Una extensión de Lisp para la programación orientada a objetos.

Gustavo ARANGO
Grupo de Investigación ALIS
Departamento de Sistemas
Universidad de los Andes
Apartado Aereo 4496
Bogotá - Colombia

Resumen:

Se va a describir una extensión minima en Lisp para permitir la programación orientada a objetos. Estos últimos se representan como funciones y pueden ser utilizados de manera muy cómoda y flexible por los programadores. Se conservan las nociones de clases, operaciones y objetos y se permite la generación de transformaciones de objetos mediante el envio de mensajes. Las clases por su lado se pueden estructurar jerárquicamente y se maneja la noción de herencia de operaciones y mensajes.

1. MOTIVACIONES.

Al modelar situaciones de la vida real uno de los enfoques que se utiliza es basar la estructura de los programas en la estructura del sistema que se modela
se hace de esta manera la transformación:
objeto del sistema ---> objeto computacional

Para extender el modelo no se requieren cambios estrategicos en el programa solo se necesitan adiciones de análogos simbólicos de objetos o de acciones del sistema que se extrae del mundo real de forma natural

Una manera de hacerlo es mediante la programación orientada objeto (ver [GOL 83], [CHA 80]) que tiene como idea esencial la de llevar el estado de cada objeto (saber cómo y cuándo cambia manteniendo en todo momento su identidad)

El estado de un objeto se puede caracterizar mediante el uso de variables de estado que mantienen la información que permite obtener la historia del objeto o explicar su comportamiento.

En un sistema complejo los objetos son rara vez completamente independientes. Cada uno puede influenciar el estado de otros a través de interacciones que sirven para acoplar los cambios en las variables de estado de diferentes objetos.

Un enfoque interesante y fácil de manejar es agrupar las variables de estado en sub-sistemas cuando éstas están relacionadas de alguna manera. Para que el enfoque sirva y sea modular se deben descomponer en objetos que correspondan a modelos de los objetos reales del sistema.

2. UN EJEMPLO PARA ILUSTRAR LO QUE SE DESEA.

Supongamos que queremos representar la siguiente información como un tipo abstracto de datos:

- Se tiene una cuenta corriente que se abre originalmente con una cierta cantidad (esta cantidad representa la variable de estado que nos determina una cuenta particular; la llamaremos saldo).
- Sobre una cuenta se pueden efectuar dos operaciones: retiros (siempre y cuando no haya sobregiro) y depósitos. Cada una de estas operaciones modifica la variable saldo.

Se desea definir la clase de todas las cuentas y sus operaciones; la manera de generar instancias de dichas clase - ie: cuentas particulares - y una forma simple de manejar estas instancias que por medio de un mensaje identifique la operación a efectuar y produzca el efecto deseado.

Por ejemplo si cuenta_banco y caja_menor son dos instancias de la clase cuenta se desea poder afectarlas de la forma siguiente:

```
(cuenta_bancos deposito 10000)
(caja_menor retiro 10000)
```

Se espera que el resultado de dichas transacciones afecte el saldo de las cuentas respectivas y que depósito y retiro sean mensajes que desencadenen la apropiada operación de retiro o de depósito sobre la cuenta mencionada; mejor aún se desea que las cosas se hagan de una forma suficientemente modular para que una nueva operación y por consiguiente un nuevo mensaje sean totalmente independientes de las instancias y de las clases.

3. LO QUE SE NECESITA.

Para lograr lo anterior se va a trabajar con las siguientes funciones Lisp:

- (clase ...) permite introducir una clase

- (operacion ...) permite introducir una operación para una clase sin restricciones se pueden crear operaciones de un mismo nombre para diferentes clases

- (mensaje ...) permite introducir una lista de relaciones entre mensajes y operaciones entre clases
  sin restricciones se pueden crear varios mensajes para una misma operación

- (objeto ...) permite crear una instancia de una clase

-(subclase ...) permite generar una versión más sofisticada de alguna clase que hereda las operaciones de su superclase

4. LA IMPLEMENTACION EN LISP.

4.1. La creación de clases.

Se usa la forma general:

```
(clase NombreClase Variables_de_Estado)
```

para nuestro ejemplo de cuenta se tiene:
(clase cuenta saldo)

si la cuenta tuviera una clave de acceso se tendría:
(clase cuenta_clave saldo clave)
nótese que las variables de estado aparecen en sucesión

La definición de esta forma es:
```
(df clase *l:
  (prog ( )
    (put (car *l:) (cdr *l:) 'estado)))
```

Se utilisan las formas *df de dm* de algunas de las versiones de Lisp (ver [CHA 80])

Lo que se está haciendo es generar una lista de propiedades para el átomo identificado por NombreClase que asigna a su propiedad **estado** la lista de las variables de estado de la clase.

Para la definición de cuenta tenemos que la lista de propiedades del atomo **cuenta** es:
estado: (saldo)

4.2. La implementación de operaciones.

Para la definición de operaciones lo que se desea lograr se ilustra con un ejemplo.

La definición de un depósito es como sigue:
(setq saldo (+ saldo monto))
donde monto corresponde a la cantidad depositada

La forma de definición de la operación es:
(operacion deposito cuenta (monto) (setq saldo (+ saldo monto)))

y corresponde a la forma general:
(operacion NombreOperacion NombreClase Parametros Cuerpo)

Lo que se desea es generar la siguiente función que es utilizable por toda

instancia de la clase cuenta sin modificación ninguna:

```
(defun deposito:cuenta
  (lambda (::l monto)
     (prog (saldo)
        (setq saldo (nombre_excl ::l 'saldo))
        (return (set saldo (+ (eval saldo) monto))))))
```

Nótese el uso de la forma defun para definir las funciones en algunas versiones de Lisp - en particular para IQ-Lisp [IQ 83]

se puede ver que:

- se agrega a la función un nuevo parámetro ::l que en el momento del uso de una instancia cualquiera de la clase va a indicar como obtener las variables de estado de la instancia;

- se modifica el nombre de la función agregandole el identificador de la clase para poder así manejar varias operaciones con el mismo nombre para diferentes clases;

- el cuerpo de la función se modifica para permitir, por un doble direccionamiento, la obtención correcta de las variables de estado.

El texto que para cada operación define una función como la presentada anteriormente es:

```
(df operacion (*n* *t* *p* *c*)
  (prog (y)
     (setq y (get *t* 'estado))
     (eval (transforme_text (nombre_excl *n* *t*) y *p* (rep_l_atom y *c*))) ))
```

la función (nombre_excl Nombre_instancia Variable) genera un nombre particular para la variable que depende de la instancia; una posibilidad de implementación en Iq-Lisp es:

```
(de nombre_excl (n var)
   (intern (at (concat n (concat ":" var)))))
```

intern, concat, at son funciones de base de IQ-Lisp (ver [IQ 83])

Para ser eficientes con el manejo del espacio, la idea es generar un cuerpo único que pueda ser compartido por cada objeto creado. Se adiciona una serie de variables locales que se identifican con las variables de estado de cada objeto y se efectua un remplazo sobre el texto dado por el usuario para definir la operación.

El remplazo textual hace las siguientes tranformaciones sobre el cuerpo

- (**setq** Variable_global ...) <~ (**set** Variable_global ...)
- (**quote** Variable_global) <~ Variable_global
- Variable_global <~ (**eval** Variable_global)

estos remplazos permiten el manejo de las variables propias del objeto creado mediante un doble direccionamiento

Para la transformación del código de definición de una operación se utilisa la función:

(rep_l_atom Nombre_Objeto Globales Cuerpo)

que remplaza en Cuerpo toda ocurrencia de una variable que pertenezca a la lista Globales por un identificador exclusivo del objeto creado y al mismo tiempo ejecuta sobre el texto las transformaciones descritas anteriormente.

El texto de definición es:

```
(de rep_l_atom (lg text)
 (cond ((null text) nil)
       ((atom (car text))
          (if (memb (cadr text) lg)
             (cond ((eq (car text) 'setq)
                    (append (list 'set (cadr text))
                            (rep_l_atom lg (cddr text))))
                   ((eq (car text) 'quote)
                    (cadr text))
                   ((memb (car text) lg)
                    (cons (list 'eval (car text))
                          (rep_l_atom lg (cdr text))))
                   (t (cons (car text)
                            (rep_l_atom lg (cdr text)))))
             (cons (if (memb (car text) lg)
                       (list 'eval (car text))
                       (car text))
                   (rep_l_atom lg (cdr text))))
       (t
         (cons (rep_l_atom lg (car text))
               (rep_l_atom lg (cdr text))))))
```

Nótese que la función if no figura en todas las versiones de Lisp. Una manera de escribirla es:

```
(df if (l) |" (cond (@ (cadr l) @ (caddr l)) (t @(car (cdddr l)))
```

Nótese que se usan los símbolos de definición de macros. |" @ |@ que se encuentran en varias versiones Lisp [CHA 80] y [IQ 83]

La función **operacion** utiliza la función auxiliar transforme_text que se encarga del trabajo de construir el texto de la función genérica; su código es:

```
(de transforme_text (*n glob *p *c)
  |"(defun @ *n
     (lambda @(cons '::l *p)
       (prog @glob
             |@(mapcar
                 (lambda (x) |"(setq @x (nombre_excl ::l (quote @x))))
                 glob)
              (return @ *c)))))
```

4.3. La implementación de mensajes.

La definición de **mensaje** es bastante sencilla; su forma corresponde a.
(mensaje NombreClase ListaMensajeOperacion)

para la definición de los mensajes de la clase cuenta se puede hacer:

```
(mensaje cuenta (retiro . retiro) (deposito . deposito))
```

o

```
(mensaje cuenta (retiro . retiro))
(mensaje cuenta (deposito . deposito))
```

si se desea definir entrada como otro posible mensaje sinonimo de depósito basta agregar:

```
(mensaje cuenta (entrada . deposito))
```

La función que maneja las correspondencias entre mensajes y operaciones es:

```
(df mensaje :l:
  (prog (clase lista)
        (setq clase (car :l:)
        (setq lista (cdr :l:)
   ciclo (put 'mensaje (nombre_excl (cdar lista) clase)
                       (nombre_excl (caar lista) clase))
         (if (null (setq lista (cdr lista))) (return))
         (go ciclo)))
```

notese que los mensajes se guardan modificados con el nombre de su clase para de esta manera permitir que se puedan usar mensajes idénticos para operaciones distintas.

4.4. La implementación de instancias u objetos.

La impementación de una instancia es también algo complejo de realizar pero por el contrario es muy simple en su forma de uso.

Definir que cuenta_banco es una instancia que en Colombia requiere un saldo mínimo de 100000 pesos se hace con:
(objeto cuenta_banco cuenta 100000)

Se desea que la definición anterior produzca la siguiente función:

```
(defun cuenta_banco
  (nlambda *l:
   (prog (x)
    (setq x
       (ver_mensaje (car *l:) 'cuenta))
    (if
       (null x)
       (return "operacion desconocida")
       (return (apply x
                    (cons 'cuenta_banco
                          (cdr  *l:))))))))
```

Nótese el uso de la forma *nlambda* (ver [IQ 83]) para definir una función que en algunas versiones de Lisp es equivalente a *df* o a *fexpr* (ver [CHA 80])

la función ver_mensaje identifica el mensaje, obtiene la operación a ejecutar o produce un mensaje de error; en el caso de una identificación correcta, la operación se aplica sobre una lista de parámetros (esta lista incluye, además de los requeridos por la operación, el identificador del objeto para poder asociar correctamente las variables de estado).

Dado el alcance de Lisp, es necesario producir al mismo tiempo la inicialización de las variables de estado del objeto.

El texto que produce lo esbozado anteriormente es:

```
(dm objeto (l)
  |" (defun @(cadr l)
     (nlambda *l:
       (prog (x)
        @(bloque (lista_set (cadr l) (get (caddr l) 'estado) (cddar l))
                 |"(setq x (ver_mensaje (car *l:) (quote @(caddr l)))))
        (if (null x) (return "operacion desconocida"))
         (return (apply x (cons (quote @(cadr l)) (cdr *l:))))))))
```

la función lista_set se encarga de inicializar los valores de las variables globales con los argumentos del llamado; su forma general es:
(lista_set Nombre_Instancia Globales Actuales)

su definición es:

```
(de lista_set (n lg la)
  (if (not (null lg))
    (bloque (set (nombre_excl n (car lg)) (car la))
            (lista_set n (cdr lg) (cdr la)))))
```

nótese que la inicialización de las variables de estado se hace por intermedio de la función lista_set y es por lo tanto necesario hacerlo como un efecto de borde que se logra usando la función **bloque**.

La función **bloque** que permite una evaluación sequencial de una serie de expresiones Lisp se puede escribir usando la definicion de macros df:

```
(df bloque (l) |" (cond (t |@ (cdr l))))
```

El código de la función ver_mensaje se presentará una vez que se explique cómo funcionan las subclases de una clase.

4.5. La implementación de subclases.

Un tipo especial de cuenta es aquella que permite un saldo en rojo. Esta cuenta funciona de la misma manera que una cuenta normal para el caso de un depósito y tiene una forma propia para los retiros. Sus variables de estado son además del saldo, un límite de sobregiro que se le concede al dueño de la cuenta.
Para este caso especial es muy útil definir la clase cuenta_rojo como una subclase de cuenta.

La forma de hacerlo corresponde a:
(subclase cuenta cuenta_rojo rojo)

que corresponde a la forma general:
(subclase NombreSuperclase NombreClase Variables_de_Estado)

Como se aprecia en el ejemplo, las variables de estado incluídas son solamente las adicionales a las de la superclase (por eso se excluye saldo)

La función **subclase** produce el mismo efecto que la función **clase** definiendo como variables de estado las de la clase sumadas a las de la superclase y creando para la clase una nueva propiedad **superclase** en la que se indica la clase que la origina.

El texto de esta función es:

```
(df subclase *l:
  (prog( )
    (put (cadr *l:) (append (get (car *l:) 'estado) (cddr *l:)) 'estado)
    (put (cadr *l:) (car *l:) 'superclase)))
```

nótese que el orden de los argumentos de append es importante si se desea que las operaciones que se heredan de la superclase funcionen correctamente al usar las variables de estado

El ejemplo anterior genera la siguiente lista de propiedades para el atomo **cuenta_rojo**:

estado: (saldo rojo)
superclase: cuenta

Para definir operaciones y mensajes de una subclase se procede de la misma manera que para la clase. A continuación se presenta la definición de un retiro y su mensaje correspondiente para cuenta_roja.

```
(operacion retiro cuenta_roja (monto)
  (if (>= (- saldo monto) (- 0 rojo))
      (bloque (setq saldo (- saldo monto)) saldo)
      "se sobrepasa el saldo rojo autorizado"))

(mensaje cuenta_rojo (retiro . retiro))
```

La función, que se encarga de escoger la operación a efectuar segun el mensaje, recorre la cadena de superclases hasta encontrar la primera operación que corresponda.

Su código es:

```
(de ver_mensaje
   (men clase)
        (if (null clase)
            nil
           (let
              ((op
                  (get 'mensaje (nombre_excl men clase))))
               (if op
                  op
                  (ver_mensaje men
                           (get clase 'superclase))))))
```

5. UN EJEMPLO DEL MODELO ORIENTADO OBJETO.

5.1. Presentación del ejemplo.

Un ejemplo de la utilización de las funciones anteriormente definidas es la definición de la clase cuenta con clave (cuenta_clave).
Estas cuentas tienen dos variables de estado: su saldo y el codigo de acceso a la cuenta. Las operaciones posibles sobre estas cuentas son el depósito, el retiro y el cambio de clave.
Cada una de estas operaciones maneja dos informaciones el monto de la transacción y el código de acceso para permitirla (en el caso de retiro o depósito) y el viejo y nuevo códigos para el cambio de la clave.

5.2. Descripción del ejemplo en términos de objetos y mensajes.

Las siguientes funciones definen completamente el objeto computacional.

```
(clase cuenta_clave saldo clave)


(operación retiro cuenta_clave (monto validacion)
  (cond ((not (eq clave validacion)) "clave de acceso invalida")
        ((>= (monto saldo)) "fondos insuficientes")
        (T (setq saldo (- saldo monto)) saldo)))

(operación deposito cuenta_clave (monto validacion)
  (if (not (eq clave validacion))
      "clave de acceso invalida"
      (bloque (setq saldo (+ saldo monto)) saldo)))

(operación clave cuenta_clave (validacion nueva_clave)
  (if (not (eq clave validacion))
      "clave de acceso invalida"
      (bloque (setq clave nueva_clave) nueva_clave)))


(mensaje cuenta_clave (retiro . retiro) (deposito . deposito)
                      (cambio . clave) (proteccion . clave))
```

Se define además una subclase de la anterior que no acepta modificaciones de la clave de acceso:

```
(subclase cuenta_clave cuenta_clave_fija)
(operacion clave cuenta_clave_fija nil "no se permite el cambio de clave")
(mensaje (cambio . clave) (proteccion . clave))
```

5.3. Ejemplo de utilización.

Al definir las instancias.

```
(objeto cuenta_suiza cuenta_clave 1000000 'pepe)
(objeto cuenta_fija cuenta_clave_fija 1000 'nada)
```

Se produce el "diálogo" siguiente:

```
(cuenta_suiza cambio 'pepe 'pastor)
pastor

(cuenta_suiza retiro 10000 'pepe)
"clave de acceso invalida"

(cuenta_suiza retiro 2000000 'pastor)
"fondos insuficientes"

(cuenta_fija cambio 'nada 'pepe)
"no se permite el cambio de clave"

(cuenta_suiza clave 'pastor 'perro)
"operacion desconocida"

(cuenta_suiza deposito 50000 'pastor)
1050000
```

6. CONCLUSIONES Y PERSPECTIVAS.

Se da el código en Lisp para comenzar a hacer programación orientada a objetos en este lenguaje y esperamos que esta implementación sea utilizada (no se incluye el texto de las macros que sirven para definir las formas de, df y dm porque este depende de las versiones de Lisp utilizadas y es por lo general muy sencillo)

Las expresiones que se usan para definir una clase, sus operaciones y la creación sus instancias es de manejo muy simple. Para efectuar cambios en los objetos creados solo se necesitan conocimientos de la sintaxis de Lisp y del campo de aplicación.

Nuestra implementación favorece la facilidad de uso de los objetos y por esta razón, la protección de los objetos y la verificación de una correcta utilización de los mensajes con los parámetros apropiados no se tienen en cuenta.
Las mejoras a considerar para nuevas versiones - además de lo citado

anteriormente - debe permitir:
- controlar las modificaciones de las operaciones de una clase,
- definir reglas y alternativas de herencia de operaciones y mensajes,
- permitir que una clase pertenezca a varias superclases, ...

Aunque el número de mejoras posibles queda abierto, es importante anotar que el núcleo presentado es suficiente para introducir la programacion orientada objeto y para producir aplicacionas interesantes.

BIBLIOGRAFIA

[ABF 85] ABELSON H., SUSSMAN G. J. y SUSSMAN J. "Structure and Interpretation of Computer Programs". McGraw_Hill. 1985.

[CHA 80] CHARNIAK E., RIESBECK C. y McDERMOTT D. "Artificial Intelligence Programming". Lawrence Erlbaum Associates. 1980

[GOL 83] GOLBERG A., y ROBSON D. "SMALLTALK-80. The language and its implementation". Addison-Wesley Publishing Company, 1983.

[IQ 83] IQ-Lisp Reference Manual. Integral Quality.