Introducción a los Algoritmos - 1er. cuatrimestre 2019

Guía 1: Expresiones y funciones

Objetivos

El objetivo de esta guía es brindar una introducción al formalismo básico que se utilizará en la materia. Introduciremos los conceptos principales involucrados apoyándonos en un terreno conocido como es la aritmética. Definimos expresiones y fórmulas aritméticas, funciones; y trabajamos con los conceptos de precedencia, tipado, evaluación de expresiones, casos, entre otros.

1. Analizá con tus compañeros las siguientes oraciones:

   Ea oaió o iee ooae iea ue a iuiee o iee oae.

   St rcn n tn vcls mntrs q l ntrr n tn cnsnnts.

   a) ¿Las comprendés?

   b) ¿Una computadora los podría comprender?

   c) ¿Cuál es la diferencia entre una persona y una computadora?

   d) ¿Qué características debería cumplir una expresión para ser "comprendida" por una computadora?

Comenzando con la aritmética

Comenzaremos trabajando con los conceptos matemáticos con los que más familiarizado estás: los números y la aritmética, pero vamos a estudiarlos desde una perspectiva formal. En primer lugar, presentaremos la ***sintaxis*** *que nos permitirá construir expresiones. La sintaxis se ocupa de definir qué símbolos podremos utilizar y de la forma correcta en que deben estar dispuestos para armar expresiones.*

Nuestras ***expresiones*** *podrán tener la siguiente estructura:*

- *Un número, por ejemplo:* 1, 35, 8. *A estas expresiones les llamaremos* ***constantes****.*
- *Una letra, por ejemplo:* x, y, z. *A estas expresiones les llamaremos* ***variables****.*
- *Combinando dos expresiones con un* ***operador****, y encerrando la expresión resultante entre paréntesis, podemos construir expresiones más complicadas. Por el momento los operadores que utilizaremos serán* $+, -, *, /, \hat{}$ *(este último símbolo corresponde a la potenciación). Por ejemplo, combinando la variable* z *con la constante* 6 *a través del operador* $-$ *se puede formar la expresión:* $(z - 6)$*. Podremos eliminar los paréntesis superfluos según las reglas que analizaremos más adelante.*
- *A su vez, cada una de estas expresiones puede volver a combinarse con otras formando expresiones más complejas, por ejemplo:* $((x + y)/(2\hat{}x)) + (6/3)$.

Para poder expresar propiedades como $2 + 1$ *es igual a la expresión* 3*, o que* x *es menor a* $x + 1$ *agregaremos símbolos de* ***relación****.*

- *Cuando combinamos dos expresiones usando un símbolo de relación decimos que construimos una* ***fórmula****. Una fórmula es un tipo particular de expresión. Los símbolos de relaciones que consideraremos por el momento serán:* $=, \neq, <, \leqslant, >$ *y* $\geqslant$*. Por ejemplo, si tomamos las expresiones* $2 * 6$ *y* $x + 7$ *y el símbolo de relación* $=$*, podemos construir la fórmula* $2 + 6 = x + 7$.

En esta materia programaremos en el lenguaje `haskell`*. Con el programa* `ghci` *podemos hablar en el lenguaje* `haskell`*. En* `ghci` *podés probar si una expresión o fórmula está construída de forma correcta. Si es correcta* `ghci` *te dará el resultado de evaluarla. Si es incorrecta* `ghci` *te dará un error.*

2. Probá en `ghci` si las siguientes expresiones están o no correctamente construídas. Si no es correcta, ¿Podés explicar cuál es el problema?

- $2 + 1$
- $2+$

3. Probá en `ghci` si las siguientes fórmulas están o no correctamente construídas. Ayuda: en `Haskell` el símbolo de relación $=$ se escribe $==$, por lo tanto lo que normalmente escribimos como $2 = 1$, en `haskell` se escribe $2 == 1$.
   - $2 = 1$
   - $2 = 2$
   - $2 =$
4. Construí dos expresiones con al menos dos operadores. ¿Cómo podrías justificar que tus expresiones están bien construidas?
5. Presentá dos expresiones que estén mal construidas y da razones para justificar porqué considerás que están mal construidas.
6. Construí una fórmula de acuerdo con las reglas precedentes. ¿Cómo podrías justificar que está bien construida?

Precedencia y tipado

*La **precedencia** de los operadores nos permite escribir fórmulas grandes de manera simple y más legible. Cuando un operador tiene mayor precedencia que otro, podemos escribir una fórmula que involucra a ambos sin necesidad de poner paréntesis, y a pesar de esto, la fórmula tiene un sentido único. Un ejemplo conocido por todos es el caso de la suma respecto a la multiplicación. El operador $*$ tiene mayor precedencia que $+$ y por ello es que normalmente interpretamos la expresión $2 + 4 * 3$ como $2 + (4 * 3)$. Esta regla es la que nos permitía en la primaria "separar en términos" una expresión algebraica. Al mismo tiempo, y por la misma regla, sabemos que no es lo mismo la expresión $(2 + 4) * 3$ que $2 + 4 * 3$. No siempre los paréntesis pueden sacarse indiscriminadamente sin cambiar el sentido de la expresión.*

*De la misma manera, cuando un operador **asociativo** se aplica múltiples veces es posible eliminar paréntesis, ya que el orden en que se efectúa la operación no altera el resultado. Por ejemplo $(2+4)+7$ es igual a $2+(4+7)$ y por lo tanto podemos escribir simplemente $2+4+7$.*

Los operadores $+$, $*$ son asociativos.

tabla de precedencia	*más arriba → mayor precedencia*
$\hat{}$	*potencia*
$*, /$	*producto y división*
$+, -$	*suma y resta*
$=, <, \leqslant, >\geqslant$	*operadores de comparación*

*Por otro lado, la noción de **tipado** nos permite distinguir expresiones que están "bien escritas", es decir que tienen sentido, que representan algún valor. El tipo de un operador o función nos dice cuál es la clase de valores que toma y cuál es la clase del valor que devuelve. En las expresiones algebraicas en general todos los valores son de tipo número, que denotamos con Num (y los **subtipos** Nat, Int, ...). Como ejemplo tomemos el caso de la suma: tanto a la izquierda del símbolo "+" como a la derecha debe haber expresiones que representen números, y el resultado total también es un número.*

Además del tipo Num utilizaremos el tipo de los valores booleanos, que denotamos con Bool. Este tipo incluye las constantes True y False que representan las nociones de verdadero y falso respectivamente. Los valores booleanos son importantes porque son el resultado de los operadores de comparación $=, <, \leqslant, >, \geqslant$.

Las nociones de precedencia y tipado son importantes para asegurar que escribimos expresiones que tienen un sentido único y bien definido. Los siguientes ejercicios tratan sobre estas nociones en el marco de las expresiones algebraicas.

7. Sacá todos los paréntesis que sean *superfluos* según las reglas de precedencia y asociatividad de los operadores aritméticos. Por ejemplo:

$$
\begin{aligned}
& (8-6) * x = \underline{(6 * (x^2))} + 3 \\
\equiv & \{ \text{ precedencia de } * \text{ por sobre } + \} \\
& (8-6) * x = 6 * \underline{(x^2)} + 3 \\
\equiv & \{ \text{ precedencia de } {}^2 \text{ sobre } * \} \\
& (8-6) * x = 6 * x^2 + 3
\end{aligned}
$$

a) $((5 + 1) + (3 * 6)) * (8 * 5)$

b) $((2^2) + 5) < (2 + 4)$

8. Introducí paréntesis para hacer *explícita* la precedencia.

   a) $5 * 3 + 4 \geqslant 7 - 7 + 3$

   b) $3 + 4 * x = 4$

9. Usá `ghci` para verificar que los parentesis que sacaste y agregaste en los 2 ejercicios anteriores no cambian el resultado de evaluar las expresiones. Ayuda: En `haskell` la expresión 2^2 se escribe `2^2`.

10. Escribí el *tipo* de los siguientes operadores y funciones: $*$, $/$, ^{2}, $\leqslant$, $\geqslant$, $=$. ¿Cuál es el tipo de esos mismos operadores en `haskell`? ¿Qué sucede con el operador $-$?

    Como ejemplo presentamos el caso del operador $+$. Este operador toma dos números y devuelve otro número. Podemos escribir su tipo en *notación funcional* listando el tipo de los parámetros y a continuación el tipo resultado:

    $$+ : Num \to Num \to Num$$

    Otra forma de escribir el tipo de este operador, útil para armar el árbol de tipado de una expresión, es en *notación de árbol*:

    $$\frac{Num + Num}{Num}$$

    En `haskell` el operador `(+)` tiene el mismo tipo. En `ghci` podemos corroborarlo de la siguiente manera:

    ```
    Main> :t (+)
    (+) :: Num a => a -> a -> a
    ```

    Esta respuesta se interpreta de la siguiente manera. El tipo del operador `(+)` es lo que está a la derecha de `=>`, es decir $a \to a \to a$, donde la variable a indica cualquier tipo. Por otro lado, lo que está a la izquierda de `=>` nos indica que el tipo a es un subtipo de Num, es decir, Nat, Int.

Evaluación

Cuando trabajamos con expresiones y fórmulas, muchas veces ***simplificamos*** *las expresiones para que sean más simples. Este proceso de simplificación está asociado a la semántica o significado que le damos a la expresión. Por ejemplo, si tenemos la expresión* $7+1$ *la reemplazamos por* 8 *pues sabemos que ambas expresiones representen el mismo número. La noción de* ***evaluación*** *está más relacionada con las características sintácticas (asociado a los símbolos) de la expresión cumpliendo una función similar.*

11. Simplificá las siguientes expresiones

    a) $70 * 47/(3 + 2)$

    b) $(3 + 1) * 10 = 25$

    c) $2 * x + y = 1 + 2 * y$

12. Evaluá en `haskell` las expresiones del ejercicio anterior. ¿Qué sucede con la última expresión?

Funciones

En matemática, se dice que una magnitud o cantidad es función de otra si el valor de la primera depende exclusivamente del valor de la segunda. Por ejemplo, supongamos que queremos viajar desde Córdoba a Buenos Aires en auto. La duración del viaje T *dependerá de la distancia* d *entre Córdoba y Buenos Aires y de la velocidad* v *que lleve el auto. Del mismo modo, el área* A *de un círculo es función de su radio* r*. Estas magnitudes a veces*

pueden vincularse a través de la proporcionalidad directa o ser inversamente proporcionales. Así, el área A de un círculo es proporcional al cuadrado de su radio, lo que se expresa con una fórmula del tipo $A = \pi * r^2$ y la duración de un viaje T es inversamente proporcional a la velocidad del vehículo ($T = d/v$). A la variable que se encuentra a la izquierda de estas fórmulas (el área, la duración) se la denomina variable dependiente, *y a las variables de las que depende (el radio, la velocidad, la distancia) se las llama* variables independiente.

De manera más abstracta, el concepto general de función se refiere a una regla que asigna a cada elemento de un primer conjunto un único elemento de un segundo conjunto. Por ejemplo, cada número entero posee un único cuadrado, que resulta ser un número natural (incluyendo el cero):

$$\begin{array}{ccccccccc} \ldots & -2 & -1 & 0 & 1 & 2 & 3 & \ldots \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ & +4 & +1 & 0 & +1 & +4 & +9 & \end{array}$$

Esta asignación constituye una función entre el conjunto de los números enteros y el conjunto de los naturales.

Diremos que una ***función*** *le "asigna" a los elementos de un conjunto, llamado* ***dominio****, elementos de otro conjunto, llamado* ***codominio****. La notación:*

$$\texttt{f} : A \to B$$

indica que `f` *es una función con dominio A y codominio B. La definición de una función tiene la forma*

$$\texttt{f}.x \doteq \langle \textit{expresión que depende de } x \rangle$$

donde `f` *es el nombre de la función y x es/son la/s variable/s independiente/s.*

13. En las siguientes definiciones identificá las variables, las constantes y el nombre de la función

    a) $\texttt{f}.x \doteq 5 * x$

    b) $\texttt{duplica}.a \doteq a + a$

    c) $\texttt{por2}.y \doteq 2 * y$

    d) $\texttt{multiplicar}.zz.tt \doteq zz * tt$

14. Escribí una función que dados dos valores, calcule su promedio.

15. Tomando las definiciones del punto 13 evaluá las siguientes expresiones. Justificá cada paso utilizando la notación aprendida. Luego, controlá los resultados en `Haskell`.

    a) $(\texttt{multiplicar}.(\texttt{f}.5).2) + 1$

    b) $\texttt{por2}.(\texttt{duplica}.(3 + 5))$

16. Tomando las definiciones en el punto 2 demostrá que *duplica* y *por*2, si son aplicadas al mismo valor, dan siempre el mismo resultado. En otras palabras, la expresión $duplica.x = por2.x$ es **válida**.

Tipado de funciones

Tomemos la función `g` *definida así:*

$$g.x.y \doteq 3x - x * y > 0$$

Esta función toma dos argumentos de tipo Num y devuelve un valor de tipo $Bool$. Así, el tipo de `g` *se declara de la siguiente forma:*

$$\texttt{g} : Num \to Num \to Bool$$

Es decir, los tipos de los argumentos se listan primero, siguiendo el orden en el que serán llamados por la función, y en último lugar se coloca el tipo del resultado de evaluar la función.

17. Dar el tipo de las funciones del ejercicio 13 y el ejercicio 14.

18. Dar el tipo de las siguientes funciones:

    a) $\texttt{g}.y \doteq 8 * y$

b) $\texttt{h}.z.w \doteq z + w$

c) $\texttt{j}.x \doteq x \leqslant 0$

Definiciones de Funciones por casos

Hay ocasiones en las que una sola fórmula no alcanza para definir una función. Así, existen funciones que para un conjunto de argumentos requieren una definición y para otro conjunto de argumentos necesitan de otra definición diferente. Es el caso, por ejemplo, de la función `espar` *que dado un natural devuelve true si el número es par o false si el número es impar.*

Una ***definición por casos*** *de una función tendrá la siguiente forma general:*

$$\begin{array}{lcll} \texttt{f}.x & \doteq & (& B_0 \to f_0 \\ & & \square & B_1 \to f_1 \\ & & & \vdots \\ & & \square & B_n \to f_n \\ & &) & \end{array}$$

donde las B_i son expresiones de tipo booleano, llamadas ***guardas*** *y las f_i son expresiones del mismo tipo que el resultado de f. Para un argumento dado el valor de la función se corresponde con la expresión cuya guarda es verdadera para ese argumento.*

Al trabajar con expresiones booleanas se hace necesario incorporar a nuestro formalismo los operadores $\wedge, \vee, \neg$ que pueden ser utilizados para combinar dos fórmulas para obtener una nueva fórmula. En `haskell` *estos operadores se escriben* `&&` *,* `||` *y* `not`*, respectivamente.*

19. Definí las funciones que describimos a continuación, luego implementalas en `haskell` . Por ejemplo:

    Enunciado: *signo* : *Int* $\to$ *Int*, que dado un entero retorna su signo, de la siguiente forma: retorna 1 si x es positivo, -1 si es negativo y 0 en cualquier otro caso.

    Solución:

    $$\begin{array}{lcll} \texttt{signo}.x & \doteq & (& x > 0 \to 1 \\ & & \square & x < 0 \to -1 \\ & & \square & x = 0 \to 0 \\ & &) & \end{array}$$

    En `haskell` se escribe así:

    ```
    sgn :: Int -> Int
    sgn x | x>0   = 1
          | x<0   = -1
          | x==0  = 0
    ```

    a) *entre0y9* : *Int* $\to$ *Bool*, que dado un entero devuelve *True* si el entero se encuentra entre 0 y 9.

    b) *rangoPrecio* : *Int* $\to$ *String*, que dado un número que representa el precio de una computadora, retorne "muy barato" si el precio es menor a 2000, "demasiado caro" si el precio es mayor que 5000, "hay que verlo bien" si el precio está entre 2000 y 5000, y "esto no puede ser!" si el precio es negativo.

    c) *absoluto* : *Int* $\to$ *Int*, que dado un entero retorne su valor absoluto.

    d) *esMultiplo2* : *Int* $\to$ *Bool*, que dado un entero n devuelve *True* si n es múltiplo de 2.
    Ayuda: usar `mod`, el operador que devuelve el resto de la división.

Combinando Funciones

En algunos ejercicios que siguen se van a utilizar algunas de las funciones que están en el Prelude. Por ejemplo:

mod 20 3 = 2 – el resto de la división entre 20 y 3 es 2.
div 14 3 = 4 – parte entera de la división entre 14 y 3 es 4.
max 8 10 = 10 – devuelve el max entre 2 números.
min 9 15 = 9 – devuelve el min entre 2 números.

20. Definir la función *esMultiploDe* : *Num*$\to$ *Num*$\to$ *Bool*, que devuelve *True* si el segundo es múltiplo del primero. Ejemplo: *esMultiploDe* 3 12 = *True*.

21. Definir la función *esBisiesto*: *Num*→ *Bool*, que indica si un año es bisiesto. Un año es bisiesto si es divisible por 400 o es divisible por 4 pero no es divisible por 100.

22. Definir la función *dispersion* : *Num*→ *Num*→ *Num*→ *Num*, que toma los tres valores y devuelve la diferencia entre el más alto y el más bajo. Ayuda: extender max y min a tres argumentos, usando las versiones de dos elementos. De esa forma se puede definir dispersión sin escribir ninguna guarda (las guardas están en max y min, que estamos usando).

23. Definir la función *celsiusToFahr* : *Num*→ *Num*, pasa una temperatura en grados Celsius a grados Fahrenheit. Para realizar la conversión hay que multiplicar por 1.8 y sumar 32.

24. Definir la función *fahrToCelsius* : *Num*→ *Num*, la inversa de la anterior. Para realizar la conversión hay que primero restar 32 y después dividir por 1.8.

25. Definir la función *haceFrioF* : *Num*→ *Bool*, indica si una temperatura expresada en grados Fahrenheit es fría. Decimos que hace frío si la temperatura es menor a 8 grados Celsius.

Tuplas

Una manera de formar un nuevo tipo es combinando los otros ya existentes en tuplas *(i.e., haciendo su producto cartesiano). El ejemplo más conocido es, quizas, el de un par de números como (3,2). (3,2) es un elemento de tipo (Num, Num). Por ejemplo, podemos definir una función que toma dos pares y los suma coordenada a coordenada de la siguiente forma*

sumaPares : (Num,Num) → (Num,Num) → (Num,Num)
sumaPares (a, b) (c, d) = (a+c, b+d)

26. Definí las funciones que describimos a continuación, luego implementalas en `haskell`.

    a) *segundo3* : $(Num, Num, Num) \to Num$, que dada una terna de enteros devuelve su segundo elemento.

    b) *ordena* : $(Num, Num) \to (Num, Num)$, que dados dos enteros los ordena de menor a mayor.

    c) *rangoPrecioParametrizado* : $Num \to (Num, Num) \to String$ que dado un número x, que representa el precio de un producto, y un par $(menor, mayor)$ que represente el rango de precios que uno espera encontrar, retorne "muy barato" si x está por debajo del rango, "demasiado caro" si está por arriba del rango, "hay que verlo bien" si el precio está en el rango, y "esto no puede ser!" si x es negativo.

    d) *mayor3* : $(Num, Num, Num) \to (Bool, Bool, Bool)$, que dada una una terna de enteros devuelve una terna de valores booleanos que indica si cada uno de los enteros es mayor que 3.

    Por ejemplo: $mayor3.(1, 4, 3) = (False, True, False)$ y $mayor3.(5, 1984, 6) = (True, True, True)$

    e) *todosIguales* : $(Num, Num, Num) \to Bool$ que dada una terna de enteros devuelva *True* si todos sus elementos son iguales y *False* en caso contrario.

    Por ejemplo: $todosIguales.(1, 4, 3) = False$ y $todosIguales.(1, 1, 1) = True$