Io Programming Guide

Introduction

Overview

Io is a dynamic prototype-based programming language. The ideas in Io are mostly inspired by Smalltalk[1] (all values are objects), Self[2] (prototype-based), NewtonScript[3] (differential inheritance), Act1[4] (actors and futures for concurrency), Lisp[5] (code is a runtime inspectable / modifiable tree) and Lua[6] (small, embeddable).

Perspective

The focus of programming language research for the last thirty years has been to combine the expressive power of high level languages like Smalltalk and the performance of low level language like C with little attention paid to advancing expressive power itself. The result has been a series of languages which are neither as fast as C or as expressive as Smalltalk. Io's purpose is to refocus attention on expressiveness by exploring higher level dynamic programming features with greater levels of runtime flexibility and simplified programming syntax and semantics.

In Io, all values are objects (of which, anything can change at runtime, including slots, methods and inheritance), all code is made up of expressions (which are runtime inspectable and modifiable) and all expressions are made up of dynamic message sends (including assignment and control structures). Execution contexts themselves are objects and activatable objects such as methods/blocks and functions are unified into blocks with assignable scope. Concurrency is made more easily manageable through actors and implemented using coroutines for scalability.

Goals

To be a language that is:

simple

- conceptually simple and consistent
- easily embedded and extended

powerful

- highly dynamic and introspective
- highly concurrent (via coroutines and async i/o)

practical

- fast enough
- multi-platform
- unrestrictive BSD/MIT license
- comprehensive standard packages in distro

Downloading

Io distributions are available at:

```
http://iolanguage.com
```

Installing First compile the Io vm:

```
make vm
sudo make install
```

Installing Addon Dependencies

Some of Io's addons require libraries that may not be installed on your system already. To install these automatically, type either:

```
su -c "sudo make aptget"
```

or:

```
su -c "make emerge"
```

or:

```
sudo make port
```

Depending on which package installer you use. (port is for OSX)

Compiling Addons

To build, from the top folder, run:

```
make
```

Binaries will be placed in the _build/binaries subfolder. To install:

```
sudo make install
```

or, if you'd like the install to simply link to your development folder:

```
sudo make linkInstall
```

and to run all unit tests:

```
make test
```

Don't worry if some of the addons won't build unless it's a particular addon that you need. Addons are just optional libraries.

Notes

To make a particular addon, you can do:

```
make AddonName
```

After doing a pull from the source control repo, be sure to do:

```
make clean; make
```

To test just the vm:

```
make testvm
```

And to update the reference documentation (found in docs/IoReference.html) from the source code:

```
make doc
```

Binaries Io builds two executables and places them in the binaries folder. They are:

```
io_static
io
```

The io_static executable contains the vm with a minimal set of primitives all statically linked into the executable. The io executable contains just enough to load the iovm dynamically linked library and is able to dynamically load io addons when they are referenced.

Running Scripts An example of running a script:

```
io samples/misc/HelloWorld.io
```

There is no main() function or object that gets executed first in Io. Scripts are executed when compiled.

Interactive Mode Running:

```
./_build/binaries/io
```

Or, if Io is installed, running:

```
io
```

will open the Io interpreter prompt.

You can evaluate code by entering it directly. Example:

```
Io> "Hello world!" println
==> Hello world!
```

Expressions are evaluated in the context of the Lobby:

```
Io> print
[printout of lobby contents]
```

If you have a .iorc file in your home folder, it will be evaled before the interactive prompt starts.

Inspecting objects You can get a list of the slots of an object like this:

```
Io> someObject slotNames
```

To show them in sorted order:

```
Io> someObject slotNames sort
```

For a nicely formatted description of an object, the slotSummary method is handy:

```
Io> slotSummary
==>  Object_0x20c4e0:
  Lobby            = Object_0x20c4e0
  Protos           = Object_0x20bff0
  exit             = method(...)
  forward          = method(...)
```

Exploring further:

```
Io> Protos
==>  Object_0x20bff0:
  Addons           = Object_0x20c6f0
  Core             = Object_0x20c4b0

Io> Protos Addons
==>  Object_0x20c6f0:
  ReadLine         = Object_0x366a10
```

Only ReadLine is seen in the Addons since no other Addons have been loaded yet.

Inspecting a method will print a decompiled version of it:

```
Io> Lobby getSlot("forward")
==> # io/Z_Importer.io:65
method(
    Importer import(call)
)
```

doFile and doString

A script can be run from the interactive mode using the doFile method:

```
doFile("scriptName.io")
```

The evaluation context of doFile is the receiver, which in this case would be the lobby. To evaluate the script in the context of some other object, simply send the doFile message to it:

```
someObject doFile("scriptName.io")
```

The doString method can be used to evaluate a string:

```
Io> doString("1+1")
==> 2
```

And to evaluate a string in the context of a particular object:

```
someObject doString("1 + 1")
```

Command Line Arguments

Example of printing out command line arguments:

```
System args foreach(k, v, write("'", v, "'\n"))
```

launchPath

The System "launchPath" slot is set to the location of the initial source file that is executed; when the interactive prompt is started (without specifying a source file to execute), the launchPath is the current working directory:

```
System launchPath
```

Syntax

Expressions

Io has no keywords or statements. Everything is an expression composed entirely of messages, each of which is a runtime accessible object. The informal BNF description:

```
exp        ::= { message | terminator }
message    ::= symbol [arguments]
arguments  ::= "(" [exp [ { "," exp } ]] ")"
symbol     ::= identifier | number | string
terminator ::= "\n" | ";"
```

For performance reasons, String and Number literal messages have their results cached in their message objects.

Messages

Message arguments are passed as expressions and evaluated by the receiver. Selective evaluation of arguments can be used to implement control flow. Examples:

```
for(i, 1, 10, i println)
a := if(b == 0, c + 1, d)
```

In the above code, "for" and "if" are just normal messages, not special forms or keywords.

Likewise, dynamic evaluation can be used with enumeration without the need to wrap the expression in a block. Examples:

```
people select(person, person age < 30)
names := people map(person, person name)
```

Methods like map and select will typically apply the expression directly to the values if only the expression is provided:

```
people select(age < 30)
names := people map(name)
```

There is also some syntax sugar for operators (including assignment), which are handled by an Io macro executed on the expression after it is compiled into a message tree. Some sample source code:

```
Account := Object clone
Account balance := 0
Account deposit := method(amount,
    balance = balance + amount
)

account := Account clone
account deposit(10.00)
account balance println
```

Like Self[2], Io's syntax does not distinguish between accessing a slot containing a method from one containing a variable.

Operators An operator is just a message whose name contains no alphanumeric characters (other than ";", "_", '"' or ".") or is one of the following words: or, and, return. Example:

```
1 + 2
```

This just gets compiled into the normal message:

```
1 +(2)
```

Which is the form you can use if you need to do grouping:

```
1 +(2 * 4)
```

Standard operators follow C's precedence order, so:

```
1 + 2 * 3 + 4
```

Is parsed as:

```
1 +(2 *(3)) +(4)
```

User defined operators (that don't have a standard operator name) are performed left to right.

Assignment Io has three assignment operators:

operator	action
::=	Creates slot, creates setter, assigns value
:=	Creates slot, assigns value
=	Assigns value to slot if it exists, otherwise raises exception

These operators are compiled to normal messages whose methods can be overridden. For example:

source	compiles to
a ::= 1	newSlot("a", 1)
a := 1	setSlot("a", 1)
a = 1	updateSlot("a", 1)

On Locals objects, updateSlot is overridden so it will update the slot in the object in which the method was activated if the slot is not found the locals. This is done so update assignments in methods don't require self to be an explicit target.

Numbers The following are examples of valid number formats:

```
123
123.456
0.456
.456
123e-4
123e4
123.456e-7
123.456e2
```

Hex numbers are also supported (in any casing):

```
0x0
0x0F
0XeE
```

Strings Strings can be defined surrounded by a single set of double quotes with escaped quotes (and other escape characters) within.

```
s := "this is a \"test\".\nThis is only a test."
```

Or for strings with non-escaped characters and/or spanning many lines, triple quotes can be used.

```
s := """this is a "test".
This is only a test."""
```

Comments Comments of the //, /**/ and # style are supported. Examples:

```
a := b // add a comment to a line

/* comment out a group
a := 1
b := 2
*/
```

The "#" style is useful for unix scripts:

```
#!/usr/local/bin/io
```

That's it! You now know everything there is to know about Io's syntax. Control flow, objects, methods, exceptions are expressed with the syntax and semantics described above.

Objects

Overview Io's guiding design principle is simplicity and power through conceptual unification.

concept	unifies
scopable blocks	functions, methods, closures
prototypes	objects, classes, namespaces, locals
messages	operators, calls, assigns, var access

Prototypes In Io, everything is an object (including the locals storage of a block and the namespace itself) and all actions are messages (including assignment). Objects are composed of a list of key/value pairs called slots, and an internal list of objects from which it inherits called protos. A slot's key is a symbol (a unique immutable sequence) and its value can be any type of object.

clone and init

New objects are made by cloning existing ones. A clone is an empty object that has the parent in its list of protos. A new instance's init slot will be activated which gives the object a chance to initialize itself. Like NewtonScript[3], slots in Io are create-on-write.

```
me := Person clone
```

To add an instance variable or method, simply set it:

```
myDog name := "rover"
myDog sit := method("I'm sitting\n" print)
```

When an object is cloned, its "init" slot will be called if it has one.

Inheritance When an object receives a message it looks for a matching slot, if not found, the lookup continues depth first recursively in its protos. Lookup loops are detected (at runtime) and avoided. If the matching slot contains an activatable object, such as a Block or CFunction, it is activated, if it contains any other type of value it returns the value. Io has no globals and the root object in the Io namespace is called the Lobby.

Since there are no classes, there's no difference between a subclass and an instance. Here's an example of creating the equivalent of a subclass:

```
Io> Dog := Object clone
==> Object_0x4a7c0
```

The above code sets the Lobby slot "Dog" to a clone of the Object object; the protos list of this new object contains only a reference to Object, essentially indicating that a subclass of Object has been created. Instance variables and methods are inherited from the objects referenced in the protos list. If a slot is set, it creates a new slot in our object instead of changing the protos:

```
Io> Dog color := "red"
Io> Dog
==> Object_0x4a7c0:
  color := "red"
```

Multiple Inheritance

You can add any number of protos to an object's protos list. When responding to a message, the lookup mechanism does a depth first search of the proto chain.

Methods

A method is an anonymous function which, when called, creates an object to store its locals and sets the local's proto pointer and its self slot to the target of the message. The Object method method() can be used to create methods. Example:

```
method((2 + 2) print)
```

An example of using a method in an object:

```
Dog := Object clone
Dog bark := method("woof!" print)
```

The above code creates a new "subclass" of object named Dog and adds a bark slot containing a block that prints "woof!". Example of calling this method:

```
Dog bark
```

The default return value of a block is the result of the last expression.

Arguments

Methods can also be defined to take arguments. Example:

```
add := method(a, b, a + b)
```

The general form is:

```
method(<arg name 0>, <arg name 1>, ..., <do message>)
```

Blocks

A block is the same as a method except it is lexically scoped. That is, variable lookups continue in the context of where the block was created instead of the target of the message which activated the block. A block can be created using the Object method block(). Example of creating a block:

```
b := block(a, a + b)
```

Blocks vs. Methods

This is sometimes a source of confusion so it's worth explaining in detail. Both methods and blocks create an object to hold their locals when they are called. The difference is what the "proto" and "self" slots of that locals object are set to. In a method, those slots are set to the target of the message. In a block, they're set to the locals object where the block was created. So a failed variable lookup in a block's locals continue in the locals where it was created. And a failed variable lookup in a method's locals continue in the object to which the message that activated it was sent.

call and self slots

When a locals object is created, its self slot is set (to the target of the message, in the case of a method, or to the creation context, in the case of a block) and its call slot is set to a Call object that can be used to access information about the block activation:

slot	returns
call sender	locals object of caller
call message	message used to call this method/block

call activated	the activated method/block
call slotContext	context in which slot was found
call target	current object

Variable Arguments

The "call message" slot in locals can be used to access the unevaluated argument messages. Example of implementing if() within Io:

```
myif := method(
    (call sender doMessage(call message argAt(0))) ifTrue(
    call sender doMessage(call message argAt(1))) ifFalse(
    call sender doMessage(call message argAt(2)))
)

myif(foo == bar, write("true\n"), write("false\n"))
```

The doMessage() method evaluates the argument in the context of the receiver. A shorter way to express this is to use the evalArgAt() method on the call object:

```
myif := method(
    call evalArgAt(0) ifTrue(
    call evalArgAt(1)) ifFalse(
    call evalArgAt(2))
)

myif(foo == bar, write("true\n"), write("false\n"))
```

Forward If an object doesn't respond to a message, it will invoke its "forward" method if it has one. Here's an example of how to print the information related lookup that failed:

```
MyObject forward := method(
    write("sender = ", call sender, "\n")
    write("message name = ", call message name, "\n")
    args := call message argsEvaluatedIn(call sender)
    args foreach(i, v, write("arg", i, " = ", v, "\n") )
)
```

Resend Sends the current message to the receiver's protos with self as the context. Example:

```
A := Object clone
A m := method(write("in A\n"))
B := A clone
B m := method(write("in B\n"); resend)
B m
```

will print:

```
in B
in A
```

For sending other messages to the receiver's proto, super is used.

Super Sometimes it's necessary to send a message directly to a proto. Example:

```
Dog := Object clone
Dog bark := method(writeln("woof!"))

fido := Dog clone
fido bark := method(
    writeln("ruf!")
    super(bark)
)
```

Both resend and super are implemented in Io.

Introspection Using the following methods you can introspect the entire Io namespace. There are also methods for modifying any and all of these attributes at runtime.

slotNames

The slotNames method returns a list of the names of an object's slots:

```
Io> Dog slotNames
==> list("bark")
```

protos

The protos method returns a list of the objects which an object inherits from:

```
Io> Dog protos
==> list("Object")
```

getSlot

The "getSlot" method can be used to get the value of a block in a slot without activating it:

```
myMethod := Dog getSlot("bark")
```

Above, we've set the locals object's "myMethod" slot to the bark method. It's important to remember that if you then want use the myMethod without activating it, you'll need to use the getSlot method:

```
otherObject newMethod := getSlot("myMethod")
```

Here, the target of the getSlot method is the locals object.

code

The arguments and expressions of methods are open to introspection. A useful convenience method is "code", which returns a string representation of the source code of the method in a normalized form.

```
Io> method(a, a * 2) code
==> "method(a, a *(2))"
```

Control Flow

true, false and nil There are singletons for true, false and nil. nil is typically used to indicate an unset or missing value.

Comparison The comparison methods:

```
==, !=, >=, <=, >, <
```

return either the true or false. The compare() method is used to implement the comparison methods and returns -1, 0 or 1 which mean less-than, equal-to or greater-than, respectively.

if, then, else The if() method can be used in the form:

```
if(<condition>, <do message>, <else do message>)
```

Example:

```
if(a == 10, "a is 10" print)
```

The else argument is optional. The condition is considered false if the condition expression evaluates to false or nil, and true otherwise.

The result of the evaluated message is returned, so:

```
if(y < 10, x := y, x := 0)
```

is the same as:

```
x := if(y < 10, y, 0)
```

Conditions can also be used in this form:

```
if(y < 10) then(x := y) else(x := 2)
```

elseif() is supported:

```
if(y < 10) then(x := y) elseif(y == 11) then(x := 0) else(x := 2)
```

ifTrue, ifFalse Also supported are Smalltalk style ifTrue, ifFalse, ifNil and ifNonNil methods:

```
(y < 10) ifTrue(x := y) ifFalse(x := 2)
```

Notice that the condition expression must have parenthesis surrounding it.

loop The loop method can be used for "infinite" loops:

```
loop("foo" println)
```

repeat The Number repeat method can be used to repeat a loop a given number of times.

```
3 repeat("foo" print)
==> foofoofoo
```

while Arguments:

```
while(<condition>, <do message>)
```

Example:

```
a := 1
while(a < 10,
    a print
    a = a + 1
)
```

for Arguments:

```
for(<counter>, <start>, <end>, <optional step>, <do message>)
```

The start and end messages are only evaluated once, when the loop starts. Example:

```
for(a, 0, 10,
    a println
)
```

Example with a step:

```
for(x, 0, 10, 3, x println)
```

Which would print:

```
0
3
6
9
```

To reverse the order of the loop, add a negative step:

```
for(a, 10, 0, -1, a println)
```

Note: the first value will be the first value of the loop variable and the last will be the last value on the final pass through the loop. So a loop of 1 to 10 will loop 10 times and a loop of 0 to 10 will loop 11 times.

break, continue loop, repeat, while and for support the break and continue methods. Example:

```
for(i, 1, 10,
    if(i == 3, continue)
    if(i == 7, break)
    i print
)
```

Output:

```
12456
```

return Any part of a block can return immediately using the return method. Example:

```
Io> test := method(123 print; return "abc"; 456 print)
Io> test
123
==> abc
```

Internally, break, continue and return all work by setting a IoState internal variable called "stopStatus" which is monitored by the loop and message evaluation code.

Importing

The Importer proto implements Io's built-in auto importer feature. If you put each of your proto's in their own file, and give the file the same name with and ".io" extension, the Importer will automatically import that file when the proto is first referenced. The Importer's default search path is the current working directory, but can add search paths using its addSearchPath() method.

Concurrency

Coroutines Io uses coroutines (user level cooperative threads), instead of preemptive OS level threads to implement concurrency. This avoids the substantial costs (memory, system calls, locking, caching issues, etc) associated with native threads and allows Io to support a very high level of concurrency with thousands of active threads.

Scheduler The Scheduler object is responsible for resuming coroutines that are yielding. The current scheduling system uses a simple first-in-first-out policy with no priorities.

Actors An actor is an object with its own thread (in our case, its own coroutine) which it uses to process its queue of asynchronous messages. Any object in Io can be sent an asynchronous message by placing using the asyncSend() or futureSend() messages.

Examples:

```
result := self foo // synchronous
futureResult := self futureSend(foo) // async, immediately returns a Future
self asyncSend(foo) // async, immediately returns nil
```

When an object receives an asynchronous message it puts the message in its queue and, if it doesn't already have one, starts a coroutine to process the messages in its queue. Queued messages are processed sequentially in a first-in-first-out order. Control can be yielded to other coroutines by calling "yield". Example:

```
obj1 := Object clone
obj1 test := method(for(n, 1, 3, n print; yield))
obj2 := obj1 clone
obj1 asyncSend(test); obj2 asyncSend(test)
while(Scheduler yieldingCoros size > 1, yield)
```

This would print "112233". Here's a more real world example:

```
HttpServer handleRequest := method(aSocket,
    HttpRequestHandler clone asyncSend(handleRequest(aSocket))
)
```

Futures

Io's futures are transparent. That is, when the result is ready, they become the result. If a message is sent to a future (besides the two methods it implements), it waits until it turns into the result before processing the message. Transparent futures are powerful because they allow programs to minimize blocking while also freeing the programmer from managing the fine details of synchronization.

Auto Deadlock Detection

An advantage of using futures is that when a future requires a wait, it will check to see if pausing to wait for the result would cause a deadlock and if so, avoid the deadlock and raise an exception. It performs this check by traversing the list of connected futures.

Futures and the Command Line Interface

The command line will attempt to print the result of expressions evaluated in it, so if the result is a Future, it will attempt to print it and this will wait on the result of Future. Example:

```
Io> q := method(wait(1))
Io> futureSend(q)
[1-second delay]
==> nil
```

To avoid this, just make sure the Future isn't the result. Example:

```
Io> futureSend(q); nil
[no delay]
==> nil
```

Yield

An object will automatically yield between processing each of its asynchronous messages. The yield method only needs to be called if a yield is required during an asynchronous message execution.

Pause and Resume

It's also possible to pause and resume an object. See the concurrency methods of the Object primitive for details and related methods.

Exceptions

Raise An exception can be raised by calling raise() on an exception proto.

```
Exception raise("generic foo exception")
```

Try and Catch To catch an exception, the try() method of the Object proto is used. try() will catch any exceptions that occur within it and return the caught exception or nil if no exception is caught.

```
e := try(<doMessage>)
```

To catch a particular exception, the Exception catch() method can be used. Example:

```
e := try(
    // ...
)

e catch(Exception,
    writeln(e coroutine backtraceString)
)
```

The first argument to catch indicates which types of exceptions will be caught. catch() returns the exception if it doesn't match and nil if it does.

Pass To re-raise an exception caught by try(), use the pass method. This is useful to pass the exception up to the next outer exception handler, usually after all catches failed to match the type of the current exception:

```
e := try(
    // ...
)

e catch(Error,
    // ...
) catch(Exception,
    // ...
) pass
```

Custom Exceptions Custom exception types can be implemented by simply cloning an existing Exception type:

```
MyErrorType := Error clone
```

Primitives

Primitives are objects built into Io whose methods are typically implemented in C and store some hidden data in their instances. For example, the Number primitive has a double precision floating point number as its hidden data and its methods that do arithmetic operations are C functions. All Io primitives inherit from the Object prototype and are mutable. That is, their methods can be changed. The reference docs contain more info on primitives.

This document is not meant as a reference manual, but an overview of the base primitives and bindings is provided here to give the user a jump start and a feel for what is available and where to look in the reference documentation for further details.

Object

The ? Operator

Sometimes it's desirable to conditionally call a method only if it exists (to avoid raising an exception). Example:

```
if(obj getSlot("foo"), obj foo)
```

Putting a "?" before a message has the same effect:

```
obj ?foo
```

List

A List is an array of references and supports all the standard array manipulation and enumeration methods. Examples:

Create an empty list:

```
a := List clone
```

Create a list of arbitrary objects using the list() method:

```
a := list(33, "a")
```

Append an item:

```
a append("b")
==> list(33, "a", "b")
```

Get the list size:

```
a size
==> 3
```

Get the item at a given index (List indexes begin at zero):

```
a at(1)
==> "a"
```

Note: List indexes begin at zero and nil is returned if the accessed index doesn't exist.

Set the item at a given index:

```
a atPut(2, "foo")
==> list(33, "a", "foo", "b")

a atPut(6, "Fred")
```

```
==> Exception: index out of bounds
```

Remove an item at a given index:

```
a remove("foo")
==> list(33, "a", "b")
```

Inserting an item at a given index:

```
a atInsert(2, "foo")
==> list(33, "a", "foo", "56")
```

foreach

The foreach, map and select methods can be used in three forms:

```
Io> a := list(65, 21, 122)
```

In the first form, the first argument is used as an index variable, the second as a value variable and the 3rd as the expression to evaluate for each value.

```
Io> a foreach(i, v, write(i, ":", v, ", "))
==> 0:65, 1:21, 2:122,
```

The second form removes the index argument:

```
Io> a foreach(v, v println)
==> 65
21
122
```

The third form removes the value argument and simply sends the expression as a message to each value:

```
Io> a foreach(println)
==> 65
21
122
```

map and select

Io's map and select (known as filter in some other languages) methods allow arbitrary expressions as the map/select predicates.

```
Io> numbers := list(1, 2, 3, 4, 5, 6)

Io> numbers select(isOdd)
==> list(1, 3, 5)

Io> numbers select(x, x isOdd)
==> list(1, 3, 5)

Io> numbers select(i, x, x isOdd)
==> list(1, 3, 5)

Io> numbers map(x, x*2)
==> list(2, 4, 6, 8, 10, 12)

Io> numbers map(i, x, x+i)
==> list(1, 3, 5, 7, 9, 11)

Io> numbers map(*3)
==> list(3, 6, 9, 12, 15, 18)
```

The map and select methods return new lists. To do the same operations in-place, you can use selectInPlace() and mapInPlace() methods.

Sequence In Io, an immutable Sequence is called a Symbol and a mutable Sequence is the equivalent of a Buffer or String. Literal strings(ones that appear in source code surrounded by quotes) are Symbols. Mutable operations cannot be performed on Symbols, but one can make mutable copy of a Symbol calling its asMutable method and then perform the mutation operations on the copy. Common string operations Getting the length of a string:

```
"abc" size
==> 3
```

Checking if a string contains a substring:

```
"apples" containsSeq("ppl")
==> true
```

Getting the character (byte) at position N:

```
"Kavi" at(1)
==> 97
```

Slicing:

```
"Kirikuro" slice(0, 2)
==> "Ki"

"Kirikuro" slice(-2)  # NOT: slice(-2, 0)!
==> "ro"

Io> "Kirikuro" slice(0, -2)
# "Kiriku"
```

Stripping whitespace:

```
"  abc  " asMutable strip
==> "abc"

"  abc  " asMutable lstrip
==> "abc  "

"  abc  " asMutable rstrip
==> "  abc"
```

Converting to upper/lowercase:

```
"Kavi" asUppercase
==> "KAVI"
"Kavi" asLowercase
==> "kavi"
```

Splitting a string:

```
"the quick brown fox" split
==> list("the", "quick", "brown", "fox")
```

Splitting by others character is possible as well.

```
"a few good men" split("e")
==> list("a f", "w good m", "n")
```

Converting to number:

```
"13" asNumber
==> 13

"a13" asNumber
==> nil
```

String interpolation:

```
name := "Fred"
==> Fred
"My name is #{name}" interpolate
==> My name is Fred
```

Interpolate will eval anything with #{} as Io code in the local context. The code may include loops or anything else but needs to return an object that responds to asString.

Ranges A range is a container containing a start and an end point, and instructions on how to get from the start, to the end. Using Ranges is often convenient when creating large lists of sequential data as they can be easily converted to lists, or as a replacement for the for() method.

The Range protocol

Each object that can be used in Ranges needs to implement a "nextInSequence" method which takes a single optional argument (the number of items to skip in the sequence of objects), and return the next item after that skip value. The default skip value is 1. The skip value of 0 is undefined. An example:

```
Number nextInSequence := method(skipVal,
    if(skipVal isNil, skipVal = 1)
    self + skipVal
)
```

With this method on Number (it's already there in the standard libraries), you can then use Numbers in Ranges, as demonstrated below:

```
1 to(5) foreach(v, v println)
```

The above will print 1 through 5, each on its own line.

File The methods openForAppending, openForReading, or openForUpdating are used for opening files. To erase an existing file before opening a new open, the remove method can be used. Example:

```
f := File with("foo.txt)
f remove
f openForUpdating
f write("hello world!")
f close
```

Directory Creating a directory object:

```
dir := Directory with("/Users/steve/")
```

Get a list of file objects for all the files in a directory:

```
files := dir files
==> list(File_0x820c40, File_0x820c40, ...)
```

Get a list of both the file and directory objects in a directory:

```
items := Directory items
==> list(Directory_0x8446b0, File_0x820c40, ...)

items at(4) name
==> DarkSide-0.0.1 # a directory name
```

Setting a Directory object to a certain directory and using it:

```
root := Directory clone setPath("c:/")
==> Directory_0x8637b8

root fileNames
==> list("AUTOEXEC.BAT", "boot.ini", "CONFIG.SYS", ...)
```

Testing for existence:

```
Directory clone setPath("q:/") exists
==> false
```

Getthing the current working directory:

```
Directory currentWorkingDirectory
==> "/cygdrive/c/lang/IoFull-Cygwin-2006-04-20"
```

Date Creating a new date instance:

```
d := Date clone
```

Setting it to the current date/time:

```
d now
```

Getting the date/time as a number, in seconds:

```
Date now asNumber
==> 1147198509.417114

Date now asNumber
==> 1147198512.33313
```

Getting individual parts of a Date object:

```
d := Date now
==> 2006-05-09 21:53:03 EST

d
==> 2006-05-09 21:53:03 EST

d year
==> 2006

d month
==> 5

d day
==> 9

d hour
==> 21

d minute
==> 53

d second
==> 3.747125
```

Find how long it takes to execute some code:

```
Date cpuSecondsToRun(100000 repeat(1+1))
==> 0.02
```

Networking All of Io's networking is done with asynchronous sockets underneath, but operations like reading and writing to a socket appear to be synchronous since the calling coroutine is unscheduled until the socket has completed the operation, or a timeout occurs. Note that you'll need to first reference the associated addon in order to cause it to load before using its objects. In these examples, you'll have to reference "Socket" to get the Socket addon to load first.

Creating a URL object:

```
url := URL with("http://example.com/")
```

Fetching an URL:

```
data := url fetch
```

Streaming a URL to a file:

```
url streamTo(File with("out.txt"))
```

A simple whois client:

```
whois := method(host,
    socket := Socket clone \
                setHostName("rs.internic.net") setPort(43)
    socket connect streamWrite(host, "\n")
    while(socket streamReadNextChunk, nil)
    return socket readBuffer
)
```

A minimal web server:

```
WebRequest := Object clone do(
    handleSocket := method(aSocket,
        aSocket streamReadNextChunk
        request := aSocket readBuffer \
                        betweenSeq("GET ", " HTTP")
        f := File with(request)
        if(f exists,
                        f streamTo(aSocket)
                ,
                        aSocket streamWrite("not found")
                )
        aSocket close
    )
)

WebServer := Server clone do(
    setPort(8000)
    handleSocket := method(aSocket,
        WebRequest clone asyncSend(handleSocket(aSocket))
    )
)

WebServer start
```

XML Using the XML parser to find the links in a web page:

```
SGML // reference this to load the SGML addon
xml := URL with("http://www.yahoo.com/") fetch asXML
links := xml elementsWithName("a") map(attributes at("href"))
```

Vector Io's Vectors are built on its Sequence primitive and are defined as:

```
Vector := Sequence clone setItemType("float32")
```

The Sequence primitive supports SIMD acceleration on a number of float32 operations. Currently these include add, subtract, multiple and divide but in the future can be extended to support most math, logic and string manipulation related operations. Here's a small example:

```
iters := 1000
size := 1024
ops := iters * size

v1 := Vector clone setSize(size) rangeFill
v2 := Vector clone setSize(size) rangeFill

dt := Date secondsToRun(
    iters repeat(v1 *= v2)
)

writeln((ops/(dt*1000000000)) asString(1, 3), " GFLOPS")
```

Which when run on 2Ghz Mac Laptop, outputs:

```
1.255 GFLOPS
```

A similar bit of C code (without SIMD acceleration) outputs:

```
0.479 GFLOPS
```

So for this example, Io is about three times faster than plain C.

Unicode

Sequences

In Io, symbols, strings, and vectors are unified into a single Sequence prototype which is an array of any available hardware data type such as:

```
uint8, uint16, uint32, uint64
int8, int16, int32, int64
float32, float64
```

Encodings

Also, a Sequence has a encoding attribute, which can be:

```
number, ascii, ucs2, ucs4, utf8
```

UCS-2 and UCS-4 are the fixed character width versions of UTF-16 and UTF-32, respectively. A String is just a Sequence with a text encoding, a Symbol is an immutable String and a Vector is a Sequence with a number encoding.

UTF encodings are assumed to be big endian.

Except for input and output, all strings should be kept in a fixed character width encoding. This design allows for a simpler implementation, code sharing between vector and string ops, fast index-based access, and SIMD acceleration of Sequence operations. All Sequence methods will do automatic type conversions as needed.

Source Code

Io source files are assumed to be in UTF8 (of which ASCII is a subset). When a source file is read, its symbols and strings are stored in Sequences in their minimal fixed character width encoding. Examples:

```
Io> "hello" encoding
==> ascii

Io> "π" encoding
==> ucs2

Io> "∞" encoding
==> ucs2
```

We can also inspect the internal representation:

```
Io> "π" itemType
==> uint16

Io> "π" itemSize
==> 2
```

Conversion

The Sequence object has a number of conversion methods:

```
asUTF8
asUCS2
asUCS4
```

Embedding

Conventions

Io's C code is written using object oriented style conventions where structures are treated as objects and functions as methods. Familiarity with these may help make the embedding APIs easier to understand.

Structures

Member names are words that begin with a lower case character with successive words each having their first character upper cased. Acronyms are capitalized. Structure names are words with their first character capitalized. Example:

```
typdef struct
{
    char *firstName;
    char *lastName;
    char *address;
} Person;
```

Functions

Function names begin with the name of structure they operate on followed by an underscore and the method name. Each structure has a new and free function.

Example:

```
List *List_new(void);
void List_free(List *self);
```

All methods (except new) have the structure (the "object") as the first argument the variable is named "self". Method names are in keyword format. That is, for each argument, the method name has a description followed by an underscore. The casing of the descriptions follow that of structure member names.

Examples:

```
int List_count(List *self); // no argument
void List_add_(List *self, void *item); // one argument
void Dictionary_key_value_(Dictionary *self,
        char *key, char *value);
```

File Names

Each structure has its own separate .h and .c files. The names of the files are the same as the name of the structure. These files contain all the functions(methods) that operate on the given structure.

IoState

An IoState can be thought of as an instance of an Io "virtual machine", although "virtual machine" is a less appropriate term because it implies a particular type of implementation.

Multiple states

Io is multi-state, meaning that it is designed to support multiple state instances within the same process. These instances are isolated and share no memory so they can be safely accessed simultaneously by different os threads, though a given state should only be accessed by one os thread at a time.

Creating a state

Here's a simple example of creating a state, evaluating a string in it, and freeing the state:

```
#include "IoState.h"

int main(int argc, const char *argv[])
{
    IoState *self = IoState_new();
    IoState_init(self);
    IoState_doCString_(self, "writeln(\"hello world!\"");
    IoState_free(self);
    return 0;
}
```

Values

We can also get return values and look at their types and print them:

```
IoObject *v = IoState_doCString_(self, someString);
char *name = IoObject_name(v);
printf("return type is a '%s', name);
IoObject_print(v);
```

Checking value types

There are some macro short cuts to help with quick type checks:

```
if (ISNUMBER(v))
{
    printf("result is the number %f", IoNumber_asFloat(v));
}
else if(ISSEQ(v))
{
    printf("result is the string %s", IoSeq_asCString(v));
}
else if(ISLIST(v))
{
    printf("result is a list with %i elements",
                IoList_rawSize(v));
}
```

Note that return values are always proper Io objects (as all values are objects in Io). You can find the C level methods (functions like IoList_rawSize()) for these objects in the header files in the folder Io/libs/iovm/source.

Bindings

Documentation on how to write bindings/addons forthcoming..

Appendix

Grammar

messages

```
expression ::= { message | sctpad }
message ::= [wcpad] symbol [scpad] [arguments]
arguments ::= Open [argument [ { Comma argument } ]] Close
argument ::= [wcpad] expression [wcpad]
```

symbols

```
symbol ::= Identifier | number | Operator | quote
Identifier ::= { letter | digit | "_" }
Operator ::= { ":" | "." | "'" | "~" | "!" | "@" | "$" |
"%" | "^" | "&" | "*" | "-" | "+" | "/" | "=" | "{" | "}" |
"[" | "]" | "|" | "\" | "<" | ">" | "?" }
```

quotes

```
quote ::= MonoQuote | TriQuote
MonoQuote ::= """ [ "\"" | not(""")] """
TriQuote ::= """"" [ not(""""")] """""
```

spans

```
Terminator ::= { [separator] ";" | "\n" | "\r" [separator] }
separator ::= { " " | "\f" | "\t" | "\v" }
whitespace ::= { " " | "\f" | "\r" | "\t" | "\v" | "\n" }
sctpad ::= { separator | Comment | Terminator }
scpad ::= { separator | Comment }
wcpad ::= { whitespace | Comment }
```

comments

```
Comment ::= slashStarComment | slashSlashComment | poundComment
slashStarComment ::= "/*" [not("*/")] "*/"
slashSlashComment ::= "//" [not("\n")] "\n"
poundComment ::= "#" [not("\n")] "\n"
```

numbers

```
number ::= HexNumber | Decimal
HexNumber ::= "0" anyCase("x") { [ digit | hexLetter ] }
hexLetter ::= "a" | "b" | "c" | "d" | "e" | "f"
Decimal ::= digits | "." digits |
        digits "." digits ["e" [-] digits]
```

characters

```
Comma ::= ","
Open ::= "(" | "[" | "{"
Close ::= ")" | "]" | "}"
letter ::= "a" ... "z" | "A" ... "Z"
digit ::= "0" ... "9"
digits ::= { digit }
```

The uppercase words above designate elements the lexer treats as tokens.

Credits

Io is the product of all the talented folks who taken the time and interest to make a contribution. The complete list of contributors is difficult to keep track of, but some of the recent major contributors include; Jonathan Wright, Jeremy Tregunna, Mike Austin, Chris Double, Rich Collins, Oliver Ansaldi, James Burgess, Baptist Heyman, Ken Kerahone, Christian Thater, Brian Mitchell, Zachary Bir and many more. The mailing list archives, repo inventory and release history are probably the best sources for a more complete record of individual contributions.

References

1 Goldberg, A et al.
Smalltalk-80: The Language and Its Implementation
Addison-Wesley, 1983

2 Ungar, D and Smith,
RB. Self: The Power of Simplicity
OOPSLA, 1987

3 Smith, W.
Class-based NewtonScript Programming
PIE Developers magazine, Jan 1994

4 Lieberman
H. Concurrent Object-Oriented Programming in Act 1
MIT AI Lab, 1987

5 McCarthy, J et al.
LISP I programmer's manual
MIT Press, 1960

6 Ierusalimschy, R, et al.
Lua: an extensible extension language
John Wiley & Sons, 1996

License