
  | >= 200  //  a comparison operator
    statements E…
  | >= 50  // another comparison operator
    statements F…
  else
    statements G…
```

The `case` statement is a pattern match system, executed in the order of definition. The remaining patterns are skipped over once a match happens. So in the case above, if `myval` is 31, it will execute statement block `C`, but not `D` which it also matches. If none of the patterns match, the `else` clause statements will be executed.

The case statement can be use with string values as well:

```
case mystring
|  "alpha"
    statements A…
| "beta"
    statements B…
```

The `else` clause is optional.

Assignment combined with conditional (ternary operator)

We allow you to create an expression using the `if` statement:

```
var a = 3 if x < 10 else 5
```

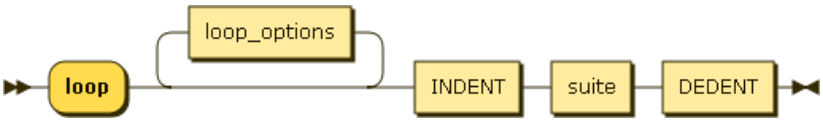
Which is the same as:

```
var a
if x < 10
  a = 3
else
  a = 5
```

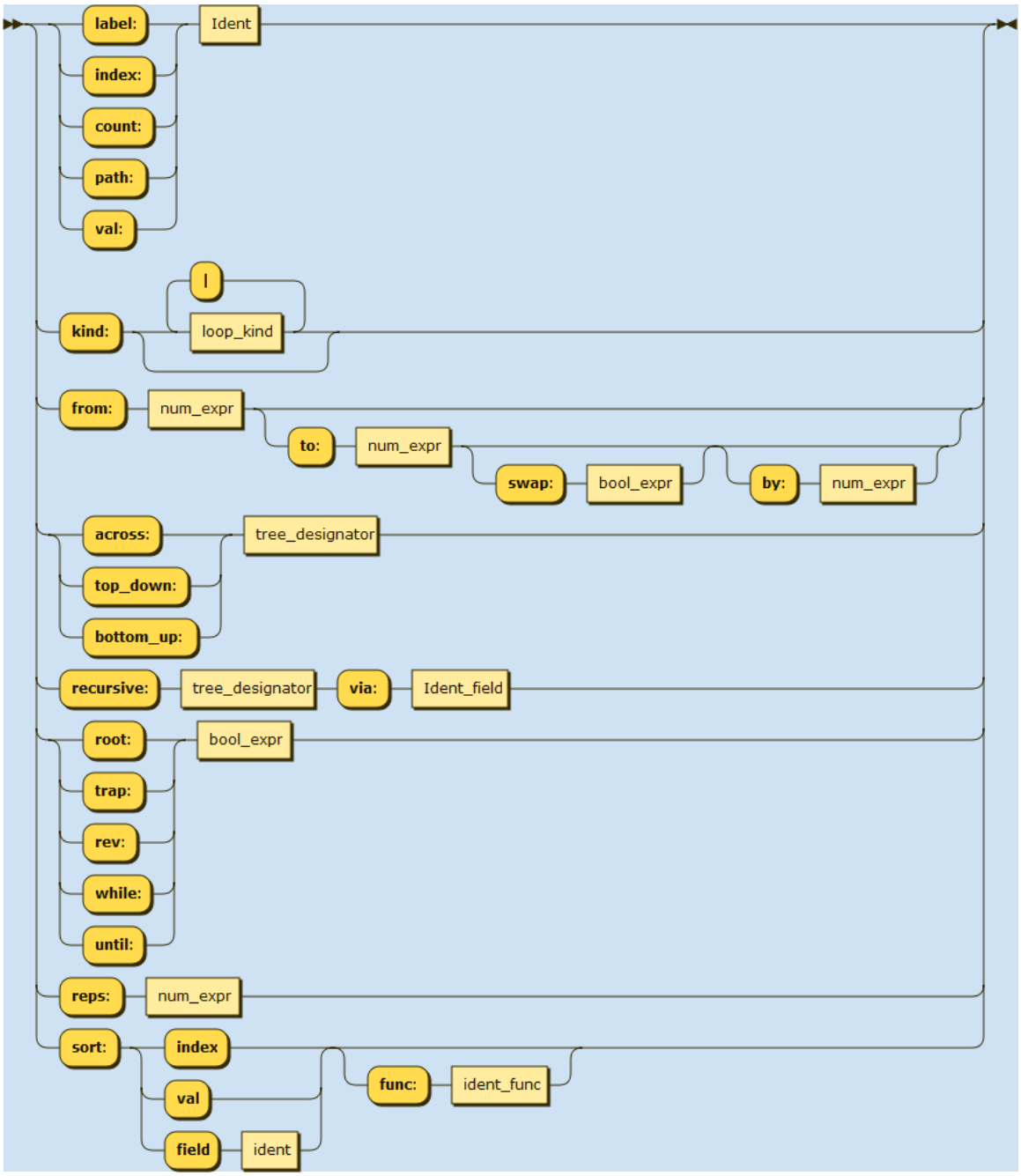
In the C language, this would be written `(x<10) ? 3 : 5`

Looping

The classic control structures of previous languages have been consolidated into a flexible iteration syntax that allows you to loop through a range of values, or visit a tree. There are many options for looping that cover all of the commonly used loop situations.



where `loop_options` is:



Inside a loop, you can exit the loop with the `exit` statement, and can skip to the bottom of the loop with the `continue` statement. Since loops can be given a label, the exit statement can use the outer label to break out of multiple loops at once:

```
loop label:outer
  loop
    exit outer   //  exit out of both loops
  // end inner loop
// end outer loop
```

The `continue` statement can also use an optional label to skip to the bottom of an outer loop.

The `exit` and `continue` statements are the last remaining remnants of the infamous `goto` statement from FORTRAN.

Example) A loop that goes forever (until an `exit` or `return` statement)

```
loop
   …statements…
```

Example) A loop that executes 10 times:

```
loop reps:10
  myfunction()
```

The above loop will execute 10 times

Example) A loop that goes until an exit statement is encountered, but has a circuit break added to detect an infinite loop:

```
loop trap:100_000
 …statements…
```

In this case the loop will go almost forever, but if it ever hits the 100,000 repetition limit a run time error will be triggered. If run time errors are off, then it will break out of the loop. Protection against infinite loops is very tricky; this syntax obviously means you have to have some idea of the maximum number of repetitions your loop could encounter, even under heavy loads. But some protection against infinite loops is helpful, because they are one of the worst forms of malfunction.

Example) A loop that executes 10 times, with an index variable that can be referenced inside the loop:

```
loop from:1 to:10 index:ix
  process_number(ix)
```

`ix` will follow the sequence: 1, 2, 3, 4, … 9, 10

Example) A loop that executes 10 times, with an index variable that counts down in reverse from 10 to 1:

```
loop from:1 to:10 rev:Y index:ix
   process_number(ix)
```

You specify the loop in the normal start to finish values, but indicate that the loop process

should be run in reverse, because the `rev` flag is ON. The loop variable `ix` will follow the sequence 10, 9, 8, 7… 3, 2, 1

Example) Loop through the values of a small set of numbers. In this example we use a matrix literal to form a one-dimensional array of numbers to iterate across:

```
loop across:[2, 3, 5, 7, 11, 13] index:ix
 check_prime(ix)
```

In this example, the ix variable will be set to 2, 3, 5, 7… etc., following the values of the matrix literal.

Example) Loop across one level of a tree, but traverse in the order of the value of the tree:

```
loop across: mytree sort:val
```

This would sort the sub-records of `mytree`. To go in descending order, add the `rev:T` keyword.

Example) Loop across an array of records, and sort by the value of a field in the record:

```
loop across:mytree sort:field age
```

In this example, we are sorting by the record field called age, assuming that `mytree` is an array of records containing the field called age. If the record doesn't have a value for age, it will have the value U, and will typically sort to the end of the list.

Example) You can specify a collating function that does a more complex job than a simple comparison of a single value. The collation function must take two formal parameters for the two records being compared. The function compares the records and returns +1, 0, or -1 to indicate the relative order of the two records.

```
loop across: mytree sort:mycollate
```

The collating function would be something like this, which compares two fields in order of priority:

```
//  this is an example collating function, returns -1 , 0, +1, U, or ERR
//  we are sorting by age then experience.
calc mycollate (
    recA : tree
    recB : tree
```

```
    ) : num

  //  check for ERR values
  if recA.age == ERR or recB.age == ERR
      return ERR

  //  check for U values
  if recA.age == U or recB.age == U
      return U

  //  sort first by age
  if recA.age < recB.age
      return -1
  if recA.age > recB.age
      return +1

  //  age is the same for both, so now sort by experience
  //  check for ERR values
  if recA.experience == ERR or recB.experience == ERR
      return ERR

  //  check for U values
  if recA.experience == U or recB.experience == U
      return U

  if redA.experience < recB.experience
      return -1
  if recA.experience > recB.experience
      return +1

  return 0
//…end collating function
```

Note: since a comparison with a `U` *or* `ERR` *value always fails, it is necessary in collating functions to exhaustively check the validity of the items being compared. If the checks for* `U` *and* `ERR` *were not present in the above collating function, if either field was* `U` *or* `ERR`*, the function would have returned 0, as the comparisons would have all dropped through.*

Example) If your loop values are such that the ending value is already greater than the starting value, the loop will be executed zero times.

```
loop from:5 to:1
  myfunction()
```

The correct way to do this loop is to go `from:1 to:5 rev:Y,` and use the reverse option to make the loop go backwards. This loop will follow the sequence: 5, 4, 3, 2, 1.

Example) The `count:` option, lets you specify an implied loop variable that is set at the top of the loop the number of times the loop has been executed, starting with 1 for the first iteration, and it is tested at the bottom of the loop:

```
loop from:117 to:120 index:ix count:mycount
    myfunction(ix, mycount)
```

Inside the loop, the variable `mycount` will follow the sequence: 1, 2, 3, 4 while the loop index `ix` is going from 117, 118, 119, 120.

Example) A loop that checks every third item in a sequence:

```
loop from:1 to:9 by:3 index:ix
  process_number(ix)
```

`ix` will follow the sequence: 1, 4, and 7. Note that the next computed value of the loop index after 7 is 10, which is past the high limit so the loop stops.

Example) A loop that goes backwards every third item:

```
loop from:1 to:9 by:3 rev:T index:ix
  process_number(ix)
```

`ix` will follow the sequence: 9, 6, and 3. Note that because the range of values was not a even multiple of the step size, the reverse sequence visited different values than the previous example which went forward, which visited 1, 4, and 7.

Example) The `exit` statement breaks out of the loop. The `continue` statement jumps to the bottom of the loop, but continues the loop.

```
loop
  statement1
  if condition1
    exit
  statement2...
```

Example) A loop that uses variables to set the beginning and ending values:

```
startval = 1
endval = 5
loop from:startval to:endval index:ix
  statement1
  statement2
```

`ix` will follow the sequence: (1, 2, 3, 4, 5).

Example) If you don't know the ordering of the starting and ending values, but want the loop to automatically swap starting and ending values, you can set the `swap:` flag to true. If you don't set the `swap:` flag, and the starting value is greater than the ending value, the loop will execute 0 times. The `swap:` flag saves you the trouble of having to swap the starting and ending values so they are in ascending order:

```
var startx = 5
var endx = 1
loop from:startx to:endx swap:T
    statement1
    statement2
```

this loop will follow the sequence: `1, 2, 3, 4, 5` because the starting and ending values were automatically swapped so that the ending value is the larger value.

Example) If you specify a combination of values for your loop that will never terminate, the compiler will flag this as an error:

```
loop from:1 to:5 by:-1
  myfunction()
```

If you are using variables for the loop limits or increments, the compiler cannot catch a potential infinite loop, so we recommend that you include a `trap` statement to protect against an infinite loop.

Example) When you attach a while condition to the loop, it is normally tested at the top of the loop:

```
loop index:loopval from:1 while:mycondition
  statement1
  statement2
```

This loop will follow the sequence 1, 2, 3, 4, 5, and at the top of the loop, the function `mycondition` is evaluated, and if not true, the loop will stop. If you want the condition to be tested at the bottom of the loop, you can add the `bottom:T` flag.

Example) You can iterate over the children of a tree, by using the `across` keyword to specify a tree that you want to traverse. There are options to perform the traversal, depth-first, breadth-first, from the top of the tree, or from the bottom leaves of the tree, and you can go

forward in ascending subscript values or in reverse.

```
var mytree = { 11:2 5:3 7:{ 1:c 2:d }}
loop index:k across:mytree
  statement1
```

This loop will traverse the tree only at the top level, stepping through the index values of 11, 5, and 7.

Example) This loop executes until a condition is true:

```
loop from:1 index:ix until:myfunction(ix)
  statements…
```

this loop will continue looping until the `myfunction()` returns true. We didn't specify an ending value, only a test; the increment is assumed to be 1 so this loop will iterate with `ix` in the sequence `1, 2, 3, 4,…` going on forever until `myfunction()` returns true.

The test is done at the top of the loop. You can either test for a true-to-false state change (via `while`), or for a false-to-true state change (via `until`). Note that if the Boolean expression used for `while` evaluates to `ERR` or `U`, the loop will be terminated immediately. This would prevent an infinite loop from happening as follows:

```
val1 = ERR
val2 = 123
loop while: (val1 < val2)
  statement1   //  this statement never gets executed
```

Similar behavior happens in the `until` syntax, where if the Boolean expression evaluates to `ERR` or `U`, the loop will be terminated. Unlike an `if` statement which will not take a branch when the expression is `ERR` or `U`, in a loop it will try to terminate as soon as possible when an error value is used.

Example: During debugging, it is very useful to put a limit on a potentially infinite loop. There is an optional keyword trap so that if the limit of repetitions is reached, if run time checks are on you can trap this error condition. This would be useful to prevent infinite loops from making a program unresponsive. If run time checks are off, the `trap` will exit the loop but not terminate the program.

```
loop index:ix from:1 until:myfunction(ix) trap:999
  statement1
```

This loop will continue looping until `myfunction` returns true. However, after 999 iterations, this loop will be stopped. The trap statement is used to trigger an error state. To limit the number of repetitions without an error, use the `reps:` keyword.

Example: During development, when you have large numbers of records that potentially could be looped across, you can use the special feature of the multiply operator in conjunction with `bool` variables, so that you can avoid several `if` statements.

```
const SHORT = T  //  turn on short loop mode
loop reps:SHORT*10 + (not SHORT)*1000
```

In the above case, if `SHORT` is `T`, then the loop will only run 10 times, otherwise it will run 1000 times.

Example: you can use conditional compilation to control when loops should be shortened, very commonly done during development:

```
loop #if $short reps:10 #end
```

Example: It is very common when using nested loops to wish to stop all the loops at once. Normally, the `exit` statement only affects the innermost loop, but you can specify a `label` for a loop, and then reference that in the exit statement. In the following example, we are searching a 3 dimensional array and when our search is successful, we stop all 3 loops by specifying the name of the loop we are breaking out of.

```
loop label:outerloop index:i from:1 to:10
   loop index:j from:1 to:10
    loop index:k from:1 to:10
      if match_function (i, j, k)
        exit outerloop
```

Example: A tree `mytree` will be traversed breadth-first, and each element that is current of value 1 is set to 123, and all others are incremented by one. And if the value is equal to 99 we quit out of the loop. In this case the loop proceeds in the top-down order, which visits the top of the tree before the lower leaves. To visit nodes in the opposite way, use the `bottom_up:` keyword.

```
var tree mytree = [ 11, 12 ; 21, 22 ; 31, 32 ]
loop top_down:mytree path:p
   -- iterator p is the fully qualified path of each element of the tree
   if p.num == 1
       p.num = 123
   elif p.num == 99
       exit
  else
       p.num += 1
```

The iterator variable `p` goes through the sequence:

```
mytree[1], mytree[2], mytree[3], mytree[1,1]…[1,2], [2,1], [2,2], [3, 1],
[3, 2]
```

<<< need tree diagram >>>>

When visiting nodes, normally the topmost root node is not included in the traversal. To force inclusion of the root node use the modifier `root:T`

You can control the kinds of nodes that are visited. Since a tree can have subscripts that are integers, enum, or strings, or links, you can specify the kinds of nodes you wish to see using the `kind:` specifier.

In this example we only want to see nodes in the tree that have a subscript that is numeric:

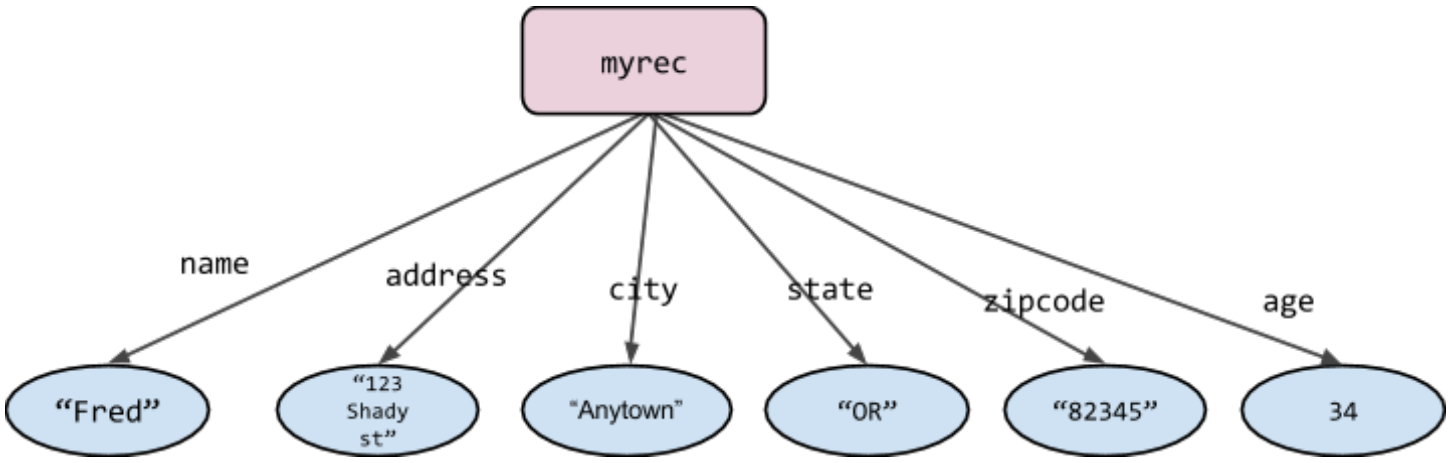
```
loop across:mytree kind:num
```

Records

Records allow you to manipulate logically related collections of data. Structured data is everywhere in the world, and inexplicably some languages have refused to incorporate them. Arrays and record types are unified in Beads, and the record fields are implemented as enumerated constant indices into a tree. This example defines a record type:

```
record a_person
  name     : str
  address  : str
  city     : str
  state    : str
  zipcode  : str
  age      : num

var a_person myrec = { name:"Fred", address:"123 Shady st",
  city:"Anytown", state:"OR", zipcode:"82345", age:34 }
```

This creates a tree structure as follows:

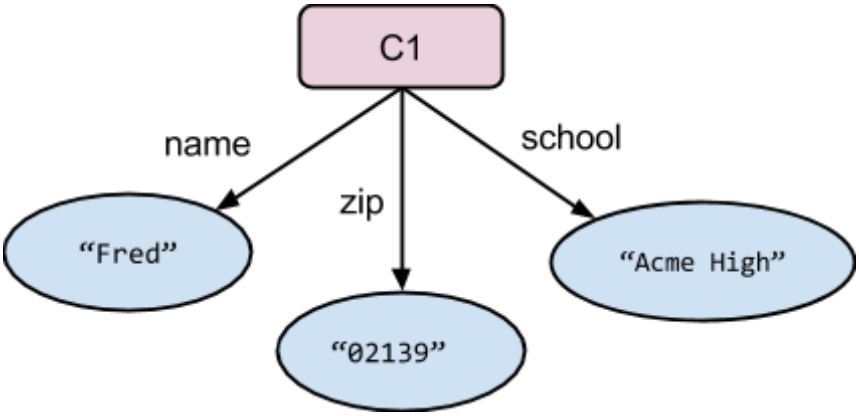


It is very common to wish to refer to different sets of fields. Field names can be shared, so below we define a different record that some of the same fields as `a_person` :

```
record a_student
    name    : str
    zipcode : str
    school  : str
```

```
var stuart <=== { name:"Stuart", zipcode:"01238", school:"Barker College" }

  const C1 = { name:"Fred", zipcode:"02139", school:"Acme High" }
```

This is the resulting tree:



In the following assignment statement, the source variable, `myperson`, is a tree containing three fields, two of which are already present in `a_person`. In a `merge` operation, the source fields will be added to the destination tree:

```
var person2 : a_person
merge stuart ===> person2     //   copies the fields name, zipcode and school  into the
```
destination record.

Records in Beads are a convenient way of creating a tree with a particular structure, and do not correspond to a fixed-size preallocated block of memory as in traditional languages. You can put multiple records into a tree node, provided the field names are distinct. It is however clearer for the reader of your code if you keep records separate.

The ability to send a record that has more fields than needed to a library routine means that a single version of a library routine can used, eliminating the need for generic modules which are very tricky to use in other languages. This is somewhat similar to inheritance in object-oriented programming, but has some added benefits. If the source record does not define a field, then a reference to the value of that field is `U`, and the program will gracefully handle this problem.

It is a common occurrence to with to refer to subsets of a record. In this example we define `a_person` with 6 fields, and `a_person_subset` with just 3 of those fields.

```
  record a_person
    name     : str
    age      : num
```

```
  address : str
  city    : str
  state   : str
  zip     : str

record a_person_subset
  name : str
  age  : num
  zip  : str
```

Records can have sub-records:

```
record a_triumvirate
  leader      : a_person
  supporter   : a_person
  odd_man_out : a_person

var t : a_triumvirate
t.leader.name = "fred"
t.supporter.name = "bob"
```

As shorthand you can include the type name of another record, which has the effect of declaring those fields at the top level. In this example, we are adding one additional field.

```
record a_person_2
  : a_person
  hobby : str
```

Tree operations

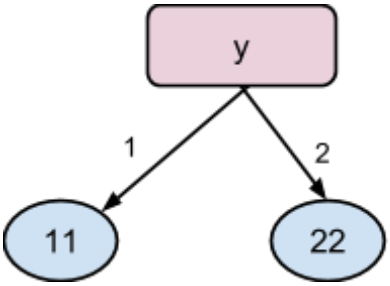
The rightwards assignment (copy) operator: `===>`

The rightwards assignment statement `x ===> y` has the same effect as the FORTRAN or C leftwards assignment statement `y <=== x,` but written left to right. As the assignment statement derives from the introduction of Arabic numerals and mathematics in the middle ages, a completely left to right expression style is the recommended practice in Beads.

```
var y : tree
```

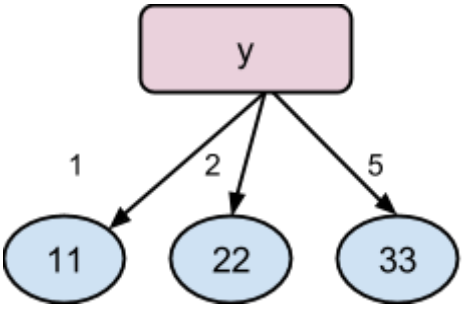
Copying a tree:

```
y <=== {11 22}    --  goes from right to left
{11 22} ===> y    --  goes from left to right
```



Copying a single value:

```
33 => y[5]      -- rightwards form
y[5] = 33       -- leftwards form
```



Note that subscripts are not required to be contiguous, and can be negative. They are constrained to be integers. Floating point value subscripts are rounded to the nearest integer, so the expression `y[2.123]` is the same as `y[2]`

Use the following record definition for the examples below:

```
record a_person
  name
  age
  sex
  eye_color
  hair_color
```

The tree copy statement

The default tree assignment operator has two forms, one for copying to the right: `===>` or copying to the left: `<===`. The copy operation replaces the destination entirely. To add values without deleting other fields that are present use the `merge` operator (which is the more commonly used operator).

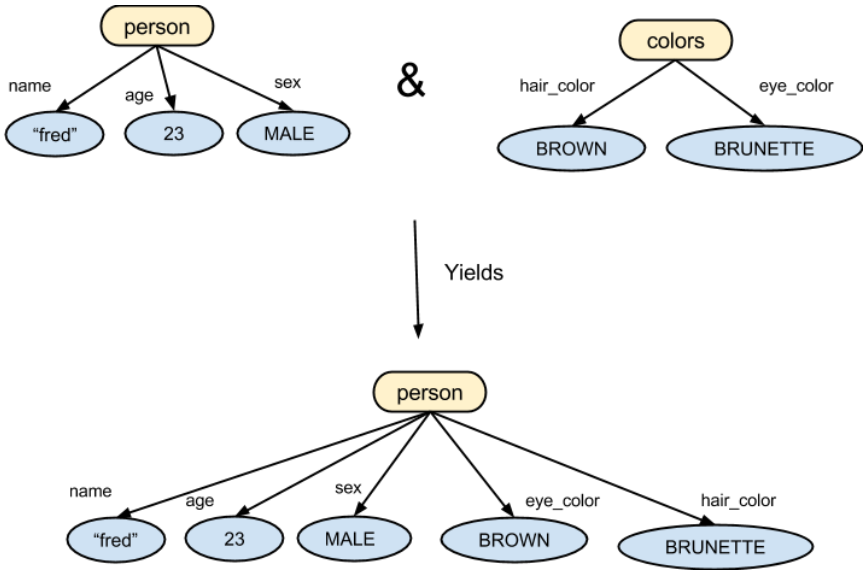
```
var person:a_person <=== { name:"fred" age:23 sex:MALE }
person <=== { name:"Bob" }
```

<<< diagram needed, showing before and after >>>

The `merge` statement

The tree merge statement takes the source, and adds it into the destination subtree, only adding or replacing values in the destination, creating a union of the fields in the destination.

```
var person : a_person <=== { name:"fred" age:23 sex:MALE }
var colors : a_person <=== { eye_color:BROWN hair_color:BRUNETTE }
merge colors ===> person
```

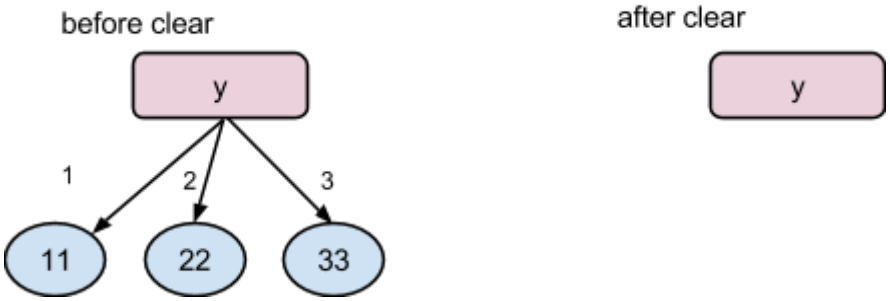


The `clear` statement

The `clear` operator erases the value of a node, and deletes all its children. It is almost the same as `trunc`, but `trunc` only deletes the children and leaves the value alone. It does not cause renumbering.

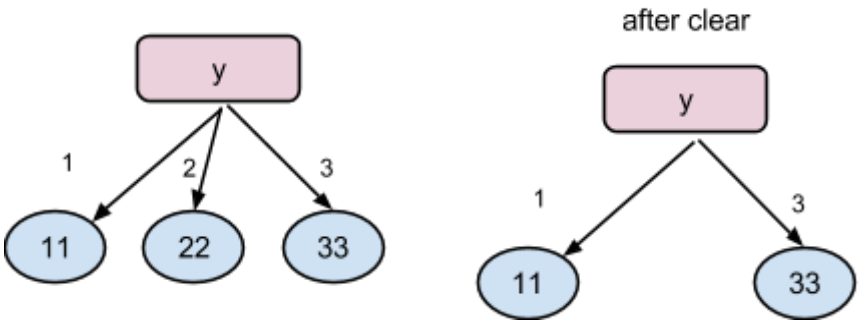
TODO: need new example…

```
var y : tree <=== [11 22 33]
clear y   -- gets rid of all the children of y
```



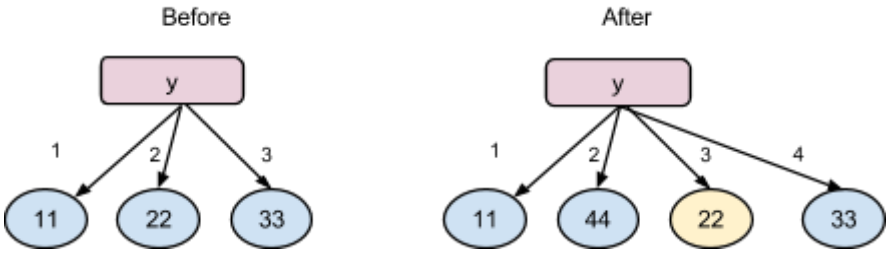
In this next example, we take a tree of three nodes and clear one of them, effectively deleting that node.

```
[11 22 33] ===> y
clear y[2]
```



The `insert` statement

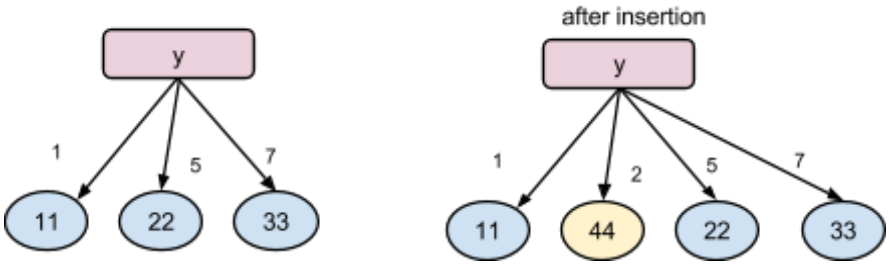
```
[11 22 33] ===> var y
insert 44 => y[2]
```



The insertion operator inserts into the array at the specified location the new value and renumbers all the subsequent subscripts by adding one. Note that if there are gaps in the subscript sequence, no adjustment of later sibling nodes will be done.

In the following example, there are gaps in the indices, so an insertion doesn't cause renumbering:

```
{ 1:11  5:22  7:33 } ===> y
insert 44 => y[2]
```

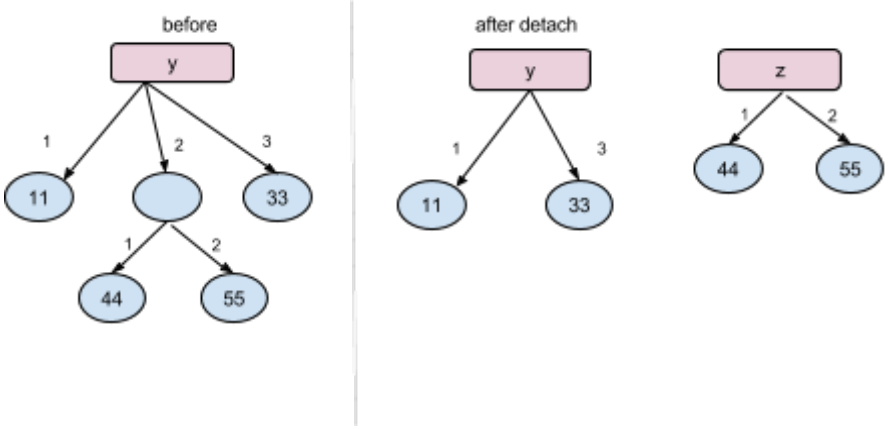


In general, insertion and deletion only affect items to the right of them, and only renumbers as many items is necessary to preserve unique integer indices.

The `move` statement

The `move` operator is equivalent to a copy, then a clear operation on the source. It moves a value or a subtree from one place to another:

```
{ 1:11  2:{ 1:44 2:55 } 3:33 } ===> var y
move y[2] ===> var z
```

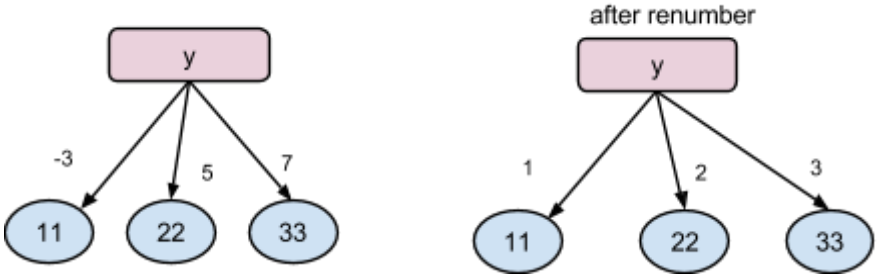


The `renum` statement

The `renum` operator renumbers the subscripts that are numeric so that they are in normal ascending sequence, starting at 1. It only renumbers the numeric subscripts, string subscripts

are unaffected.

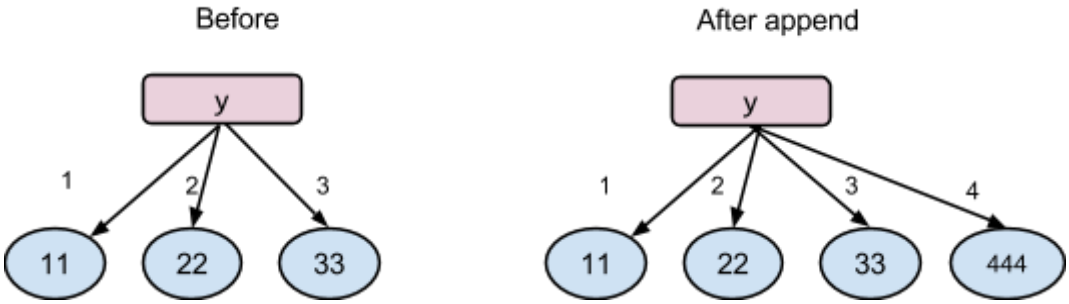
```
{ -3:11  5:22  7:33 } ===> var y
renum y
```



The append statement

A very common operation is to append a value or subtree to a tree:

```
var sub : num
var y : tree <=== [11 22 33]
append 444 => y          -- 444 becomes a new node in the tree at index 4.
```



The append operation looks at the current highest subscript, and adds 1 to calculate the new subscript. It does not affect the other elements already in the tree. If the tree has no numerical entries yet, the subscript assigned will be 1. Enumerated or string subscripts are not considered numeric and are ignored in terms of finding the highest existing subscript.

You can append subtrees or single values.

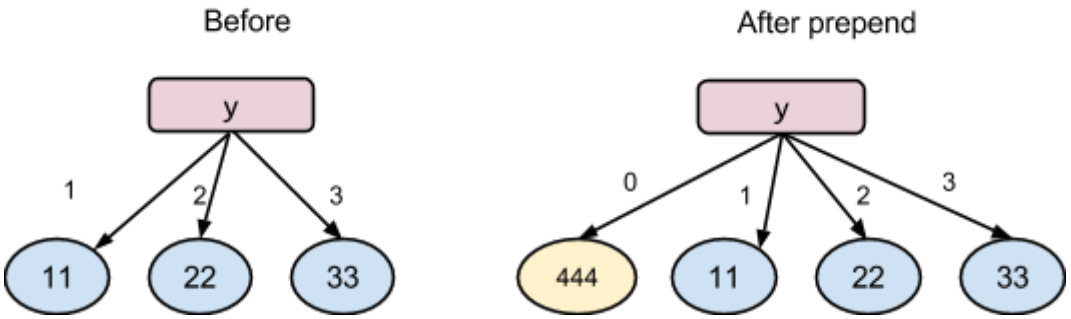
The prepend statement

The prepend statement works similarly to the append function, but instead finds the next earlier subscript. If the tree has no numerical entries yet, the subscript assigned will be 1. If you want

to know the index that was assigned, you add the `index:` suffix where you specify the variable that receives the index. The `index:` suffix is also available on the append statement.

```
prepend 444 => y index:sub
```

-- will set variable `sub` to `0`, and prepends the value `444` into the tree `y`



The `inc` and `dec` statement

As it is extremely common to increment or decrement a value by one, there is a shortcut for this operation:

```
inc x   -- same as 1 +=> x, except that inc U will result in 1

dec y   -- same as 1 -=> y, except that dec U will result in -1
```

The `touch` statement

If you want to force a value to be considered changed, but not actually modify the value, you can use the touch statement, which will trigger dependency and refresh operations:

```
touch x
```

The `toggle` statement

Since it is also very common to reverse the state of a `bool` value, there is a shortcut statement for doing this:

```
toggle x  -- is short for:

  if x == T
    x = F
  elif x == F or x == U
```

```
    x = Y
  else
    x = ERR
```

Note that `not x` is not the same as `toggle`, because `not U` evaluates to `U`

The `nop` statement

As Beads uses significant indenting, it is occasionally useful to have a no-operation statement to serve as a placeholder in an `IF` or `ELSE` clause.

```
nop
```

Functions

Unlike the C or JavaScript language, functions with no parameters don't require empty parentheses, so to disambiguate between calling a function and taking the address of a function, use the 'address-of' prefix of `adr` to indicate you want to take the address of a function rather than call the function.

```
var myval1 = action1()     -- assign the result of calling action1() into myval1
var myfunc = adr action1   --  store the address of a function into a variable
myfunc                     --  dereference the function pointer variable to call action1
```

Function parameters:

There are several kinds of function parameters:

1) The classic FORTRAN/C parameter list, where there is fixed number of parameters, and the order is significant:

```
func1 (p1, p2, p3)
```

2) Optional parameter, which receives a default value if not supplied. Optional parameters must be placed after all required parameters.

3) A variable length parameter list, indicated by preceding the final parameter by …, as in many other languages:

```
func2 (p1, p2, p3, … plist)
```

4) Named parameters, where you explicitly state the parameter names:

```
func3 (size=3, mode=1)
```

Function chaining:

In algebra/calculus, if you wish to call a series of functions, you write them in a nested notation:

```
result = func3(func2(func1(data)))
```

This is a very common operation, and to make the notation easier to read, you can chain function calls and their results using the classic pipe operator `|>` (unicode ➤) from the Unix shell:

```
func1(data) ➤ func2 ➤ func3 => result
```

This takes the data, passes it to func1, whose output is passed to func2, and then on to func3, and finally stored into result.

You can also write the above as:

```
data ➤ func1 ➤ func2 ➤ func3 => result
```

This assumes that func1, func2 and func3 were all functions declared to have a single input and return a single output. The chaining operator can yield a more fully left-to-right reading expression that uses less punctuation:

```
3.14 * val ➤ round => myresult      // same as round(3.14*round) => myresult
```

It is not a requirement that all functions in a chain have a single input. You can specify the recipient of the chained data stream by marking the parameter with a single underscore, which means "put the input here". In the next example, let's rewrite the expression:

```
func4(123, func3(func2(11, func1(data))), "third parm")
```

func4 requires 3 parameters. You can use the _ symbol to indicate which of the parameters is receiving the output from the preceding stages of the pipeline:

```
func1(data) ➤ func2(11, _) ➤ func3 ➤ func4(123, _, "third parm")
```

In the above example the output of `func1` is passed to the second parameter of `func2`, and the output of `func3` is passed as the second parameter to `func4`. Note that the pipe operator finishes the each stage in its entirety before sending the values to the next stage. Although the pipe operator version is larger in terms of space for the code, it can often be clearer in terms of indicating the order of evaluation and data flow, and cleanly expresses the pipeline nature of the expression.

A function has a name, an optional documentation comment, which is notated with 3 dashes, and parameters. Each parameter is listed indented on the next line, one parameter per line. The first field is the name, then the optional data type is specified, then an optional descriptive comment, . The end of parameters is denoted by a right parenthesis. Putting a descriptive comment after the function name, and after each parameter name is highly recommended, and compiler pragmas related to coding standards can force this to be present on every function and parameter.

```
calc cube-root( --- this function calculates the cube root of a number
  parm1 : num  --- the input value
  )
```

Since the `num` type is so common, the default data type of a parameter is assumed to be `num` if you

don't specify a data type:

```
calc calc_area( --- calculates the area of a rectangle
  width  --- the rectangle width
  height ---the rectangle height
  ) : num  --- the numeric result
```

A function can have zero parameters, in which case the parentheses after the function name can be skipped:

```
calc abort ---abort the program
```

A function can have zero parameters, and return a value. Note that dashes are also allowed in names along with underscores. This means you need to put spaces in front of minus signs, but is of great value in certain languages where the underscore is not a good breaking character. If you have no parameters you can omit the parentheses.

```
calc gen-random : num --- generates a random number
```

Functions can have any number of required positional arguments, followed by optional named arguments. In this example, we define one required parameter, and two optional parameters, which have default values of pi and 0:

```
calc indext( ---the thrammotronic index function
  rate   : num              ---the rate in meters per second
  angle  : meas = pi radians ---the angle
  gamma  : num = 0          ---the gamma factor
  ) : num  ---the thrammotronic value
```

The optional parameters must come after all required positional parameters are listed.

Special considerations for parameters with unspecified type:

If you have a function parameter that could be one of a pair or small number of types, you can use the vertical bar to list multiple types for that parameter:

```
calc myfunc (
   parm1 : num | str  ---a string or number
   )
```

To specify a wide-open flexible type for a parameter, use the type `open`, which means any kind of value can be passed.

```
calc myflex  (
  parm1 : any  ---a string or number or bitmap or…
  parm2 : any  ---a tree or number
  )
```

Inside the function, before you refer to a parameter of unspecified type, you would often need to test its data type using a standard built-in function, before using the value:

```
if datatype(parm1) == TYPE_STRING
   mystring = parm1 as str & " hello"
elif datatype(parm1) == TYPE_NUM
   myval += parm1 as num
```

In an expression, an open type variable can be casted to a different type. If you cast a `num` to a `str`, you will get a null string, and if you cast a `str` to a `num`, you will get `ERR`. There is a table in the appendix showing exactly the mapping when an incorrect conversion is attempted.

Special considerations for variable number of parameters:

```
//  DEFER variable list  ??  //
```

To specify a variable list of a parameters, use three consecutive periods to indicate the next variable has variable number of arguments. To force those parameters to have a consistent data type, specify the type (this is optional)

```
calc concatenate (
  variable mystrings : str
  )
```

Inside a function with variable number of arguments, the passed value is considered to be a tree of one dimension. Use the standard tree functions to iterate across the array, and determine the number of parameters.

Valid syntax:

```
myfunc (1 2 3)          -- all parameters supplied
myfunc (1, 2, 3)        -- commas are optional, ignored
myfunc (1 2)            -- parm3 takes default of 0
myfunc (1)              -- parm2, parm3 take default values
myfunc (1 parm2:3 parm3:4)   -- all parameters supplied
myfunc (4, parm3:11, parm2:2)  -- named parameters can go in any order
```

```
  Invalid:
myfunc ()          -- parm1 is required
```

Functions and units

If your function has a parameter of type `meas`, which is a `num` + `unit` pair, and you want to specify a default value and specific unit family, use the following syntax:

```
calc mytrig1(
  angle:angle = 30 rad    --  default value is 30 radians
  )
```

If you want to allow your function to receive a plain number (scalar), or a unit of measure, you can give the parameter a specific default unit.

```
calc mytrig2(
  angle : rad -- implied units are radians
  )
```

In the function above, you can:

```
  mytrig2 (3.14)       -- send a dimensionless quantity, radians assumed units
  mytrig2 (20 deg)     -- you are passing an angle in units of degree
  mytrig2 (30 kg)      -- will generate compile error because kg is not a unit of angle
```

This is how the standard library trig functions are defined; they have a default unit of radians, so that you can pass a dimensionless quantity of radians, or specify a physical unit of measurement.

Note that once an optional parameter is defined, all subsequent parameters must also be optional. The commas between parameters are optional. `Num` and `str` parameters are passed by value (i.e. copied into the local storage of the called function). `Tree, photo,` values are passed by reference.

If you wish to take the address of a function, for the purpose of having a function variable assignment, use the `adr` prefix to indicate "`address of`". A regular defined function with no parameters doesn't need parentheses, but to force a function call (like a function that returns a function), the parentheses will be required. To indirect through a function pointer, just call the function point as if it was a function name.

```
var myvar1 : num  = func1()    //  call func1 and store result in myvar1
var myvar2 : num  = func1      //  call func1 again
```

```
    var myvar3 : calc () = adr func1 //  store address of func1 into myvar3
```

```
To call a function variable, use parentheses to indicate calling:
```

```
    myvar3()  --  to call a function variable
```

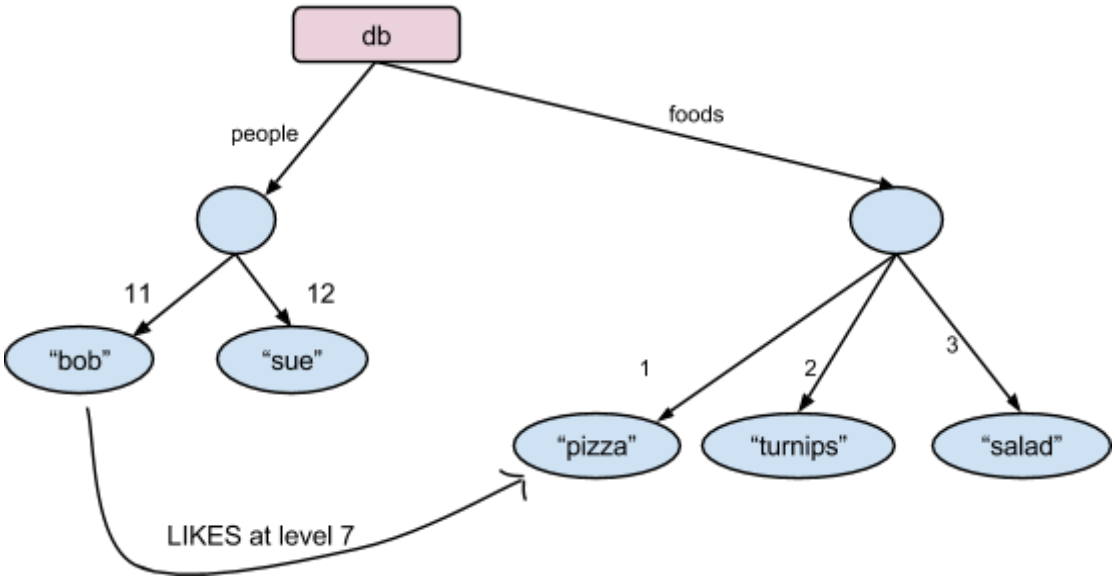
Another example of function pointer use:

```
  var funcvar1 : calc (Angle) = adr sin
  var funcvar2 : calc (Angle) = adr cos

  funcvar1 = funcvar2  //  copy a function pointer between function variables
  funcvar2 = adr tan

  myval = funcvar1(30 deg)
```

Relationships

(not fully implemented)

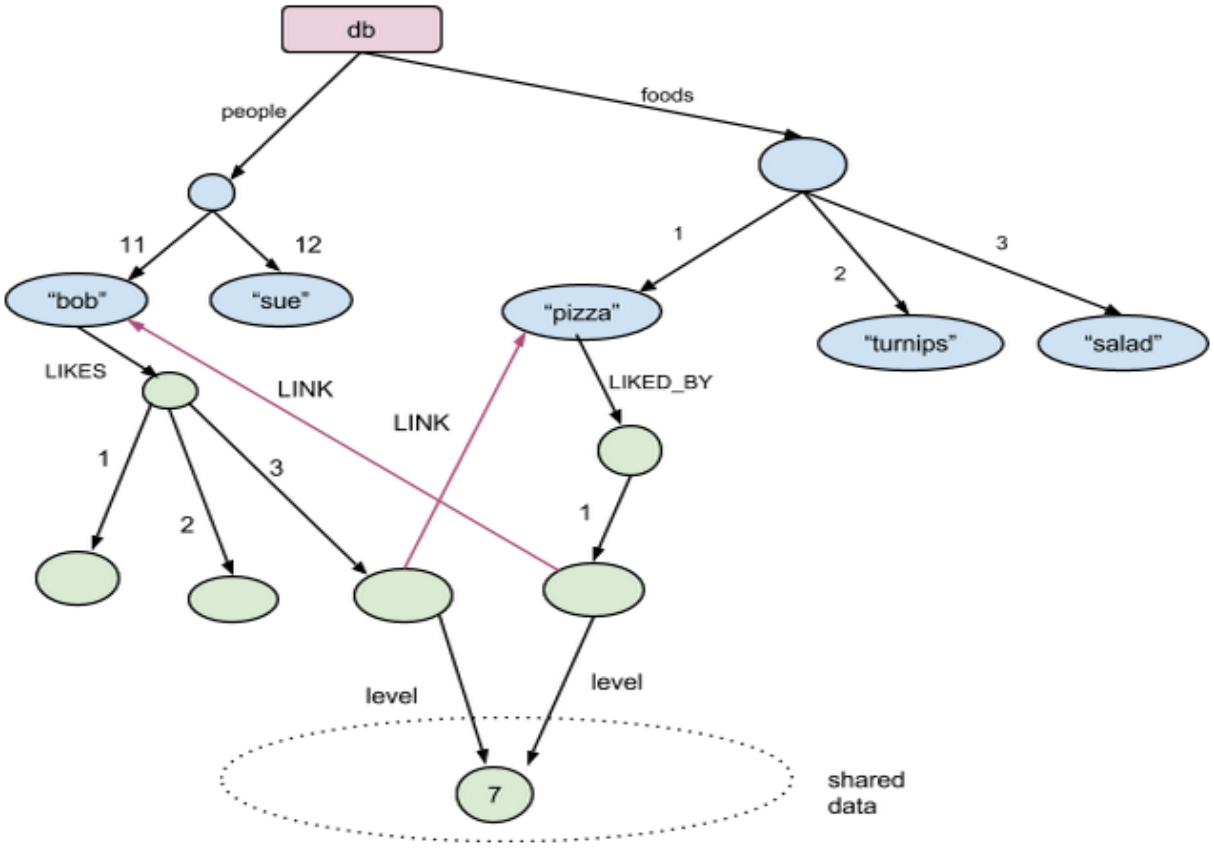
In a tree structure, one often wishes to express a relationship between two nodes. A node can point to another node, using a pointer, but a relationship consists of two pointers, brought into existence at the same time, that point to each other. The relationship has values and at the logical level is a two-way linkage between nodes. In the example below the person bob, likes the food pizza to the level of 7 out of 10. This is the logical structure that we wish to express:



In the diagram above, `bob` is at the address `db[people, 11]`. The address `db[foods, 1]` corresponds to `pizza`. To create a relationship, we use the `link` statement, which creates a bidirectional linkage between the two nodes, and also creates a third node that both nodes point to, that can store additional values. Let's say that we want to store the level of liking that bob has for pizza, in this case level 7. Since the linkage between bob and pizza can be described both ways: bob likes pizza, or pizza is liked by bob, every relationship creates a bidirectional linkage, one for the A=>B path, and one for the B=>A path. In this example, bob `LIKES` pizza, and the pizza is `LIKED_BY` bob.

```
enum
  LIKES
  LIKED_BY
  LEVEL

link db[people, 11, LIKES, ~fwd_ix] <=> db[foods, 1, LIKED_BY, ~back_ix)
db[people, 11, LIKES, fwd_ix, LEVEL] = 7
```

The link operation creates a relationship between two nodes. Each relationship can be a 1:N , N:1, or N:M relationship. In this example, bob could like multiple foods, and each food can be liked by multiple people. So the newly created relationship was put into slot #3 of the LIKES tree. Normally in the database, each path can only go downwards, but in a relationship, the linkages go across the tree to a different spot, possibly at a higher level in the tree, and then continue downwards.



In this case the path:

`db[PEOPLE, 11]` is bob

`db[PEOPLE, 11, LIKES]` is the array of 3 different things that bob likes

`db[PEOPLE, 11, LIKES, 3]` is the node holding a link to pizza or the level of liking

`db[PEOPLE, 11, LIKES, 3, LINK]` is a pointer to "pizza".

`db[PEOPLE, 11, LIKES, 3, LEVEL]` is the level of how much bob likes pizza, 7

`db[FOODS, 1]` is pizza

`db[FOODS, 1, LIKED_BY]` is the array of 1 different things that pizza is liked by

`db[FOODS, 1, LIKED_BY, 1]` is the node holding a link to bob or the level of being liked by

`db[FOODS, 1, LIKED_BY, 1, LINK]` is a pointer to bob

`db[FOODS, 1, LIKED_BY, 1, LEVEL]` is the level of the liked-by in this case 7

Note that the intensity level of the `LIKES` is the same value as the `LIKED_BY`. All data about the relationship is shared between the forward and backward link.

This notation allows a concrete representation of each node in the system, and guarantees that you can traverse all nodes by going down the tree, and by excluding `LINK` relationship paths you can easily traverse the actual nodes of the tree.

You can count the relationships of a specific kind:

`count(db[people, 11, LIKES])` would return 3 as `bob` has 3 things he likes

You can test for the forward path:

```
ix1 = find_link(db[people, 11), db[foods, 1]), LIKES)
```

Or the reverse path:

```
ix2 = find_link(db[foods, 1], db[people, 11], LIKED_BY)
```

If no relationship link was found, the index will be set to `U`. Once you have the index of a link in either direction, you can then affect the relationship, perhaps by adding a new shared value:

```
db[people, 11, LIKES, ix1, COMMENTS] = "hold the anchovies"
```

The above statement would add a comment field to the shared data section.

To destroy a relationship between two nodes, clear either side of the relationahip:

```
clear db[people, 11, LIKES, 3]
```

The above statement above will clear out the relationship between the person bob and pizza. The sub-data of the relationship, in this case the level of liking, will also be destroyed in the process. Clearing the forward link clears the back link, and clearing the back link clears out the forward link. Relationships are created or destroyed always in pairs.

Any data stored about the link is shared by the forward and back link. So if you changed the level of liking bob has for pizza to 5:

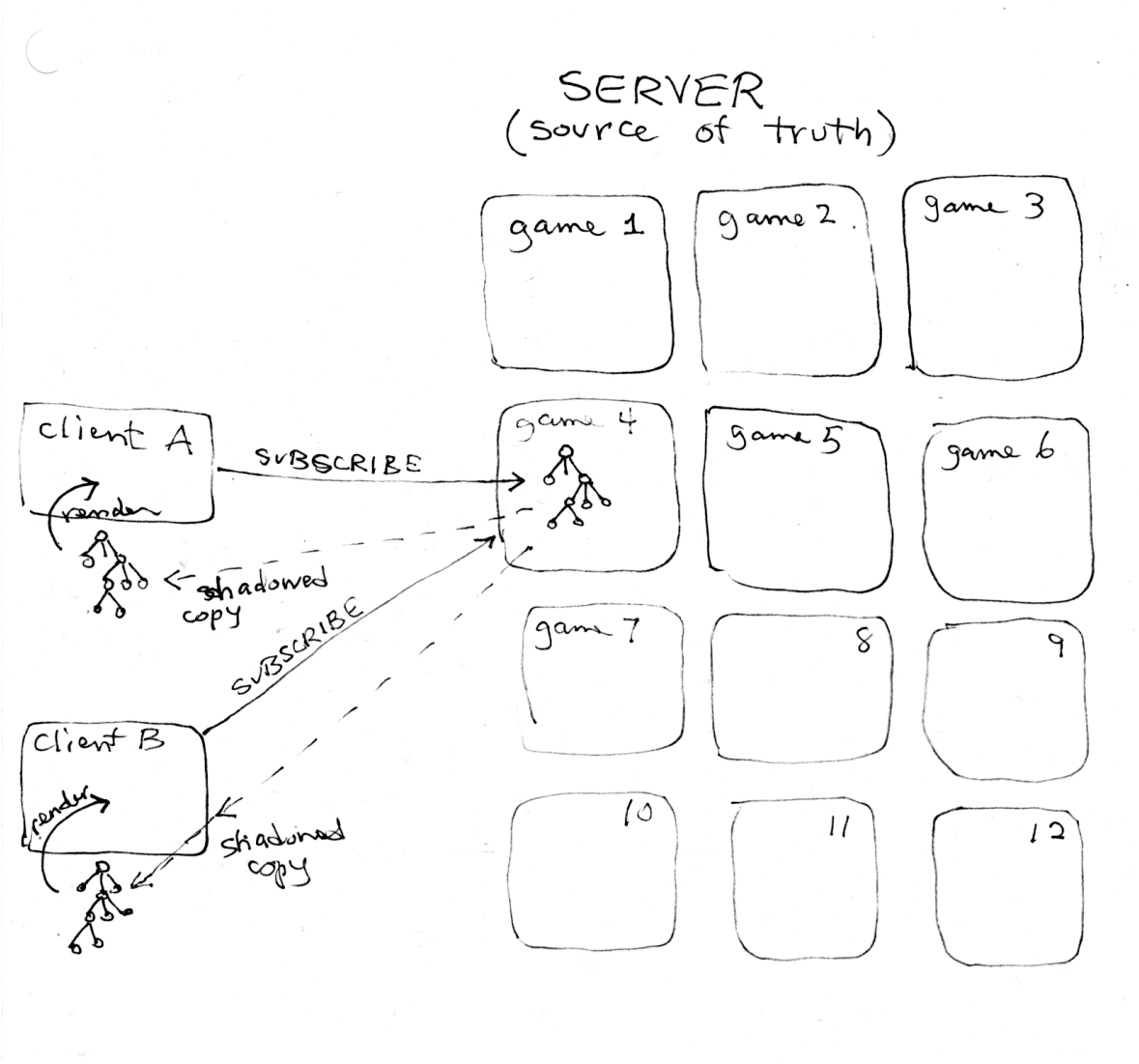
```
5 => db[people, 11, LIKES, 3, LEVEL]
```

Then it has also set the value of `db[foods, 1, LIKED_BY, 1, LEVEL]` to 5, as they are actually the same value in the tree.

Client / Server programming

The basic method of doing client/server programming in Beads is a very simple and powerful technique that is not commonly found. It relies on the tracked mutable state capability of the Beads runtime engine. In this example, we have a server holding the source of truth, which consists of 12 different game boards, each of which can have 2 players attached. Two different clients A and B have subscribed to game #4 on the server, and the state tree associated with the game has been copied to the clients; this shadow copy is a read only version of what is on the server, with a slight delay for transmission. The clients then make remote function calls on the server to play the game.

Messages are not explicitly sent by your code, instead, the runtime does the communication in the background to keep the machines synchronized.



In the first line of a Beads program you declare the purpose of the program. If it is a server program, intended to run under Node.JS for example, you would use the keyword `server`,

instead of the ordinary `program`.

When you compile for a Node.JS target, you will have available the ability to listen on sockets, and access local files. The compiler will emit `.mjs` files instead of `.html` or `.js`. A `.mjs` file is the special suffix Node.JS uses for JavaScript modules.

Unlike most systems, which use HTTP to post messages to servers, if you write your server side in Beads, you will have the ability to skip all the normal hassle of constructing an API. Instead you can use publish/subscribe feature, which connects client computers to a server, and lets the server push subtrees to the client so that each client has a shadow copy of the source of truth on the server.

The server designates a subtree of some model to be published, and then clients can subscribe to the server for the purpose of communicating with the server.

To start up a publish, the server side will call `publish_start`, which does the following:

```
publish_start(TIC_TAC_TOE, SERV_PORT, 1000, games, serv_join_game,
  serv_take_square, serv_leave_game, serv_rematch)
```

1) It establishes a secret code that will be presented in each packet sent to the server. This code prevents random hacking attempts, because even though this is a public server on a normal port, hackers will not be able to connect because they don't know the code. Any naïve system which allows hackers to poke and prod, presents a security risk. Additional security is the responsibility of your code, but this at least reduces the chance of random attacks getting any response to a negligible probability.

2) It selects the port that is going to be listened on.

3) It declares the number of kb/second that you wish to limit the server to transmit. If you have many users. It is very important not to overload a connection; else bad things happen, so you can set the server's total bandwidth ceiling in kilobytes/second. The system will do round robin with the attached users up to the limit of the bandwidth you specified.

4) It declares which state variable (always a subtree) is going to be the object of a subscription.

5) It lists all the functions that are going to be remotely callable from the client.

Effectively, the client will have a local copy of the source of truth on the server, with a slight delay. The client will not modify their local copy of the game state, but instead will perform remote function calls on the server, and any change to the state resulting from

those calls, will be automatically and invisibly transmitted to the client machine. The client machine only acts upon the changed state when the full transmission is completed. So there is no chance of a half transmission causing an invalid state to exist on the client.

In the example above, one of the calls is to serv_join_game, which by convention we recommend any remote server function names to start with serv_. This function would request joining of a game. The client has a copy of the games and can tell which ones have an opening. The client then requests to join a game. It is possible that other players at that exact moment try to join the game, and if someone beats you to the open slot, it won't cause damage because your request will be ignored.

The transmission between client and server is automatically encoded/decoded from binary for efficient transmission. To help debug, there is the ability to

To start a subscription, you use the system library function `subscribe_start`

```
subscribe_start(TIC_TAC_TOE, HOST, SERV_PORT, U, games, con1, echo:Y)
```

The arguments include the secret code, the host URL, the port to connect on, the bandwidth limit, the name of the local tree to store the copy of the server's state in, the name of the connection variable to store the connection data into, and there is the option to see server side `log` messages on the client `log`.

Being able to see the server console log on the client side, with the correct timing is of great value when debugging client/server issues.

To call a remote function you call the function as if it was a local function, but add the suffix `via` where you specify the connection variable:

```
serv_join_game(games[b.cell_seq], MY_NAME, MY_ID) via con1
```

When calling remote functions, since the answer will come perhaps 100 msec later, there is no return value; you are expecting the system to make a change in the state tree you are subscribing to.

To develop a client/server program you first program it as if the client and server are in the same machine, then you move the code that modifies the state to the server side, and add the connection information as a suffix.

Regular expressions

Regular expression patterns

In the string function library there are find/replace functions that utilize regular expressions. Rather than use the traditional UNIX escape character notation, there is a grammar to make the expressions much more readable.

Example #1) To validate an IPv4 address (e.g. 11.22.33.44),

To validate an IPv4 Internet address In JavaScript you would use the following regular expression pattern string:

```
(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\.(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5]){3}
```

if we didn't optimize with the repeat, a naïve version of the regular expression would be:

```
/(?:[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.(?:[0-9]|[1-9][0-9]|1[0-
   9][0-9]|2[0-4][0-9]|25[0-5])\.(?:[0-9]|[1-9][0-9]|1[0-9][\
0-9]|2[0-4][0-9]|25[0-5])\.(?:[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-
   5])/
```

In Beads, regular expressions are notated with a syntax that permits sub-expressions. This makes it much easier to construct patterns that can get quite long.

```
regexp octet
   group or
    digit             //  matches 0..9
    set:'1-9' digit   //  matches 10 .. 99
    '1' digit digit   //  matches 100 .. 199
    '2' set:'0-4' digit  //  matches 200 .. 249
    '25' set:'0-5'    //  matches 250 ..255

regexp IPv4
    octet '.' octet '.' octet '.' octet
```

In the above definition of the sub-expression octet, we have 5 alternatives listed in an `OR` block, covering different ranges of numbers. The definition of an IPv4 number then becomes a reference to the regular sub-expression `octet`, as it becomes a simple concatenation of 4 consecutive `octet`s, instead of the intimidating JavaScript regular syntax above. It is also uses fewer tokens, and avoids backslash characters. The character set command indicates that the following string contains a list of characters that match. Inside the character set the only escaped character is backslash, and the `-` character

indicates a range of letters, unless it is the first or last character, in which case it matches a minus sign and does not mean a range.

Example #2) To find numbers like +3.12 or -5:

In JavaScript, the expression would be:

```
/[-+]?\d+(.\d*)?/g
```

In Beads it would be:

```
regexp number global
   and
        optional set:'-+' //  the optional sign characters
        1+ digit  //  a sequence of at least 1 digit
        optional and  //  a combination of a period and some digits
            '.'
            0+ digit   //  by allowing 0, a number like +3. is valid
```

In this example, we are looking for 3 consecutive things, the set containing either one of the sign prefixes plus and minus, then at least one digit, then an optional fractional part starting with a period and zero or more digits. In this example, we do not allow a number like `.1` to match the pattern, as this is incorrect syntax in many languages, it is considered safer to write it as `0.1`

Example #3) JavaScript comment removal.

In JavaScript, comments are indicated by `//` to the end of the line, or enclosed in a block `/* … */`.
The regular expression in JavaScript would be:

```
/\/\*[\s\S]*?\*\/|\/\/.*/g
```

In Beads the same expression could be notated as:

```
regexp comments global
    or
        and
            '/*'
            0+ lazy or
```

```
            white
            not white
        '*/'
    and
        '//'
        0+ any
```

In this regular expression we are looking for any two sub-patterns, the `/*…*/` or the `//…` pattern. In the `/*…` pattern we are looking for the smallest number of characters that go between the next `*/`., and to achieve that we specify a quantity of 0 or more, but add the `lazy` suffix so that it will match the smallest number of characters. Note that the `or` of white and non-white means any character at all; we use this instead of any, because any does not include `NL` or `CR` and JavaScript comments spans multiple lines, so we use the combination of `white or not white` to match anything.

Example #4) email validation.

A simple and reliable email address pattern from:
`https://www.regular-expressions.info/email.html`

In JavaScript, the expression would be:

```
/\b[-+._%A-Z0-9]+@[-.A-Z0-9]+\.[A-Z]{2,}\b/i
```

In Beads:

```
regexp valid_email ignore_case
    boundary
    1+ set:'-+._%A-Z0-9'  //  valid lead-in characters
    '@'
    1+ set:'-.A-Z0-9'  //  valid domain name letters
    '.'
    2+ set:'A-Z'  //  final suffix has to have at least 2 chars
    boundary
```

Note that when using hyphen inside character sets, it must be the first character, because otherwise the hyphen is used to indicate character range. This is true in JS as well.

Example #5) French postal code validation

A French postal code is a 5 digit number, prefixed optionally by F-.

Example valid postal codes are F-12345 or 12345

In JavaScript, the expression would be:

```
/^(F-)?\d{5}$/gm
```

In Beads:

```
regexp french_postal_code global multiline starts ends
     optional 'F-'
     5 digit
```

Example #6) Canadian postal code validation

A Canadian postal code is a 6 or 7 digit string, with the form A1B-2C3.

In JavaScript, the expression would be:

```
/[A-Z]\d[A-Z][ \-]?\d[A-Z]\d/
```

In Beads:

```
regexp az  //  a sub-expression used several times
   set:'A-Z'

regexp canadian_postal_code
   az, digit, az // first group of 3
   optional or         // optional divider
      ' '
      '-'
   digit, az, digit    // last group of 3
```

Example #7) MAC address verification

A hardware MAC address is size pairs of hex digits: `00:aa:cc:dd:ee:ff`

In JavaScript, the expression would be:

```
[0-9a-f]{2}(:[0-9a-f]{2}){5}
```

In Beads:

```
regexp MAC_address
   2 hexdigit
   5 and
```

```
        ':'
        2 hexdigit
```

Example #8) a time expression

A time of the form `hh:mm` where hour is 00-23 and minute is 00 to 59

In JavaScript, the expression would be:

```
([01]\d|2[0-3]):[0-5]\d
```

In Beads:

```
regexp time
   or
     and
       set:"01"
       digit
     and
       "2"
       set:"0-3"
   ":"
   set:"0-5"
   digit
```

Example #9) Fancier email verification

This very complex verification system is by no means complete, but does show the exponential increase of regular expressions as you increase the nesting

In JavaScript, the expression would be:

```
/(?:[a-z0-9!#%&'*+/=?\^_`{|}~-])+(?:\.(?:[a-z0-9!#%&'*+/=?\^_`{|}~-
])+)*@(?:[a-z0-9](?:(?:[-a-z0-9])*[a-z0-9])?\.)+[a-z0-9](?:(?:[-a-z0-9])*[a-
z0-9])?/i
```

In Beads:

```
regexp EMAIL_CHAR
   //  to prevent interpretation of braces, use triple quotes around the string
   set:'''a-z0-9!#%&'*+/=?^_`{|}~-'''

//  the leadin character of a domain name doesn't allow dash
regexp DOMAIN_CHAR
```

```
   set:"a-z0-9"

//  the interior characters of a domain name allow a dash
regexp DOMAIN_INTERIOR
   set:"-a-z0-9"

//  a domain name segment has to start and end with no dashes
regexp DOMAIN_WORD
   DOMAIN_CHAR
   optional and
    0+ DOMAIN_INTERIOR
    DOMAIN_CHAR

regexp VALID_EMAIL global ignore_case
   1+ EMAIL_CHAR
   0+ and
    "."
    1+ EMAIL_CHAR
   "@"  //  the all important at sign
   1+ and
    DOMAIN_WORD
    "."
   DOMAIN_WORD
```

Example #10) HTML color specification

A hex color in HMTL can be either 3 hex digits or 6 like `#aaa` or `#aaaaaa`

In JS: `/#([a-f0-9]{3}){1,2}\b/gi`

In Beads:

```
regexp color
   '#'
   3 hexdigit
   optional group
       3 hexdigit
   end_of_word
```

Example #11) A positive number specification

A positive number is a set of digits, followed by an optional decimal point, and perhaps

fractional digits, like 123, 123., 123.45

In JS: `/\d+(\.\d*)?/g`

In Beads:

```
regexp positive_number
   1+ digit
   optional group
       '.'
       0+ digit
```

Example #12) A positive or negative number specification

A number is an optional minus sign, then a set of digits, followed by an optional decimal point, and perhaps fractional digits, like 123, 123., 123.45

In JS: `/-?\d+(\.\d*)?/g`

In Beads:

```
regexp number
   optional '-'
   1+ digit
   optional group
       '.'
       0+ digit
```

Advanced topics

Preprocessor

There is a preprocessor pass that operates at compile time.

```
@pragma  optimize:true      //  pragma flags are compiler dependent

@if..@then..@elif..@then..@else..@endif
```

example:

```
@if $checks @then
      …statements…
@elif $superduperchecks @then
      …statements…
@else
      …statements…
@endif
```

Note that the preprocessor can work on a single line:

```
const mylimit = @if checks @then 10 @else 10000 @endif
```

Beads normally uses the indenting system of Python to indicate block structure, except that only tabs are used to indicate indenting. However to facilitate computer generation of Beads code, you can switch to a non-indented section of code, by using the following equivalents to indent, and dedent and line break:

```
@{      -- equals indent
@}      -- equals dedent
@;      -- equals line break
```

To continue a line use the preprocessor symbol `@+` at the end of the line to continue.
Any leading tabs or spaces on the line following the `@+` , are effectively erased from the input stream so it is as if you kept typing on the same line. Note that a string literal cannot be continued, use the multiline string form or use string concatenation and break the string into smaller sections.

Aliases (abbreviations)

The alias is a preprocessor feature allows you to create your own shortcut for keywords in Beads. This is similar to the Assembler `EQU` statement, or the `#define` statement of the C language, which are textual substitutions performed at compile time. For example, here

we define a Chinese word as an alias for `INFINITY`:

```
@alias 无限 = INFINITY
```

Abbreviations are used frequently in Beads to centralize and share frequently used lists of parameters to the drawing functions. You just place a few style definitions at the top of your program, and it will save typing. Please note this is going to insert the tokens into the usage point; effectively the compiler is doing a find/replace, taking your single word and replacing it with the tokens. Here we define two styles:

```
@alias S_plain_text =  size:10 pt, color:BLACK, just:LEFT, font:BASE_FONT
@alias S_hilite_text = size:12 pt, color:RED, just:CENTER, font:THICK_FONT
```

And later in your program you will reference the defined style:

```
draw_text("hello", S_plain_text)
```

The style `S_plain_text` will be expanded by the compiler to become the parameters you specified at the top. An alias is slightly different than a constant; it is what would be called a macro in many other languages. We recommend using S_.... for style constants, so it is obvious to the reader that you are applying a style abbreviation.

Partial function application

Save typing a long parameter list repeatedly, by using the partial function application preprocessor command. For example, if you have a function `f1` that takes 8 parameters, and you want to call it with just two different values specified in parameters #5 and #8, create a new function `f2`, which only takes the 2 parameters you change each time:

```
@partial f2(a, b) = f1(1, 2, 3, 4, a, 5, 6, b)
```

This means that a function call to `f2 (a, b)` will create a call to `f1`, and substitute in parameters #5 and #8. The other parameters are constants. This is more efficient than creating a sub-function, as the compiler will expand this like a C macro.

Localization

In order to make your program usable and acceptable to the maximum number of people, even if it is in English only, there are multiple versions of English. For example, the phrase `"the color red"` would be spelled `"the colour red"` in the United Kingdom. So some kind of localization is

necessary for almost every program.

There are special external tools and a workflow for localization of the program's content (most importantly the string literals inside the program text), and of the program's source code itself. The Beads system is designed to promote sharing of code internationally, and the translation of the source code is a key factor in the creation of an interchangeable parts era in software.

There are tools available for use with Beads that make it easy to integrate translation dictionaries of phrases. Following a string literal with the localization key allows the string to be indexed uniquely in translation dictionaries. The two dimensional specification, group / id uniquely identifies the string.

```
mystring1 = "hi there" [mygroup/123]
```

In the example above, the translated string will be found under the string group `"mygroup"` at index `123`. There are utilities that automatically collect, extract, number, and update your source code file so that you have unique numbers for each string, and all changes are logged and tracked. To mark a string as non-localized, so that the extraction tools ignore it, use an index of 0 with no group to indicate non-translation. Non-translated strings might be used for product names:

```
"Donkey Kong"[0]    -- a non-localized string, universal in the world
```

The automatic extraction tools look for string literals that have no translation key. When first entering your strings in the program code, you just put the raw string in quotes, and let the utilities add the brackets and assign the index number.

If you wish to attach a translation note to help the human translator do a proper translation for the string, Add another slash after the index, and write the notes. In the example below the word `boot` could mean trunk of a car (British English), or a heavy shoe, and these phrases would be translated differently:

```
"look in the boot" [123/rear of car where luggage is stored]
"put on your boots" [112/heavy shoes]
```

The code above is mapped into a call to the standard library function `str_localize` by the compiler. You can directly access `str_localize` for advanced situations:

For example, you might wish to explicitly control the language to be used at execution time. The following statement retrieves the German version of a welcome string that is in the translation dictionary:

```
str_localize ("Welcome!"[myproject/13], lang:LANG_GER)
```

This code when executed will retrieve the German translation for the phrase, which is at index 13 in the translation string table called `myproject`. To query which languages are available at runtime, you can test one of your strings and see if it returns a defined string or not. The localization utilities can help you make sure that all strings are present in all supported languages. As programs grow and change over time, the utility programs are essential for keeping track of new strings added so that the program has a complete translation set in all supported languages.

When a message changes its text, the master dictionary is the source of truth, and the source code will automatically be updated by the translation tools, so in the above example when string #13 changes its English text, it will be fed back into the source code. The string form is not actually the master reference, but is there for the convenience of the reader of the code, as string #13 is not meaningful.

Sometimes the translation must vary depending on the number of items. This ability to detect single versus plural has many subtleties; in some languages there are more than two forms of a word depending on the quantity (up to six forms in Welsh!). In most European languages, there are usually just two forms: singular and plural. For example: "1 minute remaining" vs. "2 minutes remaining". Here we retrieve the correct version of a localized string depending on the quantity of the item:
[NOTE: needs some tweaking on syntax for plural translation]

```
    …assume nminutes is computed…
ss = str_localize("{nminutes} minutes remaining"[ix:12], qty:nminutes)
```

This statement would retrieve the correct form of the string by looking for a quantity match in the translation dictionary, taking into account the quantity specified, then substituting into the string the number `nminutes`. In this case we are using the translation table to retrieve not a foreign language, but our base language English, but using the ability to vary the string based on the quantity. The English translation table would look like this:

```
{nminutes} minutes remaining     QTY
1     1 minute remaining
else  {nminutes} minutes remaining
```

In the Lithuanian language however, there are 3 different forms for quantity, and zero doesn't use the same form as multiple minutes as in English. The Lithuanian translation table for this would be:

```
{nminutes} minutes remaining     QTY
```

```
0      0 minučių likusios
1      Likęs 1 minutę
else  {nminutes}-ąją likusios
```

Another advanced capability of the string table system is that you can store a 1-dimensional array of strings under a key, and use a subscript to access. If you have a string table with the key `days-of-week`, and there are 7 strings in that array, you can access the localized string array by subscripting the string:

In this case, string table `mystrings` is accessed using the key `"days of week"`, and for string number 123, then returns the 2nd entry in that list, which would likely be `"Tuesday"` when English is the current language, `"Martes"` if it was running in Spanish. In this example, we ask for the 2nd day of the week in Danish:

```
localize ("days-of-week"[mystrings/14], lang:LANG_DAN, index:2)
```

In this example, the string table for Dansk is consulted, and the key "days-of-week" is located, and then the 2nd element in the array is used for the string. For example, the Dansk translation table would appear something like this:

```
days-of-week    index
1      Mandag
2      Tirsdag
3      Onsdag
4      Torsdag
5      Fredag
6      Lørdag
7      Søndag
```

The translation system relies on a series of tables that map one language to another. There is always one hub language that allows mapping from the language of the program into the localizations. Each translation table supplies the string localizations from the base language to a second language.

Introspection

The compiler automatically creates a tree structure available anywhere inside the program that lists all variables declared at the top level in all modules. This array called `META` allows the code to know the type definitions of each variable, including the sub-records. In this way it is possible for the debugger to nicely draw the contents of a record, because one can look up the type that was specified in the source code. This does not allow you to peek inside variables

in other modules that are not exported, but is intended for writing debugging code, which needs to know the types ahead of time to draw nicely.

The variable `META` is indexed by module name, and lists all of the symbols in a module organized by kind:

```
record a_mod_def  --- a module definition
  mod_funcs : array of a_meta_rec  --- functions, indexed by func name
  mod_const : array of a_meta_rec  --- constants, indexed by name
  mod_vars  : array of a_meta_rec  --- variables, indexed by name
  mod_enums : array of str         --- enums, indexed by enum val
  mod_recs  : array of a_meta_rec  --- records, indexed by name
  mod_ufams : array of a_meta_rec  --- unit families, indexed by name
```

The elements of the tree store meta records, which have different information based on the kind of entity.

```
record a_meta_rec
  //--- data type of func returnval, or const/var, or parm
  //  array^2 of num will have typekind=TYPE_NUM, dim=2, rec=U
  vv_typek : num  --- TYPE_NUM, etc.
  vv_dim : num  ---  if present, dimensions of the array prefix
  vv_rec : str  ---  name of record type vv_typek is tree

  //--- used for function symbols
  vv_funck : num  --- function kind, FK_CALC, etc.
  vv_parms : array of a_meta_rec --- array of parms, indexed by parameter order
  vv_cat : str  --- for functions: category | subcategory | comment

  //--- used for units
  vv_canonical : str --- canonical unit family dimensions like len^2•mass^1
  vv_units : array of str --- array[str] of str

  //--- used for records
  vv_fields : array of a_meta_rec  --- array of fields in the record

  //--- used for function parameters
  vv_parmn : str  //  for parms: parm name
  vv_parmk : num  --- parameter position kind: PK_POS, PK_NAMED, PK_REST
  vv_default : str  --- string form of default value if named parm
```

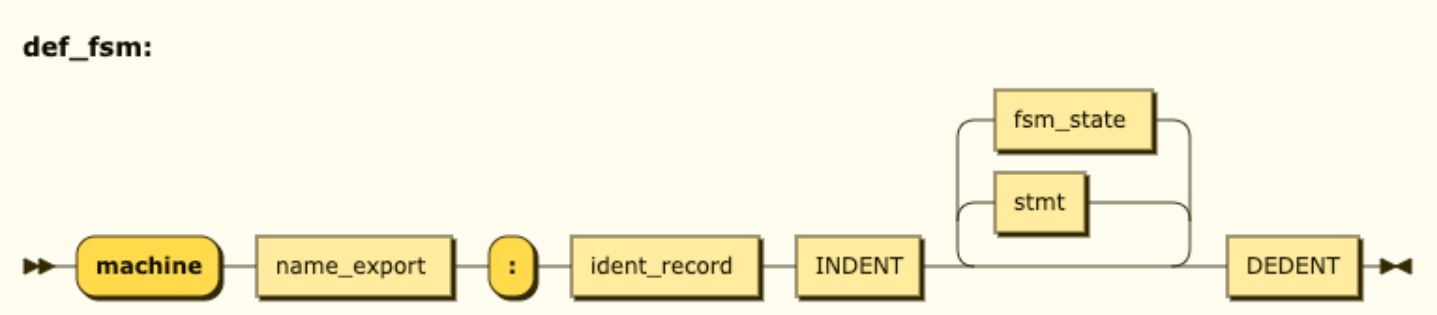
To determine the data type of a function called `mysort`:

```
META["mymodule"].mod_funcs["mysort"].vv_typek
```

The value of `vv_typek` is the type kind enum, like `TYPE_NUM` for number.

Advanced features

Finite State Machines



It is common to need to program a finite state machine. There is a special variable type called a `machine` that indicates the function has special commands available. Since you can have multiple instances of each finite state machine, you pass to the function the tree to update while the machine is running. As the machine is fed message events, which is a simple number/enum, the machine, will change state. The machine is considered to be in the first state section after it is spawned. The machine is a special kind of record variable. The `go` statement sets the next state of the machine from the inside of the machine.

Example:

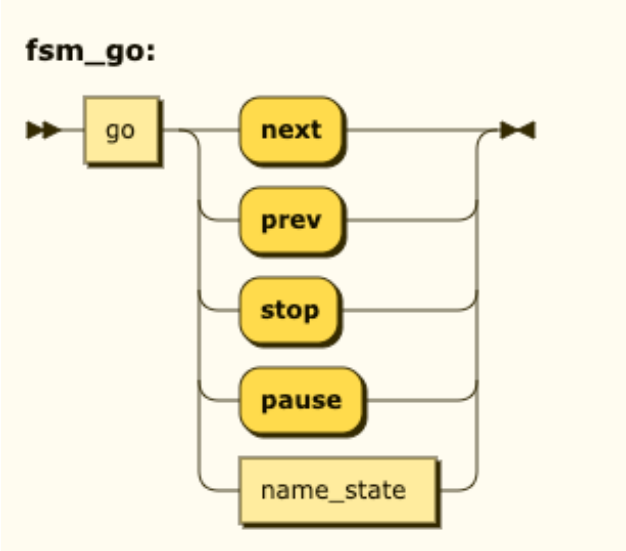
Define a state record, and create a finite state machine, using a block of data:

```
enum
  MSG_FASTER
  MSG_EXPLODE

record a_ship
  speed : num
  energy: num

var ships : array of a_ship

create my_machine at ships[1]  -- this creates a new machine using ships[1] to store
  information about the instance, in this case speed and energy
```

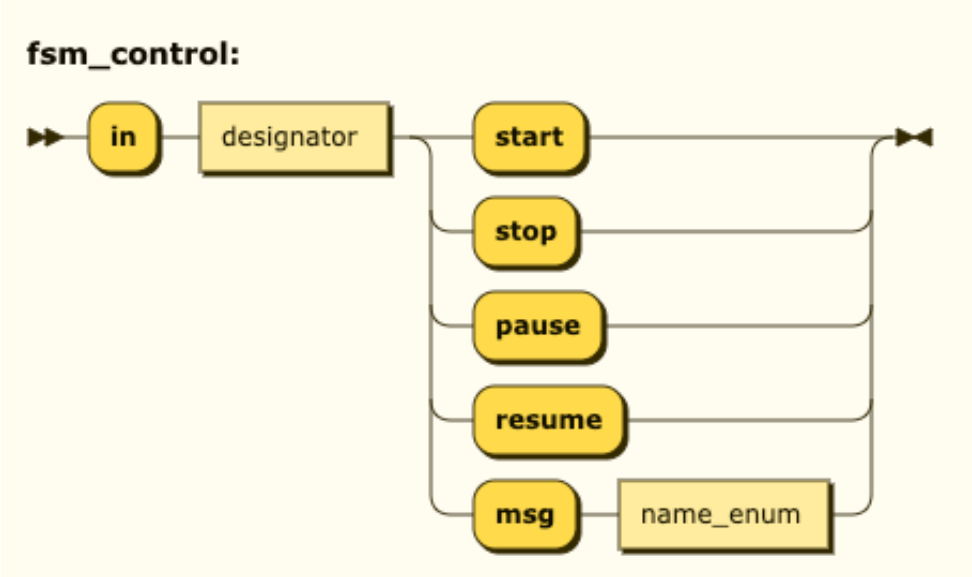
A machine block has a local state, which is stored in the subtree specified at creation time. Each state is called similar to a tracking block, and each state is its own tracking block, and events are passed to the state chunk of code. It is possible to add formal parameters to a state, in which case in order to enter the state you must supply those parameters, as a function call.

```
machine my_machine : a_ship

   state START
   | EV_ENTER  //  this message is sent on the first state upon spawning
        energy = 100
        speed = 0
        go next  // same as go CRUISING
   state CRUISING
    | EV_TICK
        ..each timer tick we run this code
    | MSG_FASTER
      energy += 100
     | MSG_EXPLODE
      //  start some animation
      …
      //  destroy ourselves
      go stop
```

Inside a machine you control the next state by using the go statement.



Outside the machine code, you control a machine variable, by using the in statement.



For example, you would say to start up a machine:

```
in myships[1] start
```

Or to send a message to the machine:

```
in myships[1] msg NETWORK_FAIL
```

Markdown string blocks

In order to be able to input markdown, a markdown block is available; this defines a string constant that is encoded using a subset of the markdown syntax:

```
mark mytext
   *this is italic*
   **this is bold**
```

Markdown blocks allow you to easily add HTML encoded text without the hassle of entering the normal tags. The following patterns are supported:

```
*italic*
**bold**
    ***bold italic***
    # H1  - header level 1
    ## H2 - header level 2
    ### H3  - header level 3
    `code`
```

Experimental / Derived quantities

(not fully implemented)

Just like a spreadsheet, you have raw input quantities that come from the user, and are stored as values, and derived quantities that use a formula. In Beads we generalize the concept of derived quantities so that given a path to a derived quantity, the derivation function uses the surrounding values in the tree to calculate the result.

For example, if we have a tree with the input quantities `INCOME` and `EXPENSES`, and a derived quantity NET, then you would declare a function to derive the NET field:

```
derive NET
    #[INCOME] - #[EXPENSES] => NET
```

the `#` prefix indicates that the path is relative to the quantity being derived.

To specify a quantity as derived, create a derive block, which is like a function to the record field in question. The following example makes all `PROD_COST` fields automatically calculated, by multiplying `PROD_PRICE` and `PROD_QTY` in any record where `PROD_COST` exists. Note that `PROD_COST` is not calculated until it is retrieved somehow (lazy evaluation). You need to allocate derived quantities, because derivation doesn't allocate them. For pattern variables you create them by using an assignment statement. To allocate a derived quantity, use the `alloc->` operator.

```
  derive PROD_COST
    #[PROD_PRICE] * #[PROD_QTY] -> PROD_COST
```

Internally there is a path associated with each instance of `PROD_COST`. If the full path to `PROD_COST` was `mydb[PRODUCTS, LIQUID, 234, PROD_COST]`, the reference `#[PROD_QTY]` inside a derive block would refer to `mydb[PRODUCTS, LIQUID, 234, PROD_QTY]`. There are circumstances where your derivation function needs to access elements higher in the tree. In that case, you declare the derivation function with parameters for the higher levels of the tree so that you can refer to those values inside the function:

```
  derive id PROD_COST
    #[id PROD_PRICE] * #[id PROD_QTY] + [PRODUCTS id DESC BOX_COST] -> PROD_COST
```

In the above example, the field `PROD_COST` is calculated using two sibling values (`PROD_PRICE` and `PROD_QTY`), and a value further up in the tree (`BOX_COST`). This allows the derivation formulas to use variables that are some distance from the record being computed.

At some place inside the derive block, you must set the value of the node that you are deriving, or you will get a compile error.

Experimental / Inherited block drawing values

(not yet implemented)

```
draw main_left promo_upper
 fill_rect(b.box, color:red, alpha:0.5)

draw main_left ?
 fill_rect(b.box, color:blue)
```

The drawing code forms a tree of code that is used to draw each sub-block. As often one shares drawing code you can use the pattern matching feature so that when the block:

```
root => main_left => promo_middle
```

is to be drawn, it looks for an exact match of that path in a draw block, and if no exact match is found, then starts examining wildcard patterns for a match. There is no draw block with an ending pattern of `promo_middle` so it looks at the parent block, which is `main_left` and looks for `main_left ?` which it does find. This pattern matching syntax permits very convenient default drawing behaviors.

The compiler can detect that you have not implemented a draw function, or that the there is a duplicate (and therefore conflicting situation). A draw function has hidden parameters passed to it, hence the need to mark the function as special, and can only be called from another draw function. A regular compute-only function cannot call a draw module. Draw functions all originate from the master draw block, which is specified at compilation time. There is one main compute function and one main draw function for a program. If you are building a console application, you have no draw module. Beads can be used to make console applications as well as graphical interactive software.

*** *contextual variables NOT YET IMPLEMENTED*

There is another feature in the draw system, which is that variables marked as contextual are visible in blocks below. This allows a lower block to refer to variables from the calling modules. It is quite common for a higher level block to select a font size for the group of items as a whole, by studying the available widths. Instead of tediously passing variables to lower blocks, you can let the lower blocks have access to the parent data. This is one of the few instances where inheritance exists in Beads. However, unlike in many other

object- oriented languages, duplicate names are not allowed in the contextual chain. Each variable name must be unique up and down in the tree of drawing blocks, so there is no uncertainty as to which value in the parent block is being referenced.

```
horz slice root
   contextual var num mysize = calc_font_size(….)
     add 100 al section_a
     add 100 al section_b
     add 100 al section_c

draw section_a
    lowlevel_draw_function("my message", size:mysize, color:red)

draw section_b
    lowlevel_draw_function("another message", size:mysize, color:red)

draw section_c
    lowlevel_draw_function("still drawing", size:mysize, color:red)
```

In the sample code above, each of the `draw` sub-blocks are able to reference the parent `draw` block's contextual variable. This saves the work of passing parameters to sub-functions. The ability to break down the major task of drawing into tiny code blocks means that we can avoid the use of long case statements and deep levels of nesting, which impede readability. Drawing functions do not have parameter lists, so contextual variables are a convenient way to pass information down to lower blocks. The compiler can detect when you are referencing a contextual variable that isn't present in the drawing chain. The compiler calculates all possible drawing block paths, and can determine if every reference is valid, perhaps a sub-block could be called in such a way as to not have the contextual variable available. Since the inheritance patterns are known at compile time and cannot be dynamically changed, the compiler is able to detect errors.

Appendices

Commonly used standard library functions

(see the `std.beads` and `str.beads` files in the `beads/lib` folder, for the definition of all the parameeters)

to_str

The `to_str` function converts a non-string data type into a string.
it has options for padding to a minimum length, etc.

solve_rect

The `solve_rect` function is one of the most commonly used function in Beads. As Beads' drawing model revolves around rectangles and sub-rectangles, you will need to be able to conveniently calculate a rectangle that fits a diverse set of constraints. Instead of using a dozen different functions, this solving function will take any sufficient set of constraints and calculate a rectangle.

The pin points of a rectangle are:

1	2	3
4	5	6
7	8	9

Some common use cases:

(A) take a larger outer rectangle, select one of the 9 pin points, and then specify the width:

```
solve_rect(basis:outer, pin:1, width:20, height:30)
```

This will take the outer rectangle, and make a sub-rectangle pinned to the upper left corner (pin #1).

(B) take a larger outer rectangle and make a sub-rectangle that is inset inside by a certain amount:

```
solve_rect(basis:outer, inset:10 pt)
```

note that inside a drawing function you can refer to points instead of pixels, as inside a drawing function there is an implied drawing context with a known dots per inch, which makes the conversion from points to pixels meaningful.

(C) given a center point, create a rectangle of a certain size:

```
solve_rect (cx:120, cy:150, width:200, height:300)
```

draw_rect

The `draw_rect` function draws a filled or framed (or both) rectangle. It can have all four corners rounded, or you can specify one corner rounding at a time. The stroke color and width of the frame can be specified, and one of the more unusual aspects of the frame is you can specify where it is drawn relative to the underlying rectangle. You can draw the frame entirely inside the rectangle (`pos = 0`), or centered on the rectangle (`pos = 0.5`, the default), or outside the rectangle (`pos = 1`). You can fill the rectangle with a color, a gradient, or a tiling bitmap pattern.

draw_oval

The `draw_oval` function is almost identical to `draw_rect`, except it draws ellipses instead of rectangles.

Built-In Unit families and their related units:

```
Scalar   -- for dimensionless quantities
   each
   dozen
   gross
   percent

Length
  meter
  angstrom
  mm
  ..etc..

Area
   sq_m
   sq_cm
   sq_mm
   ..etc..

Volume
  liter
  teaspoon
  tablespoon
  ..etc..

Time
  second
  millisec
  minute
  ..etc..

Mass
  kilogram
  gram
  tonne
  ..etc..


Angle
  radian
  gradian
  deg
```

```
revs

Frequency
 hz
 rpm

Force
 newton
 lbf

Energy
 joule
 BTU
 ..etc..

Power
 watt
 milliwatt
..etc..

Pressure
 pascal
 bar
 ..etc..

Speed
 m_per_sec
 km_per_hr
..etc..

Temperature
 degK
 degC
 degF
```

Built-in constants

Built-in constants:

`U`	Used for undefined value
`ERR`	Used for error value
`T`	Used as Boolean yes/true/on
`F`	Used as Boolean no/false/off
`INFINITY`	infinity
`-INFINITY`	negative infinity
`TAU`	2 * pi
`PI`	pi, 3.14159…
`E`	Euler's number, the base of natural logarithms 2.18..
`GOLDEN_RATIO`	golden mean constant, 1.618...

Localization workflow

When you compile a Beads module, the compiler will generate a glossary file similar to this:

#beads	level	1	gloss
angle	VAR	zzz	yyy
CENTER_X	CONST	zzz	yyy
CENTER_Y	CONST	zzz	yyy
day_hand	CONST	zzz	yyy
g	VAR	zzz	yyy
hour_hand	CONST	zzz	yyy
watch	MODULE	zzz	yyy
watch_background	CONST	zzz	yyy

The glossary file will have a name like module~gloss.txt
The first line is reserved for declaring the purpose of the file
The subsequent lines contain the fields:

A) The string in the base language
B) The part of speech in terms of program code (Constant, function, variable)
C) A placeholder for the translation of A
D) A placeholder for the translator name

A utility program would probably first attempt translation via Google Translate facility or some other translation by computer program. Then a human translator would fix up the mistakes and make it a better translation. The file would then be renamed after editing to `module~xlate~eng-amh.txt` for an Amharic (Ethiopian) translation:

#beads	level	1	xlate:amh
angle	VAR	ማዕዘን	google
CENTER_X	CONST	ማዕከላዊ_ክ	google
CENTER_Y	CONST	ማዕከላዊ_ይ	google
day_hand	CONST	ፒክ_ሰንበት_ቀን	google
g	VAR	ሰ	google
hour_hand	CONST	ፒክ_ሰልፍ_ሰዓት	google
watch	MODULE	ዳራውን	google
watch_background	CONST	ዳራውን_ይመልከቱ	google

The module translation file is then injected into a master translation dictionary, which is a database of all the strings and names in the source code. Once the translation dictionary is built, then the program code can be checked out in another language, edited, and then put back into a source code repository. The translation system guarantees that all translations are unique so

that a bidirectional translation can be accomplished.

Lexical Structure

1) Beads source code files are stored only in the UTF-8 encoding. No characters below blank are permitted except for tab, newline, and carriage return. Files using LF (Unix, Mac), CR (old Mac), or CRLF (Windows) as line delimiters are interchangeable, and are converted to LF. The file suffixes are as follows:
   `.beads` -- beads source file
   `.beads_syms` -- definitions module for a compiled library module
2) Single line comments begin with double slashes `//` or `--` or `====`, and continue to the end of the line. A `---` comment is used for document comments which are stored in the code.
3) Single line Inline comments are contained inside backquotes: `` `a minor point` `` (no nesting or line spanning).
4) Block comments begin with `/*` and end with `*/`, or you can use `(*` and end with `*)`. By alternating between the two forms of block comments, it makes it much easier to tell where the ending delimiter is.
5) Lines containing only blanks, tabs, and comments are ignored entirely (they have no effect on indentation level).
6) Aside from blank lines and multiline string literals, tabs immediately at the beginnings of lines are significant. The amount of indentation of a line is the equivalent number of tabs at the beginning of that line.
7) If one line is indented more than the preceding line, it corresponds to an `INDENT` token. It is an error for the first non-blank line in a file to be indented.
8) If one line is indented less than the preceding line, it is taken to be preceded by `DEDENT` tokens. The end of a file is preceded by enough `DEDENT` tokens to match all unmatched `INDENT` tokens.
9) String literal delimiters must be matched in pairs, even in comment sections.

Indent is normally indicated by the number of tabs beginning a line, but you can use special preprocessor commands `@[` and `@]` to simulate tabs, particularly handy when generating code by software. There is also a simulator for a line break `@;` and to mark line continuation use `@+` to start a continuation, and `@-` to end a continuation.

Exact specification of closed arithmetic space

Addition:

+	0	2	INFINITY	-INFINITY	U	ERR
0	0	2	INFINITY	-INFINITY	U	ERR
2	2	4	INFINITY	-INFINITY	U	ERR
INFINITY	INFINITY	INFINITY	INFINITY	0	U	ERR
-INFINITY	-INFINITY	-INFINITY	0	-INFINITY	U	ERR
U	U	U	U	U	U	ERR
ERR	ERR	ERR	ERR	ERR	ERR	ERR
Enum	ERR	ERR	ERR	ERR	ERR	ERR

as this table shows, the Error value `ERR` has precedence over other meta values, and the undefined meta-value `U` has the next highest.

Subtraction (first term is by column):

-	0	2	INFINITY	-INFINITY	U	ERR
0	0	2	INFINITY	-INFINITY	U	ERR
2	-2	0	INFINITY	-INFINITY	U	ERR
INFINITY	-INFINITY	-INFINITY	0	-INFINITY	U	ERR
-INFINITY	INFINITY	INFINITY	INFINITY	0	U	ERR
U	U	U	U	U	U	ERR
ERR	ERR	ERR	ERR	ERR	ERR	ERR
enum	ERR	ERR	ERR	ERR	ERR	ERR

Multiplication (first term is by column):

(note that in multiplication you can use `T` as a synonym for `1` and `F` as a synonym for `0`, so as to eliminate `if` statements to add in or ignore terms.

*	0	2	T	F	INFINITY	-INFINITY	U	ERR
0	0	0	0	0	0	0	0	ERR
2	0	4	2	0	INFINITY	-INFINITY	U	ERR

T	0	2	1	0	INFINITY	-INFINITY	U	ERR
F	0	0	0	0	0	0	U	ERR
INFINITY	0	INFINITY	INFINITY	0	0	-INFINITY	U	ERR
-INFINITY	0	-INFINITY	-INFINITY	0	-INFINITY	INFINITY	U	ERR
U	0	U	U	0	U	U	U	ERR
ERR	0	ERR	ERR	ERR	ERR	ERR	ERR	ERR
enum	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR

Division (first term is by column):

/	0	2	INFINITY	-INFINITY	U	ERR
0	0	INFINITY	INFINITY	-INFINITY	U	ERR
1	0	2	INFINITY	-INFINITY	U	ERR
INFINITY	0	0	1	-1	U	ERR
-INFINITY	0	0	-1	1	U	ERR
U	0	U	U	U	U	ERR
ERR	0	ERR	ERR	ERR	ERR	ERR
enum	ERR	ERR	ERR	ERR	ERR	ERR

Exponentiation (first term is by column):

^	0	2	INFINITY	-INFINITY	U	ERR
0	1	1	1	1	U	ERR
2	0	4	INFINITY	-INFINITY	U	ERR
INFINITY	0	INFINITY	INFINITY	-INFINITY	U	ERR
-INFINITY	0	0	0	0	U	ERR
U	0	U	U	U	U	ERR
ERR	0	ERR	ERR	ERR	ERR	ERR
enum	ERR	ERR	ERR	ERR	ERR	ERR

Arithmetic comparison operations (<, <=, >=, >):

<	0	1	INFINITY	-INFINITY	U	ERR	enum
0	F	F	F	T	U	ERR	ERR
1	T	F	F	T	U	ERR	ERR
INFINITY	T	T	F	T	U	ERR	ERR
-INFINITY	F	F	F	F	U	ERR	ERR
U	U	U	U	U	U	ERR	ERR
ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
enum	ERR	ERR	ERR	ERR	ERR	ERR	ERR

Equality operators (==, <>):

==	0	1	INFINITY	-INFINITY	U	ERR	enum
0	T	F	F	F	F	F	F
1	F	T	F	F	F	F	F
INFINITY	F	F	T	F	F	F	F
-INFINITY	F	F	F	T	F	F	F
U	F	F	F	F	T	F	F
ERR	F	F	F	F	F	T	F
enum	F	F	F	F	F	F	T (if same)

Logical NOT operator:

The `not` operator can only be applied to a `bool` expression, which has 4 possible values:

not	F	T	U	other
	T	F	U	ERR

TOGGLE operator:

The `toggle` operator is almost the same as NOT, except the toggle command will turn an undefined into true. This is handy when you are flipping a value and don't expect initialization to necessarily have happened.

toggle	F	T	U	other
	T	F	T	ERR

The AND, OR and XOR operations operate conservatively:

Logical AND operator:

and	F	T	U	other
F	F	F	U	ERR
T	F	T	U	ERR
U	U	U	U	ERR
ERR	ERR	ERR	ERR	ERR

or	F	T	U	other
F	F	T	F	ERR
T	T	T	U	ERR
U	U	U	U	ERR
ERR	ERR	ERR	ERR	ERR

xor	F	T	U	other
F	F	T	U	ERR
T	T	F	U	ERR
U	U	U	U	ERR
ERR	ERR	ERR	ERR	ERR

Conversion between types

`int32` ➔ `num`

all `int32` values can be expressed in a `num` without loss of precision

`card32` ➔ `num`

all `card32` values can be expressed in a `num` without loss of precision

`num` ➔ `int32`

```
NOT automatic
use standard conversion function that has all the cases:
     val(int32, myval, U=22, ERR=23, INFINITY=24, -INFINITY=25, ENUM=26)
```

if `num == U`, then val is `INT32_U`

if `num == ERR`, then val is `INT32_ERR`

if `num` is an enum value, val is `INT32_ENUM`

if `num == INFINITY` then val is `INT32_INFINITY`

if `num == -INFINITY` then val is `INT32_-INFINITY`

if `num` is outside 32 bit range, then pick closer of `INT32_INFINITY` or `INT32_-INFINITY`

`num` ➔ `card32`

```
NOT automatic
use standard conversion function that has all the cases:
     val(card32, myval, U=22, ERR=23, INFINITY=24, -INFINITY=25 ENUM=26)
```

if `num == U`, then val is `CARD32_U`

if `num == ERR`, then val is `CARD32_ERR`

if `num` is an enum value, val is `CARD32_ENUM`

if `num == INFINITY` then val is `CARD32_INFINITY`

if `num == -INFINITY` then val is 0

if `num` is outside 32 bit range, then pick closer of 0 or `CARD32_INFINITY`

`meas` → `num`

The magnitude of the unit is copied to the `num` without loss of precision

`num` → `meas`

The `meas` unit will be set to `scalar` unless otherwise specified

Regular Expression Correspondence

to facilitate creation of regular expressions, a graphical tool is provided that will allow quick creation of a new regular expression.

The old style JS expression patterns are compared with Beads syntax as follows:

Metacharacter		GREP encoding	BEADS notation
	Any character but end of line	`.`	`any`
	Beginning of line	`^`	`starts`
	End of line	`$`	`ends`
	Digit (0-9)	`\d`	`digit`
	Hex digit	`[0-9a-fA-F]`	`hexdigit`
	Cardinal number (positive integer)	`[0-9]+`	`card`
	Integer	`-?[0-9]+`	`int`
	Decimal number	`[-+]?[0-9]+(.[0-9]*)`	`num`
	Alphabetic (no numbers)		`alphabetic`
	Boundary	`\b`	`boundar`
	Beginning of document	`\A`	`starts`
	Any character of a set a,b,c	`[abc]`	`Set:'a-c'`
	NOT (Any character of a set)	`[^xyz]`	`Not …`
	Character in range	`[a-z]`	`Set:'a-z'`
	untypeable characters	`\\, \r, \n, \t, \x{304f}`	
	Any word character (a-z, A-Z, 0-9, _)	`\w`	`alphanum`
	NOT (any word char)	`\W`	`Not alphanum`
	Any whitespace (space, tab, CR, LF, FF)	`\s`	`white`
	NOT (any whitespace)	`\S`	`Not white`
Repeat count			
	Zero or more (greedy)	`P*`	`0+`
	Zero or more (non-greedy)	`P*?`	`0+ lazy`
	One or more (greedy)	`P+`	`1+`
	One or more (non-greedy)	`P+?`	`1+ lazy`
	Zero or one (greedy)	`P?`	`Optional`
	Zero or one (non-greedy)	`P??`	`Optional lazy`
	Exactly 3 repetitions	`P{3}`	`3`
	At least 3 repetitions	`P{3,}`	`3+`
	Between 3 and 5 repetitions	`P{3,5}`	`3-5`
Subpatterns			
	Match a pattern and remember it	`(patt)`	`group`
	Match a pattern and name it	`{?P<myname>patt}`	
	Use a backreference to a pattern	`\1, \2… \99`	
Alternation			
	Match pattern1 or pattern2	`Pat1 | Pat2`	`or`
Replacement patterns			

	Subpatterns	`\1, \2, …`	
	Text matched by entire search pattern	`&`	
	Text matched by subpattern name	`\P<myname>`	

Comparison with other languages

Any language with additional libraries can do almost anything; the key comparison is therefore what is contained inside the language, without having to learn extensive API's.

	Beads	Rust	Java	C/C++	Ruby	HTML5	Swift	Excel
Automatic recalculation of derived items	■							■
Automatic redraw of affected areas	■					■		■
Protected Arithmetic	■	■				½		■
Extensible record type	■		½		■	½	■	
Tree operations	■					½		
Globally unique enum constants	■						■	
Unification of arrays and records	■					½		
Multidimensional arrays	■	■	■	■		■	■	■
Extendable string conversions			■		■		■	
Draw shapes	■					■		■
Handle mouse/keyboard/touch events	■					■		■
Schedule actions in the future	■	½				■		
Lazy evaluation of derived quantities	■							
Renaissance proportion layout method	■							
String translation support features	■						■	
Omniscient debugger		½		½			½	½

Language Scales of sophistication

Type protection

1) easy to make a mistake; turn a number into a string accidentally
2) silent incorrect conversion (JavaScript)
3) some type checks
4) strong implicit types (Beads)
5) Airtight range checks, etc. (Modula2)

Arithmetic safety

1) + operator overloaded, can't tell what operator is actually being used (JavaScript)
2) overflows undetected
3) selective control of overflow/underflow detection (Modula-2)
4) improved DEC64 arithmetic 0.1 + 0.2 does equal 0.3 (planned for Beads)
5) infinite precision (Mathematica)

Primitive data types supported

1) numbers, strings, Boolean
2) includes one dimensional arrays (JavaScript)
3) includes multi-dimensional arrays
4) includes structured types, dates, records, sounds, etc.
5) includes graph structures (Beads)

Graphical model sophistication

1) none, all drawing is done via library routines
2) console drawing built into language
3) 2D primitives, unstructured
4) 2D primitives, strong structuring (Beads)
5) 3D drawing (unity)

Database sophistication

1) external to language (JavaScript)
2) indexed sequential or hierarchical model
3) relational database built in or merged into language (PHP)
4) entity-relation database built in
5) graph database built-in (Neo4J, Beads)

Automatic dependency calculation (also called lazy evaluation)

1) none (C, JavaScript)
2) evaluation when needed of quantities
3) automatic derivation of proper order to evaluate (Excel)
4) automatic derived virtual quantities (Beads)

5) automatic calculation of code to execute, with backtracking (PROLOG)

Automatic dependency drawing (also called auto refresh)

1) none (C, JavaScript)
2) simple redraw support (Win32)
3) automatic redraw using framework (React)
4) automatic redraw without having to use framework
5) automatic execution of drawing code that references changed quantities (Beads)

Future features under consideration

SVG graphics

Inside a `draw` function you can use the vector drawing primitives to draw filled/stroked/filled+stroked shapes. A shape block is a sub-block of a draw function, and defines a stroke and fill definition on the outside the shape block, and inside you execute the shape definition functions `rectangle`, `oval`, `move`, `line, curve` and `close` to achieve a basic set of drawing primitives. Each geometric path described can be filled with a solid color, a gradient, or filled with a bitmap pattern.

For example, to draw a blue rectangle with a red frame:

```
shape strokecolor=RED stroke=2 pt fillcolor=BLUE opacity=0.5
   rectangle x1 y1 width1, height1
```

(rectangle with red stroke, 2 points thick, filled with blue, at transparency 50%)

```
shape strokecolor=BLUE stroke= 14 px
   oval x1 y1 width1 height1
```

(stroked ellipse with no fill)

```
shape fillcolor=GREEN
   rectangle x1 y1 width1 height1
```

(filled rectangle, no stroke)

```
shape  fillcolor=BLUE strokecolor=RED stroke=2 px
   moveto x1 y1
   lineto x2 y2
   lineto x3 y3
   close    //  closes path so a fillable shape
```

(stroked & filled triangle)

```
shape  fillcolor=GREEN
 moveto 100 0
 curveto 100 0 100 50    //  first two numbers are XY of the control point
 curveto 100 100 50 100
 curveto 0 100 0 50
 curveto 0 0 50 0
```

(filled quadratic Bezier curve shape which resembles a circle)

---

future math functions

Imaginary (Complex) number functions from the module `math_im`:

- `im_abs`
- `imaginary`
- `im_argument`
- `im_conjugate`
- `im_cos`
- `im_div`
- `im_exp`
- `im_log`
- `im_log10`
- `im_log2`
- `im_power`
- `im_product`
- `im_real`
- `im_sin`
- `im_sqrt`
- `im_sub`
- `im_sum`

Matrix Arithmetic Functions from module `math_matrix`

- `matrix_invert`
- `matrix_transpose`
- `matrix_uniform`
- `matrix_identity`
- `matrix_multiply`
- `matrix_divide`
- `matrix_add`
- `matrix_subtract`
- `matrix_expand`
- `matrix_size`