Respuesta a las 5 preguntas del primer capitulo

1. Will the calculator accept a line that contains only a comment? Why not? Would it be easier to fix this in the scanner or in the parser?

En el código que proporcionó el libro no se aceptan los comentarios, pues el programa espera expresiones matemáticas para solucionarlas, este problema se podría solucionar con mayor facilidad en el parser añadiendo una regla que acepte solo EOL sin estar acompañado de una expresión.

```
calclist: /* nothing */
 | calclist exp EOL { printf("= %d\n", $1); }
/* Modificación que permite que la calculadora siga funcionando al recibir un comentario */
 | calclist EOL {}
 ;
```

2. Make the calculator into a hex calculator that accepts both hex and decimal num bers. In the scanner add a pattern such as 0x[a-f0-9]+ to match a hex number, and in the action code use strtol to convert the string to a number that you store in yylval; then return a NUMBER token. Adjust the output printf to print the result in both decimal and hex.

Para esto, debemos realizar modificaciones tanto en el scanner como en el parser.

En el scanner agregamos la expresión regular que captura los números hexadecimales, para mayor flexibilidad se incluyen tanto números en mayúscula como en minúscula:

```
[0-9]+ { yylval = atoi(yytext); return NUMBER;
/* Modificación que incluye a los números hexadecimales */}
0[xX][0-9a-fA-F]+ { yylval = strtol(yytext, NULL, 16); return NUMBER; }
```

En el parser se agrega un parámetro a la impresión de los resultados, formateando una de estas como decimal y la otra como un número hexadecimal entre paréntesis:

```
calclist: /* nothing */
 | calclist exp EOL { printf("= %d (0x%X)\n", $2, $2); }
 | calclist EOL {}
 ;
```

3. (extra credit) Add bit operators such as AND and OR to the calculator. The obvious operator to use for OR is a vertical bar, but that's already the unary absolute value operator. What happens if you also use it as a binary OR operator, for example, exp ABS factor?

Teniendo como operadores bit a bit los siguientes:
AND: &

OR: |

XOR: ^

NOT: ~

Desplazamiento a la derecha: >>

Desplazamiento a la izquierda: <<

Considerando | como el símbolo de OR y redefinimos el valor absoluto como un número entre dos |, es decir la notación matemática de valor absoluto, para evitar confusiones en el scanner sobre si una expresión requiere hallar su valor absoluto o aplicar el operador bitwise OR al encontrar un "|" este será enviado al parser, donde por las reglas de producción podrá distinguir entre un OR y  un valor absoluto.

Modificación en el scanner:

```
"|"    { return '|'; }
"&"    { return AND; }
"^"    { return XOR; }
"~"    { return NOT; }
"<<"   { return LEF; }
">>"   { return RIG; }
"("    { return '('; }
")"    { return ')'; }
```

Como se puede apreciar, además de lo solicitado se agregaron los símbolos de paréntesis, que permitirán establecer operaciones prioritarias que no siguen la jerarquía normal.

Modificación en el parser, donde debemos reorganizar la jerarquía de las operaciones, para que las expresiones se evalúen correctamente, siendo las operaciones OR, XOR y AND menos prioritarias que las operaciones aritméticas y estas menos prioritarias que las operaciones valor absoluto, paréntesis y NOT:

```
calclist: /* nothing */
 | calclist orbit EOL { printf("= %d (0x%X)\n", $2, $2); }
 | calclist EOL
 ;
orbit: xorbit
 | orbit '|' xorbit { $$ = $1 | $3; }
xorbit: andbit
 | xorbit XOR andbit { $$ = $1 ^ $3; }
andbit: desp
 | andbit AND desp { $$ = $1 & $3; }
desp: exp
 | desp LEF exp { $$ = $1 << $3; }
 | desp RIG exp { $$ = $1 >> $3; }
exp: factor
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 ;
```

```
factor: term
  | factor MUL term { $$ = $1 * $3; }
  | factor DIV term {
          if ($3 != 0 ) {
             $$ = $1 / $3;
          } else {
             yyerror("División por cero");
             $$ = 0;
            }
          }
 ;
term: NUMBER
  | NOT term   { $$ = ~$2; }
  | '(' orbit ')' { $$ = $2; }
  | '|' orbit '|'  { $$ = $2 >= 0? $2 : - $2; }
;
```

Adicionalmente, para que sea más sencillo y rápido realizar varias pruebas, permitimos recibir una entrada como archivo de texto sin usar redireccionamiento, donde cada línea es una expresión que evaluará la calculadora:

```
int main(int argc, char **argv) {
    if (argc > 1) {
        /* Si hay argumento, abrir el archivo */
        yyin = fopen(argv[1], "r");
        if (!yyin) {
            fprintf(stderr, "Error: No se puede abrir el archivo '%s'\n", argv[1]);
            return 1;
        }
        printf("Procesando archivo: %s\n\n", argv[1]);
    } else {
        /* Si no hay argumento, usar entrada estándar */
        printf("Calculadora - Ctrl+D para salir\n");
        yyin = stdin;
    }

    yyparse();

    if (yyin != stdin) {
        fclose(yyin);
    }

    return 0;
}

void yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

4. Does the handwritten version of the scanner from Example 1-4 recognize exactly the same tokens as the flex version?

No exactamente. Para entradas bien formadas ambos producen los mismos resultados, pero difieren en casos borde: Flex aplica automáticamente la regla del token más largo y maneja por defecto caracteres no reconocidos y EOF, mientras que el scanner manual depende de cómo el programador haya implementado esos casos, pudiendo comportarse de forma distinta ante entradas inesperadas.

5. Can you think of languages for which flex wouldn't be a good tool to write a scanner?

Los lenguajes naturales como inglés, francés y demás que usamos para comunicarnos en nuestro día a día no serían aptos para un scanner en Flex, pues estos lenguajes son sensibles al contexto, del mismo modo cualquier otro lenguaje que siga una gramática tipo 0 o 1.

6. Rewrite the word count program in C. Run some large files through both versions. Is the C version noticeably faster? How much harder was it to debug?

Versión C:

Declaraciones, iguales a la versión en flex, excepto que para realizar el conteo de palabras de usa una variable que nos indica si seguimos en la misma palabra para no contarla más veces de las requeridas.

```c
 /* wc_manual.c */
#include <stdio.h>
#include <ctype.h>
int main() {
    int chars = 0;
    int words = 0;
    int lines = 0;
    int in_word = 0;
    int c;
```

Lógica:

```
    while ((c = getchar()) != EOF) {
        chars++;

        if (c == '\n') {
            lines++;
        }

        if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')) {
            if (!in_word) {
                words++;
                in_word = 1;
            }
        } else {
            in_word = 0;
        }
    }

    printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
}
```

Para trabajar con archivo grandes usaremos el redireccionamiento desde la terminal, de este modo, tras crear el archivo .txt donde estará un texto largo (entrada.txt) y obtener los archivos ejecutables (ejemplo1 u ejercicio6), se ejecutarán los comandos:

./ ejercicio6 < entrada.txt

./ ejemplo1 < entrada.txt

Para comparar la velocidad de ejecución entre ambas versiones se utiliza el comando time, que mide el tiempo que tarda cada programa en procesar la entrada. Como con archivos pequeños la diferencia es imperceptible, primero se genera un archivo de mayor tamaño repitiendo el contenido de entrada.txt mil veces:

for i in $(seq 1000); do cat entrada.txt; done > entrada_grande.txt

Luego se mide el tiempo de cada versión:

time ./ejercicio6 < entrada_grande.txt
time ./ejemplo1 < entrada_grande.txt

El resultado muestra tres valores: real indica el tiempo total transcurrido, user el tiempo de CPU del proceso y sys el tiempo usado por el sistema operativo. La versión en C es ligeramente más rápida que la versión en Flex, ya que evita el overhead de las funciones generadas automáticamente por dicha herramienta, aunque la diferencia es mínima.