UNITS 9, 10 and 11: CODE OPTIMIZATION and GENERATION

Exercise: Deferred code generation.

You are asked to design an arithmetic expression calculator that translates the input sequence (infix format) into a postfix or prefix format. This type of problem is solved by carefully choosing the insertion point in each grammar rule of the semantic actions that print the translated fragments.

But this process can be very complicate due to factored productions or when the grammar grows, and especially for the prefix format.

A suggestion is to use dynamic memory to collect the translated fragments, and to rearrange them at the end of the parsing process.

David Griol Barres, Antonio Berlanga de Jesús, Jesús García Herrero, Juan Manuel Alonso Weber

CODE

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

typedef struct s_atributes {
      int value_i ;
      double value_r ;
      char *code ;
} t_atributes ;

char temp [2048] ;

char *gen_cad () ;

#define YYSTYPE t_atributes

%}

%token  INTEGER
%token  REAL

%%

axiom:      '\n'
          | expression      {
                              printf ("%s\n", $1.code) ;
                            }
                  '\n' axiom
          ;


expression:  operand                        { $$.code = $1.code ; }
          | operand  operator expression    { strcpy (temp, "") ;
                                              strcat (temp, $1.code) ;
                                              strcat (temp, $3.code) ;
                                              strcat (temp, $2.code) ;
                                              $$.code = gen_cad (temp) ; }
          ;


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


operand :    REAL         { sprintf (temp, " %lf ", $1.value_r) ;
                            $$.code = gen_cad (temp) ; }
          | INTEGER       { sprintf (temp, " %d ", $1.value_i) ;
                            $$.code = gen_cad (temp) ; }
          ;
```

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

    p = malloc (nbytes) ;
    if (p == NULL) {
         fprintf (stderr, "Error, no memory left for %d bytes \n", nbytes) ;
         fprintf (stderr, "Up to %ld bytes have been allocated in %d calls\n", nb, nv) ;
         exit (0) ;
    }

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

    return p ;
}


char *gen_cad (char *cad)     /* copies a string into dynamic memory */
{                                /* the same as strdup */
     char *p ;
     int l ;

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


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


int yylex ()
{
    unsigned char c ;
    int value_i ;
    double value_r ;
    unsigned char cad [256] ;

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

    if (c >= '0' && c <= '9') {
         ungetc (c, stdin) ;
         scanf ("%s", cad) ;                        // It is mandatory to use white spaces
                                                    //  between operands and operators
         if (strchr (cad, '.') != NULL) {
              sscanf (cad, "%lf", &value_r) ;
              yylval.value_r = value_r ;
              return (REAL) ;
         } else {
              sscanf (cad, "%d", &value_i) ;
              yylval.value_i = value_i ;
              return (INTEGER) ;
         }
    }

    return c ;
}


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