Manual de Usuario

Sintaxis del lenguaje 4GL.

Al igual que un programa en C, un programa en 4GL consta de uno o varios módulos en los que se definen variables, funciones, formularios de entrada y listados. La estructura de cada módulo será la siguiente:

```
[DATABASE <id>]
[declaraciones variables globales]
funciones
[forms]
[reports]
```

En uno de los módulos debe haber una función llamada *'main'*. La función *main* es la que se ejecuta al iniciarse el programa.

Si el programa accede a una base de datos, será necesario especificar el nombre de la misma en la primera línea del programa, mediante la línea '`DATABASE <nombre>`'

Comentarios

Se pueden incluir comentarios dentro de los programas para que su código sea más fácil de comprender. ***hy4GL*** permite introducir los comentarios entre llaves, en el estilo de Informix−4GL o bien entre caracteres /* y */, al estilo de C. El compilador detecta los comentarios anidados e ignora todo el código que encuentra entre las marcas de inicio y final de comentario.

Ej:

```
{Funcion: suma_totales;
 calcula el importe total de una fra.}
function suma_totales()
define j integer /* j lo uso para bucles */
/* ya no necesito esta variable.
define suma integer /* para ir calculando... */
*/
...
```

Declaraciones de variables:

Tipos de datos:

El lenguaje soporta los siguientes tipos de datos básicos:

- Integer

  El rango permitido para números enteros es el de un '*long int*' de C. Dependerá de la plataforma en la que se esté trabajando. En Linux, usando gcc 2.7, el rango es entre –2.147.483.647 y 2.147.483.648

- Float

  Este tipo permite usar números reales. Se implementa mediante el tipo de datos '*double*' de C, por lo que la precisión dependerá del compilador de C usado.

- Char(x)

  Permite usar cadenas de caracteres. No hay límite para el tamaño de la cadena. A diferencia del lenguaje C, no es necesario reservar un carácter para la marca de fin de cadena puesto que lo reserva el compilador automáticamente.

  Se puede omitir la longitud de la cadena. En este caso, la variable tendrá una longitud de 1 carácter.

  En este tipo de datos sólo se podrán almacenar caracteres *imprimibles*. No está pensado para almacenar datos binarios.

- Date

  Para almacenar fechas. Se permiten fechas desde el 1de Enero del año 1 (d.C.) hasta bien pasado el año 40.000 (creo que con esto no habrá problemas en el 2000).

- Time

  Permite almacenar horas con precisión hasta las milésimas de segundo.

  Aunque el tipo de datos permite almacenar la hora con esa precisión, las milésimas de segundo se pierden al realizar operaciones aritméticas.

- Datetime

  Almacena un instante, representado por fecha y hora. Las características comentadas en los dos tipos anteriores son aplicables a éste.

- Interval

  Almacena un intervalo de tiempo, representado por un número de horas, minutos y segundos. La diferencia entre dos '*datetimes*' es un intervalo.

Además de los tipos de datos básicos, se permite definir registros, formados por campos de tipos básicos. No se permite que un campo de un registro sea, a su vez, otro registro.

Declaración de variables:

Como en cualquier lenguaje de programación, ***hy4GL*** permite definir variables en las que se almacenarán los datos con los que se está trabajando. ***hy4GL*** es un lenguaje tipado: cada variable es de un tipo determinado, aunque se verá más adelante que el propio lenguaje realiza la conversión de tipos automáticamente.

El nombre de las variables debe empezar por una letra o un signo de subrayado '_', y puede contener cualquier combinación de letras, números y signos '_'.

El ámbito de una variable depende de la parte del módulo en la que se declare. Así, si una variable se declara dentro de una función, será visible (se podrá acceder a su contenido) solamente desde dentro de esa misma función. Si la variable se declara fuera de una función, será visible desde cualquier función de este módulo. No es posible usar en un módulo una variable que esté declarada en otro módulo distinto.

La sintaxis de la declaración de variables es la siguiente:

```
DEFINE <id>[,<id>,...] <tipo>
```

Ej:

```
define i integer
define hora time
define hoy,ayer date
define ahora datetime
define nombre,apellidos char(50)
define tecla char
```

Si la aplicación accede a bases de datos, es posible definir variables del mismo tipo que determinado campo de una tabla en la base de datos. La sintaxis para ello es:

```
DEFINE <id> LIKE <tabla>.<campo>
```

Para ello es necesario que se haya indicado el nombre de la base de datos en la instrucción `DATABASE <nombre>`

Ej:

```
database contabilidad
define cuenta like cuentas.numero
define importe like mayor.importe
```

Es posible definir registros que agrupen varias variables en una sola estructura. Para definir un registro se usará la siguiente sintaxis:

```
DEFINE <id> RECORD
   <nombre> <tipo>
   ...
END RECORD
```

Para acceder a un campo de un registro se especificará el nombre del registro y el del campo separados por un punto. Para referirse a la lista completa de campos de un registro, se usará el nombre del registro seguido de '.*'

Ej:

```
define cliente record
   codigo integer
   nombre char(50)
   apellidos char(50)
end record
let cliente.nombre="JUAN"
let cliente.*=busca_cliente(cliente.nombre)
```

Se puede definir un registro con la misma estructura que una tabla de la base de datos:

```
DEFINE <id> RECORD LIKE <tabla>.*
```

Por supuesto, será necesario haber indicado el nombre de la base de datos mediante la instrucción `DATABASE`

Ej:

```
database contabilidad
define reg record like mayor.*
define rcta record like cuentas.*
```

Funciones:

Las funciones se declararán con la siguiente sintaxis:

FUNCTION <nombre> ([par1 [,par2,...]])

declaraciones variables locales y parámetros

sentencias

END FUNCTION

Si la función rebice parámetros, se relacionarán los nombres de los parámetros formales entre los paréntesis. Hay que tener en cuenta que los parámetros habrá que declararlos dentro de la función, junto con las variables locales, si las hubiera.

Ej:

```
function graba_registro(r,donde)
define r record like clientes.*
define donde char(20)
   if donde="clientes" then
      insert into clientes values (r.*)
   else
      insert into proveedores values(r.*)
   end if
end function
```

La función *main*, es una excepción. Su sintaxis es:

MAIN

declaraciones variables locales

sentencias

END MAIN

La función *main* es, como se comentó anteriormente, la primera función que se ejecuta al lanzar la aplicación. La variable global *arg_count* indica el número de parámetros que se han enviado al programa, y la función *arg_val(*i*)* devuelve el parámetro número *i*. Al igual que en C, el parámetro número cero es el nombre del programa.

Ej:

```
main
define param char(50)
   if arg_count=0 then
      display "Error. Numero de parametros erroneo."
      exit program
   end if
   let param=arg_val(1)
   ...
end main
```

Una función puede devolver varios parámetros usando la sentencia *return*, cuya sintaxis es:

```
RETURN [<expresion1>[,<expresion2>[,....<expresion_n>]]]
```

El lenguaje soporta recursividad sin que exista más límites que los impuestos por el compilador de C usado

Ej:

```
function factorial(n)
define n integer
   if n=0 then
      return 1
   else
      return n*(factorial(n-1))
   end if
end function
```

Expresiones:

Expresiones simples:

hy4GL soporta las siguientes expresiones simples:

- `Variable`

  Basta con indicar el nombre de la variable. El tipo de la expresión será el mismo que el tipo de la variable. La variable debe ser de un tipo básico, no permitiéndose para este tipo de expresiones nombres de registros.

  Ej:

  ```
  define a integer
  define r record like clientes.*
  display a
  display r.nombre
  ```

- `Número`

  Se pueden usar números enteros o reales. Para indicar decimales se usa el punto '.' y no se deben usar separadores para los miles. Si el número no lleva el punto decimal, la expresión será de tipo `entero`, mientras que será de tipo `float` en caso de llevarlo.

  Ej:

  ```
  display 4
  display 4.5
  ```

- `Cadena`

  Se expresarán las cadenas entre comillas dobles "". La expresión será de tipo cadena.

  Ej:

  display "Error."

  display "No se encuentra el cliente"

Operadores USING, CLIPPED y [ ]:

Operador USING

El operador `USING` convierte una expresión numérica al formato especificado en la máscara.

La sintaxis es la siguiente:

```
expresion USING mascara
```

La máscara consistirá en una serie de caracteres '–','#','&',',' y '.'. La máscara es aplicada a la expresión numérica, devolviendo una cadena en la que los caracteres '–','#' y '&' son sustituidos por los dígitos correspondientes. El símbolo '#' es usado para rellenar con espacios a la izquierda; el símbolo '–' rellena también con espacios, excepto en la primera posición en la que no haya dígtos, que será sustituida por el signo '–' en caso de que el número sea negativo. El símbolo '&' se usa para rellenar con ceros las posiciones correspondientes. Los signos ',' se usan para separar los miles y el '.' para indicar la posición del punto decimal

Ej:

```
0 using "##,###.##"         -> "      .  "
10.23 using "##,###.##"     -> "    10.23"
1123.23 using "##,###.##"   -> " 1,123.23"
-1123.23 using "&&,&&&.&&" -> "01,123.23"
-1123.23 using "--,--&.&&" -> "-1,123.23"
```

Operador CLIPPED

El operador `CLIPPED` corta los espacios finales de una cadena. La expresión resultante es una nueva cadena. Su sintaxis es la siguiente:

```
expresion CLIPPED
```

Ej:

```
let nombre=   "JUAN         "
let apellidos="PEREZ"
let nomape=nombre+apellidos  /* nomape="JUAN         PEREZ" */
let nomape=nombre clipped+apellidos /* nomape="JUANPEREZ" */
let nomape=nombre clipped+" "+apellidos /* nomape="JUAN PEREZ*/
```

Operador [ ]

El operador [ ] permite obtener una subcadena a partir de una expresión. La nueva expresión resultante es de tipo cadena. Su sintaxis es la siguiente:

```
expresion[i,j]
expresion[i]
```

Entre los corchetes se indicarán la posición de inicio y final de la subcadena que se desea obtener. Se puede omitir la posición de final, con lo que se tomará la subcadena que comienza y termina en la posición ’i’.

Ej:

```
define ccc char(20)
define entidad,sucursal char(4)
define dc1,dc2 char
define cuenta char(10)
prompt "Introduzca su numero de cuenta (20 digitos) " for ccc
let entidad=ccc[1,4]
let sucursal=ccc[5,8]
let dc1=ccc[9]
let dc2=ccc[10]
let cuenta=ccc[11,20]
```

Llamadas a funciones

Los valores devueltos por una función son usados como expresiones. El tipo de estas expresiones dependerá del tipo de los valores devueltos por la función.

Ej:

```
define hora time
let hora=time() /* la funcion time() devuelve la hora actual */
```

Operadores binarios

Los siguientes operadores transforman dos expresiones en una nueva expresión. El tipo de la expresión resultante dependerá de los tipos de las expresiones originales.

Operador +

Este operador toma dos expresiones y devuelve una nueva expresión, resultado de sumar las dos expresiones originales. El significado de *sumar* depende del tipo de las expresiones originales. En el siguiente cuadro se detalla el comportamiento de este operador:

+	***Char*** "10"	***Integer*** (5)	***Float*** (5.5)	***Date*** (10/12/1998)	***Time*** (10:10:05)	***Datetime*** (10/12/1998 10:10:05)	***Interval*** (123:10:20)
Char ("10")	Char "1010"	Char "105"	Char (105.5"	Char "1010–12–1998"	Char "1010:10:05"	Char "1010–12–1998 10:10:05"	Char "10123:10:20"
Integer (5)	Char "510"	Integer 10	Float 10.5	Date 15/12/1998	Time 10:10:10	Datetime 10–12–1998 10:10:10"	Interval 123:10:25
Float (5.5)	Char "5.500010"	Float 10.5	Float 11	Date 15/12/1998	Time 10:10:10	Datetime 10–12–1998 10:10:10"	Interval 123:10:25
Date (10–12–1998)	Char "10–12–199810"	Date 15–12–1998	Date 15–12–1998	No permitido	Datetime 10–12–1998 10:10:05	No permitido	No permitido
Time (10:10:05)	Char "10:10:05.00010"	Time 10:10:10	Time 10:10:10	Datetime 10–12–1998 10:10:05	No permitido	No permitido	Time 13:20:25
Datetime (10–12–1998 10:10:05)	Char 10–12–1998 10:10:05.00010	Datetime 10–12–1998 10:10:10	Datetime 10–12–1998 10:10:10	No permitido	No permitido	No permitido	Datetime 15–12–1998 13:20:25
Interval (123:10:20)	Char 123:10:20.010	Interval 123:10:25	Interval 123:10:25	No permitido	Time 13:20:25	Datetime 15–12–1998 13:20:25	Interval 246:20:40

Operador –

Devuelve la expresión resultante de calcular la diferencia entre las dos expresiones originales. El comportamiento de este operador, así como el tipo de la expresión resultante depende de las expresiones originales:

–	***Char*** ***"10"***	***Integer*** ***(5)***	***Float*** ***(5.5)***	***Date*** ***(10/12/1998)***	***Time*** ***(10:10:05)***	***Datetime*** ***(10/12/1998 10:10:05)***	***Interval*** ***(123:10:20)***
Char ***("10")***	No permitido	No permitido	No permitido	No permitido	No permitido	No permitido	No permitido
Integer ***(5)***	Integer −5	Integer 0	Float −0.5	No permitido	No permitido	No permitido	No permitido
Float ***(5.5)***	Float −4.5	Float 0.5	Float 0	No permitido	No permitido	No permitido	No permitido
Date ***(10−12−1998)***	No permitido	Date 5−12−1998	Date 15−12−1998	Integer 0 diferencia en dias	No permitido	Integer 0 diferencia en dias	No permitido
Time ***(10:10:05)***	No permitido	Time 10:10:00	Time 10:10:10	No permitido	Time 00:00:00	No permitido	Time 06:59:45
Datetime ***(10−12−1998 10:10:05)***	No permitido	Datetime 10−12−1998 10:10:00	Datetime 10−12−1998 10:10:00	Interval 10:10:05	Datetime 10−12−1998 00:00:00	Interval 00:00:00	Datetime 5−12−1998 06:59:45
Interval ***(123:10:20)***	No permitido	Interval 123:10:15	Interval 123:10:15	No permitido	No permitido	No permitido	Interval 00:00:00

Operadores * y /

Estos operadores realizan las operaciones aritméticas de producto y división respectivamente. Sólo pueden operar entre números, ya sean reales o enteros. La expresión que se obtiene es de tipo real.

Si se intenta operar con expresiones que no son de tipo numérico, éstas se convierten automáticamente a tipo real.

Operadores de comparación

Para realizar comparaciones de expresiones se usan los operadores <, <=, =, !=, >= y >

```
expresion1 <  expresion2
expresion1 <= expresion2
expresion1 =  expresion2
expresion1 != expresion2
expresion1 >= expresion2
expresion1 >  expresion2
```

Las expresiones numéricas se comparan, como es de esperar, por su valor. Las expresiones de tipo cadena se comparan según su orden alfabético, teniendo en cuenta que todas las mayúsculas van delante de las minúsculas.

Las expresiones de tipo fecha/hora se comparan según su antigüedad.

De esta manera, se cumple que:

01–01–1998 < 02–01–1998

01–01–1998 10:10:25 > 01–01–1998 10:10:24

Estas expresiones devuelve 1 si se cumple la condición, y 0 en caso contrario.

Operadores lógicos

Las siguientes expresiones realizan las operaciones lógicas comunes:

```
! expresion
expresion AND expresion
expresion OR expresion
```

Estas expresiones devuelven *verdadero* (1) o *falso* (0) según se cumpla o no la condición.

Precedencia de operadores

La precedencia de los operadores en ***hy4GL*** es, de mayor a menor, la siguiente:

```
!
USING CLIPPED
[]
* /
+ -
<  <=  =  != >  >=
AND
OR
```

Se han colocado en la misma línea aquellos operadores que tienen la misma precedencia. Se pueden agrupar expresiones entre paréntesis para conseguir que las operaciones se realicen en un orden determinado

Sentencias:

C Embebido (ampliación a Informix−4GL)

El lenguaje aporta una gran funcionalidad y permite al programador relizar la mayoría de las acciones necesarias para el desarrollo de una aplicación de bases de datos. Sin embargo, siempre hay acciones que no es posible realizar directamente usando las sentencias del lenguaje 4GL o que por motivos de eficiencia es preferible realizarlas en lenguaje C.

Para estos casos, ***hy4GL*** permite insertar sentencias en C dentro del fuente en 4GL. La sintaxis para ello es la siguiente:

```
...
sentencias 4GL
.c
   [declaraciones C]
   sentencias C
c.
sentencias 4GL
...
```

Dentro del bloque de C se puede acceder a los datos almacenados en las variables del programa. Véase la *guía de implementación* para conocer los tipos de datos con los que se implementan las variables. También se pueden definir variables cuyo ámbito será el del bloque de C exclusivamente.

Ej:

```
define usuario char(50)
define directorio char(255)
/* leo el nombre de usuario y el directorio actual *
.c
   strncpy(usuario,getenv("LOGNAME"),sizeof(usuario)-1);
   getcwd(directorio,sizeof(directorio-1));
c.
if usuario="root" then
   display "No puede ejecutar el programa como root."
   exit program
end if
```

Asignación:

Para asignar valores a variables se usará la sentencia LET. Su sintaxis es la siguiente:

```
LET <id> = expresion
```

El lenguaje realiza la *conversión automática de tipos*, por lo que no es necesario que la expresión sea del mismo tipo que la variable de destino.

Ej:

```
define dt datetime
define d date
define i integer
define c char(40)
let dt=datetime() /* Asigna a dt la fecha y hora actual */
let d=dt /* Asigna a 'd' la fecha actual */
/* Asigna a 'c' una cadena con la fecha y hora contenida
   en 'dt' en el formato "dd-mm-aaaa HH:MM:SS.mmm" */
let c=dt
let i="50"+"20"  /* Asigna a 'i' el numero 5020 */
```

Asignaciones múltiples (ampliación a Informix-4GL)

Se permite realizar varias asignaciones en una misma sentencia LET. La sintaxis para ello es la siguiente[1]:

```
LET <id1>,<id2>,... = expresion1,expresion2,...
```

A la variable `id1` se le asignará el valor de la expresión `expresion1`; a la variable `id2` el valor de la expresión `expresion2`, etc. Habrá que prestar especial atención en que el número de variables de destino sea igual al número de valores devueltos por las expresiones. Especial cuidado hay que tener al llamar a funciones, ya que una función puede devolver varios valores:

```
define sw_encontrado integer
define nombre,apellidos,direccion char(50)
LET sw_encontrado, nombre, apellidos = datos_cliente("0001")
```

Esta construcción puede ser útil para operaciones como intercambio de valores entre variables

1 **Nota para los usuarios de Informix-4GL**: En Informix-4GL se usa la coma ',' para concatenar cadenas. Una consecuencia directa de esto es que no se puedan concatenar cadenas al enviar parámetros a funciones, debiendo asignar primero las cadenas a una variable y después enviar dicha variable a la función. Para evitar estos problemas, ***hy4GL*** usa el operador '+' para concaternar cadenas.

(*swap*), para lo cual no es necesario una variable temporal:

```
LET a,b=b,a
```

Al igual que en cualquier situación en la que se puede expresar una lista de variables, es posible asignar valores a todos los campos de un registro de la siguiente manera:

```
LET <id>.* = expresion1,expresion2,...
```

Ej:

```
LET cliente.*="JUAN","PEREZ","C/ RABIDA, 10",21003,"HUELVA"
```

Estructuras condicionales y repetitivas:

hy4GL ofrece las siguientes construcciones para variar el flujo del programa:

EXIT PROGRAM

Esta sentencia provoca el final inmediato del programa. Se desconecta de la base de datos (si estuviera conectado) y el terminal vuelve a su estado normal. No se borra la pantalla.

Ej:

```
define usuario like usuarios.nombre
prompt "Usuario: " for usuario
select * into usuario from usuarios where nombre=usuario
if status=notfound then
   display "Acceso denegado\n"
   exit program
end if
```

Llamadas a funciones

Se puede realizar una llamada a una función mediante la sentencia CALL. La sintaxis de esta sentencia es la siguiente:

```
CALL funcion(expr1,expr2,...) [ RETURNING var1,var2,...]
```

En caso de que la función devuelva uno o más valores, se usará la cláusula `RETURNING` para asignar estos valores a variables.

Ej:

```
define cliente record like clientes.*
prompt "Codigo del cliente: " for cliente.numero
call busca_cliente(cliente.numero) returning cliente.*
```

Condiciones IF... THEN... ELSE... END IF

Esta estructura permite la ejecución de una serie de sentencias sólo en caso de que se cumpla una condición. De manera opcional se puede indicar también un conjunto de sentencias que se ejecutarán si la condición no se cumple.

La sintaxis es la siguiente:

```
IF <condicion> THEN
   sentencias1
[ELSE
   sentencias2]
END IF
```

En primer lugar se evaluará la condición. En caso de que ésta se cumpla, se ejecutará el grupo de sentencias *sentencias1*. Si la condición no se cumple, se ejecutará el grupo de sentencias *sentencias2*.

Ej:

```
if dow(today())="Domingo" then
   display "Cerrado."
   exit program
else
   call menu_ppal()
end if
```

Bucles WHILE

Esta construcción permite ejecutar un conjunto de sentencias mientras se cumpla una condición. La condición se evaluará *antes* de entrar en el bucle, por lo que si ésta no se cumple, es posible que no se entre nunca en el bucle.

La sintaxis es la siguiente:

```
WHILE <condicion>
   sentencias
   CONTINUE WHILE
   EXIT WHILE
   ...
END WHILE
```

Dentro del bucle se pueden usar las sentencias `CONTINUE WHILE` y `EXIT WHILE`. La primera salta hasta el principio del bucle, ignorando las sentencias que se encuentran después de ésta. `EXIT WHILE` sale del bucle directamente.

Ej:

```
define i,total,parcial integer
let i=0
let total=0
/* Facturas que suman nuestro primer millon */
while total<1000000
   let i=i+1
   /* la factura numero 3 no la sumo... */
   if i=3 then
      continue while
   end if
   select importe into parcial from facturas where numero=i
   if status=notfound then
      display "Error... no se encuentra la factura n°",i
      let total=0
      exit while
   end if
   display "Fra: ",i," Importe: ",parcial using "###,###,##&"
   let total=total+parcial
end while
```

Bucles FOR

Los bucles FOR son usados para ejecutar un conjunto de sentencias variando el valor de una variable desde un valor inicial hasta un valor final.

La variable que se use para ir variando su valor debe ser de tipo numérico o de tipo fecha. En la primera iteración su valor será el de la expresión1, y se irá incrementando en una unidad hasta alcanzar el valor de la expresión2.

```
FOR <variable> = <expresion1> TO <expresion2>
   sentencias
   CONTINUE FOR
   EXIT FOR
   ...
END FOR
```

Al igual que en los bucles WHILE, se pueden usar las sentencias `CONTINUE FOR` y `EXIT FOR` para forzar una nueva iteración o para salir inmediatamente del bucle.

Ej:

```
{Mostramos todos los dias laborables de este mes
 a partir de hoy}
define m integer
define d date
let mes=month(today())
for d=today() to today()+31
   if month(d)!=mes then
      exit for
   end if
   if dow(d)="Domingo" or dow(d)="Sabado" then
      continue for
   end if
   display d
end for
```

Interfaz de usuario

Operaciones con ventanas.

La información que aparece en pantalla se muestra dentro de *ventanas*. Todo programa en ***hy4GL*** usa, al menos, una ventana que se crea automáticamente al iniciar el programa y que ocupa toda la pantalla.

OPEN WINDOW

Se pueden definir ventanas de cualquier tamaño y en cualquier posición, siempre que quepan dentro de los límites de la pantalla. Si se intenta crear una ventana de mayor tamaño que la pantalla, ésta será reducida hasta caber en pantalla. La sentencia para crear una nueva ventana es `OPEN WINDOW`, cuya sintaxis es la siguiente[2]:

```
OPEN WINDOW <nombre> AT <y>,<x> WITH <h> ROWS,<w> COLUMNS
[ATTRIBUTE ( <color> [OVER <color>][,BRIGHT][,BORDER] )]
[TITLE expresion]
```

Ej:

```
open window w1 at 1,1 with 10 rows,78 columns
  attribute(white over red,bright,border)
  title "Tareas para el dia "+today()
display "Esta ventana se cerrará en 1 segundo"
delay 1
close window w1
```

Si se indica un atributo de color, la ventana se muestra con ese atributo, y todo lo que se escriba dentro de ella tendrá ese mismo atributo si no se indica lo contrario. Si entre los atributos se incluye la opción '`BORDER`', la ventana se muestra con un borde que la rodea. El borde se dibuja en las coordenadas indicadas, y la parte útil de la ventana queda reducida en dos líneas y dos columnas, que son las que ocupa el borde[3].

Una vez que se abre una nueva ventana, ésta pasa a ser la ventana activa.Todo lo que se muestre en pantalla a partir de ese instante aparecerá en esta ventana.

Las ventanas sólo pueden ser referenciadas desde la función en la que se han creado. No se puede cerrar ni realizar ninguna operación sobre una ventana desde una función distinta, excepto escribir en ella.

Las ventanas son apilables: se puede definir una ventana *encima* de otra sin que se pierda el contenido de la ventana que queda oculta.

2 La opción TITLE es una ampliación a la sintaxis de Informix−4GL.
3 Informix−4GL muestra un borde 'exterior' a la ventana.

CURRENT WINDOW

Para cambiar la ventana activa se usa la sentencia `CURRENT WINDOW`, cuya sentencia es:

```
CURRENT WINDOW <nombre>
```

Sólo es posible activar una ventana desde la función en la que ésta se ha definido. Una vez activada una ventana, todo lo que se muestre en pantalla aparecerá en ella.

Ej:

```
if status=notfound then
   current window wmensajes
   prompt "No se encuentra el registro." for char tecla
   current window wproceso
end if
```

La ventana pasa a estar en lo más alto de la pila de ventanas. Si la ventana estaba ocultada total o parcialmente por otra ventana, ahora será visible totalmente.

MOVE WINDOW

En ocasiones es necesario mover una ventana para poder abrir otra sin ocultar la anterior. La sintaxis de la sentencia para mover una ventana es:

```
MOVE WINDOW <nombre> TO <y>,<x>
```

La ventana debe haberse creado en la misma función desde la que se mueve. Esta sentencia no cambia el estado de la ventana en cuanto a si está o no activa, o su posición en la pila de ventanas.

CLEAR WINDOW

Esta sentencia borra el contenido de la ventana activa.

CLOSE WINDOW

Para cerrar una ventana se usa la sentencia CLOSE WINDOW. La ventana debe haber sido creada desde la misma función en la que se cierra. Si ésta era la ventana activa, la nueva ventana activa pasa a ser la que estuviera inmediatamente debajo de ésta en la pila de ventanas (la última que estaba activa antes que ésta).

```
CLOSE WINDOW <nombre>
```

Presentación en pantalla

Para mostrar datos en pantalla se usa la sentencia `DISPLAY`, cuya sintaxis es la siguiente:

```
DISPLAY expresion [,expresion...] [AT y,x]
      [ATTRIBUTE ( color [over color] [,bright] )]
   [,expresion ... AT ... ATTRIBUTE...]
```

Las expresiones son presentadas siempre en la ventana activa. Si no se incluye la cláusula `AT`,las expresiones son mostradas en la posición actual del cursor. Los atributos de color con los que aparecerán las expresiones son los de la ventana activa por defecto, o los indicados en la cláusula `ATTRIBUTE`.

El cursor se mueve a la posición del último carácter después de ejecutar la sentencia DISPLAY. Si no se ha indicado la cláusula AT, se imprimirá un retorno de carro después de mostrar todas las expresiones. Si se ha llegado a la última línea de la ventana activa, todo el contenido de la misma subirá una línea.

Ej:

```
display "Nombre: ",cliente.nombre at 1,1
      attribute (cyan over blue, bright),
 "Dirección: ",cliente.direccion at 2,1
      attribute(cyan over blue)
```

Introducción de datos simples[4]

La instrucción `PROMPT` permite obterner el valor de una variable a través del teclado. El programa presentará en pantalla una expresión opcional, su ejecución se dentendrá hasta que se introduzca una cadena y se pulse la tecla Intro. La cadena es convertida al tipo de dato de la variable de destino.

Si se especifica la opción CHAR, el valor de la primera tecla pulsada es asignado a la variable de destino, sin que se espere a la pulsación de la tecla Intro.

4 Véase también la sección ’Formularios de Entrada’.

La sintaxis de la sentencia PROMPT es la siguiente:

```
 PROMPT [expresion] FOR [CHAR] <variable>
```

Ej:

```
define tecla char
define i,j integer
prompt "Introduce un numero: " for i
display "Introduce otro numero: "
prompt for j
prompt "Pulse una tecla para continuar..." for char tecla
```

Menús:

hy4GL ofrece la sentencia `MENU` para la definición de menús de opciones. Un menú consiste en una serie de opciones, cada una de las cuales tiene asociada un conjunto de sentencias que se ejecutarán en caso de ser seleccionadas. Además, es posible indicar un grupo de acciones que se ejecutarán al pulsar una tecla determinada.

La sintaxis para la definición de menús es la siguiente:

```
MENU [HORIZONTAL|VERTICAL]
[ATTRIBUTE (color [OVER color]) [ ; (color [OVER color])]]
  COMMAND <expresion> [,<expresion>]
     sentencias
  ON KEY (<id>)
     sentencias
     ...
     EXIT MENU
     ...
END MENU
```

Las distintas opciones se relacionarán en las cláusulas `COMMAND`. La primera expresión que sigue a la cláusula es la cadena que se mostrará en pantalla. La segunda expresión es opcional e indica un texto de ayuda asociado a la opción. Este texto aparecerá en pantalla cuando el cursor se sitúe sobre la opción. Una vez que el usuario pulse la tecla Intro, se ejecutarán las sentencias relacionadas tras la cláusula `COMMAND` correspondiente a la opción seleccionada.

Las opciones aparecerán en pantalla en la primera línea de la ventana activa (las que quepan), a no ser que se indique la opción `VERTICAL`, en cuyo caso aparecerán una sobre otra. La opción activa aparece marcada por un signo '–', y se muestra con los colores invertidos. Es posible cambiar los colores por defecto tanto de todas las opciones como de la opción seleccionada mediante la cláusula ATTRIBUTE. El color de las opciones se indica en primer lugar y, opcionalmente y separado del anterior por un punto y coma ';', se indicará el atributo de color de la opción activa.

Las teclas de movimiento del cursor permiten seleccionar la opción. En caso de que haya más opciones de las que caben en pantalla, el movimiento del cursor provocará que éstas vayan apareciendo. La tecla 'Intro' selecciona una opción y provoca que se ejecuten las sentencias asociadas a la misma. Una vez ejecutadas todas las sentencias, el menú sigue activo y es posible seleccionar una nueva opción.

La cláusula `ON KEY` permite indicar un conjunto de acciones que se ejecutarán al presionar una tecla determinada.

La sentencia `EXIT MENU` provoca la salida del menú.

Ej:

```
open window w_menu at 1,1 with 5 rows,78 columns
  attribute(yellow over blue, bright,border)
menu
  command "Articulos","Gestion de almacen"
     call gest_almacen()
  command "Clientes","Gestion de clientes"
     call gest_clientes()
  command "Proveedores","Gestion de proveedores"
     call gest_prov
  command "Salir","Fin de la aplicacion"
     exit menu
  on key(F1)
     open window w_ayuda at 5,5 with 10 rows,50 columns
        attribute(white over red)
     call ayuda()
     close window w_ayuda
  on key(F8)
     exit menu
end menu
close window w_menu
exit program
```

Formularios de entrada:

Se vio anteriormente la sentencia PROMPT, que permite introducir información por teclado. En esta sección se verá el uso de *formularios* de entrada de datos. Estos formularios permiten la introducción y visualización de información de una manera más compleja que la ofrecida por la sentencia PROMPT.

Los formularios de entrada permiten al usuario editar el valor de varias variables al mismo tiempo, desplazándose con el cursor entre distintos campos.

Un formulario está compuesto por uno o más campos de edición, botones y casillas de verificación. Los campos de edición aparecen en pantalla como regiones rectangulares en las que se puede introducir un texto. Las casillas de verificación son cuadros de edición que sólo admiten los valores 'X' o ' ' (activado/desactivado). Los botones permiten ejecutar acciones al ser pulsados.

Formato

El formato del formulario se define fuera de las funciones y siguiendo la siguiente sintaxis:

```
FORM <nombre>
   INPUT <nombre> FROM y,x TO y2,x2
       [ATTRIBUTE ( color [over color] [,bright] )]
   BUTTON <nombre> AT y,x
       [ATTRIBUTE ( color [over color] [,bright] )]
   CHECK <nombre>  AT y,x
       [ATTRIBUTE ( color [over color] [,bright] )]
END FORM
```

El formulario debe tener un nombre, con el que es accedido desde el programa. Los objetos que lo componen se identifican también mediante un nombre.

Al definir el formulario se pueden indicar los atributos de color que deben tener los objetos. Si no se indica nada, tendrán los mismos que la ventana sobre la que se muestre el formulario.

Manejo

`CREATE FORM <nombre>`

La primera acción que hay que realizar para poder usar un formulario de entrada es crearlo. La sentencia `CREATE FORM` crea en memoria todas las estructuras necesarias para el manejo del formulario. El formulario es creado y asociado a la ventana actual. No se puede tener dos formularios activos al mismo tiempo en la misma ventana.

`DISPLAY FORM <nombre>`

La sentencia `CREATE FORM` prepara el formulario para su uso, pero no lo muestra en pantalla.

Para ello, se necesita usar la sentencia `DISPLAY FORM`.

```
DISPLAY BY NAME <variables>
```

Esta sentencia muestra el valor de las variables especificadas. Cada variable es mostrada dentro del campo del formulario cuyo nombre sea igual al de la variable.

`INPUT BY NAME <variables>`

Esta sentencia es usada para obtener el valor de las variables indicadas a través del formulario. Al usuario se le presenta en pantalla el formulario y podrá ir introduciendo valores en los distintos campos y moverse entre ellos mediante el uso de las teclas de movimiento de cursor. El programador puede definir una serie de sentencias que se ejecutarán al salir de un campo determinado, al pulsar una tecla determinada o al pulsar sobre un botón.

La sintaxis de esta sentencia es la siguiente:

```
INPUT BY NAME <variables>
ON KEY('x')
   sentencias
ON BUTTON <nombre>
   sentencias
AFTER FIELD <nombre>
   sentencias
   ...
   VALIDATE FIELD
   ...
   EXIT INPUT
   ...
   NEXT FIELD <nombre>
   ...
AFTER FIELD <nombre>
   ...
END INPUT
```

Cuando el usuario sale de un campo (moviendo el cursor para entrar en un nuevo campo), el contenido del buffer de edición es asignado a la variable de destino. El programador puede entonces usar las cláusulas `AFTER FIELD` para controlar la validez del dato o para realizar cualquier tipo de acción.

La sentencia `VALIDATE FIELD` puede ser usada dentro de un bloque de código que se ejecuta al pulsar sobre una tecla. Esta sentencia fuerza a que se actualice el valor de la variable de destino con el contenido del campo activo actualmente. Esto es necesario hacerlo dentro de las sentencias que se ejecutan en una cláusula `ON KEY`, ya que el valor de las variables de destino no se actualiza normalmente hasta que no se ha salido del campo.

Si alguna de las sentencias modifica el valor de una variable que se está editando, será necesario enviar al formulario el nuevo valor de la variable mediante el uso de la sentencia ’`DISPLAY BY NAME`’.

Desde dentro del propio formulario es posible mover el cursor a un campo determinado usando la sentencia ’`NEXT FIELD`’ o terminar la edición mediante la sentencia `EXIT INPUT`

Ej:

```
form f_clientes
   input codigo from 1,10 to 1,6
   input nombre from 2,10 to 2,30
   input direccion from 3,10 to 3,30
   check facturar at 4,10
   button aceptar at 2,40
   button cancelar at 3,40
end form

function editar_cliente
define sw_aceptado integer
define r record
   codigo integer
   nombre char(20)
   direccion char(20)
   facturar integer
end record
   let sw_aceptado=0
   create form f_clientes
   display form f_clientes
   input by name r.*
   on key(F1)
       call ayuda()
   on button aceptar
       validate field
       let sw_aceptado=1
       exit input
   on button cancelar
       let sw_aceptado=0
```

```
        exit input
    after field codigo
        if r.codigo<100 then
            let r.codigo=0
            display by name r.codigo
            next field r.codigo
        end if
    end input
end function
```

Acceso a Bases de Datos. Sentencias SQL

El objetivo de este manual no es explicar la sintaxis del lenguaje *SQL*, sino ver de qué forma se integra este lenguaje en el 4GL implementado.

Transacciones

PostgreSQL es un gestor de bases de datos transaccional. Permite agrupar operaciones dentro de transacciones y anular todas las actualizaciones realizadas por una transacción si es necesario. Las sentencias para indicar el comienzo y final de una transacción son:

```
BEGIN TRANSACTION
```

Indica el comienzo de una transacción. Todas las actualizaciones que se realicen sobre la base de datos no serán definitivas hasta que se ejecute la sentencia `COMMIT TRANSACTION`

```
COMMIT TRANSACTION
```

Esta sentencia indica que se ha llegado al final de una transacción y que se deben fijar todas las actualizaciones realizadas sobre la base de datos.

```
ABORT TRANSACTION
```

Con esta sentencia se cancela una transacción. Todas las actualizaciones realizadas desde la sentencia `BEGIN TRANSACTION` son desechadas.

Hay que tener cuidado de no anidar transacciones. Es común cometer el error de comenzar una transacción y llamar a una función que no sabemos que también comienza otra transacción. Hasta ahí no hay problema. El problema llega cuando la segunda transacción ejecuta un ’`COMMIT TRANSACTION`’. En ese momento se están aceptando TODAS las actualizaciones realizadas a la base de datos, no sólo las de la última transacción; sin embargo, la transacción inicial aún no ha terminado.

Consultas

La sentencia SELECT de SQL busca en una o varias tablas aquellos registros que cumplan determinadas condiciones.

```
SELECT <lista_campos> [INTO <lista_variables>]
  FROM <lista_tablas>
  [WHERE <condicion>]
  [ORDER BY <lista_campos>]
```

Si se indica la cláusula `INTO`, los valores de los campos devueltos son asignados a las variables que se indiquen. Si la consulta devuelve más de una tupla, se asignarán los valores de los campos de la primera de ellas, y se presenta un aviso en pantalla.

Es posible conocer si una consulta no ha devuelto ninguna tupla. La variable global 'status' tendrá el valor notfound en caso de no encontrar registros que cumplan la condición.

Ej:

```
define r record like clientes.*
prompt "Codigo: " for r.codigo
select * into r.* from clientes
   where codigo=r.codigo
if status=notfound then
   display "No se encuentra el cliente ",r.codigo
else
   display "El cliente ",r.codigo," se llama ",r.nombre
end if
```

Inserción de valores

La sentencia de *SQL* para insertar valores dentro de una tabla es INSERT. En ***hy4GL*** se usa con la misma sintaxis que en *SQL*:

INSERT INTO <tabla> VALUES (<expresiones>)

Ej:

```
input by name r_cliente.*
on button aceptar
   if confirmar()="SI" then
      insert into clientes values (r_cliente.*)
      if status=ok then
         exit input
      else
         call mensaje_error("No se puede insertar el registro")
         exit program
      end if
   end if
end input
```

La inserción de valores puede fallar por diversas causas (que el usuario no tenga permiso para escribir en al tabla, que ya exista un registro con la misma clave primaria, etc). Para conocer si una instrucción INSERT ha conseguido o no su cometido, se puede consultar el valor de la variable global status. Ésta valdrá ok si la inserción ha sido correcta.

Borrado de valores

Para borrar valores de una tabla se usa la sentencia `DELETE` de *SQL*. Su sintaxis es la siguiente:

```
DELETE FROM <tabla> [WHERE <condicion>]
```

Si no se indica ninguna condición se borrarán TODOS los registros de la tabla.

Ej:

```
prompt "¿Desea borrar el albarán "+num_albaran+
   " ahora que está facturado? (S/N)" for char sino
if sino="s" or sino="S" then
   delete from albaranes where numero=num_albaran
end if
```

Cursores

Una consulta a la base de datos puede devolver varias tuplas. PostgreSQL ofrece dos métodos para acceder a los datos devueltos por estas consultas. Por un lado, se pueden obtener del servidor de bases de datos todas las tuplas devueltas por la consulta y, una vez en memoria, tratarlas de manera local. Por otra parte, se puede declarar un *cursor* e ir obteniendo las tuplas una a una para tratarlas. La desventaja de este último método es que PostgreSQL solamente permite hacerlo dentro de una transacción, mientras que el primero no se puede usar si la consulta devuelve gran cantidad de tuplas.

hy4GL ofrece una interfaz homogénea para usar ambos métodos. En ambos será necesario declarar un cursor. Si se opta por el primer método, se incluirá el atributo de *local* en la declaración del cursor.

La sintaxis de la declaración de un cursor es la siguiente:

```
DECLARE <nombre> [LOCAL] CURSOR FOR select...
DECLARE <nombre> [LOCAL] CURSOR FOR expresion
```

Ej:

```
declare c local cursor for
   select * from clientes where ult_compra<"1-1-1998"
```

```
Ej:
let consulta=
   "select provincia,count(*) from clientes group by provincia"
declare c cursor for consulta
```

Un cursor sólo se puede declarar una vez en el módulo fuente.

Antes de poder usar los datos devueltos por la consulta hay que abrir el cursor. Al abrir un cursor se evalúan las expresiones que se hayan incluido en las cláusulas `where` de la consulta y se ejecuta la misma.

La sentencia para abrir un cursor es la siguiente:

```
OPEN <cursor>
```

El propio compilador comprueba si se trata de un cursor declarado como local o no. En caso de que se haya declarado como local, ejecuta la consulta y obtiene del servidor TODAS las tuplas resultantes. Si no se ha declarado como local, simplemente se abre el cursor en el servidor.

Una vez abierto un cursor se pueden obtener las tuplas resultantes de la consulta con la sentencia `FETCH`, cuya sintaxis es la siguiente

```
FETCH [NEXT|PREVIOUS] <cursor> INTO <lista_variables>
```

Si el cursor se ha declarado como local, se asignan directamente a las variables de destino los valores de los campos de la siguiente tupla, que ya está en memoria. Si el cursor no se ha declarado como local, se obtiene en ese momento del servidor la siguiente tupla y sus campos son asignados a las variables.

Es posible obtener la tupla anterior en lugar de la siguiente usando la opción `PREVIOUS`

Si se ha llegado al final de la consulta (o al principio, si se ha usado `PREVIOUS`), la variable global `status` tomará el valor de `notfound`

Al finalizar el trabajo con el cursor es necesario cerrar el mismo. La sentencia `CLOSE` cierra el cursor y libera la memoria usada por el mismo (y por las tuplas resultantes de la consulta, en caso de que el cursor estuviera declarado como local). Su sintaxis es:

```
CLOSE <cursor>
```

```
Ej:
define r record like clientes.*
declare c local cursor for select * from clientes
  where codigo<10
open c
fetch c into r.*
while status!=notfound
  display "Cliente ",r.codigo," - ",r.nombre
  fetch c into r.*
end while
close c
```

EXEC_SQL

hy4GL no soporta la sintaxis completa del lenguaje *SQL*, sin embargo ofrece esta sentencia para permitir al programador ejecutar sentencias *SQL* no soportadas por el lenguaje.

Su sintaxis es la siguiente:

```
EXEC_SQL <expresion> [INTO <lista_variables>]
```

Si la sentencia ejecutada devuelve valores, éstos pueden ser asignados a variables mediante la cláusula `INTO`, y se podrá usar la variable global `status` para conocer si se han obtenido o no valores (`status!=notfound`). Por contra, si se trata de una sentencia que no devuelve valores, la variable global `status` tomará el valor `ok` si se ha ejecutado con éxito

Ej:

```
EXEC_SQL "create table tmp_numeros (i integer);"
if status!=ok then
   display "no se ha podido crear la tabla temporal"
end if
```

Ej:

```
EXEC_SQL "select max(numero) from facturas" INTO num_fac
```

FOREACH

La estructura FOREACH se usa para recorrer un cursor desde el pricipio hasta el final. La sintaxis de esta estructura repetitiva es la siguiente:

```
FOREACH <cursor> INTO <lista_variables>
   sentencias
   ...
   CONTINUE FOREACH
   ...
   EXIT FOREACH
   ...
END FOREACH
```

El cursor debe estar declarado, pero no abierto. El cursor se abrirá automáticamente al entrar en el FOREACH. Por cada tupla del mismo, sus campos son asignados a la lista de variables y se ejecutan las sentencias.

La sentencia `CONTINUE FOREACH` fuerza a que se avance hasta la siguiente tupla. , mientras que la sentencia `EXIT FOREACH` cierra el cursor y sale del bucle.

Cuando el cursor llega hasta el final éste se cierra automáticamente y termina la ejecución del bucle.

Ej:

```
define r record like diario.*
define num_cta like diario.cta
declare c cursor for select * from diario where cta=num_cta
let total=0
foreach c into r.*
   if r.debe_haber="D" then
      let total=total-r.importe
   else
      let total=total+r.importe
   end if
end foreach
display "saldo: ",total
```

BROWSE (ampliación a Informix-4GL)

Esta sentencia es usada para presentar en pantalla el resultado de una consulta a la base de datos. Es necesario definir un cursor con la consulta que se pretende visualizar. El cursor hay que declararlo como `LOCAL`.

```
BROWSE <cursor> INTO <lista_variables>
   [FROM y,x TO yy,xx]
   [TITLE <expresion]
   DISPLAY <expresiones>
   ON KEY (id)
      sentencias
   ON KEY (num)
      sentencias
   ON KEY (cadena)
      sentencias
      ...
      EXIT BROWSE
END BROWSE
```

Al llegar el programa a esta sentencia, se abre automáticamente el cursor, se ejecuta la consulta y se obtienen todas las tuplas resultantes. Para imprimir cada una de ellas, se asignan los distintos campos de la tupla actual a las variables indicadas en la cláusula `INTO`, y se imprimen las expresiones de la cláusula `DISPLAY`.

A partir de este momento, el usuario podrá desplazarse por las tuplas resultantes usando las teclas de movimiento de cursor. Pulsando la tecla ESCape, se abandona la sentencia `BROWSE` y se cierra el cursor.

El programador puede definir una serie de sentencias a ejecutar cuando el usuaro pulse una tecla determinada. Entre estas sentencias puede incluir `EXIT BROWSE`, que abandonará el `BROWSE` y cerrará el cursor.

Ej:

```
define rp record like provincias.*

declare c local cursor for select * from provincias
browse c into rp.*
   title "Matricula Provincia"
   display rp.mat+" "+rp.nombre
   on key (F1)
      call ayuda()
   on key(KEY_ENTER)
      call ver_provincia(rp.*)
   on key(F8)
      exit browse
end browse
```

Generador de Listados

El lenguaje 4GL ofrece una sintaxis sencilla a la vez que potente para generar listados. El generador de listados controla los saltos de página, encabezamientos y los cambios de grupo, lo que facilita enormemente la tarea al programador.

Formato del listado

Los listados se definen fuera de las funciones con la siguiente sintaxis:

```
REPORT <nombre>(parametros)
  declaraciones locales
  OUTPUT
     REPORT TO <fichero>
     REPORT TO PIPE <comando>
     PAGE WIDTH <num>
     PAGE HEIGTH <num>
  FORMAT
     FIRST PAGE HEADER
        sentencias
     PAGE HEADER
        sentencias
     BEFORE GROUP OF <lista_variables>
        sentencias
     ON EVERY ROW
        sentencias
     AFTER GROUP OF <lista_variables>
        sentencias
     PAGE FOOTER
        sentencias
     ON LAST ROW
        sentencias
END REPORT
```

Un listado debe tener un nombre y recibir unos parámetros, que serán la base de la información que se imprima. Al igual que una función, un listado puede usar variables locales para realizar sus cálculos internos. Estas variables habrá que definirlas junto con los parámetros formales.

La definición de un listado tiene dos secciones: la primera, `OUTPUT`, en la que se indica el nombre del fichero de salida o un comando que se ejecutará y al que se va a enviar la salida del listado. En esta sección se especifica también el tamaño (lineas y columnas) de una página, para permitir la detección automática de los saltos de página.

En la segunda sección, `FORMAT`, se indica la apariencia que va a tomar el listado. Consta de varias condiciones (todas ellas opcionales) y de las sentencias que se deben ejecutar al cumplirse cada una de las condiciones.

- `FIRST PAGE HEADER`

  Sentencias que se deben ejecutar al comenzar el listado.
- `PAGE HEADER`

  Sentencias que se deben ejecutar al comenzar cada página
- `BEFORE GROUP OF <var1>,<var2>,...`

  Sentencias que se ejecutarán cuando se vaya a producir un cambio en los valores de las variables indicadas
- `ON EVERY ROW`

  Sentencias que se ejecutan en cada línea
- `AFTER GROUP OF`

  Sentencias que se ejecutan después de producirse el cambio en los valores de las variables.
- `PAGE TRAILER`

  Sentencias que se ejecutan al final de cada página
- `ON LAST ROW`

  Sentencias que se ejecutan al final del listado

Para enviar realmente una línea al listado se usa la sentencia `PRINT`. Esta sentencia deberá estar incluida en alguna/s de las condiciones de la sección `FORMAT` de la definición del listado si se quiere que el listado produzca alguna salida.

Dentro de un listado se pueden usar las variables pageno y lineno, que indican el número de página y el número de línea dentro de la página respectivamente.

Sentencias de control de listados

Para generar un listado se usan las siguientes sentencias:

- `START REPORT <nombre>`

  Esta sentencia indica el comienzo del listado. Al ejecutarla se preparan todas las estructuras internas y se crean tablas temporales para almacenar la información que va a generar el listado.

- `OUTPUT TO REPORT <nombre> (param1, param2, ...)`

  Envía al generador de listados los parámetros indicados.

- `FINISH REPORT <nombre>`

  Da por finalizado el listado. Al ejecutar esta sentencia es cuando se genera realmente el listado a partir de todos los parámetros recibidos mediante sentencias `OUTPUT TO REPORT`

Ej:

```
function listar_alumnos()
define r record like alumnos.*
declare c cursor for selec * from alumnos order by grupo
  start report r1
  foreach c into r.*
     output to report r1(r.grupo,r.nombre)
  end foreach
  finish report r1
end function

report r1(grupo,nombre)
define grupo char(5)
define nombre char(40)
output report to pipe "lp -dlaser"
  page width 78
  page length 64
format
  first page header
     print "LISTADO DE ALUMNOS POR GRUPO   Fecha:"+today()
     print ""
     print "Nombre y apellidos"
     print "--------------------------------------"
```

```
  page header
     print "LISTADO DE ALUMNOS POR GRUPO    Pag:"+pageno
     print ""
     print "Nombre y apellidos"
     print "-------------------------------------"
  before group of grupo
     print
     print "Grupo: ",grupo
     print "--------------------------------"
  on every row
     print nombre
end report
```

Uso del compilador de 4GL.

El programa con el que se invoca al compilador es `hy4gl`. Se trata de un script que comprueba el tipo de cada uno de los archivos que recibe como parámetro y ejecuta el programa necesario para compilarlo.

hy4gl solamente procesa aquellos archivos con extension .4gl. El resto de archivos, así como todos los demás parámetros que reciba son enviados al compilador de C.

Si no se especifica el parámetro `-o` (para indicarle al compilador de C el nombre del ejecutable que debe generar, en lugar de a.out), se generará un ejecutable con el mismo nombre que el primer archivo que se mande como parámetro a hy4gl, y con extensión `.e`

Ej:

- Para generar `alumnos.e` a partir de `alumnos.4gl`:

  ```
  hy4gl alumnos.4gl
  ```

- Para generar `gestion.e` a partir de `menu.4gl`, `profesores.4gl` y `alumnos.4gl`:

  ```
  hy4gl -o gestion.e menu.4gl profesores.4gl alumnos.4gl
  ```

- Para compilar profesores.4gl y obtener el modulo objeto, pero no generar ejecutable:

  ```
  hy4gl -c profesores.4gl -o progesores.o
  ```

Variables de entorno

Las siguientes variables de entorno pueden influir en el comportamiento del compilador:

- HYVERB: Si existe esta variable de entorno, el compilador muestra en pantalla el código generado.
- C_LIBRARY_PATH: Indica la lista de directorios en los que el enlazador va a buscas las librerías
- C_INCLUDE_PATH: Indica la lista de directorios en los que el compilador de C va a buscar los archivos de cabecera.

Las siguientes variables de entorno pueden influir en el comportamiento de las aplicaciones generadas:

- HY4GLHOST y HY4GLPORT: si existen, indican el nombre del host y el puerto TCP/IP del servidor de PostgreSQL.

Indice