

1. Bucles múltiples

Transcripción de notas antiguas sobre una clase en C++ para bucles múltiples. Se trata de una generalización del método que volví a desarrollar para `olap.w`.

(no sé de cuando son las notas, pero no usaba `bool` y el código fue implementado en RPL para la HP28S, y en C++ en GAS30/GEO/`rtnlzs.h`)

La forma de implementación es mediante una clase abstracta con métodos virtuales para el control de cada uno de los bucles anidados y una para la ejecución del código del bucle (*yield*). Las clases derivadas implementan tanto el cuerpo de ejecución y además llevan la cuenta de los contadores o variable de bucle.

El bucle ejecutado es equivalente a:

```
for (init(0); check(0); next(0))
{
  yield_pre(0);
  for (init(1); check(1); next(1))
  {
    yield_pre(1);
    for (init(2); check(2); next(2))
    {
      ...
      yield_pre(n-2);
      for (init(n-1); check(n-1); next(n-1))
      {
        yield_pre(n-1);
        yield_pos(n-1);
      }
      yield_pos(n-2);
      ...
    }
    yield_pos(1);
  }
  yield_pos(0);
}
```

Notar que el argumento de `yield_...`, tiene el mismo sentido que el de las otras funciones virtuales: no se trata de una variable de bucle (algo que la clase derivada debe implementar internamente) sino del nivel del bucle que se va a ejecutar.

2. Implementación en C++

```
class MultiLoop
{
public:
  MultiLoop(int m) : n(m) {}
  void Loop();
  virtual ~MultiLoop() {}
protected:
  virtual void init(int)=0;
  virtual bool check(int)=0;
  virtual void next(int)=0;
  virtual void yield_pre(int)=0;
  virtual void yield_pos(int)=0;
private:
  const int n;
};

void MultiLoop::Loop()
{
```

```

int k=-1;
for (;;)
  if (k<0 || (k<n && check(k)))
    {
      if (k>=0)
        yield_pre(k);
      k++;
      if (k<n) init(k);
    }
  else
    if (k>0)
      {
        k--;
        yield_pos(k);
        next(k);
      }
    else
      break;
}

```

TODO: test code

3. Aplicación: continuantes

La aplicación original de estos bucles múltiples fue el cálculo de "continuantes", polinomios relacionados con las fracciones continuas (ver Knuth 4.5.3) definidos por:

$$Q_n(x_1, x_2, \dots, x_n) = \begin{cases} 1, & \text{sin} = 0; \\ x_1, & \text{sin} = 1; \\ x_1 Q_{n-1}(x_2, \dots, x_n) + Q_{n-2}(x_3, \dots, x_n), & \text{sin} > 1; \end{cases}$$

Que también cumplen:

$$Q_n(x_1, x_2, \dots, x_n) = \begin{cases} 1, & \text{sin} = 0; \\ x_1 = x_n, & \text{sin} = 1; \\ x_n Q_{n-1}(x_1, \dots, x_{n-1}) + Q_{n-2}(x_1, \dots, x_{n-2}), & \text{sin} > 1; \end{cases}$$

El cálculo mediante la fórmula recursiva ha sido implementado en C++ en `geo/rtnlizr.h` y en Ruby en `num.w`

Una forma de generar estos polinomios es eliminar pares de variables consecutivas del término $x_1 x_2 \dots x_n$, sumando los términos resultantes. El siguiente pseudocódigo realiza esta operación:

(nunca llegué a emplear este código en C++, pero implementé este algoritmo en la HP28S para convertir números en coma flotante en fracciones –o quizás no)

```

sum = 0;
for (np=0; np<=(n/2); np++)
{
  // remove np pairs
  for (i0=0; i0<=n-np*2; i0++)
  {
    remove_pair(i0);
    for (i1=i0+2; i1<n-(np-1)*2; i1++)
    {
      remove_pair(i1);
      for (i2=i1+2; i2<n-(np-2)*2; i2++)
      {
        remove_pair(i2);
        ...
        for (i_np-1=i_np-2+2; i_np-1<=n-(np-(np-1))*2; i_np-1++)
        {
          remove_pair(i_np-1);

```

```

        sum += product_of_left_elements();
        restore_pair(i_np-1);
    }
    ...
    restore_pair(i2)
}
restore_pair(i1)
}
restore_pair(i0);
}
}

```

3.1. Implementación en C++

Y esta es la implementación usando MultiLoop: m es el valor n del pseudocódigo y n el valor np (lo siguiente no es C++ válido, hay que sustituir `bitset<n>...`)

```

⟨Q cpp 3a⟩ ≡
class QLoop : public MultiLoop
{
public:
    double result() { return sum; }
private:
    vector<int> i; // initialize to size n;
    vector<double> x; // initialize to Q=x0,...x_m-1
    size_t m; // initialize to size of Q
    bitset<n> f; // initialize to full set
    double sum; // initialize to 0.0
protected:
    void init(int k) { i[k] = k ? i[k-1]+2 : 0; }
    bool check(int k) { return i[k]<=(m-(n-k)*2); }
    void next(int k) { i[k]++; }
    void yield_pos(int);
    void yield_pre(int);
};
◇

```

Fragmento definido en [3ab](#), [4abc](#).

Fragmento no usado.

```

⟨Q cpp 3b⟩ ≡
void QLoop::yield_pre(int k)
{
    f.reset(i[k]);
    f.reset(i[k]+1);
    if (k==n-1)
    {
        ⟨Producto 5a⟩
    }
}
◇

```

Fragmento definido en [3ab](#), [4abc](#).

Fragmento no usado.

```

⟨ Q cpp 4a ⟩ ≡
void QLoop::yield_pos(int k)
{
    f.set(i[k]+1);
    f.set(i[k]);
}
◇

```

Fragmento definido en [3ab](#), [4abc](#).
Fragmento no usado.

```

⟨ Q cpp 4b ⟩ ≡
QLoop::QLoop(vector<double> _x, int np)
: MultiLoop(np), x(_x), i(n), m(_x.size()), sum(0.0)
{
    f.set();
    if (np)
        Loop();
    else
    {
        ⟨ Producto 5a ⟩
    }
}
◇

```

Fragmento definido en [3ab](#), [4abc](#).
Fragmento no usado.

```

⟨ Q cpp 4c ⟩ ≡
double Q(vector<double> x)
{
    double q= 0;
    #if optimize
    ⟨ Optimization 4d ⟩
    #end if
    for (int np=0; np<x.size()/2; ++np)
    {
        QLoop loop(x,np);
        loop.Loop();
        q += loop.result();
    }
    return q;
}
◇

```

Fragmento definido en [3ab](#), [4abc](#).
Fragmento no usado.

```

⟨ Optimization 4d ⟩ ≡
switch (m)
{
    case 0: return 1.0; break;
    case 1: return x[0];
    case 2: return x[0]*x[1]+1;
    case 3: return x[0]*x[1]*x[2]+x[0]+x[2];
}
◇

```

Fragmento usado en [4c](#).

```

<Producto 5a> ≡
double prod = 1.0;
#if 0
// GAS BitSet interface
for (BitSet :: Ranges r=f; r; r.next())
  for (int j=r.inf(); j<=r.sup(); j++)
    prod *= x[j];
#else
  for (size_t j=0; j<m; j++)
    if (f[j])
      prod *= x[j];
#endif
sum += prod;
◇

```

Fragmento usado en [3b](#), [4b](#).

4. Ruby

4.1. MultiLoop: búcles múltiples genéricos

```

"goi/multiloop.rb" 5b ≡
<License 5c>
<multiloop.rb comments 5d>
module MultiLoop
  def runLoop(levels)
    # levels can be anything with to_a; in particular any enumerable
    # such as a range (0.. n)
    <loop 6a>
  end
end
◇

```

```

<License 5c> ≡
# Copyright (C) 2003–2005, Javier Goizueta <javier@goizueta.info>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
◇

```

Fragmento usado en [5b](#), [7a](#), [8](#), [9](#), [11ab](#).

```

<multiloop.rb comments 5d> ≡
=begin
multi-level loop mixin: a class that implements a multiloop must provide
  init(level), check(level), next(level), yield_pre(level) and yield_pos(level)

then, runLoop(levels) (with levels a collections of objects <1>,<2>,...,<n>)
is equivalent to:

  init(<1>)
while check(<1>)
  yield_pre(<1>)
  init(<2>)
while check(<2>)
  yield_pre(<2>)
  init(<3>)
while check(<3>)
  ...
  yield_pre(<n-1>)
  init(<n>)

```

```

    while check(<n>)
      yield_pre(<n>)
      yield_pos(<n>)
      next(<n>)
    end
    yield_pos(<n-1>)
    ...
    next(<3>)
  end
  yield_pos(<2>)
  next(<2>)
end
yield_pos(<1>)
next(<1>)
end

=end
◇

```

Fragmento usado en [5b](#), [7a](#).

```

<loop 6a> ≡
  levels = levels.to_a
  n = levels.length
  level_i = -1

  loop do
    if level_i < 0 || (level_i < n && check(levels[level_i]))
      yield_pre(levels[level_i]) if level_i >= 0
      level_i += 1
      init levels[level_i] if level_i < n
    elsif level_i > 0
      level_i -= 1
      yield_pos levels[level_i]
      self.next levels[level_i]
    else
      break
    end
  end
end
◇

```

Fragmento usado en [5b](#).

4.1.1. A more Rubysque approach

A different approach to program nested loops of indeterminate depth, more in the line of the language Ruby would be this one: given an enumerable class (one with an `each` iterator method), such that each object yielded by `each` is in turn enumerable, including this mixin in the class adds a method `multi_each` that performs a nested iteration on all the iterators returned by `each`, yielding an array that contains an element from each sub-iterator.

We achieve something equivalent to:

```

<multienum.rb comments 6b> ≡
  =begin
  #multi_each pseudo code
  def multi_each
    v = []
    self.each { |i| v << i}
    n = v.length
    tuple = Array.new(n)
    v[0].each do |v0|
      tuple[0] = v0
      v[1].each do |v1|

```

```

    tuple[1] = v1
    ...
    v[n-1].each do |vm|
      tuple[n-1]=vm
      yield tuple
    end
  ...
end
end
=end
◇

```

Fragmento no usado.

Note: we could use instance variables for some of the arguments of `each_level`, such as `v` or `tuple`, and that's probably more efficient, but we don't do that for the shake of a less intrusive solution. If instance variables were to be used, a prefix such as `multiEnumerable_` should be used.

```

"goi/multienum.rb" 7a ≡
<License 5c>
<multiloop.rb comments 5d>
module MultiEnumerable
  def multi_each(&blk)
    v = []
    self.each { |i| v << i }
    @multiEnumerable_tuple = Array.new(v.length)
    each_level(v,-1,nil,&blk)
  end
  protected
  def each_level(v,l,vl,&blk)
    n = v.length
    @multiEnumerable_tuple[l] = vl if l>=0
    yield @multiEnumerable_tuple if l==(n-1)
    if l<(n-1)
      v[l+1].each { |vi| each_level(v,l+1,vi,&blk) }
    end
  end
end
◇

```

Archivo definido en [7ab](#).

```

"goi/multienum.rb" 7b ≡
module MultiEnumerable
  public
  def multi_each_index(&blk)
    v = []
    self.each_index { |i| v << i }
    @multiEnumerable_tuple = Array.new(v.length)
    each_index_level(v,-1,nil,&blk)
  end
  protected
  def each_index_level(v,l,vl,&blk)
    n = v.length
    @multiEnumerable_tuple[l] = vl if l>=0
    yield @multiEnumerable_tuple if l==(n-1)
    if l<(n-1)
      self[v[l+1]].each_index { |vi| each_index_level(v,l+1,vi,&blk) }
    end
  end
end
◇

```

Archivo definido en [7ab](#).

4.2. Cálculo de continuantes

4.2.1. Vectores de bits

```
"goi/bitset.rb" 8 ≡
<License 5c>
class Bitset
  def initialize (n, fill_with=false)
    #bit set for 0,1,... n-1
    @n = n
    if fill_with==true
      fill
    else
      clear
    end
  end
  def get_bit(i) # true/false
    (@set & mask(i))!=0
  end
  def set_bit(i,v=true) # true/false
    if v
      @set |= mask(i)
    else
      @set &= c_mask(i)
    end
  self
  end
  def clear() # all bits to false
    @set = 0
  self
  end
  def fill () # all bits to true
    @set = full
  self
  end
  def each_true# iterator over true bits
    (0... @n).each do |i|
      yield(i) if get_bit(i)
    end
  end
  def each_false(&blk)
    (0... @n).each do |i|
      yield(i) if !get_bit(i)
    end
  end
  def each
    (0... @n).each do |i|
      yield(i, get_bit(i))
    end
  end

  private
  def base
    # 2**@n
    1 << @n
  end
  def full
    # 2**@n-1 = (1<<@n)-1
    v = 0
    @n.times { v <<= 1; v |= 1; }
    v
  end
end
```

```

def mask(i)
  # 2**i
  1 << i
end
def c_mask(i)
  ~mask(i) + base
end
end
◇

```

4.2.2. QLoop

```

"qloop.rb" 9 ≡
<License 5c>
require 'goi/multiloop'
require 'goi/bitset'
class QLoop
  include MultiLoop
  def initialize(x, np)
    @x = x
    @n = x.length
    @np = np # number of pairs to remove from x[0]*x[1]*...*x[n-1]
    @i = Array.new(@np) # indices to iterate over all pairs
    # @f = BitSet.new(@n).fill(); # with new constructor, this can be abbreviated:
    @f = BitSet.new(@n, true)
    @sum = 0
    if @np > 0
      runLoop(0...@np)
    else
      @sum = product
    end
  end
  def result
    @sum
  end

  protected
  def product()
    prod = 1
    @f.each_true do |i|
      prod *= @x[i]
    end
    prod
  end

  def init(k)
    @i[k] = k==0 ? 0 : @i[k-1]+2
  end
  def check(k)
    @i[k] <= (@n - (@np - k) * 2)
  end
  def next(k)
    @i[k] += 1
  end
  def yield_pos(k)
    # restore pair
    @f.set_bit(@i[k], true)
    @f.set_bit(@i[k]+1, true)
  end
  def yield_pre(k)
    # remove pair
    @f.set_bit(@i[k], false)
    @f.set_bit(@i[k]+1, false)
  end
end

```

```

        if k==@np-1
          # multiply non-removed factors
          @sum += product
        end
      end
    end
  end
end
◇

```

Archivo definido en 9, 10ab.

```

"qloop.rb" 10a ≡
def Q(x)
  q = 0
  (0..x.length()/2).each do |np|
    mloop = QLoop.new(x,np)
    q += mloop.result
  end
  q
end
◇

```

Archivo definido en 9, 10ab.

optimizations: 1. as in the recursive version from num.rb, for small n values, we can directly use an expression for the result

2. for the np=0 iteration (remove no pairs), the result is the product of all x[i]

3. if x.length is an even number, then the iteration np=x.length()/2 removes all pairs and yields 1

optimized version:

```

"qloop.rb" 10b ≡
def Q(x)
  q = 0
  n = x.length

  # specific results for small n
  if n==0
    return 1
  elsif n==1
    return x[0]
  #els ...# as in num.rb
  end

  # add np=0 term
  q = 1
  x.each { |xi| q *= xi }

  if n%2
    last = n/2
  else
    last = n/2-1
    q += 1
  end
  (1..last).each do |np|
    mloop = QLoop.new(x,np)
    q += mloop.result
  end
  q
end
◇

```

Archivo definido en 9, 10ab.

4.2.3. Test Program

Test Multiloop:

```

"test_multiloop.rb" 11a ≡
<License 5c>
require 'goi/multiloop'
class TestLoop
  include MultiLoop
  def initialize (n)
    @n = n
    @i = Array.new(n)
    @i.collect! { 0 }
  end
  def run
    runLoop (0...@n)
  end

  protected
  def init (k)
    @i[k] = 0
  end
  def check(k)
    @i[k]< 3
  end
  def next(k)
    @i[k] += 1
  end
  def yield_pos(k)
  end
  def yield_pre(k)
    p "Pre (#{k}) : _#{@i[0..k]}"
  end
end

test = TestLoop.new(4)
test.run
◇

```

Test MultiEnumerable:

```

"test_multienum.rb" 11b ≡
<License 5c>
require 'goi/multienum'
class TestEach
  include MultiEnumerable
  def initialize (n)
    @v = Array.new(n)
    @v.collect! { (0...3) .to_a }
  end
  def each(&blk)
    @v.each(&blk)
  end
  def run
    multi_each do |v|
      p v
    end
  end
end

test = TestEach.new(4)
test.run
◇

```

5. Otras aplicaciones

Otra aplicación de los bucles múltiples, muy relacionada con los cubos de datos de `olap.w` es la proyección de múltiples variables en una tabla relacional realizada en `simple.w`.

6. Estructuras de control

Pendiente: transcribir notas sobre estructuras de control

7. Índices

7.1. Archivos

"`goi/bitset.rb`" Definido en [8](#).
"`goi/multienum.rb`" Definido en [7ab](#).
"`goi/multiloop.rb`" Definido en [5b](#).
"`qloop.rb`" Definido en [9](#), [10ab](#).
"`test_multienum.rb`" Definido en [11b](#).
"`test_multiloop.rb`" Definido en [11a](#).

7.2. Fragmentos

`< License 5c >` Usado en [5b](#), [7a](#), [8](#), [9](#), [11ab](#).
`< Optimization 4d >` Usado en [4c](#).
`< Producto 5a >` Usado en [3b](#), [4b](#).
`< Q cpp 3ab, 4abc >` No usado.
`< loop 6a >` Usado en [5b](#).
`< multienum.rb comments 6b >` No usado.
`< multiloop.rb comments 5d >` Usado en [5b](#), [7a](#).

7.3. Identificadores