Hydra

Intro:

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

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

    odds = nums.map((x){ x * 2 - 1 })

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

    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 don'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 pkg imported directly from path
```

Using Imports:

Packages may be imported in their entirety or their exported classes, functions, and variables may be specifically imported.

```
import std.json            //json package imported from stdlib
import std.math.sqrt       //sqrt() function imported from the std.math
package
import std.sync.WaitGroup //WaitGroup class imported from the std.sync
package
import std.os.NUM_CORES    //NUM_CORES variable imported from the std.os
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 std.sync.WaitGroup
import std.math.sqrt
import std.math.INFINITY

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

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

```
import std.sync.WaitGroup as WG

var wg = new WG(3)
```

Creating A Package:

To export a top level class, top level function , or top level variable, use the `export` keyword. There are two syntaxes for this. The first lets you export at the definition of the exported item. For variables, the `export` keyword implicitely acts as a `var` keyword as well. The second, and preferred method is explicitely clumping all of your exports together at the end of the file.

First Way

```
var variable = 123.456 //not exported

export VARIABLE = '123.456' //exported

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

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

class ClassB  //not exported
    function new(name){
        this.name = name

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

export class ClassA extends ClassB  //exported
    function new(){
        super('ClassA')
    }
end
```

Preferred Way

```
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
    function new(name){
        this.name = name

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

class ClassA extends ClassB  //exported
```

```
    function new(){
        super('ClassA')
    }
end

//exports
export VARIABLE
export Func
export ClassA
```

Built In Types:

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 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. They may be passed around as values via their name.

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

class Foo
    function foo(){ //class top level function
        return _foo //returns another top level function
    }
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`, `priv`, or `gen` keywords. To make a closure into a generator, a `*` is needed before the first parenthesis. As far as semantics go, enclosed values are implicitely copied to discourage variable inconsistency in concurrent processing. However, values may be passed by reference by passing them in as parameters. 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
}

word_generator() //'omg'
word_generator() //"it's"
word_generator() //'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 prevelant 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 `this` keyword lets functions access variables in their calling context. For a top level package function, `this`, by default, refers to the package context. Note that top level package variables may also be accessed without `this`.For a class level function, `this` refers to the instance of the class that invokes it. In a closure, `this` refers to the context in which the closure was defined.

```
SIZE = 10 //exported and thus configurable package variable

priv function init(){
    do_something_with(this.SIZE) //package variable accessed through
'this'
    do_something_else_with(SIZE) //package variable directly accessed
}
```

```
class Foo

    function new(b){
        this.bar = b
    }

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

    function inc(amount){
        return (){
            return this.bar += amount
        }
    }

end

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

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

f.bar //5
```

...bind, apply, call

Array:

```
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', true, (a, b){ return a * b }]
```

Hash:

```
var hash = {}

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

if hash.has_key('c') then
    print(hash.c) //not executed
else
    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
```

String:

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

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

Int:

```
var i = 123
```

Float:

```
var f = 123.0
```

Channel:

```
var val

var c = <-->
0 -> c //channel sent 0
val = <- c //channel received 0
val //0

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

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

1 -> unbuffered //head now blocked
<-unbuffered //head now unblocked

var buffered   = <-10-> //burffered channel literal

from 0 to 8 do
    1 -> buffered //non blocking
end
```

```
1 -> buffered //head now blocked
<-buffered //head now unblocked
```

Exception:

```
throw 'an exception just cuz'

throw new Exception('the verbose way')

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

    function toString(){
        return 'Custom Error: ${this.message}'
    }
end

throw new CustomException('My custom message!!!!')
```

Classes

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

class Foo

    private //private variables

    var class_val = 7

    public //public variables

    var pub_class_val = 10

    private //private methods

    function _bar(){
        return this.a + this.b + this.DEFAULT_SIZE + class_val
    }

    public //public methods

    Foo(a, b){
        priv this.a = a //priv field
        this.b = b      //public field
        this.size = DEFAULT_SIZE
    }

    function bar(chan){
        spawn (){
            this._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: Object, 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 `this.<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 `null`.

```
class A
```

```
    A(init_num){
        this.num = init_num
    }

end

class B

    B(init_string){
        this.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 //null
```

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 do
            super.Base.error("wompppp...too small")
        else if zero == 0 then do
            print('shwweeeeet')
        else do
            this.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'
```