Manuel J. Petit de Gabriel

«Según Bartjens ...»: solución

El enunciado de dicho problema apareció en el número 146 de *Novática* (julio-agosto 2000, pp. 84-85). Es el programa B de los planteados en el 24º Concurso Internacional de Programación de ACM de 2000

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


/*
 *Toda la solución se basa en construir un vector de operadores y probar con
todos los vectores posibles a ver si al combinarlo con el vector de operandos
(la ristra de números leídos de la entrada) dan el resultado apetecido.
 *
 *Para homogeneizar el tratamiento de los datos y hacer el código completa-
mente regular y sin casos especiales, la concatenación de dos dígitos se trata
también como un operador. Así tenemos 4 operadores distintos: concatena-
ción, suma, resta y producto.
 *
 *Para facilitar la búsqueda el vector de operadores lo tratamos como un
número en base 4. Así la búsqueda exhaustiva es simplemente incrementar un
contador (en base 4) hasta que desbordamos su capacidad.
 */
enum {
    CONCAT= 0,
    PLUS,
    MINUS,
    PRODUCT
};


/*
 * next_op()
 *
 * simplemente un contador en base 4, retorna true mientras queden
combinaciones de operadores posibles. En el momento en que no quedan
combinaciones (el contador desborda) retorna falso
 */
static
bool
next_op(char *ops, int l)
{
    bool carry= true;

    for(int i= 0; (i< l) && carry; i++) {
        if(carry) {
            ops[ i] ++;
        }

        carry= false;

        if(ops[ i] > PRODUCT) {
            ops[ i] = CONCAT;
            carry= true;
        }
    }

    return !carry;
}

/*
 * validate_operation()
 *
 * comprueba que un par de vectores operando/operadores cumple con las
condiciones del enunciado. En concreto, un cero no puede aparecer a la cabeza
de otro número; si aparece un cero tiene que o bien ir precedido por otro dígito
(CONCAT) o bien seguido de un operador distinto a CONCAT
 */
static
bool
validate_operation(char const *str, char const
*ops, int l)
{
    bool leader= true;
    for(int i= 0; i< l-1; i++) {
        if((str[ i] =='0') && (ops[ i] == CONCAT) &&
leader) {
            return false;
        }
        leader= (ops[ i] != CONCAT);
    }

    return true;
}

/*
 * eval()
 *
 *a partir de un vector de 'L' operandos y 'L-1' operadores calcula el valor
de la expresión.
 *El algoritmo de cálculo es aplicar los operadores en orden de precedencia
construyendo nuevos vectores.
 *Este algoritmo es poco académico, lo correcto sería (teniendo en cuenta que
se trata de una gramática de precedencia simple) usar un evaluador con dos
pilas, una de operandos y otra de operadores, haciendo el cálculo en una sola
pasada.
 */
static
int
eval(char const *str, char const *ops, int l)
{
    int retVal= 0;
    int *num_0= new int[ l] ;
    int *ops_0= new int[ l] ;

    for(int i= 0; i< l; i++) {
        num_0[ i] = str[ i] -'0';
    }
    for(int i= 0; i< l; i++) {
        ops_0[ i] = ops[ i] ;
    }

    //
    // Primero se hacen todas las concatenaciones
    //
    int  cur_1= 0;
    int *num_1= new int[ l] ;
    int *ops_1= new int[ l] ;
    memset(num_1, 0, l);
    memset(ops_1, 0, l);

    num_1[ 0] = num_0[ 0] ;
    for(int i= 0; i< l-1; i++) {
        if(ops_0[ i] == CONCAT) {
          num_1[ cur_1] = num_1[ cur_1] *10+num_0[ i+1] ;
        } else {
            ops_1[ cur_1] = ops_0[ i] ;
            cur_1++;
            num_1[ cur_1] = num_0[ i+1] ;
        }
    }
    cur_1++;
    //
    // Y ahora todos los productos
    //
    int  cur_2= 0;
    int *num_2= new int[ l] ;
    int *ops_2= new int[ l] ;
    memset(num_0, 0, l);
    memset(ops_0, 0, l);

    num_2[ 0] = num_1[ 0] ;
```

```
    for(int i= 0; i< cur_1-1; i++) {
        if(ops_1[ i] == PRODUCT) {
            num_2[ cur_2] = num_2[ cur_2] *num_1[ i+1] ;
        } else {
            ops_2[ cur_2] = ops_1[ i] ;
            cur_2++;
            num_2[ cur_2] = num_1[ i+1] ;
        }
    }
    cur_2++;

    //
    // para terminar con las sumas
    //
    retVal= num_2[ 0] ;
    for(int i= 0; i< cur_2-1; i++) {
        if(ops_2[ i] == PLUS) {
            retVal+= num_2[ i+1] ;
        } else {
            retVal-= num_2[ i+1] ;
        }
    }

    delete [] num_0;
    delete [] ops_0;
    delete [] num_1;
    delete [] ops_1;
    delete [] num_2;
    delete [] ops_2;

    return retVal;
}

static
void
dump(char const *str, char const *ops, int l)
{
    static char *op_strings[ 4] = { «%c», «+%c», «-
%c», «*%c» } ;

    /*
     * imprimimos el primer operando y luego iteramos sobre todos los
operadores
     */
    printf(«  %c», str[ 0] );
    for(int i= 0; i< l-1; i++) {
        printf(op_strings[ ops[ i]] , str[ i+1] );
    }

    printf(«=\n»);
}

/*
 * test()
 *
 * La idea es sencilla: iteramos sobre todas las combinaciones de operadores
comprobando si dan el resultado deseado (2000).
 *
 * No empezamos por el vector CONCAT:CONCAT:....:CONCAT porque
el enunciado explícitamente indica que debe haber al menos una suma,
multiplicación o resta. El primer vector a probar es:
CONCAT:CONCAT:....:PLUS
 */
static
bool
test(char const *str)
{
    bool retVal= false;
    int l= strlen(str);

    char *ops= new char[ l-1] ;
    memset(ops, 0, l-1);

    ops[ 0] = PLUS;
    do {
        if(validate_operation(str, ops, l)) {
            if(eval(str, ops, l)== 2000) {
                dump(str, ops, l);
                retVal= true;
            }
        }
    } while(next_op(ops, l-1));

    delete [] ops;

    return retVal;
}

int
main(void)
{
    char buf[ 1024] ;
    int problem_num= 1;

    while(1) {

        if(!fgets(buf, sizeof(buf), stdin)) {
            break;
        }

        if(!strchr(buf, '=' )) {
            /*
             * la entrada no es conforme a las especificaciones así que lo
más razonable es terminar el programa
             */
            break;
        } else {
            /*
             * eliminamos de la entrada el carácter '=' y todo lo
que pueda haber detrás
             */
            strchr(buf, '=' )[ 0] = 0;
        }

        if (strlen(buf)< 2) {
            break;
        }

        printf(«Problem %d\n», problem_num);

        if(!test(buf)) {
            printf(«  IMPOSSIBLE\n»);
        }

        problem_num++;
    }
}
```