Lenguajes de Programación I

Clausuras, Alias y Sobrecarga

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Clasificación de los valores

Los lenguajes manipulan valores

- **Valores de Primera Clase**
  - Se pueden asignar a variables.
  - Se pueden pasar como argumentos.
  - Se pueden retornar como resultados.
  - Típico de tipos primitivos: números, caracteres, ...
- **Valores de Segunda Clase**
  - Sólo se puede pasar como parámetro.
  - Subrutinas en algunos lenguajes – ¿has usado `qsort()` en C?
- **Valores de Tercera Clase**
  - Ninguna de las tres cosas.
  - Las etiquetas en la mayoría de los lenguajes.

Mejorar la clase de algunos valores aumenta la flexibilidad, pero crean problemas interesantes en cuanto al alcance.

¿Y si el valor es una subrutina?

Gracias Church por los favores recibidos...

- Asociamos nombres a subrutinas para invocarlas – ¿y si usamos el nombre como un **valor** que haga *referencia* a ella?
- En lenguajes como Haskell, LISP/Scheme y Perl esto es posible – las subrutinas son objetos de **primera clase**.
- Es posible crear subrutinas **con o sin** nombre – cuando no tienen nombre se llaman *lambdas*.
- Es posible invocarlas de manera directa o indirecta.
  - Directamente – usando el nombre de la forma clásica.
  - Indirectamente – usando la referencia y algún operador especial.

Funciones que llaman funciones, crean funciones, reciben funciones y retornan funciones.

Funciones que usan funciones como argumentos

Funciones de *orden superior*

- Funciones que usan funciones como argumentos
  - Haskell

    ```
    map (*2) [ 1, 3, 5, 7 ]
    ```
  - Perl

    ```
    @pares = map { $_ * 2 } qw(1 3 5 7)
    ```
  - LISP/Scheme

    ```
    (mapcar (lambda (x) (* 2 x)) (1 3 5 7))
    ```
- `map` y `mapcar` reciben una **función** como primer argumento y una lista como segundo argumento, para producir una lista.
- Funciones anónimas en el ejmplo – no tiene por qué ser así.

...y más cosas cuando lleguemos a Programación Funcional

¿Qué tiene que ver con el alcance?

- Supongamos un lenguaje:
  - Con Alcance de bloques anidados.
  - Con Funciones de Primera (o Segunda) Clase.
- Entonces es posible:
  - Crear una función en un alcance anidado – ¡incluso anónima!
  - Retornar una *referencia* a esa función –
    o pasarla como argumento a una función.

¡La referencia a la función *sobrevive*
al **alcance donde fue definida**!

Yo dawg, we heard you liked functions. . .

```
sub crea_contador {
  my $base = shift;  # $base es el primer argumento recibido
  return sub {
    return $base++;
  }
}
# Programa principal
my $desde42 = crea_contador(42);
my $desde69 = crea_contador(69);
```

- ¿Qué va a pasar con $base?
- ¿Qué va a pasar cuando llamemos a las rutinas?

Dafuq?

Cuando los objetos locales sobreviven

- En general la extensión de la vida de un objeto local es
  - Limitada (como en los lenguajes imperativos habituales) –
    si sale de alcance, la pila se desocupa y desaparece.
  - Ilimitada (como en los lenguajes funcionales) –
    hasta que el recolector de basura las recupere.
- En nuestro ejemplo, el objeto local ($base) **debe** sobrevivir pues forma parte del ambiente de referencia de la función retornada.
- En lenguajes funcionales **todo** está en el *heap* –
  no es un problema hacer perdurar un objeto local en el tiempo.
- En lenguajes imperativos que quieren implantar funciones de primera clase, es necesario *promover selectivamente* objetos con vida local para que pasen al *heap*.

Construcción del Ambiente de Referencia Apropiado

First Class Compiler Problems

- La manera en que se construye el ambiente de referencia para funciones de primera (o segunda) clase es una decisión de diseño.
  - ¿Cómo saber cuáles valores deben preservarse? –
    a veces no hace falta "arrastrar" ninguno.
  - ¿Cuándo hacer ese cálculo?
- Saber cuáles valores deben preservarse se reduce a aplicar las reglas de Alcance Estático o Dinámico que ya conocemos.
- El cálculo preciso puede hacerse temprano o tarde –
  ¿qué quiere decir esto en términos de un valor función?
  - Temprano (estático) – cuando se *crea* la función.
  - Tarde (dinámico) – cuando se *usa* la función.

Asociación Profunda (*Deep Binding*)

- Cuando el Ambiente de Referencia se calcula en el momento en que se construye la función.
  - Primera clase – en el entorno de la definición anidada.
  - Segunda clase – en el entorno donde es pasada como parámetro.
- Se hace un *paquete* con todas las variables alcanzables en ese momento que deban sobrevivir al entorno.
- Se asocia ese paquete con el cuerpo de la función – se denomina **Clausura** (*Closure*).
- A tiempo de ejecución, cuando se invoca la función directa o indirectamente, se usa el paquete específico.
- En lenguajes compilados eso implica generar código extra para construir el paquete, promoverlo al *heap* y asociarlo a la función.

Asociación Superficial (*Shallow Binding*)

- Cuando el Ambiente de Referencia se calcula en el momento en que se invoca la función.
  - Primera clase – donde se *use* la referencia a la función para invocarla.
  - Segunda clase – donde se *use* el argumento para invocar la función que fue pasada como parámetro.
- Simplemente se determinan las variables alcanzables en relación al punto de invocación.

Cuatro posibilidades

. . . pero sólo una es práctica

- Asociación Profunda y Alcance Estático – el útil y frecuente.
  - Observar el punto de creación de función o pasaje de parámetro.
  - Conservar asociaciones locales según la cadena estática.
  - Complicada de implantar, semántica clara y sólida.
- Asociación Profunda y Alcance Dinámico.
  - Observar el punto de creación de función o pasaje de parámetro.
  - Conservar asociaciones locales según la cadena dinámica.
  - Fácil de implantar, semántica confusa, poco útil.
- Asociación Superficial y Alcance Estático – Emacs LISP.
  - Observar el punto de llamada a la función.
  - Usar las asociaciones locales según la cadena estática.
  - Fácil de implantar, semántica "razonable".
- Asociación Superficial y Alcance Dinámico – original de LISP.
  - Observar el punto de llamada a la función.
  - Usar las asociaciones locales según la cadena dinámica.
  - Trivial de implantar.

Alias

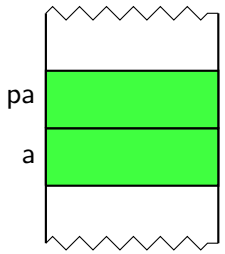
- Más de un nombre para un elemento en un alcance particular.
- Habituales en lenguajes que ofrecen apuntadores o referencias – nombre directo a un valor alcanzable con una o más indirecciones.
- Algunos lenguajes permiten múltiples nombres directos explícitamente

```
$foo = 42; *bar = *foo;
print $bar;
```

  (en este caso, alias entre variables, pero aplica para arreglos, diccionarios, *filehandles* y **funciones**).
- Hacen difícil comprender algunos programas.
- Generar código óptimo en presencia de aliases no es decidible – tópico de estudio en Lenguajes de Programación III.

Alias en C

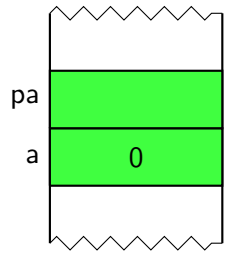
```
void foo()
{
  int a, *pa; /* <- */
  a  = 0;
  pa  = &a;
  *pa = 42;
  printf("%d %d\n",a,*pa);
}
```



Elaboración de a y *pa

Alias en C

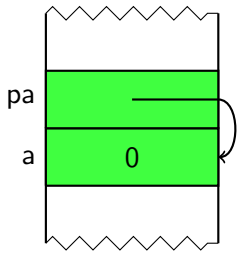
```
void foo()
{
  int a, *pa;
  a  = 0;  /* <- */
  pa  = &a;
  *pa = 42;
  printf("%d %d\n",a,*pa);
}
```



Accedemos via a

Alias en C

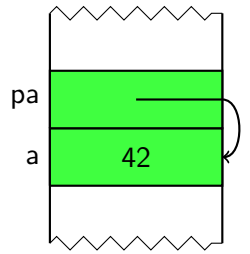
```
void foo()
{
  int a, *pa;
  a  = 0;
  pa  = &a; /* <- */
  *pa = 42;
  printf("%d %d\n",a,*pa);
}
```



*pa apunta a a – son alias.

Alias en C

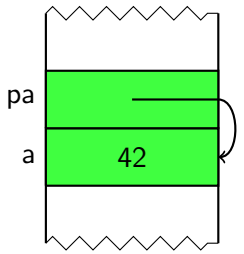
```
void foo()
{
  int a, *pa;
  a  = 0;
  pa  = &a;
  *pa = 42; /* <- */
  printf("%d %d\n",a,*pa);
}
```



Modificamos via *pa.

Alias en C

```
void foo()
{
  int a, *pa;
  a  = 0;
  pa  = &a;
  *pa = 42;
  printf("%d %d\n",a,*pa); /* <- */
}
```




```
42 42
```

Sobrecarga

Ambiguedad uno a muchos

- Nombre único para varios elementos en un alcance.
- Se dice que el nombre está *sobrecargado*.
- Casi cualquier lenguaje tiene algún nombre sobrecargado – el + en C, el `new` en Java.
- Establecida en el lenguaje como consecuencia de su diseño para simplificar la implantación y el aprendizaje.
- En ocasiones disponible al programador explícitamente.
  - Sobrecargar funciones.
  - Sobrecargar operadores.

Sobrecarga de Funciones

- Algunos lenguajes ofrecen mecanismos para sobrecargar subrutinas.
- Mismo nombre, diferente número o tipo de argumentos – la *firma*.

```
foo();
foo(int a);
foo(int a, int b);
foo(float a);
foo(int b);  /* Produce un error al compilar */
```

- Compilador (o interpretador) analiza *firma* para determinar cuál es la función particular que debe emplearse en cada caso.

```
foo(42);
foo(42.0);
foo("42",42); /* Produce un error al compilar */
```

Sobrecarga de Operadores

- Algunos lenguajes (C++, Perl, Fortran90) permiten sobrecargar operadores aritméticos con funciones definidas por el usuario.
- Típicamente se asocia a un método de una clase para agregar esa carga semántica al mismo símbolo.
  - Definir una clase `Complex` para números complejos.
  - Definir el método `Complex add_complex(Complex)` para sumar un número complejo a una instancia.
  - Asociar dicho método al símbolo + del lenguaje para poder escribir cosas como

```
a Complex;
b Complex;
c Complex;
c = a + b   // c = a.add_complex(b)
```

Cuando otras cosas parecen sobrecarga

- Algunos lenguajes (Haskell) permiten definir funciones en las cuales los *tipos* a recibir son arbitrarios siempre que su uso sea *consistente*.
- Esto es útil para definir funciones *genéricas* que pueden operar con argumentos que cumplan con las restricciones
- Ya vimos la función `map`.
  - Su primer argumento es una función.
  - Su segundo argumento es una lista.
  - `map` opera sin problemas siempre y cuando la función pasada como argumento, pueda operar sobre objetos como los que contiene la lista.
- Esto no es sobrecarga, sino polimorfismo paramétrico (que estudiaremos después). Sólo existe una función que opera sobre diferentes tipos.

Bibliografía

- [*Scott*]
  - Secciones 3.5 y 3.6.
  - Ejercicios y Exploraciones