# Appendix C: Illustrated zkl Code Examples

## Hex Dump

For programmers, a hexadecimal dump usually represents the back up against the wall: “What the *&%^ is this program *doing*?!?” and is like looking at chicken entrails. Unpleasant, yes, but none the less, you'll usually have to do it at some point.

```
>zkl hexDump Built/exception.zsc
   0: 20 20 7a 6b 48 00 00 24 | 31 2e 30 00 5a 4b 4c 20    zkH..$1.0.ZKL
  16: 53 65 72 69 61 6c 69 7a | 65 64 20 43 6c 61 73 73   Serialized Class
  32: 00 45 78 63 65 70 74 69 | 6f 6e 00 00 02 00 00 45   .Exception.....E
  48: 2b 61 74 74 72 69 62 75 | 74 65 73 3a 73 74 61 74   +attributes:stat
...
```

The above dump is an excerpt of a ZSC file dump. From this, you could verify that Asm.WriteRootClass is working correctly.

**Code**

```
fcn hexDump(in, out = File.stdout){
   format1  := "%02.x ";
   format16 := format1 * 8 + "| " + format1 * 8;
   NFmt     := "%4d: ";

   reg bytes,text, N = 0, d = Data();
   try {
      while(1){   // repeat until end of file
         bytes = in.read(16).bytes();
         text  = bytes.reduce(fcn(d,c)
            { d.append(if (0x20 <= c <= 0x7E)
                  c.toChar() else "." )},
            d.clear() ).text;
         out.writeln(NFmt.fmt(N),
             format16.fmt(bytes.xplode()),"  ",text);
         N += 16;
      }
   }
   catch(MissingArg) {     // not a full line
      n := bytes.len();
      if (n <= 8)
        out.write(NFmt.fmt(N),(format1 * n)
                  .fmt(bytes.xplode()),"  ");
      else
         out.write(NFmt.fmt(N),
           (format1 * 8 + "| " +
            format1 * (n-8)).fmt(bytes.xplode()));
      out.writeln("   "*(16-n), "  ",text);
   }
   catch(TheEnd) {}
}
```

Discussion
The code is dense.

The first thing to note is the prototype: `hexDump(in, out = File.stdout)`
Hex dump reads from an input stream (such as a File or Data) and writes to an output stream, which defaults to the console (or whatever the console has been redirected to). Thus, it is simple to create a script that dumps a file:

```
   Attributes(script);
   f := File(vm.arglist[0],"rb");
   hexDump(f);
```

This opens the file in binary mode and lets hexDump have at it. It would be even easier to do this:

```
   hexDump(File.stdin);
```

But, on windows, if there is a control-Z in the file (common in a binary file), windows will treat that as an End of File marker and close the file, which is not what we want.

Read Bytes
First, we read sixteen bytes at a time from the input stream:
   `in.read(16).bytes();` → Data(16) to List of 16 integers
The thing to note here is, if there are less than sixteen bytes available, read will read what is available. When there are no bytes available, read will throw TheEnd, which we catch. Thus, we process all bytes, even if the file doesn't contain a multiple of sixteen.

Print Bytes
Each line of sixteen bytes is written as two hex characters, a space, the next byte, etc, followed by the the bytes as text (printable characters only). The line is split into two groups of eight bytes for ease of readability. The text format for a byte is “%02x “ or “%02.16B ”; two hex digits (with leading zero if only one digit). By multiplying this string by eight, it is repeated eight times to give one group. Using this resulting string as a format, we explode the list of bytes so they become parameters to .fmt. For example: if bytes is L(8,9,10) and format is “%02x %02x | %02x”, then

```
format.fmt(bytes.xplode())
→ "%02x %02x | %02x".fmt(L(8,9,10).xplode())
→ "%02x %02x | %02x".fmt(8,9,10)
→ "08 09 | 0a"
```

Print Text

Next, we need to write the bytes as text. We'll define a printable character as one that is between space (0x20) and tilde (~, 0x7E). If the byte isn't printable, substitute a dot. The problem is how to convert a list of bytes or bunch of bytes to printable characters. The idea is to examine each byte and build a string with the printable representation of that byte. We can do this using List.reduce as a data pump; each byte is converted to printable (by a function) and then appended to the previous characters. Any of a number of objects can be used; String, Data, Pipe, etc. We'll use Data out of a (probably misguided) sense of efficiency. Here are the steps:

1. `fcn toPrintable(c)`
   ```
        { if (0x20 <= c <= 0x7E) c.toChar() else "." }
   ```
   This function takes an integer and returns a printable character. Note that we can do the comparison without using "and".
2. `fcn append(data,c) { data.append(toPrintable(c)); }`
   This will append the printable character to a Data.
3. `d = Data(); text = bytes.reduce(append, d.clear() );`
   Here we walk each byte, convert it and append it to a Data. The Data is returned. D is cleared each time so we can reuse it.
4. `text = text.text;`
   And finally!, the printable text is extracted from the Data.

And to show that we are really cool programmers, all these steps are packed into one very long statement.

If you prefer to work with Strings, lines 2 and 3 would become:

```
fcn append(str,c) { str + toPrintable(data,c); }
text = bytes.reduce(append,"");
```

and line 4 is eliminated.

Partial Lines

And lastly, we have to handle partial lines (those will less than 16 bytes). We know when that happens because fmt will throw a MissingArg if it doesn't have sixteen bytes to format. We catch that and deal with one of two cases: eight bytes or less (one group) or two groups. Since we know that fmt threw the error, we also know that bytes and text are valid (as they were calculated before the error occurred).

## Factorial

The classic recursive factorial program can be written in a clever way. We'll actually look at cleverness inside of clever, just to to be clever. A factorial program is written four ways.

### Code

1. `fcn fact(n){ return(n and n*fact(n-1) or 1); }`

2. `fcn fact(n){ n and n*self.fcn(n-1) or 1 }`

Compare to:

3.
   ```
   fcn fact(n){    // the recursive part.  input: x  output: x!
     if (0==n) return(1);      // 0! = 1
     return( n*fact(n-1) );    // n! = n*(n-1)!
   }
   ```

4.
   ```
   fcn fact(n){
       if (0==n) 1;
       else n*fact(n-1);
   }
   ```
5. `fcn fact(n){(1).reduce(n,fcn(N,n){N*n},1)}`

<u>Discussion</u>

The first thing to note is that (1) and (2) are exactly the same. (2) is just lazy since it knows that last calculation is the block result so it doesn't have to explicitly "return". (3) is almost the same as (4), (4) is actually a tiny bit bigger (two bytes) but the compiled code is essentially the same. (5) is just a loop using a lambda function.

So, how does it work? First, both programs are only valid for non-negative integers and the results get huge very fast so there is a fairly low limit on those integers before overflow occurs. With that out of the way, we can perform an examination. We'll do this the old fashioned way: plug and grind.

1. `fact(0)` → `return(0 and 0*fact(0-1) or 1)` → `return(False or 1)` → `return(1)`
   The key here is and/or. 0 is equivalent to False (`0.toBool()` is False) and `(False and X)` is False, so `0*fact(0-1)` is never evaluated and the "or" clause is the winner.
2. `fact(1)` → `return(1 and 1*fact(1-1) or 1)` → `return(True and fact(0))` → `return(fact(0))` → `return(1)`
   Here, 1 is equivalent to True and `(True and X)` is `X` so the "or" clause is ignored (assuming that X is non-zero, which we know to be true in this case).
3. `fact(2)` → `return(2 and 2*fact(2-1) or 1)` → `return(True and 2*fact(1))` → `return(2*fact(1))` → `return(2*1)` (substituting 2) → `return(2)`
   Basically the same as 2, noting that `2.toBool()` is True and using the result obtained in 2. The program performs steps 3, then 2, then 1 (which is why it is recursive) but we've wave our hands here and save some brain cells.
4. And so on for other integers.

What does the compiler think of this cleverness? Not much, the compiled code is all of two bytes smaller than the more legible code.

<u>But Wait! There's More</u>

```
fcn factTail(x,N=1) {
   if (0 == x) return(N);
   return(self.fcn(x - 1,x * N));
}
factTail(7) → 5040
```

The code is longer and isn't as “pretty”. But! it doesn't recurse. That's right, the compiler converts this into an iterative function though the magic of tail recursion. The generated code becomes something like this:

```
   fcn factTail(x,N=1) {
      if (0 == x) return(N);
      x -= 1; N *= x;
      goto factTail;
   }
```

What about integer overflow? `factTail(100)` → 0 so something is clearly wrong. Let's use infinite precision integers to avoid this problem:

```
   BigNum := Import("zklBigNum");
   factTail(BigNum(100))
   → 933262154439441526816992388562667004907159682643816214
      685929638952175999932299156089414639761565182862536979
      208272237582511852109168640000000000000000000000000
```

## Processing Text Files with Scripts and Pipes

zkl can be used to process text files in a manner somewhat analogous to Perl but not as concisely and with a different “flow”.
In this this example, the goal is to convert a look up table written in C to a gperf hash table. The command line will look like:

```
   >zkl extractTable < list.c | gperf | zkl gperf -i list
```

There are two zkl scripts (extractTable.zkl and gperf.zkl) that are used; one to extract the table from C code and prep it for gperf and the other to post process the gperf output. gperf (http://www.gnu.org/software/gperf/) is the GNU perfect hash table generator. gperf hash tables are used by the zkl VM but gperf produces C code that needs to be modified so that it can be compiled into the VM.

### Examples of Text

From list.c (over 4,000 lines of C with five tables):

```
   static Instance *
   List_makeReadOnly(Instance *self, pArglist arglist, pVM vm) {
      return convertToROList(self);
   }
   static const MethodTable listMethods[] =
   {
      "create",             List_create,
      "toString",           List_toString,
            // utility methods
      "makeReadOnly",       List_makeReadOnly,
      0,                    0
   };
```

What gperf wants to see the above transformed into:

```
      MethodTable
      %struct-type
      %language=ANSI-C
      %readonly-tables
      %delimiters=,
      %enum
      %omit-struct-type
      %%
      "create",       List_create,
      "toString",     List_toString,
      "makeReadOnly", List_makeReadOnly,
```

We'll skip the two gperfs; their output is copious and extraneous for this example.

**Code**

```
//-*-c-*-
/* extractTable.zkl: Extract a Method or Property table
 * from a C file and convert it to gperf format
 */
Attributes(script);               // (1)

var structname = "MethodTable";
var name       = "";


Import("Utils.Argh")(               // (2)
L("propertyTable","p","Extract a property table",
             fcn { structname = "PropertyTable" }),
L("methodTable",  "m","Extract a method table (default)", fcn {}),
L("+name",    "n","The name of the table", fcn(arg) { name = arg }),
).parse(vm.arglist);


walker := File.stdin.walker();      // (3)
//cFile  := File.stdin.readln(*);   // (4)
//walker := cFile.walker();

   // Match "MethodTable ???[] = " or
   //  "MethodTable methodTable[] = "
tableName := "*%s*%s\\[]*=*".fmt(structname,name);  // (5)


try { // look for: MethodTable methodTable[] =      // (6)
   while (not walker.next().matches(tableName)) {}
   walker.next();
}
catch(TheEnd) {
   File.stderr.writeln("Extract: Didn't find ",name);
   System.exit(1);
}
```

```
println(                                                // (7)
structname,"\n",
"%struct-type\n",
"%language=ANSI-C\n",
"%readonly-tables\n",
"%delimiters=,\n",
"%enum\n",
"%omit-struct-type\n",
"%%");

while (not (line := walker.next().strip()).matches("0,*0")){ // (8)
   if (not line or line.matches("//*")) continue;
   println(line);
}
```

Discussion
This code is very linear:

1. Make it clear to everybody (from the user to the compiler) that this is a script.
2. Parse the command line options to find out what table to look for.
3. Create a way to read the file a line at a time.
4. Another way to read the file: read the entire C file into a list and read from that list.
5. Create a search pattern
6. Find the table.
7. Write the gperf header.
8. Read the table, modify it to gperf format and write it.

You might be wondering why ".zkl" isn't specified on the command line

```
>zkl extractTable < list.c | gperf | zkl gperf -i list
```

for extractTable.zkl or gperf.zkl. This isn't a typo, the zkl shell will look for ".zkl" and ".zsc" files and also search the vault.

In Depth:

1. The script attribute isn't necessary in this case but it does make it clear what the intent of this code is.
2. Use the Argh class to parse the command line. Standard stuff. If an option is found (--propertyTable, --methodTable, or --name), a function is called to deal with it. If there is an error, Argh throws an exception and VM exits with a non-zero value, which breaks the pipe line.

```
>zkl extractTable -X  < list.c | gperf | zkl gperf
Unknown option: X
Options:
  --methodTable (-m) : Extract a method table (default)
  --name (-n) <arg>: The name of the table
  --propertyTable (-p) : Extract a property table
VM#32 caught this unhandled exception:
   NameError : Unknown option: X
Stack trace for VM#32
   ...
(standard input): The input file is empty!
```

`Import("Utils.Argh")(`*`optionParameters`*`).parse(`*`cmdLineArgs`*`)`
Import searches for the named class (Argh), creates a copy and calls the init function with parameters. The parse method/function of the new Argh instance is then called with the command line args. How is the command line converted into the arglist? Since a script is basically a class constructor and constructors don't have a named parameter list, the relevant parts of argv are copied to vm.arglist.
The option parameters describe each possible command line option. Each is a list: long name (with a "+" if a argument is required), short name, description and optional function.

3. This line is the key to this script. We explicitly create a Walker so we can process the file at our pace (rather than implicitly via foreach). Doing this allows us to structure our look at the file in way better suited to our needs.
4. (2) isn't the only way to do this, it isn't even the only other way. If we needed to edit the file (delete, add and modify lines, as gperf.zkl does), reading the entire file into a list of lines and using a walker on that works well; the walker also acts as a "cursor" into the file.
5. Since the table name isn't hard coded, we have to create the search pattern on the fly. We want to be a bit clever about this and not require a table name be specified. If we are looking for a method table (the default), structname is "MethodTable". Our two patterns are:
   ```
   "*MethodTable*\[]*=*"
   ```
   Or, if a name is specified (eg "listMethods"):
   ```
   "*MethodTable*listMethods\[]*=*"
   ```
   Both these patterns will match
   ```
   "static const MethodTable listMethods[] ="
   ```
6. Here we step through the file. We look at each line until one matches the pattern we are looking for. If we find it, we skip the next line ("{") and go to the next step. If we search the entire file and don't find a matching line, walker throws TheEnd. We could leave it at that but the user would probably be at a loss to figure what the hell went wrong. So we catch the exception and print a more meaningful error to standard error:
   ```
   >zkl extractTable -n fooMethods < list.c | gperf | zkl gperf
         Extract: Didn't find fooMethods
         (standard input): The input file is empty!
   ```
7. Nothing special here, just print a gperf header to standard out. Each gperf declaration needs to be on a separate line, so "\n" (newline) is used in println to specify that.
8. Walk the rest of the table, until the end of the table is reached.
   The end of the table is marked by a line that is "0,0", so we can stop when we find it. There is a lot going on in the while line. First, the line is read, leading and trailing space is stripped from it and it is stored in the "line" register. If line is "0,[*space*]0", it is the end of the table and the string *matches* method will return True. Note, if the line is "0,0," (trailing comma, a valid terminator for a C array initializer), this will miss it.

   We don't want to send blank or comment lines to gperf (it doesn't like them) so we skip them. Since we stripped out white space, a blank line is "" (which has zero length, which is what `not` tests for in strings). A comment line is a C++ style comment on a blank line ("  // comment") (a convention I force for tables, comments at the end of a non-blank line is fine). If the line matches either of these conditions, skip that line.

Otherwise, just print the table line.

## Roman Numbers

Ever consider how cool it would be to write

```
   Roman.CCXX + Roman.XLII → CCLXII(262)
```

Well, you can! If you can solve this problem: how to do you get a class to meaningfully deal with a method/fcn/etc that is unknown at compile time? In this case, it makes absolutely no sense to have a fcn/variable for every possible roman numeral, so there needs to be another way for the Roman class to accept an arbitrary roman number. The normal way would be to write Roman("CCXX") but, for this example, we want to write Roman.CCXX. The solution is use late binding to redirect the unknown method references. Most dynamic languages offer support for this in some way, shape or form. For example, Ruby has the "method_missing" method. This idea for this example came from Ruby Quiz #22, "Roman Numerals".

### Code

```
    var Roman = RomanNumber;
```

Yes, that's it. Aside from the RomanNumber class, which is a run of the mill boring class. In use:

```
   var Roman = RomanNumber;   // Force late binding
   println(Roman.CCXX);                → "CCXX(220)"
   r := Roman.XLII; println(r);        → "XLII(42)"
   println(Roman.CCXX + Roman.XLII);   → "CCLXII(262)"
   println(Roman.XL + 2);              → XLII(42)
   Roman.IA;                           → ValueError
   println(RomanNumber.toRoman(42));   → "XLII"
```

### How It Works

Just what is the man behind the curtain doing to make this work? First, we can't let the compiler statically bind to the RomanNumber class, so we store it in a variable, which will force the compiler to use late binding. Next, we link into the "method not found" method, which the VM uses to resolve something that can't be found. When we write `Roman.CCXX`, the compiler turns this into `RomanNumber.resolve("CCXX")`, which isn't in RomanNumber, so the VM calls `RomanNumber.__notFound("CCXX")`.

For extra credit, what does `Roman.CCXX("II")` do? How about `Roman.CCXX.II`? If you answered 2 and 2, you are correct. If you know why, pat yourself on the back.

### More Code

Here is the code for the RomanNumber class. It is a quick hack in every sense of the word, written solely to explore, so tread carefully if you want to use it for more than that. It is basically a port, with garnish, of Jason Bailey's solution to the Ruby Quiz #22.

```
var romans = L(   // a list of lists
   L("M", 1000), L("CM", 900), L("D",  500), L("CD", 400),
   L("C",  100), L("XC",  90), L("L",   50), L("XL",  40),
   L("X",   10), L("IX",   9), L("V",    5), L("IV",   4),
   L("I",    1));

class RomanNumber {
   var value, text;
        // create a new instance so we can add two Roman numbers
   fcn __notFound(name) { return(self(name)); }
   fcn init(text) {
      self.text = text;
      value     = toArabic(text);
   }

      // romanNumber needs to be upper case
   fcn toArabic(romanNumber) {
      if (not RegExp("^[CDILMVX]+$").matches(romanNumber))
         throw(Exception.ValueError("Not a Roman number: %s"
                                          .fmt(romanNumber)));
      reg value = 0;
      foreach R,N in (romans) {     // eg "C",100
         while (0 == romanNumber.find(R)) {
            value += N;
            romanNumber = romanNumber[R.len(),*];
         }
      }
      return(value);
   }

   fcn toRoman(i) {           // convert int to a roman number
      reg text = "";
      foreach R,N in (romans)
         { z := i / N; text += R * z; i = i%N; }
      return(text);
   }
   fcn toString   { return("%s(%s)".fmt(text,value)); }
   fcn toInt      { return(value); }
   fcn __opAdd(R) { self(toRoman(value + R.toInt())) }
}
```

Since `toArabic` and `toRoman` are probably not obvious, here are some hints.

```
toArabic("XLII"):
      R = "M",  N = 1000, value =  0, romanNumber = "XLII"
      R = "CM", N =  900, value =  0, romanNumber = "XLII"
      …
      R = "XL", N =   40, value =  0, romanNumber = "XLII"
      R = "X",  N =   10, value = 40, romanNumber = "II",
      …
      R = "I",  N =    1, value = 40, romanNumber = "II"
      R = "I",  N =    1, value = 41, romanNumber = "I"
      R = "I",  N =    1, value = 42, romanNumber = ""
```

```
toRoman(42):
      R = "M",  N = 1000, z = 0, text = "",     i = 42
      …
      R = "L",  N =   50, z = 0, text = "",     i = 42
      R = "XL", N =   40, z = 1, text = "XL",   i =  2
      R = "X",  N =   10, z = 0, text = "XL",   i =  2
      …
      R = "IV", N =    4, z = 0, text = "XL",   i =  2
      R = "I",  N =    1, z = 2, text = "XLII", i =  0
```

## Device Drivers

Device drivers. Ugh. They are always changing and you hate recompiling your code to add a new driver. If you write to a "virtual" driver, one that every "real" device driver has to conform to, you load can drivers at run time and not touch your "main" code.
Here is a (very simple) virtual driver:

```
   class VirtualDriver {
      var name = "VirtualDriver";
      fcn read(n)  { println("Reading from ",name); }
      fcn write(x) { println("Writing to ",name); }
      fcn open     { println("Opening ",name); }
      fcn close    { println("Closing ",name); }
   }
```

And now, our code that writes to the hardware driver, plus a proxy class that insulates us from the actual driver[1]:

```
   class Hardware {
      var [mixin] driver = VirtualDriver;
      fcn installDriver(driver){ self.driver = driver; }
      fcn doCoolThings {
         driver.open();
         driver.write("This is a test");
         driver.read(10);
         driver.close();
      }
   }
```

When we tell the hardware to do cool things, we get:

```
   hardware := Hardware();
   hardware.doCoolThings();
   → Opening VirtualDriver
     Writing to VirtualDriver
     Reading from VirtualDriver
     Closing VirtualDriver
```

OK, now that we have the stub of our hardware controller, let's write a driver for the Gizmo 56 hardware, install and test it. We'll cheat and create the simplest driver possible:

```
   class GizmoDriver(VirtualDriver) {
      fcn init { name = "Gizmo56 driver"; }
   }
```

1 As we can't have dynamic parents.

Install a new instance of the driver in the existing Hardware instance and run another test:

```
   hardware.installDriver(GizmoDriver());
   hardware.doCoolThings();
```

→ Opening Gizmo56 driver
   Writing to Gizmo56 driver
   Reading from Gizmo56 driver
   Closing Gizmo56 driver

This example shows that we can write our high level device controller and install device drivers at run time. There are no recompiles, no relinking, no touching the controller. There is no need to have any drivers resident at run time, once the device is queried or a configuration file is read, the appropriate driver can be loaded and easily installed. You can even swap between drivers on the fly.
The advantage to using a mixin var (vs a normal var) is that the compiler can check for invalid method calls. For example, if the Hardware class were to call "driver.foo()", the compiler would flag this as a syntax error because foo isn't in VirtualDriver. If it wasn't a mixin, this check would be punted to runtime.

## Generators

Generators allow you to create iterators that rely on a continuing series of calculations or recursion. For example, consider the following function to calculate prime numbers[2].

```
fcn findPrimes{
   println(2);    // vm.yield(2);
   n:=3; primes:=L(2);
   while(True) {
      if (not primes.filter1(fcn(p,n){n % p == 0},n))
      {    // a new prime number has been found
         println(n);    // vm.yield(n);
         primes.append(n);
      }
      n+=2;    // Prime numbers (>2) are odd
   }
}
```

This will print an infinite list of primes:
   2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 …
Which is nice but what if we want to use the primes for calculations? Enter Generators. zkl Generators are built with fibers (co-operative threads) and fibers use the VM `yield` method to yield a value, so if we replace the `println` lines (above) with `vm.yield`, we have written a Generator. All that is left to do is to package it:

```
   g := Utils.Generator(findPrimes);
   println(g.next());
```

→ 2
Or, to repeat the "print the infinite list" example:

```
   foreach prime in (Utils.Generator(findPrimes)){ prime.println();}
```

→ 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 …

2 findPrimes was modified from the Wikipedia article on Generators

In this example, foreach asks the Generator to return a Walker, which it does, then foreach uses that Walker to iterate.
To get the next twenty primes: `g.walk(20)`

The important thing to note about findPrimes is that finding the next prime number requires all of the previously found primes. It is expensive to ask for `primes[n]`, `primes[n+1]`, etc, which a Walker would typically do.

Code
```
class Generator{ // → Walker
   fcn init(f,args){
      var [const] fiber = vm.createFiber(start,vm.pasteArgs());
      returnClass(walker());
   }
   fcn walker{ (0).walker(*).tweak(next); }
   fcn [private] start(f,args){
      vm.yield(vm);
      f(vm.pasteArgs(1));      // f(args), only called once
      return(Void.Stop); // all done, stop walking
   }
   fcn [private] next{ fiber.resume() }
}
```

In this case, fibers are used as a continuation. A prime is computed, returned, the fiber stalls, and the VM state is frozen. When another prime is requested, the VM is thawed,  the fiber resumes as if nothing had happened and calculates the next prime.

Let's walk through the code and see how fibers are used. First, “init” creates a fiber running the “start” function with whatever parameters where passed in. The first thing the fiber does is to yield the “continuation” (the VM running the fiber). The fiber is now stalled but is ready to start calculating primes. A Walker is created and returned to put a nice wrapper on the Generator. Nothing happens until somebody calls “next” (via `walker.next()`). When it is called, the fiber resumes. Now start has to repackage the parameters. Init didn't know what parameters were actually passed in, if any, so it just punts whatever it is given over the fence. Now we have to deal with them. We know the function is the first item, and that the rest of the list needs to be converted into parameters to the function:

- `Generator(findPrimes)` → `start(findPrimes)` → `findPrimes()`
- `Generator(findPrimes,1,2,3)` → `start(findPrimes,1,2,3)` → `findPrimes(1,2,3)`

Now,  program execution will stay in findPrimes, until findPrimes is done (which it never is). Once you find all the primes you want, you can abandon the generator and let the garbage collector “take care” of it. If findPrimes had a built in limit and exited once it reached that limit, the Generator would `return(Void.Stop))`, which would signal the Walker that it is finished and the foreach loop would end.

A much more efficient prime finder is this sieve, which also uses Generators. It is modification of a Python program posed to StackOverflow[3]

```
fcn postponed_sieve(){              # postponed sieve, by Will Ness
   vm.yield(2); vm.yield(3);        # original code David Eppstein,
   vm.yield(5); vm.yield(7);
   D:=Dictionary();                 #       ActiveState Recipe 2002
   ps:=Utils.Generator(postponed_sieve); # a separate Primes Supply
   p:=ps.pump(2,Void);              # (3) a Prime to add to dict
   q:=p*p;                          # (9) when its sQuare is
   c:=9;                            # the next Candidate
   while(1){
      if (not D.holds(c)){        # not a multiple of previous primes
         if (c < q) vm.yield(c);  #   a prime, or
         else{   # (c==q):        #   the next prime's square:
            add(D,c + 2*p,2*p);   #     (9+6,6 : 15,21,27,33,...)
            p=ps.next();          #     (5)
            q=p*p;                #     (25)
         }
      }else{                        # 'c' is a composite:
         s:=D.pop(c);               #   step of increment
         add(D,c + s,s);            #   next multiple, same step
      }
     c+=2;                          # next odd candidate
   }
}
fcn add(D,x,s){                     # make no multiple keys in Dict
   while(D.holds(x)){ x += s }      # increment by the given step
   D[x]=s;
}
```

To generate the first 200,000 primes:

```
primes:=Utils.Generator(postponed_sieve);
N:=0d200_000;
primes.pump(N,Void,Console.println);
println("The first %,d primes.".fmt(N));
```

## Sequence/List Comprehension

List comprehension is used in functional programming to express, in a concise manner, building lists with nested loops. The resulting lists can be infinitely long. Consider this line of Haskell:

```
   take 10 [ 2*x*y | x<-[0..],x^2>3, y<-[1,3..x],y^2<100-x^2 ]
```

→ [4,6,18,8,24,10,30,50,12,36]

which means take the first 10 items from the [infinite] list created by the stuff in []s, which is an expression (2xy) and two loops with guards. The first loop has x going from zero to forever where x squared is greater than three and the second has y as 1 to x counting by two as long as y matches some calculation. Or

```
foreach x in ([0..]){
   if (x*x > 3){
```

3//http://stackoverflow.com/questions/2211990/how-to-implement-an-efficient-infinite-generator-of-prime-numbers-in-python/10733621#10733621

```
      foreach y in ([1..x,2]){
         if (y*y < 100-x*x) yield(2*x*y);
      }
   }
}
```

So, how to write a function that does comprehension? How about some recursion:

```
Given: an action, parameters and a list of sequences and filters:
   If the list is empty, run action(parameters) and return the result
   else
     Get the first sequence/filter combo from the list
        If sequence is a function, call it to get the sequence
     Get an item (from the sequence) and add it to the parameters
     Run each filter with those parameters.
     If all the filters pass, recurse (with the reduced list)
```

The Haskell equivalent becomes:

```
gerber[4](fcn(x,y){2*x*y},
   T( [0..], fcn(x){x*x>3} ),
   T( fcn(x){[1..x,2]}, fcn(x,y){y*y<100-x*x} ) )
   .walk(10) → L(4,6,18,8,24,10,30,50,12,36)
```

Not as pretty as Haskell but it works.

And here is the code:

```
grind := fcn(action,[T]args,[T]sfs){
   if(not sfs){  // at the bottom, do action
      r:=action(args.xplode());
      vm.yield(r);
      return();  // back out of recursion
   }
```

sfs is the list (of lists) containing sequences and filters (with a mixin to indicate it is expected to be a list).

Since the parameters are packaged in a list, they need to be unpackaged when calling action; .xplode() does that. This is going to be packaged in a Generator (so infinite lists aren't a problem), so we yield the result.

```
   sf:=sfs[0];   // seq & filters, T(fCreateWalker,...)
   sequence:=sf[0];
   if (self.fcn.isType(sequence))
      sequence=sequence(args.xplode());
   else if(Walker.isType(sequence))
      sequence.reset();
```

This bit of code is a bit funny. If sequence is a Walker, we need to reset it because we iterate over it multiple times. (If it is a collection, such as "abc", a [new] Walker will be created automagically). If it is a function, that function encodes the sequence, eg `fcn(x){[1..x,2]}`. The function is passed the parameters that are in play at that point.

4 Because it purées and strains data.

```
   foreach arg in (sequence){ // create or reuse Walker
      arglist:=args.append(arg);
      r:=sf[1,*].runNFilter(False,0,arglist.xplode());
      if (not r) self.fcn(action,arglist,sfs[1,*]);
   }
};
```

And this is the meat. runNFilter runs each filter, and, if a filter returns False, runNFilter stops and returns True (strangely enough). If all the filters return True, False if returned (ie no failures) and we have a good set of values. Trim the list of sequences and filters and recurse.

And now, the finishing touch: add a wrapper to simplify the call and return a Generator (see the previous example):

```
fcn gerber(action,sequencesAndFilters){
<the grind function, see above>
   return(Utils.Generator(
     grind,action,T,T(vm.pasteArgs(1))));
}
```

The nice thing about this code is that it works not just for numbers but for any collection that has a walker.[5]

5 Note: For the “do it now” case, there is a more concise solution using recursive pumps.