PrePLAI: Scheme y Programación Funcional

Éric Tanter

September 25, 2014

last updated: Thursday, September 25th, 2014

Este mini-curso tiene como objetivo entregarle las nociones básicas para programar en Scheme siguiendo buenas prácticas de programación funcional. Esta materia es indispensable para luego poder seguir el curso de lenguajes, que sigue el libro PLAI de Shriram Krishnamurthi.

Se usa el lenguaje Racket, un super Scheme con variadas y poderosas librerías, y su ambiente de desarrollo, DrRacket.

Agradezco comentarios, sugerencias y correcciones; ¡no dude en contactarme por email!

Agradecimientos: Gracias a los miembros del laboratorio PLEIAD y del curso de Lenguajes de Programación de la Universidad de Chile por sus correcciones y sugerencias, en particular a Javiera Born, Ismael Figueroa y Rodolfo Toledo.

1 Elementos Básicos

El ambiente de desarrollo (IDE) DrRacket divide inicialmente el espacio en dos partes, una para definiciones, y otra para interactuar con el lenguaje directamente, al estilo de un terminal Unix. Usaremos la forma interactiva para empezar a explorar el lenguaje.

1.1 Tipos primitivos

Números, incluyendo fracciones y números imaginarios:

```
> 1
1
> -3
-3
> 4.02
4.02
> 6.02e+23
6.02e+23
> 1+2i
1+2i
> 4/3
4/3
```

Booleanos:

```
> #t
#t
> #f
#f
```

Strings (incluyendo caracteres Unicode):

```
> "hola"
"hola"
> "hola \"mundo\" feliz"
"hola \"mundo\" feliz"
> "λx:(μα.α→α).xx"
"λx:(μα.α→α).xx"
```

Símbolos:

```
> 'hola
'hola
```

Un símbolo es un valor atómico. Por ende, determinar la igualdad de dos símbolos es barato (tiempo constante), a diferencia de una comparación de strings (lineal).

Otros tipos primitivos incluyen caracteres, bytes y byte strings, void, undefined, y estructuras de datos como pares, listas, cajas, tablas de hash, y vectores. Veremos algunos más adelante.

1.2 Usar funciones predefinidas

Scheme es un lenguaje con sintáxis prefijada con paréntesis. El primer elemento después del paréntesis "(" es la función a aplicar, y todo lo demás hasta el paréntesis ")" correspondiente son los argumentos (separados por espacios). Por ejemplo, para aplicar funciones aritméticas (que reciben un número variable de argumentos):

```
> (+ 1 2)
3
> (* 2 4 5)
40
> (- 1 1/4)
3/4
```

Eso significa que los paréntesis en Scheme tienen un significado (¡y muy importante!). Las pueden ver como el equivalente de las paréntesis de aplicación de funciones y procedimientos en otros lenguajes (por ejemplo `foo(1)` en C se escribe `(foo 1)` en Scheme).

Y similarmente para otras funciones matemáticas y lógicas:

```
> (+ 1 (- 3 4))
0
> (sqrt -1)
0+1i
> (or (< 5 4)
      (equal? 1 (- 6 5)))
#t
> (and (not (zero? 10))
       (+ 1 2 3))
6
```

Como se puede apreciar en el último ejemplo, en Scheme, todo lo que no es `#f` es verdadero.

Manipulación de strings:

```
> (string-append "ho" "la")
"hola"
> (string-length "hola mundo")
```

```
10
> (substring "Apple" 1 3)
"pp"
> (string->symbol "Apple")
'Apple
```

Hay varias formas de imprimir, por ejemplo `printf`:

```
> (printf "hola~n")
hola

> (printf "hola ~a ~s~n" "mundo" "feliz")
hola mundo "feliz"
```

Scheme es un lenguaje seguro (¡no hay *segmentation faults*!). Al invocar una función con argumentos del tipo equivocado, se genera un error:

```
> (+ 1 "hola")
+: contract violation
  expected: number?
  given: "hola"
  argument position: 2nd
  other arguments...:
   1
```

Existe un dialecto de Racket, llamado Typed Racket, que es estáticamente tipeado. Puede encontrar información en la documentación.

El mensaje de error explica claramente lo que pasó. Es importante notar que el error se reporta en tiempo de ejecución. Scheme no es un lenguaje estáticamente tipeado, al igual que JavaScript y Python, entre otros.

Ejercicio: imprima (con `printf`) la raíz cuadrada de `-1/4`.

1.3 Condicionales

El clásico:

```
> (if (> 4 10)
      "hola"
      "chao")
"chao"
> (if (> 12 10)
      "hola"
      "chao")
```

```
"hola"
```

Los `if`s se pueden anidar:

```
> (if (> 2 3)
      (printf "hola")
      (if (> 6 5)
          (printf "chao")
          #f))
chao
```

Pero es más cómodo usar `cond` para este tipo de condicional múltiple:

El uso de paréntesis cuadrados es totalmente opcional, es solamente con fines de legibilidad. Veremos otros usos de esos paréntesis conformemente a las buenas prácticas de la comunidad Racket a lo largo del curso.

```
> (cond [(> 2 3) (printf "hola")]
        [(> 6 5) (printf "chao")]
        [else #f])
chao
```

1.4 Definir identificadores

Se puede definir identificadores tal que queden globalmente disponibles. Esas definiciones se pueden hacer en la parte de definiciones del IDE:

```
(define MAX 100)
```

Después de hacer click en el botón "Run", el identificador `MAX` queda enlazado con el valor `100`, disponible para interactuar:

```
> MAX
100
> (< 25 MAX)
#t
```

También se puede definir un identificador en la parte de interacciones del IDE. De esta manera, el identificador queda definido para toda la sesión de interacción. Hacer click en "Run" inicia una nueva sesión, en la cual el identificador no estará definido.

```
> (define x 10)

> (+ x 1)
11
```

Como hemos visto, la sintáxis de los identificadores en Scheme, incluyendo para nombres de funciones, es bastante liberal. Casi cualquier secuencia de caracteres que no sea un espacio es un identificador: `+`, `number?`, `pass/fail`, `set!`, `λ=>α`, etc. son identificadores totalmente válidos. En realidad, solamente los paréntesis (cualquier forma) y los caracteres `,.;''|\` son especiales (y obviamente, las secuencias de caracteres que forman los números).

```
> (define !λ=>-α/β?☺ "weird")

> !λ=>-α/β?☺
"weird"
```

Se puede también introducir identificadores con alcance local con `let`:

```
> x
10
> (let ([x 3]
        [y 2])
    (+ x y))
5
> x
10
> y
y: undefined;
 cannot reference undefined identifier
```

El identificador `x` dentro del `let` esconde el `x` definido globalmente. Pero una vez que uno sale del `let`, el identificador original sigue disponible e inalterado. Similarmente, `y` es introducido solamente en el cuerpo del `let`, de manera que no está definido una vez fuera del cuerpo del `let`.

Notar que `let` no permite usar un identificador en el cálculo de otro identificador introducido después en el mismo `let`. Para esto hay que usar `let*`, que es equivalente a `let`s anidados:

```
> (let ([a 3]
        [b (+ a a)])
    b)
a: undefined;
 cannot reference undefined identifier
> (let* ([a 3]
         [b (+ a a)])
    b)
6
```

Ejercicio: introduzca localmente dos identificadores locales, asociados a números, y retorne el máximo.

1.5 Definir funciones

Además de las funciones existentes, se pueden definir funciones propias:

```
(define (double x)
  (+ x x))

> (double 2)
4
```

También se puede definir una función directamente en la parte de interacciones del IDE:

```
> (define (foo x)
    (if (< x 10)
        (printf "menor")
        (printf "mayor")))

> (foo 4)
menor

> (foo 11)
mayor
```

Más adelante veremos herramientas metodológicas para §4 “Definir Funciones”.

Ejercicio: defina la función `(sum a b)` que suma sus dos parámetros.

Ejercicio: defina la función `(pick-random x y)` que retorna en forma aleatoria uno de sus argumentos. (La función `(random)` genera un número aleatorio entre `0` y `1`).

1.6 ¿Dónde encuentro información sobre...?

La mejor manera de explorar Racket es usar el Help Desk (menú Help en DrRacket), que abre la documentación de Racket en local (la documentación también está disponible en la web). Las partes más importantes son el Racket Guide para una explicación al estilo "tutorial", y la Racket Reference para una descripción exhaustiva del lenguaje Racket y de todas las funciones y librerías incluidas.

¡La documentación de Racket es sumamente útil y completa! Acostúmbrese a usarla desde ya.

2 Estructuras de Datos

Aparte de los datos primitivos simples que hemos visto, Scheme soporta múltiples estructuras compuestas, como pares, listas, y vectores.

2.1 Pares

La estructura más básica es el par, que pega dos valores juntos.

```
> (cons 1 2)
'(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```

Ejercicio: defina la función `(pair-add1 p)` que recibe un par de números y retorna un nuevo par dónde los dos elementos fueron incrementados en `"1"`.

2.2 Listas

Toda estructura se puede codificar con pares. En particular, las listas pueden ser construidas encadenando pares (con `cons`), usando `empty` (o `null`, o `'()`) como último elemento.

```
> empty
'()
> (cons 2 empty)
'(2)
> (cons 1 (cons 2 (cons 3 empty)))
'(1 2 3)
```

Es posible crear listas en forma más conveniente con `list`:

```
> (define l (list 1 2 3 4 5 6))

> l
'(1 2 3 4 5 6)
```

Las listas se concatenan con `append`:

```
> (append (list 1 2 3) (list 4 5 6))
'(1 2 3 4 5 6)
```

Lo cual es muy distinto de `cons`:

```
> (cons (list 1 2 3) (list 4 5 6))
'((1 2 3) 4 5 6)
```

Como una lista es en realidad unos pares encadenados, se puede usar `car` y `cdr` (y sus combinaciones) para acceder a cualquier elemento de una lista:

```
> (car l)
1
> (cdr l)
'(2 3 4 5 6)
> (car (cdr l))
2
> (cadr l)
2
> (cadddr l)
4
```

Existen funciones con nombres más amigables para hacer lo anterior:

```
> (first l)
1
> (rest l)
'(2 3 4 5 6)
> (second l)
2
> (fourth l)
4
```

Se habrá dado cuenta que pares y listas aparecen con un ' (quote) en los resultados del evaluador. Recuerde que ya hemos visto que ' se usa para los símbolos:

```
> x
x: undefined;
 cannot reference undefined identifier
> 'x
'x
```

Resulta que quote en realidad dice a Scheme: "no evalúa lo que sigue, consideralo como un dato". Notará que la sintáxis de una lista es muy parecida a la de una aplicación de función. De hecho:

```
> (1 2 3)
application: not a procedure;
 expected a procedure that can be applied to arguments
  given: 1
  arguments...:
   2
   3
```

Scheme se queja que `1` no es una función. O sea, `(1 2 3)` realmente significa aplicar la función `1` (que no existe) a los argumentos `2` y `3`, al igual que en `(+ 2 3)`. Para considerar `(1 2 3)` como una lista literal, no hay que evaluar:

```
> '(1 2 3)
'(1 2 3)
> (second '(1 2 3))
2
```

Una lista literal (es decir, creada con quote) es distinta a una lista creada con `list`, porque al aplicar `list`, se evaluan sus argumentos:

```
> (list 1 2 (+ 1 2))
'(1 2 3)
> '(1 2 (+ 1 2))
'(1 2 (+ 1 2))
```

Otro ejemplo:

```
> (second '(define (inc x) (+ 1 x)))
'(inc x)
```

Esa facilidad que tiene Scheme de representar programas como datos abre puertas fascinantes como generación de programas, extension de lenguajes, reflexión, etc. Solamente veremos un poco de eso en la última parte del curso donde usaremos macros para definir sistemas de programación con objetos dentro de Scheme mismo. ¡Paciencia! ☺

Finalmente, existen muchas funciones predefinidas para manipular (pares y) listas. Por ejemplo, `length` para obtener el largo de una lista, `list-ref` para acceder a una posición dada, `reverse` para invertir una lista, etc. También existen funciones para iterar, filtrar y buscar.

En Racket, las listas y los pares son immutables. Existen versiones mutables de pares y listas, aúnque no las usaremos en el curso.

Ejercicio: defina la función `list-pick-random` que retorna un elemento al azár dentro de una lista. (La función `(random k)` retorna un número aleatorio entre `0` y `k-1`.)

2.3 Vectores

Un vector en Scheme es (casi) como un arreglo en C. La diferencia importante es que el acceso a un vector es seguro: si el índice es fuera del rango del vector, se lanza un error. La mayor diferencia entre arreglos y listas es que el acceso a cualquier posición se hace en tiempo constante (con `vector-ref`). Por comparación, dado que una lista es una estructura encadenada, el acceso a cualquier posición se hace en tiempo lineal.

```
> (define v (vector 1 2 3))

> (vector-ref v 2)
3
```

Por convención, las funciones que modifican de manera imperativa el estado del programa tienen un nombre que termina con el símbolo `!`.

Otra característica de los vectores es que son, por lo general, mutables:

```
> (vector-set! v 2 "hola")

> v
'#(1 2 "hola")
```

La única excepción son los vectores literales, que son immutables:

```
> (define v2 #(1 2 3))

> (vector-set! v2 0 -10)
vector-set!: contract violation
  expected: (and/c vector? (not/c immutable?))
  given: '#(1 2 3)
  argument position: 1st
  other arguments...:
   0
   -10
```

Finalmente, es posible convertir vectores en listas y vice-versa:

```
> (vector->list v2)
'(1 2 3)
> (list->vector '(1 2 3))
'#(1 2 3)
```

Ejercicio: defina la función `vector-set-random!` que cambia un elemento al azár dentro de un vector por un valor dado.

3 Programar con Funciones

Hemos visto cómo usar funciones, pero no nos hemos preocupado de entender qué son las funciones en Scheme. Por ejemplo:

```
> +
#<procedure:+>
```

El mismo programa en C o Java daría un error, porque en estos lenguajes, las funciones (procedimientos, métodos) no son valores. La interacción anterior demuestra que en Scheme, al igual que en muchos lenguajes (ML, Haskell, Scala, Python, Clojure, etc.), las funciones sí son valores. Esta característica es una de las perlas de la programación funcional, dando luz a formas muy poderosas de componer programas.

3.1 Funciones parametrizadas por funciones

Dado que las funciones son valores, una función puede recibir a otra función como parámetro. Esto permite abstraer ciertos procesos. Por ejemplo, si uno quiere agregar `1` a todos los elementos de una lista, puede recorrer la lista y construir una nueva lista con los elementos incrementados. De igual manera, si uno quiere convertir todos los números de una lista en strings, tiene que recorrer la lista y aplicar la conversión a cada elemento. Este recorrido puede ser parametrizado por la función a aplicar a cada elemento. La función que hace esto se llama `map`:

```
(define my-list '(1 2 3 4 5))

> (map add1 my-list)
'(2 3 4 5 6)
> (map number->string my-list)
'("1" "2" "3" "4" "5")
```

Otro procesamiento típico de una lista de elementos es calcular un valor consolidado a partir de todos los elementos de la lista. Por ejemplo, sacar la suma de los elementos de una lista. O determinar cuál es el máximo de la lista.

```
> (foldl + 0 my-list)
15
> (foldl + 100 my-list)
115
> (foldl max 0 my-list)
5
```

`foldl` reduce una lista a un valor usando la función (de dos parámetros) que uno le pasa como primer parámetro. Parte aplicando esta función al primer elemento de la lista, usando como valor inicial el segundo parámetro (`0` en el primer ejemplo). El resultado así obtenido es usado con el segundo elemento de la lista, y así sucesivamente. Como este recorrido procede de izquierda a derecha, se llama `foldl`, por `fold` (doblar) `left`. Existe la función `foldr` que procede al revés, partiendo con el último elemento de la lista.

`fold`(`l`/`r`) también es conocido como `reduce` en otros lenguajes (más precisamente, en general `reduce` es como `foldl` donde el valor inicial es el primer elemento de la lista). La composición de `map` y `reduce` es la base del framework MapReduce de Google, que es clave para el procesamiento escalable de consultas sobre grandes cantidades de datos.

Para entender bien la diferencia entre `foldl` y `foldr`, es instructivo detallar paso a paso la evaluación de los siguientes ejemplos:

```
(foldl + 0 '(1 2 3))
↦ (+ 3 (+ 2 (+ 1 0)))
↦ (+ 3 (+ 2 1))
↦ (+ 3 3)
↦ 6

(foldr + 0 '(1 2 3))
↦ (+ 1 (+ 2 (+ 3 0)))
↦ (+ 1 (+ 2 3))
↦ (+ 1 5)
↦ 6
```

Ejercicio: detalle paso a paso la evaluación de las siguientes expresiones, explicando así el resultado obtenido:

```
> (foldl cons '() my-list)
'(5 4 3 2 1)
> (foldr cons '() my-list)
'(1 2 3 4 5)
```

Hay muchas más funciones que reciben otras funciones. Las posibilidades son infinitas. Un ejemplo muy práctico también es seleccionar en una lista todos los elementos que cumplen cierta condición. Esto se hace con `filter`, pasandole un predicado como parámetro. También existe `sort`, parametrizado por una función de comparación.

```
> (filter even? my-list)
'(2 4)
> (sort my-list >)
'(5 4 3 2 1)
```

Programar de esa manera es mucho más declarativo y conciso que la programación imperativa tradicional. La clave está en el poder de abstracción que permite tener funciones (programas) como parámetros de otras funciones.

Ejercicio: estudie la documentación de las funciones para iterar, filtrar y buscar, y úselas sobre listas de su elección.

Ejercicio: defina, usando `foldl`, la función `(reject lst pred)` que retorna la lista de los elementos de la lista `lst` que no cumplen con el predicado `pred`.

3.2 Funciones anónimas

Como hemos visto, existe la función `add1` que incrementa su parámetro:

```
> add1
#<procedure:add1>
> (add1 2)
3
```

Si bien esta función es útil, por ejemplo para pasarla como parámetro de `map`, uno podría preguntarse: ¿existe `add2`?

```
> add2
add2: undefined;
 cannot reference undefined identifier
```

No existe. Claramente se puede definir simplemente:

```
(define (add2 x) (+ x 2))

> (map add2 '(1 2 3))
'(3 4 5)
```

Similarmente, si necesitamos incrementar de `124` a todos los elementos de una lista, podemos definir `add124`. Sin embargo, tener que darle un nombre a tal función es un poco artificial, dado que es muy probable que no la necesitemos más adelante.

En Scheme, uno puede definir una función anónima usando la forma sintáctica `lambda`. Por ejemplo:

```
> (lambda (x) (+ x 124))
#<procedure>
```

Como vemos, una función anónima así definida es también un valor. Por lo que se puede pasar como parámetro (usamos la notación λ (ctrl-\) por estética):

```
> (map (λ (x) (+ x 124)) '(1 2 3))
'(125 126 127)
> (foldl (λ (x y) (and x y)) #t '(a b c d #f e))
#f
```

Ahora que sabemos crear funciones anónimas, podemos entender que la forma para definir funciones que vimos antes:

```
(define (foo x) x)
```

no es nada más que azúcar sintáctico para la definición de un identificador asociado a la lambda correspondiente:

```
(define foo (λ (x) x))
```

También se puede introducir funciones con nombre local:

```
> (let ([f (λ (x) x)])
    (f 10))
10
> f
f: undefined;
 cannot reference undefined identifier
```

Para introducir una función local que es recursiva (es decir, que se llama a sí misma), hay que usar `letrec`:

```
> (letrec ([fact (λ (n)
                   (if (< n 2) 1
                       (* n (fact (sub1 n))))))])
    (fact 10))
3628800
```

(¿Qué pasa si uno usa `let` y no `letrec`?)

Ejercicio: use la función `findf` para buscar el primer elemento múltiplo de `13` en una lista dada (retorna `#f` si no hay); defina el predicado como una función anónima. Puede usar la función `modulo`.

3.3 Funciones que producen funciones

Tener la posibilidad de crear funciones anónimas en cualquier parte permite hacer algo sumamente poderoso: definir funciones que crean otras funciones. O sea, generadores de

funciones. Por ejemplo, volviendo a nuestras funciones `add1`, `add2`, `add124`, vemos que sus definiciones son muy similares. Lo único que cambia es el número usado en la suma con el parámetro.

Podemos ahora definir una única función `addn` que es capaz de fabricar una de esas funciones:

```
(define (addn n)
  (λ (x)
    (+ x n)))
```

Con `addn` podemos crear y nombrar los variantes que queremos:

```
(define add10 (addn 10))
(define add124 (addn 124))

> (add10 12)
22
> (map add124 my-list)
'(125 126 127 128 129)
```

Y también podemos crear una función y usarla directamente, sin nombrarla:

```
> (map (addn 20) my-list)
'(21 22 23 24 25)
```

Aquí va un último ejemplo:

```
(define (more-than n)
 (λ (m)
   (> m n)))

> (filter (more-than 10) '(13 2 31 45 9 10))
'(13 31 45)
> (filter (more-than 20) '(13 2 31 45 9 10))
'(31 45)
```

Esa capacidad de crear dinámicamente funciones similares pero con atributos propios (por ejemplo, el valor de `n` en las distintas funciones creadas por `addn`) es muy similar a la creación de objetos a partir de clases en Java. Volveremos a explorar esta conexión en el OOPLAI.

Ejercicio: defina la función `(negate pred)` que retorna un predicado que es la negación de `pred`.

Ejercicio: use `negate` y `filter` para definir la función `(reject lst pred)` (que vimos anteriormente) de manera más concisa aún.

Esta técnica se llama currificación (*currying*)

Ejercicio: defina la función `(curry f)` que, dado una función de tipo (A B $\rightarrow$ C), retorna una función equivalente de tipo A $\rightarrow$ (B $\rightarrow$ C). Es decir, permite que una función que toma dos parámetros los tome uno por uno. Ejemplo

```
> (define cons1 ((curry cons) 1))

> (map cons1 '(a b c d))
'((1 . a) (1 . b) (1 . c) (1 . d))
```

Esta técnica se llama *memoization*.

Ejercicio: defina la función `(memoize f)` que, dado una función de tipo (A $\rightarrow$ B), retorna una función equivalente que mantiene un cache <*argumento* $\rightarrow$ *valor*>. Para el cache, use una tabla de hash.

4 Definir Funciones

4.1 Metodología general

Cuando tiene que implementar una función que cumple cierto objetivo, es importante seguir los siguientes pasos, *en este orden*:

- *Entender* lo que la función tiene que hacer.
- Escribir su *contrato*, o sea, su tipo (cuántos parámetros, de qué tipo, qué retorna).
- Escribir la *descripción* de la función en comentario.
- Escribir los *tests* unitarios asociados a esta función, sobre los distintos posibles datos de entradas que pueda tener (casos significativos).
- **Finalmente**, *implementar* el cuerpo de la función.

Es crucial dejar la implementación para lo último, especialmente, para después de definir los tests. Primero, escribir los tests le obliga a entender bien concretamente cómo se va a usar la función que necesitan definir. Segundo, hacer esto antes de la implementación asegura que sus tests están diseñados para evaluar la función en base a sus requerimientos, no en base a lo que acaba de implementar.

Para los tests, usaremos las funciones `test` y `text/exn` provistas por el lenguaje PLAY (ver §5.2 “Tests”).

```
; double-list :: (Listof Number) -> (Listof Number)
; duplica los elementos de la lista
(define (double-list l)
  (map (λ (x) (* x 2)) l))

(test (double-list '(1 2 3)) '(2 4 6))
(test (double-list empty) empty)
```

Aquí dos tests bastan, porque cubren el espectro de posibilidades (lista vacía, y lista no vacía). Una variante del mismo ejemplo maneja el caso donde el argumento no es una lista:

```
; double-list :: (Listof Number) -> (Listof Number)
; duplica los elementos de la lista
; lanza un error si el parámetro no es una lista
(define (double-list l)
  (if (list? l)
      (map (λ (x) (* x 2)) l)
      (error "ERROR: not a list!!!!")))
```

Note que el contrato de `double-list` indica que se tiene que pasar una lista de números. En el curso, asumiremos que los clientes respetan los contratos, por lo que no será necesario hacer este tipo de validación.

```
(test (double-list '(1 2 3)) '(2 4 6))
(test (double-list empty) empty)
(test/exn (double-list 4) "not a list")
```

Aquí hemos agregado un test que asegura que la función reacciona adecuadamente cuando uno pasa un argumento que no es una lista.

4.2 Procesar estructuras recursivas

4.2.1 Inducción

De sus clases de matemática, ha aprendido una forma poderosa de establecer propiedades: el razonamiento por inducción. Para demostrar P(n) para cualquier n natural, basta mostrar:

- P(0): P es cierto para 0 (el caso inicial puede ser distinto según los casos).
- P(n) $\Rightarrow$ P(n+1): asumiendo P cierto para un n dado, demostrar que es cierto para el siguiente.

Resulta que hay una conexión directa muy fuerte entre esta herramienta lógica y la programación, especialmente cuando se trata de definir funciones que procesan datos recursivos.

Primero, hay que entender que los números naturales pueden ser definidos como datos en forma recursiva también:

⟨*nat*⟩ ::= `0`
| `(` `add1` ⟨*nat*⟩ `)`

Esta notación se llama BNF (Backus-Naur Form). Una especificación BNF es un sistema de reglas de derivación. Se usa mucho para describir gramáticas de lenguajes formales, pero no solamente.

O sea, un natural es o `0`, o el sucesor de un natural. El conjunto de los números naturales es el conjunto mínimo que cumple con ambas reglas (mínimo porque no contiene más elementos que aquellos obtenidos con esas reglas).

Por ende, el principio de inducción descrito anteriormente corresponde a esta definición: para establecer una propiedad sobre un natural `n`, hay que demostrarla para `0` y para el caso `(add1 n)`. La occurrencia de ⟨*nat*⟩ en la segunda regla de derivación se conecta directamente con la hipótesis de inducción. O sea, para demostrar la propiedad para `(add1 n)`, tenemos el derecho de asumir la propiedad para `n`. De manera equivalente, para demostrar la propiedad para `n`, tenemos el derecho de asumir la propiedad para `(sub1 n)`.

4.2.2 Inducción y programación recursiva

Lo anterior se aplica directamente a la programación. Por ejemplo, si queremos determinar el factorial de `n`, podemos razonar por inducción, o sea, programar en forma recursiva:

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

Note como el cuerpo de la función sigue el mismo patrón que la gramática de ⟨*nat*⟩. ¡Cualquier función que procesa un natural puede ser definida siguiendo ese patrón! El patrón es:

```
(define (f n)
  (if (zero? n) ... ; caso base                          ...
      (f (sub1 n)) ...)) ; paso inductivo
```

La llamada recursiva a `(f (sub1 n))` corresponde al uso de la hipótesis de inducción (asumiendo el resultado de `f` para `n-1`). En la gramática, se nota la llamada recursiva porque ⟨*nat*⟩ esta definido en función de ⟨*nat*⟩.

Otro ejemplo, una función que determina si un número es par:

```
(define (even? n)
  (if (zero? n)
      #t
      (not (even? (sub1 n)))))
```

4.2.3 Inducción estructural y metodología

La idea de la inducción se generaliza a estructuras arbitrarias más complejas que `<nat>`. Se llama inducción estructural. A partir del momento en que pueden describir la gramática BNF de la estructura que su función esta procesando, ¡ya pueden definir "ciegamente" el template del cuerpo de la función!

Esto significa que el último paso de la metodología general que vimos anteriormente se descompone en dos:

- Entendiendo la estructura del dato que procesa la función, escribir el patrón recursivo correspondiente.
- Completar el cuerpo de la función, siguiendo el análisis por casos.

Gracias a esta metodología refinada, pueden tener la definición casi completa de su función (documentación, contrato, tests, y patrón recursivo) ¡sin ni siquiera haber empezado a programar lo específico de la función!

4.2.4 Pattern matching sobre estructuras recursivas

Una técnica tradicional para trabajar con estructuras de datos (incluyendo las recursivas) es el reconocimiento de patrones, más conocido en inglés como *pattern matching*. La idea es definir un caso para cada constructor de la estructura de datos, siguiendo así la gramática del mismo.

Racket tiene un soporte muy flexible para pattern matching sobre valores arbitrarios. Esta operación se realiza con la expresión `match`. Su forma de uso es:

```
(match target-expr
  [pattern expr ...+] ...)
```

Es decir, una expresión `match` está conformada por la expresión que queremos procesar, `target-expr`, y una o más *cláusulas* que corresponden a un patrón y una expresión. La semántica es similar a la de `cond`: se intenta hacer match de `target-expr` con cada patrón en forma secuencial, y se evalúa la expresión `expr` de la primera cláusula donde el matching es exitoso.

En la documentación de Racket se indica el uso básico de `match`. En general, cualquier valor literal puede ser matcheado:

```
> (match "hola"
    ["hola" 1]
    ["chao" 2])
1


> (match (and #t #f)
    [#t "bueno"]
    [#f "malo"])
"malo"
```

En caso que ningún patrón coincida con la expresión, se genera un error:

```
> (match 1
    [2 #t]
    [3 #f])
match: no matching clause for 1
```

Para definir un caso por defecto al final de la lista de patrones, puede usar el patrón `[else ...]` que siempre realiza matching (es posible usar cualquier identificador, pero por convención usaremos `else`). Por ejemplo:

```
> (match 1
    [2 #t]
    [3 #f]
    [else #t])
#t
```

Además es posible hacer matching sobre los constructores de pares, listas y vectores; y sobre los constructores de estructuras arbitrarias. Racket provee un expresivo lenguaje de patrones, que puede consultar en la documentación.

El uso de pattern matching facilita de gran manera la programación, en particular para los intérpretes que desarrollaremos en el curso, y muestra claramente la conexión entre una estructura de datos recursiva y las operaciones que se realizan sobre ella. Por lo tanto, desde ahora en adelante y como regla general, usaremos pattern matching para procesar estructuras de datos.

Note que por razones de eficiencia los números naturales no están definidos inductivamente en Racket. Por lo tanto la siguiente expresión produce un error al ser evaluada:

```
(match n
 [0 ...]
 [(add1 m) ...])
```

4.3 Ejemplos

Esta sección muestra como se aplica la metodología anterior a través de múltiples ejemplos.

Como ejercicio, implemente la función `even?` usando `match` y el patrón `app`.

4.3.1 Procesar listas

Una lista puede ser descrita con la siguiente gramática BNF:

⟨*list*⟩ ::= `(list)`
| `( cons` ⟨*val*⟩ ⟨*list*⟩ `)`
⟨*val*⟩ ::= cualquier valor

Por ende, el patrón de cualquier función que procesa listas es el siguiente:

```
(define (f l)
  (match l
   [(list) ...] ; caso base
   [(cons h t) ... h ... (f t) ...])) ; caso recursivo
```

El pattern matching sobre listas es pervasivo en distintos lenguajes funcionales. Dada una lista con cabeza (head) `h` y cola (tail) `t`, su patrón en ML es `h::t`, mientras que en Haskell es `h:t`.

Observe que usamos `(list)` para denotar el constructor de una lista vacía. También es posible utilizar el valor literal `'()`. Sin embargo, hay que tener cuidado y no usar el identificador `empty`, ya que a pesar de denotar una lista vacía, como se aprecia en este código:

```
> empty
'()
> (eq? empty '())
#t
> (eq? empty (list))
#t
```

no es un constructor de esta estructura de datos, ni tampoco un valor literal. Recuerde que el lenguaje de patrones que hemos descrito hasta ahora sólo considera valores *literales* (como `'()`) y constructores (como `list` o `cons`). Al utilizar `empty` como patrón, éste se considera como un identificador librecuyo matching será siempre exitoso (al igual que `else`).

Aquí está una función que calcula el largo de una lista:

```
; length :: List -> Number
; retorna el largo de la lista
(define (length l)
  (match l
    [(list) 0]
    [(cons h t) (+ 1 (length t))]))

(test (length (list)) 0)
(test (length '(1 2 3 4)) 4)
```

Y otra que determina si una lista contiene un elemento dado:

```
; contains? :: List Any -> Boolean
; determina si la lista contiene el elemento dado
(define (contains? l v)
  (match l
   [(list) #f]
   [(cons h t) (or (equal? h v) (contains? t v))]))

(test (contains? (list) 5) #f)
(test (contains? '(1 2 3 4) 4) #t)
(test (contains? '(1 2 3 4) 5) #f)
```

Ejercicios: defina las siguientes funciones que operan sobre listas: `sum`, `reverse`, `map`, `foldl`

4.3.2 Procesar árboles binarios

Un árbol binario es o una hoja (con un valor), o un nodo con un valor y dos hijos, que son árboles también. O sea, podemos describir un árbol binario con el siguiente BNF:

⟨*bt*⟩ ::= ⟨*val*⟩
| `(` `list` ⟨*val*⟩ ⟨*bt*⟩ ⟨*bt*⟩ `)`

Aquí decidimos representar una hoja simplemente con el valor que contiene, y un nodo con una lista de tres elementos. Más adelante veremos una forma más abstracta de representar datos (ver §5.3 "Estructuras de datos").

Note que en la segunda regla de derivación, el no-terminal ⟨*bt*⟩ aparece dos veces. Esto se traduce en el patrón con dos llamadas recursivas:

```
(define (f bt)
 (match bt
  [(list val left-bt right-bt) ...(f left-bt)...(f right-
bt)...] ; caso recursivo
  [val ...])) ; caso base
```

Observe que en esta definición de la estructura fue necesario aplicar el matching sobre el caso recursivo antes que sobre el caso base. La razón es que de lo contrario el patrón `val` aplicaría en todas las situaciones, pues simplemente asocia ese identificador con el valor sobre el cual se aplica. En el siguiente capítulo veremos que esta precaución ya no será necesaria para las estructuras desarrolladas en el curso.

Por ejemplo:

```
; bt-contains? :: BinTree Any -> Boolean
; determina si el árbol contiene el elemento dado
(define (bt-contains? bt v)
 (match bt
   [(list val left-bt right-bt) (or (equal? val v)
                                    (bt-contains? left-bt v)
                                    (bt-contains? right-bt v))]
   [val (equal? val v)]))

(test (bt-contains? 3 3) #t)
(test (bt-contains? 3 4) #f)
(test (bt-contains? '(1 (2 3 4) (5 (6 7 8) 9)) 10) #f)
(test (bt-contains? '(1 (2 3 4) (5 (6 7 8) 9)) 7) #t)
```

Ejercicios: defina las siguientes funciones que operan sobre árboles binarios: `sum`, `max`, `map-bt`, `foldl-bt`

4.3.3 Procesar listas anidadas

Para terminar, consideremos un ejemplo un poco más complicado: listas anidadas, o sea, listas que pueden contener otras listas. (Esto puede ser visto como una forma de representar árboles n-arios también.)

La gramática de las listas anidadas es:

⟨*nl*⟩ ::= `(list)`
| `(` `cons` ⟨*elem*⟩ ⟨*nl*⟩ `)`
⟨*elem*⟩ ::= ⟨*val*⟩
| ⟨*nl*⟩

A diferencia de los casos anteriores, tenemos un no-terminal más (⟨*elem*⟩), en el cual una de las alternativas implica recursión. Esto implica un procesamiento "en profundidad" además de "en anchura". La idea sigue siendo la misma para definir el patrón:

En general, `(? p)` es un patrón que matchea si el predicado `p` es cierto para el valor. Aquí, `(? list?)` matchea si el valor es una lista.

```
(define (f l)
  (match l
    [(list) ...] ; caso base
    [(cons h t)
     (match h ; h puede ser <val> o <nl>
       [(? list?) ...(f h) ... (f t)...] ; <nl>
       [else ... (f t) ...])])) ; <val>
```

Observe que nuevamente se intenta primero el caso más específico ⟨*nl*⟩ antes del caso general ⟨*val*⟩.

Por ejemplo:

```
; nl-contains? :: List Any -> Boolean
; determina si la lista contiene el elemento dado, recorriendo en
profundidad
(define (nl-contains? l v)
  (match l
    [(list) #f]
    [(cons h t)
     (match h
       [(? list?) (or (nl-contains? h v) (nl-contains? t v))]
       [else (or (equal? h v) (nl-contains? t v))])]))

(test (nl-contains? (list) "hola") #f)
(test (nl-contains? '(1 2 3 4) 3) #t)
(test (nl-contains? '(1 2 3 4) 5) #f)
(test (nl-contains? '(1 (2 3 () 4) (11 (((12))) (5 (6 7 8) 9))) 10) #f)
(test (nl-contains? '(1 (2 3 () 4) (11 (((12))) (5 (6 7 8) 9))) 7) #t)
```

Note que en la definición anterior, es posible evitar repetir el mismo llamado recursivo `(nl-contains? t v)` sacando el `or` fuera del `(match h ...)`:

```
(define (nl-contains? l v)
  (match l
    [(list) #f]
    [(cons h t)
     (or (match h
           [(? list?) (nl-contains? h v)]
           [else (equal? h v)])
         (nl-contains? t v))]))
```

Ejercicios: defina las siguientes funciones que operan sobre listas anidadas: `map*`, `foldl*`, y `max` (usando `foldl*`).

5 El Lenguaje PLAY

Racket es a la vez un lenguaje y una plataforma para definir varios lenguajes y sus extensiones, permitiendo construir programas a partir de módulos definidos en múltiples lenguajes. Por ejemplo, este documento está escrito en el lenguaje Scribble, una especie de LaTeX moderno adecuado para documentar programas. El documento OOPLAI, que usaremos para estudiar OOP en el curso, también fue escrito en Scribble. Otros lenguajes que fueron definidos con Racket incluyen lenguajes pedagógicos, un Racket con tipos estáticos, Datalog para programación lógica, un lenguaje para definir aplicaciones web, un lenguaje para especificar lenguajes formalmente, y muchos más...

El curso de lenguajes no explora esta faceta de Racket, pero sí usaremos variantes de Racket. El curso usa principalmente el lenguaje PLAY, que está basado en el lenguaje PLAI. PLAI agrega esencialmente funciones convenientes para especificar tests y para definir y manejar estructuras de datos.

PLAY reutiliza los mecanismos para tests de PLAI, y simplifica el uso de pattern matching sobre las estructuras de datos, de forma que sea análogo al caso de matching sobre listas. Además posee una sintáxis más sucinta para definir valores, basándose también en el uso de pattern matching.

Para instalar el lenguaje PLAY debe utilizar la versión más reciente de DrRacket, e instalar el paquete play: File > Package Manager > especificar `play` en Package Source, y luego Install.

5.1 Definiendo Identificadores

PLAY introduce la forma `def` que permite introducir identificadores de forma simple o utilizando pattern matching. En el primer caso, `def` se utiliza de la misma forma que `define`. Por ejemplo:

```
(def x-max 100)
(def y-max 100)
```

Cuando desarrollamos funciones que involucran varios cómputos intermedios, es una buena práctica introducir identificadores para referirse a esos valores, y así mejorar la legibilidad del código.

Por ejemplo, considere que representamos puntos como un par de coordenadas. Podemos calcular la distancia entre dos puntos como:

```
; distance :: (cons Int Int) (cons Int Int) -> Float
; Calcula la distancia entre dos puntos
(define (distance p1 p2)
```

```
  (def x1 (car p1))
  (def x2 (car p2))
  (def y1 (cdr p1))
  (def y2 (cdr p2))
  (sqrt (+ (expt (- x1 x2) 2) (expt (- y1 y2) 2))))
```

```
> (distance (cons 0 0) (cons 1 1))
1.4142135623730951
> (distance (cons 3 4) (cons 9 0))
7.211102550927978
```

Aquí utilizamos `def` para nombrar las coordenadas de cada punto, de la misma forma que usaríamos `define`. Sin embargo, acceder a los valores de una estructura arbitraria puede ser bastante engorroso. Utilizando una definición basada en patrones se simplifica bastante el código:

```
; distance :: (cons Int Int) (cons Int Int) -> Float
; Calcula la distancia entre dos puntos
(define (distance p1 p2)
  (def (cons x1 y1) p1)
  (def (cons x2 y2) p2)
  (sqrt (+ (expt (- x1 x2) 2) (expt (- y1 y2) 2))))
```

```
> (distance (cons 0 0) (cons 1 1))
1.4142135623730951
> (distance (cons 3 4) (cons 9 0))
7.211102550927978
```

`def` es un alias de `match-define`.

5.2 Tests

El lenguaje PLAY introduce `test` para verificar que una expresión evalúa al resultado esperado:

```
> (test (+ 1 2) 3)
(good (+ 1 2) 3 3 "at line 10")

> (test (+ 1 2) 4)
(bad (+ 1 2) 3 4 "at line 11")
```

Además, el lenguaje PLAY introduce `test/exn` para testear por errores:

```
(define (mysqrt x)
  (if (and (number? x) (positive? x))
      (sqrt x)
      (error "mysqrt: should be called with positive number")))

> (test/exn (mysqrt -1) "positive")
(good (mysqrt -1) "mysqrt: should be called with positive number"
"positive" "at line 13")
```

El segundo argumento de `test/exn` tiene que ser un sub-string del mensaje de error esperado.

Es importante notar que `text/exn` solo detecta errores reportados explícitamente por nuestro código usando `error`, como en el ejemplo anterior. No detecta errores lanzados implicitamente por el runtime del lenguaje:

```
> (test/exn (/ 1 0) "by zero")
(exception (/ 1 0) "/: division by zero" '<no-expected-value> "at line 15")
```

Por defecto PLAY muestra mensajes de éxito o fallo de los tests. Para mostrar mensajes sólo para los test fallidos, se debe evaluar:

```
(print-only-errors #t)
```

La evaluación de `print-only-errors` afecta todas las expresiones que le siguen, y en general basta con utilizarla al principio del archivo.

5.3 Estructuras de datos

Racket incluye una forma básica de definir nuevas estructuras de datos. El lenguaje PLAY provee una forma más elaborada y práctica, que usaremos en el curso. Se llama `deftype`. Con `deftype`, uno introduce un nombre para la estructura de datos, y luego especifica uno o más variantes de este tipo de dato. Por ejemplo:

```
(deftype BinTree
  (leaf value)
  (node value left right))
```

`deftype` en conjunto con el uso de pattern matching (`match` y `def`) reemplazan la funcionalidad de `define-type` y `type-case` provistos por el lenguaje PLAI

Note que cada variante (en este caso `leaf` y `node`) pueden tener distintas cantidades de atributos (incluso ninguno). Cada atributo se define simplemente con un nombre (por ej. `value`).

Cuando se usa `deftype`, se generan automaticamente varias funciones. Primero, se crean constructores, uno por variante, que permiten construir datos:

```
> (leaf 10)
(leaf 10)
> (node 10 (leaf 3) (node 4 (leaf 1) (leaf 2)))
(node 10 (leaf 3) (node 4 (leaf 1) (leaf 2)))
```

También se generan accesores, para inspeccionar datos existentes. Los accesores tienen el nombre *variante-atributo*, por ejemplo:

```
> (leaf-value (leaf 10))
10
> (def y (node 1 (leaf 2) (leaf 3)))

> (node-right y)
(leaf 3)
> (leaf-value (node-left y))
2
```

Finalmente, se generan predicados de tipo, para poder clasificar datos respecto al tipo general y sus variantes:

```
> (BinTree? 10)
#f
> (BinTree? (leaf 10))
#t
> (BinTree? '(10))
#f
> (BinTree? (node 10 (leaf 1) (leaf 2)))
#t
> (leaf? 10)
#f
> (leaf? (leaf 10))
#t
> (node? '(1 2 3))
#f
> (node? (node 3 (leaf 1) (leaf 2)))
#t
> (node? (leaf 5))
#f
```

Para hacer un análisis por casos de una estructura definida usando `deftype`, simplemente utilizamos pattern matching sobre cada uno de sus constructores o variantes. Por ejemplo:

```
(define (contains? bt n)
  (match bt
    [(leaf v) (equal? v n)]
    [(node v l r) (or (equal? v n)
                      (contains? l n)
                      (contains? r n))]))

> (contains? (leaf 1) 2)
#f
> (contains? (node 10 (leaf 3) (node 4 (leaf 1) (leaf 2))) 1)
#t
```

El lenguaje Python soporta una una versión limitada de asignación basada en patrones, para el caso particular de tuplas y listas.

También podemos usar `def` para acceder a los componentes de la estructura:

```
> (def y (leaf 3))

> (def (node v l r) (node 1 (leaf 2) y))

> v
1
> l
(leaf 2)
> r
(leaf 3)
```

Note que a diferencia de la representación de árboles binarios usando listas del capítulo anterior, ahora no debemos preocuparnos del orden en que van los patrones correspondientes a los constructores de cada variante de la estructura.

La razón de fondo es que en una estructura de datos definida inductivamente usando `deftype` se cumple que:

- Los constructores son funciones inyectivas. Por ejemplo `(equal? (leaf a) (leaf b))` evalúa a `#t` solamente si `(equal? a b)` es `#t`.
- Valores construidos con distintos constructores nunca son iguales.
- No existe otra manera de construir valores de la estructura si no es con las variantes definidas.

Por lo tanto, si seguimos la receta de diseño y hacemos matching solamente sobre las variantes de una estructura, no tendremos que preocuparnos de matchings accidentales o inesperados.

La única precaución que debemos tener es recordar ser exhaustivos en el análisis de casos, para asegurarnos que las funciones sean completas para el tipo de datos considerado.

Ejercicios: Defina las funciones `map-tree` y `sum-tree` que operan sobre `BinTree`s.

5.4 Un último ejemplo

A modo de último ejemplo, considere un mini-lenguaje para describir expresiones aritméticas con números, adición y substracción:

⟨*expr*⟩ ::= `(` `num` ⟨*number*⟩ `)`
| `(` `add` ⟨*expr*⟩ ⟨*expr*⟩ `)`
| `(` `sub` ⟨*expr*⟩ ⟨*expr*⟩ `)`

Note que la definición BNF nos da la gramática del lenguaje o, en forma equivalente, una descripción de la estructura de los árboles de sintáxis abstracta. Podemos describir el tipo de dato `Expr` directamente con `deftype`:

```
(deftype Expr
  (num n)
  (add left right)
  (sub left right))
```

Un procesamiento simple de esta estructura es *evaluar* una expresión para obtener su valor número. Por ejemplo, podríamos representar la expresión `1 + (5 - 2)` con la siguiente estructura:

```
(add (num 1)
     (sub (num 5) (num 2)))
```

y usar `calc` para obtener su valor:

```
(calc (add (num 1)
           (sub (num 5) (num 2))))
```

La definición de `calc` es directa, siguiendo la receta que estudiamos para procesar estructuras de datos (ver §4.2 “Procesar estructuras recursivas”). Primero, derivamos el patrón de `calc` basándonos directamente en la definición del tipo de dato `Expr`:

```
(define (calc expr)
  (match expr
```

```
    [(num n) ...]
    [(add l r) ... (calc l) ... (calc r) ...]
    [(sub l r) ... (calc l) ... (calc r) ...]))
```

Lo cual es muy simple de completar:

```
; calc :: Expr -> number
; retorna el valor de la expresión dada
(define (calc expr)
  (match expr
    [(num n) n]
    [(add l r) (+ (calc l) (calc r))]
    [(sub l r) (- (calc l) (calc r))]))

(test (calc (add (num 1)
                 (sub (num 5) (num 2))))
      4)
```

Note como `deftype` y `match` calzan perfectamente con esta metodología.