Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

Departamento de Computación - FCEyN - UBA

Programación funcional - clase 1

Funciones Simples - Recursión - Tipos de datos

Algoritmos y programas

- Aprendieron a especificar problemas
- El objetivo es ahora pensar algoritmos que cumplan con las especificaciones planteadas
  - Puede haber varios algoritmos que cumplan una misma especificación
- Una vez que se tiene un algoritmo, se escribe un programa que lo implementa
  - Expresión formal de un algoritmo
  - Lenguajes de programación
    - Sintaxis definida
    - Semántica definida
    - Ejemplos: Haskell, C, C++, C#, Java, Smalltalk, Prolog, etc.
- ¿Cómo es un lenguaje típico?
- **¿Cómo se le dice a una computadora lo que tiene que hacer?**

Paradigmas de lenguajes de programación

- Paradigma = definición del modo en el que se especifica el cómputo (que luego es implementado a través de programas)
- Representa una "toma de posición" ante la pregunta: ¿cómo se le dice a la computadora lo que tiene que hacer?
  - Todo lenguaje de programación pertenece a un paradigma
- Estado del arte:
  - Paradigma de programación imperativa: C, Basic, Ada, Clu
  - Paradigma de programación en objetos: Smalltalk
  - Paradigma de programación orientada a objetos: C++, C#, Java
  - Paradigma de programación funcional: LISP, Gopher, Haskell
  - Paradigma de programación en lógica: Prolog

Paradigma de programación funcional

- Programa = colección de funciones
- Recordar que un programa es un aparato que transforma datos de entrada en un resultado
- Los lenguajes funcionales nos dan herramientas para explicarle a la computadora cómo calcular esas funciones
- Herramienta fundamental: definición de funciones

Haskell: programación funcional

Haskell B. Curry (1900–1982)

- **Haskell:** lenguaje de programación funcional
- **Hugs** $\subseteq$ **Haskell**

Expresiones, Valores y Tipos

- Los valores se agrupan en tipos.
  - `Int`, `Float`, `Bool`, `Char`
    (como en el lenguaje de especificación)
  - Hay también tipos compuestos (por ejemplo, pares ordenados)
    y tipos definidos por el programador
- Una expresión es una tira de símbolos que denota un valor.
  - `2`
  - `1+1`
  - `(3*7+1)/11`
  - Todas representan el mismo valor de tipo **Int**.

Formación de expresiones

- Expresiones atómicas (las más simples)
  - También se llaman formas normales
  - Son la forma más intuitiva de representar un valor
  - Ejemplos:
    - `2`
    - `False`
    - `(3, True)`
  - Es común llamarlas "valores" aunque no son un valor, sino que *denotan* un valor, como las demás expresiones
- Expresiones compuestas
  - Se construyen combinando expresiones atómicas con operaciones
  - Ejemplos:
    - `1+1`
    - `1==2`
    - `(4-1, True || False)`

Expresiones mal formadas

- Algunas cadenas de símbolos no forman expresiones
  - Por problemas sintácticos:
    - `+*1-`
    - `(True`
    - `('a',)`
  - ... o por error de tipos:
    - `2 + False`
    - `2 || 'a'`
    - `4 * 'b'`
- Para saber si una expresión está bien formada, aplicamos:
  - Reglas sintácticas
  - Reglas de asignación de tipos (o de inferencia de tipos)

Programa funcional

Un programa en lenguage funcional es un conjunto de ecuaciones que definen una o más funciones.

¿Para qué se usa un programa funcional?

- Para reducir expresiones
- Las ecuaciones orientadas junto con el mecanismo de reducción describen algoritmos (definición de los pasos para resolver un problema)

Ejemplos:

- ```
  doble:: Int -> Int
  doble x = x+x
  ```
- ```
  fst:: (a,b) -> a
  fst (x,y)= x
  ```
- ```
  dist:: (Float,Float) -> Float
  dist (x,y) = sqrt (x^2+y^2)
  ```


Más ejemplos

- ```
  signo:: Int -> Int
  signo 0 = 0
  signo x | x > 0 = 1
  signo x | x < 0 = -1
  ```
- ```
  promedio ::(Float,Float) -> Float
  promedio1 (x,y) = (x+y)/2
  ```
- ```
  promedio2:: Float -> Float -> Float
  promedio2 x y = (x+y)/2
  ```
- ```
  fact:: Int -> Int
  fact 0 = 1
  fact n | n > 0 = n * fact (n-1)
  ```
- ```
  fib:: Int -> Int
  fib 1 = 1
  fib 2 = 1
  fib n | n > 2 = fib (n-1) + fib (n-2)
  ```


Definiciones recursivas

- Propiedades de una definición recursiva
  - Tiene que tener uno o más casos base
  - Las llamadas recursivas del lado derecho tienen que acercarse más al caso base
- En cierto sentido, es el equivalente computacional de la inducción para las demostraciones
- La recursión reemplaza a la necesidad de estructuras de control iterativas (ciclos)

Asegurarse de llegar a un caso base

- Consideremos esta especificación:
  problema $par(n : \text{Int}) = result : \text{Bool}\{$
  requiere $n \geq 0$;
  asegura $result == (n \mod 2 == 0)$;
  $\}$
- ¿Este programa cumple con la especificación?
  ```
  par:: Int -> Bool
  par 0 = True
  par n = par (n-2)
  ```
- Se arregla de alguna de estas formas:
  ```
  par 0 = True
  par 1 = False
  par n = par (n-2)
  ```
  ```
  par 0 = True
  par n = not (par (n-1))
  ```

Ecuaciones

- Usamos ecuaciones para definir funciones.
- Por ejemplo:

```
doble :: Int -> Int
doble x = x + x
```

- Para determinar el valor de la aplicación de una función se reemplaza cada sub expresión por otra, según las ecuaciones
- Ecuaciones orientadas:
  - Lado izquierdo: expresión a definir
  - Lado derecho: definición
  - Cálculo de valor de una expresión: reemplazamos las subexpresiones que sean lado izquierdo de una ecuación por su lado derecho

Reducción

Es el proceso de reemplazar una subexpresión por su definición, sin tocar el resto

- La expresión resultante puede no ser más corta
- ... pero seguramente está "más definida", en el sentido de que más cerca de ser una forma normal

Ejemplo: `doble x = x + x`

`doble (1 + 1)` $\rightsquigarrow$ `(1 + 1) + (1 + 1)`
$\rightsquigarrow$ `2 + (1 + 1)` $\rightsquigarrow$ `2 + 2` $\rightsquigarrow$ `4`

También podría ser:

`doble (1 + 1)` $\rightsquigarrow$ `doble 2`
$\rightsquigarrow$ `2 + 2` $\rightsquigarrow$ `4`

Transparencia referencial

Es la propiedad del lenguaje que garantiza que el valor de una expresión depende exclusivamente de sus subexpresiones.

- Cada expresión del lenguaje representa siempre el mismo valor en cualquier lugar de un programa
- Es muy útil para verificar correctitud (demostrar que se cumple la especificación)
  - Podemos usar propiedades ya probadas para sub-expresiones
  - El valor de una expresión valor no depende de la historia
  - La correctitud vale en cualquier contexto

Es una propiedad muy importante en este paradigma.

- En otros paradigmas, el significado de una expresión puede depender del contexto

Tipos de datos

- En Haskell, todo valor pertenece a algún tipo de datos
  - Las funciones son valores, y también tienen tipo
  - Ejemplo: el tipo "funciones de enteros en enteros"
- Toda expresión bien formada denota un valor. Entonces, toda expresión tiene un tipo (el del valor que representa).
- Haskell es un lenguaje fuertemente tipado
  - No se pueden pasar elementos de un tipo a una operación que espera argumentos de otro
- También tiene tipado estático
  - No hace falta hacer funcionar un programa para saber de qué tipo son sus expresiones
  - El intérprete puede controlar si un programa tiene errores de tipos

Notación para tipos

- `e :: T`
  - La expresión `e` es de tipo `T`
  - El valor denotado por `e` pertenece al conjunto de valores llamado `T`
- Ejemplos:
  - `2 :: Int`
  - `False :: Bool`
  - `'b':: Char`
  - `doble :: Int -> Int`
- Sirve para escribir reglas y razonar sobre Haskell
- También se usa dentro de Haskell
  - Indica de qué tipo queremos que sean los nombres que definimos
  - El intérprete chequea que el tipo coincida con el de las expresiones que lo definen
  - Podemos obviar las declaraciones de tipos pero nos perdemos la oportunidad del doble chequeo

Polimorfismo

- Se llama polimorfismo al hecho de que una función puede aplicarse a distintos tipos de datos, sin redefinirla.
- En Haskell los polimorfismos se escriben usando variables de tipo y conviven con el tipado fuerte.
- Ejemplo de una función polimórfica: la función identidad

```
id :: a -> a
id x = x
```

"`id` es una función que dado un elemento de algún tipo `a` devuelve otro elemento de ese mismo tipo"

Polimorfismo

```
id :: a -> a
id x = x
```

¿De qué tipo es la aplicación de `id` a un valor?

- `id 3 :: Int`, porque instancia `id :: Int -> Int`
- `id True :: True`, porque instancia `id :: Bool -> Bool`
- `id doble :: (Int -> Int)`, porque instancia `id :: (Int -> Int) -> (Int -> Int)`

Aplicación de funciones

En programación funcional (como en matemática) las funciones también son valores.
Notación `f :: T1 -> T2`

- La operación básica que podemos realizar con ese valor es la aplicación
  - Aplicar la función a un elemento para obtener un resultado
- Sintácticamente, la aplicación se escribe como una yuxtaposición (la función seguida de su parámetro).
- Por ejemplo, sea `f :: T1 -> T2`, y `e` de tipo `T1` entonces `f e` es una expresión de tipo `T2`.
  Sea `doble :: Int -> Int`, entonces `doble 2` representa un número entero.

Currificación

- Diferencia entre `promedio1` y `promedio2`
  - `promedioA :: (Float, Float) -> Float`
    `promedioA (x,y) = (x+y)/2`
  - `promedioB :: Float -> Float -> Float`
    `promedio2 x y = (x+y)/2`
- La notación se llama currificación
- En primera instancia, evita el uso de varios signos de puntuación (comas y paréntesis)
  - `promedioA (promedioA (2, 3), promedioA (1, 2))`
  - `promedioB (promedioB 2 3) (promedioB 1 2)`