Introducción

Programación funcional para la física computacional

14 de febrero de 2023

Lenguajes de programación y sus conceptos básicos

A veces los mayores descubrimientos vienen de darse cuenta de qué es lo que no se puede hacer. Ese es el caso de las computadoras, a partir de saber lo que no podemos automatizar, es que podemos construir computadoras para automatizar ciertos procesos, apoyarnos en cálculos extensos para el caso de la física e incluso estudiar sobre fenomenos de la naturaleza y la sociedad, como sucede con casi todas las ciencias sean naturales, formales y hasta sociales.

Los lenguajes de programación prácticos, usados día a día y que son un poco más amables con los programadores, son una estructura complicada, pero las ideas importantes en la labor de la programación son simples.

Conceptos básicos

Programa: Descripción de como se manipula la información.

Lenguaje de programación: Lenguaje formal, artificial usado para dar instrucciones a una computadora.

Lenguaje de alto nivel: En lugar de pensar en controlar el *hardware* directamente, se piensa en el programa.

Un paradigma de programación es un enfoque para programar una computadora basado en una teoría matemática o un conjunto coherente de principios.

- **Programación estructurada**: Uso extensivo de flujo de control estructurado: if, $while$, for.
- **Programación orientada a objetos**: una gran cantidad de abstracciones de datos con características comunes y una jerarquía: física de partículas y C++.
- **Programación lógica**: manejar estructuras simbólicas complejas de acuerdo a las reglas de la lógica (ejemplo pendiente).
- **Programación funcional**: Usa funciones matemáticas en lugar de declaraciones para expresar las ideas. Como no depende de los estados hay menor riesgo de **no determinismo**.

Por lo regular los lenguajes populares de programación tienen a lo más dos paradigmas (como C++ y java), pero lo mejor sería que todo lenguaje de programación acepte múltiples paradigmas. Multiparadigma es el término que se usa para nombrar, como si fuera una cualidad específica y no algo deseable en todo lenguaje.

Programación orientada a objetos

Uno de los paradigmas más populares, uno de los primeros a llegar a las computadoras en laboratorios y personales.

De sus primeras implementaciones se menciona al lenguaje *Simula67* que fue una ampliación de *Algol60*, el primero en introducir los conceptos de **objeto**, **clase**, **subclases** y **herencia**.

Ya en 1980 aparecen $C++$ y $Smaltalk-80$, siendo el primero uno de los lenguajes más utilizados aún hoy en día. Java y C++ son lenguajes de programación orientada a objetos y basta con buscar en internet para notar que son dos de los lenguajes más utilizados en casi cualquier ámbito.

Dentro de la física suele ser muy utilizado en simulaciones de física de partrículas. Una parte importante del software para el análisis de datos y simulaciones para el LHC está enm $C++$ aunque partió un poco del $FORTRAN$ y empieza a incluir $Python$. El poder definir clases y objetos facilita el manejo de las partículas elementales en las simulaciones, o los árboles de datos en el análisis de los experimentos.

Programación lógica

Como menciona [Kowalski1988] definir la programación lógica puede ser un poco ambiguo. "La programación lógica comparte con la demostración automática de teoremas el uso de la lógica para representar conocimiento y el uso de la deducción para resolver problemas al derivar consecuencias lógicas". pero a su vez dice que "difiere de la demostración mecánica de teoremas en dos formas distintas pero complementarias: (1) explota el hecho de que la lógica puede utilizarse para expresar definiciones de funciones computable y procedimientos, y (2)explota el uso de procedimientos de prueba que realizan deducciones de manera dirigida por el objetivo, para correr tales definiciones como programas".

Uno de los lenguajes de programación lógica más utilizados es $Prolog$, con aplicaciones en inteligencia artificial. No sé de algún uso en la física.

Programación funcional

Como su nombre lo dice, la programación funcional parte de funciones. Podría pensarse que un poco como $Fortran$ no está pensado para crear aplicaciones, si no sólo resolver problemas matemáticos, pero no, su impronta ha sido tan dinámica hasta llegar a tener software para la creación musical (*tidal cycles*) y para la transformación de formatos de texto (*pandoc*). Realmente es una manera de concentrarse en las relaciones entre valores más que en los objetos con que se trabaja.

Usemos el clásico ejemplo de la función como una caja negra, como se puede ver en la figura . La función recibe uno, dos o más argumentos, valores, objetos, en general entradas, y regresa una o más salidas o resultados.

Podría ser una función que recibe como entrada un programa y te dice si tu programa está bien escrito y dará un resultado (**¿es eso posible?**).

¿Cómo diferencia una función si se le da un vector, un escalar, un sonido o incluso un programa de computación? Eso tiene que ver con los **tipos**, que son colecciones de valores agrupados juntos por ciertas características que comparten. Como estamos

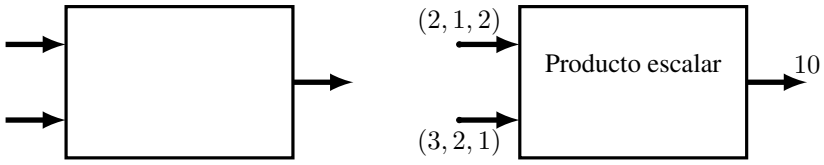

Figura 1: Una función representada como caja negra.

pensando funcionalmente: podemos aplicar relaciones parecidas entre esos valores, les aplicamos los mismos tipos de funciones.

En el caso de la figura (b) los tipos de las entradas son vectores, el tipo de la salida es escalar.

Haskell es el lenguaje de programación funcional que usaremos en parte de este curso, *python* permite implementarlo pero no de manera nativa como lo hace *Haskell*. Recibe su nombre por Haskell B. Curry, matemático especializado en lógica que trabajo ampliamente en el calculo λ, una teoría matemática de las funciones que veremos con un poco más de detalle más adelante.

Cuestiones generales de los lenguajes de programación

Pensemos en la cuestión más sencilla al usar un lenguaje de programación en nuestra área de trabajo, pensaríamos en hacer una calculadora lo más amplia posible, hasta incluir matrices, cálculos de números complejos. ¿Qué necesitamos?

- Operaciones sencillas: suma, resta, multiplicación, producto escalar, multiplicación de matrices...
- Variables: atajos a valores, sólo se asignan una vez. Cada variable cuenta con un identificador y valor guardado, que viene de un estado de memoria. Para esto existen enunciados declarativos.
- Funciones: ya las explicamos más o menos a detalle en la sección pasada.
- Recursión: poder definir una función a partir de ella misma, en el caso de que dependa del valor anterior.
- Abstracciones funcionales: usar funciones para construir abstracciones, pensemos en una cebolla, varias capas de funciones.

Para entender un poco de como es eso de la recursión definamos la función factorial, primero como y¡una función, pero además como una función recursiva, hagámoslo en $python$.

```
1  def fact(x): #definimos la funcion fact
2    if x==0:   #caso base
3        return 1
4    else:
5        return x*fact(x-1)  #para los demas casos
6
7    x=int(input("Valor a calcular factorial:")) #dar valor
8    print(fact(x))  #imprime el resultado
```

La función es definida en la línea 1, el argumento será la variable x, a dársele en el momento que se desee evaluar. Como saben el factorial de 0 vale 1 por definición, el resto de valores se obtiene recursivamente a partir del anterior, así en general $n! = n \times (n-1)!$. Revisen el programa con unos pocos valores a ver que obtienen.

Para trabajar con estructuras más allá de números podemos hacer uso de las listas, para desarrollar el tema de esta sección veamos un ejemplo algo abstracto pero que puede ser útil: el triángulo de Pascal.

El triángulo de Pascal

El triángulo de Pascal es útil en combinatoria y está relacionado al teorema binomial. No dudo que les pueda ser útil más adelante, pero a esta altura lo veremos como un ejemplo al trabajar con tipos más elaborados que un escalar, los trataremos como listas de números.

En la fila 1 empieza con un 1, la regla para construir los subsecuentes valores es ir en orden descendente sumando las parejas de números justo arriba. Las orillas se suponen llenas de ceros, así seguida de la fila 1 sigue una pareja de números, el de la izquierda es la suma del 1 de la fila 1 con su vecino de la izquierda, que está vacío y tomamos como cero. Esto nos da 1, de manera similar para el lado derecho, como puede verse en el cuadro 2.

$n = 1$:				1				
$n = 2$:			1		1			
$n = 3$:		1		2		1		
$n = 4$:	1		3		3		1	
$n = 5$:	1		4		6		4		1

Cuadro 1: Primeros 5 niveles del triángulo de Pascal

Ahora queremos construir una función que nos dé el valor de la fila n. Hagámoslo de manera similar al factorial, en $python$ y pensando en funciones. Esta vez no trabajaremos con números naturales, si no con listas de números naturales, en los que no sólo tendremos que operar sobre los números, también sobre la estructura.

Pensando de nuevo en definir funciones lo que haríamos sería algo más o menos como lo que se ve en la figura 2.

Figura 2: Desarrollo de arriba $pa'bajo$

Datos: El valor de la primera fila del triángulo de Pascal
Resultados: Cualquier fila n del triángulo de Pascal
inicialización;

```
1    function Pascal(n):
2      if n==1 then return [1];
3      else
4        return SumaLista(CorrerListaDerecha(n-1),CorrerListaIzquierda(n-1));
5      end
6    end
7
```

Esto se llama desarrollo de software de "arriba a abajo" (*top-down*), proponemos las funciones que resuelvan los detalles y luego llenamos los huecos.

¿Cómo serían esa funciones? Lo primero que tenemos que ver es que los objetos que nos regresa la función Pascal son listas, lo que le damos de entrada es un natural que identifica la fila del triángulo. Veamos qué es una lista.

Una lista la podemos entender como una cadena de ligas, cada liga contiene dos cosas, un elemento de la lista y la referencia al resto de la cadena. Una lista vacía puede escribirse como $[\ ]$ o simplemente como nil. A esta lista vacía se le van agregando elementos[1].

En $python$ una lista se define encerrada entre corchetes, así la lista vacía de nombre a sería $a = []$. Para agregar elementos se puede utilizare la función $append$. De esta forma podemos construir la lista $[8, 2{,}3, -4]$ de la siguiente forma:

```
a=[]   #crea una lista vacia
a.append(8)    #va agregando elementos
a.append(2.3)
a.append(-4)

print(a)  #imprime la lista
```

Noten que los va agregando en el orden empezando de izquierda a derecha, tal como leemos las palabras del español. Pero podemos usar algunos atajos para agregar elementos por ambos lados, las funciones de concatenación de listas que se escriben como $+$, pero no debe confundirse con una suma de números, es una operación distinta al aplicarse en listas.

```
a=[2.3]   #crea la lista de un solo valor, el central
a=[8]+a+[-4]  #concatenamos por izquierda y derecha

print(a)  #imprime la lista
```

De esta forma podemos construir una función que nos dé la n-eva fila del triángulo de Pascal, lo que debemos hacer es generar dos listas, a una se le agregará por la izquierda un cero y a la otra por la derecha, para ser sumadas a continuación.

```
def pascal(n):
  if n==1:
    return [1]
  else:
    x=[0]+pascal(n-1)
    y=pascal(n-1)+[0]
    return [i+j for i,j in zip(x,y)]

  n=int(input(''Fila (1,2,3,..)?:''))
  print(pascal(n))
```

Corrección

¿Cómo sabemos si lo que hacemos es correcto? Podemos comprobarlo para la línea 1, 2, 3,..., pero ¿para la línea 20?

Yo espero recuerden sus cursos de álgebra (la de primer semestre, sea superior o para físicos, en ambas se veía por lo regular el tema de inducción. Al igual espero les hayan constado que hay tres tipos de inducción, son equivalentes pero cada tipo nos da detalles sobre dónde estamos trabajando.

[1] Esta terminología se tomó del lenguaje de programación *lisp*, cuyo nombre deriva de *list processing*.

Para probar que nuestro programa es correcto necesitamos de un modelo matemático de las operaciones del lenguaje de programación, es decir, una semántica del lenguaje. Con esta semántica debemos especificar lo que queremos que el programa haga: especificación del programa. Por último se utilizan técnicas matemáticas para razonar sobre el programa, usando la semántica.

Los tres tipos de inducción son la normal que conocen, sobre los números naturales, la inducción fuerte y la inducción bien fundada. Por ser la más general me centraré en la inducción bien fundada, pero como mencioné, son equivalentes.

Definición 1 *$(A, \preceq)$ es llamado un conjunto bien fundado si y sólo si todo subconjunto no vacío M de A contiene al menos un elemento minimal m con respecto a la relación de orden $\preceq$.*

Entonces podemos ir más allá de los naturales siempre y cuando podamos dar una relación de orden para el conjunto sobre el que estamos trabajando.

Teorema 1 *(Principio de inducción Noetheriana) Sea $(A, \preceq)$ un conjunto bien fundado. Para probar que una propiedad $P(x)$ es verdadera para todos los elementos x en A es suficiente probar las siguientes propiedades:*

(a) ***Base inductiva:*** *$P(x)$ es verdad para todos los elementos minimales de A.*

(b) ***Paso inductivo*** *Para cada no minimal x en A, si $P(y)$ es verdad para toda $y \prec x$ (**hipótesis inductiva**), entonces $P(x)$ es verdad.*

Para darles un ejemplo no me detendré en el caso del triángulo de Pascal, pero sí en el factorial que antes hicimos. Más que un ejercicio es un ejemplo ilustrativo.

Ejemplo: Demuestra que el factorial es una función total.

Recordemos que nuestra función factorial está definida de la siguiente forma:

$$f(0) = 0! = 1$$
$$f(x) = x(x-1)!,\ x > 0 \text{ y } x \in \mathbb{N}.$$

Entonces aplicamos la inducción:

- **Base inductiva:** Estamos trabajando en los naturales más el 0 ($\mathbb{N}_0$), el minimal es el 0 y por la definición de la función $f(0) = 0! = 1$, ya está.
- **Hipótesis inductiva:** Supongamos es cierto para $n > 0$ con $n \in \mathbb{N}_0$, en ese caso tenemos que $f(n) = n!$.
- **Paso inductivo:** Probemos que se cumple para $n + 1$. Para este caso por definición $f(n+1) = (n+1)n!$, como supusimos que $n!$ está valuado por consiguiente $(n + 1)n!$ está valuado también, es solo una multiplicación de naturales que da un natural $\square$.

Complejidad

Ahora viene un ejercicio entretenido aunque puede llegar a preocupar en algún momento: determinar el tiempo de ejecución de nuestro programa.

Por un lado es complicado dar un tiempo que valga para toda computadora, el tiempo que tarde dependerá mucho de la arquitectura, el RAM, si se corren más programas

a la par, incluso la temperatura del equipo. Pero sobretodo nos interesa saber si podemos correr el programa en tiempos considerables, por un lado humanamente posibles, por otro viables para la tarea que buscamos.

El tiempo de ejecución de un programa como función del tamaño de entrada, hasta un factor constante, es llamada la complejidad en el tiempo del programa. Es decir, existe una $t(n)$ que nos da el tiempo aproximado de la ejecución, un factor constante más o menos.

¿Porqué tan impreciso se preguntarán? Si el tiempo es $t(3s) = K \times p(3)$ donde $p(3)$ es un polinomio valuado en 3 no es lo mismo $K = 1$ que $K = 1,000,000$. Por un lado ya mencionamos que es una medida muy aproximada, pero además lo interesan te es saber que no se disparará el tiempo de ejecución, es muy diferente $t(n) = n^3$ que $t(n) = 3^n$.

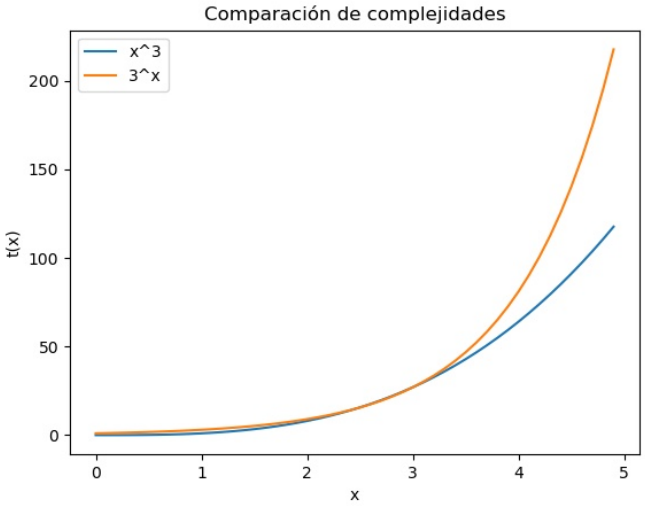

Figura 3: Comparando la complejidad temporal $t(x) = x^3$ con $t(x) = 3^x$ para valores muy pequeños.

¿Qué sucede para nuestro programa que calcula la fila n del triángulo de Pascal? Notemos que como lo tenemos definido para calcular la fila 28 llama dos veces a calcular la fila 27, cada una de estas llama dos veces a calcular la fila 26, lo que equivale a 4 llamadas de la función, en cada paso se va multiplicando por dos, va como 2^n, ¡en el paso 28 hará 268435456 llamadas a la función!

¿Cómo podemos mejorarlo? Llamando solo una vez a la función.

```
29  def pascal(n):
30    if n==1:
31      return [1]
32    else:
33      r=pascal(n-1)
34      x=[0]+r
35      y=r+[0]
36      return [i+j for i,j in zip(x,y)]
37
38    n=int(input(''Fila (1,2,3,..)?:''))
39    print(pascal(n))
```

Prueben el tiempo y verán las mejoras en el tiempo, esta vez cada paso sólo llama una vez a la función.

Evaluación perezosa

Imaginen que desean obtener todos los números primos posibles, más que eso, operar con ellos. Sin saber mucho de computación podemos estar seguros que esa labor es imposible, ya que es una cantidad de números infinitos. Todo lo que solemos operar en una computadora, o por medioas físicos es finito ¿como operar sobre estructuras infinitas?

En ese caso se utiliza la evaluación perezosa, su principal característica es que el cálculo se realiza sólo cuando se necesita. Para poner un ejemplo, en la programación normal, "chambeadora" si quieres obtener un elemento de una secuencia de números primero debes decirle a la máquina cuál es esa secuencia de números. Ya sea que definas un arreglo o una lista, pero debes determinarlos todos. En la evaluación perezosa puedes trabajar con la estructura infinita y solo calcular el elemento cuando sea necesario evaluar, sin necesidad de generar toda la lista o arreglo antes.

Pongamos un ejemplo, el más sencillo es tratar de obtener todos los naturales, definir una lista de todos ellos es imposible, es infinita, pero podemos trabajar con el generador de esos números:

```
def nats(n): #generador de naturales
    yield n
    yield from nats(n+1)

s=nats(1) #iniciamos el generador
```

Esto no se ve nada impresionante, estamos sumando un número al anterior, pero esta implementación sencilla ya es nuestra entrada al *infinito y más allá*, porque realmente estamos trabajando con un generador. En gran medida es una función que puede generarnos todos los naturales pero que además podemos operar con ella (nótese que ya estamos hablando de programar con funciones). Y se preguntarán ¿Y cómo la uso? Veamos un ejemplo.

Veamos un algoritmo griego con más de 2000 años de antigüedad, el colador de Eratostenes. ¿Cómo puedo obtener todos los números primos? Podemos hacerlo por fuerza bruta, es decir, por la simple definición dejar que el programa haga las operaciones, que cheque que cada número solo es múltiplo de el mismo, pero como se imaginarán al poco tiempo la tarea se volverá tardada. A Eratostenes se le ocurrió una manera que lleva menos tiempo: se ponen todos los número naturales sobre la rejilla del colador, en esa primera colada todos los múltiplos de 2 se quedarán como asientos y los demás número fluirán a la siguiente etapa. En esta etapa todos los múltiplos de 3 se quedan sedimentados y pasan los siguientes números. El 4 ya se quedo unh sedimento arriba así que sigue el 5, y así se continúa.

Referencias

[1] Thompson, Simon J.. "Haskell - the craft of functional programming." International computer science series (1996).

[2] Van Roy, Peter. "Programming paradigms for dummies: what every programmer should know." (2009).

1:	2	3	4	5	6	7	8	9	10	11	...
2:	3	5	7	9	11	...					
3:	5	7	11	...							
5:	7	11	...								
7:	11	...									
11:	...										

Cuadro 2: Obteniendo números primos

[3] Van Roy, Peter, Haridi, Seif. "Concepts, Techniques, and Models of Computer Programming". *The MIT Press*, (2004). ISBN: 0262220695

[4] Kowalski, Robert. "The early years of logic programming",*Communications of the ACM*, **31**, 1, (1988).