Código Fuente

Apuntes de programación. Ejemplos de código en PHP, C, Javascript, jQuery, Pascal, Java y C++.

Entradas populares

Arreglos o arrays en C++

Un arreglo en C++ es un conjunto de datos que se alamacenan en memoria de manera contigua con el mismo nombre. Para difenciar los elementos ...

Ejemplo de switch en java

El siguiente es un ejemplo del uso de la sentencia swtich en Java /** Ejemplo de uso switch. ***/ public class Ejercicio03 { ...

Suma de dos números en JAVA

//----JAVA SUMA DE DOS VARIABLES. import javax.swing.JOptionPane; public class SumarNumeros { public static void main(String args[]...

El método SPLIT en Java

Veamos un ejemplo sencillo de como utilizar el método split en Java. En este ejemplo convertiremos una cadena a un array de cadenas. pu...

Cálculo de raíces cuadradas en java

El siguiente ejemplo muestra como hacer uso de java.lang.Math para el cálculo de raíces cuadradas. class raices { private static ...

Como leer un archivo de texto en Java

En el siguiente ejemplo se muestra como leer un archivo de texto utilizando Java. El archivo se lee línea por línea, y a medida que avanza l...

Números aleatorios en C++

Veamos un ejemplo que genere números aleatorios del 1 al 20. //Ejemplo generar numeros

2/21/2008

Las raíces de Lisp

Traducción

El siguiente documento es una traducción bastante libre del documento publicado por Paul Graham en su sitio web [The Roots Of Lisp], es un ensayo sobre los fundamentos del lenguaje de programación Lisp.

Autor: Paul Graham, borrador Enero 18, 2002

En 1960, John McCarthy publicó un notable paper en el cuál, hizo por la programación algo parecido a lo que Euclides hizo por la geometría [1]. Mostró de que manera con un puñado de operadores simples y una notación para funciones, se puede construir un lenguaje de programación. Llamó a este lenguaje Lisp, por "List Processing", debido a que una de sus principales ideas era usar una estructura de datos simple llamada *list*, para el código y los datos.

Vale la pena comprender lo que McCarthy descubrió, no sólo como un hito en la historia de las computadores, sino como un modelo de lo que la programación está tendiendo a convertirse en nuestro propio tiempo. Parece que se han producido dos claros modelos consistentes de programación hasta la fecha: el modelo de C y el modelo de Lisp. Una receta popular para los nuevos lenguajes de programación en los pasados 20 años fue tomar el modelo de C y agregarle, pequeñas partes tomadas del modelo de Lisp, tales como tipado en tiempo de ejecución y recolección de basura.

En este artículo se intentará explicar en los términos más simples posibles lo que McCarthy descubrió. El punto no es sólo aprender acerca de un interesante resultado teórico que alguien averiguó hace cuarenta años, sino mostrar de dónde los lenguajes partieron. La característica inusual de Lisp -en realidad, la definición de la calidad de Lisp- es que puede escribirse a si mismo. Para entender lo que McCarthy quiere decir con esto, vamos a recorrer sus pasos, con su notación matemática traducida en código Common Lisp ejecutable.

aleatorios del 1 al 20. #include <iostream> ...

Arreglo de caracteres en C++

Hasta el momento vimos ejemplos de arreglos enteros. De la misma manera podríamos crear un arreglo de caracteres, lo que nos permite manip...

Busqueda binaria en C++

La búsqueda binaria sólo se puede implementar si el arreglo está ordenado. La idea consiste en ir dividiendo el arreglo en mitades. Por ...

Comparación de cadenas: strcmp y strncmp en C++

La función strcmp: //prototipo de strcmp int strcmp (const char *cadena1, const char *cadena2); Sirve para comparar la cadena cadena1 ...

enlaces

Learn Spanish

Truco

Jugar al Domino Diamond Online

1. Siete operadores primitivos

Para empezar, se define una *expresión*. Una expresión es a la vez un *átomo*, el cual es una secuencia de letras (e.g. `foo`), o una *lista* de cero o más expresiones, separadas por espacios en blanco y encerradas por paréntesis. Algunas expresiones:

```
foo
()
(foo)
(foo bar)
(a b (c) d)
```

La última expresión es una lista de cuatro elementos, el tercero de ellos es a su vez una lista de un elemento.

En aritmética la expresión 1 + 1 tiene el valor 2. Las expresiones válidas en Lisp también tienen valores. Si de una expresión `e` se obtiene el valor `v`, se dice que `e` *devuelve* `v`. El próximo paso es definir que tipo de expresiones puden existir, y que valor devuelve cada tipo.

Si una expresión es una lista, se llama al primer elemento el *operador* y a los elementos restantes los *argumentos*. Se van a definir siete operadores primitivos (en el sentido de axiomas): `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, y `cond`.

1. (quote x)

`(quote x)` devuelve `x`. Por legibilidad abreviamos `(quote x)` como `'x`.

```
> (quote a)
a
> 'a
a
> (quote (a b c))
(a b c)
```

Nota traducción: la mayoría de las veces traduciremos *quoted*, como *escapar*

2. (atom x)

`(atom x)` devuelve el átomo `t` si el valor de `x` es un átomo o la lista vacía. En Lisp se acostumbra a usar el átomo `t` para representar verdadero, y la lista vacia para representar falso.

```
> (atom 'a)
t
> (atom '(a b c))
()
> (atom '())
t
```

Ahora que se tiene un operador cuyo argumento es evaluado se puede mostrar para que sirve `quote`. Escapando una lista con `quote` evitamos su evaluación. Una lista no escapada dada como argumento a un operador como `atom` es tratada como código:

```
> (atom (atom 'a))
t
```

en cualquier lugar en dónde aparezca una lista escapada se trata como una mera lista, en este caso una lista de dos elementos:

```
> (atom '(atom 'a))
()
```

Esto es similar a la manera en que se usan las comillas en inglés. Cambridge es una ciudad en Massachusetts que tiene cerca de 90.000 personas. "Cambridge" es una palabra que contiene nueve letras.

Escapar puede parecer un concepto algo lejano, porque muy pocos lenguajes tienen algo como esto. Está muy relacionado a una de las más distintivas características de Lisp: código y datos están hechos de las mismas estructuras de datos, y el operador `quote` es la manera en que se distingue entre ellos.

3. (eq x y)

`(eq x y)` devuelve `t` si los valores de `x` e `y` son el mismo átomo, o los dos la lista vacía, y `()` en cualquier otro caso.

```
> (eq 'a 'a)
t
> (eq 'a 'b)
()
> (eq '() '())
t
```

4. (car x)

`(car x)` espera que el valor de `x` sea una lista, y devuelve su primer elemento.

```
> (car '(a b c))
a
```

5. (cdr x)

`(cdr x)` espera que el valor de `x` sea una lista, y devuelve todo lo posterior al primer elemento.

```
> (cdr '(a b c))
(b c)
```

6. (cons x y)

`(cons x y)` espera que el valor de `y` sea una lista, y devuelve el valor de `x` seguido por los elementos del valor de `y`.

```
> (cons 'a '(b c))
(a b c)
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> (car (cons 'a '(b c)))
a
> (cdr (cons 'a '(b c)))
(b c)
```

7. (cond (p_1 e_1)...(p_n e_n))

`(cond (`p_1 e_1`)...(`p_n e_n`))` es evaluado de la siguiente manera. La expresiones `p` son evaluadas en orden hasta que una devuelva `t`. Cuando una es encontrada, el valor de la expresión `e` correspondiente es devuelto como el valor de toda la expresión `cond`.

```
> (cond ((eq 'a 'b) 'first)
            ((atom 'a) 'second))
second
```

En cinco de los siete operadores primitivos, los argumentos son siempre evaluados cuando una expresión que comienza con ese operador es evaluada.[2] Se llama a un operador de ese tipo una `función`.

2. Denotar funciones

A continuación se define una notación para describir funciones. Una función se expresa como `(lambda (`p_1`...`p_n`) e)`, dónde p_1`...`p_n son átomos (llamados `parámetros`) y `e` es una expresión. Una expresión cuyo primer elemento es una expresión

`((lambda (`p_1`...`p_n`) e) `a_1`...`a_n`)`

es conocida como una `llamada de función` y su valor es cálculado de la siguiente manera. Cada expresión a_i es evaluada. Luego `e` es evaluada. Durante la evaluación de `e`, el valor de cualquier ocurrencia de uno de los p_i , es el valor del correspondiente a_i en la llamada de función más reciente.

```
> ((lambda (x) (cons x '(b))) 'a)
(a b)
> ((lambda (x y) (cons x (cdr y)))
      'z
      '(a b c))
(z b c)
```

Si una expresión tiene como primer elemento un átomo `f` que no es uno de los

operadores primitivos

```
(f a1...an)
```

y el valor de `f` es una función `(lambda (p1...pn) e)` luego el valor de la expresión es el valor de

```
((lambda (p1...pn) e) a1...an)
```

En otras palabras, los parámetros pueden ser usados tanto como operadores o como argumentos en expresiones:

```
> ((lambda (f) (f '(b c)))
      '(lambda (x) (cons 'a x)))
(a b c)
```

Hay otra notación para funciones que permite a la función referirse a sí misma, así se brinda una manera conveniente de definir funciones recursivas.[3] La notación

```
(label f (lambda (p1...pn e))
```

denota una función que se comporta como `(lambda (p1...pn) e)`, con la propiedad adicional de que una ocurrencia de `f` en `e` evalua a la expresión `label`, como si `f` fuese un parametro de la función.

Suponiendo que se quisiera definir una función `(subst x y z)`, la cual toma una expresión `x`, un átomo `y`, y una lista `z`, y devuelve una lista como `z` pero con cada instancia de `y` (a cualquier profundidad de anidamiento) en `z` reemplazada por `x`.

```
> (subst 'm 'b '(a b (a b c) d))
(a m (a m c) d)
```

Se puede denotar esta función como

```
(label subst (lambda (x y z)
               (cond ((atom z)
                      (cond ((eq z y) x)
                            ('t z)))
                     ('t (cons (subst x y (car z))
                               (subst x y (cdr z)))))))
```

Se abreviará `f = (label f (lambda (p1...pn) e))` como

```
(defun f (p1...pn) e)
```

así

```
(defun subst (x y z)
```

```
(cond ((atom z)
       (cond ((eq z y) x)
             ('t z)))
      ('t (cons (subst x y (car z))
                (subst x y (cdr z))))))
```

A proposito, aquí se ve como conseguir una cláusula por omisión en una expresión `cond`. Una cláusula cuyo primer elemento es `'t` será siempre éxitosa. Así

```
(cond (x y) ('t z))
```

es equivalente a lo que tal vez se escribiría en un lenguaje con síntaxis como:

```
if x then y else z
```

3. Algunas funciones

Ahora que se tiene una manera de expresar funciones, se definen algunas nuevas en términos de los siete operadores primitivos. Primero es conveniente introducir algunas abreviaciones de patrones comunes. Se usará el `cxr`, dónde `x` es una secuencia de *a*s o *d*s, como una abreviación para la composición correspondiente de `car` y `cdr`. Así, por ejemplo `(cadr e)` es una abreviación de `(car (cdr e))`, la cual devuelve el segundo elemento de `e`.

```
> (cadr '((a b) (c d) e))
(c d)
> (caddr '((a b) (c d) e))
e
> (cdar '((a b) (c d) e))
(b)
```

También, se usará `(list` $e_1 ... e_n$ para `(cons` e_1 `...` `(cons` e_n `'()) ... )`.

```
> (cons 'a (cons 'b (cons 'c '())))
(a b c)
> (list 'a 'b 'c)
(a b c)
```

Ahora se definen algunas funciones nuevas. Se cambiaron los nombres de estas funciones agregando un punto al final. Esto distingue a las funciones primitivas de las definidas en términos de estas, y además evita colisiones con funciones existentes en Common Lisp.

1. (null. x)

`(null. x)` prueba si su arguemento es la lista vacia.

```
(defun null. (x)
  (eq x '()))
> (null. 'a)
()
```

```
> (null. '())
t
```

2. (and. x y)

`(and. x y)` devuelve `t` si ambos argumentos lo hacen, y `()` en cualquier otro caso.

```
(defun and. (x y)
  (cond (x (cond (y 't) ('t '())))
        ('t '())))
> (and. (atom 'a) (eq 'a 'a))
t
> (and. (atom 'a) (eq 'a 'b))
()
```

3. (not. x)

`(not. x)` devuelve `t` si su argumento devuelve `()`, y `()` si su argumento devuelve `t`.

```
(defun not. (x)
  (cond (x '())
        ('t 't)))
> (not. (eq 'a 'a))
()
> (not. (eq 'a 'b))
t
```

4. (append. x y)

`(append. x y)` toma dos listas y devuelve la concatenación de estas.

```
(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons (car x) (append. (cdr x) y)))))
> (append. '(a b) '(c d))
(a b c d)
> (append. '() '(c d))
(c d)
```

5. (pair. x y)

`(pair. x y)` toma dos listas de igual longitud y devuelve una lista de listas de dos elementos conteniendo pares sucesivos de un elemento de cada una.

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list (car x) (car y))
               (pair. (cdr x) (cdr y))))))
> (pair. '(x y z) '(a b c))
((x a) (y b) (z c))
```

6. (assoc. x y)

`(assoc. x y)` toma un átomo `x` y una lista `y` de la forma creada por `pair.`, y devuelve el segundo elemento de la primera lista en `y` cuyo primer elemento es `x`.

```
(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))
> (assoc. 'x '((x a) (y b)))
a
> (assoc. 'x '((x new) (x a) (y b)))
new
```

4. La sorpresa

Así, se pudo definir funciones que concatenan listas, sustituyen una expresión por otra, etc. Una notación elegante, tal vez, pero. ¿Ahora qué? Ahora viene la sorpresa. Se puede también, es resultado de, escribir una función que actue como un interprete para nuestro lengauje: una función que tome como un argumento cualquier expresión Lisp, y devuelva su valor. Aquí está:

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom (eval. (cadr  e) a)))
       ((eq (car e) 'eq)    (eq   (eval. (cadr  e) a)
      (eval. (cadr  e) a)))
       ((eq (car e) 'car)   (car  (eval. (cadr  e) a)))
       ((eq (car e) 'cdr)   (cdr  (eval. (cadr  e) a)))
       ((eq (car e) 'cons)  (cons (eval. (cadr  e) a)
      (eval. (caddr e) a)))
       ((eq (car e) 'cond) (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
   (cdr e))
    a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
     (cons (list (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
     (append. (pair. (cadar e) (evlis. (cdr e) a))
       a)))))

(defun evcon. (c a)
  (cond ((eval. (caar c) a)
  (eval. (cadar c) a))
 ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
 ('t (cons (eval. (car m) a)
    (evlis. (cdr m) a)))))
```

La definición de `eval.` es más larga que cualquiera de las otras vistas anteriormente. Vamos a considerar como trabaja cada parte.

La función toma dos argumentos: `e`, la expresión a ser evaluada, y `a`, una lista que representa los valores que los átomos fueron tomanado al aparecer como parámetros en las llamadas de función. Esta lista es llamada `entorno`, y es de la forma creada por `pair.`. ~~Su fin es construir y buscar en las listas que se escribieron para `pair.` y `assoc.`~~It was in order to build and search these lists that we wrote `pair.` and `assoc`.

La columna de `eval.` es una expresión `cond` con cuatro cláusulas. Como se evalua una expresión depende del tipo que sea. La primer cláusula evalua átomos. Si `e` es un átomo, se busca su valor en el entorno:

```
> (eval. 'x '((x a) (y b)))
a
```

La segunda cláusula de `eval.` es otra `cond` para el manejo de expresiones de la forma `(a ...)`, dónde `a` es un átomo. Esto incluye todos los usos de los operadores primitivos, y existe una cláusula para cada uno.

```
> (eval. '(eq 'a 'a) '())
t
> (eval. '(cons x '(b c))
           '((x a) (y b)))
(a b c)
```

Todos ellos (excepto `quote`) llaman a `eval.` para encontrar el valor de sus argumentos.

Las dos últimas cláusulas son más complicadas. Para evaluar una expresión `cond` llamamos a una función auxiliar llamada `evcond.`, la cual trabaja a su manera a través de las cláusulas recursivamente, buscando una en la cual el primer elemento devuelva `t`. Cuando encuentra dicha cláusula devuelve el valor del segundo elemento.

```
> (eval. '(cond ((atom x) 'atom)
                  ('t 'list))
           '((x '(a b))))
list
```

La parte final de la segunda cláusula de `eval.`. Maneja llamadas a funciones que han sido pasadas como parametros. Trabaja reemplazando el átomo con su valor (que debería ser una expresión `lambda` o `label`) y evaluando la expresión resultante. Así

```
(eval. '(f '(b c))
       '((f (lambda (x) (cons 'a x)))))
```

se convierte en

```
(eval. '((lambda (x) (cons 'a x)) '(b c))
       '((f (lambda (x) (cons 'a x)))))
```

que devuelve `(a b c)`.

Las dos últimas cláusulas en `eval.` manejan llamadas a funciones en las cuales el primer argumento es una expresión `lambda` o `label`. Una expresión `label` es evaluada empujando la lista del nombre de la función y la función en si misma dentro del entorno, y luego llamando a `eval.` en una expresión con el `lambda` interior sustituido por la expresión `label`. Esto es,

```
(eval. '((label firstatom (lambda (x)
                            (cond ((atom x) x)
                                  ('t (firstatom (car x))))))
         y)
       '((y ((a b) (c d)))))
```

que se convierte en

```
(eval. '((lambda (x)
     (cond ((atom x) x)
           ('t (firstatom (car x)))))
   y)
 '((firstatom
    (label firstatom (lambda (x)
                       (cond ((atom x) x)
                             ('t (firstatom (car x)))))))
   (y ((a b) (c d)))))
```

la cual termina por devolver `a`.

Finalmente una expresión de la forma `((lambda (`$p_1 ... p_n$`) e) `$a_1 ... a_n$`)` es evaluada llamando primero a `evlist.` para obtener una lista de valores `(`$v_1 \ldots v_n$`)` de los argumentos $a_1 \ldots a_n$, y luego evaluando `e` con `(`p_1 v_1`) ... (`p_n v_n`)` agregado al frente del entorno. Así

```
(eval. '((lambda (x y) (cons x (cdr y)))
         'a
         '(b c d))
       '())
```

se convierte en

```
(eval. '(cons x (cdr y))
       '((x a) (y (b c d))))
```

que termina devolviendo `(a c d)`.

5. Consecuencias

Ahora que se entiende como `eval` trabaja, volvamos un paso atras y consideremos lo que significa. Lo que se tiene aquí es un

extraordinariamente elegante modelo de computación. Usando sólo `quote`, `atom`, `eq`, `car`, `cdr`, `cons`, y `cons`, se pudo definir una función, `eval`., que en realidad implementa nuestro propio lenguaje, y luego usándo se puede definir cualquier función adicional que se quiera.

Por supuesto que ya existen modelos de computación-el más notable es la máquina de Turing. Pero los programas de la máquina de Turing no son demasiado ~~fáciles~~ edifying de leer. Si se quiere un lenguaje para describir algoritmos, se debe desear algo más abstracto, y ese fue uno de los objetivos de McCarthy al definir Lisp.

Al lenguaje que él definió en 1960 le faltaba un monton. No tenía ~~efectos laterales~~ side-effects, ejecución secuencial (la cual sólo es útil con efectos laterales), no tenía números prácticos[4] y alcance dinámico. Pero estas limitaciones pueden ser remediadas, sorprendentemente con muy código adicional. Steele y Sussman muestran como hacerlo en un famoso paper llamado "The Art of the Interpreter"[5].

Si se entiende la función `eval` de McCarthy, se entiende más que sólo un estadío en la historia de los lenguajes. Estas ideas están aún presentes en el núcleo semántico de Lisp en la actualidad. Así que estudiando el paper original de McCarthy nos muestra, en un sentido, lo que Lisp realmente es. No es tanto algo que McCarthy diseño sino algo que descubrió. No es intrínsicamente un lenguaje para AI o para realizar prototipos rápidos, o cualquier otra tarea de ese nivel. Es lo que obtienes (o una de las cosas que obtienes) cuando intentas ~~axiomatizar la computación~~ axiomatize computation.

En el tiempo, el lenguaje medio, queriendo decir el lenguaje utilizado por el programador medio, ha ido acercándose considerablemente a Lisp. Así que entiendiendo `eval` estarás entendiendo lo que probablemente sea el principal modelo computacional en el futuro.

---

1 Recursive functions of symbolic expressions and their computation by machine (Part I) Communications of the ACM in April 1960. (El paper original) Volver

2 Las expresiones que comienzan con los otros dos operadores, `quote` and `cond`, son evaluadas de manera diferente. Cuando una expresión es escapada su argumento no es evaluado, es simplemente devuelto como el valor de toda la expresión escapada. Y en una expresión `cond` válida, sólo un camino con forma de L de subexpresiones será evaluado. Volver

3 Lógicamente no se necesita definir una nueva notación para esto. Se puede definir funciones recursivamente con la notación que se tiene hasta el momento usando una función en funciones llamada Y combinator. Puede que McCarthy no supiese acerca del Y combinator cuando escribió su paper, en cualquier caso la notación con etiqueta (labe) es más leíble. Volver

4 Es posible realizar aritmética en el Lisp de McCarthy de 1960 utilizando una lista de `n` átomos para representar el número `n`. Volver

5 Guy Lewis Steele, Jr. y Gerald Jay Sussman, "The Art of the interpreter, or the Modularity Complex (Part Zero, One, and Two)," MIT AI Lab Memo 453, Mayo de 1978. Volver

Publicado por santi en 10:21 p. m.

Etiquetas: lenguajes de programación, lisp

No hay comentarios.:

Publicar un comentario

Entrada más reciente Página Principal Entrada antigua

Suscribirse a: Comentarios de la entrada (Atom)