El Despegue

Lenguajes de Programación — Tarea 2

Eduardo Acuña Yeomans*

22 de agosto de 2022

Resuelve los problemas en un archivo llamado `problemas.rkt`, escribe tus pruebas en un archivo llamado `problemas-pruebas.rkt`. Para cada procedimiento que implementes, debes escribir pruebas suficientes para contemplar una clase de entradas para cada posible caso. En algunos problemas se pide escribir o responder preguntas, esto lo debes incluir en un documento `reporte.tex` implementado en TEX (o algúna descendiente) y `reporte.pdf` su versión compilada. Debe quedar extremadamente claro a qué problema corresponde cada parte del reporte.

1 Plait

En partes de este curso, vamos a utilizar el lenguaje Plait, el cuál está basado en Racket pero utiliza un sistema de tipos parecido al de ML.

Sigue el tutorial de Plait para familiarizarte con el lenguaje, puedes omitir únicamente la sección titulada *State* (1.14 la última del tutorial).

Problema 1: Implementa el lenguaje aritmético en un archivo llamado `arithlang.rkt` e incluye pruebas de tu intérprete en un archivo llamado `arithlang-tests.rkt`.

Tu implementación debe incluir la definición de los procedimientos `parse`, `desugar` e `interp`, así como de `eval` definida de la siguiente forma:

```
(define (eval [input : S-Exp]) : Number
  (interp (desugar (parse input))))
```

La siguiente definición de `parse` corresponde al lenguaje núcleo (por lo que es incompleta e incorrecta), pero puede ser ilustrativa para implementar el procedimiento `parse` que entregarás en tu propuesta:

```
(define (parse [s : S-Exp]) : ArithC
  (cond [(s-exp-number? s) (numC (s-exp->number s))]
        [(s-exp-list? s)
         (let ([ls (s-exp->list s)])
           (case (s-exp->symbol (first ls))
             [(+) (plusC (parse (second ls)) (parse (third ls)))]
             [(*) (multC (parse (second ls)) (parse (third ls)))]
             [else (error 'parse "operación aritmética malformada")]))]
        [else (error 'parse "expresión aritmética malformada")]))
```

Las pruebas relevantes para esta tarea deben realizarse sobre `eval`, no sobre `parse`, `desugar` o `interp`, pero puedes escribir pruebas para estos procedimientos en otro archivo para tu uso personal. En las pruebas que entregues, debes suponer que las funcionalidades del lenguaje funcionan, por ejemplo, no pruebes que el constructor de una variante funcione.

*eduardo.acuna@unison.mx

2 Recursión generativa

Recordemos que para plantear un procedimiento recursivo se sugiere seguir la forma que tienen las entradas. Se descomponen los argumentos en sus componentes estructurales inmediatas y se procesan estas componentes. Si una de estas componentes inmediatas tiene la misma forma que la entrada, el procedimiento se dice ser *estructuralmente recursivo*.

La mayoría del código en el mundo se conforma de estos procedimientos, sin embargo, algunos problemas no pueden ser resueltos con este enfoque.

Para resolver estos otros problemas, podemos utilizar la *recursión generativa*, una forma de recursión que es estríctamente más poderosa que la recursión estructural.

El estudio de la recursion generativa es tan antiguo como las matemáticas y es usualmente llamado el estudio de los *algoritmos*. Las entradas de un algoritmo representan un problema. Un algoritmo tiende a reacomodar el problema en un conjunto de subproblemas, soluciona esos, y combina sus soluciones en una solución general. Comúnmente, algunos de estos nuevos problemas *generados* son de la misma forma que el problema en cuestión, en cuyo caso el algoritmo puede ser reutilizado para resolverlos (es decir, el algoritmo es recursivo), pero su recursión utiliza nuevos datos generados, en lugar de componentes estructuralmente inmediatas de la entrada.

Diseñar un procedimiento recursivo generativo es una actividad más ad hoc que diseñar un procedimiento recursivo estructural. Sin embargo, varios de los principios discutidos en clase pueden ser utilizados en el diseño de algoritmos. La parte central del diseño de algoritmos es la parte de "generación", que frecuentemente se refiere a dividir el problema, y descubrir una forma novedosa de dividir un problema suele requerir un alto grado de intuición, creatividad, familiaridad y entendimiento del problema.

2.1 Recursión sin estructura

Supon que estamos trabajando en un sistema para la edición colaborativa de código. Cada vez que una programadora modifica el buffer, el sistema transmite el contenido a las instancias que participan en la sesión.

Tu problema consiste en diseñar un procedimiento `bundle`, que prepara el contenido del buffer para su transmisión. Partes de una lista de cadenas unitarias (cadenas que solo contienen un carácter). La tarea del procedimiento es empaquetar estas cadenas unitarias en pedazos más grandes para producir una lista de cadenas (trozos) de cierta longitud (tamaño de trozo).

El predicado `unit-string?` nos permite saber si un objeto dado `x` es una cadena unitaria.

```
(define (unit-string? x)
  (and (string? x)
       (= (string-length x) 1)))
```

La entrada a nuestro procedimiento satisface el predicado `unit-string-list?` el cuál verifica que un objeto dado `x` sea una lista cuyos elementos son cadenas de tamaño 1.

```
(define (unit-string-list? x)
  (or (null? x)
      (and (pair? x)
           (string? (first x))
           (= (string-length (first x)) 1)
           (unit-string-list? (rest x)))))
```

Los procedimientos auxiliares `explode` e `implode`, inversas una de otra, permiten partir de una cadena de caracteres para obtener una lista de cadenas unitarias o viceversa.

```
(define (explode s)
  (unless (string? s)
    (error 'explode "esperaba una cadena, pero recibí: ~e" s))
  (map string (string->list s)))
```

```
(define (implode ls)
  (unless (unit-string-list? ls)
    (error 'implode "esperaba una lista de cadenas unitarias, pero recibí: ~e"
    ↪  ls))
  (apply string-append ls))
```

Con esta información, posiblemente ya te imaginas como comenzar la implementación:

```
;; empaqueta trozos de una lista de cadenas unitarias `s' en cadenas de tamaño
↪  `n'
(define (bundle s n)
  ....)
```

Veamos ejemplos de qué debe calcular el procedimiento `bundle`. Consideremos una lista de cadenas unitarias `(explode "abcdefgh")` y un tamaño de trozo `2`, para las cuáles se obtiene el resultado

```
'("ab" "cd" "ef" "gh")
```

Verifica en Racket el valor de la llamada a `explode`, no debe haber sorpresas.

Considera ahora un tamaño de trozo `3`, donde tenemos una letra sobrante ¿Qué debe producir `bundle`? Ya que el enunciado del problema no nos dice qué hacer en este caso, podemos imaginarnos dos escenarios válidos:

1. Produce `'("abc" "def" "g")`, es decir, considera la última letra como sobrante.
2. Produce `'("a" "bcd" "efg")`, es decir, considera la primera letra como sobrante.

¿Puedes pensar en otros?

Para simplificar la situación, elegirmos la primera opción como el resultado deseado, por lo que es conveniente escribir la prueba correspondiente:

```
(check-equal? (bundle (explode "abcdefg") 3)
              (list "abc" "def" "g"))
```

Las pruebas deben también describir qué ocurre en otros casos en la frontera. En este contexto, *la frontera* se refiere cuando a `bundle` se le pasa una lista que es demasiado pequeña para el tamaño de trozo dado:

```
(check-equal? (bundle '("a" "b") 3)
              (list "ab"))
```

También significa que debemos considerar qué pasa cuando a `bundle` se le da `'()`. Por simplicidad, elegimos `'()` como resultado deseado:

```
(check-equal? (bundle '() 3)
              '())
```

Otra alternativa pudo ser `'("")`, ¿qué otras pudieron ser convenientes?

Problema 2 Escribe todos los casos de prueba que consideres necesarios para `bundle`.

Veamos entonces cómo proceder con la implementación de `bundle`. De acuerdo a la técnica discutida en clases (el enfoque estructural), consideramos cuatro rutas:

```
;; s considerado atómico
;; n considerado compuesto
(define (bundle s n)
  (cond
    [(zero? n) ....]
    [else (.... (bundle s (sub1 n)) ....)]))
```

```
;; s considerado compuesto
;; n considerado atómico
(define (bundle s n)
  (cond
    [(empty? s) ....]
    [else (.... (bundle (rest s) n) ....)]))
```

Las primeras dos propuestas no van a funcionar, ya que la función debe desmenuzar cada uno de los dos argumentos, por lo que no podemos tratar alguno como atómico.

```
;; s considerado compuesto
;; n considerado compuesto
(define (bundle s n)
  (cond
    [(and (empty? s) (zero? n)) ....]
    [else (.... (bundle (rest s) (sub1 n)) ....)]))
```

Esta tercer propuesta se basa en la suposición que los dos argumentos se procesan al mismo tiempo, lo cuál está cerca de la solución, salvo que `bundle` claramente tiene que "resetear" el tamaño de trozo a su valor original en intervalos regulares.

```
;; s considerado compuesto
;; n considerado compuesto
(define (bundle s n)
  (cond
    [(and (empty? s) (zero? n)) ....]
    [(and (pair? s) (zero? n)) ....]
    [(and (empty? s) (positive? n)) ....]
    [else (.... (bundle s (sub1 n))
                .... (bundle (rest s) n) ....)]))
```

Esta última propuesta procesa los argumentos de forma independiente, desacoplando los argumentos demasiado ya que la lista y el tamaño de trozo deben procesarse juntos. Nos damos por vencidos, admitiendo que el enfoque estructural aparenta ser inútil para este problema.

Problema 3: Implementa los siguientes dos procedimientos:

1. Implementa el procedimiento `(take l n)` la cuál regresa una lista de los primeros `n` elementos de `l`, o cuantos elementos haya si `l` tiene menos de `n` elementos.
2. Implementa el procedimiento `(drop l n)` la cuál regresa una lista con los elementos de `l` excepto los primeros `n`, o la cadena vacía si `l` tiene menos de `n` elementos.

Entonces podemos implementar `bundle` de la siguiente manera:

```
(define (bundle s n)
  (cond
    [(null? s) null]
    [else
     (cons (implode (take s n))
           (bundle (drop s n) n))]))
```

Problema 4: Asegúrate que `bundle` pasa todas tus pruebas y que estas cubran todos los posibles casos.

Observa que el subproblema no se resuelve sobre la estructura inmediata de la lista, si no con el procedimiento `drop` la cuál elimina los primeros `n` elementos. Esta definición de `bundle` pudiera parecer inusual, pero las ideas subyacentes son intuitivas y no muy diferentes a los problemas que hemos trabajado. Si consideramos un tamaño de trozo `1`, entonces `bundle` se especializa a una definicón recursiva estructural. Además, `drop` garantiza la producción de una parte integral de la lista, no alguna versión arbitrariamente modificada.

Problema 5: La llamada `(bundle '("a" "b" "c") 0)` es un buen uso de `bundle`? ¿qué produce? ¿por qué?
Problema 6: Define el procedimiento `list->chunks`. Consume una lista `l` de valores arbitrarios y un natural `n`. El resultado es una lista de trozos de tamaño `n`. Cada trozo representa una sub-secuencia de elementos en `l`. Implementa `bundle` usando `list->chunks`.
Problema 7: Define `partition`. Toma una cadena `s` y un natural `n`. Produce una lista de trozos de cadenas de tamaño `n`. Para cadenas no vacías `s` y enteros positivos `n` se tiene que

```
(equal? (partition s n) (bundle (explode s) n))
```

es `#true`. No utilices esta igualdad como la definición de `partition`, utiliza `substring`.

El procedimiento `partition` es más cercano a lo que utilizaríamos en la práctica en el sistema descrito.

2.2 Recursión que ignora estructura

Considera el siguiente procedimiento de ordenamiento, toma una lista de números y regresa una lista con los mismos elementos pero en orden descendente.

```
(define (isort ls)
  (if (empty? ls)
      null
      (insert (first ls)
              (isort (rest ls)))))
```

Es el infame *insertion sort*, el procedimiento auxiliar `insert` toma un número `n` y una lista de números `ls` en orden descendente y regresa una lista con los elementos de `ls` pero que contempla también a `n` y manteniendo la misma condición de orden.

```
(define (insert n ls)
  (cond
    [(empty? ls)       (list n)]
    [(>= n (first ls)) (cons n ls)]
    [else              (cons (first ls) (insert n (rest ls)))]))
```

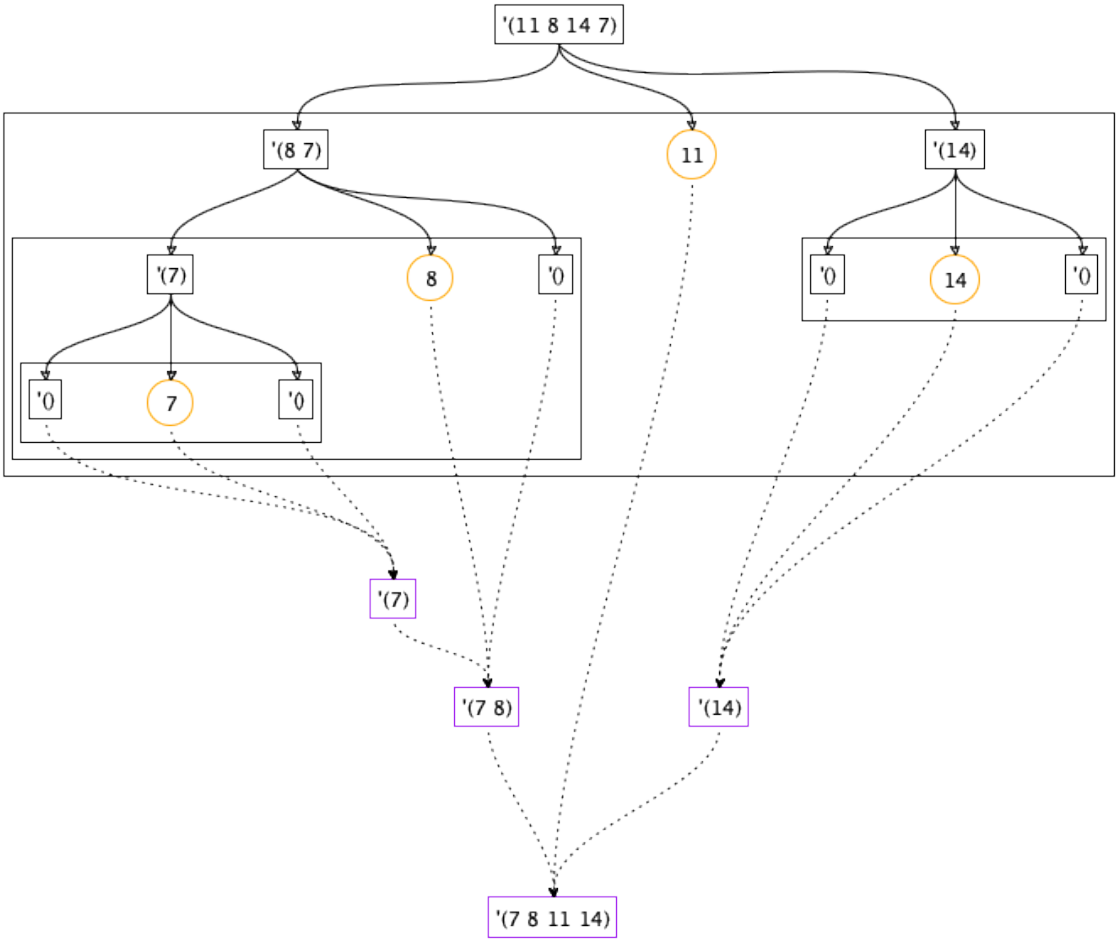
Observa que a pesar de utilizar un procedimiento auxiliar y reacomodar la estructura original de la lista, ambos procedimientos son recursivos estructurales. En otras palabras, `isort` es un procedimiento recursivo estructural que reprocesa el resultado de las recursiones naturales.

Problema 8: Modifica esos procedimientos para poder ordenar en orden ascendente cualquier tipo de valor a partir de un argumento adicional.

El algoritmo de C.A.R. Hoare conocido coloquialmente conocido como *quicksort* (aunque algunos lo recordarán como "aquel algoritmo que una vez implementé pero no recuerdo bien cómo es"), realiza el proceso de ordenamiento de una forma radicalmente diferente y se ha convertido en un ejemplo clásico de recursión generativa. El paso generativo utiliza la estrategia de "divide y vencerás". Esto es, divide las instancias no triviales del problema en dos problemas similares más pequeños, resuelve estos subproblemas y luego combina sus soluciones en una solución para el problema original. En el caso del algoritmo quicksort, el objetivo intermedio es dividir la lista en dos:

- una que contiene todos los elementos que son estríctamente menores que el primero;
- otra que contiene todos los elementos que son estríctamente mayores que el primero.

Entonces las dos listas más pequeñas son ordenadas usando quicksort y una vez que tengamos estas soluciones, sus resultados se componen con el primer elemento colocado a la mitad. Debido a su particular rol, al primer elemento en la lista se le conoce como el *elemento pivote*.



Problema 9: Dibuja un diagrama como el de la figura anterior pero para la lista `'(11 9 2 18 12 14 4 1)`.

Aquí está una implementación de `quicksort`:

```
(define (quicksort ls)
  (cond
    [(empty? ls) null]
    [else
     (define pivot (first ls))
     (append (quicksort (smallers ls pivot))
             (list pivot)
             (quicksort (largers ls pivot)))]))
```

Problema 10: Implementa los procedimientos `smallers` y `largers`, son recursivos estructurales. Uno debe regresar los elementos que son estríctamente menores al pivote y otro debe regresar los elementos que son estríctamente mayores al pivote.
Problema 11: Si la entrada a `quicksort` contiene varias repeticiones de un número, va a regresar una lista estrictamente más corta que la entrada. Responde el por qué y arregla el problema.

Problema 12: Modifica `quicksort` para ordenar listas con valores de cualquier tipo en cualquier orden aceptando un argumento adicional.

A pesar de que el algoritmo quicksort reduce rápidamente el tamaño del problema, en muchos casos utilizarlo es inapropiadamente lento. Es por eso que muchos programadores utilizan quicksort para problemas grandes y otros algoritmos para problemas suficientemente pequeños.

Problema 13: Implementa una versión de `quicksort` que utilice `isort` si la longitud de la entrada está por debajo de un umbral. Determina este umbral utilizando la función `time`, escribe el procedimiento que seguiste para encontrar este umbral.
Problema 14: Utiliza `filter` para definir `smallers` y `largers`.
Problema 15: Implementa `smallers` y `largers` como procedimientos locales internos en `quicksort`.

2.3 Adaptación de los principios de diseño

Debemos representar la información del problema como datos en el lenguaje de programación de nuestra elección. Elegir una representación para los datos de un problema influye cómo pensamos sobre el proceso computacional, por lo que es necesario un poco de planeación con anticipación, o bien, estar listos para explorar otros posibles caminos que contemplen otras representaciones.

También necesitamos una firma para el procedimiento y un enunciado de su propósito. Ya que el paso generativo no tiene conexión con la estructura inmediata de las entradas, el enunciado debe ir más allá del **qué** es lo que el procedimiento calcula y explicar también **cómo** es que se calcula el resultado.

Es útil explicar el **cómo** con ejemplos. Mientras que ejemplos en la perspectiva estructural meramente especificaban cuál salida debe producirse para algúna entrada, el propósito de los ejemplos en la perspectiva generativa es explicar la idea subyacente detrás del proceso computacional. Es muy importante poder comunicar el entendimiento de las ideas clave de los algoritmos a futuras lectoras del código.

La discusión previa sugiere una plantilla general para recursiones generativas. Se distinguen dos clases de problemas: aquellos que son *triviales* y aquellos que no lo son. Si un problema dado es trivialmente solucionable, el algoritmo produce la solución correspondiente. Por ejemplo, el problema de ordenar una lista vacía o con un elemento es trivial, pero ordenar una lista con varios elementos no es trivial. Para estos problemas no triviales, los algoritmos usualmente generan nuevos problemas del mismo tipo que el problema dado, los resuelven de forma recursiva y finalmente combinan sus soluciones en una solución general.

Basado en esta guía, todos los algoritmos tienen mas o menos la siguiente organización:

```
(define (recursión-generativa problema)
  (cond
    [(trivialmente-solucionable? problema)
     (determinar-solución problema)]
    [else
     (combinar-soluciones
      .... problema ....
      (recursión-generativa (generar-problema-1 problema))
      ....
      (recursión-generativa (generar-problema-n problema)))]))
```

Esta plantilla sugiere solo un plan general, no un código concreto. Cada pieza de la plantilla sirve para recordarnos pensar en los siguientes cuestionamientos:

- ¿Qué es trivialmente solucionable del problema?
- ¿Cómo se resuelven los problemas triviales?
- ¿Cómo el algoritmo genera nuevos problemas que son más fáciles de resolver que el original? ¿Hay más de un nuevo problema?
- ¿Es la solución del problema dado la misma que la solución de uno de los nuevos problemas?, ¿o debemos combinar las soluciones para crear una solución para el problema original? en ese caso, ¿necesitamos algo de la información del problema original?

Para definir el algoritmo como procedimiento, debemos expresar las respuestas a estos cuatro puntos como procedimientos y expresiones en términos de la forma de los datos y su representación en el lenguaje de programación.

Una vez que termines de implementar el algoritmo, es momento de probarlo. La idea en esta parte es descubrir y eliminar errores. Recuerda probar problemas en las fronteras, así como propiedades y características especificadas explícita o implicitamente sobre el problema, así como desiciones prácticas que hayas incorporado en el código que no sean especificadas en el problema.

Problema 16: Repasa tus soluciones a los problemas anteriores y comenta en cada función su firma, *qué* es lo que hace y *cómo* lo hace. En caso de ser recursión estructural, el *cómo* puede ser breve ya que la recursión debe naturalmente modelarse sobre la estructura de la entrada. Puedes inspirar tus firmas en los contratos de Racket, por ejemplo si el procedimiento `foo` toma un entero y una lista de símbolos, y regresa una cadena de caracteres la firma pudiéra ser

```
;; foo : integer? (listof symbol?) -> string
```

2.4 Terminación

La recursión generativa introduce un aspecto nuevo al cómputo realizado por las recursiones estructurales: la no-terminación. Un procedimiento como `bundle` pudiera nunca producir un valor o señalar un error para algunas entradas.

Contrasta esto con las recursiones estructurales. Cada recursión consume una subestructura inmediata de la entrada y ya que los datos se construyen de forma jerárquica, la entrada se encoge en cada paso de la recursión. Eventualmente llegamos a una subestructura atómica de la entrada y la recursión se detiene.

De acuerdo a la discusión previa sobre recursión generativa, estas pueden divergir. Un algoritmo puede generar nuevos problemas sin restricción alguna. Si restringimos a que los nuevos problemas sean "más pequeños" que el problema dado, la terminación si está garantizada. Pero imponer esta restricción complicaría el diseño de funciones como `bundle`. La teoría nos dice que esta restricción no es viable en general, si la imponemos van a existir funciones que no podremos implementar.

Es por esto que se incorpora al método para plantear soluciones recursivas generativas un paso adicional: la justificación de terminación. Se argumenta por qué cada llamada recursiva funciona correctamente en un problema que es más pequeño que el original, en algunos casos se debe demostrar que este es el caso. También cuando la implementación no termine en algunos casos debemos ejemplificarlos y caracterizar la clase de argumentos de entrada que causan la no terminación.

Problema 17: Modifica `bundle` para verificar que su entrada permita la terminación y que señale un error en caso contrario.

Problema 18: Considera la siguiente definición de `smallers`, uno de los procedimientos utilizados en `quicksort`, responde en qué puede fallar al utilizar esta versión modificada en el procedimiento de ordenamiento.

```
(define (smallers l n)
  (cond
    [(empty? l) '()]
    [else (if (<= (first l) n)
              (cons (first l) (smallers (rest l) n))
              (smallers (rest l) n))]))
```

2.5 Tomando decisiones

Cuando interactuamos con una función `f` que ordene listas de números, es imposible saber si `f` es `isort` o `quicksort` (o algúna otra) a partir de las entradas recibidas y salidas producidas. Ambas funciones se comportan de forma observablemente equivalente. Entonces, ¿cuál debemos incluir en nuestros lenguajes de programación?, o más general, cuando podemos diseñar una función utilizando recursión estructural y recursión generativa, ¿cómo podemos elegir cuál implementar?

Para ilustrar las consecuencias de esta decisión, consideramos el clásico problema de encontrar el máximo común divisor de dos enteros positivos, llamado `gcd` (por su nombre en inglés *Greatest Common Divisor*).

Considera los siguientes dos procedimientos.

```
(define (gcd-structural n m)
  (define (find-largest-divisor k)
    (cond [(= i 1) 1]
          [(= (remainder n i) (remainder m i) 0) i]
          [else (find-largest-divisor (- k 1))]))
  (find-largest-divisor (min n m)))
```

El procedimiento `gcd-generative` toma en cuenta la observación de que para dos enteros positivos ℓ el más grande y s el más pequeño, el máximo común divisor es igual al máximo común divisor de s y el residuo de ℓ dividido entre s, es decir

```
(gcd l s) == (gcd s (remainder l s))
```

```
(define (gcd-generative n m)
  (define (find-largest-divisor max min)
    (if (= min 0)
        max
        (find-largest-divisor min (remainder max min))))
  (find-largest-divisor (max n m) (min n m)))
```

Problema 19: Describe con tus propias palabras cómo funciona `find-largest-divisor` de `gcd-structural`. Responde por qué comienza desde `(min n m)`.

Problema 20: Describe con tus propias palabras cómo funciona `find-largest-divisor` de `gcd-generative`.
Problema 21: Utiliza la función `time` para determinar cuál de las dos implementaciones es más eficiente, escribiendo tu respuesta con los tiempos de ejecución obtenidos con ambos procedimientos para valores "pequeños", "medianos" y "grandes". Justifica qué valores usaste en cada una de estas mediciones y por qué los consideraste de ese "tamaño".
Problema 22: Piensa y describe por qué no siempre es la mejor opción elegir el procedimiento más eficiente en tiempo de ejecución. Utiliza criterios que no sean el de "eficiencia".

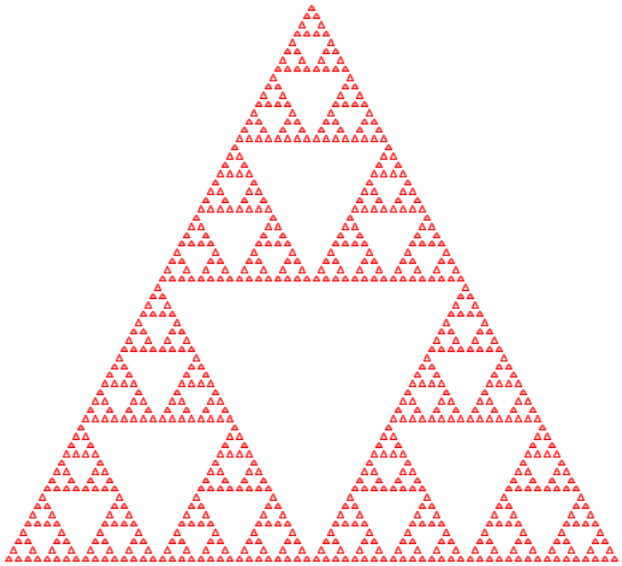
2.6 El Pilón

A continuación se muestra una definición que hace uso de las bibliotecas `pict` y `racket/draw` para construír la imagen de un triángulo equilátero. Toma un tamaño de lado `side`, el ancho de la línea `width` y el color de la línea `color` para regresar un objeto de tipo `pict`.

```
(define (triangle side width color)
  (define w side)
  (define h (* side (sin (/ pi 3))))
  (define (draw-it ctx dx dy)
    (define prev-pen (send ctx get-pen))
    (define path (new dc-path%))
    (send ctx set-pen (new pen% [width width] [color color]))
    (send path move-to 0 h)
    (send path line-to w h)
    (send path line-to (/ w 2) 0)
    (send path close)
    (send ctx draw-path path dx dy)
    (send ctx set-pen prev-pen))
  (dc draw-it w h))
```

Otras figuras geométricas simples pueden ser definidas de esta forma, sin embargo, lo interesante no es que podamos dibujar un triángulo en la pantalla. Una vez que se tiene la imagen del triángulo, se puede combinar con otras imágenes de acuerdo a una posición horizontal o vertical alineado hacia los extremos o el centro. Con esto podemos definir un procedimiento recursivo que genere la imagen de un fractal, estructuras autosimilares en múltiples escalas.

Por ejemplo, podemos generar la imagen del triángulo de Sierpinski tomando tres copias de un triángulo de Sierpinski de la mitad del tamaño y acomodándolas en una configuración donde dos de ellas se encuentran lado a lado y la tercera se encuentra centrada sobre las anteriores dos, obteniendo una figura como la siguiente:



El siguiente procedimiento nos permite producir esta figura.

```
(define (sierpinski side)
  (cond [(<= side 4) (triangle side 1 "red")]
        [else
         (define half (sierpinski (/ side 2)))
         (vc-append half (hc-append half half))]))
```

Problema 23: Implementa un procedimiento que genere otro fractal, toma en consideración la discusión de esta tarea, caracteriza el tipo de recursividad que utilizaste y justifica la terminación de tu procedimiento.
