Lenguajes Funcionales

Implementación

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Objetivos

Objetivos

Los objetivos de este tema son:

- Presentar los elementos básicos de un lenguaje funcional.
- Presentar las implementaciones de esos elementos en distintos lenguajes y ver sus ventajas e inconvenientes.
- Analizar con más profundidad algunos detalles de la implementación para entender el coste de ciertas operaciones y su diseño en los lenguajes funcionales.

Introducción

La Programación Funcional es un paradigma de programación que trata los cálculos y los algoritmos como la evaluación de funciones matemáticas.

Evita los conceptos de estado del programa y mutabilidad.

Utiliza la evaluación de funciones en lugar de la modificación del estado del programa (variables).

Un lenguaje funcional permite:

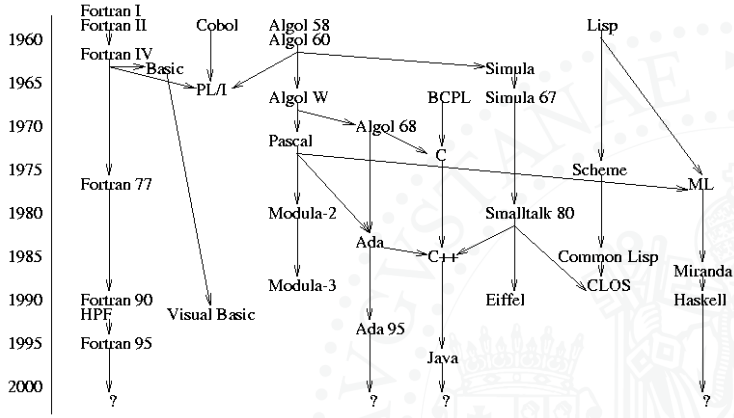
- Implementación pura de funciones.
- Recursión.
- Funciones de Primera Clase.
- Formas Funcionales.
- Currying.
- Estrategias de evaluación no estricta.

Un lenguaje funcional ofrece como base:

- un mecanismo de aplicación de funciones (*evaluación*)
- un conjunto de funciones primitivas
- un conjunto de *formas funcionales* para combinarlas (*composición*)
- algún tipo de estructura de datos para almacenar información (vectores, listas, tuplas, etc)

Los lenguajes funcionales han ido evolucionando como cualquier otra rama de los lenguajes de programación:

- Lisp (Scheme, Common Lisp, Erlang, Clojure)
- ML (CAML, OCAML, Nemerle, F#)
- Miranda
- Haskell


1960
1965
1970
1975
1980
1985
1990
1995
2000
Fortran I
Fortran II
Fortran IV
Basic
Cobol
PL/I
Algol 58
Algol 60
Algol W
Algol 68
Pascal
Simula
BCPL
Simula 67
C
Lisp
Scheme
ML
Fortran 77
Modula-2
Ada
C++
Smalltalk 80
Common Lisp
CLOS
Miranda
Haskell
Modula-3
Eiffel
Fortran 90
HPF
Visual Basic
Ada 95
Java
Fortran 95
?
?
?
?

Lisp

Lisp es el lenguaje funcional más antiguo y más ampliamente utilizado todavía.

Se inventó como un lenguaje para el procesado de listas, cuya necesidad creció con la aparición del área de la Inteligencia Artificial.

El interés por la Inteligencia Artificial apareció a mediados de los años 50 con el objetivo de estudiar problemas como el procesado del lenguaje natural, el almacenamiento y la recuperación de la información por parte del cerebro humano, la demostración automática de teoremas,...

Lo que todos esos problemas tenían en común era la necesidad de desarrollar métodos para procesar listas de datos simbólicos.

La estructura de datos más utilizada en ese momento era el vector de números.

El concepto de **procesado de listas** fue desarrollado por Newell, Shawn y Simon, que desarrollaron una teoría y el lenguaje **IPL**. Su bajo nivel impidió su aceptación, pero demostraron la posibilidad del procesado de listas mediante computador.

IBM se interesó a mediados de los 50 en el área de la demostración automática de teoremas. En esa época estaba en desarrollo el proyecto FORTRAN, e IBM decidió que el procesado de listas debía hacerse utilizando el FORTRAN.

De esta decisión salió el **FLPL** (*FORTRAN List Processing Language*).

Ninguno de los dos lenguajes tuvo mucho éxito ni se popularizó.

En el verano de 1958, **John McCarthy**, que trabajaba en el MIT, pasó una estancia en el IBM Information Research Department.

Su trabajo consistió en investigar sobre el cálculo simbólico y desarrollar un conjunto de especificaciones para un sistema que permitiera poder realizar dichos cálculos de forma automática.

Como ejemplo piloto eligió el problema de derivación de expresiones algebraicas.

De su estudio extrajo la conclusión de que la herramienta que necesitaba debía ofrecer:

- Tratamiento recursivo de funciones.
- Expresiones condicionales.
- Control implícito de la gestión de memoria.

Dado que el lenguaje FLPL (básicamente Fortran) no soportaba ninguna de estas especificaciones se puso de manifiesto que había que desarrollar otro lenguaje.

Cuando McCarthy volvió al MIT, se unió con M. Minsky y formaron el *MIT AI Project*.
Su resultado fue la primera versión de LISP que se suele denominar **LISP puro**.

LISP ha evolucionado durante muchos años pero en la actualidad su diseño no contempla los últimos conceptos de los lenguajes funcionales.
Además todos los dialectos del LISP incluyen características típicas de los lenguajes imperativos, tales como variables, sentencias de asignación, o iteración.

Existen sólo dos tipos de valores:

- Atomos
- Listas

Atomos

Son secuencias de caracteres numéricos o alfabéticos, algunas de ellas con significados predefinidos:

6800
T , t
NIL , nil
A
patata

Los átomos simbólicos (átomos de caracteres) son indivisibles.

Listas

Son estructuras de datos lineales.
Los elementos de las listas pueden ser átomos u otras listas.

Se especifican delimitando sus elementos entre paréntesis.

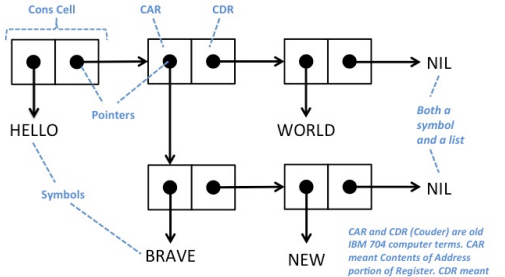
```
1 (A B C D)
2 (A (B C) D (E (F G)))
3 ()
4 NIL
```

La implementación de las listas es la típica lista enlazada.

Listas

La implementación de las listas es la típica lista enlazada.

```
1 (HELLO (BRAVE NEW) WORLD)
2
3 CONS = REG = [CAR,CDR]
```



McCarthy creyó que el procesado de listas podría servir para estudiar de una forma más natural la computabilidad, que en aquel tiempo se estudiaba utilizando las máquinas de Turing.

En el caso de las máquinas de Turing, se puede construir una máquina universal de Turing capaz de copiar el funcionamiento de cualquier otra máquina de Turing.

De este concepto vino la idea de construir una función LISP universal capaz de evaluar cualquier otra función LISP.

A partir de esa idea, se tomaron las siguientes decisiones:

- Utilizar la misma notación para expresar las funciones que para expresar los datos.
- Como ya se había elegido para los datos la notación mediante listas con paréntesis se decidió inventar convenciones para la definición de funciones y llamadas a las funciones de la misma manera.

Para las llamadas a las funciones se eligió la forma prefija:

```
1 (nombre-funcion argumento-1 ... argumento-n)
```

Se definió la funcion universal **eval** que podria evaluar cualquier otra función lisp.
Un intérprete de Lisp es un bucle infinito que lee un dato y se lo pasa a la función **eval** (**REPL**, *read-eval-print loop*):

- si el dato es un átomo, el resultado es el propio átomo
- si el dato es una lista, se supone que es una llamada a una función
- para no evaluar una lista se usa *quote*-ing.

Lisp **REPL**:

```
[1]> 1
1
[2]> (+ 1 2)
3
[3]> (1 2 3)
*** - EVAL: 1 is not a function name...
[4]> '(1 2 3)
(1 2 3)
```

Las listas permiten representar tanto datos como funciones, de manera simbólica y en notación prefija.
Este tratamiento unifica el concepto de datos y código, con lo que el código se puede ver como datos (valor de primera clase) y los datos se pueden ejecutar (si forman un trozo de código válido).

$x + y$	::=	`(plus x y) ::= (+ x y)`
$2x + 3y$	::=	`(+ (* 2 x) (* 3 y))`
$ax^2 + bx + c$	::=	`(+ (* a (exp x 2)) (+ (* b x) c))`

Eso permite construir código nuevo en tiempo de ejecución...

Las listas permiten representar tanto datos como funciones, de manera simbólica y en notación prefija.

Eso permite construir código nuevo en tiempo de ejecución...

```
[1]> (setq f '(+ 2 3))
(+ 2 3)
[2]> f
(+ 2 3)
[3]> (eval f)
5
[4]> (setq g (cons '* (cdr f)))
(* 2 3)
[5]> g
(* 2 3)
[6]> (eval g)
6
```

Funciones primitivas:

- Manejo de listas: `car, cdr, cons, list, member, append`
- Predicados: `eq, atom, null`
- Definición de funciones: `defun`
- Variables: `set setq`

Manejo de listas:

```
[3]> (car '(a b c))
A
[4]> (cdr '(a b c))
(B C)
[5]> (cons 'a '(b c))
(A B C)
[6]> (cons '(a) '(b c))
((A) B C)
```

Como implementaríais esas funciones a bajo nivel ?

Definición de funciones (polimórficas):

```
1  [1]> (defun suma (x y) (+ x y))
2  SUMA
3  [2]> (suma 1 2)
4  3
5
6  [5]> (defun average (x y) (/ (+ x y) 2))
7  AVERAGE
8  [6]> (average 4 5)
9  9/2
10 [7]> (average 4.0 5.0)
11 4.5
```

Expresiones condicionales:

```
1   (defun stepf (x)
2     (if (<0 x) 0 1)
3   )
4
5   (defun rectf (a b x)
6     (cond ((< x a) 0)
7           ((> x b) 0)
8           (t       1)
9     )
10  )
```

Recursividad:

```
1  (defun factorial (n)
2    (if (= 0 n)
3      1                          ; caso base
4      (* n (factorial (- n 1)))  ; caso recursivo
5    )
6  )
7
8  (factorial 4)
9  24
```

Funciones como valores de primera clase y funciones de orden superior:

```
1  [29]> (defun sqr (x) (* x x))
2  SQR
3  [32]> (mapcar 'sqr '(1 2 3 4 5))
4  (1 4 9 16 25)
```

Funciones anónimas (funciones *lambda*):

```
[33]> (mapcar (lambda (x) (+ 1 x)) '(3 4 5))
(4 5 6)
```

Common Lisp es una versión posterior al denominado Lisp Puro, que combina características de otros lenguajes y dialectos de Lisp en un lenguaje estándar:

- variables, locales y gobales, con alcance estático y dinámico
- tipos de datos
- otras estructuras de datos: registros, vectores, cadenas...
- herramientas para la encapsulación de módulos
- extensiones para orientación a objetos (CLOS)

ML

ML son las siglas de **Meta Language**.

Desarrollado por Robin Miller hacia 1970.

Objetivo: Mecanismo de pruebas formales asistidas por computador para el sistema de Lógica para Funciones Computables de Edimburgo. También se le encontró utilidad como lenguaje general para la manipulación de símbolos.

Standard definido en 1983.

Características principales:

- tipado fuerte para ofrecer seguridad
- polimorfismo para soportar componentes genéricos
- inferencia de tipos
- tipos de datos algebraicos
- ajuste de patrones
- modularidad

Tipos de datos:

- simples: `bool, int, real, string`
- compuestos: listas, registros, tuplas, funciones

Declaraciones:

```
1  val i : int;
```

La **lista homogénea** es el tipo fundamental de estructura de datos:

```
1  [1,2,3,4]
2  ['a', 'b', 'c']
3  [true, false]
4  [], nil
```

Se pueden declarar como:

- notación elegante: τ list
- notación ASCII: 't list

Operadores sobre listas:

- hd

```
- hd [3,4,5];
val it = 3 : int
```

- tl

```
- tl [3,4,5];
val it = [4,5] : int list
```

- ::

```
- 3::[4,5];
val it = [3,4,5] : int list
```

Conjunto de objetos no homogéneo, no modificable:

```
1  - (2,"a",3.5);
2  val it = (2,"a",3.5) : int * string * real
3  - #2 it;
4  val it = "a" : string
```

Se puede declarar alias de tipos:

```
type intpair = int * int;
```

No son tipos nuevos, simplemente son alias.

Soporta la definición de tipos polimórficos:

```
type 't pair = 't * 't;
val a = (2,3) : int pair;
```

Las variables no son huecos de memoria, son referencias a objetos que se construyen mediante llamadas a funciones.

Se pueden construir funciones mediante el operador **fn**:

```
1  val succ = fn (x: int) : int => x + 1;
```

Existe una version abreviada:

```
1  fun succ (x: int) : int = x + 1;
```

La inferencia de tipos nos permite omitir las signaturas de algunos parámetros y resultados:

```
1  - fun succ x = x+1;
2  val succ = fn : int -> int
3
4  - fun succ x = x+1.0;
5  val succ = fn : real -> real
```

La inferencia de tipos nos permite omitir las signaturas de algunos parámetros y resultados:

```
- fun add (x:int) (y:int) :int = x + y;
val add = fn : int -> int -> int

- fun add (x:int) y = x + y;
val add = fn : int -> int -> int
```

Este proceso se conoce como sistema de tipos de Hindley–Milner.

Ofrece expresiones condicionales:

```
fun factorial n =
    if n = 0 then 1
             else n * factorial (n-1);
```

Y expresiones bloque (con definiciones locales):

```
fun dist (x0,y0) (x1,y1) =
  let
     val dx = x1 - x0
     val dy = y1 - y0
  in
     sqrt (dx * dx + dy * dy)
  end
```

Podemos definir ecuaciones con guardas y ajuste de patrones (*pattern matching*) como alternativa a las expresiones condicionales:

```
1  fun factorial n =
2        if n = 0 then 1
3                 else n * factorial (n-1)
4
5  fun factorial 0 = 1
6    | factorial n = n * factorial (n - 1)
7
8  fun len [] = 0
9    | len (_::xs) = 1 + len xs
```

Podemmos definir tipos de datos recursivos:

```
1  datatype 't list = null
2                   | node of ('t * 't list);
3
4  datatype 't tree = null
5                   | node of ('t tree * 't * 't tree);
```

Notas:

- Un programador imperativo debe definir un procedimiento que inserte un nuevo entero modificando un árbol dado.
- Un programador funcional debe definir una función de inserción que devuelva un nuevo árbol con el nuevo entero insertado. El árbol original permanecerá sin modificación.

Inserción en un árbol ordenado:

```
1  fun insert newitem null = node (null, newitem, null)
2    | insert newitem node(left, olditem, right) =
3        if newitem <= olditem
4           then node (insert newitem left, olditem, right)
5           else node (left, olditem, insert newitem right)
```

La curryficación (o *currying*) es una forma especial de representar funciones en lenguajes funcionales.
Tomemos la definición de la función power:

```
1  fun power (n,b) =
2        if n=0 then 1.0
3               else b * power (n-1,b);
```

Podemos también escribirla como:

```
1  fun powerc n b =
2        if n=0 then 1.0
3               else b * powerc n-1 b;
```

Cuál es la diferencia ?

- power (n, b): recibe una tupla con dos parámetros
- powerc n b: recibe dos parámetros independientes

Cualquier función con varios parámetros permite evaluación parcial. Se dice que está en *forma de Curry*:

```
1  val sqr  = powerc 2;
2  val cube = powerc 3;
3
4  val ocho = cube 2;
5  val nueve = sqr 3;
```

- filter:

```
filter : ('t -> bool) -> 't list -> 't list
```

- map:

```
map    : ('t -> 's)   -> 't list -> 's list
```

Otros lenguajes

Otros lenguajes

- Scheme
- Miranda
- **Haskell**

Lenguajes Funcionales

Implementación

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20