3.4.1 Cuestiones resueltas de la Sesión 4

4.1 Prueba las siguientes expresiones para comprobar el funcionamiento de la precedencia. ¿A qué conclusión llegas?

2*3+1 2+3*1 1+3*2 1*3+2

Las expresiones se siguen evaluando de derecha a izquierda sin tener en cuenta las precedencias de los distintos operadores.

4.2 Reforma la gramática sustituyendo el *no terminal operando* por el de *expresion* en aquellas producciones que contengan una operación binaria. Recuerda que habíamos introducido operando para evitar la ambigüedad debida a la doble recursivdad (expresion→expresion+expresion). Prueba de nuevo las expresiones anteriores.

Pasamos de:

```
expresion:    operando                { $$  = $1; }
            | operando '+' expresion  { $$  = $1 + $3; }
            | operando '-' expresion  { $$  = $1 - $3; }
            | operando '*' expresion  { $$  = $1 * $3; }
            | operando '/' expresion  { $$  = $1 / $3; }
            ;
```

a:

```
expresion:    operando                 { $$  = $1; }
            | expresion '+' expresion  { $$  = $1 + $3; }
            | expresion '-' expresion  { $$  = $1 - $3; }
            | expresion '*' expresion  { $$  = $1 * $3; }
            | expresion '/' expresion  { $$  = $1 / $3; }
            ;
```

Con este cambio, funcionarán las precedencias definidas. Lo que conseguimos es que en caso de una ambigüedad el analizador opte por evaluar aquella rama que tenga mayor prioridad en función de los operadores que intervienen.

4.3 Amplía la gramática con las producciones necesarias para incluir variables en el lenguaje. Estas podrán intervenir dentro de las expresiones y estarán identificadas por una secuencia alfanumérica, siendo el primer carácter una letra. No se contempla la definición previa de variables (se entiende que serán de tipo real). La implementación se pide en los últimos apartados.

Es necesario ampliar la gramática para que pueda reconocer variables como un elemento más dentro del lenguaje. Las dos situaciones en las que pueden intervenir son como un término más dentro de una expresión, y en la asignación de un valor, cosa que se deja para el siguiente apartado. Ampliamos la gramática para tratar con variables:

[Nivel Sintáctico]

Axioma → *Expresion* <*intro*> *Resto_expr*
Resto_expr → *Axioma* | <*lambda*>
Expresion → *Termino* | *Expresion* + *Expresion* | *Expresion* – *Expresion* |
Expresion * *Expresion* | *Expresion* / *Expresion*
Termino → *Operando* | + *Operando* | – *Operando*
Operando → *Numero* | *(Expresion)* | ***Variable***

[Nivel Léxico]

Numero → *Digito* | *Digito Numero* | . *Decimal*
Decimal → *Digito* | *Digito Decimal*
Digito → *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*
Variable* → *Letra* | *Letra RestoVar
RestoVar* → *Letra* | *Digito* | *Letra RestoVar* | *Digito RestoVar
Letra* → *A* | *B* | … | *Z* | *a* | *b* | … | *z

Se han asignado las nuevas producciones al nivel más apropiado (léxico o sintáctico). Aprovechamos para introducir un no terminal (*Termino*) más entre *Expresion* y *Operando* para tratar todos los casos que se puedan dar con signos unarios.

4.4 Incluye la operación de asignación de una expresión a la variable.

Podemos crear una nueva rama de tipo expresión en la que el operador sea la asignación:

Expresion → *Termino* | *Expresion* + *Expresion* | *Expresion* – *Expresion* |
Expresion * *Expresion* | *Expresion* / *Expresion* | ***Variable* = *Expresion***

El problema en este caso es que esta formulación da como válidos casos como:
1+2+a2=2 <intro>
5

Parece mejor idea crear una rama distinta en la gramática que trate las asignaciones por separado:

Axioma → *Expresion* <intro> *Resto_expr* | ***Variable* = *Expresion* <intro> *Resto_expr***

4.5 Crea el analizador lexicográfico correspondiente.

Para el analizador lexicográfico tendremos que incluir en las primera sección:

```
. . .
%token VARIABLE
. . .
```

Y añadir unas líneas al código de yylex:

```
. . .
    if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
         ungetc (stdin, c) ;
         scanf ("%s", &yylval) ;
         return VARIABLE ;
    }
. . .
```

En el analizador léxico nos aparece el problema de tener que asignar un valor cuando se detecta el *token* de tipo variable. No está bien definido qué hacer con ese valor y cómo. De ahí que se marque en rojo esa inconsistencia: *yylval* es una variable que está definida con el tipo de la pila de *bison YYSTYPE*, en este caso de tipo double; no es muy

adecuado asignarle una ristra de caracteres. Tampoco sabríamos qué hacer con ella en el analizador sintáctico.

4.6 ¿Qué problema ves a la hora de implementar un intérprete que sea operativo?

A nivel sintáctico no hay ningún problema. El parser reconoce las sentencias que contienen variables. Pero el nivel semántico no está bien resuelto: el principal problema está a la hora de asociar la semántica con las nuevas producciones. ¿Qué acción hay que realizar a la hora de asignar el valor a una variable o de operar con ella?

En ambos casos nos haría falta un mecanismo para almacenar el valor que se le asigne a cada variable. Habitualmente esto se resuelve a través de lo que se denomina Tabla de Símbolos (TS), por ejemplo una tabla hash en la que se van incluyendo los símbolos definidos por el usuario (variables, funciones, etc) y almacenando los valores necesarios.

A falta de este mecanismo, sólo podemos diseñar un analizador sintáctico que acepte asignaciones de valores a variables e imprima el resultado de dicha asignación.
Cuando una variable intervenga en una expresión, poco se podrá hacer, ya que no podemos recuperar su valor.

```
axioma:      expresion '\n'   { printf ("Expresion=%lf\n", $1) ; }  r_expr
           | variable '=' expresion { printf ("Var=%lf\n", $3) ; }  r_expr
           ;
. . .
termino:     operando                                  { $$ = $1; }
           | '+' operando %prec SIGNO_UNARIO           { $$ = $2; }
           | '-' operando %prec SIGNO_UNARIO           { $$ = -$2; }
           ;

operando:    VARIABLE                                { ; } /* ¿que semántica? */
           | '('  expresion  ')'                       { $$ = $2; }
           | NUMERO                                    { $$ = $1; }
           ;
```

3.5 Uso de variables sencillas en la calculadora

En esta nueva versión únicamente se va a añadir el manejo de variables simples. Los identificadores de las variables constarán de una única letra, es decir, tendremos 26 variables representadas por los caracteres *A, B, C, . . ., Z*, siendo indistinto su uso en mayúscula o minúscula. Será necesaria la operación de asignación de valores. Esto implica una ampliación de la gramática.

G5

[Nivel Sintáctico]

Axioma → *Expresion <intro> Resto_expr* | ***Variable = Expresion <intro> Resto_expr***
Resto_expr → *Axioma* | *<lambda>*
Expresion → *Termino* | *Expresion + Expresion* | *Expresion – Expresion* |
*Expresion * Expresion* | *Expresion / Expresion*
Termino → *Operando* | *+ Operando* | *– Operando*
Operando → *Numero* | *(Expresion)* | ***Variable***

[Nivel Léxico]

Variable → ***A*** | ***B*** | ***C*** | … | ***Z*** | ***a*** | ***b*** | ***c*** | … | ***z***
Numero → *Digito* | *Digito Numero* | *. NumeroEntero*
NumeroEntero → *Digito* | *Digito NumeroEntero*
Digito → *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

-

Para manejar las variables necesitamos un espacio de memoria para almacenar sus respectivos valores, cosa que podemos solucionar con un vector de tipo coma flotante. Lo denominaremos *memoria[26]* . El acceso a las diferentes variables se realiza indexando dicho vector. Así, para obtener el valor asociado a *A* tendríamos que acceder a la posición 0 del vector *memoria*, para el de *B* a la posición 1, y así sucesivamente.

El uso de estas variables significa que el analizador lexicográfico debe ser capaz de detectar un nuevo tipo de *token* (*VARIABLE*). Para comunicar este token al analizador sintáctico le hará falta, además de devolver el *token VARIABLE*, indicar de cuál de ellas se trata: lo más sencillo será devolver también el índice que haga referencia a ella.

Si recordamos, al encontrar el *token NUMERO* el analizador léxico devuelve a través de la variable *yylval* el valor asociado al *token* (el valor numérico). Vemos que la idea de representar y transmitir un valor numérico en coma flotante no casa exactamente con la idea de un índice que es un valor entero. Por elegancia vamos a recurrir a una estructura de datos dual. Esta estructura, una *union* en el lenguaje *C*, *bison* la empleará tanto para la variable *yylval* como para la pila del analizador sintáctico.

```
%{                          /* SECCION 1 */
#include <stdio.h>
double memoria [26] ;       /* Se define una zona de memoria para las variables */
%}
%union {                    /* El tipo de la pila (del AP) tiene caracter dual */
      double valor ;        /*  - valor numerico real */
      int indice ;          /*  - indice para identificar una variable */
}                           /* SECCION 2 */
```

Estas definiciones además generan dos tipos en *bison* (<*valor*> e <*indice*>) que nos van a servir para definir correctamente los *token* y *no terminales* que se emplean en la gramática.

Obviamente un *NUMERO* tendrá tipo <*valor*>, mientras que *VARIABLE* usará el tipo <*indice*> que servirá para identificar y acceder a una variable. Hay que prestar atención a los no terminales (*expresion*, *operando*) para los que vemos rápidamente que en todos los casos devuelven valores numéricos, por lo que su tipo también será <*valor*>. En el caso de *axioma*, no nos hace falta definir tipo, ya que no tiene que transmitir nada.

```
%token  <valor>  NUMERO    /* Todos los token tienen un tipo para la pila   */
%token  <indice> VARIABLE
%type   <valor>  expresion /* Se asocia tambien a los No Terminales un tipo */
%type   <valor>  termino operando
```

En la sección correspondiente a la gramática añadiremos las reglas necesarias con sus correspondientes acciones semánticas.

```
axioma:       expresion '\n'              { printf ("Expresion=%lf\n", $1) ; }
                        r_expr
            | VARIABLE '=' expresion '\n' { memoria [$1] = $3;
                                            printf ("%c=%lf\n", $1+'A', $3);
                                          }
                        r_expr
            ;

termino:      operando                          { $$ = $1; }
            | '+' operando %prec SIGNO_UNARIO   { $$ = $2; }
            | '-' operando %prec SIGNO_UNARIO   { $$ = -$2; }
            ;

operando:     VARIABLE                  { $$ = memoria [$1]; }
            | NUMERO                    { $$ = $1; }
            | '(' expresion ')'         { $$ = $2; }
            ;
```

Puesto que hemos definido *VARIABLE* como tipo *indice* usaremos su valor en *$1* como índice en el vector de variables. Hay que prestar atención al hecho de que mientras *$1* hace referencia al tipo de *VARIABLE*, *$$* lo será del mismo tipo que *operando* (<*valor*>).

Por último, nos queda por completar el analizador léxico para permitir la identificación de las variables. Cuando se detecta que el carácter leido está en el rango [‘A’. . .’Z’] o [‘a’. . .’z’] se convierte en un índice en el rango [0. . .26] mediante la resta de los códigos ASCII. Al definir la estructura de *yylval* como una union, debemos acceder al campo correspondiente como si fuera un registro: *yylval.indice*. De igual forma, para asignar el valor de un número deberemos emplear *yylval.valor*.

```
int yylex ()
{
. . .
    if (c == '.' || c >= '0' && c <= '9') {
         ungetc (c, stdin) ;
         scanf ("%lf", &yylval.valor) ;
         return NUMERO ;
    }

    if (c >= 'A' && c <= 'Z') {
         yylval.indice = c - 'A' ;  /* resta a c el valor ascii de A */
         return VARIABLE ;
    }

    if (c >= 'a' && c <= 'z') {
         yylval.indice = c - 'a' ;  /* resta a c el valor ascii de a */
         return VARIABLE ;
    }

 . . .
}
```

-
-
-

CUESTIONES ABIERTAS:

5.1 Amplía la calculadora para que admita las variables a, b, …, z como variables distintas a A, B, … Z.

5.2 Crea el analizador léxico adecuado para la calculadora obtenida en el punto anterior con la herramienta *flex*.

calc5.y

```
%{                      /* SECCION 1 */
#include <stdio.h>
double memoria [26] ;   /* Se define una zona de memoria para las variables */
%}
%union {                /* El tipo de la pila (del AP) tiene caracter dual */
      double valor ;    /*  - valor numerico real */
      int indice ;      /*  - indice para identificar una variable */
}                       /* SECCION 2 */
%token  <valor>  NUMERO    /* Todos los token tienen un tipo para la pila   */
%token  <indice> VARIABLE
%type   <valor>  expresion /* Se asocia tambien a los No Terminales un tipo */
%type   <valor>  termino operando
%right  '='             /*  es la ultima operacion que se debe realizar */
%left   '+' '-'         /*  menor orden de precedencia */
%left   '*' '/'         /*  orden de precedencia intermedio */
%left   SIGNO_UNARIO    /*  mayor orden de precedencia */
%%
                        /* SECCION 3: Gramatica - Semantico  */
axioma:       expresion '\n'              { printf ("Expresion=%lf\n", $1) ; }
                      r_expr
            | VARIABLE '=' expresion '\n' { memoria [$1] = $3;
                                            printf ("%c=%lf\n", $1+'A', $3);
                                          }
                      r_expr
            ;

r_expr:                     /* lambda */
            | axioma
            ;

expresion:    termino                     { $$ = $1; }
            | expresion '+' expresion     { $$ = $1 + $3;  }
            | expresion '-' expresion     { $$ = $1 - $3;  }
            | expresion '*' expresion     { $$ = $1 * $3;  }
            | expresion '/' expresion     { $$ = $1 / $3;  }
            ;

termino:      operando                          { $$ = $1; }
            | '+' operando %prec SIGNO_UNARIO   { $$ = $2; }
            | '-' operando %prec SIGNO_UNARIO   { $$ = -$2; }
            ;

operando:     VARIABLE                  { $$ = memoria [$1]; }
            | NUMERO                    { $$ = $1; }
            | '(' expresion ')'         { $$ = $2; }
            ;
%%
```

***calc5.y* (continuación)**

```
                        /* SECCION 4  Codigo en C */
int n_linea = 1 ;

int yyerror (mensaje)
char *mensaje ;
{
    fprintf (stderr, "%s en la linea %d\n", mensaje, n_linea) ;
}

int yylex ()
{
    unsigned char c ;

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

    if (c == '.' || (c >= '0' && c <= '9')) {
         ungetc (c, stdin) ;
         scanf ("%lf", &yylval.valor) ;
         return NUMERO ;
    }

    if (c >= 'A' && c <= 'Z') {
         yylval.indice = c - 'A' ;  /* resta a c el valor ascii de A */
         return VARIABLE ;
    }

    if (c >= 'a' && c <= 'z') {
         yylval.indice = c - 'a' ;  /* resta a c el valor ascii de a */
         return VARIABLE ;
    }

    if (c == '\n')
          n_linea++ ;
    return c ;
}

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