Ejercicio: generación diferida de código.

Se propone el siguiente ejercicio para mostrar el uso de memoria dinámica para generar código simplificando la colocación de las acciones semánticas.

Se pide diseñar una calculadora de expresiones aritméticas que traduzca el código de entrada en otro en formato postfijo. Este tipo de problema se resuelve eligiendo con cuidado el punto de inserción en la gramática de las acciones semánticas que imprimen los fragmentos de traducción.

Pero este proceso se puede complicar considerablemente debido a factorizaciones o cuando la gramática crece. Por ello escogemos en este caso la impresión a una salida temporal, lo cuál nos permite reordenar posteriormente los fragmentos traducidos. La salida temporal se puede implementar mediante ficheros temporales, o vectores de tipo carácter (string). Para un procesamiento más efectivo, escogemos vectores de tipo carácter en memoria dinámica.

Para aplicar esta solución emplearemos un vector temporal de caracteres (una variable global) en el que generamos un fragmento de código (el que corresponda a un nodo del árbol sintáctico). Aquí emplearemos la función sprintf, que imprime sobre un vector, o las funciones strcpy y strcat [1].

En el momento que haya que transmitir el fragmento traducido a un nodo superior del árbol sintáctico, deberemos volcar el contenido del vector temporal a memoria dinámica y emplear un puntero a dicho fragmento en memoria dinámica para transmitirlo como atributo. Para ello nos servirá la función gen_cad.

En los nodos superiores los fragmentos recibidos en memoria dinámica se pueden unir empleando de nuevo sprintf, strcpy y strcat para crear mediante gen_cad fragmentos más completos y elaborados.

La impresión del código traducido se realizará cuando tengamos una traducción completa (la de una expresión incluida en una línea).

La ventaja de este esquema es que para convertir el traductor de notación postfija (en la salida) a un traductor a prefija o infija (también en la salida), únicamente hay que cambiar el orden de conectar los fragmentos traducidos:

Para salida infija:

```
expresion:   …
           | operando operador expresion  { strcpy (temp, "") ;
                                            strcat (temp, $1.codigo) ;
                                            strcat (temp, $2.codigo) ;
                                            strcat (temp, $3.codigo) ;
                                            $$.codigo = gen_cad (temp) ; }
```

Para salida postfija:

```
expresion:   …
           | operando operador expresion  { strcpy (temp, "") ;
                                            strcat (temp, $1.codigo) ;
                                            strcat (temp, $3.codigo) ;
                                            strcat (temp, $2.codigo) ;
                                            $$.codigo = gen_cad (temp) ; }
```

Para salida prefija:

```
expresion:   …
           | operando operador expresion  { strcpy (temp, "") ;
                                            strcat (temp, $2.codigo) ;
                                            strcat (temp, $1.codigo) ;
                                            strcat (temp, $3.codigo) ;
                                            $$.codigo = gen_cad (temp) ; }
```

[1] Recordamos las diferencias entre `strcpy(<str1>, <str2>)` y `strcat(<str1>, <str2>)`. La primera copia `<str2>` en `<str1>` borrando lo que pudiese haber previamente en `<str1>`. La segunda concatena `<str1>` con `<str2>`. Por ello usaremos `strcpy` la primera vez para inicializar `temp`, y `strcat` las siguientes veces para añadir cadenas posteriores.

Esto se puede reescribir de forma más breve usando la función sprintf [2].

```
expresion:   …
          | operando operador expresion  { sprintf (temp, "%s %s %s",
                                              $1.codigo,$2.codigo,$3.codigo) ;
                                           $$.codigo = gen_cad (temp) ; }
```

Para salida postfija:

```
expresion:   …
          | operando operador expresion  { sprintf (temp, "%s %s %s",
                                              $1.codigo,$3.codigo,$2.codigo) ;
                                           $$.codigo = gen_cad (temp) ; }
```

Para salida prefija:

```
expresion:   …
          | operando operador expresion  { sprintf (temp, "%s %s %s",
                                              $2.codigo,$1.codigo,$3.codigo) ;
                                           $$.codigo = gen_cad (temp) ; }
```

Esta técnica permite realizar traducciones que no se podrían hacer de forma directa, por ejemplo, las traducciones a notación prefija.

Una desventaja es que requiere una gestión de memoria dinámica mucho más elaborada. Una vez creado un fragmento en memoria dinámica, convendría liberar la que corresponde a sus componentes. Aún así, para expresiones con árboles sintácticos muy profundos, el consumo de memoria dinámica podría aumentar de forma exponencial hasta agotarse al ascender en el árbol.

La solución más lógica sería representar la traducción en una estructura de árbol, en vez de en una estructura lineal. De esta forma cada fragmento generado quedaría "definitivo" en su correspondiente nodo del árbol sintáctico abstracto.

Indicaciones adicionales

Estudiad con detalle los siguientes aspectos:

- Se está usando una estructura de tipo `struct` cómo tipo de pila, en vez del `%union` de otras ocasiones.
- No se usan las directivas `%type` para asignar el atributo (campo del `struct`) que va a devolver cada No Terminal. Por eso en las acciones semánticas se accede a cada atributo de forma literal. Por ejemplo: `$$.codigo = $1.codigo ;`
- Es importante entender que para sintetizar las traducciones en las cadenas es importante usar memoria dinámica. Para ello se usa la función `mi_malloc().`
- También hay que entender por qué es necesario volver a reservar memoria cada vez que se concatenan traducciones parciales. Para ello se usa la función `gen_cad()`.
- Las funciones `mi_malloc()` y `gen_cad()` tienen versiones propias en C (`malloc()` y `strdup()`). Pero se incluyen en versión propia en el código para añadir funcionalidades extra y para que quede más clara su función.

-

[2] `sprintf (<cadena>, <formato>, <argumentos>*)` es equivalente a la función `fprintf (<fichero>, <formato>, <argumentos>*).` En el caso de `sprintf` se imprimen los argumentos en la cadena proporcionada.

Código

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct s_atributos {
      int valor_e ;
      double valor_r ;
      char *codigo ;
} t_atributos ;

char temp [2048] ;

char *gen_cad () ;

#define YYSTYPE t_atributos

%}

%token  ENTERO
%token  REAL

%%

axioma:      '\n'
           | expresion      {
                               printf ("%s\n", $1.codigo) ;
                            }
                  '\n' axioma
           ;


expresion:   operando                       { $$.codigo = $1.codigo ; }
           | operando operador expresion    { strcpy (temp, "") ;
                                              strcat (temp, $1.codigo) ;
                                              strcat (temp, $3.codigo) ;
                                              strcat (temp, $2.codigo) ;
                                              $$.codigo = gen_cad (temp) ; }
           ;


operador:  '+'            { $$.codigo = gen_cad (" + ") ; }
         | '-'            { $$.codigo = gen_cad (" - ") ; }
         | '*'            { $$.codigo = gen_cad (" * ") ; }
         | '/'            { $$.codigo = gen_cad (" / ") ; }
         ;


operando:    REAL         { sprintf (temp, " %lf ", $1.valor_r) ;
                            $$.codigo = gen_cad (temp) ; }
           | ENTERO       { sprintf (temp, " %d ", $1.valor_e) ;
                            $$.codigo = gen_cad (temp) ; }
           ;
```

```
%%
char *mi_malloc (int nbytes)
{
    char *p ;
    static long int nb = 0;
    static int nv = 0 ;

    p = malloc (nbytes) ;
    if (p == NULL) {
         fprintf (stderr, "Error, no queda memoria disponible para %d bytes mas\n",
nbytes) ;
         fprintf (stderr, "Se han reservado %ld bytes en %d llamadas\n", nb, nv) ;
         exit (0) ;
    }

    nb += (long) nbytes ;
    nv++ ;

    return p ;
}


char *gen_cad (char *cadena)     /* copia a un string en memoria */
{                                /* dinamica el argumento        */
     char *p ;
     int l ;

     l = strlen (cadena)+1 ;
     p = (char *) mi_malloc (l) ;
     strcpy (p, cadena) ;
     p [l] = '\0' ;
     return p ;
}


int yyerror (char *mensaje)
{
    fprintf (stderr, "%s\n", mensaje) ;
}


int yylex ()
{
    unsigned char c ;
    int valor_e ;
    double valor_r ;
    unsigned char cadena [256] ;

    do {
         c = getchar () ;
    } while (c == ' ') ;

    if (c >= '0' && c <= '9') {
         ungetc (c, stdin) ;
         scanf ("%s", cadena) ;                      // Es obligatorio usar espacios en
                                                     //  blanco entre los operadores...
         if (strchr (cadena, '.') != NULL) {         //  para que este scanf solo lea
                                                     //  un operador
              sscanf (cadena, "%lf", &valor_r) ;
              yylval.valor_r = valor_r ;
              return (REAL) ;
         } else {
              sscanf (cadena, "%d", &valor_e) ;
              yylval.valor_e = valor_e ;
              return (ENTERO) ;
         }
    }

    return c ;
}


int main ()
{
    yyparse () ;
}
```