The Logix Tutorial

For Logix 0.4

Tom Locke. <tom@livelogix.com>

The Logix Tutorial – Table of Contents

1 Introduction

1.1 Welcome

Welcome to the Logix Tutorial! You've arrived a little early – the current alpha version of Logix is still a little rough around the edges. You should interpret what you discover here as a flavor of what is to come, rather than the definitive guide to Logix.

One implication of this is that Logix is going to be changing in incompatible ways. If you start writing Logix code, you can expect to be making changes before it will work with future Logix releases. For an idea of aspects of Logix that are likely to be changing, have a look at http://logix.livelogix.com/future-work.html.

Have fun!

1.2 Status and Warnings

Logix is currently in use on a daily basis. *It is nevertheless definitely alpha quality*.

1.2.1 Error Messages

By far the biggest issue for users will be error reporting. While some error reporting is already quite helpful, some is lacking.

You are likely at some point to find an error in your code reported not with a nice tidy message, but with a stack-trace into the parser. You then have two options.

- Stare hard at your code until you spot your error :-)
- Try to figure out what happened based on the stack-trace

The second option is clearly going to require some understanding of the internals of the parser. Plus there's always the possibility that your code is correct and you've discovered a bug in the parser :-)

1.3 Examples in this Tutorial

The examples in this tutorial are intended to be tried at the Logix prompt. You are strongly encouraged to go through the tutorial at the computer, and try the examples as they come up.

One thing to bear in mind – the layout in this tutorial doesn't allow for very long lines in the examples. It will generally be best to enter the examples on a single line, even if they span multiple lines in the tutorial. If, at the Logix prompt, you enter a line that ends in a colon, the prompt will allow you to enter continuation lines until you enter a blank line.

2 Installation and starting the Logix shell

2.1 Installing

Logix installs as a single package, `logix`, in your site packages directory. Once you have downloaded the source distribution from http://logix.livelogix.com, unpack it to create the directory

```
Logix-0.4
```

To install, execute the following command in that directory

```
> python setup.py install
```

That's it!

2.2 The Logix Shell

Logix runs on top of Python. The Logix interactive top-level can be used with or without the enhanced python shell *IPython* (http://ipython.scipy.org/).

2.2.1 Standard Python Shell

To get to the interactive Logix prompt simply call `logix.interact()`

```
> python
>>> import logix
>>> logix.interact()
(type logix.license for more information)
Welcome to Logix
[std]:
```

Or

```
> python –c "import logix; logix.interact()"
(type logix.license for more information)
Welcome to Logix
[std]:
```

2.2.2 IPython Shell

The module `logix.ipy` provides some support for IPython. With IPython and Logix, you can flip between Python and Logix modes on the fly. The namespace (local variables) remains constant, so for example, you can define a function in Logix, and call it from Python.

The function `logix.ipy.toggleLogixMode` can be called to switch between Logix mode and regular Python mode. It should be passed the IPython application object `__IP`.

```
In [1]: import logix.ipy
In [2]: logix.ipy.toggleLogixMode(__IP)
(type logix.license for more information)
Welcome to Logix

[std]:
```

It is a good idea to set up an IPython 'magic command' for this, such as '`lx`'. (see http://ipython.scipy.org/doc/manual).

```
def magic_lx(self,parameter_s=''):
    import logix.ipy
    logix.ipy.toggleLogixMode(__IP)
```

You can then just enter `lx` to flip between Logix and Python modes.

3 Introduction For Python Folks

Logix is not one language but many. When you start Logix, the 'current language' will be S*tandard Logix* (`std`). Python folks might want to switch to a more Python-like syntax:

```
[std]: setlang logix.baselang
```

This puts you into *Base Logix* (`base`), a Python-like language which Logix can translate into Python byte code. Other Logix languages are generally implemented by translation into Base Logix.

You can now bash away as if you were at the regular Python prompt. There are some differences with Python however. There are some things that work in Base Logix that are not valid Python, and there are some nuances of Python syntax that are not supported.

3.1 Expression Based Syntax

The main difference with Python is that Logix has an expression-based syntax, which is to say there is no statement/expression distinction. For example, try

```
[base]: f = lambda x: (print x; x+1)
[base]: f(5)
5
6
```

As in Python, the `lambda` expression returns a function. Unlike Python, `print` also yields an expression (with a side-effect). The `;` is an operator that takes two expressions and evaluates them in left-to-right order, returning the result of the second.

The Logix `if` returns a value:

```
[base]: x = if 2+2==4: "I thought so" else: "hmmm..."
[base]: x
 'I thought so'
```

From this example you can also see that Logix's white-space rules differ from Python's. In fact Logix's white-space rules are more like those of Haskell (www.haskell.org). In Logix:

- Any number of single-line blocks can occupy the same line
- An indented line where no block is expected, is a continuation line (Logix does not recognize '\' as a line continuation character)
- There is no `pass` statement. An empty block is an empty block.

Base Logix also has a `do` operator that allows you to introduce a block of code anywhere you can have an expression. The result of the `do` is the result of the last expression in the block.

```
[base]: vals = [3,6,8,2,4,7]
[base]: average = do:
            sum = 0
            for x in vals: sum += x
            float(x) / len(vals)
[base]: average
1.166666666666667
```

3.2 Unsupported Python Syntax

Base Logix does not currently support the following from Python:

- Concatenation of `"adjacent" "string" "literals"`
- Back-quote (`` `x` `` for `repr(x)`)
- Tuple patterns in argument lists (e.g. `def f(a, (b, c)): ...`)
- The continuation line character '\' (indented lines are continuation lines)

4 Introduction For Lisp Folks

First off, Lispers will be happy to learn that Logix has both code quoting and a procedural macro system. Logix programs are not made from lists, they are made from user-defined object types. More on this in *7 - Languages: Extending and Creating*.

4.1 Function Calls

The Logix prompt can recognize multiple languages. It initially expects *Standard Logix* (`std`). Simple function calls in Standard Logix look a lot like Lisp.

```
[std]: max 4 (min 5 6)
5
```

Note parentheses are not required on the outer function. In fact, parentheses do not have the special status they do in Lisp – they are just a scoping operator as in most programming languages. A function-call expression is simply a sequence of expressions:

$$func\ arg_1\ arg_2 \ldots arg_n$$

Logix handles first-class functions as does Python – i.e. it is Scheme-like not Lisp-like. The astute reader may be wondering how to call a zero argument function. E.g.

```
[std]: from random import random
[std]: print random
<built-in method random of Random object at 0x009B0368>
```

The function was not called. To call it, use a special postfix operator `()`.

```
[std]: print random()
0.746969538583
```

Note that `()` is a special operator. It is *not* empty parentheses. (in case you're wondering, the empty list in Standard Logix is `[]`)

4.2 Other Operators

As well as the function call syntax, Standard Logix has many operators with a rich syntax.

```
[std]: x = 1 + 2
[std]: print if x == 3: 'good' else: 'oh dear'
'good'
[std]: aTuple = 1, 2, 3, 4
[std]: [x * 2 for x in aTuple]
[2, 4, 6, 8]
```

4.3 White-space Sensitive Syntax

Blocks are delimited by indentation:

```
[std]: for x in range 10:
     :    stars = "*" * x
     :    print stars
```

There are a couple of other tricks done with white-space. More later.

4.4 Lexical Rules

Identifiers in Logix follow C like lexical rules. However, any character can be used when extending the syntax, i.e. when defining new operators (see *7 - Languages: Extending and Creating*).

4.5 Python Semantics

Logix is implemented as a font-end to Python, so a good deal of Python semantics are exposed. Lisp folks need to watch out for a few things. One way to go would be to read up on Python (docs.python.org), but for the impatient, here's a few highlights:

- Variables spring into existence when first assigned to
- There are no block-local variables. All variables created (i.e. assigned to) inside a function share the same function-local namespace.
- Variables assigned to outside any function are global.
- To assign to a global variable inside a function, first use the `global` statement

```
def f(x):
    global g
    g = x
```

- Lexical closures are supported, but captured variables are read-only inside the closure. (note: it seems this restriction may only apply to the Python compiler and not to the Python VM, we are investigating removing this restriction for Logix)

4.6 Symbol Operator

In Logix, # is the comment character. To write a literal symbol in Standard Logix, use **~**

```
[std]: ~foo
~foo
```

Note this operator has another trick up its sleeve:

```
[std]: ~("foo".upper() + "!")
~FOO!
```

4.7 Quasiquoting

Logix supports the back-quote operator just like Lisp:

```
[std]: `max 4 5
<std: max 4 5>
```

Unlike Lisp, the result is not a list, but an *operator* object. In this example, the operator represents a function call.

We can also quote operators:

```
[std]: `1 + 2
<std:+ 1 2>
```

The returned value is an instance of the + operator from the language `std`, with operands `1 2`, it reads as a prefix notation of the expression.

Inside the quote, the backslash can be used like the comma in Lisp:

```
[std]: var = ~x
[std]: ` \var += 1
<base:+= x 1>
```

(note that the += operator is inherited from another language: Base Logix)

Multiple backslashes can be used to escape nested quotes just as with multiple commas in Lisp.

The `\*` operator is the equivalent of Lisp's `,@` i.e. it *splices* a list into the quoted value.

```
[std]: l = [3, 45, 21]
[std]: `max \*l
<std: max 3 45 21>
```

You can evaluate these quoted expressions using `logix.eval`

```
[std]: logix.eval `max \*l
45
```

4.8 Macros

Defining macros is performed hand-in-hand with defining new operators with custom syntax. This is covered in more depth in *7 - Languages: Extending and Creating*. By way of a quick example:

```
[std]: defop 50 expr "++" macro ex: `\ex += 1
```

Here we have defined a new postfix operator with a binding value of 50, implemented as a macro. The macro function takes a single argument (`ex`) and returns the expanded code template.

```
[std]: x = 1
[std]: x++
[std]: x
2
```

You can experiment and debug using `logix.macroexpand` and `logix.macroexpand1`.

```
[std]: logix.macroexpand `a++
<base:+= a 1>
[std]: logix.macroexpand `player.score++
<base:+= (base:. player score) 1>
```

5 Standard Logix

Standard Logix is a Logix-defined language like any other. It is the language that the shell starts up in by default. This section describes Standard Logix and its various operators.

Standard Logix is experimental! It is likely to go through many upheavals before becoming a stable language. One of the great strengths of a meta-language like Logix is that it supports this kind of iterative, organic language design.

Standard Logix has a fairly transparent translation into Base Logix, and hence into Python. It feels very much like programming Python with a different syntax. If you don't know Python, you are strongly recommended to learn some before programming Standard Logix.

The implementation of Standard Logix can be found in `logix/std.lx` (at the time of writing, just 324 lines of code)

5.1 Python Basis

Standard Logix inherits all of the operators from Base Logix (Logix's Python-like language). In other words, pretty much everything you know from Python is available just the same in Standard Logix, unless stated otherwise in this section (e.g. the function-call syntax is different).

5.2 Function Calls

Function calls are simply a sequence of expressions:

```
[std]: max 3 6 4
6
```

Nested function calls need to be parenthesized.

```
[std]: max 4 5 (min 10 11)
10
```

There's nothing special going on with these parentheses – they are simply grouping the call to `min`, just as you would use parentheses in any expression. Parentheses are not part of the function call syntax as they are in Lisp, Python and many other languages.

The function itself can be the result of an expression:

```
[std]: myFile.write "hello"
```

Zero argument functions are called with a special postfix operator `()`

```
[std]: from random import random
[std]: print random
<built-in method random of Random object at 0x009B0368>
[std]: print random()
0.746969538583
```

This is not empty parentheses, it is a special operator.

5.3 Keyword Arguments

Keyword arguments are supported, with semantics equivalent to Python

```
[std]: result = db.search 'foo' caseSensitive=False
```

5.4 Lists

List syntax is like Python's.

```
[std]: [1,2,3]
[1, 2, 3]
```

List comprehensions are available.

```
[std]: [x**2 for x in range 5 if x != 2]
[0, 1, 9, 16]
```

A compact syntax is available for simple numeric ranges (it returns a Python `xrange`)

```
[std]: for x in [1..10]: print x
1
2
...
9
10
```

List subscripts cannot use the same syntax as Python – it would conflict with the function-call syntax, e.g.:

```
myList[3]
```

is in fact a call to the function `myList` with one argument, the list `[3]`. List subscripts instead look like this:

```
[std]: x = [1..10]
[std]: x/[3]
4
```

There must not be a space between the `/` and the `[`. Slices are similar:

```
[std]: "Hi there"/[3:]
'There'
[std]: "Hi there"/[::-1]
'erehT iH'
```

Logix introduces a data-type called an `flist` – a 'field list'. It is a list that also contains named elements. It is somewhat like a combination of a list and a dict. In Standard Logix, if a list literal contains named fields, an `flist` is returned instead of a `list`.

```
[std]: [1, 2, a=3, b=4]
[1, 2, a=3, b=4]
```

The list comprehension is similarly extended to support flists.

```
[std]: names = 'a', 'b'
[std]: values = 1, 2
[std]: [n=v for n, v in zip names values]
[a=1, b=2]
```

This always produces an flist with no elements. A list and an flist can be added together (+ operator) to overcome this.

5.5 Dictionaries

Syntax for dictionaries differs from Python (although, confusingly, dicts still display using Python syntax – hey, it's alpha!)

```
[std]: ['a':1, 'b':2]
{'a':1, 'b':2}
```

The empty dict is written:

```
[std]: [:]
{}
```

Logix also has dict comprehensions – just add a colon to the regular list comprehension syntax:

```
[std]: [ord x: x for x in "hello"]
{104: 'h', 108: 'l', 101: 'e', 111: 'o'}
```

As with Python, setting and getting dict entries uses the same syntax as list indexing.

```
[std]: d = [ord x: x for x in "hello"]
[std]: d/[101]
'e'
```

For the common case where the keys of a dict are names, a convenience operator is provided.

```
[std]: d = dict a=1 b=2
[std]: d/a
1
```

The divide operator is renamed to `div` (it is still infix).

5.6 Object Attributes

The traditional '`.`' operator is available, but with some additional smarts. Firstly it is a *smartspace* operator. Smartspace is a Logix feature that allows white-space around operators to effect the scope of the operator. The motivation for using smartspace is that without it, OO expressions in Standard Logix can be awkward. Consider for example the following Python expression:

```
# A Python expression
db.find(department).getStaff(name).emailAddr
```

In Standard Logix, you might write that

```
((db.find department).getStaff name).emailAddr
```

Which is much less readable than the Python version. Would it look better without the parentheses?

```
# Incorrect Standard Logix
db.find department.getStaff name.emailAddr
```

That looks all wrong – it looks like you are trying to do the following.

```
db.find (department.getStaff) (name.emailAddr)
```

In fact, that *is* how it is parsed (we'll see the correct way to write it momentarily).

By using a smartspace operator, OO statements are much easier to read. When there is no space around the '`.`', parentheses are implied. For example, these two are equivalent:

```
text.setFont userprefs.standardFont

text.setFont (userprefs.standardFont)
```

Whereas the previous example must be written:

```
db.find department .getStaff name .emailAddr
```

The space before the '.' causes the expression to be parsed the same as the original Python. This style may be unfamiliar, but it quickly becomes very easy to both write and read.

The '.' operator has some extra features. The equivalent of `getattr` can be achieved like this

```
[std]: # equivalent to: getattr obj x
[std]: obj.(x)
[std]: # equivalent to: getattr obj x None
[std]: obj.(x, None)
```

It is relatively common to need to test on an object field, where the object reference may be `None`, like this:

```
[std]: if account and account.active: ...
```

Standard Logix provides a convenience operator `.?` for this situation. The following is entirely equivalent:

```
[std]: if account.?active: ...
```

5.7 Sequence Operators

Some of the basic operators have variations for convenient operations on sequences.

`seq.*f`	is equivalent to	`[x.f for for x in seq]`
`seq/*[key]`	is equivalent to	`[x/[key] for x in seq]`
`seq/*foo`	is equivalent to	`[x/foo for x in seq]`
`a ,* b ,* c`	is equivalent to	`zip a b c`

5.8 Defining Functions

Function definitions look like Python without the parentheses or commas:

```
[std]: def lifeChangingFunction a b: return a+b
```

The `return` is in fact not required. Without it, the result of the function is the result of the last statement in the function body.

```
[std]: def lifeChangingFunction a b: a+b
[std]: def myMax a b: if a > b: a else: b
```

5.8.1 Argument Preconditions

Standard Logix allows argument preconditions to be defined inside the argument list. A precondition can either be

- a type, in which case an `isinstace` check is performed, or
- A predicate function that tests if the value is valid

```
[std]: def reverse s(str): s/[::-1]
[std]: reverse 'foo'
'oof'
[std]: reverse 12
ArgPredicateError Traceback (most recent call last)
d:\desktop\livelogix\python\<console> in <interactive>()
d:\desktop\livelogix\python\<console> in reverse(s)
ArgPredicateError: reverse: arg-predicate failed for 's' (got
12)
```

Using Logix's multiple language capabilities, the argument precondition is actually an expression in a special purpose language. This language is currently not very developed – it has only one operator `.` The `?` operator is used to indicate that `None` is a valid argument. E.g. this function must be passed an `Employee`:

```
[std]: def promote employee(Employee): ...
```

While this one can be passed an `Employee` or `None`

```
[std]: def promote employee(Employee?): ...
```

(note that in many languages, including Java, the null value matches any type. In Python and hence in Logix, `None` is an instance of `NoneType` and matches no other type)

In the future there will be other operators, e.g. for conjoining and disjoining argument predicates.

5.9 Lightweight Lambdas

There is a nice lightweight syntax for anonymous functions:

```
{ <arguments> | <expression> }
```

For example

```
[std]: map {x|x**2} (range 5)
[0, 1, 4, 9, 16]
```

In the common case where there is only a single argument, the argument spec can be omitted. The single argument is called `it`.

```
[std]: map {it**2} (range 5)
[0, 1, 4, 9, 16]
```

(The same syntax is used in the Groovy scripting language for the Java VM - http://groovy.codehaus.org/)

A lambda function with multiple statements can be created by combining the statements with either the `;` operator or the `do` operator.

The lambda operator supports argument preconditions.

5.10 Variable Function Arguments

Just as in Python, functions can receive variable arguments and variable keywords:

```
[std]: def f *args **kws: print args, kws
```

The calling syntax has to be slightly different to Python's, to avoid ambiguity with the * and ** operators.

```
[std]: f 1 2 *:myList **:myDict
```

5.11 Testing Lists: `forany` and `forall`

The `forall` operator evaluates true if the test holds for all elements of a list:

```
if bark > bite forall bark, bite in dogs:
    print "You don't scare me!"
```

Notice that as with list comprehensions, list elements can be extracted into any valid assignable place (here we unpacked a pair into two variables).

The `forany` operator has deeply mysterious semantics that I doubt anyone will be able to fathom:

```
if weight > link.maxWeight forany link in chain:
    chain.break()
```

`forall` will stop iterating as soon as a test evaluates false. `forany` will stop iterating as soon as any test evaluates true.

5.12 Building Lists: `listfor` and `listwhile`

List comprehensions are great for succinctly building lists within expressions, but sometimes they don't suffice, particularly if a temporary value is needed.

The `listfor` and `listwhile` operators are just like `for` and `while`, except they have a result: a list made from the result of the loop body for each iteration (the result of the loop body is the result of the last expression). The operators also have an `if` clause similar to that found in list comprehensions.

Say you wanted to do:

```
[std]: x = [line.split()[0] + "," + line.split()[1]
            for line in myfile]
```

With list comprehensions, you can't avoid calling `split` twice. With `listfor` it looks like

```
[std]: x = listfor line in myfile:
     :         fields = line.split()
     :         fields[0] + "," + fields[1]
```

5.13 Scanning Lists: `valfor` and `breakwith`

It is common to use a `for` loop to search a list for a particular value, `break`-ing out of the list when the value is found. Python provides a very useful `else` clause that can be used to provide a default when the value is not found:

```
# Name of first employee in sales department
for e in employees:
    if e.department == 'sales':
        result = e.name
else:
    result = None
```

This however, is typical 'inside out' imperative programming – it is not clear that the whole construct is simply assigning a value to `result`.

For this situation, Logix provides `valfor`, to be used in conjunction with `breakwith`.

```
result = valfor e in employees:
             if e.department == 'sales'
                 breakwith e.name
         else:
             None
```

The construct is useful, but the keywords `valfor` and `breakwith` are rather clumsy – they are liable to change.

5.14 Function Pipelines

Python programmers have generally found list comprehensions to provide a nicely readable alternative to the `map` function, especially when multiple steps are required. For example, without list comprehensions, an expression like

```
[len(str(x)) for x in l]
```

requires something like

```
map len (map str l)
```

or

```
map {len (str it)} l
```

Neither are very pleasing to read. Logix provides another alternative which is both clear and concise: an infix map operator. With this operator, expressions read like a 'pipeline'.

```
l *> str *> len
```

In other words "pipe `l` through `str`, and then `len`".

In a similar vain, `?>` filters a list according to a predicate function, and `.>` simply applies a function to a single value. Here is a pipe that uses all three.

```
names ?> {it.endswith '.dat'} *> str.lower .> ", ".join
```

Such expressions have the advantage that the order of operations reads left to right. The equivalent list comprehension reads backwards:

```
[", ".join (str.lower f)
 for f in names if f.endswith ".dat"]
```

5.15 Timing code

A minor but handy feature is provided for timing code. The `do` operator can be given a `timed` clause, with a message.

```
[std]: do timed "big loop": for x in range 10**7:
big loop block time: 2.328000
```

5.16 String Literals

Standard Logix includes a great idea for indentation-friendly multi-line string literals by Greg Ewing (http://nz.cosc.canterbury.ac.nz/~greg).

```
longString = """|Eveything after the | is included.
               |Nothing needs to be escaped.
               |Leave an empty line at the end
               |to terminate with a newline (like this...)
               |
               "
```

The double-quote at the end is optional – it keeps emacs syntax highlighting on track!

6 Logix Modules

Logix has a module system similar to Python's. Logix modules have the suffix "`.lx`" and are compiled to "`.lxc`" files. The `limport` statement will import Logix modules found on the standard `PYTHONPATH`. `limport` is not yet fully cooked: `limport as` is supported but there is no equivalent of Python's `from`. Also, importing from packages has a quirky result:

```
[std]: limport mypackage.mymodule
[std]: mypackage
NameError: name 'mypackage' is not defined
[std]: mymodule
<module 'mypackage.mymodule' from 'mypackage/mymodule.lxc'>
```

In other words, `limport a.b` currently behaves more like `from a limport b`. This will change!

6.1 Packages

With the current implementation, Logix cannot be used to implement package files. If you wish to create a package, you must create a Python `__init__.py` file as usual. Once created in this manner, packages may contain Logix modules (`.lx` files).

7 Languages: Extending and Creating

Now we get to the good stuff! Logix as a meta-language – a language for creating languages.

7.1 Operator Based Syntax

In Logix, a language is essentially just a collection of operators. Logix languages do not have a master grammar. Instead, each operator has its own 'mini-grammar'. Each operator also has a defined binding-value and associativity that determine how operators are combined into expressions. This makes it very easy to add operators to an existing language, without worrying about the nuances of an existing grammar.

Having languages defined by operators rather than by a grammar, is about more than just convenience, it also captures a language design philosophy. This philosophy can be summed up as "first-class, run-time representations for everything". Python is a great example of a language that adheres to this principle. Java is not.

To illustrate this approach, take method modifiers as an example. In Java, specifying method modifiers is achieved via the grammar – you type the modifiers at the start of the method signature. Suppose however, you don't want to commit to the modifiers statically. Maybe you want to pull them from a configuration database. You can probably pull this off in Java using the reflection API, but it won't be pretty. Java discourages dynamic programming.

The Logix philosophy says if you want method modifiers, have a run-time representation, and a run-time mechanism for applying modifiers to a method. For example, modifiers could simply be a tuple of symbols (see *4.6 - Symbol Operator*). The `def` operator could be modified to take a modifier operand on its left-hand-side, e.g.:

```
~static, ~synchronized def foo self: ...
```

Now suppose we have looked up our dynamic modifiers in the database, and have them in a variable `mods` (a tuple of symbols), we can define our method simply:

```
mods def foo self: ...
```

Building structures according to a grammar forces you to commit to those structures in your text editor. Building structures from operators allows for run-time dynamism.

Despite the restriction that Logix programs are nothing but large expressions built from operators, Logix tries to be as syntactically flexible as possible. For this reason, the 'mini-grammar' in a Logix operator can be not-so-mini. Logix encourages but does not enforce the "first-class, run-time representations for

everything" principle. For example, Standard and Base Logix support list comprehensions, which have quite a complex syntactic structure. The components of a list comprehension (e.g. the `for` and `if` clauses) are not assembled by operators at run-time (that would probably be a dynamism too far!)

(Aside: because manipulating code programmatically is easy in Logix, you could in fact assemble a list comprehension, or anything else, at runtime – if you don't mind the overhead of run-time parsing and code generation)

7.1.1 Limitations

There may often be a need to emulate existing languages in Logix. For example, Base Logix is an emulation of Python. Because the emulated language will probably be grammar based, this can be tricky, and require some creative use of Logix's capabilities. For example, Python programmers probably think of the `not in` operator as a variant of the `in` operator. In Base Logix however, `not in` is implemented as a variant of the `not` operator – i.e. the `not` operator has an optional `in` extension. The emulation may be imperfect as a result, for example we cannot give `not in` a different binding-value to `not`.

7.2 Defining Operators

The special operator `defop` is used to create a new operator. A `defop` at the top-level of a module creates a temporary operator – one that cannot be used outside of the module. By default, all languages have the `defop` operator (this behavior can be overridden).

The syntax of `defop` is:

```
defop ['l' | 'r' ]
      ['smartspace']
      <binding>
      <syntax>
      [ <implementation> ]
```

In other words, the definition specifies:

- The associativity of the operator – left or right
- Whether or not 'smart' white-space rules apply to the operator.
- The binding-value – how tightly the operator binds to its operands.
- The syntax for the operator. The syntax also defines the *operator-token*.
- The implementation (more later)

We will start with a very simple prefix operator: `>>` as a shorthand for print:

```
[std]: defop 0 ">>" expr func x: print x
```

This reads as:

- define an operator `>>`

- with binding value 0
- with syntax: "`>>`" followed by an expression (i.e. it is a prefix operator)
- implemented as a function (`func`) which prints its single argument (`x`)

```
[std]: >> 45
45
```

The operand – in this case just a literal `45` – was evaluated, and passed to the operator function, which printed it.

```
[std]: >> "hi there " * 3
hi there hi there hi there
```

Here we see that `*` binds more tightly than `>>` because we gave 0 as the binding-value. You can also observe this by quoting the expression

```
[std]: `>> "hi there" * 3
<std~:>> (std:* 'hi there' 3)>
```

We will see more on quoting later, for now we just need to know that it returns the code as a data structure – an abstract syntax tree (AST) if you like (the data is considerably more straightforward than you might expect from an AST, for example that `3` is not something like an `IntergerLiteralNode`, it's just a `3`, an `int`). These data structures display in a manner that can be read as a fully parenthesized prefix notation, so they are ideal for testing how an expression was parsed. (note the language for the `>>` operator has displayed as "`std~`" as opposed to just `std`. The `~` indicates this is a temporary, local language.

We could easily define `>>` as a postfix operator

```
[std]: defop 0 expr ">>" func x: print x
[std]: 101 >>
101
```

The syntax has changed from `">>" expr` to `expr ">>"`. Either way, the *operator-token* is `>>`, which means the second `defop` has overwritten the previous one. In general, how does Logix derive the operator-token from the syntax definition?

The syntax definition must begin with

- a literal-rule (e.g. `">>"`), or
- `expr` or `[expr]` immediately followed by a literal-rule

In either case, the operator-token is defined by the literal-rule.

7.2.1 Operators for New Languages

We have so far seen how to create temporary operators. To define permanent operators that reside in a language, we have to use `deflang` to create a new

language. We will have a proper look at `deflang` in *7.7 Multiple Languages*, but a quick sneak peak will be useful at this point.

```
deflang myLanguage:
    defop ...
    defop ...
```

This creates a new language `myLanguage`. Any `defop` inside the `deflang` adds an operator to the language.

7.3 Operator Syntax

Logix provides a custom language for defining syntax. We have already seen some features of this language:

- `"..."` means parse literal text – a 'terminal symbol' in grammar speak
- `expr` means parse an expression
- A sequence of terms, e.g. `">>" expr` means parse those things in sequence

If you are familiar with parser toolkits, you may be expecting to see some variant of BNF. Logix syntax definitions are similar to BNF, but what is that `expr` rule? Is `expr` a non-terminal symbol – a named rule defined elsewhere? No. Because Logix syntax is operator based, not grammar based, the meaning of `expr` is hard-wired. It means 'parse any valid expression in this language'. Every time you add an operator to the language, it can be used wherever any expression is expected.

We can do quite a lot with just `expr`, literal rules and sequences, e.g.

Prefix operator:	`"not" expr`
Postfix operator:	`expr "++"`
Infix operator:	`expr "+" expr`
Mixfix operator:	`expr "?" expr ":" expr`
Keyword:	`"break"`

7.3.1 Binding and Associativity

Before delving into the rest of the syntax language, lets take a look at binding and associativity. We will use quoting to see how Logix has combined the operators into an expression.

```
[std]: defop 50 expr && expr
[std]: defop 40 expr || expr
[std]: `a && b || c    # && binds more tightly
<std:|| (std:&& a b) c>
```

We can of course be explicit using parentheses, but look carefully at the code-data.

```
[std]: `a && (b || c)
<std:&& a (std:( (std:|| b c))>
```

The parentheses don't just reshape the code-data, they appear as part of the code-data. Parentheses in Logix are just another operator – they can perform computation if you like.

Now let's change the operator precedence.

```
[std]: defop 60 expr || expr    # || now binds more tightly
than &&
[std]: `a && b || c
<std~:&& a (std:|| b c)>
```

Now take a look at associativity

```
[std]: ` a && b && c
<std~:&& (std:&& a b) c>
```

The operator is left associative – this is the default. Let's change it.

```
[std]: defop r 50 expr && expr
[std]: ` a && b && c
<std~:&& a (std:&& b c)>
```

7.3.2 The Syntax Language

OK – time to take a wider look at the syntax language

Rule	Matches
$rule_1$ $rule_2$ ... $rule_n$	A sequence of terms in the given order
`"..."`	The given literal text
`expr`	Any expression in the language
`term`	A single term of an expression (see below)
`symbol`	A Python identifier (i.e. a name) (also matches a quote-escaped expression)
`token`	Any single token
`block`	A indentation delimited block of lines
`eol`	An end-of-line token
rule `+`	One or more occurrences of *rule*
rule `*`	Zero or more occurrences of *rule*
$rule_1$ `|` $rule_2$ `|` ... `|` $rule_n$	Choice between n alternative rules
`[` *rule* `]`	Optional rule

`freetext` */regex/*	
`freetext upto` */regex/*	See the later section on free-text
`optext` */regex/*	

We shall see these in action in some examples from Base and Standard Logix.
The `is` / `is not` operator from Python:

```
defop 35 expr "is" ["not"] expr
```

Python dict literals:

```
defop 0 "{" [ expr ":" expr ("," expr ":" expr)* ] "}"
```

Python's `import`:

```
defop 0 "import" symbol ("." symbol)* ["as" symbol]
```

Python's `print` statement:

```
defop 15 "print" ( ">>" expr [("," expr)+ [","]]
                 | [expr ("," expr)* [","]] )
```

Notice how similar this is to the Python's grammar production for `print`.
Compound statements like `if` and `for` are also just operators:

```
defop 0 "if" expr ":" block
            ( [eol] "elif" expr ":" block )*
            [ [eol] "else" ":" block ]

defop 0 "for" expr "in" expr ":"
              block
              [ [eol] "else" ":" block]
```

Note the use of an optional `eol` to allow these statements to continue over more than one line.

7.3.3 Expressions and Terms

[Note: term is probably the wrong, er, term for this concept – it needs changing]
As well as the `expr` rule, the syntax language provides a `term` rule, which was described above as describing a single `term` of an expression. More specifically, a term is:

- A numeric literal
- A name
- A prefix operator, followed by the right-hand-side of the operator

- A term, followed by an infix operator, followed by the right-hand-side of the operator.

Here, by prefix operator we mean any operator with no left-hand-side (the right-hand-side may be arbitrarily complex), and by infix operator, we mean any operator that does have a left-hand-side (again, the right-hand-side may be complex). For example, a list comprehension is considered a prefix operator for these purposes.

We can see then, for example, a function call in Standard Logix, such as

```
f x y z
```

is not a valid `term` – there are no infix operators to join the parts together.

The meaning of `expr` varies from language to language, and we shall come back to it in *7.7 Multiple Languages*.

In Base Logix an `expr` will match a function call or a subscript, or a sequence of both (`term` will not). In Standard Logix, an `expr` will match a function call.

7.3.4 Limitations and Issues

The syntax language is flexible enough to define syntax that will not parse properly. Logix is alpha enough to leave you pretty much on your own in this regard. In the future there will be some formal rules, and Logix will enforce them as far as possible.

For the time being, the most important rule concerns defining choices. The parser will attempt to recognize the rules in the order they appear in the syntax definition (left to right). If a match is found the parser moves on, so put more specific rules first. For example, notice how the order of the alternatives in the above syntax for `print` is reversed compared to the standard Python grammar production.

7.4 From Syntax to Code-Data

As in Lisp, Logix programs are just Logix data. Whereas Lisp programs made from lists of lists, Logix programs are made from a richer (but just as easily manipulated) data structure. We call this code-data.

- The Logix parser takes text as input and outputs *code-data*
- The Logix compiler takes *code-data* as input and outputs Python *byte-codes*.

Both the parser and compiler are fully available at run-time.

The structure of code-data is determined by the syntax of the operators. When you define an operator with `defop`, Logix creates a corresponding class. Occurrences of the operator in your source become instances of this class in the code-data.

In general, code-data is comprised of

- Operator instances

- Basic data types including primitive types, lists, tuples, dicts and `None`
- Symbols (instances of `logix.Symbol`)

The basic data types become the equivalent literal values in the compiled program. Symbols become variable names.

You can experiment with the parsing of source-code into code-data using the back-quote. For example, observe that numeric literals in the source simply become the equivalent run-time value in code-data:

```
[std]: `1
1
```

While names become symbols

```
[std]: `a
~a
```

You can experiment with the compilation process using `logix.eval` which compiles and evaluates the code-data you pass, and returns the result. For example, observe that basic literals evaluate to themselves:

```
[std]: logix.eval `1
1
[std]: logix.eval `a
NameError: name 'a' is not defined
[std]: a = 10
[std]: logix.eval `a
10
```

You can also see from this example that the `eval` method uses the existing environment.

Structured code-data is created when operators are parsed. For example

```
[std]: defop 50 expr '+++' expr func a b: a+b
[std]: exp = `a +++ b
[std]: exp
<std~:+++ a b>
[std] type exp
<operator std~ +++>
[std]: vars exp
{'__operands__': [a, b]}
[std]: exp/0
~a
[std]: exp/1
~b
[std]: a, b = 1, 2
[std]: logix.eval exp
3
```

The type `<operator std~ +++>` is in fact a dynamically created class (created when we executed the `defop`). From the source-code "`a +++ b`" the parser built an instance of this class, with the operands stored in an flist in the `__operands__` attribute. The operands are also available via the subscript operator, as seen above.

If the syntax defines an operand as optional, the value in the code-data will be `None` when that operand is omitted.

```
[std]: defop 50 [expr] '+++' [expr]
[std]: `1 +++
<std~:+++ 1 None>
[std]: `+++ 1
<std~:+++ None 1>
[std]: `+++
<std~:+++ None None>
```

Repeating structures generate multiple operands

```
[std]: defop 50 '+++' (expr ".")*
[std]: `+++ 1. 2. 3.
<std~:+++ 1 2 3>
[std]: `+++
<std~:+++>
```

Notice that the literal "`.`" does not appear in the code-data. The literal is assumed to be merely delimiting the syntax, and of no subsequent interest. An exception to this rule occurs when the literal is optional, in this case the programmer will want to know if the literal appeared or not:

```
[std]: defop 50 '+++' expr ['!']
[std]: `+++ 1
<std~:+++ 1 None>
[std]: `+++ 1 !
<std~:+++ 1 '!'>
```

More accurately, a literal that appears in a *sequence* of syntax rules, is dropped. So the literal in the sequence `(expr '.')` was dropped, while the literal in `['!']` was kept.

Parsing a choice rule results in the code-data from whichever choice matched

```
[std]: defop 50 '+++' ("/" expr | ":" expr "." expr)
[std]: `+++ / a
<std~:+++ a>
[std]: `+++ : a . b
<std~:+++ a b>
```

The astute reader may anticipate a problem:

```
[std]: defop 50 '+++' ("/" expr | ":" expr)
[std]: `+++ / a
<std~:+++ a>
[std]: `+++ : a
<std~:+++ a>
```

Both alternatives parsed into identical code-data, making it impossible for the to distinguish between the two. The solution brings us to a new topic – syntax annotation.

7.4.1 Syntax Annotation

The syntax language supports annotations that provide control over the structure of the resulting code-data. There are three kinds of annotation: rule names, optional-rule alternatives and trivial rules.

Named Rules

Rule names allow operands to be indexed by name rather than position:

```
[std]: defop 50 $left:expr '+++' $right:expr
[std]: exp = ` 1 +++ 2
[std]: exp
<std~:+++ right=2 left=1>
[std]: exp/left
1
[std]: exp/right
2
```

For any non-trivial operator, it is advisable to use names for the operands - your implementation function will be more resilient to changes in your syntax.

If the named rule is a sequence or a repeating rule, a nested flist appears in the code-data:

```
[std]: defop 0 'myfrom' $module:(symbol ("." symbol)*)
                        "import"
                        $names:(symbol ("," symbol)*)
[std]: `myfrom a.b.c import x, y, z
<std~:myfrom names=[x, y, z] module=[a, b, c]>
```

If an anonymous sub-list is required, The name can be omitted:

```
[std]: defop 0 'myfrom' $:(symbol ("." symbol)*)
                        "import"
                        $:(symbol ("," symbol)*)
[std]: `myfrom a.b.c import x, y, z
<std~:myfrom [a, b, c] [x, y, z]>
```

Optional-rule Alternatives

Optional-rule alternatives allow an alternative value to appear in the code-data when the optional syntax is not present.

```
[std]: defop 50 '+++' [expr]/boo
[std]: `+++
<std~:+++ 'boo'>
```

'-' has a special meaning: omit the operand altogether.

```
[std]: defop 50 '+++' [expr]/-
[std]: `+++ a
<std~:+++ a>
[std]: `+++
<std~:+++>
```

As we have seen, the default alternative in None. When an optional rule is *named* however, the alternative defaults to omit '-':

```
[std]: defop 0 "+++" $a:[expr]
[std]: `+++ 1
<std~:+++ a=1>
[std]: `+++
<std~:+++>
```

Trivial Rules

A trivial-rule is a rule that always matches and consumes no input. It is given a label that appears in the code-data as a string. These are particularly useful for identifying which path was selected in a choice rule:

```
[std]: defop 50 '+++' (<colon> ":" expr | <slash> "/" expr)
[std]: `+++ /1
<std~:+++ 'slash' 1>
[std]: `+++ :1
<std~:+++ 'colon' 1>
```

Another use of a trivial-rule is in adding a null choice to a choice rule

```
[std]: defop 50 '+++' ( <colon> ":" expr
                      | <slash> "/" expr
                      | <nothing>)
[std]: `+++
<std~:+++ 'nothing'>
```

In this situation, you may want to omit the label:

```
[std]: defop 50 '+++' ( <colon> ":" expr
                      | <slash> "/" expr
                      | <>)
[std]: `+++
<std~:+++>
```

A further use of this empty trivial-rule, is in omitting literals from the code-data. Recall that literals that occur in a *sequence* of rules are dropped. In this example

```
[std]: defop 50 "foo" ("!" | expr "?")
```

The "`!`" is not in a sequence, and hence will appear in the code data.

```
[std]: `foo !
<std~:foo "!">
```

Whereas in this definition

```
[std]: defop 50 "foo" ("!" <> | expr "?")
```

The "`!`" is in a sequence, and hence will be dropped from the code data.

```
[std]: `foo !
<std~:foo>
```

To see more examples of code-data, have a look at the definitions in `logix/std.lx` and `logix/base.lx`, and try quoting some of the Standard and Base Logix operators.

7.5 Free-text

Logix has powerful facilities for creating languages that incorporate unparsed text or *free-text*. These features can be used to create simple things like string or regex literals, or entire languages such as XML, where the text outside of tags should not be parsed. You could also create literate programming languages where only text specially marked will be parsed – the rest is documentation.

Free-text parsing is available through the syntax-rules `freetext` and `optext`.

7.5.1 The `freetext` rule

The `freetext` syntax-rule is used for recognizing blocks of pure text, where the parser will not look inside the text at all (except to see where it ends).

```
freetext /<regex>/
```

Or

```
freetext upto /<regex>/
```

The regex defines where the free-text will end. If `upto` is present, the regex defines the terminator, for example:

```
[std]: defop 0 '"' freetext upto /"/
```

(This is not a very good string literal operator, because it doesn't support escaped quotes inside the string.)

Without the `upto`, the regex specifies what will be included in the free-text:

```
[std]: defop 0 'name:' freetext /[a-z]*/ func x: x
[std]: name:tom
'tom'
```

The `upto` version can match free-text over multiple lines, whereas without `upto`, it is a syntax error if the terminator is not found on the current-line.

With `upto`, there is some control over where normal parsing will resume. Usually, the terminator will be discarded and parsing will resume immediately after. If however, the regex match contains a parenthesized group, parsing will resume at the start of the group. This allows tricks such as

```
defop 0 "text" freetext upto /(!|\?)/ ("!" ... | "?" ...)
```

In this example, the syntax of the remainder of the rule depends on whether the text ended with "`!`" or with "?".

7.5.2 The `optext` rule

The `optext` syntax rule is also used to recognize free-text, but with `optext`, the parser looks inside the free-text for operators. This allows us to have text with embedded operators, like XML tags for example.

```
optext /<regex>/
```

Or

```
optext@<language> /<regex>/
```

The regex always defines the terminator (as in `freetext upto`). It will generally be necessary to define a separate language with the operators that can be embedded in the text, and specify that language with the `@` clause.

There is a restriction on the kinds of operators that can be embedded in `optext`. The operators must:

- Have no left-hand-side, and either
  - be 'enclosed', i.e. always end with a literal, or
  - have a zero binding value.

In the second case – when the operator is not enclosed, the parser will continue parsing the right-hand-side until the end of the line. Note that line continuation rules (i.e. using indentation) still apply. Once the end of the line is reached, the parser goes back to recognizing free-text.

This example shows how to implement a sub-set of XML (with no attributes or empty tags).

```
deflang xmlcontent:
    defop 0 "<" $tag:freetext /[a-z0-9\-\:\.\_]+/ ">"
            $content:optext /</
            "/" $endTag:freetext /[a-z0-9\-\:\.\_]+/ ">"

defop 0 "<xml>" optext@xmlcontent /<\/xml>/
```

(Checking that the start and end tags match up will have to be done by the implementation function.)

7.5.3 Code-data for Free-text

Not surprisingly, free-text rules insert strings into the code data. Each occurrence of a `freetext`, will create a single string in the code-data.

```
[std]: defop 0 '"' freetext upto /"/
[std]: `"a string!"
<std~:" 'a string!'>
```

With `optext` the result is a list, alternating between strings and operators. The following example uses the `xmllang` from the previous section.

```
[xmllang]: `<xml>Hey - it's like xml!</xml>
<xmllang:<xml> "Hey - it's like xml!">
[xmllang]: `<xml>Hey - it's <i>like</i> xml!</xml>
<xmllang:<xml> "Hey - it's "
               (xmlcontent:< content=['like']
                            tag='i'
                            endTag='i')
               ' xml!'>
```

7.6 Implementation

We have already had a sneak preview of making these operators *do* something. We defined a simple alias for print.

```
[std]: defop 0 ">>" expr func x: print x
```

The operator is implemented with a function (as opposed to a macro); the single operand is passed to the argument `x`, and is printed.

As we saw previously, the syntax of `defop` is:

```
defop ['l' | 'r' ]
      ['smartspace']
      <binding>
      <syntax>
      [ <implementation> ]
```

Where <implementation> is:

```
('func' | 'macro') <argument-spec> ':' block
```

<argument-spec> is the same as for regular functions in Standard Logix, or if you prefer, it is like a Python argument list without the parentheses or commas. (Note operator arguments do not yet support argument predicates. They will!).

7.6.1 Functions

When the implementation begins func, the operator is implemented as a function. Any unnamed operands are passed to the function as positional arguments, in the same order that they appear in the syntax definition. Any named operands are passed as keyword arguments.

When operands are optional, you will generally want to provide a default value for the corresponding argument:

```
[std]: defop 0 ">>" $val:[expr] func val='huh?': print val
[std] >> 'hi'
hi
[std] >>
huh?
```

For operators defined in this way, the function is available at run-time. You can access it using the operator-quote '``'. When Logix encounters an operator-quote, it next parses a single token and returns the operator that token represents.

```
[std]: defop 0 "+++" expr func x: x + 1
[std]: ``+++
<operator std +++>
```

The operator function is available via the func attribute.

```
[std]: map [1..3] ``+++.func
[2, 3, 4]
```

7.6.2 Macros

When the implementation begins macro, the operator is implemented as a compile-time macro. If you are a Lisper, you are on familiar ground – the design of Logix's macro system was heavily influenced by Lisp macros. If you are not familiar with Lisp (or similar) macros, things are about to get interesting!

Brief Introduction to Macros

(This section contains no Logix specific information – feel free to skip ahead.)

All programmers are familiar with sub-routines (a.k.a. procedures or, with apologies to the purists, functions). They are a mechanism to capture some piece of computation and give it a name. This *abstraction* mechanism makes programs smaller: in memory, in source code, and crucially, in the programmer's brain. Abstractions allow us to forget about problems we have already solved, and to think more clearly about larger problems. Abstraction is the essence of programming – it is the weapon we use to fight the mind-boggling complexity of the problems we routinely tackle.

It is unfortunate then, that the sub-routine abstraction is rather limited. There are a very large class of programming patterns that it cannot capture. Every programmer has experienced this limitation while typing in a well rehearsed pattern of code, and filling in the blanks. The canonical example might be

```
for (int i = 0; i < len; i++) { ... }
```

Weren't computers supposed to eliminate mechanical repetition?

Macros are another kind of abstraction based on the concept of *source code transformation*. A macro mechanism is a *pre-processor* that takes the code you typed, and mangles it into some other form before the compiler gets a look-in. An individual macro is a kind of function – it takes fragments of your source-code as arguments, and returns a new source-code fragment. In a full procedural macro system, such as in Lisp or Logix, the macro-function is simply a regular function, will full access to the functionality and libraries of the language. The difference is that the function runs during this pre-processing phase, rather than at runtime. Also, in Lisp and in Logix, the source-code fragments that macro functions operate on are not simply bits of text (as they are in C's very limited macro pre-processor), they are structured data – abstract syntax trees.

In some ways macros are a kind of catch-all abstraction – they can capture any repeating patterns of code that defeat the language's other abstraction mechanisms. Powerful stuff! (Too powerful, in fact, according to some. A language with macros essentially allows any programmer to also be a language designer. With such a situation, what kind of language will you end up with? Answer: exactly the language that *you* want – except when you are maintaining someone else's code of course. Let the debate continue!)

Wherever a pattern of code is found to crop up again and again, and when those repetitions vary in some systematic manner, a macro can be employed to make the code more concise and more abstract. The macro

engine will *transform* your compact version of the pattern into the full version. If, for example you often see the pattern

```
for (int i = 0; i < len; i++) { ... }
```

with only the variable names and loop body differing from one occurrence to the next, then you have an opportunity to employ a macro. You might define a macro looking like

```
count i upto len { ... }
```

The macro engine would transform this short form into the longer form that the compiler understands, and (joy of joys!) you will never have to type it out the long way again.

Logix macro operators are implemented by a function, just like the regular operators we have already seen. The difference is that this function is called during the macro-expansion phase – after parsing and before compilation. With regular operators, the operands are first evaluated, and the results are passed to the implementation function. With macro operators, the operands are parsed, and the resulting code-data is passed to the implementation function. The function assembles and returns some new code-data, which is inserted into the overall parsed code in place of the original macro call.

By way of a simple example, we can create a macro that 'zaps' a variable (sets its value to `None`). It is generally a good starting point to think about the kind of code you want to generate. In this case

```
x = None
```

What would the equivalent code-data look like? We can find out with the back-quote operator

```
[std]: `x = None
<std:= x None>
```

To produce that structure programmatically, we can use the operator-quote.

```
[std]: ``= ~x None
<std:= x None>
```

Here, ``= returned the operator class. We instantiated the object in the normal way – simply by calling the class, passing the operands as arguments.

We can verify this works correctly using `logix.eval`.

```
[std]: x = 108
[std]: logix.eval (``= ~x None)
[std]: x is None
True
```

We can now build the `zap` operator:

```
[std]: defop 0 'zap' expr macro placex: ``= placex None
[std]: x = 108
[std]: zap x
[std]: x is None
True
```

The only difference in our code-data is that the target of the assignment is now parameterized. We used the name `placex`, because the target of the zap is not a variable, but any assignable place, e.g.:

```
[std]: zap foo.baa.zob
```

The `x` suffix is conventional – an abbreviation for *expression*, i.e. the parameter is a place expression. The suffix reminds the reader that the variable holds an *expression* (i.e. some code-data), not a run-time value.

Two useful functions for learning about and debugging macros are `logix.macroexpand` and `logix.macroexpand1`. They both take a some code-data, perform macro expansion, and return the result. `macroexpand` can be passed any code-data, and expands *all* macros it contains. If the resulting code-data also contains macro operators, these are also expanded, and so on until no macro operators remain. `macroexpand1` expands only a single macro at the top level of the passed expression.

For example:

```
[std]: logix.macroexpand `zap x
(base.= x None)
```

We introduced a technique here which is very useful when writing macros: before starting, use the back-quote operator to discover what kind of code-data we should assemble. Here, for example, is the `count-upto` macro (from the introduction to macros). First we should see what a simple counting loop looks like as code-data:

```
[std]: `for i in [0..i]: dosomething
<base:for i (std:[ 0 i 'range') body=[dosomething]>
```

Now we can define the operator:

```
[std]: defop 0 "count" expr "upto" expr ":" block
           macro placex tox body:
               ``for placex (``[ 0 tox 'range') body=body
[std]: count i upto 3: print i
0
1
2
3
```

If you think the above code-data expression looks somewhat painful, you're right! You should learn about quasiquoting.

7.6.3 Quasiquoting

Implementing macros is much easier using quasiquoting, for example the previous `count-upto` macro looks like this:

```
[std]: defop 0 "count" expr "upto" expr ":" block
           macro placex tox body:
               `for \placex in [0..\tox]: \*body
```

The back-quote operator we have been using all along is in fact a *quasiquote* operator, which means it has extra smarts when used in conjunction with the quote-escape operator '\' (equivalent to the comma in Lisp – we prefer to keep the comma free for other uses). Quasiquoting is a templating mechanism. With it you can generate code-data easily, by plugging parameterized code-data into a known template structure. For example:

```
[std]: varname = ~foo
[std]: ` \varname = None
<std:= foo None>
```

The quote expression returned an assignment expression as expected, but the target of the assignment (`foo`) came not from the quoted code, but from a run-time value – the contents of the variable `varname`. The `\` operator is called quote-escape because its operand is not quoted – it is evaluated, and the result is plugged into the resulting code-data.

With this simple extension of the quote operator, we can now use quoting in our macro implementations. Here is `zap` revisited:

```
[std]: defop 0 'zap' expr macro placex: ` \placex = None
```

The code-data to perform the assignment now looks pretty much like a regular assignment statement. The only difference is that the target of the assignment is escaped (or, if you prefer, parameterized), because it will vary from one application of `zap` to the next.

7.6.4 Variable Capture and gensyms

When creating macros, you need to be aware of an issue known as variable capture. Consider this operator:

```
defop 0 'repeat' expr ':' expr
    macro countx exp: `for i in range \countx: \exp
```

Nice and easy right? Wrong!

```
[std]: i = 'crucially important data'
[std]: repeat 2: print i
0
1
[std]: i
1
```

The expanded code modified the variable `i`, which happened to be in use!

Whenever your macro-generated code required variables, you need to pick a name that you know will not be in use. Fortunately there is an operator that does this for you. `gensyms` creates symbols with names that are guaranteed to be distinct from any other names that might be in use.

```
[std]: gensyms a b
[std]: a
~#a2
[std]: b
~#b3
```

A correct implementation of `repeat` would look like:

```
defop 0 'repeat' expr ':' expr
    macro countx exp:
        gensyms i
        `for \i in range \countx: \exp
```

Note that `gensyms` is part of Standard Logix. To create a gensym from other languages, call the function `logix.gensym()`. The function can be passed an optional string which will be incorporated into the name of the gensym (which helps make macro-generated code-data more readable).

7.6.5 Splicing

We have seen how the quote-escape operator inserts a single value into quoted code-data. Sometimes we may need to insert all the items from a sequence, i.e. to 'splice' the sequence into the code-data. The `\*` operator does this. You may have noticed it being used in `count-upto`:

```
[std]: defop 0 "count" expr "upto" expr ":" block
           macro placex tox body:
               `for \placex in [0..\tox]: \*body
```

Because the parameter `body` comes from a `block` operand, it will contain a sequence of statements. These statements need to be spliced into the code-data in order to generate the correct structure.

7.6.6 Local-Module Escape

The code that a macro expands to will often need to access specific modules or functions. The macro implementer needs to take care because there is no relying on the namespace where the macro is expanded. The local-module-escape "\@" provides a convenient means to access the macro-defining module, from the expanded code (i.e. in the macro-using module).

In the following example, the expanded code needs access to the `re` module.

```
import re
deflang relang:
    defop 0 "regex" symbol ":" freetext /.*/
        macro name r: ` \name = \@.re.compile \r.strip()
```

The \@ expands to (code that evaluates to) a reference to the current module.

7.6.7 Nested quotes

Sometimes, it is necessary to nest a quoted expression inside another quote. This is most common when defining macro-defining macros (yes, a macro can expand to a `defop` – it works!). To escape both the quotes, use a double escape, i.e. '\\'. In general, multiple backslashes may be used together and each one will escape one quote. By way of an example, the operator `makePrinterOp` is a macro defining macro. It creates a new operator that simply prints itself. (the operator `lit` creates a literal-rule in the syntax, it is only generally used in operator-defining macros.)

```
[std]: defop 0 "makePrinterOp" symbol
           macro sym:
               s = str sym
               `defop 0 lit \s macro: `print \\s + '!'
[std]: makePrinterOp argh
[std]: argh
argh!
```

7.6.8 Context Aware Macros

Usually, the expansion of a macro is entirely determined by its contents. In other words, the macro function is a pure-function, where the result depends only on the arguments.

In a few situations however, it is necessary to build macros where the expanded code also depends on the *context* of the macro. An example of this is the `breakwith` macro from Standard Logix. The result of the `breakwith` expression needs to be assigned to a temporary variable – a gensym which is created in the surrounding `valfor` macro.

If a macro function defines an argument `__context__`, it will be passed a `MacroContext` object. The object is like a dict with support for nested scopes. Any value set in the context will be available to nested macros in the same module, unless the value is shadowed by a nested macro.

For an example, see the definition of `valfor`/`breakwith` in Standard Logix.

7.7 Multiple Languages

Logix is a multi-language programming system. There may be many languages in existence at one time, and we can freely switch between them. In this section we shall look at how to switch between languages, how to create new languages and how to create operators that elegantly combine multiple languages.

7.7.1 `setlang`

`setlang` switches to a different current language. It takes a single operand – a language object. We have seen three languages in this tutorial: Standard and Base Logix, and the syntax-rule language. The corresponding language objects are available via the `logix` module.

```
[std]: def f a b: a+b
[std]: setlang logix.baselang
[base]: f(1,3)
4
[base]: setlang logix.stdlang
[std]: f 1 3
4
# DON'T DO THIS!
[std]: setlang logix.syntaxlang
[syntax]: expr '+' expr
<SequenceRule (expr "+" expr)>
```

If you ignored the above warning, have fun trying to get back to `logix.stdlang`! The syntax language does not have the dot or the `setlang` operator. If you are in IPython, with a magic-command to get back to Python, you are in luck:

```
[syntax]: lx
In [40]: # Now in Python
In [41]: __currentlang__ = logix.stdlang
In [42]: lx
[std]:
```

From which we can also see that the interactive Logix top-level stores the current language in `__currentlang__`.

Another use of `setlang` is to create a block-local language. If you place a `setlang` statement in a block, the remainder of the block (or up to the next `setlang`) will be parsed in the specified language.

The `setlang` operator can be used in source files in the same manner as we have seen here.

Some important points to be aware of:

- If you have used `defop` at the top-level, the operators created will be temporary. When you change languages with `setlang`, those operators will be lost.
- If you use `setlang` in a block (i.e. not at the top-level), it does not behave like a regular statement – it is a compiler pragma. The expression that specifies the new language is evaluated at parse-time, as soon as the parser encounters the `setlang`. The expression is evaluated as if it were at the top-level: the only variables visible will be globals that have already been created by previous top-level statements. Woe betide you if you use an expression with side-effects in a `setlang`!

7.7.2 deflang

We have seen how to use `defop` to create (or redefine) operators. So far, the operators have been temporary – not part of a language that can be re-used in other parts of the program. As well as at the top-level, `defop` can be used with a `deflang`.

```
deflang <name> [ "(" <base-language> ")" ] ":" <body>
```

`deflang` creates a new language (an instance of `logix.Language`). Inside the body, the new language can be populated with operators and other features. The body of a `deflang` is the only place, other than the top-level, that a `defop` can be used.

Here is a simple language that only knows how to do one thing: add.

```
[std]: deflang addlang:
     :     defop 50 expr "+" expr func a b: a+b
```

Here is how to experiment with the new language:

```
[std]: std = logix.stdlang                    (we'll need this in a moment)
[std]: setlang addlang
[addlang]: 1                                        (numeric literals work)
1
[addlang]: "a"
ERROR                    (even string literals are language-defined operators)
[addlang]: 1 + 2
3
[addlang]: 1 – 2
ERROR                                  (the language only has the + operator)
[addlang]: setlang std
[std]:
```

Did it surprise you that `setlang` was available? It was inherited. The language that defines `setlang` (as well as `defop`, `deflang` and a few others) is the default base-language. More on language inheritance in *7.7.7 Language Inheritance.*

As well as operator definitions, `deflang` can contain regular statements. In a similar fashion to Python classes, any variables created in the block become attributes of the language. This gives us a convenient place to implement support functions for operators. This is a ridiculous example but you get the idea:

```
deflang addlang:
    def add a b: a + b

    defop 50 expr "+" expr func a b: addlang.add a b
```

Note the local function was accessed as an attribute of the language object.

Using `setlang` within `deflang`

Inside the `deflang`, a `setlang` can be used to switch to an alternative implementation language. It is even possible to `setlang` to the language being created. If you do this, the new operators will become available one by one, as they are defined.

7.7.3 Expressions and Terms Again: The Continuation Operator

We have seen that the meaning of `expr` is dependent on the language. It is defined by a special operator called the *continuation operator*. An `expr` is like a `term`, but where the end of a `term` would be, the parser continues parsing the `expr` according to the syntax of the continuation operator.

Recall that the following Standard Logix function call is not a valid `term`.

```
f x y z
```

There are no infix operators to join the parts together. You can think of the continuation operator as an invisible infix operator that is inserted where a `term`

would end. If the operator was visible and explicit, the above function call might look like

```
f __continue__ x y z
```

Which would be valid syntax, if the operator was defined something like this:

```
defop 100 expr "__continue__" term*
```

To define the continuation operator for a language, include a definition just like this one, except for one difference. The left-hand-side of the continuation operator is not specified, so the definition would actually look like

```
defop 100 "__continue__" term*
```

The continuation operator is not used explicitly of course. If a language included the above definition, the original statement:

```
f x y z
```

would be valid.

In code-data, the continuation operator is displayed as an operator with no token:

```
[std]: `f a + b
<std:+ (std: f a) b>
```

Note how the plus is displayed "`std:+`" whereas the continuation operator that combined the `f` and `a` is simply "`std:`".

7.7.4 The Switchlang Operator

To evaluate a sub-expression in a given language, use the *switchlang* operator:

```
(:<language> <expression-in-that-language>)
```

For example, we can use the operator to create syntax-rules

```
[std]: print (:logix.syntaxlang 'a' 'b')
<SequenceRule ("a" "b")>
```

As with `setlang`, the expression that defines the language will be evaluated at parse-time, as if it was at the top-level (see *7.7.1 setlang*)

7.7.5 Language Specific Operands

The syntax-rule language has another trick up its sleeve. The language of an operand may be specified using '@'.

```
[std]: rl = logix.syntaxlang
[std]: defop 0 "printrule" expr@rl func x: print x
[std]: printrule 'a' 'b'+ block
<SequenceRule ("a" "b"+ block)>
```

As well as specifying a language, you can use @^ to specify the operand should be parsed in whatever language was in effect prior to the current language.

[Need an example]

7.7.6 The Outer-Language Operator

In the previous example, we had to employ the local variable `rl`, since the dot cannot be used within the syntax definition. An alternative is to use the outer-language operator:

```
 (^ <expr>)
```

which simply evaluates an expression in the language that was in effect prior to the current language. There is often a need to embed a general expression (e.g. a Standard Logix expression) inside a domain-specific expression (e.g. a syntax-rule expression). This is the purpose of the outer language operator.

Here is `printrule` again using the outer-language operator.

```
defop 0 "printrule" expr@(^logix.syntaxlang) func x: print x
```

Note that you could define the outer-language operator yourself:

```
defop 0 "(^" expr@^ ")" func x: x
```

7.7.7 Language Inheritance

Often, one does not want to define an entire language from scratch, but to create a language that is mostly like some existing language, with some new or redefined operators. Logix supports this through language inheritance. The `deflang` operator allows a (single) base-language to be specified, for example:

```
[std]: deflang mylang(logix.stdlang):
     :     ...
```

In this example, the new language inherits all of the Standard Logix operators. New operators may be added, and existing ones redefined.

When no base language is specified, it defaults to `logix.langlang`. To create a completely empty language with no operators at all:

```
[std]: deflang empty(None):
     :     ...
```

7.7.8 The Alternative to Inheritance: `getops`

Language inheritance is useful when you want to create a new language that is largely like an existing language. An alternative facility for reusing existing operators is `getops`.

```
getops <language> [, <operator> <operator> ...]
```

`getops` is for the situation where a new language is mostly unlike existing languages, but you wish to reuse a few existing operators.

```
[std]: setlang logix.baselang
[base]: 1 isa int
     1 isa int
       ^
SyntaxError: unexpected 'isa'
[base]: getops logix.stdlang, isa *>
[base]: 1 isa int
True
[base]: [1, 2, 3] *> lambda x: x*2
[2, 4, 6]
```

In this example the imported operators are temporary – they will be lost on the next `setlang`. To make them permanent members of a language, use `getops` inside a `deflang`.

```
deflang mylang:
    getops stdlang, { forany forall
    ... rest of language definition (may include more getops)
```

Note that we always mention the operator simply by giving the operator token. That is why the previous example has the unusual appearance of an open brace without the corresponding close brace (to import the lightweight lambda syntax `{...}`).

Another benefit of `getops` is that it allows libraries to provide special-purpose operators in a language neutral manner. To see the benefit consider the following definition which creates a new language, adding some XML-like syntax to Standard Logix.

```
deflang xmllang(logix.stdlang):
    defop 0 "<xml>" ...
```

The commitment to Standard Logix is unfortunate – why can't Base Logix code also have access to this operator? Worse, what happens if we have many such languages, each that add one or two operators to Standard Logix? How do we combine them if our program needs several of the new operators?

A better approach is to use `getops`. First, define `xmllang` so that it does not extend Standard Logix.

```
deflang xmllang:
    defop 0 "<xml>" optext@xmlcontent /<\/xml>/
```

Next, in the module where the XML syntax is required:

```
import xmllang
getops xmllang.xmllang, <xml>
```

Again, bear in mind that in this example, the imported operator is temporary – it will be lost on the first `setlang`.

Finally, a `getops` that specifies no operators will import all the operators from a given language. For example:

```
import xmllang
getops xmllang.xmllang
```

7.7.9 Operator Base-class

As has been described, each `defop` creates a new class to represent the defined operator in code-data. By default, the class has the base class

`logix.language.BaseOperator`.

You can customize this behavior by assigning your own class to the language attribute `operatorBase` inside a `deflang`. If you set this attribute to a custom class, it is advisable that the class inherits from `BaseOperator`.

8 Logix from Python

Logix can easily be used as a component of a regular Python project.

8.1 The `logix` Module

The `logix` module is a regular Python module and provides the following methods:

`logix.init()`

Initialize Logix. The standard languages are created.

`logix.imp(logixModuleName)`

Import a Logix module (`.lx`) file. The argument is the module name in standard dotted format. The imported module is returned. E.g.:

```
config = logix.imp("utils.config")
```

`logix.execfile(filename, [globals])`

Parse and execute the file `filename`, in the (optionally) supplied environment. The parse begins in Base Logix.

`logix.parse(language, filename=None, mode='parse', execenv=None)`

Parse source-code in the target language. Returning either the parsed code-data, or the compiled code (a Python code-object).

`language`	The language to parse. (in an active parse, the language may be changed during the parse by a `setlang`)
`src`	The source-code. Either a string or a file object. File objects should be opened in universal-newline mode.
`filename`	Optional filename (string). Used in error messages.

`mode` — A string to specify the parse mode. All modes except `interactive` will parse multiple lines. All modes except `execmodule` return code-data.

- `exec` — Expand macros. Evaluate top-level items and items in `deflang`s.
- `execmodule` — Like `exec` but returns a Python `codeobject`.
- `interactive` — Parses a single expression only. Expands macros, evaluates language expressions in `setlang` and `switchlang`. `defop`, `deflang` and `getops` have no effect during the parse.
- `parse` — Parse only. No macro expansion. No evaluation (language extension features have no effect during the parse).
- `expand` — Parse and expand macros. No evaluation (language extension features have no effect during the parse).

`execenv` — A dict containing the execution environment (name bindings) for code-evaluation during the parse. An `execenv` is required if the mode is `exec`, `execmodule`, or `interactive`.

`logix.eval(codeData, [globals, [locals]])`

Evaluate the code-data either in the calling environment, or in the supplied environment. Returns the result. If the code-data contains macro operators, these will be expanded before evaluation.

The following languages are available (once you have called `logix.init()`):

`logix.baselang`

Base Logix

`Logix.stdlang`

Standard Logix

`logix.syntaxlang`

The syntax-rule language.

`logix.langlang`

The 'language definition language' which defines `defop`, `deflang`, `setlang`, `getops` and a few others. This is the default base language for new languages (when no base language is specified in a `deflang`).

`logix.quotelang`

Defines the quasiquote operator and quote-escape operator. The base language for `logix.langlang`.

8.2 Example

This short example shows how Logix might be used to implement a command-line interface to a Python application. The language `commandlang` has been defined in `commandline.lx`. The operator implementation functions have been written to access the application via a global variable `app`.

We don't wish to support multiple languages at our command-line, so we do not need to evaluate during parsing. We therefore use the default parse mode (`'parse'`).

The main body of the application might look like:

```
import logix

commandline = logix.imp("commandline")
commandlang = commandline.commandlang

theApp ...create main application object...

env = dict(app=theApp)

while 1:
    line = raw_input("> ")
    codeData = logix.parse(commandlang, line)
    logix.eval(codeData, env)
```

9 Logix-Test

Logix-Test is a Logix powered unit testing framework. It is available as a separate download from the Logix download page, and installs as the package `ltest`. The package consists of two major components. A command-line test-runner (`ltest.runner`), and a test language (`ltest.testlang`) which is an extension of Standard Logix.

The test language provides:

- Extremely concise definition of tests: boilerplate has been kept to an absolute minimum
- Compact assertion operators
- Object pattern-language for conveniently implementing object tests
- Advanced mock-object support

With test-driven-development the ratio of test code to application code often approaches one-to-one. With so much test code being written, there is a great deal to be gained by using a language tailored to the task. Tests developed using Logix-Test are considerably more compact, easier to maintain and more readable than tests written in a general purpose language.

For more examples of the features described in this section, see the unit-tests for Logix itself, which are written using the `ltest` framework. They are included in the standard distribution.

9.1 Defining Tests

Across various unit-testing frameworks, the structure of tests and test-suites has a lot in common. Testing always consists of three stages

- Setup – create some pertinent application data
- Operations – perform some operations that are under test
- Assertions – test if the outcome of the operations is as expected

In practice, the test operations and assertions are often intermingled.

In some cases, a fourth stage is required to keep the state of the system clean for subsequent tests

- Teardown – ensure side-effects do not affect subsequent tests

The state created by the setup phase is sometimes called the *fixture*. Because a number of tests will commonly require a similar fixture, frameworks typically provide mechanisms to re-use the setup phase.

In the standard Python unit-test framework, for example, the fixture is created in a method of a class. The class may then contain multiple test methods, each of which share the same fixture. The fixture may also be inherited and extended in other test classes using regular Python inheritance.

The `ltest` framework provides similar facilities but is not class based. Instead it used nested scopes to make test definition much more concise.

The overall structure of an `ltest` file is:

```
limport ltest.testlang
setlang testlang.testlang

import <stuff-to-be-tested>

defsuite <suite-name>:
    <setup>

    deftest myFirstTest:
        <test operations>
        <test assertions>

    deftest myOtherTest:
        <test operations>
        <test assertions>

    <teardown>
```

For example, here are some tests on the built-in string type.

```
limport ltest.testlang
setlang testlang.testlang

defsuite main:
    print "Setup"
    empty = ""
    one = "a"
    a = "ho hum"

    deftest append:
        print "In test: append"
        empty + "!" ?= "!"
        one + "!"   ?= "a!"
        a + "!"     ?= "ho hum!"

    deftest count:
        print "In test: count"
        empty.count 'x' ?= 0

        one.count 'a' ?= 1
        one.count 'b' ?= 0

        a.count 'x' ?= 0
        a.count 'o' ?= 1
        a.count 'h' ?= 2

    print "Teardown"
    print
```

There are a few things to note here.

- The ?= operator is ‘assert equal’
- Regular nested scoping rules allow variables created in the setup phase to be used directly in the tests.
- Some redundant print statements have been included to demonstrate the order of execution.

The framework is designed to be use from the Logix prompt. The test file is simply a regular module. The suite can be run be calling it:

```
[std]: limport examples.stringtest
[std]: stringtest.main()
Setup
In test: append
Teardown

Setup
In test: count
Teardown

Ran 2 tests, 9 assertions, with 0 failures
```

From the print statements, we can see the tests were executed in the order they appear in the source-code, and that the setup and teardown code was executed before and after each test.

Another level of structure can be introduced with `defgroup`:

```
defsuite main:
    print "Setup"
    empty = ""
    one = "a"
    a = "ho hum"

    deftest append:
        print "In test: append"
        ...
    deftest count:
        print "In test: count"
        ...

    defgroup unicode:
        print "Setup unicode"
        uempty = unicode empty
        uone = unicode one
        ua = unicode a

        deftest backToStr:
            print "In test: backToStr"
            str uempty ?= empty
            str uone   ?= one
            str ua     ?= a

        print "Teardown unicode"

    print "Teardown"
    print
```

`defgroup` introduces a nested set of tests with their own setup and teardown.

For each test in the sub-group, both the top-level setup and the group setup will run, then the test runs, and finally the group teardown and top-level teardown runs.

```
[std]: limport examples.stringtest2
[std]: stringtest2.main()
Setup
In test: append
Teardown

Setup
In test: count
Teardown

Setup
Setup unicode
In test: backToStr
Teardown unicode
Teardown

Ran 3 tests, 12 assertions, with 0 failures
```

Test groups created with `defgroup` can also contain further sub-groups, nested to any depth.

9.2 The Test-Runner

As we have seen, the test-runner is used from the interactive prompt. It is started by invoking the required suite function.

If a test fails, the code around the problem is displayed, along with any message associated with the failure. The test-runner then presents a prompt. The prompt recognizes a number of simple commands

`q`	Quit testing without running any more tests.
`d`	Enter the debugger at the point of failure. [Currently requires IPython]
`c`	Continue running tests. If the test failed due to an exception, testing resumes with the next test. If the problem was an assertion failure, testing continues within the same test.
`r`	Redo the same test from the beginning.
`e`	Jump emacs to the point of failure. [May not work in your environment, uses `gnuserve`. See `ltest.runner.emacsto`]

The suite function accepts the following keyword arguments:

`verbose`	True/false. The runner will report its progress.
`Select`	Only execute the specified test or test-group. E.g. `select="network.errors"`

9.3 Assertion Operators

The following assertion operators are available:

`<expr> ?`	Assert true
`<expr> ?!`	Assert false
`<expr> ?= <expr>`	Assert equal
`<expr> ?!= <expr>`	Assert not equal
`<expr> ?? <expr>`	Assert LHS matches RHS pattern
`<expr> ?raises <expr>`	Assert LHS raises an exception matching RHS pattern

The last two operators use object-patterns, described in the next section.

9.4 Object Patterns

Often, the subject of assertions are not simple values but complex objects, containing many attributes which may themselves be object structures. There are two common approaches to this.

- Use multiple assertions, one for each attribute and sub-attribute.
- Provide an equality testing method, and compare the object to a known value created in the test-code.

Writing many assertions quickly gets tedious. Equality tests can be lengthy and are not very informative: when two objects are found to be not equal, there will be no indication why. The tester is often forced to use a post-mortem debugger to discover what part of the object state caused the inequality.

Logix-Test object patterns are an alternative which is often preferable. They provide two main advantages: they are very concise, and if a match fails, the pattern matching mechanism will report exactly where.

The pattern captures the expected type, attribute values, and 'contents' of an object ('contents' refers to the values returned by the subscript operator, or if you prefer, by the `__getitem__` method).

We can explore the syntax by switching to the test-language at the Logix prompt.

```
[std]: limport ltest.testlang
[std]: setlang testlang.testlang
[testlang]:
```

9.4.1 Testing Object Type

A simple pattern that matches objects of type `str` looks like.

```
{:str}
```

Patterns are designed to be used within the context of a test, using the pattern assertion operator described above, e.g.:

```
someObject ?? {:str}
```

They can also be used at the prompt by calling the `test` method. It returns `True` if the pattern matches, or an diagnostic message if it does not.

```
[testlang]: {:str}.test 'a'
True
[testlang]: print {:str}.test 1
: wrong type:
1
Expected type: str
Found type   : int
```

Note that subtypes will also match.

```
[testlang]: {:basestring}.test 'a'
True
```

Strict type matching can be specified by appending a '`!`' to the type.

```
[testlang]: print {:basestring!}.test 'a'
: wrong type:
'a'
Expected type: str
Found type   : basestring
```

9.4.2 Testing Attributes

The pattern can also include tests on object attributes.

```
[testlang]: class O:
[testlang]: o = O()
[testlang]: o.a=1
[testlang]: o.b=2
[testlang]: {:O a=1}.test o
True
[testlang]: print {:O a=2}.test o
.a: not equal:
2
---
1
[testlang]: {:O a=1 b=2}.test o
True
```

Note that the pattern is not an exhaustive description of the object. Any attributes not mentioned are simply not tested.

To simply test is a value is a true value:

```
[testlang]: {:O a?}.test o
True
[testlang]: o.a=0
[testlang]: print {:O a?}.test o
.a: 0 is not True
```

If you do not wish to specify a type, put * for the type.

```
[testlang]: {:* a=1 b=2}.test o
True
```

9.4.3 Testing Contents

If the object supports the list-indexing operator (`/[ ]` in Standard Logix), the pattern can test the values returned for given keys. Currently only string and integer keys are supported.

Values that appear directly in the pattern are compared with keys 0, 1, 2 and so on.

```
[testlang]: l = [5, 6, 7, 8]
[testlang]: {:* 5 6 7 8}.test l
True
[testlang]: {:* 5 6}.test l
True
```

To test for a specific key, use `/` in front of the key.

```
[testlang]: {:* /3=8}
True
```

To test for all the numeric keys use `/*`

```
[testlang]: {:* /*=[5, 6, 7, 8]}.test l
True
[testlang]: print {:* /*=[5, 6, 7]}.test l
: length mismatch:
[5, 6, 7]
---
[5, 6, 7, 8]
```

String keys can also be used. Names are automatically quoted.

```
[testlang]: d = dict foo=1 baa=2
[testlang]: {:* /foo=1}.test d
True
[testlang]: print {:* fo=1}.test d
: no such location /fo
```

Explicit quotes can also be used.

```
[testlang]: {:* /"foo"=1}.test d
True
```

9.4.4 Nested Patterns

The value with which a comparison is made can itself be a pattern

```
[testlang]: o.a = O()
[testlang]: o.a.x = 'ho'
[testlang]: o.a.y = 'hum'
[testlang]: {:O a={:O x='ho' y='hum'}}.test o
True
[testlang]: {:O a={:O x='ho' y='hummm'}}.test o
.a.y: not equal:
'hummm'
---
'hum'
```

Lists can also contain patterns

```
[testlang]: o.l = [1, [2, 3, 4], 5]
[testlang]: {:O l=[1, {:* /2=4}, 5]}.test o
True
```

9.4.5 Test Functions

So far we have tested only for equality. If you compare an attribute with a function, the function is called as a predicate.

```
[testlang]: {:O l={len it == 3}}.test o
True
[testlang]: print {:O l={len it == 2}}.test o
.l: test failed for:
[1, [2, 3, 4], 5]
```

9.4.6 Using Regexes.

If value being tested is a string, a regex can be used as the test value.

```
[testlang]: import re
[testlang]: o.s = "Ugh! So many features to document"
[testlang]: {:O s=/fe[a-z]t/}.test o
True
```

9.4.7 Object Expression

At the end of the pattern, an arbitrary test can be included after an &. The tested object is available as `it`. For example

```
{:Customer & it in preferred}
```

9.5 Mock-Objects

Mock-objects are a great idea for unit tests. They can make tests much more compact and maintainable. They make it much easier to isolate tests, i.e. to avoid accessing parts of the program that are not the subject of the test. They yield tests that 'fail fast'. They help avoid making your code test aware. They make it much easier to test for error conditions (e.g. simulate a network error). (See www.mockobjects.com for more information).

However, mock-objects have their down side too. Mock-object frameworks tend to be rather limited in their facilities for expressing the ordering of expectations. Sometimes you may be forced to write over-constraining tests – say, requiring certain events to occur in sequence, when in fact all you care about is that they all occur once. On other occasions, it may be difficult to specify that expectation X must occur before expectation Y, particularly if they are expectations of different mock-objects.

Another down-side, is that mock-objects can be very tedious to write.

Logix-Test provides a mock-object framework that attempts to address these difficulties. It features a compact, declarative syntax for creating mock-objects and specifying expectations. It supports a flexible style of ordering expectations, which can be applied to both individual mock-objects, and whole groups.

9.5.1 Creating Mock-Objects

The operator `defmob` creates a new mock-object. The mock is given a name and an expectation. Here is a simple mock called '`foo`' that expects a single call to the method `callme`, and will return `None` to the caller:

```
defmob foo callme() -> None
```

The general form is

```
defmob <name> <expectation>
```

The expectation is written in a special mini-language. Here are some expectations in the language:

Expression	Expectation
`turnRight() -> True`	Call to `turnRight` with no arguments. Returns `True`.

`accelerate 50 -> None`	Call to accelerate with argument 50. Returns `None`.
`speed = 100`	Set attribute `speed` to 100.
`speed -> 50`	Get attribute `speed`. Returns 50
`/foo -> 0`	Get item `"foo"`. Returns 0
`/(x) -> 0`	Get item `x` (i.e. the key is the run-time value of variable `x`). Returns 0
`/foo = 100`	Expect item `"foo"` to be set to 100. [NOT IMPLEMENTED]

`seq:`
`  <expectation1>`
`  <expectation2>`
	The given expectations in sequence

`<exp1> ; <exp1>`	(alternative syntax for `seq`)

`par:`
`  <expectation1>`
`  <expectation2>`
	The given expectations in any order

`<exp1> || <exp2>`	(alternative syntax for `par`)
`<expectation>*`	Zero or more occurrences of the given expectation

For example

```
defmob customer par:
    name -> "I'm a mock customer" *
    seq:
        salary = 100
        promoteTo 'manager' -> True
        salary -> 150
```

This mock object expects the `salary` attribute to be set to 100, followed by a call to `promoteTo`, and finally expected the `salary` attribute to be accessed. It also expects any number of accesses to the `name` attribute, arbitrarily interleaved with this sequence.

9.5.2 Expecting Values

As seen, a method call expectation can contain expectations for the values passed as arguments. An attribute-set expectation contains an expectation for the value being set.

There are three ways in which these expected values can be expressed: By equal value, by predicate function and by object pattern. The default is that the expected value and the value actually passed should be equal. E.g.

```
promoteTo 'Manager' -> True
```

and

```
speed = 100
```

If however, you pass a function as the expected value, it will be used as a predicate to test the value actually passed.

```
speed = {it > 100}
```

If you do not wish to constrain the value, you could define a function like:

```
def any x: True
```

This can be used directly in expectations:

```
promoteTo any -> True
```

Take care if the expected value is itself a function. Say you expect the function `f` to be passed. The following expectation is incorrect:

```
callback = f
```

The function `f` will be interpreted as a predicate and called by the mock-object framework! Instead write

```
callback = {it==f}
```

Finally, if the expected value is an object pattern, the value actually passed will be tested against this pattern.

```
setSalesContact {:SalesExec rank=4} -> None
```

9.5.3 Mock Groups

The purpose of a mock object is to simulate, for testing purposes, the environment that some piece of code expects to be in. The expected environment will often consist of multiple objects, and it is useful to be able to specify patterns of expectations in terms of the whole group. For example “expect a call to `x.m` before any calls to `y.m`”.

This is the purpose of the `mobgroup` operator.

```
mobgroup <name> <name> ... expecting <group-expectation>
```

`mobgroup` allows multiple mock-objects to be created, and a single expectation to be defined for the entire group. The expectation language is extended so that the traditional dot operator can be used to specify which object the expectation

applies to. The `/` operator can also be used to specify item access on a particular mock-object.

For example:

```
mobgroup employee department expecting par:
    department.name -> 'sales' *
    seq:
        department/"manager" -> employee
        par:
            employee.salary -> 1000000 *
            employee.name -> 'mock employee *
            employee.fire() -> None
```

9.5.4 Faking Object Type

Sometimes, the code being tested will incorporate type-checks (such as `isinstance`). In this case, it may not function correctly with mock-objects. To overcome this you can create mock objects that extend a class of your choosing. All attribute access to the mock-object is intercepted, so the behavior will be unaffected.

To define a mock object with a base class using `defmob`:

```
defmob mockCustomer(Customer) ...
```

And using `mobgroup`:

```
mobgroup e(Employee) dpt(Department) ...
```

9.5.5 Confirming Pattern Completion

Mock objects will raise an exception if the client code does something unexpected. If however, the problem is that the client code did *not* do all the things that were expected, there may not be an exception raised. In other words, some tests will need to conclude by confirming that the pattern of expectations has completed.

To confirm an individual mock-object has received all of its expectations, call the `confirmDone` function from module `testlang`.

```
testlang.confirmDone myMob
```

To confirm a group expectation has fully completed, keep a reference to the group object, and call its `confirmDone` method.

```
g = mobgroup a b c expecting ...
... test actions
g.confirmDone()
```

9.5.6 Debugging

The expectation language defines a postfix operator `:debug` which can be applied to any expectation. This causes the test to break into the debugger when that expectation is met.

```
defmob customer par:
    name -> "I'm a mock customer" *
    seq:
        salary = 100
        promoteTo 'manager' -> True :debug
        salary -> 150
```