The Basics - 'hello world'

Let's start with the classic 'hello world' program.

It doesn't get much simpler than outputting the text "Hello World!" on the screen.

```
print("Hello World!")
```

That is, unless you prefer not to use double quotes:

```
print('Hello World!')
```

You can use either single or double quotes to designate a string value, which represents a literal piece of text.

Also, you may have noticed: no semicolons here! Lens can intelligently figure out where semicolons would otherwise need to go in other popular languages.

However, there is no consequence for adding semicolons at the end of each line:

```
print('Hello Lens!');
```

Commenting your code

You can create a 'comment' by entering two forward slashes '//' before a line.

Alternatively, you can surround text with '/*' and '*/' to create a block comment.

```
// This is a comment, and is not evaluated as code.

/* This is also a comment, using the block notation. */
```

```
// prints a greeting message.

print('Hello World!')
```

It is a good practice to write some comments explaining the purpose of a given block of code.

In case you forget what certain parts of the code are supposed to do, you can simply read the surrounding comments.

Lens, as a first programming language

If you are unsure about which language is best to begin learning to code, Lens is for you.

Lens uses concepts and paradigms that are taught in math and science classes. This means that you can apply your prior education to the concepts of programming, thereby easing the learning process.

Furthermore, Lens can be conceptually translated into any mainstream programming language, and vice versa.

In other words, if you can write something in Lens, you will be able to write equivalent code in almost any other language. This means that by learning Lens, you will automatically have a basic knowledge of Java, JavaScript, C#, Scala, Ruby, Python, and more.

Lens, coming from other languages

Although Lens is a highly functional language, its multi-paradigm syntax is more reminiscent of object-oriented languages such as JavaScript, Python, and Scala.

You can therefore choose a highly imperative, statement-based approach, or perhaps a recursive conditional/pattern-based data flow, or maybe an enormous chain of operator combinators — or maybe a mix of all of them. Lens can be written in whatever style you're comfortable using.

Lens' goal for advanced developers is to reduce otherwise annoying or redundant operations, without sacrificing maintainability or effectiveness.

The constraint-based type system provides far more control over data flow than in dynamic-typed languages such as JavaScript and Python, while providing more versatility than static-typed languages such as Java, C#, C++, and so on.

Moreover, concepts such as optional values and argument assertions are ingrained into the language itself, allowing for greater control over a value's state at any given point.

Variables and Assignment

Keeping track of information may be done using variables.

There are two ways to create a basic variable. The first way is very simple:

```
a = 5
```

You can now refer to this variable instead of a literal number. For example:

```
a = 5

print(a) // outputs '5'
```

You can also re-assign a variable to a new value.

```
someValue = 1
```

```
print(someValue) // 1

// re-assign someValue

someValue = 2

print(someValue) // 2
```

Properly using variables will allow you to write far more useful and effective code.

You can also declare variables explicitly using the 'var' keyword.

```
var a = 5
```

This will prevent you from accidentally re-assigning a previously defined variable.

Advanced Variables

Lens contains several special assignment expressions, which allow for highly compact logical assignments.

```
var a = 1, b = 2 // comma-seperated assignments

(a, b) = (3, 4) // batch assignment

var tuple = (1, 2, 3)
```

```
tuple[0] = 4 // assigned the first index of 'tuple' to 4; result is
(4, 2, 3)


var map = {a: 1, b: 2, c: 3}


map.(a, b, c) = (6, 7, 8)
print(map) // {a: 6, b: 7, c: 8}
```

Functions and Expressions

Lens is a highly functional programming language.

This means that everything can be treated like a mathematical equation.

For instance, the linear mathematical function f(x) = 2x + 1 can be represented in Lens:

```
f(x) = 2 * x + 1
```

And more specifically:

```
def f(x: num): num = 2 * x + 1
```

This function can be used to calculate a corresponding value from the given input.

```
f(x) = 2 * x + 1
```

```
print(f(5)) // outputs '11', which is 2 * 5 + 1
```

The above function may also be written in this manner:

```
def f(x)

{

      var y = 2 * x + 1

      return y

}
```

However, functions can perform much more complex and useful tasks.

Although functions are designed to return a value, it is also possible to create a reusable block of code.

```
def action(data)

{

      print("received: " + data)

}

action("Hello Lens") // outputs "Hello Lens" to the console
```

Functions are a concise, maintainable alternative to writing repetitive lines of code.

First-Class Functions

Lens treats functions like any other object.

In other words, you can pass functions through other functions. This has some extremely powerful implications.

For instance, it is possible to prompt a function to call another function to achieve some dependent result.

```
var someString = "Demonstrative Text with potentially variable length"


// partial consumer - only invokes the specified action if the
provided condition matches 'someString'

def somePartial(condition: str > bool, action: str>) =
if(condition(someString)) action(someString)


// some hypothetical functions

def someCondition(s) = size(s) < 10

def anotherCondition(s) = true

def someAction(s) = print(s)

def anotherAction(s) = print("This one appends " + s)


// potential use cases
```

```
somePartial(someCondition, someAction) // invokes
someAction(someString), but only if the length of someString is less
than 10

somePartial(anotherCondition, anotherAction) // always invokes
anotherAction(someString)
```

However, you can also define a function without requiring a name. This is called a lambda expression, and can be designated using the arrow function:

```
somePartial((s) => s[0] == 'H', (s) => print(s + "!")) // prints the
value of 'someString' with an exclamation point, only if the first
character is 'H'

somePartial(s => true /* optional parenthesis */, => print(flatten(#))
/* anonymous parameter */) // Always prints a list of each character
in 'someString'
```

This allows for some highly compact logical structures, such as the Lens foreach function:

```
var range = 1 <> 10 // defines a range list, from 1 to 10 (inclusive)


foreach(range) n => print(n) // outputs the values 1-10

foreach(range) => print(#) // anonymous shorthand for the same
operation
```

```
foreach(range)(print) // does exactly the same thing, because the
'foreach' function actually returns a function. See 'currying' below.
```

Function Currying

Argument currying is a useful tool for simplifying otherwise complex algorithms.

It is possible to define multiple 'tiers' of parameters for a function. Essentially, the function can be called gradually, with each part defining the next.

Here is an example of how currying may be used to create a mapping function; every value in a list is transformed by the provided function.

```
def createMapper(list: any[])(mapFunction: any > any)

{

      var result = []

      var i = 0

      while(i < size(list))

      {

            result |+ mapFunction(list[i])

            i++

      }

      return result
}
```

You can then use this function to perform some highly compact operations:

```
var values =

[

    (1, 2, 3)

    (4, 5, 6)

    (7, 8, 9)

]

var result = mapFunction(values) tuple => tuple[0] // returns the first value from each tuple

print(result) // output is '[1, 4, 7]'
```

This is essentially shorthand for calling two separate functions:

```
var mapper = createMapper(values) // returns a function that takes another function as a parameter

var result = mapper(tuple => tuple[0]) // returns '[1, 4, 7]'
```

Function Parameters

Due to the Lens type system, arguments and parameters are handled differently than in other languages.

Every function is considered to have exactly one input and output. However, that input/output can potentially be undefined (the Lens Equivalent of 'null'), or a tuple containing multiple values.

Parameter Types

A basic parameter simply defines the name (and optionally the type) of an argument.

```
def example(arg) {} // any argument, which can potentially accept undefined (e.g. example())

def example(arg: val) {} // accepts any argument except undefined

def example(arg: SomeType) {} // accepts any argument which can be conformed to SomeType
```

If a type is provided, the input value may be converted in order to meet the constraint requirements. For instance:

```
def printNumber(n: num) = print(n)


print(5) // valid input

print("10") // valid input; implicit conversion from 'str' to 'int'

print("abc") // invalid input; throws an exception
```

The ability for an argument to meet the requirements of a given parameter is knows as its conformity.

Parameters can be used to decompose complex objects.

A simple example of this is utilizing tuple objects.

```
def sumTriple(a, b, c) = a + b + c



var values = (1, 4, 5)

print(sumTriple(values)) // output is '10'
```

This operation would otherwise require messy operator overloading in other languages.

Using a traditional programming paradigm, the above code would be represented like this:

```
print(sumTriple(values[0], values[1], values[2]))
```

Tuple parameters make data more understandable.

Functions that accept multiple arguments should utilize a tuple parameter. This is extremely simple:

```
def multiArgs(a, b, c) = { ... }

multiArgs(1, "str", ())
```

In some cases, it is useful to create nested tuple parameters.

```
type Vec2 = (num, num)



def addXY((x1: num, y1: num), (x2: num, y2: num)): (num, num) = (x1 +
x2, y1 + y2)



var v1 = (1, 4) as Vec2

var v2 = (-2, 2) as Vec2



addXY(v1, v2) // returns (-1, 6)

addXY((v1[0], v1[1]), (v2[0], v2[1])) // returns (-1, 6)
```

Using complex parameters significantly reduces the verbosity of otherwise relatively simple tasks.

Anonymous parameters are highly convenient.

When defining a logical operation, adding many named parameters can collude the namespace.

Because Lens is designed to be a highly functional language, parameters do not require names in some cases.

```
def namedSumInt(a: int, b: int) = a + b

def anonymousSumInt(#int, #int) = # + #

def paramlessDoubler = (#, #)


print(namedSumInt(1, 2)) // outputs '3'

print(anonymousSumInt(1, 2)) // outputs '3'

print(paramlessDoubler(5)) // outputs '(5, 5)'
```

Note that if only one parameter is specified, or none at all, the anonymous # symbol will represent the entire input value.

For more information, please view the documentation corresponding to data structures and types.

Operator Definitions

Operators let you code more efficiently, but with less maintainability.

Functions may be assigned to an operator for convenience. Due to operator precedence, this can lead to some incredibly concise algorithms.

However, operator precedence may also lead to unexpected results. Make sure to parenthesize operations where applicable.

```
// define an operator that facilitates Math.max(..)

opr(num %^ num):num = Math.max


print(1 %^ 5) // output is '5'

print(5 %^ 1) // output is '5'


// define a plus-or-minus operator using a lambda expression

opr(num +/- num) = (a: num, b: num) => (a + b, a - b)


print(1 +/- 3) // output is '(4, -2)'
```

The optimal structure of a program is a balance between operators and explicit functions. Be sure to keep this in mind when defining complex operator-based algorithms.

Operators are not ideal for creating a coherent architecture. Since they are not associated with a textual name, it is sometimes very difficult to understand what operators are supposed to do.

Therefore, the use of custom operators should be considered primarily for small-scale or placeholder implementations.

Please note that operators are not scoped, as is the case with functions, and therefore are always defined globally.

This is important when overloading operators, because in some cases an object may match an unintended operator defined somewhere unrelated to the current scope.

```
Please note that due to the enormous amount of pages explaining
operator precedence and every default operator in the language,
amounting to around 30 pages, we have left out this documentation
section. Please refer to the webpage Shelf docs for more information.
```

Numbers (num)

Numbers are the simplest and perhaps most intuitive value objects in Lens. They can be used to store data, increment counters, perform calculations, and provide an easy way to implement mathematical principles in your code.

Defining a number is extremely easy.

```
var integer = 1

var decimal = 2.5


var sum = integer + decimal

print(sum) // 3.5


while(integer < 10)

{

      integer++

}


print(integer) // 10
```

Strings (str)

A string value represents a literal piece of text.

This is the type of value you would use to create a "Hello World" program, because you are outputting actual letters instead of a stored value.

```
print("A literal string value") // output is 'A literal string value'


var someString = "abc"

print(someString) // output is 'abc'


var name = "Joe"

var greeting = "Hello, " + joe + "!"

print(greeting) // 'Hello, Joe!'
```

Booleans (bool)

There are two possible boolean values: true and false.

This is a highly useful data type for checking conditions or creating a logical pattern. For instance, you can use an 'if' statement to only run a block of code if a certain condition is met.

```
someBool = true

if(someBool)

{

     print("someBool is true!") // this will always execute

}


if(someBool == false)

{

     print("someBool is false!") // this will never execute

}


if(someBool)

{

     print("someBool is true!") // this will always execute, because
someBool is true

}

var n = 5

print(n == 5) // 'true'

print(n > 10) // 'false'
```

Undefined (void)

The 'void' data type only has one literal value: undefined.

If you wish to represent something that does not have a value, you can use 'undefined' or '()' to represent this.

```
var someValue = 5

if(someValue?)

{

      print(someValue) // '5'

}

var someVoid = ()

if(someVoid?)

{

      // never executes

}

else

{

      print(someVoid ? "some default value") // 'some default value'

}
```

List Objects

You may be familiar with list types from almost any other language. They operate in the way you might expect, by storing a sequential list of values.

Lens lists are apparently similar to JavaScript arrays. They allow for the storage of a modifiable collection of objects.

```
var list = [1, 2, 3, 4]


print(list[0]) // '1'

print(list..reverse) // '[4, 3, 2, 1]'


list |+ 5


print(list) // '[1, 2, 3, 4, 5]'


print(list is int[]) // true

print(list is int[4]) // true

print(list is int[1+]) // true

print(list is int[5+]) // false
```

Map Objects

Maps represent a set of properties, each with a key and associated value.

Similar to lists, maps are a highly useful and common programming language feature. Lens uses a syntax that is reminiscent of JavaScript's object literal notation:

```
var dependentValue = 456

var map = {

      someProp: 5

      emptyProp: ()

      listProp: [8, 16, 24, {a: 'A', b: 'B'}]

      nestProp: {

            funcProp: t: str => format("<< $t >>")

            dependentProp: dependentValue

      }

}

// Note that properties do not require commas between lines.

// This is a highly convenient syntactic improvement over JSON
notation.
```

```
print(map.someProp) // '5'

print(map.emptyProp ? "no value") // 'no value'

print(listProp[3]["b"]) // 'B'

print(list.nestProp.t("Some String")) // '<< Some String >>'

print(list[nestProp][dependentProp]) // '456'
```

Tuple Values

Tuples are the defining Lens data structure.

A tuple is essentially a group of values jointed together.

A logical example of a tuple is a vector, which contains both a magnitude and direction. Alternatively, a vector can be expressed as a group of numbers, each representing a dimension of displacement.

Lens allows for a number of highly convenient and powerful operations using tuples.

```
type UnitVector3D = (num, num, num) // tuple type: (i, j, k)

// note that the above type declaration can be shorthanded as 3(num)


var vector = (8, -12) as UnitVector2D // define a tuple value
representing a vector
```

```
// calculate vector magnitude

def mag(#Vector) = Math.sqrt(# ^ (n => n**2) ^^ opr(num + num))

print(mag(vector)) // '15'
```

Async Values

Lens provides a means of dealing with asynchronous values concisely and effectively.

In effect, it is possible to reference a value that does not exist yet, which is especially useful for loading web content or running very large operations.

Async values may be created using the 'async' keyword.

```
var a1 = async someTimelyProcess()

var a2 = async {

      sleep(10) // pause for 10 seconds

      return ("some", {complex: "value"})

}


// This example uses reverse modulus notation. Please refer to
'Operators' for more information.
```

```
a1 =>
{
      // callback; waits until a1 is completed, then executes this code with the result
      var result = #

      // do something with the result of the operation
      print(result)
}

// this line does not actually modify a2
// instead, it waits until the value exists, and then updates the property accordingly
a2[1].key = value

// creates a new async object referring to the first index of a2
var a3 = a2[0]

// waits 10 seconds, then prints the result of a2: '("some", {complex: "value"})'
// identical to a2(print), although 'print % a2' is more
understandable

print % a2


await(a1, a2, a3)((a, b, c) =>

{

     // waits for all specified async values, and then executes a
callback with all results

})
```

Conditional Expressions

If-else logic is one of the most important features in any programming language.

Lens allows for conditional expressions to be used as statements or inline, as demonstrated below.

```
if(true) print("This will always execute")

if(false) print("This will never execute")


var condition = true
```

```
if(condition)

{

     // some block of code that only executes if 'condition' is true

}


condition = false


if(!condition) print("This only executes if 'condition' is false")


var number = 5 + 2 + 3


if(number == 10) print("5 + 2 + 3 is 10")

else print("5 + 2 + 3 is not 10") // never executes


// inline conditionals - not possible in statement-based procedural
languages

number = if(condition) 5 else if(number == 10) 75 else 0


print(number) // 75; 'condition' is false and 'number' is 10
```

Loop Expressions

A common feature of procedural programming languages is a means of looping a block of code until a condition is no longer satisfied.

```
var list = [], i = 0

while(i < 5)

{

      i++

      list |+ i

}


print(list) // '[1, 2, 3, 4, 5]'
```

Match Expressions

In many languages, the simplest form of pattern matching is an enormous, verbose chain of if-else statements. This can be difficult to understand and maintain.

Alternatively, many languages contain some form of a 'switch' statement, which performs a low-level entry-point-based matching paradigm.

Lens combines both principles intos 'match' expression, which can be used to create a pattern-matching paradigm known as polymorphic dispatch as seen in languages such as XTend and Scala.

```
var dataObject = {

      someKey: "some string value"

      inner: {a: 'A', b: 'B'}

}



var result = dataObject match {

      case map: {someKey: str, inner} => "key is " + map.key + ", 
inner.a is " + map.inner.a

      case #{inner: {a, b}} => # + " doesn't match because above case 
accepted the value"

      case default => "Not an intended object: " + default

}
```

Asynchronous Expressions

The 'async' expression provides an extremely concise manner of seperating slow-running tasks into a seperate thread, so that they run parallel to the remaining code.

Please refer to the Data Types reference for more information on async return values.

```
result = async

{

      sleep(10) // wait 10 seconds

      print("This is printed 10 seconds after the message below")

      return 5

}


print("This is printed before the async method is completed")


result =>

{

      // this block of code executes after everything above has
completed

      print(#) // '5'

}
```

Try/Catch Expressions

In order to be especially useful, languages must have some way to control errors such as incorrect user input, misformatted data, and other problematic program states.

Lens provides a dynamic try-catch expression to prevent errors from undesirably crashing the entire program.

```
try

{

	// inline try-catch expressions are particularly concise

	var someValue =

		try someRiskySupplier()

		catch(e: SomeCustomException) throw "Failed to perform risky 
operation: " + e.toMessage()


	someRiskyOperation(someValue)

}

catch

{

	print("An error occurred: " + #)

}


print("This will execute even if an error occurred");
```