Grupo Jueves 12:00-14:00 semanas B

- Práctica 3-

Autor: Sergio Ros Alcázar
NIP: 874792
Autor: Irene Pascual Albericio
NIP: 871627

EJERCICIO 1:

1. Resumen:
   - En este ejercicio, el objetivo principal era compilar los archivos fuente proporcionados y analizar los resultados obtenidos. Para el programa "calcOrig", hemos detectado 16 avisos durante la compilación, concretamente conflictos de reduce/reduce. Estos warnings ocurren cuando existen dos o más reglas que pueden aplicarse a la misma secuencia de entrada, generando ambigüedad en el proceso de análisis. Aun así, el programa funcionaba correctamente para las operaciones más básicas y sencillas. Sin embargo, para operaciones más complejas, como 100/10/5, en donde debería devolver 2, devuelve 50, ya que otorga preferencia a la parte derecha de la operación; o con sumas y/o restas enlazadas como 5+5+7, devuelve error.

   - Sin embargo, en “calcMejor”, estos errores dejan de aparecer. Las diferencias notables entre "calcOrig" y "calcMejor" se centran en que en "calcOrig" no era posible agregar múltiples operaciones consecutivas, lo que resultaba en un "error de sintaxis". Por el contrario, en "calcMejor" esta opción estaba habilitada y funcionaba correctamente, debido a que, mientras que en “calcOrig” la regla de producción factor se ha realizado de una manera lineal, en “calcMejor” no, por lo que realiza correctamente todas las operaciones

EJERCICIO 2.1

1. Resumen:
   - En esta actividad, se requiere que la calculadora pueda procesar números enteros expresados en decimal o en una base alternativa, que puede ser seleccionada dentro de un rango de 2 a 10. Tras ingresar "b=número" en una línea, cualquier secuencia de dígitos que estén dentro del rango de 0 a b-1 y que finalice con la letra "b", será interpretada como un número entero en la base b correspondiente. Para ello, hemos hecho diversas modificaciones en el fichero base de Calcmejor. En primer lugar, en el fichero perteneciente a bison, hemos añadido dos tokens, IG(equivalente a =), y B(equivalente a la base), para que el valor y la base puedan ser interpretados por bison. Además, hemos añadido una variable natural b, predefinida a 10, para que esa sea la base por defecto al inicializar el ejercicio. Por último, hemos añadido una regla de producción adicional, la cual asignará el valor deseado a nuestra variable b(base):

- calclist B IG NUMBER EOL {b=$4;}

– Respecto al fichero de flex, hemos añadido las siguiente reglas:

```
[0-9]+  {yylval = atoi(yytext); return(NUMBER);}
[0-9]+"b"    {
                if(b<11 && b >1){
                    yylval = atoi(yytext);
                    int res = 0;
                    int i = 0;
                    while (yylval != 0) {
                        if((yylval % 10 ) < b){

                            res = res + yylval %10 * (pow(b,i));
                            yylval = yylval/10;
                            i++;
                        }
                        else{
                            i=0;
                            printf("El numero en base %d no es correcto", b);
                            return(yytext[0]);
                        }
                    }
                    yylval = res;
                    return (NUMBER);
                }
                else{
                    printf("La base intruducida %b no esta entre 2 y 10", b);
                    return(yytext[0]);
                }
            }
```

– Así, si no se introduce un número en base b, se hará la operación de forma normal, con los números en base decimal, mientras que, si introducimos un número en base b, se ejecutará el código descrito anteriormente, y, si es válido, devolverá a bison “NUMBER”, ya traducido a decimal. Como se puede observar, hemos tenido en cuenta que, en el caso de que el valor de b sea mayor a 10 o el número el cual debe ser traducido a base b, tenga algún digito mayor o igual que b, el ejercicio hará saber que el número no es correcto y devolverá syntax error.

2. Pruebas.

– Respecto a las pruebas, como se puede observar en la siguiente imagen, hemos tenido en cuenta casi todos los casos que podían dar lugar a errores o ambigüedades, dándonos en todos los casos el resultado esperado.

```
hendrix02:~/SEGUNDO/TComp/  bison -yd ej21.y
hendrix02:~/SEGUNDO/TComp/ flex ej21.l
hendrix02:~/SEGUNDO/TComp/ gcc lex.yy.c y.tab.c -lfl -lm -o ej21
y.tab.c: In function 'yyparse':
y.tab.c:1146:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
       yychar = yylex ();
                ^
y.tab.c:1317:7: warning: implicit declaration of function 'yyerror' [-Wimplicit-function-declaration]
       yyerror (YY_("syntax error"));
       ^
hendrix02:~/SEGUNDO/TComp/ ./ej21
30+3
=33
30b-3
=27
b=2
10b
=2
b=6
554b*6-(100/10*2b-4)
=1268
17b+2
El numero en base 6 no es correcto
syntax error
hendrix02:~/SEGUNDO/TComp/
```

Ejercicio 2.2

1. Resumen:
   - En este ejercicio, se solicita que la calculadora siga la misma funcionalidad que en el ejercicio anterior, permitiendo la inserción de números enteros en cualquier base numérica desde 2 hasta 10. Asimismo, se requiere que todas las líneas de entrada finalicen con el carácter ";" a menos que se trate de un cambio de base, en cuyo caso finalizará con ";b", y el resultado obtenido de la operación se mostrará en la base b especificada; mientras que si finaliza con ";", se presentará en formato decimal.
   - Para realizar este ejercicio, en primer lugar, hemos añadido varios tokens en el fichero .y (bison) para que la entrada pudiera ser interpretada por éste. Concretamente, hemos añadido IG(=), BASE(b), ESCR_BASE(;b) y BASE10(;). Respecto a las reglas de producción, hemos añadido dos. La primera, es la misma que la del ejercicio anterior, que asignaba un nuevo valor a la base, y una segunda, la cual es la encargada de traducir el resultado de la operación a la base solicitada.
     - calclist BASE IG NUMBER EOL {b=$4;}
     - calclist exp ESCR_BASE EOL

   - Dentro de esta última calclist, hemos realizado el código correspondiente para realizar este cambio de base, teniendo en cuenta que la base debe encontrarse entre 2 y 10. En el caso de que así sea, el resultado mostrado estará en base b.

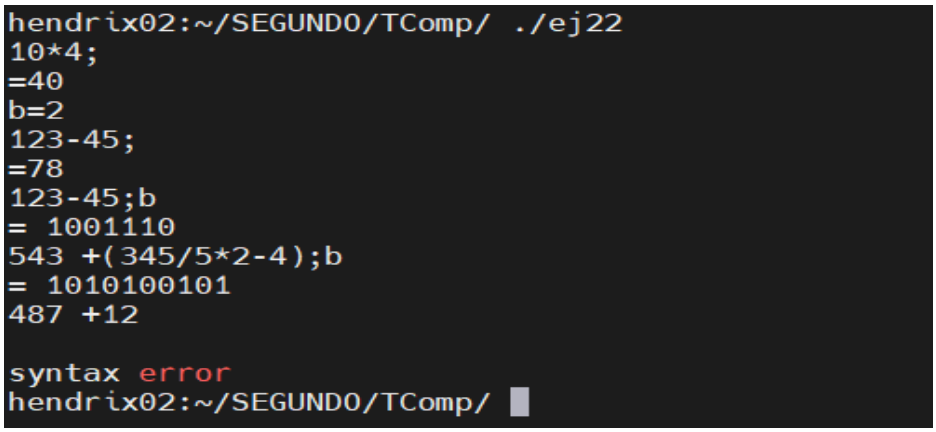
```
calclist : /* nada */
    | calclist exp BASE10 EOL { printf("=%d\n", $2); }
    | calclist BASE IG NUMBER EOL { b= $4 ;}
    | calclist exp ESCR_BASE EOL {
      if(b<11 && b >1){
          int num = $2;
          int res = 0;
          int i = 1;
          while (num!=0){
              res = res + (num % b) * i;
              num = num/b;
              i = i*10;
          }
          printf("= %d \n", res);
      }
      else{
          printf("La base intruducida %b no esta entre 2 y 10", b);
          return 0;

      }
    }
```

En el fichero de flex(.l), hemos añadido las siguientes reglas, en concordancia con los tokens añadidos en bison:

- "b" {return (BASE);}
- ";b" {return (ESCR_BASE);}
- ";" {return (BASE10);}
- "=" {return(IG);}

2. Pruebas:

   – Respecto a las pruebas, hemos realizado todos los posibles casos y operaciones que podrían dar lugar a errores o ambigúedades, y todos los resultados han sido correctos.



Ejercicio 2.3

1. Resumen
   – En este ejercicio, se requiere adaptar la calculadora para incorporar una variable de acumulación. Se van a incluir en la gramática la capacidad de asignar valores a esta función y de hacer referencia a ella. Las asignaciones se presentarán con la estructura "acum:=expresión". En primer lugar, en fichero de bison(.y), hemos añadido (de nuevo) el token DES(:=), y el token AC(acum). Además, como pedía el enunciado, hemos introducido una variable entera llamada ac inicializada a 0.

   – Además, hemos añadido dos nuevas reglas de producción. La primera, que hace referencia a la estructura pedida en el enunciado, que permite asignar valores a la variable y de esta forma la variable ac tendrá asignado el valor de la expresión introducida; y la segunda, que permite que al introducir acum, se muestre su valor actualizado.
     - calclist AC DES exp EOL { ac= $4 ;}
     - calclist AC { printf("=%d\n", ac);}

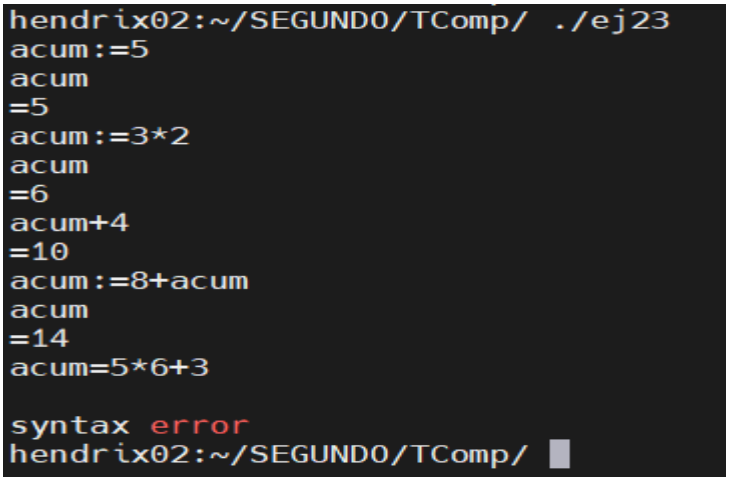
- Esto nos permite hacer asignaciones a esta variable acumuladora, pero necesitamos hacer operaciones con ella y referencias. Para ello, la hemos añadido como valor en la regla de factor simple, de forma que se pueda usar tanto números como nuestra variable para realizar operaciones y ser referenciada, como se puede observar en esta imagen.

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

- Por último, en el fichero flex (.l) hemos añadido algunas reglas conforme con los tokens añadidos previamente en bison:
    - ":=" {return(DES);}
    - "acum" {return(AC);

2. Pruebas

- Respecto a las pruebas para ver si el ejercicio había sido realizado correctamente, éstas han sido muy significativas, pues tras comprobar muchos casos en los que la variable ac podría fallar, todos los resultados han sido correctos, como podemos ver en la siguiente imagen.



Como podemos ver, todas han sido correctas y en el último caso, donde en vez de poner := para la asignación hemos puesto el igual, ha dado error.

EJERCICIO 3:

1. Resumen:

   - Hemos creado dos archivos, uno en flex y otro en bison.
     El archivo de flex lo hemos usado para leer los caracteres "x", "y" y "z", los cuales mandábamos a bison para que los utilizase como tokens. Además, indicaba también si leía un salto de línea o si leía cualquier otro carácter (lo cual generaba error).
     Respecto al archivo de bison, hemos declarado primero los tokens obtenidos del archivo en flex y luego creado las distintas normas: inicial, S, B y C.
     La norma inicial servía principalmente para indicar que se iba a empezar por S y para marcar el final de la cadena.
     Una vez comenzaba en la norma S, iba detectando si las letras eran las correctas y saltando a las diferentes normas si era necesario.
     Además, al terminar en x, y o z, podía significar el final de la palabra, por lo que si esas letras iban concatenadas al fin de línea, lo llevaba a la norma inicial, para mostrar ese fin de palabra.

   - Ahora procederemos a explicar el lenguaje que se reconoce:
     El lenguaje reconocido es aquel que empieza por el símbolo no terminal S, a través del cual podemos llegar al símbolo no terminal C y así llegaríamos al B.
     Con S podemos enlazarlo consecutivamente durante n veces:
     S → CxS → CxCxCxCxCx.....S , y S puede ser también épsilon por lo que:
     S → (Cx)^n, para todo n>=0.
     A partir de C podemos...:
     C → xBx → xxCyx → xxxxxxx...xCy... xxxxxxx
     → x^(2m)C((y+épsilon)x)^m, para todo m>=0
     C → xBx → xxCx
     C → z (aquí termina la derivación de C)
     Por tanto, la expresión obtenida para representar el lenguaje L(G)
     L(G) = {(x^(2m)z((y+épsilon)x)^m)x)^n | m,n >= 0}

2. Pruebas:

```
C:\Users\irene\OneDrive\Escritorio\tcomppr3>ej3.exe
zx

xxzxx
zxzx
xxzyxx
xxxxzxyxx
xxxxzxyxxzx

xxzyxxxxzyxx
zxzxzxzxzxxxzxzzcxxzxzxzzzx
syntax error

C:\Users\irene\OneDrive\Escritorio\tcomppr3>
```