Grupo Jueves 12:00 – 14:00 semanas A

- Práctica 3 -

Autor: Inés Román Gracia

NIP: 820731

Ejercicio 1

El archivo *calcOrig.y* corresponde a la gramática de una calculadora que transforma expresiones aritméticas de enteros positivos en un solo entero positivo (el resultado).

Al compilar aparece un *warning* con el mensaje “16 reduce/reduce conflicts”, ya que hay algunas expresiones que pueden transformarse de más de una forma y dar lugar a errores en el resultado final.

Estos son algunos errores de ambigüedad que he podido comprobar:

- La calculadora da resultados erróneos al realizar divisiones seguidas de multiplicaciones sin paréntesis. Esto se debe a que no hay un orden de prioridad entre estas dos operaciones. Por ejemplo, “9/3*3” daría como resultado “1” y no “3” como sería de esperar. Esto se debe a que *factor*, que es la expresión de la derecha, puede ser sustituido por otra operación de división o multiplicación.

  Este problema se ve solucionado en la mejora de esta gramática, en el archivo *calcMejor.y*, en el que las operaciones de multiplicación y división tienen prioridad según su orden de aparición de izquierda a derecha. El factor de la derecha en una multiplicación o en una división tiene que ser un entero positivo y no puede ser una expresión. Así, cuando aparecen varias de estas operaciones seguidas, estas se realizan en orden de derecha a izquierda y no ocurren los fallos que se producían en *calcOrig.y*.

- Debido a la ambigüedad de la gramática y a que no existe una preferencia, algunas expresiones que son correctas, la calculadora puede identificarlas como fuera del lenguaje de la gramática. Por ejemplo, al introducir la expresión “1*1 + 3” da “syntax error”. Esto se debe a que la calculadora puede sustituir “1 + 3” por el token *exp* y llegar a la expresión “*exp * exp*” que no puede volver a ser sustituida.

  En la calculadora *caclMejor.y*, esto no ocurre porque se ha definido el orden de prioridad entre las operaciones.

Ejercicio 2.1

Para realizar este ejercicio he modificado la calculadora mejorada y he creado los archivos *ej21.y* y *ej21.l* a partir de *calcMejor.y* y *calcMejor.l* respectivamente.

En el archivo *ej21.l* he añadido un estado que me permite reconocer una línea en la que se especifica el valor de la base *b* y una expresión que me permite identificar los números seguidos de la letra "b" que están escritos en la base *b* antes definida, y si no, por defecto *b* toma el valor 10. Además, tras leer un número en otra base, se deberá transformar a base decimal antes de que la calculadora pueda operar con él.

Estado que reconoce las líneas que especifican el valor de la base:

```
"b"[ \t]*"=" {BEGIN(detbase); return(BASE);}
<detbase>[2-9]|10 {b = atoi(yytext);}
<detbase>\n {BEGIN(INITIAL); return(EOL);}
```

Código que reconoce un número en otra base y su transformación a decimal:

```
[0-9]+b    {
    num = atoi(yytext);
    suma = 0;
    pot = 1;
    while(num != 0){
        suma = suma + (num%10)*pot;
        num = num/10;
        pot = pot*b;
    }
    yylval = suma;
    return(NUMBER);
}
```

La variable *num* contiene el valor del número leído seguido de una "b", *suma* es el valor del número leído en otra base cuando se transforma en decimal, *b* es el valor de la base definida y *pot* es una variable auxiliar que ayuda en la transformación a decimal.

En el fichero *ej21.y* se añade un nuevo token: BASE, que devuelve *ej21.l* cuando se especifica la base. Además, se añade una transformación más al estado inicial *calclist* que nos permite reconocer las líneas en las que se ha especificado una base:

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

Ejercicio 2.2

Para realizar este ejercicio he modificado la calculadora mejorada y he creado los archivos *ej22.y* y *ej22.l* a partir de *calcMejor.y* y *calcMejor.l* respectivamente.

En *ej22.l* he añadido el mismo estado que en *ej21.l* que me permite leer las líneas que especifican el valor de la base y, además, he añadido otro estado para leer el final de una línea con operaciones aritméticas, ya que hay dos formas de acabar una instrucción. Después de ";" devolvemos un token y si añadimos una "b" devolvemos otro token distinto:

```
; {BEGIN(fin);}
<fin>\n {BEGIN(INITIAL); return(EOL);}
<fin>b[ \t]*\n {BEGIN(INITIAL); return(EOLB);}
```

En *ej22.y*, añadimos como en el ejercicio anterior una transformación de *calclist* para las líneas en las que se especifica la base. Además, añadimos también a *calclist* otra transformación para leer las instrucciones con una "b" al final y calculamos el valor del resultado en el valor de la base que se haya especificado, y si no, por defecto el valor de la base será 10.

```
calclist : /* nada */
     | calclist BASE EOL { b = $2; }
     | calclist exp EOL { printf("=%d\n", $2); }
     | calclist exp EOLB {
        num = $2;
        pot = 1;
        suma = 0;
        while(num != 0){
            suma = suma + (num % b) * pot;
            pot = pot * 10;
            num = num / b;
        }
        printf("=%d\n", suma);
    }
     ;
```

Ejercicio 2.3

Para realizar este ejercicio he modificado la calculadora mejorada y he creado los archivos *ej23.y* y *ej23.l* a partir de *calcMejor.y* y *calcMejor.l* respectivamente.

En *ej23.l* añadimos las siguientes expresiones:

```
"acum" {return(OAC);}
"acum"[ \t]*":=" {return(DAC);}
```

Los token *OAC* Y *DAC*, que indican que se trabaja con el acumulador como un operando o para asignarle un valor respectivamente, serán devueltos a *ej23.y*.

En *ej23.y* se añaden los tokens *OAC* y *DAC* y el valor entero *ac* que guarda el valor del acumulador.

A *calclist* se le añade una transformación que permite reconocer las líneas en las que se le asigna un valor al acumulador a partir de una expresión:

```
calclist : /* nada */
     | calclist exp EOL { printf("=%d\n", $2); }
     | calclist DAC exp EOL { ac = $3; }
     ;
```

Y a *factorsimple* se le añade otra transformación que reconoce el acumulador como un operando:

```
factorsimple :  OP exp CP { $$ = $2; }
           | NUMBER
           | OAC {$$ = ac; }
           ;
```

Ejercicio 3

Con *ej3.l* reconocemos los caracteres “x”, “y” y “z” que son los caracteres terminales de la gramática *ej3.y*.

Para cada uno de estos caractares terminales y para el carácter de salto de línea, que nos indica cuando hemos acabado de leer una expresión, devolvemos un token. Además, ignoramos los espacios y devolvemos el resto de caracteres, si estos aparecen en alguna de nuestras expresiones, al ejecutar el programa este mostrará “syntax error” por pantalla.

En *ej3.y* definimos la gramática propuesta en el enunciado:

```
S: /* nada */
     | C X S
     ;
B:   X C Y
     | X C
     ;
C:   X B X
     | Z
     ;
```

A esta gramática, deberemos añadir un estado inicial que nos permita leer las líneas hasta el siguiente salto de línea de forma que cada línea contenga una expresión, perteneciente o no, al lenguaje definido por la gramática del enunciado. Si la expresión escrita es reconocida por el lenguaje definido por la gramática de *ej3.y*, el programa seguirá ejecutándose; y en caso contrario, terminará la ejecución y se mostrará por pantalla “syntax error”.

Transformaciones del estado inicial añadido a la gramática del enunciado en el archivo *ej3.y*:

```
inicio: /* nada */
     | inicio S EOL
     ;
```

El lenguaje que define esta gramática es $\mathrm{L} = \{(x^{2n}\mathrm{z}((y + \varepsilon)x)^n\ x)^* \mid \mathrm{n} \geq 0\}$