Language

General syntax and semantics

```
//             Statement
//                 v
//  ,----------------------------.
//     input parameters
//             v                .- optional statement separator (just newline is sufficient)
//       ,----------.           v
    max (x, y, 0, 100) -> my_int ;
//  `----------------'    `----'
//         ^                 ^
// function/expression     labelled outputs
```

A statement is made up of an "expression" on the left-hand-side, and "labels" on the right-hand-side.

There can be zero, one, or many outputs to label per statement, depending on the expression.

As with most languages, the input parameters of a function can themselves be expressions, though it's important that each such expression has only one output.

Subcontracting and Abstraction participation are also considered functions, where the inputs and outputs are specific to the defined interface.

The language is declaritve**Mosaic** macros cannot be exported to PDF., **not** imperative

Mosaic macros cannot be exported to PDF., and labels are immutable. That is, a label can be defined (and assigned a value) only once, and the ordering of statements does not affect the time at which labels get values.

Literal values

Inputs and Outputs are strongly typed. The most easily comprehended types are what we call "immediate" types. These are generally values which an Agent can manipulate directly for the purpose of communication with other Agents at design-time.

We refer to "immediate" types by capitalising their name, thus distinguishing them from "abstractions". `INTEGER` and `STRING` are the most obvious cases, and we are allowed to use "literal values" in our Autopilot (and Pilot) source to set them with a value.

Integers

We can set `INTEGER` s with literal values by use of a few notations; the choice should be based on what's most comfortable in a given situation.

```
42            // decimal 42
-0x2A         // hexidecimal representation of -42
0o52          // octal representation of 42
-0b00101010   // binary representation of -42 (*not* interpretting as twos compliment)
// The integers can range from -2^63 to (2^63)-1
```

Again notice that the type is `INTEGER`. We do not support floating point (fractional, etc) immediate value types.

Strings

As with most language, we also have numerous options for setting `STRING` s with literal values.

```
"Hello\n"                       // a string with six characters, the first five being "Hello", and the last being a Line Feed
"Hello\x0a"                     // same as above, but using hex character escape sequence
"\x48\xb8"                      // two hex bytes; notice that each one must be escaped separately
"This \"quote\" and \\ backslash"   // quotes and backslashes must be escaped
```

Declarations

defaults

`defaults: <layer>, <variation>, <platform>, <user>`

define the default identities to be used inside the Pilot/Autopilot file

this is used only for convenience, rather than having to fully qualify supplier names in the Pilot/Autopilot file

the degree to which you define defaults is optional; commonly the `layer`, `variation`, and `platform` defaults are defined

also, it is very common to define the default `variation` as "default".

```
defaults: data, default, x64
```

asset

`(name::STRING) -> document::SITE`

locally define the name of a document which will be produced in a project; this is useful only at the very top of the contracting chain, that is, in the Pilot file.

```
asset("hello-world") -> project_construction_site

// This is most commmonly used directly as an input parameter to a "core" supplier, like so:
sub new/program(asset("hello-world")) -> {
  // program details go here...
}
```

Communications

job

`job /<layer>/<verb>/<subject>/<variation>/<platform>  ( <requirement parameters> ) <obligation parameters> :`

Defines the design of, and the parameter labels for an Agent.

Every part of the design classification (i.e. `layer`, `verb`, `subject`, `variation`, `platform`) **must** be defined explicitly.

The requirement and obligation parameters are entirely dependent on the contribution definition in the Valley

```
job /data/new/integer/default/x64(the_code_position, minimum_value, maximum_value) the_integer_variable :
    // work goes here
end
```

sub

`sub /<layer>/<verb>/<subject>/<variation>/<platform>  ( <requirement parameters> ) -> <obligation parameters>`

Subcontract a supplier

Every part of the design classification (i.e. `layer`, `verb`, `subject`, `variation`, `platform`) **must** be defined explicitly, or implicitly from a 'defaults' statement.

the `requirement` and `obligation` parameters are entirely dependent on the contract specification definition in the Valley

```
//                                     contributor            hosted parameters
//                                          v                        v
//                                       ,---.                  ,------------.
  sub /data/read/integer/default/linux-x64@julie (read, 0, 100) -> int, available
//      `-----------------------------------'        `------------'
//                    ^                                    ^
//               classification                  requested parameters

// And here's an example with default values in use
  sub new/integer($, -10, 42) -> my_int
```

join

`join /<layer>/<subject>/<variation>/<platform> ( <requirement parameters> ) -> <obligation parameters>`

Participate as a `join` in a Collaboration group

Every part of the identifier (i.e. `layer`, `subject`, `variation`, `platform`) **must** be defined explicitly, or implicitly from a 'defaults' statement.

The requirement and obligation parameters are entirely dependent on the collaboration definition in the Valley

```
join integer(the_integer_variable) -> minimum_value, maximum_value, memory_handle
```

host

`host  /<layer>/<subject>/<variation>/<platform> ( <requirement parameters> ) -> <obligation parameters>`

participate as the host in a Collaboration group

Every part of the identifier (i.e. `layer`, `subject`, `variation`, `platform`) **must** be defined explicitly, or implicitly from a 'defaults' statement.

The requirement and obligation parameters are entirely dependent on the collaboration definition in the Valley

```
host integer(0, 100, memory_handle) -> the_integer_variable
```

deliver

`(document::SITE, input::STRING)`

return document content for a project; useful only to /byte layer Agents

```
deliver(code_site, "\x49\b8" + pack("int64le", var_address))    //=> there is no return value (purely used for the communication side-effect -- returning bytes)
```

def

The `def` statement is used to define a block of statements (a.k.a a macro) which can shorten expressions and avoid repetition within expressions. The `def` statement defines a block of contracting which terminates with an `end` statement.

Macro definition example

```
def doubleInt(flow_in, integer_in) integer_out:
  sub add/integer/with-constant-to-new-result(flow_in, integer_in, integer_in) -> integer_out
end
```

Macro instantiation example

```
flow -> {
  doubleInt($, i1) -> i2
  doubleInt($, i2) -> i3
}
```

Operations

Agents can perform runtime operations with the immediate values. These operations are familiar to those found in most other languages.

Operator precedence also follows familiar rules:

order	operator/s
1	`-` (sign), `+` (sign), `!`
2	`%`
3	`*`, `/`
4	`-`, `+`
5	`>`, `<`, `>=`, `<=`
6	`==`, `!=`

7	`&&`
8	`||`

The lower order operations are applied first. As usual, the ordering can be affected (or made explicit) by the use of parentheses.

!

`(input::BOOLEAN) -> output::BOOLEAN`

invert boolean value (i.e. logical "NOT")

```
!true   //=> false
!false  //=> true
```

%

`(dividend::INTEGER, divisor::INTEGER) -> remainder::INTEGER`

integer division remainder of two integers; the remainder is an integer, with the quotient discarded

```
20 % 5        //=> 0
24 % 5        //=> 4
-21 % 5       //=> -1
```

*****

`(op1::INTEGER, op2::INTEGER) -> result::INTEGER`

`(op1::STRING,  op2::STRING)  -> result::STRING`

product of two integers, or the repetition of a string

```
2 * 21        //=> 42
"whoop " * 3  //=> "whoop whoop whoop "
```

/

`(dividend::INTEGER, divisor::INTEGER) -> quotient::INTEGER`

integer division quotient of two integers; the quotient is an integer, with the remainder discarded

```
20 / 5        //=> 4
24 / 5        //=> 4
-21 / 5       //=> -4
```

-

`(op1::INTEGER, op2::INTEGER) -> result::INTEGER`

difference of two integers

```
99 - 12       //=> 87
```

+

`(op1::INTEGER, op2::INTEGER) -> result::INTEGER`

`(op1::STRING,  op2::STRING)  -> result::STRING`

sum of two integers, or the concatenation of two strings

```
13 + 29                     //=> 42
"hello, " + "world complier!" //=> "hello, world compiler!"
```

>, <. >=, <=

`(op1::INTEGER, op2::INTEGER) -> result::BOOLEAN`

integer comparisons

```
3 < 42    //=> true
6 >= 20   //=> false
```

==, !=

`(op1::INTEGER, op2::INTEGER) -> result::BOOLEAN`

`(op1::STRING,  op2::STRING)  -> result::BOOLEAN`

`(op1::BOOLEAN, op2::BOOLEAN) -> result::BOOLEAN`

integer, string, or boolean comparisons

```
2 == 7               //=> false
"hello" != "world"   //=> true
true == false        //=> false
```

&&

`(op1::BOOLEAN, op2::BOOLEAN) -> result::BOOLEAN`

true only if both inputs are true (i.e. logical "AND")

```
true && true    //=> true
true && false   //=> false
```

||

`(op1::BOOLEAN, op2::BOOLEAN) -> result::BOOLEAN`

true if either input is true (i.e. logical "OR")

```
true || false   //=> true
false || false  //=> false
```

Functions

A particular set of operations are what we categories as "functions". They are performed with immediate values, in a format similar to traditional function calls.

max

`(op1::INTEGER, op2::INTEGER, ...) -> result::INTEGER`

largest value in a list of integers (all positives are larger than negatives)

```
max(3, -42, 7)    //=> 7
```

min

`(op1::INTEGER, op2::INTEGER, ...) -> result::INTEGER`

smallest value in a list of integers (all negatives are smaller than positives)

```
min(3, -42, 7)    //=> -42
```

concat

`(op1::STRING, op2::STRING, ...) -> result::STRING`

concatenation of one or more strings (functionally equivalent to the "+" function)

```
concat("hello, ", "world", " complier", "!") //=> "hello, world compiler!"
```

len

`(input::STRING) -> output::INTEGER`

length of a string value

```
len("aloha")  //=> 5
```

maxlen

`(input::STRING) -> output::INTEGER`

maximum length of a string; this is known before the actual string value is known and is useful for space-reservation protocols, as well as bounds checks

```
maxlen(pack("int64le", my_address))   //=> 8 (this result is known before ~my_address~ is even known)
```

trunc

`(input::STRING, max_length::INTEGER) -> output::STRING`

truncate a string to a be no longer than a given maximum length; if the input string is already no longer, the output is same as input

```
trunc("stop waiting", 4)    //=> "stop"
trunc("namaste", 100)       //=> "namaste"
```

replace

`(input::STRING, old_substring::STRING old, new_substring::STRING) -> output::STRING`

replace each occurence of a substring with a different string; strings are treated literally, that is, "Hey" does not equal "hey".

```
replace("Good going", "go", "tim")                 //=> "Good timing"
replace("I can not do that, not now", "not", "")    //=> "I can do that, now"
```

int2str

`(input::INTEGER) -> output::STRING`

decimal string representation of an integer

```
int2str(42)       //=> "42"
```

pack

`(format::STRING, input::INTEGER) -> output::STRING`

binary represenation of an integer value; useful for /byte layer Agents.

valid `format` values are limited to:

- `"int64le"` -- represent as a 64-bit value (8 bytes), little-endian, signed
- `"int32le"` -- represent as a 32-bit value (4 bytes), little-endian, signed
- `"int8"` -- represent as an 8-bit value (1 byte), signed

```
pack("int64le", 400080)   //=> "\xd0\x1a\x06\x00\x00\x00\x00\x00"
pack("int8", 10)          //=> "\n"
pack("int8", -10)         //=> "\xf6"
```

pad

`(pad_char::STRING, input::STRING, quantum::INTEGER) -> output::STRING`

pad a string so that its length is a multiple of the quantum, using a specified character to pad the end of the string

note that the current implementation limits the specified `pad_char` that it must be exactly one byte

```
pad("-", "hello", 7)      //=> "hello--"
pad("-", "hello", 5)      //=> "hello"
pad("-", "hello", 4)      //=> "hello---"
pad("-", "", 4)           //=> "----"
```

escape

`(input::STRING) -> output::STRING`

escape a string according to interpretation rules of a string in Autopilot/Pilot files; mostly just useful for certain /behaviour layer Agents

```
escape("This \" has \n some \xff garbage")    //=> "This \\\" has \\n some \\xff garbage"
```

Conditionals

Flow control has three patterns of use, to be selected dependent on your nature of work.

```
// the most basic form of conditional; useful when you want to conditionally "tack-on" work:
if condition::BOOLEAN then
  // <statements>
end

// slightly more advanced form of conditional; useful when you want to always "tack-on" work, but the particular work can change based on parameters
// functionally equivalent to two "if" statements with inverted conditions
if condition::BOOLEAN then
  // <statements>
else
  // <statements>
end

// the most broadly applicable form of conditional; useful when you want to "weave-in" work which can change based on parameters
if condition::BOOLEAN then
  // <statements>
  // expression::<output_type>
else
  // <statements>
  // expression::<output_type>
end -> result::<output_type>
```

Note that the current implementation does not support `elsif`; instead you should use `if`s nested inside `else`s.

Labels defined within the statements sections can **not** be used outside of the sections. To do so implies the work is not being "tacked-on" but rather "weaved-in" to the design. You should note that the "weaved-in" form of conditional does have output labels. The values associated with these output labels are defined by the last line within the `then` and `else` sections.

Conditional Examples

tack-on form:

```
if max_length < 10 then
  42 -> x
  sub set/integer($, my_int, x)
end

if !(max_length < 10) then
  sub set/integer($, my_int, 10)
end

// It's illegal to use ~x~ out here
```

tack-on form, with else:

```
if max_length < 10 then
  42 -> x
  sub set/integer($, my_int, x)
else
  sub set/integer($, my_int, 10)
end

// It's still illegal to use ~x~ out here
```

weave-in form:

```
if max_length < 10 then
  sub new/integer($, 0, 50, 0) -> i
  42 -> v
  sub set/integer($, i, v)
  [v, i]
else
  sub new/integer($, 0, 20, 0) -> i
  sub set/integer($, i, 10)
  [10, i]
end -> x, my_int

// We can use ~x~ and ~my_int~ out here because

```

```
14 // we defined the ~if~ statement such that it evaluates to values.
```

This is but a tiny taste of the possibilties created by conditional statements. More examples will follow in future.