An Introduction to Retro

Getting Started

Choosing a VM

Retro runs on a virtual machine. This has been implemented in many languages, and allows easy portability to most platforms.

The table below lists the current implementations, and the features they support. For most users, we recommend using *C*, *Python*, or *Ruby*, as these are feature complete, and can be setup quickly and easily.

Language	File	A	B	C	D	E	F	G	Building
Assembly	retro.s	x	x	x	x	x	x	x	as retro.s -o retro ld retro.o -o retro
C	retro.c	x	x	x	x	x	x	x	gcc retro.c -o retro
C#	retro.cs	x	x		x	x	x	x	gmcs retro.cs
F#	retro.fsx	x	x	x	x	x	x	x	
Forth	retro.fs	x	x	x		x	x	x	
Go	gonga/	x	x	x	x	x	x	x	cd gonga && make
Lisp	retro.lisp	x				x	x	x	
Java	retro.java	x				x	x	x	javac retro.java
Lua	retro.lua	x				x	x	x	
Perl	retro.pl	x				x	x	x	
Python	retro.py	x	x	x	x	x	x	x	
Ruby	retro.rb	x	x	x	x	x	x	x	

A. save image

B. "include", "needs"

C. file i/o

D. query host environment

E. get current time

F. passes core tests

G. passes vocabulary tests

The Image File

Once you have selected (and built, if necessary) a VM, you will need to put it and the *retroImage* into a directory. You should then be able to start the VM and interact with Retro.

The Library

If you are using the Assembly, C, F#, Forth, Go, Python, or Ruby VM implementations, you can also copy or symlink the *library* directory into the same directory as your VM and *retroImage*.

This is optional, but copying it over is recommended as it simplifies loading libraries and handling dependencies.

Basic Interactions

When you start Retro, you should see something like the following:

```
Retro 11.1 (1234567890)

ok
```

At this point you are at the *listener*, which reads and processes your input. You are now set to begin exploring Retro.

Normally Retro will process input as soon as whitespace is encountered [1]. This limits editing options [2], but serves to simplify the listener significantly.

To exit, run **bye**:

```
bye
```

Exploring the Language

Names And Numbers

At a fundamental level, the Retro language consists of whitespace delimited tokens representing either names or numbers.

The *listener* will attempt to look up tokens in the *dictionary*. If found, the information in the *dictionary header* is used to carry out the actions specified in the name's *definition*.

If a token can't be found in the dictionary, Retro tries to convert it to a number. If successful, the number is pushed to the *data stack.* If not, an error is reported.

Retro permits names to contain any characters other than space, tab, cr, and lf. Names are *case sensitive*, so the following are three *different* names from Retro's perspective:

```
foo Foo FOO
```

The Compiler

To create new functions, you use the compiler. This is generally started by using **:** (pronounced *colon*). A simple example:

```
: foo 1 2 + putn ;
```

Breaking this apart:

```
: foo
```

This creates a new function named *foo* and starts the compiler. A **:** should always be followed by the name of the function to create.

```
1
```

Normally this would push a *1* to the data stack. However, since the compiler is active, the listener will compile the code needed to push a *1* to the stack into the definition instead.

```
2
```

And again, but compile a *2* instead of a *1*.

```
+
```

Since + is a normal function, the listener compiles a call to it rather than calling it immediately.

```
putn
```

putn is a function that takes a number from the stack and displays it. When encountered in a defintion, the compiler will lay down a call to it and continue.

```
;
```

Functions are terminated with a **;**. This is a special case as **;** is a *compiler macro*, and is *called at compile time*, but *ignored when the compiler is not active.*

Hyperstatic Global Environment

This now brings up an interesting subpoint. Retro provides a *hyper-static global environment.* This can be difficult to explain, so let's take a quick look at how it works:

```
: scale ( x-y ) a @ * ;
a ?
1000 variable: a
: scale ( x-y ) a @ * ;
3 scale putn
>>> 3000
100 a !
3 scale putn
>>> 300
5 variable: a
3 scale putn
>>> 300
a @ putn
>>> 5
```

Output is marked with **>>>**.

Note that we create two variables with the same name (*a*). The definition for *scale* still refers to the old variable, even though we can no longer directly manipulate it.

In a hyper-static global environment, functions continue to refer to the variables and earlier functions that existed when they were defined. If you create a new variable or function with the same name as an existing one, it only affects future code.

Classes

Getting back to function creation, it's time for a clarification: in Retro, the listener is unaware of how to handle a dictionary entry and has no concept of the difference between compiling and interpreting.

The actual work is handled by something we call *class handlers*.

Each dictionary header contains a variety of information:

Offset	Description
0	link to previous
1	class handler
2	xt
3+	name of function

When a token is found, the listener pushes the contents of the xt field and the class handler field to the stack, then calls the **withClass** function. This then calls the *class handler* function, which does something with the *xt* (pointer to the actual compiled code or data).

So, when you enter:

```
1 2 +
```

What actually happens is this:

1. The listener tries to find *1* in the dictionary. This fails, so *1* is pushed to the stack, and the *.data* class handler is pushed to the stack. *withClass* then passes control to *.data*.
2. The *.data* class looks at the *compiler* variable, sees that it's off, and then leaves the *1* on the stack.
3. This is repeated for the *2*.
4. When **+** is encountered, it is found to exist in the dictionary. The *xt* is pushed to the stack, and the *.word* class handler is pushed. Then *withClass* is called.
5. *withClass* passes control to *.word*, which checks *compiler*, sees that it is off, and then calls the *xt* corresponding to the definition of **+**.

When you create a definition, the flow is altered slightly:

1. The listener tries to find *1* in the dictionary. This fails, so *1* is pushed to the stack, and the *.data* class handler is pushed to the stack. *withClass* then passes control to *.data*.
2. The *.data* class looks at the *compiler* variable, sees that it's on, and lays down the code needed to push *1* to the stack.
3. This is repeated for the *2*.
4. When + is encountered, it is found to exist in the dictionary. The *xt* is pushed to the stack, and the *.word* class handler is pushed. Then *withClass* is called.
5. *withClass* passes control to *.word*, which checks *compiler*, sees that it is on, so compiles the necessary code to call the *xt* corresponding to the definition of +.

This model differs from Forth (and most other languages) in that the listener is kept out of the loop. All actions are handled by the function classes. A useful side effect is that additional classes can be created at any time, and assigned to any named functions or data structures.

The following classes are defined by default:

Function	Description
.word	This is the class handler for normal functions. If the *compiler* is off, it executes the function passed to it. If the *compiler* is on, it compiles a call to the function.
.compiler	This class handler is used for functions that act as compile-time macros. The function pointer is executed if the *compiler* is on. If off, it ignores pointer.

.primitive	Used for a small set of functions that can map directly to Ngaro instructions. This acts the same as *.word*, but inlines the machine code at compile time rather than lay down a call.
.macro	Used for general macros. Functions with this class are always executed.
.data	This is used for data structures. If *compiler* is off, it leaves the pointer on the stack. If the *compiler* is on this compiles the value into another function.
.parse	Special class used for *parsing prefixes*. Acts the same as *.macro*

By default, colon definitions are given a class of *.word*, and entries made by **create**, **variable**, and **constant** get a class of *.data*. To assign the *.macro* class or the *.compiler* class, use either **immediate** or **compile-only** after the **;**.

Data Structures

You can create named data structures using **create**, **variable**, **variable:**, **constant**, and **elements**.

Constants

These are the simplest data structure. The *xt* is set to a value, which is either left on the stack or compiled into a definition.

```
100 constant ONE-HUNDRED
```

By convention, constants in Retro should have names in all uppercase.

Variables

A variable is a named pointer to a memory location holding a value that may change over time. Retro provides two ways to create a variable:

```
variable a
```

The first, using **variable**, creates a name and allocates one cell for storage. The memory is initialized to zero.

```
10 variable: b
```

The second, **variable:**, takes a value from the stack, and creates a name, allocates one cell for storage, and then initializes it to the value specified. This is cleaner than doing:

```
variable a
10 a !
```

Custom Structures

You can also create custom data structures by creating a name, and allocating space yourself. For instance:

```
create test
  10 , 20 , 30 ,
```

This would create a data structure named *test*, with three values, initialized to 10, 20, and 30. The values would be stored in consecutive memory locations. If you want to allocate a buffer, you could use **allot** here:

```
create buffer
  2048 allot
```

The use of **allot** reserves space, and initializes the space to zero.

Elements

Elements are a hybrid between variables and custom data structures. They create a series of names that point to consecutive cells in memory.

```
3 elements a b c

100 a !
200 b !
300 c !

a @+ putn
>>> 100
@+ putn
>>> 200
@ putn
>>> 300
```

Strings

In addition to the basic data structures above, Retro also provides support for string data.

Creating a string simply requires wrapping text with quotation marks:

```
"this is a string"
"  this string has leading and trailing spaces  "
```

When creating strings, Retro uses a floating, rotating buffer for temporary strings. Strings created in a definition are considered permanent.

You can obtain the length of a string using either **getLength** or **withLength**:

```
"this is a string" getLength
"this is also a string" withLength
```

getLength will consume the string pointer, while **withLength** preserves it.

Comparisons

Strings can be compared using **compare**:

```
"test 1"  "test 2"  compare  putn
>>> 0
"test"  "test"  compare  putn
>>> -1
```

The comparisons are case sensitive.

Searching

For a Substring

Substrings can be located using **^strings'search**. This will return a pointer to the location of the substring or a flag of 0 if the substring is not found.

```
"this is a long string"
"a long" ^strings'search
.s puts
```

For a Character

Searching for specific characters in a string is done using **^strings'findChar**. This will return a pointer to the string starting with the character, or a flag if 0 if the character is not found.

```
"this is a string"
'a ^strings'findChar
.s puts
```

Extracting a Substring

Retro provides three functions for splitting strings.

The first, **^strings'getSubset**, takes a string, a starting offset, and a length. It then returns a new string based on the provided values.

```
"this is a string"
5 8 ^strings'getSubset
.s puts
```

The other two are **^strings'splitAtChar** and **^strings'splitAtChar:**. The first form takes a string and character from the stack and returns two strings. The second takes a string and parses for a character.

```
"This is a test. So is this" '. ^strings'splitAtChar  puts puts
"This is a test. So is this" ^strings'splitAtChar: .  puts puts
```

Trim Whitespace

Leading whitespace can be removed with **^strings'trimLeft** and trailing whitespace with **^strings'trimRight**.

```
: foo
  cr "    apples"   ^strings'trimLeft puts
     "are good!   " ^strings'trimRight puts
  100 putn ;
foo
```

Append and Prepend

To append strings, use **^string'append**. This consumes two strings, returning a new string starting with the first and ending with the second.

```
"hello,"  " world!" ^strings'append puts
```

A varient exists for placing the second string first. This is **^strings'prepend**.

```
: sayHelloTo ( $- ) "hello, " ^strings'prepend puts cr ;
"world" sayHelloTo
```

Case Conversion

To convert a string to uppercase, use **^strings'toUpper**.

```
"hello" ^strings'toUpper puts
```

To convert a string to lowercase, use **^strings'toLower**.

```
"Hello Again" ^strings'toLower puts
```

Reversal

To reverse the order of the text in a string, use **^strings'reverse**.

```
"hello, world!" ^strings'reverse puts
```

Implementation Notes

Strings in Retro are null-terminated. They are stored in the image memory. E.g., assuming a starting address of 12345 and a string of "hello", it will look like this in memory:

```
12345 h
12346 e
12347 l
12348 l
12349 o
12350 0
```

You can pass pointers to a string on the stack.

Prefixes

Before going further, let's consider the use of prefixes in Retro. The earlier examples involving variables used @ and **!** (for *fetch* and *store*) to access and modify values. Retro allows these actions to be bound to a name more tightly:

```
variable a
variable b

100 !a
@a !b
```

This would be functionally the same as:

```
variable a
variable b

100 a !
```

```
a @ b !
```

You can mix these models freely, or just use what you prefer. I personally find that the prefixes make things slightly clearer, but most of them are completely optional [3].

Other prefixes include:

Function	Description
&	Return a pointer to a function or data structure
+	Add TOS to the value stored in a variable
-	Subtract TOS from the value stored in a variable
@	Return the value stored in a variable
!	Store TOS into a variable
^	Access a function or data element in a vocabulary
'	Return ASCII code for following character
$	Parse number as hexadecimal
#	Parse number as decimal
%	Parse number as binary
"	Parse and return a string

Quotes

In addition to colon definitions, Retro also provides support for anonymous, nestable blocks of code called *quotes*. These can be created inside definitions, or at the interpreter.

Quotes are essential in Retro as they form the basis for conditional execution, loops, and other forms of flow control.

To create a quote, simply wrap a sequence of code in square brackets:

```
[ 1 2 + putn ]
```

To make use of quotes, Retro provides *combinators*.

Combinators

A combinator is a function that consumes functions as input. These are divided into three primary types: compositional, execution flow, and data flow [4].

Compositional

A compositional combinator takes elements from the stack and returns a new quote.

cons takes two values from the stack and returns a new quote that will push these values to the stack when executed.

```
1 2 cons
```

Functionally, this is the same as:

```
[ 1 2 ]
```

take pulls a value and a quote from the stack and returns a new quote executing the specified quote before pushing the value to the stack.

```
4 [ 1+ ] take
```

Functionally this is the same as:

```
[ 1+ 4 ]
```

curry takes a value and a quote and returns a new quote applying the specified quote to the specified value. As an example,

```
: acc ( n- )  here swap , [ dup ++ @ ] curry ;
```

This would create an accumulator function, which takes an initial value and returns a quote that will increase the accumulator by 1 each time it is invoked. It will also return the latest value. So:

```
10 acc
dup do putn
dup do putn
dup do putn
```

Execution Flow

Combinators of this type execute other functions.

Fundamental

do takes a quote and executes it immediately.

```
[ 1 putn ] do
&words do
```

Conditionals

Retro provides four combinators for use with conditional execution of quotes. These are **if**, **ifTrue**, **ifFalse**, and **when**.

if takes a flag and two quotes from the stack. If the flag is *true*, the first quote is executed. If false, the second quote is executed.

```
-1 [ "true\n" puts ] [ "false\n" puts ] if
 0 [ "true\n" puts ] [ "false\n" puts ] if
```

ifTrue takes a flag and one quote from the stack. If the flag is true, the quote is executed. If false, the quote is discarded.

```
-1 [ "true\n" puts ] ifTrue
 0 [ "true\n" puts ] ifTrue
```

ifFalse takes a flag and one quote from the stack. If the flag is false, the quote is executed. If true, the quote is discarded.

```
-1 [ "false\n" puts ] ifFalse
 0 [ "false\n" puts ] ifFalse
```

when takes a number and two quotes. The number is duplicated, and the first quote is executed. If it returns true, the second quote is executed. If false, the second quote is discarded.

Additionally, if the first quote is true, **when** will exit the calling function, but if false, it returns to the calling function.

```
: test ( n- )
  [ 1 = ] [ drop "Yes\n" puts ] when
  [ 2 = ] [ drop "No\n" puts  ] when
  drop "No idea\n" puts ;
```

Looping

Several combinators are available for handling various looping constructs.

while takes a quote from the stack and executes it repeatedly as long as the quote returns a *true* flag on the stack. This flag must be well formed and equal *-1*.

```
10 [ dup putn space 1- dup 0 <> ] while
```

times takes a count and quote from the stack. The quote will be executed the number of times specified. No indexes are pushed to the stack.

```
1 10 [ dup putn space 1+ ] times
```

The **iter** and **iterd** varients act similarly, but do push indexes to the stacks. **iter** counts up from 0, and **iterd** counts downward to 1.

```
10 [ putn space ] iter
10 [ putn space ] iterd
```

Data Flow

These combinators exist to simplify stack usage in various circumstances.

Preserving

Preserving combinators execute code while preserving portions of the data stack.

dip takes a value and a quote, moves the value off the main stack temporarily, executes the quote, and then restores the value.

```
10 20 [ 1+ ] dip
```

Would yield the following on the stack:

```
11 20
```

This is functionally the same as doing:

```
10 20 push 1+ pop
```

sip is similar to **dip**, but leaves a copy of the original value on the stack during execution of the quote. So:

```
10 [ 1+ ] sip
```

Leaves us with:

```
11 10
```

This is functionally the same as:

```
10 dup 1+ swap
```

Cleave

Cleave combinators apply multiple quotations to a single value or set of values.

bi takes a value and two quotes, it then applies each quote to a copy of the value.

```
100 [ 1+ ] [ 1- ] bi
```

tri takes a value and three quotes. It then applies each quote to a copy of the value.

```
100 [ 1+ ] [ 1- ] [ dup * ] tri
```

Spread

Spread combinators apply multiple quotations to multiple values. The asterisk suffixed to these function names signifies that they are spread combinators.

bi* takes two values and two quotes. It applies the first quote to the first value and the second quote to the second value.

```
1 2 [ 1+ ] [ 2 * ] bi*
```

tri* takes three values and three quotes, applying the first quote to the first value, the second quote to the second value, and the third quote to the third value.

```
1 2 3 [ 1+ ] [ 2 * ] [ 1- ] tri*
```

Apply

Apply combinators apply a single quotation to multiple values. The at sign suffixed to these function names signifies that they are apply combinators.

bi@ takes two values and a quote. It then applies the quote to each value.

```
1 2 [ 1+ ] bi@
```

tri@ takes three values and a quote. It then applies the quote to each value.

```
1 2 3 [ 1+ ] tri@
```

each@ takes a pointer, a quote, and a type constant. It then applies the quote to each value in the pointer. In the case of a linear buffer, it also takes a length.

```
( arrays )
create a 3 , ( 3 items ) 1 , 2 , 3 ,
a [ @ putn space ] ^types'ARRAY each@

( buffer )
"hello" withLength [ @ putc ] ^types'BUFFER each@

( string )
"HELLO" [ @ putc ] ^types'STRING each@

( linked list )
last [ @ d->name puts space ] ^types'LIST each@
```

Conditionals

Retro has a number of functions for implementing comparisons and conditional execution of code.

Comparisons

Function	Stack	Description
=	ab-f	compare a == b
>	ab-f	compare a > b
<	ab-f	compare a < b
>=	ab-f	compare a >= b
<=	ab-f	compare a <= b
<>	ab-f	compare a <> b
compare	$$-f	compare two strings
if;	f-	if flag is true, exit function
0;	n-?	if n <> 0, leave n on stack and continue if n = 0, drop n and exit function
if	fqq-	Execute one of two quotes depending on value of flag
ifTrue	fq-	Execute quote if flag is not zero
ifFalse	fq-	Execute quote if flag is zero
when	nqq-n	Execute second quote if first quote returns true. Exits caller if second quote is executed.

Namespaces

Sometimes you will want to hide some functions or data structures from the main dictionary. This is done by wrapping the code in question in double curly braces:

```
23 constant foo

{{
  1 constant ONE
  2 constant TWO
  : foo ONE TWO + ;
  foo
```

```
}}

foo  ( refers to the first foo; the second foo is now hidden )
```

When the closing braces are encountered, the headers for the functions following the opening braces are hidden.

If you want to hide some functions, but reveal others, you can add **---reveal---** into the mix:

```
{{
  1 constant ONE
  2 constant TWO
---reveal---
  : foo ONE TWO + ;
}}
```

At this point, *foo* would be visible, but the constants would be hidden.

Vocabularies

Vocabularies allow grouping of related functions and data, and selectively exposing them. Active vocabularies are searched before the main dictionary and the order for searching is configurable at runtime.

Creation

```
chain: name'
  ...functions...
;chain
```

Vocabulary names should generally be lowercase, and should end with a single apostrophe.

Exposing and Hiding

Use **with** to add a vocabulary to the search order. The most recently exposed vocabularies are searched first, with the global dictionary searched last.

```
with console'
```

The most recent vocabulary can be closed using **without**.

```
without
```

You can also close all vocabularies using **global**.

```
global
```

As a simplification, you can reset the search order and load a series of vocabularies using **with|**:

```
with| console' files' strings' |
```

Direct Access

It is possible to directly use functions and variables in a vocabulary using the **^** prefix.

```
^vocabulary'function
```

As an example:

```
: redWords ^console'red words ^console'normal ;
```

This is recommended over exposing a full vocabulary as it keeps the exposed functions down, helping to avoid naming conflicts.

Vectored Execution

One of the design goals of Retro is flexibility. And one way this is achieved is by allowing existing colon definitions to be replaced with new code. We call this *revectoring* a definition.

Function	Stack	Description
:is	aa-	Assign the function (a2) to act as (a1)
:devector	a-	Restore the original definition of (a)
is	a"-	Parse for a function name and set it to act as (a)
devector	"-	Parse for a function name and restore the original definition

Example:

```
: foo ( -n ) 100 ;
: bar ( -  ) foo 10 + putn ;
bar
>>> 110
[ 20 ] is foo
bar
>>> 30
devector foo
bar
>>> 110
```

This technique is used to allow for fixing of buggy code in existing images and adding new functionality.

Input and Output

Getting away from the quotes, combinators, compiler, and other bits, let's take a short look at input and output options.

Console

At the listener level, Retro provides a few basic functions for reading and displaying data.

Function	Stack	Description
getc	-c	Read a single keypress
accept	c-	Read a string into the text input buffer
getToken	-$	Read a whitespace delimited token and return a pointer
putc	c-	Display a single character
puts	$-	Display a string
clear	-	Clear the display

space	-	Display a blank space
cr	-	Move cursor to the next line

The **puts** function handles a number of escape sequences to allow for formatted output.

Code	Use
n	newline
[	ASCII 27, followed by [
\	Display a
'	Display a "
%d	Display a number from the stack (decimal)
%o	Display a number from the stack (octal)
%x	Display a number from the stack (hexadecimal)
%s	Display a string from the stack
%c	Display a character from the stack
%%	Display a %

As an example:

```
3 1 2 "%d + %d = %d\n" puts
>>> 2 + 1 = 3

: I'm ( "- )
  getToken "\nHello %s, welcome back.\n" puts ;

I'm crc
>>> Hello crc, welcome back
```

Footnotes

---

1 With some VM implementations, Retro will not process the input until the enter key is pressed. This is system-level buffering, and is not the standard Retro behavior. There are external tools included with Retro to alter the behavior to match the standard.

2 You can not use Retro with tools like *rlwrap*, and editing is limited to use of backspace. The arrow keys are not supported by Retro.

3 The exceptions here would be the *&* prefix for obtaining a pointer inside a definition and the *"* prefix for parsing strings. All of the others can be worked around or ignored easily.

4 The terminology and some function names are borrowed from the Factor language.