# 75.31 Teoría de Lenguaje

Adrián Mouly (92501)

Primer Cuatrimestre de 2012

## 1. Ejercicio Nº1

### 1.1. Enunciado

Considere el siguiente programa en OZ:

```
1  local Res in
2        local Pi in
3              thread
4                    Pi = 3.14
5                    Res = Pi+Y
6              end
7              thread
8                    {Delay 10000}
9                    Y = 6.86
10                   {Browse Res}
11             end
12       end
13 end
```

Ejecutarlo en la máquina abstracta vista en clase. Responda y justifique las siguientes preguntas:

1. ¿La aparición del identificador de variable Pi en la 5º línea está libre o ligada?.
2. ¿Cuando se comience a ejecutar la línea 5 del código, la variable a la que se refiere el identificador Res está ligada a un valor o su valor es indeterminado?.
3. ¿La aparición del identificador de variable Y en la 5º línea está libre o ligada?.
4. Después de 10 segundos y luego que se haya ejecutado completamente el segundo hilo (entre las líneas 7 y 11) ¿Qué es lo que pasa?.

## 1.2. Resolución

Cada estado de la computación se representa como (**ST**, **σ**), donde **σ** es el store de variables y ST es una pila de sentencias semánticas de la forma:

[( **<s>**$_0$ , **E**$_0$ ), ( **<s>**$_1$ , **E**$_1$ ), …], siendo **<s>**$_i$ una sentencia y **E**$_i$ el conjunto que mapea identificadores de esa sentencia con variables del store. La ejecución del programa anterior resulta entonces:

a. **([(local Res in ... end, {Y->y})], {y})**
   La única sentencia apilada es el programa completo. Supongo que la variable Y ya fue ligada anteriormente en el programa principal.

b. **([(local Pi in ... end, {Y->y, Res->res})], {y, res})**
   En este paso se ligan las variables antes definidas y se procede a ejecutar el contenido de este local.

c. **([([thread … end, thread … end], {Y->y, Res->res, Pi->pi}) ], {y, res, pi})**
   Se reconocen las dos sentencias de threads y se procede a ejecutarlos por separado.

d. **({ [(thread … end)] , [(thread Pi = 3.14 Res = Pi+Y end), {Res->res, Pi->pi}] }, {Y->y}, {y, res, pi})**
   Se crea un hilo a parte del principal, con el contenido del primer thread que fue declarado. En el thread principal continúa apilado el segundo thread declarado.

e. **({ [], [(Pi=3.14 Res=Pi+Y), {Res->res, Pi->pi}] , [(thread {Delay 10000} Y=6.86 {Browse Res} end)] }, {Y->y}, {y, res, pi->3.14})**
   Se desapila el último thread, quedando conformados 3 threads, en el cual el principal se encuentra vacío.

f. **({ [(Pi=3.14 Res=Pi+Y), {Res->res, Pi->pi}] , [{Delay 10000} Y=6.86 {Browse Res}] }, {Y->y}, {y->6.86, res, pi->3.14})**
   El sistema mata el primer thread y se continúa con la interpretación del tercero, el cual liga el valor de "Y" luego de una espera de 10000 milisegundos.

g. **({ [(Pi=3.14 Res=3.14+6.86), {Res->res, Pi->pi}] , [{Delay 10000} Y=6.86 {Browse Res}] }, {Y->y}, {y->6.86, res->10, pi->3.14})**
   El primer hilo estaba pausado esperando el valor de Pi, el cual es finalmente asignado, y por ende prosigue su ejecución.

h. **({ [{Delay 10000} Y=6.86 {Browse 10}] }, {Y->y}, {y->6.86, res->10, pi->3.14})**
Al terminar la ejecución del primer hilo y al estar todas sus variables ligadas, termina su ejecución y el sistema lo mata, pasando la ejecución principal al segundo hilo, donde se muestra por pantalla el resultado final.

i. **({ $\phi$ }, { $\phi$ }, {y->6.86, res->10, pi->3.14})**
Al terminar de ejecutarse todo el segundo hilo, este queda vacío esperando que el sistema lo finalice.

**1.** El identificador de Pi, en la 5ta línea se encuentra ligado a una variable interna, y esta misma a su vez se encuentra ligada a un valor “3.14”.

**2.** Cuando se comience a ejecutar esta línea, la variable a la que se refiere el identificador “Res”, tiene un valor indeterminado.

**3.** El identificador de variable “Y” se encuentra libre considerando el scope mencionado, ya que se encuentra declarada en un scope superior.

La ligadura de su valor va a depender de la ejecución del segundo hilo. En caso de haberse ejecutado la línea 9, la variable a la que se refiere “Y” (“y”) va a estar intentar ligarse a un valor, en caso contrario va a estar libre (podría ligarse en otro hilo u otro fragmento de código).

**4.** En el segundo hilo, “y” se liga con el valor “6.86” y la función “Browse” se detiene. Al mismo tiempo, en el otro hilo, se determina el valor de “Res”, finalizando el mismo. En el segundo hilo, al estar determinado el valor de “Res”, puede concluir su ejecución mostrando por pantalla el valor “10”.

# 2. Ejercicio Nº2

## 2.1. Enunciado

Dada la siguiente tabla con algunas de las características del lenguaje de programación Oz,
completar la columna de características equivalentes en su lenguaje preferido.

No se olvide de completar en el título el nombre del lenguaje.

La característica equivalente a completar, debe tener una sintaxis definida en su lenguaje
favorito y ejecutarse de forma equivalente a la característica en Oz.

S, S1 y S2 representan sentencias arbitrarias.
S1 S2 significa una secuencia de dos sentencias
V una expresión que se puede calcular (por ejemplo A + 3)

1. Describa cualquier inconveniente que encuentre al realizar el pasaje a su lenguaje favorito indicando claramente las diferencias semánticas.

2. Existe algún equivalente en su lenguaje favorito a la sentencia case de Oz, si existe descríbala y si no existe proponer como se podría simular en su lenguaje.

## 2.1. Resolución

| Caractrística en Oz | Equivalente en Go |
|---|---|
| skip | ; |
| S1 S2 | S1; S2; |
| local X in S1 end | { var x <type>; S1; } |
| X=V | var X <type>;  var V <type> = X; |
| if X then S1 else S2 end | if X { S1 } else { S2 } |
| {X Y1 Y2 Y3} | X(Y1, Y2, Y3) |

**1.** Una de las diferencias con Go, es que la sentencia “skip” se puede representar mediante una sentencia vacía o dejando en blanco el contenido de un bloque. Por ejemplo “if X {} else {}” es una sentencia válida para el compilador. A su vez, la utilización del “;” no es obligatoria pero permite aclarar el código.

Otra diferencia, es que la definición de variables en Oz se realiza para un determinado entorno al comienzo del bloque. En cambio en C/Go, esta definición puede ser realizada en cualquier momento dentro de un scope.

**2.** Actualmente existe un compilador para el lenguaje Go que permite utilizar Pattern Matching en el lenguaje, ampliando el compilador estándar de Go.

Al igual que Oz, en Go también se utiliza la sentencia “case”, pero al ser Go un lenguaje de tipado estático, se analiza no solo el valor de un patrón, sino también su tipo.

Un inconveniente surge a la hora de analizar patrones recursivamente, es decir, analizar un patrón dentro de otro patrón, ya que Go no soporta herencia de tipos, y sin esta característica sería imposible analizar una lista dentro de otra lista, por ejemplo. Para esto, se aplica el concepto de “trait”.

Mediante estas técnicas implementadas sobre el compilador estándar de Go, se da soporte al Pattern Matching en este lenguaje.

Para demostrar el funcionamiento de estos conceptos, utilizamos un ejemplo donde se aplican.

Código ejemplo Pattern-Matching:

```
1  package main
2
3  import "fmt"
4
5  /* Al definir un trait, va a ser madre de otros tipos. */
6  type Expr trait {}
7
8  /* Defino 3 subtipos de Expr. */
9  type Constante casestruct borrows Expr {
10     value  int
11 }
12
13 type Variable casestruct borrows Expr {
14     name   string
15 }
16
17 type Multiple casestruct borrows Expr {
18     left   Expr
19     right  Expr
20 }
21 /* Fin definición de subtipos. */
22
23 func main() {
24     /* Creo una entrada inicializa con las constantes 1 y 20. */
25     entrada := Multiple(Constante(1), Constante(20))
26     fmt.Printf("%s\n", entrada)
27
28     /* Trata de matchear con distintos casos. */
29     match entrada {
30         /* Caso en el cual coincide tipo y valor. */
31         case Multiple(Constante(10), Constante(20)):
32                   fmt.Printf("Matchea tipo y valor (10,20)")
33
34         /* Caso donde se realiza la asignación
35            Constante(1) => Constante(x) -> x := 1
36            Constante(20) => y := Constante(20) */
37         case Multiple(Constante(x),y):
38                   fmt.Printf("Machea x=%v, y=%v\n", x, y)
39     }
40 }
```

En este ejemplo, se puede observar que fueron creados 4 tipos, de los cuales el primero sería el padre de los otros 3 subtipos. El tipo “Multiple”, puede almacenar dos valores, los cuales pueden ser del tipo padre “Expr” (Expresión).

En la función “main” se inicializa una estructura del tipo “Multiple” con dos valores constantes. Luego, con la sentencia “match” analiza los distintos casos, e imprime aquél caso que matchea.

El primer “case” lo que hace es crear un nuevo “Multiple” con dos valores constantes (10 y 20), y el lenguaje se encarga de comparar el tipo (en este caso tipo “Constante”, el cual matchea) y luego el valor con el cual fue creado (en este caso 10 y 20, lo cual no matchea con la estructura almacenada en “entrada”), dando por resultado, en este caso, “false”, ya que no coincide completamente con la estructura “entrada”.

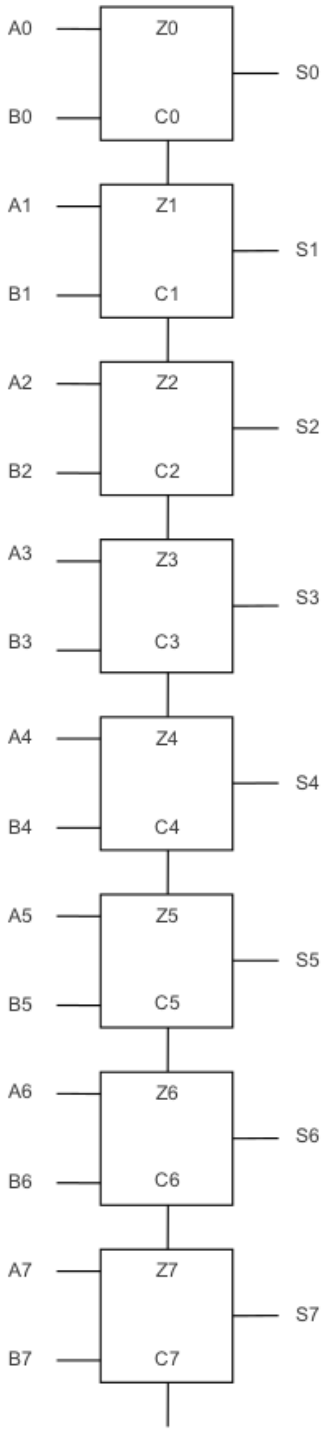
El segundo “case” crea un nuevo “Multiple”, pero al ser inicializado con dos variables (x, y) las cuales no fueron declaradas, el lenguaje se encarga de asignarlas como alias de los valores de inicialización de “entradas”. Esto produce que el matcheo sea positivo, ya que “x” es una “Constante” y “y” solo se trata de una variable, tomando el valor inicializado.

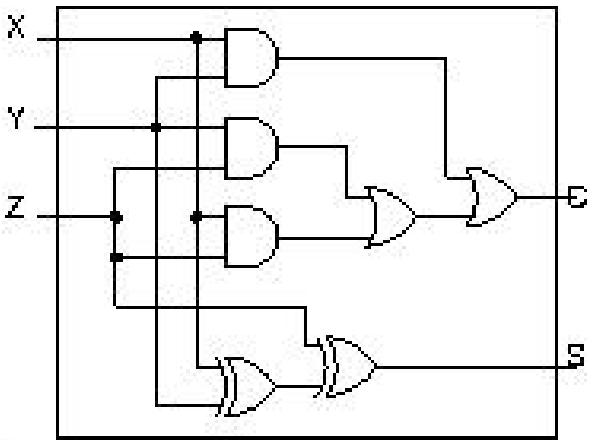
Este sistema permite realizar cosas más complejas, por ejemplo “Constante(Var(Expr(Constante(1))))” y de esta forma realizar un Pattern-Matching recursivo sobre cualquier tipo.

# 3. Ejercicio Nº3

## 3.1. Enunciado

Leer el capítulo 4 del libro y simular un sumador de 8 bits con compuertas lógicas, según el diagrama esquemático presentado a continuación. Cada compuerta debe ejecutarse en su propio hilo.



### 3.2. Resolución

```
declare R1 R2 R3 R4 R5

fun {ConstruirCompuerta F}
    fun {$ Xs Ys}
        fun {CicloCompuerta Xs Ys}
            case Xs#Ys of (X|Xr)#(Y|Yr) then
                {F X Y}|{CicloCompuerta Xr Yr}
            end
        end
    in
        thread {CicloCompuerta Xs Ys} end
    end
end

proc {SumadorCompleto X Y Z ?C ?S}
    K L M CAnd COr CNand CNor CXor
in
    CAnd = {ConstruirCompuerta fun {$ X Y} X*Y end}
    COr = {ConstruirCompuerta fun {$ X Y} X+Y-X*Y end}
    CNand= {ConstruirCompuerta fun {$ X Y} 1-X*Y end}
    CNor = {ConstruirCompuerta fun {$ X Y} 1-X-Y+X*Y end}
    CXor = {ConstruirCompuerta fun {$ X Y} X+Y-2*X*Y end}

    K={CAnd X Y}
    L={CAnd Y Z}
    M={CAnd X Z}
    C={COr K {COr L M}}
    S={CXor Z {CXor X Y}}
end


fun {SumadorOchoBit X1 X2 X3 X4 X5 X6 X7 X8 Y1 Y2 Y3 Y4 Y5 Y6 Y7
Y8}
    A0=X8|_ B0=Y8|_ C0 Z0=0|_ S0
    A1=X7|_ B1=Y7|_ C1 Z1     S1
    A2=X6|_ B2=Y6|_ C2 Z2     S2
    A3=X5|_ B3=Y5|_ C3 Z3     S3
    A4=X4|_ B4=Y4|_ C4 Z4     S4
    A5=X3|_ B5=Y3|_ C5 Z5     S5
    A6=X2|_ B6=Y2|_ C6 Z6     S6
    A7=X1|_ B7=Y1|_ C7 Z7     S7
in
    {SumadorCompleto A0 B0 Z0 C0 S0}
    {SumadorCompleto A1 B1 C0 C1 S1}
    {SumadorCompleto A2 B2 C1 C2 S2}
    {SumadorCompleto A3 B3 C2 C3 S3}
    {SumadorCompleto A4 B4 C3 C4 S4}
    {SumadorCompleto A5 B5 C4 C5 S5}
```

```
49      {SumadorCompleto A6 B6 C5 C6 S6}
50      {SumadorCompleto A7 B7 C6 C7 S7}
51
52      C7|S7|S6|S5|S4|S3|S2|S1|S0
53  end
54
55  R1={SumadorOchoBit 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1}
56  R2={SumadorOchoBit 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1}
57  R3={SumadorOchoBit 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1}
58  R4={SumadorOchoBit 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1}
59  R5={SumadorOchoBit 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1}
69
61  {Browse R1}
62  {Browse R2}
63  {Browse R3}
64  {Browse R4}
65  {Browse R5}
```

En este código se implementa un SumadorOchotBit, el cual recibe dos números de 8 bits y devolviendo la suma de ellos, más un valor de acarreo para su mejor verificación.

Como funciones auxiliares se definieron las distintas compuertas lógicas enunciadas en el gráfico, siendo estas parte del SumadorCompleto. Este se encarga de abstraer los distintos casos enunciados en el gráfico. Las compuertas son creadas en base a la función ConstruirCompuerta, la cual recibe una condición de compuerta y la ejecuta recursivamente.

Finalmente, la función SumadorOchoBit se encarga se controlar el stream de datos que se quiere sumar, y enviando estos valores al SumadorCompleto. Una vez realizada la suma, se devuelve una lista con el resultado final y un valor de acarreo.

En el programa principal se realiza una prueba del contador, sumando varios pares de números por separado, y luego mostrando los resultados por pantalla.

# 4. Ejercicio Nº4

## 4.1. Enunciado

Implementar en Oz, un procedimiento que reciba una lista con letras y muestre por pantalla la frecuencia de aparición de cada letra, ordenada por frecuencia y a igual cantidad de apariciones ordenadas alfabéticamente.

Usar como estructura de datos un diccionario con las siguientes características:

1. Diccionario declarativo, seguro y desempaquetado.
2. Diccionario con estado, inseguro y empaquetado. ¿Para qué puede ser útil un tipo de datos con esas características? Comparar con las clases en Python.

Por ejemplo dada la lista de entrada:

```
[e l q u e s a b e s a b e y e l q u e n o e s j e f e]

Debería mostrar:

e: 9
s: 3
a: 2
b: 2
l: 2
q: 2
u: 2
f: 1
j: 1
n: 1
o: 1
y: 1
```

El TAD Diccionario debe soportar las siguientes primitivas:

**{NewDicc}**: devuelve un diccionario vacío
**{Put Dicc Clave Valor}**: inserta el par Clave:Valor en el diccionario
**{Get Dicc Clave}**: devuelve el Valor asociado a la Clave
**{Domain Dicc}**: devuelve una lista con todas las claves presentes en el diccionario Dicc.

### 4.1 Resolución

Implementación del Diccionario Declarativo en **Oz**.

Archivo **ej4-1.oz**

```
local NuevoWrapper NuevoStack NuevoDicc Put Get Dominio CargarFrec
AgregarOrdenado Push Pop EstaVacio Frecuencias Wrap Unwrap
MostrarFrecuencias ListaInicial ListaFrecuencias ElDiccionario in

   % --- Funciones Auxiliares de Wrapper y Stack ---

   % Procedimiento NuevoWrapper -> Crea las funciones Wrap y
Unwrap.
   proc {NuevoWrapper ?Wrap ?Unwrap}
      local Key={NewName} in
       fun {Wrap X}
          fun {$ K}
             if K==Key then X end
          end
       end
       fun {Unwrap W} {W Key} end
      end
   end

   % Función NuevoStack -> Devuelve un Wrapper vacío
   fun {NuevoStack} {Wrap nil} end

   % Función Push -> Recibe un Stack y un Elemento. El Elemento es
agregado al Stack.
   fun {Push UnStack Elem} {Wrap Elem|{Unwrap UnStack}} end

   % Función Pop -> Recibe un Stack. Devuelve un Elemento del
Stack.
   fun {Pop UnStack ?Elem}
      case {Unwrap UnStack} of X|S1 then
       Elem=X
       {Wrap S1}
      end
   end

   % Función EstaVacio -> Recibe un Stack. Devuelve un booleano en
caso positivo o negativo.
   fun {EstaVacio UnStack} {Unwrap UnStack}==nil end

   % --- Funciones del Diccionario ---

   % Función NuevoDicc -> Crea un nuevo diccionario utilizando un
Stack.
   fun {NuevoDicc} {NuevoStack} end

```

```
   % Función Put -> Agrega elementos al diccionario. Recibe un
Diccionario y agrega un Valor y Clave.
   fun {Put Dicc Clave Valor}
      local DiccAux Elemento ElementoAux in
       if {EstaVacio Dicc} then
          ElementoAux=counter(clave:Clave valor:Valor)
          {Push Dicc ElementoAux}
       else
          DiccAux={Pop Dicc Elemento}
          if Elemento.clave==Clave then
             {Put DiccAux Clave Elemento.valor+1}
          else
             {Push {Put DiccAux Clave Valor} Elemento}
          end
       end
      end
   end

   % Función Get -> Recibe un Dicconario y la Clave a obtener su
valor. Devuelve el valor de la clave.
   fun {Get Dicc Clave}
      local DiccAux Elemento in
       if {EstaVacio Dicc} then nil
       else
          % Diccionario Auxilar con un elemento menos.
          DiccAux={Pop Dicc Elemento}
          if Elemento.clave==Clave then
             Elemento.valor
          else
             % Analiza recursivamente el Diccionario Auxiliar
restante.
             {Get DiccAux Clave}
          end
       end
      end
   end

   % Funcion Dominio -> Recibe un diccionario por parámetros.
Devuelve la lista de claves que existen en el Dicc.
   fun {Dominio UnDicc}
      if {EstaVacio UnDicc} then nil
      else
       local DiccAux Elemento in
          DiccAux={Pop UnDicc Elemento}
          Elemento.clave|{Dominio DiccAux}
       end
      end
   end

   % Funcion CargarFrecuencia -> Recibe una lista de claves y
devuelve un diccionario cargado con la frecuencia de cada clave en
la lista.
```

```
   fun {CargarFrec UnaLista}
      case UnaLista of
       nil then {NuevoDicc}
      [] H|T then
       local DiccAux Clave in
          Clave=H
          DiccAux={CargarFrec T}
          {Put DiccAux Clave 1}
       end
      end
   end

   % Función AgregarOrdenado -> Agrega un elemento ordenando la
lista según frecuencia y clave. Devuelve la lista con el nuevo
elemento.
   fun {AgregarOrdenado ListaOrigen Elemento}
      % Analiza la lista segun sea el caso.
      case ListaOrigen of nil then Elemento|nil
      [] H|T then
       % La frecuencia de H mayor <-> H se analiza primero
       if H.valor > Elemento.valor then
          H|{AgregarOrdenado T Elemento}
       else
          % La frecuencia de H menor <-> El Elemento es la cabeza
          if H.valor<Elemento.valor then
             Elemento|H|T
          else
             % A igual Frecuencia y Clave Mayor <-> El Elemento
es la cabeza
             if H.clave>Elemento.clave then
              Elemento|H|T
             else
              % A igual Frecuencia y Clave Menor <-> Sigue
primera la Cabeza y se analiza el resto
              H|{AgregarOrdenado T Elemento}
             end
          end
       end
      end
   end

   % Función Frecuencias -> Recibe un Diccionario y devuelve una
nueva Lista de elementos ordenada por Frecuencia y Clave.
   fun {Frecuencias UnDicc}
      if {EstaVacio UnDicc} then nil
      else
       local DiccAux ListaAux Elemento in
          DiccAux={Pop UnDicc Elemento}
          ListaAux={Frecuencias DiccAux}
          {AgregarOrdenado ListaAux Elemento}
       end
      end
```

```
150    end
151
152    % Procedimiento MostrarFrecuencias -> Imprime en pantalla las
153 frecuencias recorriendo una Lista ordenada.
154    proc {MostrarFrecuencias UnaLista}
155       case UnaLista of H|T then
156        {Browse [H.clave ':' H.valor]}
157
158        % En caso de que la cola sea una lista, la analiza.
159        case T of X|Y then
160           {MostrarFrecuencias T}
161        else
162           skip
163        end
164       end
165    end
166
167    % Se crea el Wrapper con sus dos funciones.
168    {NuevoWrapper Wrap Unwrap}
169
170    % Programa Principal -> Imprime la lista de letras y la
171 frecuencia de las letras.
172
173    % Lista de letras que se analizará.
174    ListaInicial=[e l q u e s a b e s a b e y e l q u e n o e s j e
175 f e]
176
177    % Carga la frecuencia de las letras al Diccionario.
178    ElDiccionario={CargarFrec ListaInicial}
179
180    % Genera una nueva Lista con los datos Ordenados.
181    ListaFrecuencias={Frecuencias ElDiccionario}
182
183    % Muestra el resultado final pedido.
184    {MostrarFrecuencias ListaFrecuencias}
185 end
186
187
188
```

## 4.2. Resolución

Implementación de un Diccionario con estado, inseguro y empaquetado en **Oz**.

Archivo **ej4-2.oz**

```
% Función NuevoDicc -> Crea un nuevo Diccionario y define las
funciones del mismo.
declare fun {NuevoDicc}

local Diccionario EstaVacio DiccionarioBorrar DiccionarioDominio
DiccionarioMostrar DiccionarioGet DiccionarioPut
DiccionarioOrdenarFrec Empty Domain Get Put Show Remove
CargarDatos OrdenarFrecuencia in

      % Diccionario con Estados.
      Diccionario={NewCell nil}

      % Funciones que implementan el funcionamiento de un
Diccionario.

      % Función EstaVacio -> Devuelve verdadero o falso según el
contenido del Diccionario.
      fun {EstaVacio UnDicc}
       UnDicc==nil
      end

      % Función DiccionarioBorrar -> Recibe un Diccionario Inicial
y devuelve un Diccionario alterado.
      fun {DiccionarioBorrar DiccInicial Clave}
       case DiccInicial
       of H|T then
          if H.clave==Clave then
             T
          else
             H|{DiccionarioBorrar T Clave}
          end
       else nil
       end
      end

      % Función DiccionarioGet -> Recibe un Diccionaro y la Clave
a obtener. Devuelve el valor asociado a Clave.
      fun {DiccionarioGet UnDicc Clave}
       case UnDicc
       of H|T then
          if H.clave==Clave then H.valor
          else
             {DiccionarioGet T Clave}
          end
```

```
45          else 0
46          end
47       end
48
49       % Función DiccionarioPut -> Recibe un Diccionario y la
50 Clave/Valor a agregar.
51       fun {DiccionarioPut DiccInicial ClaveA ValorA}
52          local DiccAux Elem ElemAux in
53             % Si el Diccicionario Inicial esta vacio, se agrega el
54 elemento.
55             if {EstaVacio DiccInicial} then
56                ElemAux=counter(clave:ClaveA valor:ValorA)
57                ElemAux|DiccInicial
58             else
59                DiccAux=DiccInicial.2
60                Elem=DiccInicial.1
61                % Si la cavle ya existe, se suman los valores y se
62 retorna una nueva lista.
63                if Elem.clave==ClaveA then
64                   ElemAux=counter(clave:ClaveA
65 valor:Elem.valor+ValorA)
66                   ElemAux|DiccAux
67                else
68                   % Si es otro caso, analiza el siguiente elemento.
69                   Elem|{DiccionarioPut DiccAux ClaveA ValorA}
70                end
71             end
72          end
73       end
74
75       % Función DiccionarioDominio -> Devuelve una lista con las
76 claves del Diccionario.
77       fun {DiccionarioDominio UnDicc}
78          if {EstaVacio UnDicc} then nil
79          else
80             local Elem in
81                Elem.clave|{DiccionarioDominio {DiccionarioGet
82 UnDicc Elem.clave}}
83             end
84          end
85       end
86
87       % Función DiccionarioOrdenarFrec -> Recibe un Diccionario y
88 lo ordena según la Frecuencia.
89       fun {DiccionarioOrdenarFrec UnDicc}
90          local AgregarOrdenado in
91             % Función AgregarOrdenado - Recibe una lista y devuelve
92 otra Lista con el elemento agregado segun el orden.
93             fun {AgregarOrdenado ListaInicial Elem}
94                case ListaInicial of
95                   nil then Elem|nil
96                [] H|T then
```

```
97              if H.valor>Elem.valor then
98                 H|{AgregarOrdenado T Elem}
99              else
100                if H.valor<Elem.valor then
101                 Elem|H|T
102                else
103                 if H.clave>Elem.clave then
104                    Elem|H|T
105                 else
106                    H|{AgregarOrdenado T Elem}
107                 end
108                end
109             end
110            end
111         end
112
113         if {EstaVacio UnDicc} then nil
114         else
115            local DiccAux ListaAux Elem in
116             DiccAux=UnDicc.2
117             Elem=UnDicc.1
118             ListaAux={DiccionarioOrdenarFrec DiccAux}
119             {AgregarOrdenado ListaAux Elem}
120            end
121         end
122       end
123      end
124
125      % Funcion DiccionarioMostrar -> Muestra el contenido del
126 Diccionario.
127      proc {DiccionarioMostrar UnaLista}
128       case UnaLista of H|T then
129          {Browse [H.clave ':' H.valor]}
130          case T of X|Y then
131             {DiccionarioMostrar T}
132          else
133             skip
134          end
135       end
136      end
137
138      % Funciones que encapsulan el funcionamiento del
139 Diccionario.
140
141      % Función Empty - Diccionario -> Determina si el Diccionario
142 esta o no vacío.
143      fun {Empty}
144       {EstaVacio @Diccionario}
145      end
146
147      % Función Dominio - Diccionario -> Devuelve una lista con
148 todas las claves presentes en el diccionario.
```

```
149       fun {Domain}
150        {DiccionarioDominio @Diccionario}
151       end
152
153       % Función Get - Diccionario -> Devuelve el Valor asociado a
154 la Clave.
155       fun {Get Clave}
156        {DiccionarioGet @Diccionario Clave}
157       end
158
159       % Función Put - Diccionario -> Inserta valores al
160 Diccionario.
161       proc {Put Clave Valor}
162        Diccionario:={DiccionarioPut @Diccionario Clave Valor}
163       end
164
165       % Función Remove - Diccionario -> Borra valores asociados a
166 la Clave.
167       fun {Remove Clave}
168        Diccionario:={DiccionarioBorrar @Diccionario Clave}
169       end
170
171       % Función Mostrar - Diccionario -> Muestra el contenido del
172 Diccionario.
173       proc {Show}
174        {DiccionarioMostrar @Diccionario}
175       end
176
177       % Función OrdenarFrecuencia - Diccionario -> Ordena el
178 diccionario segúna la frecuencia.
179       proc {OrdenarFrecuencia}
180        Diccionario:={DiccionarioOrdenarFrec @Diccionario}
181       end
182
183       % Procedimiento CargarDatos - Diccionario -> Recibe una
184 lista y carga la información al Diccionario.
185       proc {CargarDatos UnaLista}
186        case UnaLista
187        of H|T then
188           % Carga la cabeza.
189           {Put H 1}
190           % Carga la cola recursivamente.
191           {CargarDatos T}
192        [] nil then skip
193        end
194       end
195
196       % Asignación de métodos al Diccionario.
197       elDiccionario(diccionario: Diccionario
198                 empty: Empty
199                 get: Get
200                 put: Put
```

```
                remove: Remove
                domain: Domain
                ordenarFrecuencia: OrdenarFrecuencia
                cargarDatos: CargarDatos
                mostrar: Show)
   end
end

local ElDiccionario ListaLetras in
   % Se crea la frase inicial.
   ListaLetras=[e l q u e s a b e s a b e y e l q u e n o e s j e
f e]

   % Se crea un diccionario.
   ElDiccionario={NuevoDicc}

   % Se cargan los datos al Diccionario en base a la lista de
letras.
   {ElDiccionario.cargarDatos ListaLetras}

   % Ordena el contenido del Diccionario por Frecuencia.
   {ElDiccionario.ordenarFrecuencia}

   % Muestra el contenido del Diccionario.
   {ElDiccionario.mostrar}
end

```

Implementación de un Diccionario con estado, inseguro y empaquetado en **Go**.

Archivo **ej4-2.go**

```
1  package main
2
3  import (
4        "fmt"
5        "sort"
6  )
7
8  /* Creo una estrcutura de mapa ordenado */
9  type mapaOrdenado struct {
10       m map[string]int
11       s []string
12 }
13
14 func (sm *mapaOrdenado) Len() int {
15       return len(sm.m)
16 }
17
18 func (sm *mapaOrdenado) Less(i, j int) bool {
19       return sm.m[sm.s[i]] > sm.m[sm.s[j]]
20 }
21
22 func (sm *mapaOrdenado) Swap(i, j int) {
23       sm.s[i], sm.s[j] = sm.s[j], sm.s[i]
24 }
25
26 func ordenarClaves(m map[string]int) []string {
27       sm := new(mapaOrdenado)
28       sm.m = m
29       sm.s = make([]string, len(m))
30       i := 0
31       for key, _ := range m {
32             sm.s[i] = key
33             i++
34       }
35
36       sort.Sort(sm)
37       return sm.s
38 }
39
40 /* Estrucutra de Diccionario */
41 type Diccionario map[string]int
42
43 /* Creo un Diccionario vacío y lo devuelvo. */
44 func NuevoDicc() (nuevoDicc Diccionario) {
45       nuevoDicc = make(map[string]int)
46
47       fmt.Println("Se creo el dicc")
48
```

```
49        return nuevoDicc
50  }
51
52  /* {Put Dicc Clave Valor}: inserta el par Clave:Valor en el
53  diccionario */
54  func (unDicc *Diccionario) Put(clave string, valor int) {
55        (*unDicc)[clave] = valor
56  }
57
58  /* {Get Dicc Clave}: devuelve el Valor asociado a la Clave */
59  func (unDicc *Diccionario) Get(clave string) (valor int, existe
60  bool) {
61        if (*unDicc)[clave] != 0 {
62              return (*unDicc)[clave], true
63        }
64
65        return 0, false
66  }
67
68  /* {Domain Dicc}: devuelve una lista con todas las claves
69  presentes en el diccionario Dicc */
70  func (unDicc *Diccionario) Domain() (unaLista []string) {
71        unaLista = ordenarClaves((*unDicc))
72
73        return unaLista
74  }
75
76  func main(){
77        /* Creo una Cadena y un Diccionario */
78        unaCadena := "elquesabesabeyelquenoesjefe"
79        unDiccionario := NuevoDicc()
80
81        /* Cargo el Diccionario con los caracteres de la Cadena */
82        for _,vC :=range(unaCadena){
83
84              vD, ok := unDiccionario.Get(string(vC))
85
86              if ok {
87                    /* El valor vC existe y se aumenta a vC+1 */
88                    unDiccionario.Put(string(vC), vD+1)
89              } else {
90                    /* El valor vC NO existe y se aumenta a 1 */
91                    unDiccionario.Put(string(vC), 1)
92              }
93        }
94
95        dominio := unDiccionario.Domain()
96
97        /* Recorre e imprime el diccionario ordenado */
98        for _,v :=range(dominio){
99              cant, _ := unDiccionario.Get(v)
100             fmt.Println(v, ":", cant)
```

```
101         }
102 }
```

Se implementó el diccionario del punto 4.2, ya que Go utiliza un modelo de estados. También es del tipo empaquetado ya que las funciones están incorporadas a los tipos. Y por último, es inseguro ya que el encapsulamiento puede ser definido por el programador debido a que el lenguaje no fuerza a hacerlo.