Grupo Miércoles 10:00-12:00 semanas B

—Práctica 3—

Autor: Athanasios Usero

NIP: 839543

NOTA:

Se va a emplear, a la hora de trabajar con expresiones regulares, la notación que se utiliza en *Flex,* con el fin de hacer más fácil la posterior lectura de los ficheros .l .

Ejercicio 1:

1. Resumen

El fichero *calcOrig.y* trata de construir la gramática para definir una calculadora sencilla con las operaciones de suma, resta, división y multiplicación y sus composiciones. Con este fin, los tokens declarados se asociarán al reconocimiento de paréntesis, números u operadores (además del final de línea, que indica el fin de una operación). Igualmente, se definen los siguientes símbolos no terminales: calclist, el símbolo de inicio, que reconoce 0 o más expresiones con finalización; exp que construye expresiones de mínima prioridad (sumas y restas) o factores si estas no existen; factor, que construye operaciones de multiplicación o división entre factores y expresiones, o términos si no existen estas; y term, que reconoce términos de mayor prioridad (números u operaciones entre paréntesis).

El fichero *calcMejor.y* tiene el mismo objetivo que el anterior, pero es una versión mejorada, puesto que hay ciertos casos para los que la implementación del primer fichero dará fallos. Esto tiene relación con casos de ambigüedad a la hora de derivar, pues se observa que existe una especie de puente bidireccional entre el símbolo no terminal *exp* y *factor.* Un factor (en caso de no ser tan solo un término), será la multiplicación/división de una expresión y otro factor, mientras que una expresión (en caso de no ser únicamente un factor), es una suma/resta de una expresión y un factor. Pero, desde esta perspectiva, para el caso de una multiplicación seguida de una suma, por ejemplo, se podrían trazar estos caminos:

$$\exp \rightarrow \exp ADD\ factor \rightarrow factor\ ADD\ factor \rightarrow$$
$$\exp MUL\ factor\ ADD\ factor \rightarrow factor\ MUL\ factor\ ADD\ factor \rightarrow$$
$$term \exp MUL\ factor\ ADD\ factor \rightarrow$$
$$NUMBER\ \ MUL\ NUMBER\ ADD\ NUMBER$$

(primero se suma y luego se multiplica)

$$\exp \rightarrow \exp MUL\ factor \rightarrow factor\ MUL\ factor$$
$$\rightarrow factor\ MUL \exp\ SUM\ factor$$
$$\rightarrow factor\ MUL\ factor\ SUM\ factor \rightarrow^{*}\ NUMBER\ MUL\ NUMBER\ ADD\ NUMBER$$

(primero se multiplica y luego se suma)

Para resolver este problema de biyección, en el que una multiplicación sigue la misma estructura que una suma, la calculadora mejorada ofrece una nueva perspectiva, encapsulando cada operación de aquella con menor prioridad. Consecuentemente las reglas de producción correspondientes a la multiplicación/división constituyen la operación entre un factor y un factorsimple (número o expresión entre paréntesis), de modo que una expresión es la operación entre varios factores, los cuales son la operación de menor prioridad de varios números o expresiones inmediatas (entre paréntesis).

2. Pruebas

Para el siguiente fichero de entrada:

ENTRADA:

```
7 + 5¶
8 + 2 * 5¶
2 * 8 /  4 ¶
(2 * 8) / 4¶
2 * (8 / 4) ¶
(2 + 2 * 2) + (2 + 2 * 2) ¶
2 / 2 * 1¶
(2 + 2 * 2) * (2 + 2 * 2) ¶
2 * (5 + 8) ¶
(2 * 5) + 8¶
2 * 5 +  8¶
(2 + 2 * 2) * (2 + 2 * 2) + 2¶
2 / 4 + 8 * 3¶
```

Todo funcionará correctamente, excepto las tres últimas operaciones, dado que en un punto hay un producto/división precediendo a una suma/resta, lo que dará en un caso de ambigüedad.

SALIDA: (Para calcMejor toda operación funciona correctamente)

```
number=7
number=5
=12
number=8
number=2
number=5
=18
number=2
number=8
number=4
=4
number=2
number=8
number=4
=4
number=2
number=8
number=4
=4
number=2
number=2
number=2
number=2
number=2
number=2
=12
number=2
number=2
number=1
=1
```

```
number=2
number=2
number=2
number=2
number=2
number=2
=36
number=2
number=5
number=8
=26
number=2
number=5
number=8
=18
number=2
number=5
number=8

syntax error
```

Ejercicio 2.1:

1. Resumen

Para modificar la calculadora y que está también acepte enteros en cualquier base b entre 2 y 10, se trabajará sobre todo en el programa de flex, pues es el analizador léxico el que tratará números en base b.

En primer lugar, en fichero que trabaja con Bison, definiremos dos nuevos tokens, pues con nuestra modificación aceptamos dos nuevos símbolos: IG, que se asocia a la operación de igualdad; y BASE, que se asocia al indicador de la base con la que trabajar. Así, tan solo se añade una nueva regla de producción al símbolo no termina calclist:

$$calclist \rightarrow calclist\ BASE\ IG\ NUMBER\ EOL$$

Cuando el analizador identifique esta regla gramatical, asignará a una variable entera *base (*almacena el valor de la base con la que se pueden codificar números y se exporta al fichero flex) el valor NUMBER, que no es más que el número en las cadenas de tipo "b=7\n".

En el fichero con el que trabajará flex se crean tres nuevos patrones:

$$r_1 = " = "$$

$$r_2 = b$$

$$r_3 = [0-9]^+ b$$

r1 y r2 devuelven un token de tipo IG y BASE, respectivamente. Por otra parte, r3 reconoce cualquier número que hayamos codificado en una base b durante una operación. Por ello, ante este patrón se reseteará a 0 una variable entera *conv* que servirá para la conversión progresiva del número en una base *base* a decimal. De este modo, como sabemos que el *yytext* es una cadena de caracteres, iteramos en bucle hasta llegar a la penúltima componente (pues la última corresponde a la *b)* y se suma a *conv* el valor del dígito en base k *base* ponderado por su posición.

2. Pruebas

ENTRADA:

```
b=2
10 + 3
10b + 3
b=4
3b + 30b
3 + 12
b=10
10b / 2
10 / 2
b=7
11143b + 12443
11143 + 12443
```

SALIDA:

```
=13
=5
=15
=15
=5
=5
=15267
=23586
```

Ejercicio 2.2:

1. Resumen

Para la modificación de la calculadora que impone un delimitador al final de una expresión y permite expresar en una base entre 2 a 10 el resultado de dicha operación, se trabajará fundamentalmente con bison, dado que más bien hay que realizar la conversión de un valor que es un patrón gramatical.

En el fichero que trabaja con flex, además de las reglas r1 y r2 de la anterior sección, se añaden los patrones:

$$r_1 = ";"$$

$$r_2 = ";"b$$

Ambos patrones representan la manera en la que se expresa un resultado. Por ello, a estos se les asocia la acción de devolver los tokens PC y LC, respectivamente. Estos son añadidos a los ya existentes porque permiten modificar las reglas de producción para calclist:

$$calclist \rightarrow calclist \exp\ PC\ EOL$$

$$calclist \rightarrow calclist \exp\ LC\ EOL$$

La primera regla de producción representa líneas con operaciones codificadas en decimal, por lo que a estas se les da el valor de *exp.* La segunda regla de producción viene asociada a las líneas con operaciones para ser expresadas en una base *base.* De manera análoga a la cuestión anterior, se almacena en una variable *aux* el valor de *exp* y se inicializa *conv* con el primer paso del algoritmo de división para pasar de decimal a base (se procede a dividir sucesivamente el número en base decimal entre la nueva base hasta que sea 0 y el resto de esa división queda almacenado en conv y ponderado por su posición). La algoritmo se aplica sobre el valor absoluto del resultado, después utilizando el valor de un entero (simulador de booleano), se reasigna al resultado en base el signo correspondiente.

2. Pruebas

ENTRADA:

```
b=2
10 + 3;
10 + 3;b
b=4
3 + 30;b
3 + 12;
b=10
10 / 2;b
10 / 2;
b=7
11143 + 12443;
11143 + 12443;b
b=5
1234 * 234;b
5 - 10;b
b=8
2500 / 5;b
```

SALIDA:

```
=13
=1101
=201
=15
=5
=5
=23586
=125523
=33220011
```

Ejercicio 2.3:

1. Resumen

La resolución de este ejercicio se fundamenta en que el acumulador es una variable escrita sobre el papel y que solo puede aparece en un proceso de asignación (se trata como cadena de texto) o de referencia (se trata como valor numérico).

Por una parte, se ha modificado el fichero de la calculadora que trabaja con bison añadiendo un nuevo token AS, que corresponde a una asignación de la variable acumuladora; y además declarando una variable de tipo entero *ac* en la que se almacenará efectivamente este valor y con la que también trabajará el fichero flex. Así, se introduce solo una regla de producción al símbolo inicial calclist:

$$calclist \rightarrow calclist\ AS \exp\ EOL$$

Correspondiente a una línea en la que se le asigna a *ac* un nuevo valor, de ahí que se le asigna a dicha variable el valor de *exp.*

Por otra parte, en el fichero de flex se han añadido dos nuevos patrones sintácticos:

$$r_1 = acum\text{" := "}$$

$$r_2 = acum$$

r1 está asociada a una instrucción de asignación, por lo que se procederá devolviendo el token AS. r2 devolverá un token NUMBER que toma como valor aquel que tenga la variable *ac,* pues cuando el fichero de entrada es correcto sintáctica y semánticamente, la cadena “acum” solo se emplea en operaciones matemáticas.

2. Pruebas

ENTRADA:

```
acum:= 5
acum
acum:= 3*2
acum
acum+4
acum:= 8 + acum
acum:= acum / acum
acum
acum:= acum - acum
acum
acum:= acum - 5 * 8 + 5
acum / 5
(acum + 5) / 5
```

SALIDA:

```
=5
=6
=10
=1
=0
=-7
=-6
```

Ejercicio 3:

1. Resumen

La implementación de este ejercicio se basa en la conversión sintáctica de la gramática a la sintaxis de bison y flex. El programa tomará un fichero con una línea si espacios en blanco y comprobará si la cadena en esa línea pertenece a la gramática.

En el fichero bison se han declarado cuatro tokens Z, X, Y y EOL correspondientes a los símbolos no terminales del alfabeto Σ y al salto de línea. De este modo, cada vez que en un flujo de entrada flex reconozca los símbolos z, x o y, devolverá el token correspondiente.

Se han definido también los tres símbolos no terminales de Γ S, B C y el símbolo no terminal inicial entrada (correspondiente a la entrada de una cadena con final de línea), con sus respectivas reglas de producción:

$$inicio \rightarrow \epsilon \mid s\ EOL$$

$$s \rightarrow c\ X\ s \mid \epsilon$$

$$b \rightarrow X\ c\ Y \mid X\ c$$

$$\text{c} \rightarrow \text{X b X} \mid \text{Z}$$

No hay ninguna acción asociada, pues el programa solo escribe en caso de entrada errónea.

Por otra parte, para obtener el lenguaje reconocido por la gramática, se ha procedido primero ha simplificar la gramática inicial sustituyendo y eliminando el símbolo no terminal C, obtenido la siguiente gramática equivalente:

$$\text{S} \rightarrow \text{xBxxS} \mid \text{zxS} \mid \epsilon$$
$$B \rightarrow xxBxy \mid xxBx \mid xzy \mid xz$$

Se puede observar que las reglas de producción para el caso del símbolo no terminal inicial describen el siguiente lenguaje:

$$L_S = (L_1 + L_2)^*, donde\ L_1 = \{xL_Bxx\}\ y\ L_2 = \{zx\}$$

Vamos a derivar sucesivamente el patrón gramatical que describe el lenguaje L1, y observamos una cierta semejanza:

$$\text{xBxx} \rightarrow \text{xxxBxyxx} \rightarrow \text{xxxxxBxxyxx} \rightarrow \text{xxxxxxxBxyxxyxx} \rightarrow \text{xxxxxxxxzyxyxxxyxx}$$

$$xBxxxx \rightarrow xxxBxxx \rightarrow xxxxxBxyxxx \rightarrow xxxxxxxBxxyxxx \rightarrow xxxxxxxxzyxxyxxx$$

Es apreciable que toda cadena comienza con un número par n + 2 de *x seguidos de una z* y una y (o no). El resto de la cadena es menos trivial, pero se puede deducir por la simetría de las reglas de producción de B que aparecerán la mitad de símbolos *x (n/2)* más 2 y un número de símbolos *y* comprendidos entre 0 y n/2. Además, esta segunda subcadena comienza con el símbolo *x,* no puede haber dos símbolos *y* seguidos y terminará con los dos símbolos *x* presentes en la cadena inicial. Formalmente, L1 describe el lenguaje:

$$L_1 = \{x^{2(n+1)} z(y+\epsilon)(x\,(y+\epsilon))^n xx \mid n \geq 0\}$$

Teniéndose pues que el lenguaje descrito por la gramática es el siguiente:

$$L(G) = L_S = (L_1 + L_2)^*$$

2. Pruebas

Se han obtenido cadenas correctas y erróneas tanto a partir de la gramática como del lenguaje obtenido. Se muestran las cadenas de prueba junto a los resultados.

ENTRADA/SALIDA:

```
// Entrada correcta

Xxxxxxxxzyxxyxyxxxxxxxxxxzyxxyxyxx
```