CONSTRUCCIÓN DE UN DEPURADOR PORTABLE DE CÓDIGO. PROCESO DE GENERACIÓN DE CÓDIGO DEPURABLE. INFORMACIÓN DE DEPURACIÓN

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 segundo artículo empezaremos a dar las nociones necesarias para construir un depurador de código. El depurador será una biblioteca en ANSI C que podrá ser utilizado por cualquier compilador que genere código C. Se verá la necesidad de tener información acerca del programa que se pretende depurar y se definirá el formato a usar para definir ésta información.

Proceso de generación de código depurable.

El proceso de generación de código ejecutable para un compilador que genera código C como código objeto, queda reflejado en la siguiente figura:

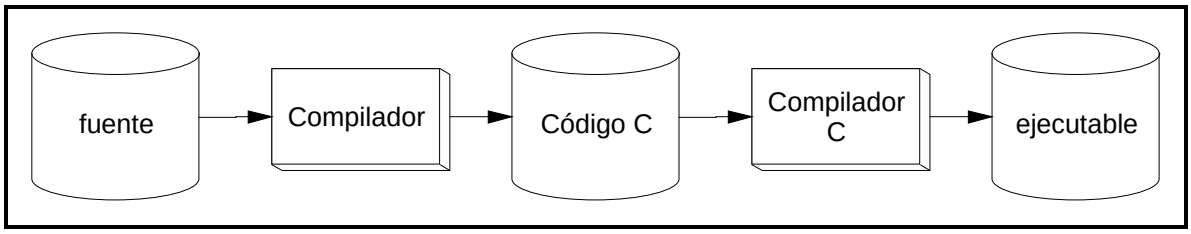


Figura 1. Proceso de generación de código ejecutable.

El esquema anterior queda modificado si queremos generar código depurable. A continuación, en la figura 2., se muestra el proceso de generación de código depurable que debe llevar a cabo un compilador que genere código C intermedio, es decir, nuestro caso.

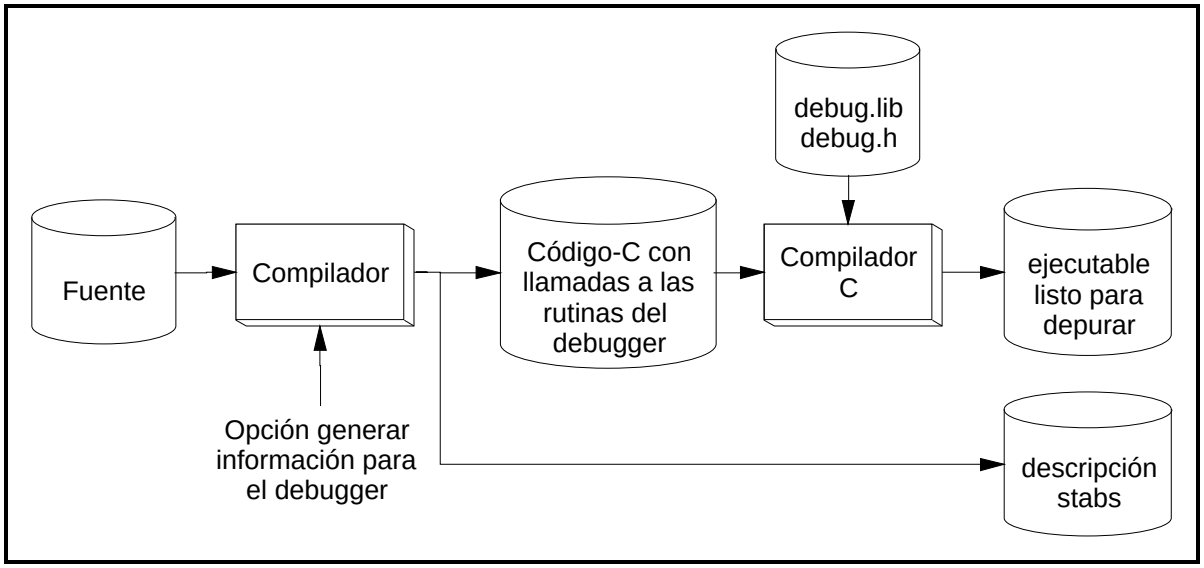


Figura 2. Proceso de generación de código depurable.

Como se puede observar, es necesario habilitar la opción que permite que el compilador genere información para el proceso de depuración; ésta información la sitúa el compilador en un fichero distinto al del código objeto y se usa únicamente en la fase de inicialización del depurador. Además de la información de descripción de stabs, el compilador tiene que insertar en el código C llamadas a la funciones del depurador, estas funciones se verán en artículos posteriores, pero podemos adelantar que permitirán inicializar el depurador, asignar a los distintos objetos la dirección donde se encuentran en memoria, abrir un nuevo ámbito, cerrar un ámbito, trazar el programa y liberar la memoria ocupada por el depurador.

Una vez generado el código C por el compilador, necesitamos construir un ejecutable definitivo con un compilador de C; éste archivo ejecutable se obtiene compilando el código C generado, y enlazándolo con la biblioteca que constituye el depurador. Es decir, el depurador que pretendemos construir será una biblioteca con una serie de funciones, que enlazada con el código C generado por un compilador permitirá obtener programas autodepurables.

Stabs. Necesidad de la información de depuración.

Para ver por qué es necesario disponer de información de depuración vamos a ver lo que haría un determinado depurador de código para poder visualizar el valor de una variable.
Supongamos que estamos depurando el siguiente programa:

```
PROGRAM INCREMENTAR;
      VAR A:INTEGER;
BEGIN
      A := A + 1;
END.
```

Supongamos que el usuario pregunta por el valor de la variable *"A"* antes de que se produzca el incremento. El depurador debería saber que *"A"* es una variable, además si queremos hacer uso de *"A"* en expresiones, o asignarle un determinado valor, deberíamos conocer su tipo, así podría saber si la expresión o asignación es correcta o no. Evitaríamos de esta forma expresiones del tipo:

A < "Del cerdo me gusta hasta los andares"
A = 12.34E-12

Por lo tanto, para que el depurador pueda ejecutar los comandos del usuario, es necesario que tenga cierta información de los archivos fuentes que va a depurar. Atendiendo al ejemplo anterior, de la variable *"A"* nos interesaría saber que se trata de una variable y que su tipo es entero, además debemos conocer la dirección de memoria en la que se encuentra si queremos usar su valor. La información anterior podría venir dada con la siguiente cadena de caracteres (stab): "*Vi*", donde *V* indica que es una variable e *i* que el tipo de la variable es entero.

Podemos generalizar, diciendo que para que el depurador pueda realizar sus funciones típicas, según se ejecuta el programa que queremos depurar, éste debe de conocer al programa, es decir, tiene que disponer de información de todos los objetos a los que el usuario, tanto explícita como implícitamente pueda acceder.

La información de depuración será generada por el compilador y se le pasará al depurador a través de un fichero.

Stabs. Formato general.

Una vez explicado el proceso general de generación de código depurable y visto la necesidad de tener información de depuración, pasaremos a describir el formato de los stabs, es decir, como podemos describir mediante simples cadenas de caracteres los distintos elementos del programa que son de interés para el depurador.

La idea de especificar mediante una simple cadena de caracteres el stab que describe un objeto de nuestro programa, traerá como consecuencia más inmediata, la implementación por parte del depurador de un autómata eficiente que la reconozca. Las cadenas de stabs una vez reconocidas serán transformadas a árboles de sintaxis abstractas. De esta forma evitaremos el tener que a analizar la cadena de stab de un determinado objeto cada vez que el usuario introduzca un comando, ya que se tiene esta información estructurada con los árboles.

Cabe mencionar que las posibilidades que ofrecen los stabs son enormes en cuanto a la potencia que se le puede dar al depurador. Posiblemente, para implementar mejoras futuras en nuestro depurador haya que modificar y/o añadir cadenas de stabs que hagan más fácil dichas mejoras.

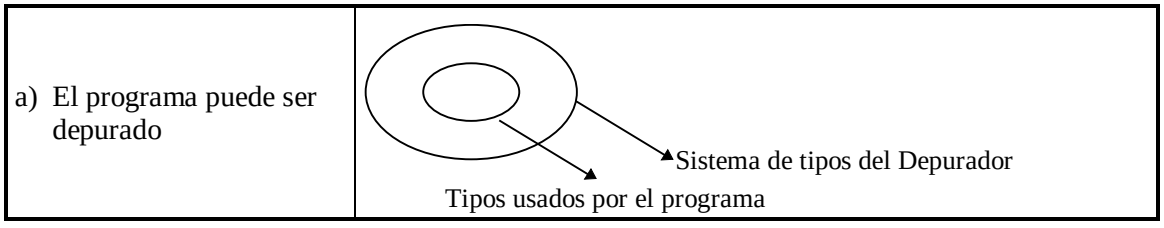
El formato general usado será:

Nombre_objeto *"cadena de stab"*

La *cadena de stab* tiene dos partes diferenciables:

- *Clase de objeto:* Nos dice si el objeto es una variable, una constante, un tipo definido por el usuario, una rutina o un parámetro.

- *Descriptor de tipo:* Se refiere al tipo concreto del objeto, es decir, qué conjunto de valores puede permitir. El sistema de tipos del depurador debe de ser lo más amplio posible para poder depurar una mayor cantidad de programas. Es decir, se debe contemplar los tipos más frecuentes en cualquier lenguaje de programación. De no existir un sistema de tipos relativamente amplio, podríamos encontrarnos con que no podemos depurar algún programa. En la figura 3. podemos ver gráficamente ésta idea:



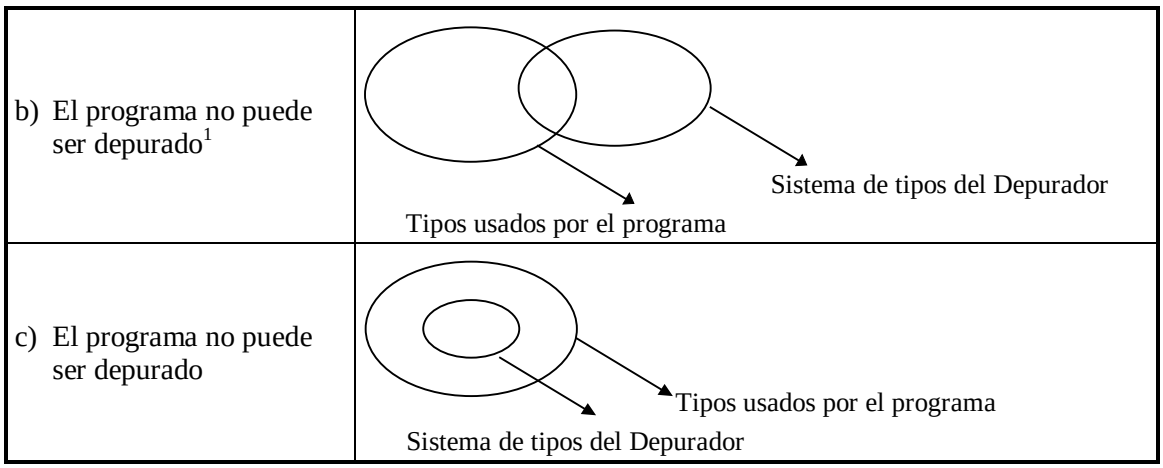


Figura 3. Importancia de un sistema de tipos amplio

La gramática concreta, en formato YACC, que se propone para almacenar la información de depuración es la siguiente[2]:

```
stab : tobjeto tipo
     | tipo
     ;

tobjeto : 'V'     /* Variable */
        | 'C'     /* Constante */
        | 'P'     /* Parámetro */
        | 'U'     /* Tipo definido por usuario */
        | 'R'     /* Rutina */
        ;

tipo : 'i'        /* Entero */
       | 'b'      /* Byte */
       | 'f'      /* Real */
       | 'o'      /* Booleano */
       | 'c'      /* Char */
       | 'v'      /* Void */
       | 's' DIG  /* String, DIG representa la longitud de la cadena */
       | 'a' tipo ';' DIG ',' DIG ';' tipo  /* Array, el primer tipo es el
                                               del indice y el segundo es
                                               el tipo base*/
                                           /* DIG ',' DIG hace referencia
                                               a los límites del array */
       | 'e' '+' lista_valores ';'       /* Enumeracion */
       | 'r' ':' lista_campos_reg ';'    /* Registro */
       | 'u' ':' lista_campos_un ';'          /* Union */
       | 'p' tipo       /* Puntero*/
       | '$' DIG        /* Descomposicion de Tipo */
       ;

lista_valores : ID ',' DIG
```

[1] Cuando decimos que el programa no puede ser depurado, lo que realmente estamos diciendo es que no podremos acceder usando el depurador a los objetos cuyo tipo no se encuentre contemplado en el sistema de tipos del depurador. Pero sí podremos trabajar con el resto de objetos cuyos tipos estén englobados en el sistema de tipos del depurador.

[2] Los identificadores en mayúsculas y los caracteres entre comillas denotan a los tokens de la gramática. Los identificadores en minúscula son los símbolos no terminales, los cuales derivan en otros símbolos no terminales ó en tokens mediante las reglas de producción de la gramática.

```
              | lista valores ':' ID ',' DIG
              ;

lista_campos_reg : campo_reg
                 | lista_campos_reg ':' campo_reg
                 ;

lista_campos_un : campo_un
                | lista_campos_un ':' campo_un
                ;

campo_reg : ID ',' tipo
          | PAD  /* Padding */
          ;

campo_un : ID ',' tipo
         ;
```

De la gramática anterior es necesario comentar varios detalles:

1. Descomposición de tipos.($).

Dependiendo de la complejidad del tipo que se pretenda describir mediante la cadena de stab, ésta puede resultar más o menos larga. Además, se dan casos en los que se declaran variables que son del mismo tipo. Por ejemplo, consideraremos el tipo siguiente:

```
TYPE
        T1 = UNION
                e : INTEGER;
                a : ARRAY [1..12] OF CHAR;
        END;
```

La cadena de stab que se generaría sería:

T1 "Uu:e,i:a,ai;1,12;c;"

Nos encontramos ante dos problemas que pueden hacer ineficiente la utilización de los stabs: en primer lugar, la longitud de las cadenas de stabs podría ser relativamente largas y por tanto difíciles de leer. Y en segundo lugar, la redundancia que se produce al tener siempre que generar la misma cadena de stabs por cada objeto que se declare de ese tipo.

Pensando en estos problemas, el depurador incorpora un mecanismo que nos permite descomponer la cadena de stabs en trozos, que podremos reutilizar en cualquier parte del fichero que contiene los stabs. Este mecanismo se basa en definir símbolos del tipo $n, donde n es un número entero positivo, mayor o igual que 1, que el depurador considere como una especie de macro. Por ejemplo, para el caso anterior, podríamos hacer lo siguiente:

T1 "U$1"
$1 "u:e,i:a,$2;"
$2 "ai;1,12;c"

Con lo que consiguiríamos:

- Por un lado, resulta más intuitiva que en el caso de tener una sola cadena de stabs.
- Por otro lado, ahora por cada variable del tipo T1 sólo tendremos que hacer referencia a $1.

2. Padding.

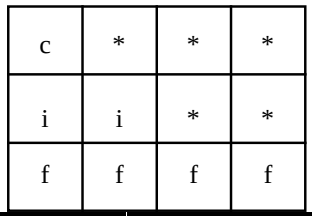
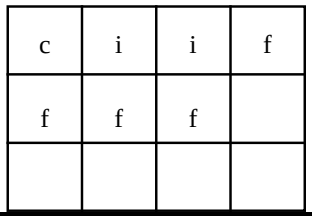
Para los registros, en vez de la descripción de un determinado campo, podría aparecer un determinado número de '*' indicando que el compilador está alineando los datos en memoria[3]. Esto lo consiguen con bytes de relleno o padding. El número de asteriscos nos indica el número de bytes de relleno o padding que está usando el compilador para alinear los datos, ésta cantidad la tendrá que tener en cuenta el depurador a la hora de acceder a los campos del registro, para poder trabajar correctamente con los objetos de tipo registro.

Veamos lo anterior gráficamente, suponiendo lo siguiente:

- Procesador con una palabra de 4 bytes.
- Tamaño de un caracter : 1 byte.
- Tamaño de un entero : 2 bytes.
- Tamaño de un real : 4 bytes.
- Cada cuadrado del siguiente gráfico representa 1 byte.

Veremos lo que ocurre con la siguiente declaración:

```
VAR reg: RECORD
            c:CHAR;
            i:INTEGER;
            f:REAL;
         END;
```

a) Compilador que hace uso de Padding	b) Compilador que no hace uso de Padding
	
Reg "Vr:c,c:***:i,i:**:f,f;"	reg "Vr:c,c:i,i:f,f;"

Fichero de stabs.

Ya se ha visto para qué objetos y de qué forma se definen los stabs de un programa para que el depurador tenga información de ellos. Ahora pretendemos indicar el formato del fichero donde vamos a almacenar toda esta información (los stabs), Además veremos como nos va a aportar una nueva información, esto es información sobre los ámbitos.

Las reglas de ámbito (scoping) indican donde una variable es válida, dónde se crea y dónde se anula. Una variable podrá usarse únicamente cuando está dentro de su ámbito (el ámbito para la que fue definida). Los ámbitos se pueden anidar, de forma que, por ejemplo, una determinada variable definida en un ámbito superior puede quedar oculta por otra una variable de igual nombre definida en otro ámbito inferior.

[3] La alineación de datos en memoria es una técnica de optimización muy utilizada por los compiladores actuales. Al alinear los datos tomando como tamaño base la palabra del computador, se consigue velocidad en los accesos a memoria y transferencias de datos; evitándose también los accesos desalineados.

El depurador tiene que tener en cuenta en todo momento el ámbito en que se encuentra y las variables que son visibles, para ello hará uso de una pila interna en la que iremos apilando y desapilando la información del ámbito en el que hayamos entrado o salido respectivamente. La información de cada ámbito vendrá dada por una tabla de símbolos donde almacenaremos para cada objeto, el árbol correspondiente a su stab y la dirección de memoria en la que se encuentra (para poder acceder, asignarle valores y operar con su contenido).

El formato del fichero podría ser el siguiente:

```
[NOMBRE_AMBITO1]
nombre_objeto1,1   "cadena_stab1,1"   "fichero"  línea  columna
nombre_objeto1,2   "cadena_stab1,2"   "fichero"  línea  columna
...

[NOMBRE_AMBITO2]
nombre_objeto2,1   "cadena_stab2,1"   "fichero"  línea  columna
nombre_objeto2,2   "cadena_stab2,2"   "fichero"  línea  columna
...
...

[NOMBRE_AMBITON]
nombre_objetoN,1   "cadena_stabN,1"   "fichero"  línea  columna
nombre_objetoN,2   "cadena_stabN,2"   "fichero"  línea  columna
...
```

Según el formato anterior, estamos definiendo todos los objetos, ordenados según el ámbito en el que se encuentra definidos. Los nombres de estos objetos son los identificadores usados para cada uno de ellos en el programa fuente. Para los nombres de los ámbitos se pueden utilizar aquellos de las funciones y/o procedimientos que definen dicho ámbito. Si se diera el caso de que el lenguaje que se compila generase bloques de código que definieran ambitos, el compilador debería generar nombres "artificiales" para identificar dichos bloques.

El primero de los ámbitos suele ser un ámbito de nombre especial que identifica a los objetos que se definen globalmente (variables globales, tipos definidos por el usuario globalmente ...).

Tanto "fichero", como "línea" y "columna", son datos que el depurador no utiliza. Solamente se incluyen en el formato del fichero para que sean usados por otras herramientas, como podría ser el caso del Browser.

Veremos a continuación el fichero de stabs que se tendría que generar para un programa que muestra en pantalla la tabla de factoriales.

```
1:  PROGRAM TABLA_FACTORIAL;
2:  VAR n:INTEGER;
3:
4:  FUNCTION Factorial (numero:INTEGER):REAL;
5:  BEGIN
6:        IF numero=0 THEN
7:          Factorial:=1
8:        ELSE
9:          Factorial:=numero*Factorial(numero-1);
10: END;
11:
12: BEGIN;
```

```
13: WRITELN('NUMERO':10,'FACTORIAL':25);
14: WRITELN;
15: FOR n:=1 TO 20 DO
16:       WRITELN(n:8,Factorial(n):27:0);
17: END.
```

El programa saca por pantalla una tabla de valores y los factoriales de esos valores (desde 1 hasta 20). El fichero de stabs sería el siguiente:

```
[PASCAL_MAIN]
WRITELN                 "Rv"
n                       "Vi"
Factorial               "Rf"

[Factorial]
numero                  "Pi"
```

Cabe destacar los siguientes detalles:

- "PASCAL_MAIN" hace referencia al programa principal del fuente en pascal.
- La función "WRITELN" se encuentra en una de las bibliotecas básicas que se suministran con el compilador de Pascal; los diseñadores de compiladores pueden optar por no suministrar la información de depuración ni insertar las llamadas al depurador en su código, lo que provocaría el no poder depurarlas. Suministrar dos versiones, con información de depuración y llamadas al depurador insertadas en su código, y sin información de depuración. Con esta alternativa el usuario podría escoger entre permitir o no permitir la depuración de este tipo de funciones.