Iteration in Perl

Abram Hindle
abram.hindle@ualberta.ca
Department of Computing Science
University of Alberta
http://softwareprocess.es/

Warning

- This talk is not completely idiomatic

Iteration

- Programming is useful because we can repeat tasks
- Iteration is one of the fundamental building blocks of most programming languages
- Iteration usually refers to repetition
  - In mathematics iterating a functions means applying it repeatedly
  - In programming iteration can refer to any method of repetition.

Concepts

- Block
  - A chunk of code
  - Usually the part that gets repeated
  - Usually follows scope rules
- Condition
  - An expression that evaluates to true or false
  - Commonly used to determine if a loop continues

Concepts

- Strict – Everything gets done
- Lazy – Something gets done when needed.
- In Order – Execute in Sequence
- Out of Order – Execute in any order
- Dependency – One value depends on another

While Loops

- Repeats a block until a condition is met.
- Loop invariant executes first

```
# while loop
my $condition = 1;
while ($condition) {
    # ...
    $condition = !$condition;
}
# condition is False

#how many lines ?
my $count = 0;
$count++ while(<STDIN>);
```

```
my $x = 10;
while ($x > 0) {
    $x--;
}
# x is 0

# maybe you're not sure how
# many iterations you need?
my $x = 100.0;
while ($x > 1) {
    $x /= 3;
}
```

For Loop

- Iterate with a condition or over a collection of elements.

```
# for loop
my $sum = 0;
for my $i (1..9) {
    $sum += $i;
}
# sum is 45
```

```
my $s = "";
for my $elm ("a","b","c") {
    $s .= $elm;
}
# s is abc
```

```
# for loop
my $sum = 0;
for ( my $i = 1 ; $i < 100000; $i++ ) {
    $sum += $i;
}
# sum is 4 999 950 000
```

Recursion

- Arbitrary flow control
- Good for iterating datastructures like trees
- Watch out for stackoverflows!

```
# recursive way
sub recsum {
    if (@_) {
        my $v = shift @_;
        return $v + recsum(@_);
    }
    return 0;
}

print recsum(1..10);
```

OO Iteration (Iterators)

- Object with a `next()` method and **has_next()**

```
# The OO way
package OnlyEvens;
use Moose;
has seq => (is=>'rw');
has index => (is => 'rw',
    default => 0);
sub has_next {
    my ($self) = @_;
    return $self->index < scalar(@{$self->seq});
}
sub next {
    my ($self) = @_;
    my $v = $self->seq->[$self->index];
    $self->index($self->index + 2);
    return $v;
}
1;
```

```
my $iter = OnlyEvens->new(
        seq => [1..10] );
while ($iter->has_next()) {
    print $iter->next().$/;
}
```

Order

- Did you notice something?
- Everything iterated in order.
- But what if order doesn't really matter?

Map

- In mathematics iterating a functions means applying it repeatedly
- A map function applies 1 function to all elements in a collection and produces a new collection of the results of that function
  - Usually this is in order
  - But you don't have to do it in order
- Map f (a,b,c) => (f(a), f(b), f(c))
- Map square (1,2,3) => (1,4,9)

Map Example

```
# add 1 to a list
my @v = (1..30);
my @u = map { $_ + 1 }  @v;
# u is now [2,3,4,..,31], v is still [1,2,3]

use File::Basename;

my @v = ("/home","/file","/usr/local");
my @u = map { basename $_ } @v;
my @u = map(uc,@v);
#['/HOME', '/FILE', '/USR/LOCAL']

use LWP::Simple;

my @urls = ("http://cbc.ca","http://gc.ca","http://alberta.ca");
my @pages = map { get $_ } @urls;
```

Parallelism with Map

- If you think in “map” then you can parallelize with map
- Limit dependencies of a block in order to parallelize the computation!

```
# this is why you want blocks with
# few dependencies!
# slower
use Parallel::parallel_map;
sub square { $_[0] * $_[0] }
my @u = parallel_map {square($_)}  1..1000000;
print scalar(@u);
#999999
#IO is good candidate for parallelism
my @pages = parallel_map { get $_ } @urls;
```

Reduce

- Linear, 1 at a time
- Collapse a collection in a single value via an operator or function of 2 args
  - f(e1, f(e2, f(e4, ... f(e99,e100)))...)))
  - add(e1, add(e2, add(e4, ... add(e99,e100)))...)))
- Sum is a reduce

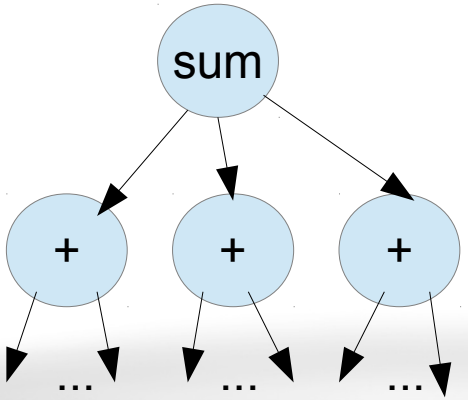
```
use List::Util qw(reduce sum);
# 1 + 2 + 3 + ... + 99 + 100
print(reduce { $a + $b } 1..100);
print(sum(1..100));
#5050
```

Reduce Can Be Parallel (Sometimes)

```
use List::Util qw(reduce sum);
use List::MoreUtils qw(part);
use POSIX;
sub split_list {
    my ($n, @args) = @_;
    my $i = 0;
    my $total = ceil(@args / $n);
    return part { int( ($i++) / $total ) } @args;
}
sub parallel_square {
    parallel_map { $_ * $_ } @_;
}
sub parallel_sum {
    my @args = @_;
    sum( parallel_map { sum(@$_) } split_list(4, @args) );
}
my $sum = parallel_sum( parallel_square( 1 .. 100 ) );
```

Trees, Communtativeness and Initialization

- Can your problem be modelled as a TREE?
- Problems with commutative or associative parts can often be modelled as a tree of computation.
- Different branches may be executed in Parallel.
- One can reduce dependencies by avoid initialization (e.g. sum = 0)



Conclusions

- Main forms of perl iteration:
  - For / While / Iterators / Recursion / Map / Reduce
- Reducing dependencies in blocks allows iteration to be parallelized.
- Consider if order or strictness can be are actually needed?
- These concepts apply to other languages as well.