fOOrth ~ FNF

A Developer's Notebook

Where:

fOOrth **is an object oriented slant on the classic FORTH language,**

and

FNF is Not FORTH

or

just plain fOOrth because, unlike GNU, FNF is hard to say.

By ***Peter Camilleri***

Draft: August 26, 2014

Introduction to the Developer's Notebook:

The fOOrth language is very much a work in progress in regards its goals, design, syntax, compiler, interpreter, and implementation. It is simply not known at this time what directions these aspects will take or what rabbit holes will have to be plumbed as the version 1_00_00 code is being sought. As such, this text is very much a stream of consciousness look at a changing and dynamic system. I have made an effort to try to make it as coherent as possible and where I have failed (or will fail) to do so, I apologize.

What's with the name fOOrth?

The name of programming language fOOrth is a malamanteau[1] of forth and the “OO” of object oriented programming systems. It is short, easy to pronounce, slightly witty, somewhat pithy and a unique opportunity to describe an actual word as being a malamanteau. It is a quaint sounding, but incorrect mash up of both the words and concepts involved.

So in what way is fOOrth a malapropism? Quite simply, fOOrth is NOT forth, while the sound of the name clearly draws one to that conclusion. Further, while FNF is not forth, fOOrth is also not FNF, since fOOrth is easy to pronounce while FNF is not.

Why does fOOrth even exist?

In truth, no sane rational justification can be offered for the creation of fOOrth. The fOOrth project exists on many levels and for many reasons. Some of these are:

1. The fOOrth language project was a study case for learning the Ruby programming language.
2. The fOOrth language project was an examination of the creation of efficient lexical scanners and parsers for domain specific languages.
3. The old school languages FORTH and Smalltalk both possess an conceptual purity that modern languages like Java, C#, or even Ruby seem to lack. The fOOrth language is a thought experiment born out of this fascinating tradition of combining FORTH with object oriented design principles.
4. I was bored and creating the fOOrth language system provided a creative outlet and escape from the dull routine of my days. Now if only it could pay some bills!

The thing to bear in my is that all of these reasons testify to at least some of the truth, but none, or in fact, not even all of them encompass the whole reason. I suppose that in the end, fOOrth existed in my mind, it was there all along, I just had to go out and reveal it.

1 Malamanteau, see XKCD 739 at http://xkcd.com/739/ as well as Wikipedia::Malamanteaux

The evolution of fOOrth.

Like most efforts of this nature, the fOOrth project went through many developmental stages. This section deals with the growth in fOOrth leading up to version 00_04_32. Progress after that point is discussed in other sections of this document.

The earliest code bases, labeled -1.N.N did not run code of any kind at all. At this level, development focused on the mechanics of reading source text in Ruby and parsing them in a way useful for a forth like language.

Development versions have all been of the form 0.N.N. 0.1.N through 00_04_31 represent a process of language enhancement and refinement that brought more and more of the power of FORTH into existence, enhanced and simplified by the underlying flexibility of Ruby.

As of version 00_04_31, the (Ruby) code generated for a simple high level word might look like this:

```
>: double dup + ;

colon lambda {|vm| vm.call('dup'); vm.call('+'); }
Added 'double' to the dictionary.
```

Note that the code is translated into a lambda with a single parameter, 'vm', short for virtual machine. The high level words are implemented as a hash look-up by that virtual machine.

At that time the language vocabulary was:

```
!              "              (            ! )"             )abort
)abort"        )all_words     )current_voc   )debug         )finish
)path_words    )quit          )start         )system        )version
)voc_path      )vocs          )words         *              */
+              +compile       +compile"      +execute       +execute"
+loop        ! ,              ,asm         ! ,asm"        ! -
-execute       -execute"      -i           ! -j           ! .
."             .c             .l             .r             /
0<             0<=            0<=>           0<>            0=
0>             0>=            1+             1-             2+
:              ;            ! ;empty       ! ;immediate  ! <
<=             <=>            <>             <builds      ! =
>              >=             >r             ?abort         ?abort"
?dup           @              [compile]    ! _<<            _<<"
_abort         _av"           _re"           _rv"           _se"
_vm            _wv"           again        ! alias:         and
begin        ! cj             const:         cr             csd
debug          do           ! does>        ! double         drop
dsd            dup            else         ! emit           execute
false          fwd:           i            ! if           ! invert
j            ! lj             load           load"          load_string
load_string"   loop         ! max            min            minus
mod            nip            nop            not            or
```

```
over          pick          r             r>            rdrop
ref:          repeat      ! rj            rot           rpick
space         spaces        split         swap          test
that        ! then        ! this        ! to_f          to_i
to_r          to_s          true          tuck          until       !
var:          version       voc_create"   voc_delete"   voc_dismiss"
voc_employ"   voc_rebuild   while       ! xor           {           !
}
```

A fairly large set of words including the basics of stack manipulation, control structures, function definition and even more advanced concepts like <builds does> meta-definitions and hierarchical vocabularies.

Still, all told, at this level, the fOOrth system was at best, a messy hybrid of FORTH and Ruby with no clear purpose or direction. It was clear that progress could be made by one of two major paths of progress:

1. Continue adding actual object oriented programming features to the existing fOOrth system. This creates a language that is a hybrid of a Ruby object oriented emulation of a threaded programming system and programming system with an object oriented programming layered on top of a non-object oriented base.
2. Start with a clean slate, realizing the key language components as native fOOrth object oriented entities, interacting directly and not through clumsy intermediaries. The programming model would appear to be fOOrth through and through.

It must be stressed that the lofty goals of the second approach carry with them substantial risk. Up to this point, very little had been achieved in the way of object oriented programming and the fOOrth dialect of these concepts was sketchy and not proven out. The very composition of such an object was not well understood let alone ready to be built upon.

Neither of these options seems very appealing. On reflection, it would seem that the most prudent course of action would be to prototype these new object oriented design concepts in the version 00_05_XX versions of fOOrth and then start with a clean slate in version 00_06_XX or what ever version it makes sense to do so.

No, I changed my mind. 00_05_XX will be a clean slate. 00_04_32 will be kept around as a reference and for testing ideas, but the time has come to start fresh!

The fOOrth Parser:

The Parser class used to process fOOrth source code into chunks or words using rules that are quite different from those used by FORTH. The following are the fOOrth syntax rules.

```
          <word> ::= {<white_space>}* <comment_start>|<word_seq>

 <comment_start> ::= '('|'//'

      <word_seq> ::= {<word_char>}+ (('"' <string body>)|<end_char>)
```

```
    <word_char> ::= not_any('"'|<white_space>|<end of input>)

  <string body> ::= {<string char>|<escape seq>}+ <string_close>
     <end_char> ::= (<white_space>|<end of input>)
  <string char> ::= not_any('\'|'"'|<ctrl_char>|<end of line>)
<string_close> ::= (<end of line>|'"')
   <escape seq> ::= ('\"'|'\\'|'\'<end of line>)

  <white_space> ::= (' '|<ctrl_char>|<end of line>)
    <ctrl_char> ::= ($00..$1F|$7F)
  <end of line> ::= condition(no more text on the current line)
<end of input> ::= condition(no more text left from the source)
```

Where:

```
      <A> Stuff  ≡  Token A is defined as Stuff.
            A B  ≡  Token A followed by token B.
            A|B  ≡  Token A or token B.
           {A}*  ≡  Zero or more repetitions of token A.
           {A}+  ≡  One or more repetitions of token A.
          A..B   ≡  A range from A to B, inclusive.
Not_any(A | B)   ≡  Anything except token A or token B.
  Condition(X)   ≡  Describes a condition in the source rather than
                    actual character data.
```

The design and specification of an objected oriented parser are still ongoing.

The Design of the fOOrth Language:

Method Types:

There are a number of design issues centered around the design of methods in the fOOrth language system. Some of these are:

Explicit vs Implicit Receivers:

The first issue is that fOOrth has two kinds of methods:

- Conventional methods sent to an explicitly named object receiver.
- Special methods sent implicitly to the virtual machine.

This ignores the whole issue of short forms for send messages to the object itself. If the following code is written:

```
"Testing" to_s
```

is to_s sent to the string testing or self. Maybe a leading ~ can be used as a shorthand so that this code:

```
"Testing" ~to_s
```

would be equivalent to:

```
"Testing" this to_s
```

Immediate methods:

Another issue is that of immediate methods. Immediate methods execute immediately, even when the virtual machine is in compile or deferred modes. This can work in FORTH because the compiler is able to look up the actual word definition and determine if it is an immediate word. With methods, it's not that simple. The receiver is not known until the code is run, by which time the immediacy is long gone. There are some possible solutions to this dilemma:

- Make immediate mode words only accessible to direct, virtual machine words.
- Tag the immediacy to the method selector rather than the method. This would force all methods with the same name to have the same immediacy setting.
- Provide some way to hint to the compiler as to the type of the receiver so that the immediacy of the method may determined (guessed at?).

Method Selectors:

Are method selectors to be legitimate objects in the their own right? It would seem prudent to do so. Selectors already possess a type (method vs vm) and characteristics (normal vs immediate) as well as a name, so a selector class seems to make sense. This would allow selectors to manipulated as data within fOOrth to augment meta-programming capabilities.

The fOOrth virtual machine:

The design of the virtual machine is a reflection of the language paradigm that it executes. This section reviews where the fOOrth virtual machine has been and what future directions it may yet take. As such, this section is very much a stream of consciousness look at a changing and dynamic system.

Basic Virtual Machine Components:

The fOOrth virtual machine is in many regards, a fairly conventional FORTH virtual machine. It consists of many recognizable entities derived from FORTH such as:

- A last in, first out data stack.
- A last in, first out return or control stack.
- A hashed dictionary of code, data, and meta definitions as well as mechanisms for searching and updating this dictionary.
- A buffer for code being interpreted or compiled

While similar on the surface, these structures are implemented in Ruby. Whereas the FORTH stack consists of a region of 16 bit integers, the fOOrth stack is an array of objects that can contain an infinite variety of data types without concern for static data typing. This object based freedom is best illustrated by comparing FORTH and fOOrth code snippets:

FORTH	fOOrth
`2 4 + .  ==> 6`	`2 4 + .  ==> 6`
`2.0 4.0 F+ F.  ==> 6.0`	`2.0 4.0 + .  ==> 6.0`
Not sure of the FORTH code needed here.	`"Hello" " World" + .` `==> "Hello World"`

Note the uniformity and simplicity of the fOOrth code. These are benefits of the polymorphism and late binding that separates fOOrth from its primitive ancestor. Those features serve to greatly simplify many programming tasks and is a major benefit of the hybrid programming model.

Executing Code:

This section examines the semantics of the implementation of procedural and object oriented segments of code.

Procedural Code; Function Calls:

All early versions of fOOrth implement code by two mechanisms, that reflect two basic modes:

In Execute mode fOOrth words are being parsed and executed one at a time. For example, if the user types:

```
4 5 + .
```

at the command prompt, each token “4”, “5”, “+” and “.” are executed one at a time by Ruby code. For the non-numeric literals, this involves looking up the words in the system dictionary and then calling these words as follows:

```
word.call(self)
```

where “word” is the looked up entry from the dictionary and self is a reference to the virtual machine that is executing the code.

By contrast, in compiled and deferred modes, code is accumulated in a string that when evaluated yields a lambda block that contains the code required. In compiled mode, this code block is placed into the dictionary as a new definition. Deferred mode is required for those cases where execution does not proceed one word at a time. Typically this involves control structures and the like. In the case of deferred execution, the block is executed when it

contains a completed control structure.

The nature of this code block has evolved over the various versions. The following shows a summary of this evolution:

```
: double dup + ;
```

Early code:

```
lambda {|vm| vm.dictionary('dup').call(vm); #etc... }
```

Intermediate code:

```
lambda {|vm, dict| dict('dup').call(vm);    #etc... }
```

Current code:

```
lambda {|vm| vm.call('dup');                #etc... }
```

The current code is the most compact, but it may not be the best code because it simply hides the complexity in multiple layers of code. From a performance point of view, the intermediate code may have been the fastest since it factors out the fetch of the dictionary while avoiding unnecessary method invocation overhead. In addition, it may be improved since not have been necessary to pass the dictionary explicitly. It may have simply been enough to capture it in the scope of the lambda block. In effect:

```
dict = @dictionary
colon lambda {|vm| dict('dup').call(vm);          #etc... }
```

Note that while the dictionary may be captured in this way, the virtual machine reference cannot. This is due to that fact that while the dictionary is shared by all threads, each thread has it own instance of the virtual machine. Thus for each thread, the block must execute against the specific virtual machine of that thread and not a global instance of one.

Further it must be noted that this idea has not been tested at this time! The reason that this has not been pursued is quite simply, the idea may already be obsolete. In order to add true object oriented programming concepts to fOOrth, the nature and location of the dictionary must undergo some very major changes. These are discussed in the next section.

Now, consider the following method of the Ruby BasicObject class:

instance_exec obj.instance_exec(<args>∗) { \| args \| . . . } → other_obj

Executes the block with self set to obj, passing args as parameters to the block. It returns that last value of the block.

This would allow the self in the lambda block to be the correct virtual machine. While current code invokes code blocks by saying (in fOOrth_word.rb):

```
def call(vm)
  @block.call(vm)
end
```

The code could instead look like:

```
def call(vm)
  vm.instance_exec(&@block)
end
```

This brings a number of benefits. It means that the virtual machine need not be an explicit argument to the blocks. It also means that instance variables of that virtual machine are accessible in the lambda block. The next evolution of the code could look like:

```
lambda { instance_exec(&(@cache['dup']);     #etc... }
```

Now, conceive that this would be combined with the just-in-time compiler described in the section Object Oriented Dictionary to produce the following:

```
lambda { dup;                                #etc... }
```

WOW! Now THAT is awesome! Well, method names will have to be mangle so we get:

```
lambda { f34;                                #etc... }
```

Procedural Code; Counted Loops:

Consider the following small bit of code:

```
10 0 do i . loop
```

The implementation of this code has changed substantially over the course of the development of fOOrth. Note that the code has been reflowed for clarity and readability.

The first cut produced this elegant code:

```
vm.push(10); vm.push(0);
lambda {|vm|
  ((vm.pop)...(vm.pop)).each {|i| vm.push(i); vm.call('.'); };
}
```

As nice as this code is, there are a few problems with it:

1. There seems to be no way to adjust the index value as required by the FORTH “+loop” control word.
2. While the index of the current loop is available, there seems to be no way to access the index of any “outer” loop.
3. Counting in reverse is possible by using reverse_each instead of each, but this is a resource intensive operation.

As a result, in the latest version, this code has morphed to the following:

```
vm.push(10); vm.push(0);
```

```
lambda {|vm|
  vm.vm_do {|i,j| vm.push(i[1]); vm.call('.'); i[1] += 1};
}
```

It is noteworthy that the i and j parameters are actually arrays of data copied from the control stack. Each contains four elements:

i[0] contains the symbol :vm_do which is a syntax marker for the loop.

i[1] contains the current index of the loop. Thus the i[1] += 1 at the end of the loop is incrementing this value.

i[2] contains the end value of the loop minus 1.

i[3] contains the start value + the end value minus 1.

The loop proceeds as long as i[1] ≤ i[2]. The i[3] entry exists to allow the loop to run backward. To support this feature the -i word translates to:

```
vm.push(i[3]-i[1]);
```

Object Oriented Code:

In addition, the existing code generation schemes are not capable of advancing from object based programming to object oriented programming. In general, there are two issues here:

1. How will the concept of Object receives Method be expressed in the syntax of the fOOrth language? This aspect will not be furthered in this section.
2. How will the semantics of Object receives Method be expressed in the virtual machine implementation? There are a number options available here, listed below.

Assumptions: The the receiving object is already on the data stack.

```
vm.pop.call(vm, 'Method');
```

This is a good first crack at it. The “call” method of all fOOrth objects would do a hierarchical search, starting locally and moving up through super classes looking for the method.

On some level it would be really neat if we could instead generate:

```
vm.pop.Method(vm);
```

The problem here being one of name space incompatibility, clashes, and collisions between fOOrth and Ruby. That problem however can be solved by postulating the existence of a method name mapping hash that transforms fOOrth method names to harmless but unique Ruby names. Thus the code might look like:

```
vm.pop.fOOrth_3419(vm);
```

Even better, the Ruby method_missing hook could be used as a simple just-in-time (JIT) compiler, searching the method hierarchy and creating methods on the fly. Much more work

on this matter is required before it can be fully evaluated.

Back-filling the concepts in Procedural Code; Function Calls: we get:

```
pop.fOOrth_3419;
```

The structure of the dictionary:

In early versions of fOOrth, the dictionary was a simple hash shared by all the virtual machines. The index of the hash was a string that was the name of the word, and the value of the hash was the actual word as a Ruby object.

This had the draw back that there existed only one name space, and that collisions were resolved by simply clobbering old definitions with new ones with no way of going back.

Classical FORTH had this idea of vocabularies. Like most things in FORTH it was a very simple, and yet hard to use implementation of name spaces. It should come as no surprise that fOOrth created something somewhat more elaborate and easy to use.

Vocabularies now consist of hashes, one per vocabulary. As before, they are a mapping of names to words. A list of all these called vocabularies is a hash that maps the names of the vocabularies to their matching dictionaries. That is a hash of hashes. The vocabulary called 'System' is initially created by default. These parts so far are system global.

Each virtual machine adds three more components.

vocabulary_list – an array of references to vocabularies in the order they are used.
current_vocabulary – a reference to the vocabulary where new definitions are added.
cache – hash containing a summary of all the vocabularies in the vocabulary_list.

Object Oriented Dictionary:

Object oriented programming requires that methods be tied to objects or classes rather than the system as a whole or a vocabulary. Thus as elaborate as the above system is, it is not adequate for the requirements of fOOrth.

The question that arises is “Should it be swept away and replaced by a purely object oriented approach?” At first blush, the answer would seem to be yes. This is somewhat based on the appeal of the concept of “pure”. When we think of something being pure, we envision that it is not contaminated with other things or substances. That it stays true to the original. And yet, while it is far beyond the scope of this document, history is littered with tragic examples of “purity” gone very wrong[2]! Hybrid cars exist for a reason. Compromise makes good engineering sense.

Consider the often cited bromide that in Ruby, “Everything is an object.” It's simply not true.

2 See http://en.wikipedia.org/wiki/Salem_witch_trials and http://en.wikipedia.org/wiki/Racial_purity

This is illustrated in the following snippet:

```
if a > b
  puts "a is greater"
end
```

In this code, “a” and “b” are objects as is the result of “a > b”. Further puts is a method sent to the implied object “self” and the string literal “a is greater“. So what's left. The “if” and “end” are not part of this object oriented dance. A check of the Ruby boolean classes TrueClass and FalseClass revels no “if” method anywhere. The “if” statement is purely a figment of the compiler's imagination.

So, is there a counter-example that is object oriented? Yes, in Smalltalk:

```
(a > b) ifTrue: [Transcript show: 'a is greater']
```

In this example, the ifTrue: is a message sent to the boolean receiver computed by a > b and the block of code that follows is an argument to the ifTrue: message.

Then again, while the object oriented underpinnings here may be a bit too extreme, the message names ifTrue and ifFalse do seem to have some attractive features. Consider:

```
10 0 do i 1 and ifTrue ."Odd" ifFalse ."Even" ifEnd cr doEnd
```

It does seem a lot clearer than the traditional:

```
10 0 do i 1 and if ."Odd" else ."Even" then cr loop
```

Still should it be ifEnd and doEnd or the more usual endif and enddo? Choices, choices...

However: none of this has anything to do with the method dictionary structures. All objects will be instances of Object (called XfOOrthObject in the underlying Ruby code). It will define very basic facilities:

1. A reference to the class of the object.
2. A method missing hook.

The actual method dictionary will reside in the Class (XfOOrthClass) object. This will add:

1. A name string.
2. A reference to a parent class.
3. A hash of instance methods. Maps instance method name => lambda
4. A hash of class methods. Maps class method name => lambda

All fOOrth classes will be instances of XfOOrthClass in Ruby. The key to this all will be the method missing hook. It will serve as a just in time compiler for fOOrth.

Some concept code, a first take follows:

```
def method_missing(name, *args, &block)
  defn = @fOOrth_class.look_up(name)
```

```
    if defn
      define_method(name, &defn)
      send(name, *args, &block)
    else
      @fOOrth_class.does_not_understand(name)
    end
  end
```

I'm not sure, but since define_method is a private method I may need to use:

```
self.class.send(:define_method, name, &defn)
```

There... maybe...

The thing is, these methods get added to the Ruby class of that object. That means that to avoid collisions in the method names, each fOOrth class will need to be a unique sublcass of the parent XfOOrthObject or XfOOrthClass class. This should not be too hard given how easy it is to create classes:

new Class.new(super_class=Object) h { . . . } i → cls

Creates a new anonymous (unnamed) class with the given superclass (or Object if no parameter is given). If called with a block, that block is used as the body of the class. Within the block, self is set to the class instance.

So it looks like we'll need a mapping hash for classes too. It maps name => class
or...
It could be just a symbol in the symbol hash and classes just another type of symbol!
And... vm methods could also be other types of symbols.. called fOOrth Words!

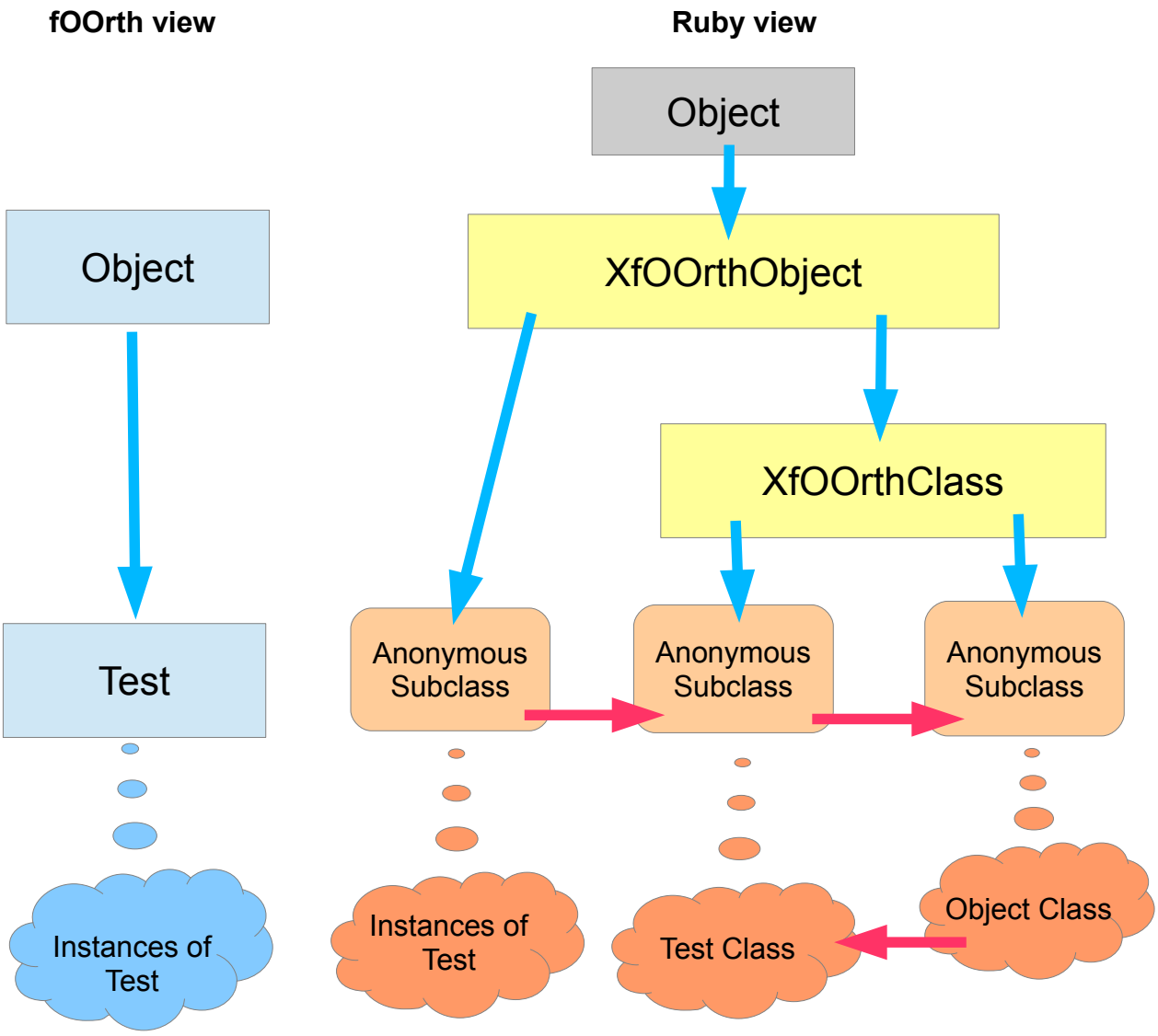
Most interesting!

Mapping fOOrth OO to Ruby OO:

Mapping classes and objects:

Consider the code:

```
class: Test ;
```



Where:

Boxes	Represent Classes.
Rounded Boxes	Represent Anonymous classes.
Clouds	Represent Instances of a class.
Cyan Lines	Depict class inheritance.
Fuchsia Lines	Represent a reference to an object.

Mind you... I am having second thoughts about using BasicObject as the root of my hierarchy. It is really lame and *really* broken. It lacks the most basic methods needed to instantiate an instance of that class. Further the trick of accessing the Kernel module explicitly with ::Kernel does not work well. For example, ::Kernel.methods produces all sorts of extraneous junk. This make me wonder if ::Kernel.object_id is also broken, to say nothing of the dire warning about redefining the object_id method!

OK... It looks like the object_id is required by the pp command. Still that warning... and the fact that most of the debug tools won't work. No, base the hierarchy on Object and not

BasicObject.

As ugly as it will be, I am leaning toward always mangling method names.

Mapping Method Names:

The SymbolTable class has a really simple interface. It has two unique methods.

```
map(symbol, option='Normal')
unmap(internal)
```

This method returns a string which is the internal name for that symbol. If the symbol is already in the symbol table, map retrieves it. If it does not yet exist, map creates it. Access to the internal hash is protected by a mutex so that this operation is thread safe.

Note that if the symbol exists and the map options do not match those of the stored symbol, the operation fails.

The unmap method translates an internal name back to the symbol name.

Note the unmap function throws an exception if the internal string is not found.

This class is to be accessible from fOOrth code, most likely through the map_symbol and unmap_symbol words.

```
(sym::string opt::string  map_symbol  int::string)
(int::string  unmap_symbol  sym::string opt::string)
```

Types and Categories of Symbols:

In classical FORTH, it is traditionally considered that there are two types of symbols. Regular symbols and immediate symbols. Regular symbols execute in execute mode and are compiled into definitions in compile mode. Immediate symbols generally do bad things in execute mode and execute in compile mode. While compiling they direct the actions and code creation of the compiler and are the basis for most meta-programming in FORTH.

This simplified view of things masks an underlying complexity. In fOOrth, this complexity takes the form of an increase in the number of types of symbols. This is based on a number of criteria including:

1. The receiver of the method.
2. The “immediacy” of the method
3. The relationship (if any) of the arguments to the receiver.

The symbol type “zoo” of the eight or nine species is presented below:

Type of Message	Receiver of the Message:	
	Virtual Machine[3]	**An Object**[4]
Regular message sent to the default receiver.	:word Implements. In the block: “self” = vm	:method Implements. In the block: “self” = the object, vm is a hidden argument.
Regular message to “this”	Not applicable.	:method Implements. In the block: “self” = the object, vm is a hidden argument.
Monadic Operator	:monadic Provides scaffolding In the block: “self” = vm	:monadic Implements. In the block: “self” = the object, vm from a thread variable.
Dyadic Operator	:dyadic Provides scaffolding In the block: “self” = vm	:dyadic Implements. In the block: “self” = the object, vm from a thread variable.
Immediate	:immediate + :word Implements In the block: “self” = vm	:immediate + :method Needed?[5] In the block: “self” = the object, vm is a hidden argument.

Note that not all of these require different symbols to represent them. For example, where a message night look like:

```
self my_message
```

This can be shortened to:

```
~my_message
```

And both are still of the species :method, the compiler will make the connection automatically to the calling convention to be used.

3 The receiver in this case is the virtual machine, which is assumed to always be available in any method.
4 An Object refers to any object, including the current Virtual Machine. The primary difference is how the message receiver is determined.
5 While no use for this is known at this time, it seems short sighted to remove this sort of meta-programming capability from objects.

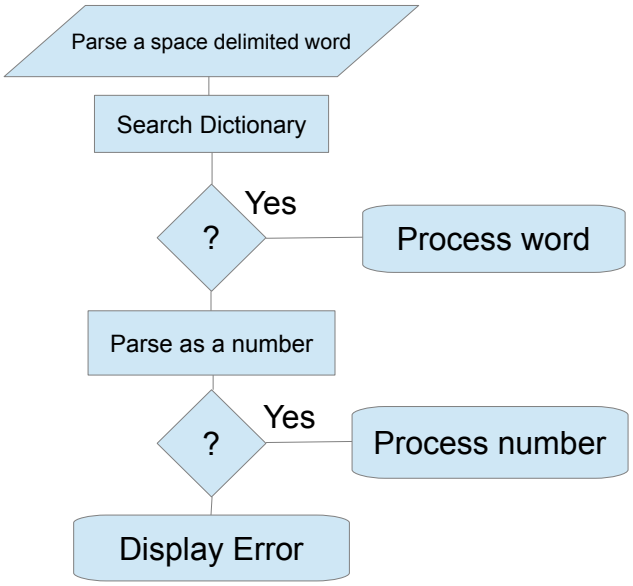
Some Fresh Ideas

After a suitably long hiatus, we return to fOOrth. I suppose that the aspect of the existing design that bothers me the most is the myriad, complex set of symbol types currently proposed for entities in fOOrth. Currently this is specified as:

```
#==== Category: sym_type
#The type of symbol being mapped. These are one of the following:
#* :empty - An empty method, no code is present or generated.
#* :word  - A classic FORTH method. Essentially a method of the VM.
#* :class - A class definition. The block returns the class object.
#* :method - An instance method of an arbitrary class.
#* :dyadic - A dyadic operator.
#* :local_variable - A local variable.
#* :instance_variable - An instance variable.
#* :thread_variable - A thread variable.
#* :global_variable - A global variable.
#==== Category: immediate
#Is this an immediate symbol that executes even in compile or deferred
#modes or is it a standard symbol that compiles or executes normally?
#* <default> - Normal priority.
#* :immediate - Priority execution.
```

FORTH – The starting point.

We start trying to figure out this mess by looking at where it all started; namely FORTH. In the original language, all code is composed of chunks called words that are placed in a dictionary. To get around the limitations of a single name space, vocabularies were added, but we can ignore these for now. When code is being interpreted or compiled, the task is really very simple. Here's a summary of classic FORTH:



Overall, very simple and very elegant. This simplicity is a large part of the charm of the FORTH language, and it is something that really ought to be preserved to the greatest possible extent.

OK, so enter fOOrth. We want to add object oriented programming, functional programming, and a more expressive and flexible syntax. This means one of two things is going to happen:

1. Simplicity is lost and we end up with an extremely complex system that is inconsistent and full of bug filled exceptions that destroy the integrity of the design.
2. A new, comprehensive vision of how things are done unifies diverse concepts under one simple world view.

So far all attempts at getting a handle on this issue have tended toward scenario number 1. This, clearly, is not where we want to be.

So why are things so complex?

Consider; in FORTH, words are essentially just methods on a virtual machine object. There is only ever one receiver for all code methods. In fOOrth things are much more complex. For starters, there are many more choices as to the receiver. These can include:

1. The virtual machine
2. The top of the data stack
3. The object associated with the method (self)
4. The second element of the data stack!

Then there are issues as to where to search for the methods. These can include:

1. The virtual machine dictionary.
2. The dictionaries of the receiver's class hierarchy, for those cases where we know the class of the receiver.
3. Local method definitions.
4. Control flow contexts.
5. Some sort of global symbol repository for the cases where we do NOT know the class of the receiver.

The fun goes on. FORTH has words, but fOOrth has:

1. Simple virtual machine words.
2. Methods applied to a class of objects.
3. Methods applied to an individual object.
4. Methods that exist locally to a method context.
5. Methods that exist locally to a control structure context.