Hydra

Intro:

```
function main() {
    var nums, noms, odds, greetings, wg

    greetings = <-->
    nums = [1, 2]
    noms = ["Tim", "Eston", "Aaron", "Ben"]

    odds = nums.map((x){ return x * 2 - 1 })
    wg   = new WaitGroup(odds.length)

    for num in odds do
        spawn (){
            var msg  = "${noms[num]} says hello from a spawned head!"
            msg -> greetings
            wg.done()
        }
    end

    spawn (){
        wg.wait()
        close(greetings)
    }

    for greeting in greetings do
        println(greeting)
    end
}
```

Packages:

Importing From Packages:

Packages are imported using the `import` keyword. Each package can be imported seperately or a list of packages can be imported at once

There are three places a package can be imported from. The first is from the `std` namespace. This houses all of the packages in the standard library. As such, you donâ€™t need to worry about where those files are on your system. The second is from the `pkg` namespace. These files must be in a specific configurable folder and directory structure which is setup automatically if the built in package manager is used via `$ hydra install <pkg name>`. The third option is a full path to a .hy file or a relative path from the current file to the package file being imported.

```
import std::json, std::http, std::crypto //multiple imports
```

```
import std::math       //math pkg imported from stdlib
import pkg::socketio   //socketio pkg imported from configured packages folder
```

```
import ./util/utils.hy //utils package imported directly from path
```

Using Imports:

Packages may be imported in their entirety or their exported classes, functions, and variables may be specifically imported with the `from` keyword.

```
import json, os, sync from std           //multiple package imports from stdlib
import sqrt, ceil from std::math         //multiple function imports from std::math
import Socket, Client from pkg::socketio //multiple class imports from std::socketio
import Logger from ./util/utils.hy       //Logger class imported from utils package
```

If a whole package is imported, its functions, classes, and variables are accessed through the package name. If the package is imported from a relative or full path, the package will be able to be accessed through the name of the .hy file without the extension

```
import std::math
import std::sync
import ./util/utils.hy

var s = math.sqrt(4)
var wg = new sync.WaitGroup(3)
var inf = math.INFINITY
var stage_name = utils.generate_stage_name()
```

Otherwise they can be used directly.

```
import WaitGroup from std::sync
import sqrt, INFINITY from std::math
import Logger from ./util/utils.hy

var wg = new WaitGroup(3)
var s = sqrt(4)
var inf = INFINITY
var logger = new Logger()
```

Imports may also be renamed with the `as` keyword.

```
import WaitGroup from std::sync as WG
import Logger from ./util/utils.hy as L

var wg = new WG(3)
var logger = new L()
```

Creating A Package:

To export a top level class, top level function , or top level variable, use the `export` keyword. By default, exported items are reachable by the variable name they are exported with. Like `import <part> from <package> as <newname>` , export items can also be exported with specific names using the `as` keyword.

```

var variable = 123.456 //not exported

var VARIABLE = '123.456' //exported
```

```
function _Func(a, b){ //not exported
    return 15 + a + b, b - a - 11
}

function Func(a, b){ //exported
    return _Func(b, a)
}

class ClassB  //not exported
    ClassB(name){
        @name = name

        while true do
            //something...
        end
    }
end

class ClassA extends ClassB  //exported
    ClassA(){
        super('ClassA')
    }
end

//exports
export VARIABLE //when this package is imported, VARIABLE can be reached by <mypkg>.VARIABLE
export Func as funcy //when this package is imported, Func can be reached by <mypkg>.funcy
export ClassA
```

Multiple items may also be exported at once

```
export RWFile, RFile, WFile, W_ONLY //multiple exports
```

Built In Types:

Int:

Integers can be in base 10, hex, or binary form.

```
var i = 123
var h = 0xBEEF1234
var b = 0b11010110
```

Float:

Floats can only be expressed in base 10 and must have at least one number after the decimal place. Otherwise they will be treated as an Int.

```
var f = 123.0
```

String:

Strings can use single or double quotes and can be formatted via interpolation or concatenation. Literal strings can also be created delimited by three single quotes on either side. Literal strings require no escaping.

```
var str1 = 'abc'
var str2 = "123"

var interpolated = 'str1: ${str1}' //'str1: abc'
var concatenated = 'str2: ' + str2 //'str2: 123'

var literal = '''<div>What you see is what you get</div>'''
```

Regex:

Regular Expressions can be instantiated in two ways. As a literal delimited by `/` or by passing a pattern string to the Regex constructor. String interpolation can be used with either form of instantiation.

```
var reg1 = /(abc)(123)!*/
var reg2 = new Regex('(abc)(123)!*')

reg1.exec('abc123!!!') //['abc123!!!', 'abc', '123']
reg2.exec('abc123!!!') //['abc123!!!', 'abc', '123']

var name = 'hydra'
var reg3 = /${name} is\sawesome!*/
var reg4 = new Regex('${name} is\sawesome!*')

reg3.exec('hydra is awesome!!!!') //['hydra is awesome!!!!']
reg4.exec('hydra is awesome!!!!') //['hydra is awesome!!!!']
```

Function:

There are two types of functions. The first is a named, top level, function and the second is an anonymous closure. Both of these may also have their semantics changed based on whether or not they are generators. A generator function returns an instance of a generator which acts like a function with the exception that it can `yield` a value which will suspend its execution, giving the value and control back to its caller. The next time it is called it will resume execution from where it is left off.

Top level functions are functions that are at the top level of their class or package scope. Package functions may be passed around as values via their name.

```
function _foo(){ //non exported package top level function
    return bar
}

export class Foo
    function foo(){ //class top level function
        return _foo //returns another top level function...circumventing _foo not being exported
    }
end
```

Closures are anonymous functions that enclose over the values in their current scope. Syntactically, they differ from top level functions in that they do not need the `function` or `gen` keywords. To make a closure into a generator, a `*` is needed before the first parenthesis. As far as semantics go, enclosed values, with the exceptions of Channels and generator instances, are implicitly copied to discourage variable inconsistency in concurrent processing. However, variables may be

bound to parameters of the closure so that they may be referenced directly. This has implications on how to invoke a closure but we will get to that in a second.

```
var name = 'Charlie'
var whats_my_name = (){
    print(name)
}

whats_my_name() //Charlie
name = 'Chuckles'
whats_my_name() //Charlie

var name2 = 'Charlie'
var whats_my_name2 = (n){
    print(n)
}(name2) //n now bound to name...this does NOT invoke the closure

whats_my_name2() //Charlie
name2 = 'Chuckles'
whats_my_name2() //Chuckles

function get_big_string(){
    return "omg it's a generator function!!!"
}

var word_generator = *(){
    var words = get_big_string().split()
    for word in words do
        yield word
    end
}

var words = word_generator()
words() //false, 'omg'
words() //false, "it's"
words() //false, 'a'
```

As you may have noticed, the trailing parenthesis on the closure definition do not invoke it. Instead they serve as a short syntax to bind values to the parameters of a closure. This has two implications. The first, and more prevalent, one is that if you have a closure that has bound parameters and takes in new parameters on each call, you will have to place all of the bound parameters at the front of the parameter list. The second, is that if you want to immediately invoke a closure then it will have to be wrapped in parenthesis so that it can be evaluated and then invoked.

```
var name = 'Charlie'
var whats_my_name = (n, suffix){
    print(n + suffix)
}(name) //n now bound to name, suffix is not bound to anything

whats_my_name(' the king') //Charlie the king
name = 'Chuckles'
whats_my_name(' the chump') //Chuckles the chump

var num = (()){
    return 14
})() //closure invoked
```

The `@` operator lets functions access properties of the instance of the class they are defined on. In other words, `@` refers

to the instance of the class that invokes it. In a closure, `@` refers to the instance of the class that the closure was defined in.

```
class Foo

    Foo(b){
        @bar = b
    }

    function baz(){
        @bar += 2  //class variable accessed through '@'
        return bar //ERROR: 'bar' not in reachable context
    }

    function inc(amount){
        return (){
            return @bar += amount
        }
    }

end

var f = new Foo(1)
var plus2 = f.inc(2)

plus2() //3
plus2() //5

f.bar //5
```

Generators:

Generator instances are created by calling generator functions, both named and anonymous. Once created, generators can be used to generate a sequence of values. To get a value from a generator, simply call it like a function. It will return a boolean signifying if it is done followed by the variables that the generator yields. There are two built in generators that generate a range of numbers. The first is an exclusive range generator that can be instantiated by either `<start> upto <end>` or `<start> .. <end>`. The second range generator is inclusive and can be instantiated by either `<start> through <end>` or `<start> ... <end>`.

```
gen function nums(max){
  for i in 0 upto max do
    yield i
  end
}

var double = *(generator){
  for i in generator do
    yield i * 2
  end
}

var list, doubles
list = nums(3)
doubles = double(list)

doubles() //false, 0
doubles() //false, 1
```

```
doubles() //false, 2

doubles() //true, undefined
```

More often than not, generators yield values without needing any feedback. However, values can be passed back to a yielding generator by passing them as function parameters to the instance. The generator will get the values the next time it is called after it yields and it will resume execution from there.

```
gen function echoer(start1, start2){
  var msg1, msg2 = start1, start2
  while true do
    msg1, msg2 = yield msg1 + ' ' +  msg2
  end
}

var echo = echoer('hey', 'there')
echo() //'hey there'
echo('good', 'lookin') //false, 'good lookin'
echo('how', 'YOU')  //false, 'how YOU'
echo('doin', '?') //false, 'doin ?'
```

Because generators need to be consistent even when being used concurrently, they are protected by a mutex so that only one head can run any generator instance at a time.

```
var nums = 0 upto 10

spawn (){
  for i in nums do
    print(i) //0,1,3,6,7,8 ***actual order depends on scheduling***
  end
}

spawn (){
  for i in nums do
    print(i) //2,4,5,9 ***actual order depends on scheduling***
  end
}
```

Array:

Arrays grow dynamically and can hold any other type of object. This allows for nesting of hashes and other arrays as well. They can also be initialized in place.

```
var arr = []
arr.push(1) // arr now [1]
arr.push([]) // arr now [1, []]
arr.pop() // returns []...arr now [1]
arr[0] // 1

//array literal with initial values
arr = [1, [], {'key' : 'val'}, 'string', /\"(hi*)\"/, true, (a, b){ return a * b }]
```

Hash:

Hashes map one object, the key, to another, the value. While they are usually used to map strings to objects, any object can

be used as a key. If the key is a string, its value can be obtained by accessing a property of the hash with the same name as the string ie. `hash.<key>`. Hashes have a `hash.get(key)` function that takes any key and returns the value. Syntax sugar for the `get` method is similar to getting a value at an array index, ie. `<hash>[<key>]`. The `hash.put(key, val)` method and `<hash>[<key>] = <val>` can be used to put things in the hash. If you attempt to acces a key that is not in the hash it will return the `undefined` value. If you then try to access a propert on `undefined` it will throw an error. The `has_key` method returns true if the hash has the specified key. String and Number keys are evaluated by their value so that calling `get` on two different String/Number objects with the same value will return the same value from the hash. All other objects used as a key will only return the same value if they are a reference to the same instance.

```
var hash = {}

hash.a = 'a'
hash['b'] = 'b'

if hash.has_key('c') then
    print(hash.c) //not executed
else if hash.c then
    print('holy cow hash.c exists!') //not executed as undefined is falsy
else                                 //note if hash.c was any other falsy value this would also no
    print("hash.c doesn't exist") //executed
end

print(hash) //{ a : 'a', b : 'b'}
hash.remove('a')
print(hash) //{ b : 'b'}

hash.mult = (x, y){
    return x * y
}
hash.mult(2, 5) //10

hash.arr = [0, 1, 2, 3, 4, 5]
hash.arr[3] //3

var arr1 = []
var arr2 = []
var arr3 = arr1

hash[arr1] = 1
hash[arr2] = 2

hash[arr1] //1
hash[arr2] //2
hash[arr3] //1
```

Channel:

Channels act as queues that are shared between different concurrent heads. Channels can be instantiated by their unbuffered literal `<-->` or their buffered literal which takes an expression that evaluates to an integer `<-int_expr->`. While channels have `send` and `recv` functions, the `->` and `<-` operator can be used to send to or receive from a channel respectively. When using the send and recieve operators, the channel ALWAYS goes on the right.

```
var val
```

```
var c = <-2->
0 -> c //channel sent 0
val = <- c //channel received 0
print(val) //0

c.send(1) //channel sent 1
val = c.recv() //channel received 1
print(val) //1
```

Unbuffered channels block their current head until another one receives from the channel. Likewise, receives on an unbuffered channel block the current head until there is something in the channel to receive from.

```
//HEAD1
var unbuffered = <-->   //unbuffered channel literal

spawn(){
    //HEAD2
    //this head blocked until  HEAD1 puts a value in the channel
    <-unbuffered //HEAD1 unblocked
}

1 -> unbuffered //HEAD1 now blocked
```

Buffered channels do not block on sends unless they are full. Like unbuffered channels they do block on receive until there is something in the channel to receive.

```
//HEAD1
var buffered = <-10-> //buffered channel literal

from 0 upto 10 do
    1 -> buffered //10 non blocking sends...channel full
end

spawn(){ //HEAD2 (assume this doesn't start until after the next instruction)
    <-buffered //HEAD1 now unblocked
}

1 -> buffered //buffered channel is full...HEAD1 now blocked
```

Exception:

Throwing an exception is as easy as using the `throw` keyword and giving it a string. Alternatively, the Exception class can be extended so that extra information can be given to it. When `throw` is given an Exception or subclass of an Exception, it calls `to_string` on the object for the error message.

```
throw 'an exception just cuz' //'ERROR: an exception just cuz'

throw new Exception('the verbose way') //'ERROR: the verbose way'

class CustomException extends Exception
    CustomException(message){
        this.message = message
    }

    function to_string(){
        return 'Custom Error: ${this.message}'
```

```
    }
end

throw new CustomException('My custom message!!!!') //'Custom Error: My custom message!!!!'
```

Classes:

Classes have both public and private class wide and instance specific variables. Functions on the other hand are just for an instance. Instance variables and functions are referenced with the `@` operator while class variables are referenced with the `#` operator. Private variables and functions have names that start with `_`. As class variables can be accessed from any instance of the class, they are implicitly protected on reads/writes by a mutex so that they are head-safe.

```
export DEFAULT_SIZE = 10 //exported package field

class Foo

    //private class variable

    #_class_val = 7

    //public class variable

    #pub_class_val = 10

    //private methods

    function _bar(){
        return @a + @b + @DEFAULT_SIZE + #_class_val
    }

    //constructor

    Foo(a, b){
        @_a = a //priv field
        @b = b      //public field
        @size = DEFAULT_SIZE
    }

    //public methods

    function bar(chan){
        spawn (){
            @_bar() -> chan
        }
    }

    function change_class_val(val){
        #_class_val = val //implictely protected by mutex on reads/writes
    }

end

var foo = new Foo(1,2)
foo._a    //ERROR: foo, of type Foo, does not have a public field '_a'
foo.b    //2
foo.size //10
```

```
var c = <-->
foo.bar(c)
<-c //20

var foo2 = new Foo(1,2)
foo2.change_class_val(17)
foo2.bar(c)
<-c //30

foo.bar(c)
<-c //30

Foo.pub_class_val //10
Foo.pub_class_val = 25
Foo.pub_class_val //25
```

extends:

Classes can extend multiple classes. The order in which they extend those classes is important. When a subclass tries to access a superclass's property via `@<property>`, Hydra checks the first Class after the `extends` keyword and then the second and then the third until it either finds the correct property or throws an error.

super(s):

The `super` keyword can be used to access the methods and variables of a class's superclasses. A superclasses constructor can be called one of three ways. The first is to use `super` similar to a namespace ie. `super.<name of class>(/* params... */)`. The second is to use the `supers()` function. This is for when a class inherits from more than one superclass. It takes a variable number of arrays, each containing the parameters to pass to its respective superclass constructor. Note that when using this method order matters. The last way is a simplified syntax for when you either only extend one superclass or you only want to call the constructor of the first superclass after the `extends` keyword. In this case the parameters are passed to `super` as a function call. If a superclass's constructor is not called, all of the variables that the constructor would have initialized are set to the `undefined` value.

```
class A

    A(init_num){
        @num = init_num
    }

end

class B

    B(init_string){
        @string = 'In B ' + init_string
    }

end

class C extends A

    C(){
        super(10)
```

```
    }

end

class D extends A, B

    D(){
        //First way to call the constructor of multiple superclasses
        super.A(15)
        super.B('From D')
    }

end

class E extends A, B

    E(){
        //Second way...order matters here
        supers(
            [20],
            ['From E']
        )
    }

end

class F extends A, B

    F(){
        //Wrong order
        supers(
            ['From F'],
            [25]
        )
    }

end

class G extends A, B

    G(){
        super(30)
    }

end

var c = new C()
c.num //10

var d = new D()
d.num //15
d.string //'In B From D'

var e = new E()
e.num //20
e.string //'In B From E'

var f = new F()
f.num //'From F'
```

```
f.string //'In B 25'

var g = new G()
g.num //30
g.string //undefined
```

The `super` keyword can also be used to call a superclass's method directly. Again, `super` may be used similar to a namespace.

```
class Dummy
    Dummy(){}

    function error(msg){
        print('lols im not what you meant to call')
    }
end

class Base
    Base(){}

    function error(msg){
        print('Error: ${msg}')
    }
end

class Foo extends Dummy, Base
    Foo(){}

    function bar(zero){
        if zero < 0 then
            super.Base.error("wompppp...too small")
        else if zero == 0 then
            print('shwweeeeet')
        else
            @error('wompppp...too big')
        end
    }
end

var f = new Foo()
f.bar(-1) // 'Error: wompppp...too small'
f.bar(0)  // 'shwweeeeet'
f.bar(1)  // 'lols im not what you meant to call'
```

Control Structures:

For In Loop:

A for in loop takes a generator or class instance and loops through its values. If a class instance is given, for in will look for a public generator function on the object with the name 'for_in' to get its generator from. Alternatively, for in can be given a generator instance that it will call directly. The variables between the 'for' and 'in' are restricted to the loop scope and after every iteration they are passed back into the generator so that it can take into account their change if need be. Otherwise the generator can ignore the change and make the loop un-alterable once it starts. If the object after 'in' is not a generator, changing it in the loop will only change the loop if that objects 'for_in' generator function takes it into account.

```
function map_for_in(){
  var map = { one: 1, two: 2 }

  for key, val in map do
    print('key: ' + key + ' val: ' + val)
  end
}

function arr_for_in(){
  var arr = ['1', '2', '3']

  for i, val in arr do
    print('index: ' + i + ' val: ' + val)
  end
}

function str_for_in(){
  var str = 'abc123'

  for i, char in str do
    print('index: ' + i + ' char: ' + char)
  end
}

function chan_for_in(chan){
  for val in chan do
    print('recieved ' + val + ' from channel')
  end

  print('channel closed')
}
```

As long as the for in loop gets a generator instance, it doesn't matter if it came from a closure or is returned from a function call.

```
class Binary_Tree

  Binary_Tree(){
    //initialization...
  }

  gen function preorder(maxdepth=-1){
    //yield values in preorder...
  }

  function inorder(maxdepth=-1){
    return *(){
      //yield values in inorder...
    }
  }

  function postorder(maxdepth=-1){
    var copy_me = @property
    var genner = *(){
      //yield values in postorder...
      //copy property instead of giving direct access to it
    }
    return genner()
```

```
  }

end

btree = new Binary_Tree()

//add a bunch of stuff...

for val in btree.preorder() do //invoking generator function to get generator instance
  //print out a preorder representation
end

for val in btree.postorder() do //invoking function to get generator instance
  //print out a postorder representation
end

for val in (btree.inorder())() do //invoking function to get generator closure then invoking
                                  //generator closure to get generator instance
  //print out a inorder representation
end
```

While Loop:

A while loop takes a boolean literal, a comparison, a variable, or a function call as its condition and runs a block of code while the condition evaluates to a truthy value. The condition will be run before each iteration of the loop including the first one. New variables may not be created in the condition portion of the while loop.

```
function while_loop(){
  var bool = true

  while bool do
    bool = some_func_call()
  end
}

function comparison_while_loop(){
  var char = get_next_char()

  while char != EOF do
    print(char)
    char = get_next_char()
  end
}

function func_call_while_loop(){
  while still_running() do
    print('still going!')
  end
}
```

Looping Keywords:

In both for in and while loops, the `continue` and `break` key words can be used to alter the control flow. The `continue` keyword makes the loop skip executing the rest of the code in that iteration and starts the next iteration immediately. The `break` keyword stops the loop all together and passes control to the next statement outside of the loop.

```
for i in 0 upto 10 do
  if i % 2 != 0 then continue end
  if i == 8 then break end
  print(i) //0, 2, 4, 6
end

print('break just popped out of the loop')
```

Given Is Statement:

The `given is` statement goes through each one of its arms comparing the object after `given` to the expected object(s) of the arm. If the expected object(s) is a class, the arm is executed if the given object is an instance of the class. If the expected object(s) is a string or number the arm will execute if the given object has the same value. If the expected object(s) is a function the arm will execute if the given object is the same function. For class methods this is only the case when both objects are methods on the same instance. For closures, generator instances, regular expressions, and any other object they must be the same instance. If the comparison in the arm evaluates to true, the code in that arm is run and the next arm is evaluated. That is to say that all of the arms could be executed unless the `break` keyword is used to pop out of the `given is` statement. If none of the arm conditions are true, the statement will effectively do nothing unless a default block of code, which is always run if the `given is` statement gets to it, is given. A default block of code is similar to a regular arm except it comes at the end of the statement and starts with `else do` instead of `is <expected object(s)> do`.

```
function given_is(obj){
  given obj
    is String do
      string_stuff(obj)
      break
    is Array do
      array_stuff(obj)
      break
    is 0 or 2 do //allow to check for multiple cases
      number_stuff(obj)
      break
    else do
      default()
  end
}
```

Wait_For Statement:

The `wait_for` statement lets you sudo-randomly choose and communicate over an arbitrary number of sending/receiving channels. It takes channel send/receive cases and checks to see which ones would not block if executed. From the pool of executable cases, it chooses one and executes it. If none of the cases are executable it blocks until one is ready and then executes it. A default case can be added to the end which will run if no other case is executable.

```
function wait_for_either_or(in_chan1, in_chan2, out_chan){
  var recvd, clsd

  while true do
    wait_for
      either recvd, clsd <- in_chan1 then
        if clsd then
          stop()
        else
          do_something(recvd) -> out_chan
```

```
        end
      or recvd, clsd <- in_chan2 then
        if clsd then
          break
        else
          do_something2(recvd) -> out_chan
        end
      or do
        default()
      end
    end
  end
}
```

Value/Reference Semantics:

Function Parameters:

String, Int, Float, and Boolean parameters get passed by value. Parameters that are a Hash, Array, Channel, Regex, Generator instance, or any other Object get passed by reference.

```
function negate(b){
  b = !b
}

var bool = true
negate(bool)
print(bool) //true


function inc(a){
  a += a
}

var num = 1
inc(num)
print(num) //1

var string = "a"
inc(string)
print(string) //"a"

var arr = [1]
inc(arr)
print(arr) //[1, 1]

function change(h){
  h.1 = 2
}

var hash = {'1': 1}
change(hash)
print(hash) //{ '1' : 2 }

function sender(c){
  1 -> c
```

```
}

var chan = <-->
spawn sender(chan)
print(<-chan) //1

function caller(genrtr){
  print(genrtr())
}

var g = *(){
  for i in 0 upto 3 do
    yield i
  end
}

var genrtr = g()

print(genrtr()) //0
caller(genrtr)  //1
print(genrtr()) //2
```

Assignment:

Assignments have the same semantics as function parameters. When you assign an object to a variable, the variable takes the value of the object if it is a String, Int, Float, or Boolean. Otherwise, the variable references the actual object and any changes to the variable will be reflected in all other references to the object. Reassigning the variable simply makes it reference a new object or take on a new value. It does not effect the previous object it referenced.

```
class Builtins
  Builtins(){
    @str  = "a"
    @num  = 1
    @bool = true

    @hash = { '1' : 1 }
    @arr  = [1]
    @chan = <-2->
  }

  function changer(){
    return (){
      @str  = "b"
      @num  = 2
      @bool = false
      @hash = { '2' : 2 }
      @arr  = [2]
      1 -> @chan
    }
  }

end

var builtin = new Builtins()

var s, n, b, h, a, c
```

```
s = builtin.str
print(s) // "a"
print(s.upcase()) // "A"
print(builtin.str) //"a"

b = builtin.bool
print(b) // true
print(b.negate()) // false
print(builtin.bool) //true

h = builtin.hash
print(h) // { '1' : 1 }
h.2 = 2
print(builtin.hash) //{ '1' : 1, '2' : 2 }

a = builtin.arr
print(a) // [1]
a.push(2)
print(builtin.arr) // [1, 2]

c = builtin.chan
1 -> c
print(<-builtin.chan) //1
```

Binding Closure Parameters:

With the exception of channels and the `@` instance reference operator, closures copy all values that they close around. However, variables may be passed by reference by binding them to parameters. The closure will then see any changes to variables bound to its parameters by outside mutation. The syntax for binding is a parenthesised list of expressions following the closure definition.

```
//Closure closing over values, copying some and taking references of others

var num, str, bool, hash, arr, chan

num = 1
bool = true
str  = 'a'
hash = { '1' : 1 }
arr  = [1]
chan = <-2->

var clos = (){
  num  = 2
  bool = false
  str  = 'b'
  hash = { '2' : 2 }
  arr  = [2]
  1 -> chan
}

clos()

print(num) // 1
print(bool)// true
print(str) // 'a'
print(hash)// { '1' : 1 }
```

```
print(arr) // [1]
print(<-chan)// 1

//-------------------------------------
//Closure closing around, and thus copying implicitly an object

var builtins = new Builtins()

var clos2 = (){
  builtins.num  = 2
  builtins.bool = false
  builtins.str  = 'b'
  builtins.hash = { '2' : 2 }
  builtins.arr  = [2]
  1 -> builtins.chan
}

clos2()

print(builtins.num) // 1
print(builtins.bool)// true
print(builtins.str) // 'a'
print(builtins.hash)// { '1' : 1 }
print(builtins.arr) // [1]
print(<-builtins.chan)// 1

//-------------------------------------
//Closure being passed a reference

var clos3 = (b){
  b.num  = 2
  b.bool = false
  b.str  = 'b'
  b.hash = { '2' : 2 }
  b.arr  = [2]
  1 -> b.chan
}

builtins = new Builtins()

clos3(builtins)

print(builtins.num) // 2
print(builtins.bool)// false
print(builtins.str) // 'b'
print(builtins.hash)// { '2' : 2 }
print(builtins.arr) // [2]
print(<-builtins.chan)// 1

//-------------------------------------
//Closure with bound parameter

builtins = new Builtins()

var clos4 = (b){
  b.num  = 3
  b.bool = true
  b.str  = 'c'
  b.hash = { '3' : 3 }
```

```
  b.arr  = [3]
  2 -> b.chan
}(builtins)

clos4()

print(builtins.num) // 3
print(builtins.bool)// true
print(builtins.str) // 'c'
print(builtins.hash)// { '3' : 3 }
print(builtins.arr) // [3]
print(<-builtins.chan)// 2

//-------------------------------------
//Closure capturing @<property>

builtins = new Builtins()
var change = builtins.changer()

print(builtins.num) // 1
print(builtins.bool)// true
print(builtins.str) // 'a'
print(builtins.hash)// { '1' : 1 }
print(builtins.arr) // [1]

change()

print(builtins.num) // 2
print(builtins.bool)// false
print(builtins.str) // 'b'
print(builtins.hash)// { '2' : 2 }
print(builtins.arr) // [2]
print(<-builtins.chan)// 1
```