Introduction to Perl: Seventh Lecture

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

January 2015

- Most programmers come to Perl from imperative/OO languages like C and Java, so there's a tendency to use Perl as a *Super C*.
- But Perl has many *functional programming* techniques which we can use in our own programs:
  - **map** and **grep**
  - **code references for higher-order functions**
  - **passing functions around as values**
  - **data-driven programming: coderefs in data structures**
  - **coderefs are closures**
  - **function factories: functions that return functions!**
  - **iterators, finite and infinite**
  - **currying**
  - **lazy evaluation - handling infinite Linked lists**
- So in this lecture, I'm going to try to persuade you that *Perl is a functional language*. Well, sort of.
- I'm using the new `Function::Parameters` syntax throughout.

- We've already seen Perl's built-in **map** and **grep** operators, enabling you to transform every element of a list, or select interesting elements from a list, but we haven't stressed that these are higher order functions.
- For example, **eg1**:

```
my @orig       = (1,2,3,4);                             # 1,2,3,4
my @double     = map { $_ * 2 } @orig;                  # 2,4,6,8
my @twicelong  = map { $_, $_ * 2 } @orig;              # 1,2,2,4,3,6,4,8
my %doublehash = map { $_ => $_ * 2 } @orig;            # 1=>2, 2=>4, 3=>6, 4=>8

my @odd  = grep { $_ % 2 == 1 } @orig; my $odd=join(',',@odd);  # (1,3)
my @even = grep { $_ % 2 == 0 } @orig; my $even=join(',',@even);# (2,4)
print "odd: $odd, even: $even\n";

my @sq   = grep { my $r=int(sqrt($_)); $r*$r == $_ } @orig;     # (1,4)
my $sq = join(',',@sq);
print "sq: $sq\n";
```

- Recall that **map** and **grep** are roughly:

map OP ARRAY is

```
my @result = ();
foreach (ARRAY)
{
        push @result, OP($_);
}
```

grep OP ARRAY is

```
my @result = ();
foreach (ARRAY)
{
        push @result, $_ if OP($_);
}
```


- The most fundamental Functional Programming concept is passing functions around as values.
- You can do this in Perl using a *coderef*, a reference to a function. Like a *pointer to a function* in C terms.
- For example: **eg2** and **eg3**:

```
fun double_scalar($n)
{
        return $n * 2;
}

my $coderef = \&double_scalar;

# TIME PASSES...

my $scalar = $coderef->( 10 );

print "scalar: $scalar\n";
```

```
fun double_array(@x)
{
        return map { $_ * 2 } @x;
}

my $coderef = \&double_array;

# TIME PASSES...

my @array = $coderef->( 1, 2, 3 );

my $str = join(',',@array);
print "array: $str\n";
```

- Produces 20 and (2,4,6) as output.
- Note that a considerable amount of time may pass between taking the reference and invoking the referenced function, symbolised by **TIME PASSES** above.

- Can generalise this to **eg4**:

```
fun double_scalar($n)
{
        return $n * 2;
}

fun double_array(@x)
{
        return map { $_ * 2 } @x;
}

fun apply( $coderef, @args )
{
        return $coderef->( @args );
}

my $scalar = apply( \&double_scalar, 10 );
print "scalar: $scalar\n";

my @array = apply( \&double_array, 1, 2, 3 );
my $str = join(',',@array);
print "array: $str\n";
```

- The results are the same as before.
- Do we need to name little helper functions like `double_scalar()` that are only used to make a coderef via `\&double_scalar`? No!

- Use *anonymous coderefs* as in **eg5**:

```
fun apply( $coderef, @args )
{
        return $coderef->( @args );
}

my $scalar = apply( fun ($x) { return $x * 2 }, 10 );
print "scalar: $scalar\n";

my @array = apply( fun (@x) { return map { $_ * 2 } @x }, 1, 2, 3 );
my $str = join(',',@array);
print "array: $str\n";
```

- If we add a prototype to `apply()` via:

```
fun apply($coderef,@args) :(&@)       # or sub (&@) { my($coderef,@args)=@_;..
```

(Here, & tells Perl the given argument *must be a coderef*.)

- Then add the following inside `apply()`:

```
local $_ = $args[0];
```

(`local` saves the old value of the global $_, before setting it to the given value, the new value persists until `apply()` returns when the old value is restored.)

- Now we can write `map` like code using $_ in a code block:

```
my $scalar = apply { $_ * 2 } 10;
```


- Coderefs can be built into data structures such as:

```
my %op = (
        '+' => fun ($x,$y) { return $x + $y },
        '-' => fun ($x,$y) { return $x - $y },
        '*' => fun ($x,$y) { return $x * $y },
        '/' => fun ($x,$y) { return $x / $y },
);
```

- Then a particular coderef can be invoked as follows:

```
my $operator = "*"; my $x = 10; my $y = 20;
my $value = $op{$operator}->( $x, $y );
```

- Use to build a *Reverse Polish Notation (RPN)* evaluator:

```
fun eval_rpn(@atom)                         # each atom: operator or number
{
    my @stack;                              # evaluation stack
    foreach my $atom (@atom)
    {
        if( $atom =~ /^\d+$/ )              # number?
        {
            push @stack, $atom;
        } else                              # operator?
        {
            die "eval_rpn: bad atom $atom\n" unless exists $op{$atom};
            my $y = pop @stack; my $x = pop @stack;
            push @stack, $op{$atom}->( $x, $y );
        }
    }
    return pop @stack;
}
```


- The above RPN evaluator, with some more error checking and example calls such as:

```
my $n = eval_rpn( qw(1 2 3 * + 4 - 5 *) );
```

is **eg6**. Try it out.

- This technique is often called *data-driven* or *table-driven* programming, very easy to extend by modifying the table.
- For example, add the following operators (giving **eg7**):

```
my %op = (
....
        '%'     => fun ($x,$y) { return $x % $y },
        '^'     => fun ($x,$y) { return $x ** $y },
        '>'     => fun ($x,$y) { return $x > $y ? 1 : 0 },
        'swap'  => fun ($x,$y) { return ($y, $x) },
);
```

- %, ^ and > are conventional binary operators, but note that `swap` takes 2 inputs and produces 2 outputs - the same two, swapped!
- This works because whatever the operator returns, whether one or many results, is pushed onto the stack.

- To vary the number of inputs each operator takes, change the data structure and code slightly (giving **eg8**).
- First, change the data structure:

```
my %op = (
        '+'     => [ 2, fun ($x,$y) { return $x + $y } ],
        '-'     => [ 2, fun ($x,$y) { return $x - $y } ],
        '*'     => [ 2, fun ($x,$y) { return $x * $y } ],
        '/'     => [ 2, fun ($x,$y) { return $x / $y } ],
        '%'     => [ 2, fun ($x,$y) { return $x % $y } ],
        ...
);
```

- Here, each hash value is changed from a coderef to a *reference to a 2-element list*, i.e. a 2-tuple, of the form: `[ no_of_args, code_ref ]`.
- So each existing binary operator `op => function` pair becomes:

```
op => [ 2, function ]
```

- But now we can add unary and trinary ops as follows:

```
my %op = (
        ...
        'neg'   => [ 1, fun ($x) { - $x } ],
        'sqrt'  => [ 1, fun ($x) { sqrt( $x ) } ],
        'ifelse' => [ 3, fun ($x,$y,$z) { $x ? $y : $z } ],
);
```


- The operator invocation code changes to:

```
my( $nargs, $func ) = @{$op{$atom}};
my $depth = @stack;
die "eval_rpn: stack depth $depth when $nargs needed\n"
        if $depth < $nargs;
my @args = reverse map { pop @stack } 1..$nargs;
push @stack, $func->( @args );
```

- The `args = reverse map {pop} 1..n` line is cool:-)
- We can now write a call such as:

```
my $n = eval_rpn( qw(7 5 * 4 8 * > 1 neg 2 neg ifelse) );
```

- This is equivalent to the more normal expression:

```
if( 7*5 > 4*8 ) -1 else -2
```

- Which, because $35 > 32$, gives -1.
- Change the 5 to a 4, this (because $28 <= 32$) gives -2.
- One could make further extensions to this RPN calculator, in particular variables could be added easily enough (store them in a hash, add get and set operators). But we must move on.

- So far, we've only seen passing coderefs into functions.
- However, you can write a **function factory** which *constructs and returns a coderef*. For example:

```
fun timesn($n)
{
        return fun ($x) { return $n * $x };
}
```

- `timesn(N)` delivers a newly minted coderef which, *when it is later called with a single argument*, multiplies that argument by N.
- For example (**eg9**):

```
my $doubler = timesn(2);
my $d = $doubler->(10);         # 20

my $tripler = timesn(3);
my $t = $tripler->(10);         # 30

print "d=$d, t=$t\n";
```

- Subtlety: in C at runtime, a function pointer is simply a machine address. In Perl, a coderef is a **closure:** a *machine address plus a private environment*. In this case, each `timesn()` call has a different local variable `$n` which the coderef must remember.

- Objection 1: the previous example only used one coderef at a time. Replace the calls as follows (**eg10**):

```
my $doubler = timesn(2);
my $tripler = timesn(3);

foreach my $arg (@ARGV)
{
        my $f = $arg%2 == 1 ? $doubler : $tripler;
        my $x = $f->($arg);
        print "f->($arg)=$x\n";
}
```

- Here, we select either the doubler or the tripler based on dynamic input - the doubler if the current command line argument is odd, else the tripler. So `eg10 1 2 3 4` generates `2 6 6 12`.
- Objection 2: `$n` was a known (constant) value when the coderef was built. Did Perl rewrite it as a constant?
- We can disprove this idea - a coderef can change it's environment!

```
fun makecounter($n)
{
        return fun { return $n++ };
}
```

- To use `makecounter()` write (**eg11**):

```
my $c1 = makecounter( 10 );

my $v;
$v = $c1->(); print "c1: $v\n";
$v = $c1->(); print "c1: $v\n";
$v = $c1->(); print "c1: $v\n";
```

- Every time `$c1` is called, it retrieves the current value of it's private variable `$n`, increments it for next time, and returns the previous value. So we get `10 11 12`.
- This is a special type of closure called an *iterator*. Calling an iterator to deliver the next value is called *kicking the iterator*.
- Objection 3: anyone can juggle one ball. Can you have more than one counter? Yes! **eg12** shows this:

```
my $c1 = makecounter( 10 );
my $c2 = makecounter( 100 );
my $v;
$v = $c1->(); print "c1: $v\n"; # 10
$v = $c1->(); print "c1: $v\n"; # 11
$v = $c2->(); print "c2: $v\n"; # 100
$v = $c1->(); print "c1: $v\n"; # 12
$v = $c2->(); print "c2: $v\n"; # 101
$v = $c1->(); print "c1: $v\n"; # 13
```


- So far, our iterators have generated infinite sequences. But an iterator can terminate when it finishes iterating (like `each`):
- Return `undef` as a sentinel to inform us that the iterator has finished. For example:

```
fun upto( $n, $max )
{
	return fun {
		return undef if $n > $max;
		return $n++;
	};
}
```

- Call this with code like (**eg13**):

```
my $counter = upto( 1, 10 );
while( my $n = $counter->() )
{
	print "counter: $n\n";
}
```

- When run, this counts from 1 to 10 and then stops. Multiple counters work fine - because the closure environment includes `$n` and `$max` - **eg14** shows an example (omitted here).

- Easy to define map and grep for iterators:

```
#
# $it2 = map_i( $op, $it ): Equivalent of map for iterators.
#	Given two coderefs ($op, an operator, and $it, an iterator),
#	return a new iterator $it2 which applies $op to each value
#	returned by the inner iterator $it.
#
fun map_i( $op, $it ) :(&$)
{
	return fun {
		my $v = $it->();
		return undef unless defined $v;
		local $_ = $v;
		return $op->($v);
	};
}
```

- Now, we can write (**eg15**):

```
my $lim = shift @ARGV || 10;
my $scale = shift @ARGV || 2;

my $c = map_i { $_ * $scale } upto( 1, $lim );

while( my $n = $c->() ) { print "$n,"; }
print "\n";
```

- When run with `lim=10, scale=3`, this produces:

`3,6,9,12,15,18,21,24,27,30,`

- `grep_i($op, $it)` is not much more complicated, **eg16** shows it (omitted here).

- A hard-core functional programming feature is **Currying**: the ability to *partially call a function* - to provide (say) a 3-argument function with it's first argument and deliver a 2-argument function.
- Simple to do:

```
fun curry( $func, $firstarg )
{
	return fun {
		return $func->( $firstarg, @_ );
	};
}
```

- Call this with code like (**eg17**):

```
fun add($a,$b) { return $a + $b };

my $plus4 = curry( \&add, 4 );		# an "add 4 to my arg" func
my $x = $plus4->(10);			# x=10+4 i.e. 14
print "x=$x\n";
```

- As expected, the `$plus4` function acts exactly as an *add 4 to my single argument* function, delivering 14 as the result.

- One of the coolest features of functional programming languages is **lazy evaluation** - the ability to handle very large or even infinite data structures, evaluating only on demand.
- It's surprisingly easy to add laziness in Perl:
- Let's extend last lecture's linked `List` module to work with *lazy linked lists* (sometimes known as *streams*).
- Only one design change is needed: allow a list tail to *either* be an ordinary *nil-or-cons* list *or a coderef* - a **promise** to deliver the next part of the list (whether empty or nonempty) on demand.
- When `$list->headtail()` splits a node into head `$h` and tail `$t`, need to detect (via `ref($t) eq "CODE"`) whether `$t` is a promise (coderef).
- If `$t` is a promise, we **force the promise:** invoke the promise function, delivering the real nil-or-cons tail list:

```
my( $h, $t ) = @$self;
$self->[1] = $t = $t->() if ref($t) eq "CODE";    # FORCE A PROMISE
return ( $h, $t );
```

- Note that after forcing the promise, we assign the result back into `$self->[1]` in case the same list node is re-evaluated later.

- Note: a lazy list may be finite or infinite. Given an infinite list `$inflist`: `$inflist->len`, `$inflist->rev` and `$inflist->append($second_list)` will never terminate. This can't be solved - it's inevitable!
- Fortunately, we have already engineered the concept of *"show only the first N elements"* into `$inflist->as_string()` so that's ok.
- Perhaps we should set the system-wide limit to a reasonably large value, rather than leaving it zero (meaning unlimited):

```
our $as_string_limit = 40;
```

- Having modified and syntax checked `List.pm`, check that it still works with lists with no promises - i.e. non lazy lists (**eg18**):

```
use List;
$List::as_string_limit = 8;

# list_upto: return a non-lazy list of numbers between $min and $max
fun list_upto( $min, $max )
{
        return List->nil() if $min > $max;
        return List->cons( $min, list_upto($min+1, $max) );
}

my $list = list_upto( 100, 200 );
print "first few elements of upto(100,200) List: $list\n";
```


- Then, give it a lazy list (**eg19**) by adding a `fun {}` or `sub {}` coderef wrapper on the `list_upto($min+1,$max)` call:

```
return List->cons( $min, fun { list_upto($min+1, $max) } );
```

- Without this, it was a conventional recursive function to generate a list. By *delaying the recursive call* until it's actually needed, we make it lazy.
- In this case, despite producing identical output, the lazy version never computes or stores elements 108..200.
- Can define *map-like* and *grep-like* operators for lazy lists. Here's `map_l($op, $list)`:

```
return List->nil() if $list->isnil;
my( $h, $t ) = $list->headtail;
local $_ = $h; # set localised $_ for op
return List->cons( $op->($h), fun { map_l( $op, $t ); } );
```

- Note that we've not made this a method, as we prefer to keep the map-like syntax rather than swap the arguments around in order to have the list (object) as the first argument. Instead we've given it a non clashing name and exported it.

- `grep_l($op, $list)` is:

```
while( ! $list->isnil )
{
  my( $h, $t ) = $list->headtail;
  local $_ = $h;                          # set localised copy of $_
  if( $op->($h) )                         # for the filter operation call
  {
    return List->cons( $h, fun { grep_l( $op, $t ) } );
  }
  $list = $t;
}
return List->nil;
```

- Using `map_l($op, $list)` and `grep_l($op, $list)`, we can write rather pretty mathematical-style code. For example, start with an infinite list of odd numbers (**eg20**):

```
use List;
$List::as_string_limit = 8;

# $list = stepup( $n, $step ) - return an infinite list n, n+step, n+2*step...
fun stepup( $n, $step )
{
        return List->cons( $n, fun { stepup($n+$step,$step); } );
}

my $odds = stepup( 1, 2 );
print "first few odds: $odds\n";
```

- Which produces:

```
first few odds: [1,3,5,7,9,11,13,15,17,19...]
```

- Now generate an infinite list of even numbers by:

```
my $evens = map_l {$_ + 1} $odds;
print "first few evens: $evens\n";
```

  Unsurprisingly, this produces:

```
first few evens: [2,4,6,8,10,12,14,16,18,20...]
```

- Now select only even numbers greater than 7:

```
my $evengt7 = grep_l {$_ > 7} $evens;
```

  Which produces:

```
first few even gt7: [8,10,12,14,16,18,20,22,24,26...]
```

- Finally, select the subset that are exact squares:

```
my $squares = grep_l { my $r = int(sqrt($_)); $r*$r == $_ } $evengt7;
```

  Which produces:

```
first few even perfect squares > 7: [16,36,64,100,144,196,256,324,400,484...]
```

- Of course, this sequence of calls could be written as (**eg20a**):

```
my $evensgt7 = stepup( 8, 2 );
my $squares  = grep_l { my $r = int(sqrt($_)); $r*$r == $_ } $evensgt7;
```


- Can provide a `merge_l( $cmp, $list1, $list2 )` list operator to merge two sorted lists using a sort-like comparator, and using it (**eg21**):

```
my $odds  = stepup( 1, 2 );
my $evens = stepup( 2, 2 );
my $all   = merge_l { $a <=> $b } $odds, $evens;
```

  What do you get it by merging odd and even integers? All integers!

- A better example might be (**eg22**):

```
# $list = power( $n, $p ) - return an infinite list n, n*p, n*p^2..
fun power( $n, $p )
{
        return List->cons( $n, fun { power($n*$p,$p); } );
}

my $twos   = power( 1, 2 );             # powers of 2
my $threes = power( 1, 3 );             # powers of 3
my $fives  = power( 1, 5 );             # powers of 5

my $m23    = merge_l { $a <=> $b } $twos, $threes;
my $m235   = merge_l { $a <=> $b} $m23, $fives;
my $all    = grep_l  { $_ > 1 } $m235;
print "first few merged values: $all\n";
```

- Here's a use for **currying** the comparator into `merge_l` (**eg22a**):

```
my $numeric_merge = curry( \&merge_l, fun { $a <=> $b } );
my $m235          = $numeric_merge->( $numeric_merge->( $twos, $threes ), $fives );
my $all           = grep_l { $_ > 1 } $m235;
```