Programación Funcional vs. Programación Lógica

Joaquín Arias[1]

[1]CETINIA, Universidad Rey Juan Carlos

Introducción: Programación Declarativa

- ¿Que es la programación declarativa?
  - Paradigma de programación diferente a la imperativa (R) o la orientada a objetos (Java).
  - Los programas especifican las propiedades del problema a resolver.
  - La ejecución del programa consiste en "encontrar" la(s) solución(es).

Introducción: Programación Declarativa

- ¿Que es la programación declarativa?
  - Paradigma de programación diferente a la imperativa (R) o la orientada a objetos (Java).
  - Los programas especifican las propiedades del problema a resolver.
  - La ejecución del programa consiste en "encontrar" la(s) solución(es).

Asignación Destructiva vs. Recursión

```r
# Suma lista de números en R
sumaLista <- function(list) {
  sum <- 0
  for (n in list)
    sum <- sum + n
  return(sum) }
# Imprime el resultado de la suma, 10
print(sumaLista(list(1,2,3,4)))
```

```haskell
-- Suma lista de números en Haskell
sumaLista :: [Int] -> Int
sumaLista [] = 0
sumaLista (n : list) = n + (sumaLista list)

-- Imprime el resultado de la suma, 10
main = print (sumaLista [1,2,3,4])
```

Introducción: Programación Declarativa

- ¿Que es la programación declarativa?
  - Paradigma de programación diferente a la imperativa (R) o la orientada a objetos (Java).
  - Los programas especifican las propiedades del problema a resolver.
  - La ejecución del programa consiste en "encontrar" la(s) solución(es).

Asignación Destructiva vs. Recursión

```r
# Suma lista de números en R
sumaLista <- function(list) {
  sum <- 0
  for (n in list)
    sum <- sum + n
  return(sum) }
# Imprime el resultado de la suma, 10
print(sumaLista(list(1,2,3,4)))
```

```haskell
-- Suma lista de números en Haskell
sumaLista :: [Int] -> Int
sumaLista [] = 0
sumaLista (n : list) = n + (sumaLista list)

# Imprime el resultado de la suma, 10
main = print (sumaLista [1,2,3,4])
```

- Ejemplos de programación declarativa:
  - Lenguajes algebraicos: Maude, SQL.
  - Lenguajes lógicos: Prolog, ASP, Logica by Google.
  - Lenguajes funcionales: Haskell, Scala by EPFL.

Programación Funcional

Programación Funcional: Introducción

- La programación funcional esta basada en funciones matemáticas.
- Función: Una función es una regla de correspondencia entre dos conjuntos de tal manera que a cada elemento del primer conjunto le corresponde uno y sólo un elemento del segundo conjunto.
- Cualquier función computable puede expresarse y evaluarse con el cálculo lambda.
- El cálculo lambda fue usado por Church para resolver el Entscheidungsproblem (1936):
  - No hay un algoritmo que determine si dos expresiones lambda arbitraria son equivalentes.

Programación Funcional: Cálculo Lambda

- Introducción al cálculo lambda.
- Reglas de formación de las expresiones lambda (λ-expresiones):
  - `x` es una λ-expresión si `x` es una variable.
  - `(λx.t)` es una λ-expresión (función) si `t` una expresión y `x` una variable.
  - `(t s)` es una λ-expresión (aplicación) si `t` y `s` son expresiones.

Programación Funcional: Cálculo Lambda

- Introducción al cálculo lambda.
- Reglas de formación de las expresiones lambda (λ-expresiones):
  - `x` es una λ-expresión si `x` es una variable.
  - `(λx.t)` es una λ-expresión (función) si `t` una expresión y `x` una variable.
  - `(t s)` es una λ-expresión (aplicación) si `t` y `s` son expresiones.

Evaluando λ-expresiones

Función identidad aplicada al 3: `((λx.x) 3)` $\equiv$ `3`
Función suma aplicada al 2 y el 3: `(((λx.λy.x+y) 2) 3)` $\equiv$ `((λy.2+y) 3)` $\equiv$ `(2+3)`
Función identidad aplicada a la suma: `((λx.x) (λx.λy.x+y))` $\equiv$ `(λx.λy.x+y)`

Programación Funcional: Haskell

- Debe su nombre a Haskell Curry (1900-1982).
- Dada una función f del tipo $f : (X_1 \times X_2 \times \cdots \times X_n) \to Z$ decimos que su currificación es:
  - Una secuencia de funciones con un único argumento:
    $$curry(f) : X_1 \to X_2 \to \cdots \to X_n \to Z.$$

haskell.org

```haskell
-- Suma a y b
suma :: Int -> Int -> Int                    -- declaración de la función suma
suma a b = a + b                             -- definición con dos argumentos
suma = \a -> \b -> a + b                     -- definición currificada
-- Sucesor de a
sucesor :: Int -> Int
sucesor = suma 1                                          -- aplicación parcial
-- Toma una función y una lista y devuelve una lista
aplica :: (Int -> Int) -> [Int] -> [Int]
aplica _ [] = []
aplica f (n : list) = ((f n) : (aplica f list))

main = print (aplica sucesor [1,3,4])                     -- imprime [2,4,5]
```

Programación Funcional: Booleanos en cálculo lambda

- Primero implementamos las expresiones If-then-else, True y False:
  - If-then-else: `λx.λy.λz.x y z`
  - true: `λx.λy.x`
  - false: `λx.λy.y`

Programación Funcional: Booleanos en cálculo lambda

- Primero implementamos las expresiones If-then-else, True y False:
  - If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
  - true: $\lambda x.\lambda y.x$
  - false: $\lambda x.\lambda y.y$

If-then-else True P Q $\equiv$

Programación Funcional: Booleanos en cálculo lambda

- Primero implementamos las expresiones If-then-else, True y False:
  - If-then-else: $\lambda x.\lambda y.\lambda z.x\ y\ z$
  - true: $\lambda x.\lambda y.x$
  - false: $\lambda x.\lambda y.y$

If-then-else True P Q $\equiv (\lambda x.\lambda y.\lambda z.\ x\ y\ z)\ (\lambda x.\lambda y.x)\ P\ Q \equiv (\lambda x.\lambda y.x)\ P\ Q \equiv P$

- Implementación usando Haskell:

```
1  if_then_else = \x -> \y -> \z -> x y z
2  true = \x -> \y -> x
3  false = \x -> \y -> y
4
5  k = if_then_else true 3 2          -- ¿cuánto vale k?
```

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Luego, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ false \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ true\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ false\ true \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Luego, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ false \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ true\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ false\ true \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

And True False $\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Luego, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ false \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ true\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ false\ true \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned} And\ True\ False &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\ &(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv False \end{aligned}$$

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Luego, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ false \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ true\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ false\ true \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

And True False $\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$
$(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv$ False
Or True False $\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv$

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Luego, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ false \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ true\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ false\ true \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\text{And True False} \equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False}$$

$$\text{Or True False} \equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True}$$

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Luego, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ false \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ true\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ false\ true \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned}
&\text{And True False} \equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\
&\qquad (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv \text{False} \\
&\text{Or True False} \equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\
&\qquad (\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv \text{True} \\
&\text{Not True} \equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv \ldots \equiv (\lambda x.\lambda y.y) \equiv \text{False}
\end{aligned}$$

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Luego, basadas en estas expresiones definimos And, Or y Not:
  - And: $\lambda p.\lambda q.p\ q\ false \equiv \lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y)$
  - Or: $\lambda p.\lambda q.p\ true\ q \equiv \lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q$
  - Not: $\lambda p.p\ false\ true \equiv \lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x)$

$$\begin{aligned}
And\ True\ False &\equiv (\lambda p.\lambda q.p\ q\ (\lambda x.\lambda y.y))\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\
&(\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2)\ (\lambda x.\lambda y.y) \equiv (\lambda x_2.\lambda y_2.y_2) \equiv False \\
Or\ True\ False &\equiv (\lambda p.\lambda q.p\ (\lambda x.\lambda y.x)\ q)\ (\lambda x_1.\lambda y_1.x_1)\ (\lambda x_2.\lambda y_2.y_2) \equiv \\
&(\lambda x_1.\lambda y_1.x_1)\ (\lambda x.\lambda y.x)\ (\lambda x_2.\lambda y_2.y_2) \equiv (\lambda x.\lambda y.x) \equiv True \\
Not\ True &\equiv (\lambda p.p\ (\lambda x.\lambda y.y)\ (\lambda x.\lambda y.x))\ (\lambda x_1.\lambda y_1.x_1) \equiv \ldots \equiv (\lambda x.\lambda y.y) \equiv False
\end{aligned}$$

- Implementación usando Haskell (cont.):

```
5  my_and = \x -> \y -> x y false
6  my_or = \x -> \y -> x true y
7  my_not = \x -> x false true
8
9  k = if_then_else (my_and true false) 3 2          -- ¿cuánto vale k?
```

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Aunque podríamos considerar otras expresiones para And, Or y Not:
  - And_2: $\lambda p.\lambda q.p\ q\ p$
  - Or_2: $\lambda p.\lambda q.p\ p\ q$
  - Not_2: $\lambda p.\lambda x.\lambda y.p\ y\ x$

¿Cuántos argumentos tiene?

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Aunque podríamos considerar otras expresiones para And, Or y Not:
  - And_2: $\lambda p.\lambda q.p\ q\ p$
  - Or_2: $\lambda p.\lambda q.p\ p\ q$
  - Not_2: $\lambda p.\lambda x.\lambda y.p\ y\ x$

¿Cuántos argumentos tiene?

Deberes para casa

Programación Funcional: Booleanos en cálculo lambda (cont.)

- Aunque podríamos considerar otras expresiones para And, Or y Not:
  - And_2: `λp.λq.p q p`
  - Or_2: `λp.λq.p p q`
  - Not_2: `λp.λx.λy.p y x`

¿Cuántos argumentos tiene?

Deberes para casa

- Implementación (requiere tipos) usando Haskell (cont.):

```haskell
{-# LANGUAGE Rank2Types #-}
type CB = forall a . a -> a -> a

my_and :: CB -> CB -> CB
my_and = \p -> \q -> p q p
my_or :: CB -> CB -> CB
my_or = \p -> \q -> p p q
my_not :: CB -> CB                    -- parece que tiene 1 argumento
my_not = \p -> \x -> \y -> p y x

k = if_then_else (my_not false) 3 2   -- ¿cuánto vale k?
```

Programación Funcional: Características y Ventajas

- Características:
  - Evaluación de funciones vs. ejecución de instrucciones (recursión vs. iteración).
  - El valor de una función sólo depende de sus argumentos (siempre se obtiene el mismo valor para los mismos argumentos: transparencia referencial).
  - Las funciones son "ciudadanos de primera clase" (argumentos y/o valores)
- Ventajas:
  - Código más limpio, conciso y expresivo.
  - Sin efectos secundarios, al ser el estado inmutable.
    - Adecuado para sistemas concurrentes/paralelos.
  - Permite verificación formal y demostración automática.

Programación Funcional: Características y Ventajas

- Características:
  - Evaluación de funciones vs. ejecución de instrucciones (recursión vs. iteración).
  - El valor de una función sólo depende de sus argumentos (siempre se obtiene el mismo valor para los mismos argumentos: transparencia referencial).
  - Las funciones son "ciudadanos de primera clase" (argumentos y/o valores)
- Ventajas:
  - Código más limpio, conciso y expresivo.
  - Sin efectos secundarios, al ser el estado inmutable.
    - Adecuado para sistemas concurrentes/paralelos.
  - Permite verificación formal y demostración automática.

Concatenar listas

```
1  concatenar :: [a] -> [a] -> [a]                -- declaración de la función
2  concatenar [] list = list                      -- caso base
3  concatenar (x:xs) list = (x: (concatenar xs list))  -- llamada recursiva
4
5  k = concatenar [1,2] [3,4]                     -- ¿cuánto vale k?
```

Programación Lógica

Programación Lógica: Introducción

- La programación lógica esta basada en lógica de 1^{er} orden (LPO).
- Predicados: Un predicado es una afirmación sobre propiedades de un objeto y/o una relación entre dos o más objetos.
- A partir de un conjunto de fórmulas en LPO podemos inferir nuevo conocimiento.
- Aristóteles formalizó el razonamiento humano (s. IV a.C.):
  - Todos los hombres son mortales. Sócrates es un hombre. Luego Sócrates es mortal.
  - Cuya forma normal clausular es:

$$\{\neg Hombre(x) \vee Mortal(x), Hombre(socrates), \neg Mortal(socrates)\}$$

  - Aplicando resolución de Robinson con unificación se deduce la cláusula vacía por lo tanto el argumento es válido.

Programación Lógica: Cláusulas de Horn

- Introducción a las cláusulas de Horn, definidas por Alfred Horn en 1951.
- Dada una cláusula (disyunción de literales) cualquiera $L_1 \vee L_2 \vee \cdots \vee L_n$, es una cláusula de Horn si tiene como máximo un literal positivo y esta reescrita como un implicación:
  - $\neg p \vee \neg q \vee \cdots \vee \neg t \vee u$ es una regla y se reescribe como $p \wedge q \wedge \cdots \wedge t \rightarrow u$
  - u, sin literales negados, es un hecho y se reescribe como u
  - $\neg p \vee \neg q \vee \cdots \vee \neg t$, sin literal positivo, es una consulta $p \wedge q \wedge \cdots \wedge t \rightarrow$

Programación Lógica: Cláusulas de Horn

- Introducción a las cláusulas de Horn, definidas por Alfred Horn en 1951.
- Dada una cláusula (disyunción de literales) cualquiera $L_1 \vee L_2 \vee \cdots \vee L_n$, es una cláusula de Horn si tiene como máximo un literal positivo y esta reescrita como un implicación:
  - $\neg p \vee \neg q \vee \cdots \vee \neg t \vee u$ es una regla y se reescribe como $p \wedge q \wedge \cdots \wedge t \rightarrow u$
  - u, sin literales negados, es un hecho y se reescribe como u
  - $\neg p \vee \neg q \vee \cdots \vee \neg t$, sin literal positivo, es una consulta $p \wedge q \wedge \cdots \wedge t \rightarrow$

Implementación en Prolog

$\neg Hombre(x) \vee Mortal(x)$:
```
mortal(X) :- hombre(X).
```
$Hombre(socrates)$:
```
hombre(socrates).
```
$\neg Mortal(socrates)$:
```
?- mortal(socrates).
```

Programación Lógica: Prolog

- Existen diversos interpretes: Ciao Prolog, Swi-Prolog, etc.
- Dada una función f del tipo $f : (X_1 \times X_2 \times \cdots \times X_n) \to Z$ se puede definir un predicado F:
  - Que relaciona los argumentos de entrada con la salida: $F : (X_1 \times X_2 \times \cdots \times X_n \times Z) \to \{true, false\}$.

swi-prolog.org

Concatenar listas

```
1  concatenar([],Lista,Lista).
2  concatenar([X|Xs],Lista,[X|N_Lista]) :- concatenar(Xs,Lista,N_Lista).
3
4  ?- concatenar([1,2],[3,4],Lista).                    % ¿Cuanto vale Lista?
```

- Al evaluar una consulta sin variables, p.ej., `?- mortal(socrates)`, Prolog contesta `yes` si la consulta es consecuencia lógica del programa (`no` en caso contrario).
- Al evaluar una consulta con variables, p.ej., `?- concatenar([1,2],[3,4],Lista)`, Prolog devuelve la(s) sustitución(es) que la hace(n) consistente: `Lista = [1,2,3,4]`

Programación Lógica: ... hay algo más

- Mientras las funciones se evalúan a un único resultado...
  ... los predicados pueden "consultarse" de diferentes formas.

Programación Lógica: ... hay algo más

- Mientras las funciones se evalúan a un único resultado...
  ... los predicados pueden "consultarse" de diferentes formas.
  - La consulta `?- concatenar([1,2], Lista, [1,2,3,4]` devuelve `Lista = [3,4]`

Programación Lógica: ... hay algo más

- Mientras las funciones se evalúan a un único resultado...
  ... los predicados pueden "consultarse" de diferentes formas.
  - La consulta `?- concatenar([1,2], Lista, [1,2,3,4]` devuelve `Lista = [3,4]`
  - ... y `?- concatenar(ListaA, ListaB, [1,2,3,4]` devuelve 5 respuestas:

Programación Lógica: ... hay algo más

- Mientras las funciones se evalúan a un único resultado...

  ... los predicados pueden "consultarse" de diferentes formas.

  - La consulta `?- concatenar([1,2], Lista, [1,2,3,4]` devuelve `Lista = [3,4]`
  - ... y `?- concatenar(ListaA, ListaB, [1,2,3,4]` devuelve 5 respuestas:
    1. `ListaA = [], ListaB = [1,2,3,4]`
    2. `ListaA = [1], ListaB = [2,3,4]`
    3. `ListaA = [1,2], ListaB = [3,4]`
    4. `ListaA = [1,2,3], ListaB = [4]`
    5. `ListaA = [1,2,3,4], ListaB = []`

Programación Lógica: ... hay algo más

- Mientras las funciones se evalúan a un único resultado...
  ... los predicados pueden "consultarse" de diferentes formas.
  - La consulta `?- concatenar([1,2], Lista, [1,2,3,4]` devuelve `Lista = [3,4]`
  - ... y `?- concatenar(ListaA, ListaB, [1,2,3,4]` devuelve 5 respuestas:
    1. `ListaA = [], ListaB = [1,2,3,4]`
    2. `ListaA = [1], ListaB = [2,3,4]`
    3. `ListaA = [1,2], ListaB = [3,4]`
    4. `ListaA = [1,2,3], ListaB = [4]`
    5. `ListaA = [1,2,3,4], ListaB = []`
- Esto permite implementar un único predicado para codificar/decodificar mensajes:

Traductor código Morse

```
1  char2morse('A','.-').   char2morse('B','-...').   char2morse('C','-.-.').  ...
2
3  ?- char2morse('B', Morse)                    % devuelve Morse = '-...'
4  ?- char2morse(Char, '-.-.')                  % devuelve Char = 'C'
```

Programación Lógica + Restricciones (CLP): ... mucho más

- Las restricciones nos permite expresar relaciones entre variables mediante ecuaciones
- P.ej., podemos definir la relación de una hipoteca como:

P=principal, T=time periods, R=repayment each period, I=interest rate, B=balance owing.

```
mg(P, T, _, _, B) :- T = 0, B = P.
mg(P, T, R, I, B) :- T >= 1, NP = P + P*I - R, NT = T - 1, mg(NP, NT, R, I, B).
```

Igualmente podemos preguntar de diferentes maneras ... muy diferentes.

Programación Lógica + Restricciones (CLP): ... mucho más

- Las restricciones nos permite expresar relaciones entre variables mediante ecuaciones
- P.ej., podemos definir la relación de una hipoteca como:

P=principal, T=time periods, R=repayment each period, I=interest rate, B=balance owing.

```
mg(P, T, _, _, B) :- T = 0, B = P.
mg(P, T, R, I, B) :- T >= 1, NP = P + P*I - R, NT = T - 1, mg(NP, NT, R, I, B).
```

Igualmente podemos preguntar de diferentes maneras ... muy diferentes.

```
?- mg(1000, 10, 150, 0.10, B).
  B = 203.13 ?
```

```
?- mg(P, 10, 150, 0.10, 0).
  P = 921.68 ?
```

Programación Lógica + Restricciones (CLP): ... mucho más

- Las restricciones nos permite expresar relaciones entre variables mediante ecuaciones
- P.ej., podemos definir la relación de una hipoteca como:

P=principal, T=time periods, R=repayment each period, I=interest rate, B=balance owing.

```
mg(P, T, _, _, B) :- T = 0, B = P.
mg(P, T, R, I, B) :- T >= 1, NP = P + P*I - R, NT = T - 1, mg(NP, NT, R, I, B).
```

Igualmente podemos preguntar de diferentes maneras ... muy diferentes.

```
?- mg(1000, 10, 150, 0.10, B).
  B = 203.13 ?

?- mg(P, 10, 150, 0.10, 0).
  P = 921.68 ?

?- mg(P, 10, R, 0.10, B).
  P = 6.14*R + 0.38*B ?
```

Programación Lógica + Restricciones (CLP): ... mucho más

- Las restricciones nos permite expresar relaciones entre variables mediante ecuaciones
- P.ej., podemos definir la relación de una hipoteca como:

P=principal, T=time periods, R=repayment each period, I=interest rate, B=balance owing.

```
mg(P, T, _, _, B) :- T = 0, B = P.
mg(P, T, R, I, B) :- T >= 1, NP = P + P*I - R, NT = T - 1, mg(NP, NT, R, I, B).
```

Igualmente podemos preguntar de diferentes maneras ... muy diferentes.

```
?- mg(1000, 10, 150, 0.10, B).
  B = 203.13 ?

?- mg(P, 10, 150, 0.10, 0).
  P = 921.68 ?

?- mg(P, 10, R, 0.10, B).
  P = 6.14*R + 0.38*B ?
```

- Esto permite ... mucho más:

Multiplicar matrices y/o obtener la inversa

```
multiplicar(MatrizA, MatrizB, MatrizC) :- ...

?- multiplicar([[1,2],[2,5]], [[3,1],[4,1]], Mult)        % Mult = [[11,3],[26,7]]
?- multiplicar([[1,2],[2,5]], Inversa, [[1,0],[0,1]])     % Inversa = [[5,-2],[-2,1]]
```

Para terminar

UTD HackReason 20xx: 14-15 Enero (Día Mundial de la Lógica)

UTD HackReason 20xx: 14-15 Enero (Día Mundial de la Lógica)