CONSTRUCCIÓN DE UN DEPURADOR PORTABLE DE CÓDIGO. FUNCIONES DE LA BIBLIOTECA. EJEMPLO DE USO DEL DEPURADOR.

José L. Arjona Fernández, José M. Prieto Pérez, R. Corchuelo Gil
Dpto. de Lenguajes y Sistemas Informáticos
Facultad de Informática y Estadítica
Universidad de Sevilla
Web: http://www.lsi.us.es

En este cuarto artículo veremos las funciones que componen la biblioteca que constituye el depurador, veremos cual es el significado de cada una de estas funciones y cuando se aconseja su uso. Para finalizar veremos para un pequeño ejemplo el comportamiento del depurador.

Funciones de la biblioteca de depuración.

Cómo se comento en el segundo de los artículos, el depurador de código está constituido por una biblioteca ANSI C. Pues ahora es el momento de comentar las funciones necesarias para hacer que el programa sea autodepurable. Además de generar los stabs, el compilador debe de intercalar en el código C generado, llamadas a las rutinas del depurador. En este apartado detallaremos el significado de cada una de estas funciones que el compilador tiene que insertar.

Los prototipos de las rutinas que suministra la biblioteca, son las siguientes:

```
void debug_init (char *fich_stabs);
void debug_link (char *id,void *dir);
void debug_open (char *nombre_ambito);
void debug_close (void);
void debug_end (void);
void debug_trace (unsigned num_linea, char *fich_fuente);
```

Comentaremos ahora el sentido de cada una de ellas:

- **debug_init:**

Es la rutina de inicialización del depurador, es decir, se lee el fichero de los stabs, y se inicializan todas las variables internas usadas por el mismo. Sin una llamada a esta rutina, nada funcionará en el depurador. Por lo tanto será la primera función que se ejecutará de la librería y del programa fuente. Como parámetro de entrada se le pasa el nombre del fichero donde están almacenados los stabs.

La información de stabs, después de analizar léxica y sintácticamente el fichero de stabs, será almacenada en forma de árboles, organizada en tablas de símbolos (Una tabla de símbolos para cada ámbito). Es decir, tendremos en cada tabla de símbolos una entrada por cada elemento del programa y para cada uno de estos elementos almacenaremos en forma de árbol la información de depuración (stab).

- **debug_link:**

Permite asociar a una variable, constante o parámetro una dirección de memoria. Es necesario que se le asigne una dirección a todos estos objetos para que el depurador conozca su posición en memoria y pueda trabajar con ellos (calcular expresiones, modificar valores, ...). Lo que haríamos sería actualizar en la tabla de símbolos correspondiente un campo que haría referencia a la dirección de ese elemento. Si no se usa correctamente esta función, el depurador accedería a posiciones de memoria que no se corresponde al objeto que estamos estudiando, lo que provocaría efectos impredecibles.

El lugar correcto para colocar las llamada a **debug_link** es, dentro de cada ámbito, antes de la primera llamada a la función **debug_trace** para ese ámbito y despúes de la llamada a **debug_open**[1].

Los parámetros, para poder llamarla son los siguientes: en primer lugar la cadena de carácteres que identifica a ese objeto, seguidamente se le dará la dirección de memoria en la que se encuentra.

- **debug_open:**

Se utiliza para especificar al depurador que se abre un nuevo ámbito. En ella lo que hacemos es apilar información sobre dicho ámbito en la pila de tablas de símbolos interna del depurador. Esta característica nos permitirá soportar el anidamiento estático[2] que poseen algunos lenguajes.

Como parámetro acepta la cadena que identifica al ámbito en el fichero de stabs.

- **debug_close:**

Cierra el ámbito actual, por lo tanto es necesario llamar a esta función cada vez que se sale de un ámbito. En ella desapilamos la información acerca de ese ámbito que apilabamos cuando se llamó a la rutina con **debug_open**.

El parámetro que toma, es la cadena de caracteres que hace referencia al nombre del ámbito del que se está saliendo.

- **debug_end:**

Llamada que se realiza al final del programa y su objetivo es únicamente liberar la memoria ocupada por las estructuras internas del depurador.

- **debug_trace:**

Es la llamada que se encarga de la traza del programa, es decir, se encargará de decidir si se tiene que parar el programa en ejecución para que el usuario pueda introducir algún comando o si

---

[1] El lector podría pensar en hacer las llamadas a **debug_link** en otro lugar, por ejemplo, antes de hacer la llamada a **debug_open**, pero en ese caso tendríamos problemas a la hora de depurar funciones recurrentes. Además si aún no se ha abierto el ámbito, ¿Cómo se encuentra el símbolo?.

[2] El anidamiento estático hace referencia a la propiedad que tienen algunos lenguajes de programación (cómo PASCAL, MODULA..), por la que dentro de un procedimiento ó función se pueden definir otros procedimientos y funciones, siendo visibles únicamente para ese ámbito.

por el contrario tiene que seguir ejecutando el programa. También se encargará de analizar los comandos del usuario y mostrará el resultado de la evaluación de dichos comandos.

Veamos ahora un ejemplo completo y un ejemplo de traza sobre él con el depurador ya construido:

Con este ejemplo pretendemos construir la función de Euler: "*Dado un número entero mayor que cero A, la función de Euler devuelve el número de enteros positivos inferiores a A y que además son primos con A*". Para saber si dos números son primos, nos basaremos en la siguiente propiedad: "*Dos números son primos entre sí cuando su máximo común divisor es la unidad*". Para implementar la función máximo común divisor usaremos el algoritmo de Euclides.

A continuación mostramos el programa que hace lo anterior en PASCAL (Euler.pas):

```
1:  PROGRAM FUNCION_DE_EULER;
2:
3:  VAR numero:INTEGER;
4:
5:  FUNCTION Mcd(p,q:INTEGER):INTEGER;
6:  VAR resto:INTEGER;
7:  BEGIN
8:    REPEAT
9:      BEGIN
10:       resto:= p MOD q;
11:       p:=q;
12:       q:=resto;
13:     END;
14:   UNTIL resto=0;
15:   mcd:=p;
16: END;
17:
18: FUNCTION Euler (x:INTEGER):INTEGER;
19: VAR i,contador:INTEGER;
20: BEGIN
21:   contador:=0;
22:   FOR i:=1 TO x-1 DO
23:     IF Mcd(x,i)=1 THEN
24:       contador:=contador+1;
25:   Euler:=contador;
26: END;
27:
28: BEGIN
29:   WRITELN('NUMERO - FUNCION DE EULER');
30:   WRITELN;
31:   numero:=81;
32:   WRITELN(numero,' - ',Euler(numero));
33: END.
```

El fichero de stabs que tendría que generar el compilador sería (Euler.stb):

```
 [PASCAL_MAIN]
WRITELN                 "Rv"
numero                  "Vi"
Mcd                     "Ri"
Euler                   "Ri"

[Mcd]
p                       "Pi"
q                       "Pi"
resto                   "Vi"
```

```
[Euler]
x                       "Pi"
i                       "Vi"
contador                "Vi"
```

El fichero de código C, con las llamadas a la biblioteca del depurador insertadas sería (Euler.c):

```
#include <stdio.h>
#include "debug.h"

int numero;

void PASCAL_MAIN(void);
int Mcd(int,int);
int Euler (int);

void main (void)
{
    debug_init("euler.stb");
    PASCAL_MAIN();
    debug_end();
    return;
}

void PASCAL_MAIN(void)
{
   debug_open("PASCAL_MAIN");
   debug_link("numero", &numero);

   debug_trace(29,"euler.pas");
   printf(" NUMERO - FUNCION DE EULER");
   printf("\n");
   debug_trace(30,"euler.pas");
   printf("\n");
   debug_trace(31,"euler.pas");
   numero=81;
   debug_trace(32,"euler.pas");
   printf("%d  -  %d",numero,Euler(numero));
   printf("\n");
   debug_trace(33,"euler.pas");
   debug_close();
   return;
}

int Mcd(int p,int q)
{
   int resto;

   debug_open("Mcd");
   debug_link("resto",&resto);
   debug_link("p",&p);
   debug_link("q",&q);
   do {
   debug_trace(8,"euler.pas");
      debug_trace(10,"euler.pas");
      resto= p % q;
      debug_trace(11,"euler.pas");
      p = q;
      debug_trace(12,"euler.pas");
      q = resto;
   } while (resto!=0);
```

```
  debug_trace(15,"euler.pas");
  debug_close();
  return p;
}

int Euler (int x)
{
  int i,contador;

  debug_open("Euler");
  debug_link("i",&i);
  debug_link("contador",&contador);
  debug_link("x",&x);

  debug_trace(21,"euler.pas");
  contador = 0;
  for (i=1;i<=(x-1);i++)
    {
  debug_trace(22,"euler.pas");
    debug_trace(23,"euler.pas");
    if (Mcd(x,i)==1)
      {
       debug_trace(24,"euler.pas");
       contador = contador + 1;
      }
    }
  debug_trace(25,"euler.pas");
  debug_close();
  return contador;
}
```

Vamos a usar el depurador:

```
Depurador simbólico de código portable. Versión 1.1
Copyright(C) 1998. José Manuel Prieto Pérez y José Luis Arjona Fernández
Tutor: Rafael Corchuelo Gil.
Este programa viene sin ningún tipo de GARANTIA;
para obtener más detalles teclea 'show w'.
Es gratuito y se te permite y agradece que lo
redistribuya bajo ciertas condiciones;
teclea 'show c' para más detalles.
```

* Ejecutemos en primer lugar algunas instrucciones:

```
29: WRITELN(' NUMERO - FUNCION DE EULER');
(euler.pas:29)? next
  NUMERO - FUNCION DE EULER
30: WRITELN;
(euler.pas:30)? next
31: numero:=81;
(euler.pas:31)? next
32: WRITELN(numero,' - ',Euler(numero));
```

* Vamos a cambiar el valor del número al que se le calcula la función de Euler:

```
(euler.pas:32)? watch numero
(Entero) 81
(euler.pas:32)? set numero = numero - 71
(euler.pas:32)? watch numero
(Entero) 10
```

* ¿Por qué línea íbamos?

```
(euler.pas:32)? line
32: WRITELN(numero,' - ',Euler(numero));
```

* ¿Qué es el objeto Euler?

```
(euler.pas:32)? whatis Euler
```

```
Rutina que devuelve Entero
* Entremos en la función Euler
(euler.pas:32)? next
21: contador:=0;
* Vamos a obtener un poco más de información:
(euler.pas:21)? whatis i
Variable de tipo Entero
(euler.pas:21)? whatis x
Parametro de tipo Entero
* Nos pararemos en la última iteración del bucle FOR existente en el cuerpo de la función
* Euler:
(euler.pas:21)? break #22 i==x-1
(euler.pas:21)? vbreak
Lista Breakpoints [euler.pas].
* linea #22
    1:  (i==(x-1))
(euler.pas:21)? run
22: FOR i:=1 TO x-1 DO
* Veamos que verdaderamente es la última iteración:
(euler.pas:22)? watch i
(Entero) 9
(euler.pas:22)? watch x-1
(Entero) 9
* Entremos en Mcd:
(euler.pas:22)? next
23: IF Mcd(x,i)=1 THEN
(euler.pas:23)? next
8: REPEAT
* Veamos que es lo que devuelve la función Mcd:
(euler.pas:8)? break #15
(euler.pas:8)? run
15: mcd:=p;
(euler.pas:15)? watch p
(Entero) 1
*Por tanto se tiene que ir por la rama verdadera del condicional IF que está en la función
* Euler:
(euler.pas:15)? next
24: contador:=contador+1;
* Exactamente, ahora a por el resultado final!
(euler.pas:24)? run
 10  -  4
```