Grupo Lunes 8:00-10:00 semanas B

- Práctica 3 -

Autor: Paula Soriano

NIP: 843710

Autor: Carlota Moncasi

NIP: 839841

Ejercicio 1:

calcOrig corresponde a la siguiente gramática:

C -> ϵ | C exp EOL

exp -> FACTOR exp add FACTOR exp sub FACTOR

FACTOR -> term | exp mul FACTOR | exp div FACTOR

term -> number | op exp cp

De la que se deriva:

C -> ϵ | C exp \n

exp -> FACTOR exp + FACTOR exp - FACTOR

FACTOR -> term | exp *FACTOR | exp / FACTOR

term -> [0-9] | (exp)

Al compilar el fichero de Bison, recibimos el siguiente mensaje de error:
warning: 16 reduce/reduce conflicts [-Wconflicts-rr]

Hemos probado la calculadora para las siguientes operaciones aritméticas: 2787+344, 32341*9875, 5453-565, 7585/747, las cuales funcionaban perfectamente. Sin embargo, la expresión **"**3*6+12", muestra *syntax error*, a diferencia de "12+3*6", que lo resuelve correctamente.

Las diferencias que observamos entre calcOrig y calcMejor es que en la ejecución del primero, escribe los números a operar y tras ellos el resultado de la operación, mientras que en el segundo se muestra directamente el resultado. En *calcOrig.l* aparece dentro de la expresión factor el término term, el cual más abajo se encuentra descrito como:

```
term :   NUMBER { printf("number=%d\n", $1); }
         | OP exp CP { $$ = $2; }
         ;
```

En *calcMejor.l*, la descripción de factor es diferente a la anterior. Además, cuenta con la definición de factorsimple. Todo esto se aprecia en el siguiente código :

```
factor:          MUL factorsimple { $$ = $1 * $3; }
                 | factor DIV factorsimple { $$ = $1 / $3; }
                 | factorsimple
                 ;
factorsimple :   OP exp CP { $$ = $2; }
                 | NUMBER
                 ;
```

La expresión "3*6+12" no reconocida antes, con calcMejor sí es reconocida y la operación es calculada correctamente.

Ejercicio 2:

- **Apartado 1.**

  Resumen:
  En este apartado se pide modificar la calculadora para que acepte enteros en decimal o en en otra base b entre 2 y 10.
  Hemos añadido el patrón **"b" {return(B);}** para reconocer la aparición de la letra "b" y creado su token correspondiente en el *ej22.y*, denominado **B**. De la misma manera, hemos modificado *calclist* del siguiente modo:

```
calclist : /* nada */
        | calclist exp EOL { printf("=%d\n", $2); }
        | calclist B EQ NUMBER EOL { b = $4; }
        ;
```

  para reconocer el patrón b=x, siendo x un número entre 2 y 10.
  Hemos desarrollado el siguiente código en flex:

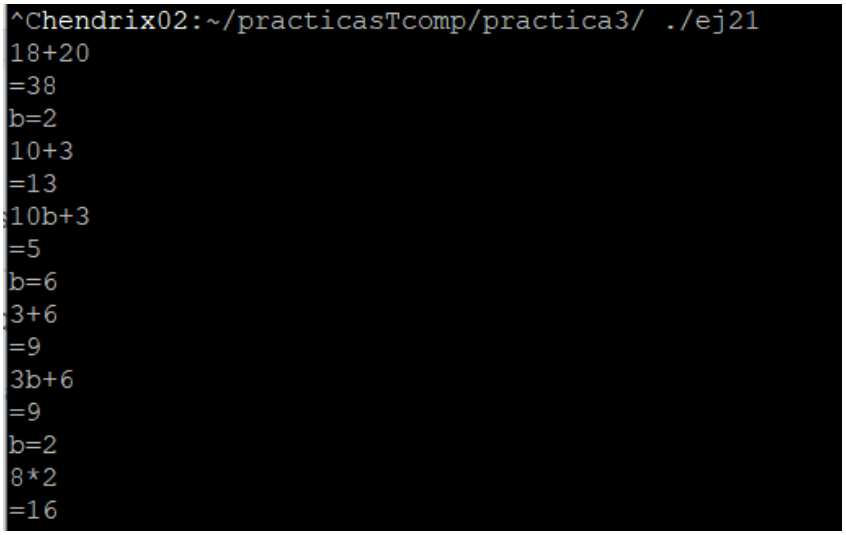
```
[0-9]+"b" {yylval=atoi(yytext);
int result=0;

        int base=b;
        result += (yylval%10);
        yylval/=10;
        while(yylval!=0) {
                result += (yylval%10)*b;
                b=base*b;
                yylval/=10;
        }
        yylval = result;
        return(NUMBER);}
```

  Este código convierte a decimal el número reconocido justo antes de la letra b en la base b, extraída previamente del "b=x".

  Pruebas:
  Para realizar las pruebas a la hora de comprobar su funcionamiento, debemos compilar utilizando el siguiente comando:
  gcc lex.yy.c y.tab.c -lfl -o ej21
  Comprobamos su funcionamiento con distintas bases y operaciones. En todos los casos nos aparecen los resultados esperados, por lo que es correcto su funcionamiento.

Estas son algunas de las pruebas que hemos llevado a cabo:



- **Apartado 2.**

  Resumen:

  Hemos creado los siguientes patrones en el *ej22.l* para reconocer las apariciones de punto y coma y de punto y coma seguido de una b, ya que en el enunciado se pide modificar la calculadora para que todas las líneas de entrada terminen con “;” o bien “;b”:

  ";" {return(PC);}
  ";b" {return(PCB);}

  Así, hemos definido en el *ej22.y* dos tokens denominados **PC** (para el reconocimiento de “;”) y **PCB** (para el reconocimiento de “;b”), y declarado una variable de tipo entero que hemos inicializado a 10 y a la cual hemos denominado b,

  Al igual que en el apartado anterior, hemos añadido el patrón **"b" {return(B);}** para reconocer la aparición de la letra b y creado su token correspondiente en el *ej22.y*, denominado **B**.

Por otra parte, en el *ej22.y*, hemos modificado *calclist* de la siguiente manera:

```
calclist : /* nada */
        | calclist exp PC EOL { printf("=%d\n", $2); }
        | calclist B EQ NUMBER EOL { b = $4; }
        | calclist exp PCB EOL { int op = $2;
             const int MAX=100;
                    int res[MAX];
                    int cont=MAX-1;
                    while(op!=0){
                       res[cont]= op%b;
                       op /= b;
                       cont--;
                    }
                    printf("=");
                    for(int j=cont+1; j<MAX; j++){
                       printf("%d",res[j]);
                    }
                    printf("\n");
                    }
        ;
```

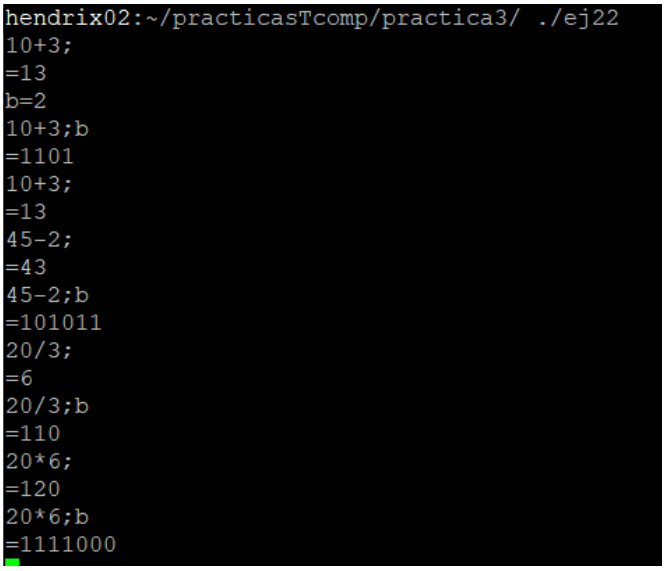
Hemos añadido PC y PCB para que después de la aparición de una expresión sea necesario reconocer un “;” para llevar a cabo la operación, tal y como se especifica en el enunciado. Además, hemos creado el código de la parte inferior para transformar el resultado de la operación a la base b determinada previamente.

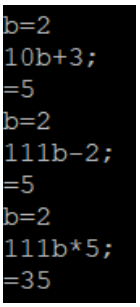
También hemos añadido, al fichero *ej22.l* de flex, la instrucción del apartado 1 anterior, que reconoce patrones como “10b+1” y los transforma a base decimal, pero con “;” al final en este caso:

```
[0-9]+"b" {yylval=atoi(yytext);
        int result=0;
        int base=b;
        result += (yylval%10);
        yylval/=10;
        while(yylval!=0) {
                result += (yylval%10)*b;
                b=base*b;
                yylval/=10;
        }
        yylval = result;
        return(NUMBER);}
```

Pruebas:

Comprobamos su funcionamiento con distintas bases y operaciones. En todos los casos nos aparecen los resultados esperados, por lo que es correcto su funcionamiento.
Estas son algunas de las pruebas que hemos llevado a cabo:





- **Apartado 3.**

Resumen:

Hemos creado en el *ej23.y* los tokens ACUM y EQU, así como los siguientes patrones en el *ej23.l,* para reconocer las apariciones de “acum” y de “:=”

"acum" {return(ACUM);}
":=" {return(EQU);}

En el *ej22.y*, hemos declarado e inicializado a 0 una variable de tipo entero a la que hemos denominado “acum”. Además, hemos modificado *calclist* de la siguiente manera para poder asignar valores a la variable acum, como se ve en la línea inmediatamente superior a la compuesta por el “;”:

calclist : /* nada */
| calclist exp EOL { printf("=%d\n", $2); }
| calclist ACUM EQU exp EOL { acum = $4; }
;

Pruebas:
Comprobamos su funcionamiento con distintas bases y operaciones. En todos los casos nos aparecen los resultados esperados, por lo que es correcto su funcionamiento.

Estas son algunas de las pruebas que hemos llevado a cabo:

```
hendrix02:~/practicasTcomp/practica3/ ./ej23
acum:=5
acum
=5
acum:=3*2
acum
=6
acum+4
=10
acum:=8+acum
acum
=14
acum/7
=2
acum:=acum-2
aum

syntax error
```

Ejercicio 3:

Resumen:

S-> CxS | ϵ

B-> xCy | xC

C-> xBx | z

Para el reconocimiento de las apariciones de “x”, “y” y “z”, hemos creado los siguientes patrones en *ej3.l* : el reconocimiento de “x” devuelve **X** en *ej3.y*, el de “y” devuelve **Y** y “z” devuelve **Z**, siendo **X**, **Y** y **Z** tokens definidos en *ej3.y*.

Para saber cuál es el lenguaje descrito por la gramática G debemos fijarnos también en la gramática descrita por S, B y C. El lenguaje descrito por la gramática S es **(Cx)+**, el cual hemos averiguado tras observar todas las posibles cadenas que se pueden formar; **S -> CxS -> CxCxS -> …** Esto termina cuando S es sustituida por épsilon. En el caso de la gramática descrita por B, encontramos diversas cadenas posibles, algunos ejemplos de estas serían los siguientes:
B-> xCy-> x(xBx+z)(y+ϵ)-> x(x(xBx+z)(y+ϵ)x+z)(y+ϵ)-> … Finalmente, en el caso de la gramática descrita por C, hemos encontrado diversas cadenas que la cumplen:
C-> xBx-> xxC(y+ϵ)x ->..., aumentando x exponencialmente a 2n, mientras la cadena (y+ϵ)x incrementa exponencialmente en n, siendo n un número mayor o igual que cero. Esto termina cuando C es sustituida por una z.

Finalmente, llegamos a la siguiente expresión para representar el lenguaje G:
L(G) = {(x^(2n)z((y+ϵ)x)^(m)x)^n | n, m >= 0}

Pruebas:

Comprobamos su funcionamiento con distintas expresiones. En todos los casos nos aparecen los resultados esperados, por lo que es correcto su funcionamiento.

Estas son algunas de las pruebas que hemos llevado a cabo:

```
hendrix02:~/practicasTcomp/practica3/ ./ej3
zx
xxzxx
xxxxzyxyxx
xxzyxxxxzyxx
xxxxzyxxxxxxxzxyxx
zxx

syntax error
```

La última de ellas muestra por pantalla el mensaje *syntax error* porque hemos introducido una expresión que no pertenece al lenguaje.