```
    # Compile HelloWorld.c to an executable named
HelloWorld. -o for specifying the output file name
    gcc -o HelloWorld HelloWorld.c
    # Preprocess HelloWorld.c and save the output to
HelloWorld.i for assembly code generation. '>' for output
file name setting
    gcc -E HelloWorld.c > HelloWorld.i
    # Generate assembly code from HelloWorld.i with Intel
syntax
    gcc -S -masm=intel HelloWorld.i
    # Assemble the assembly code in HelloWorld.s to an
object file HelloWorld.o
    as -o HelloWorld.o HelloWorld.s
    # Disassemble HelloWorld.o and save the output to
HelloWorld.dump. Disassemble convert to human readable
format. -d works for disassembling
    objdump -M intel -d HelloWorld.o > HelloWorld.dump
    # Compile HelloWorld.c to an object file HelloWorld.o
    gcc -c -o HelloWorld.o HelloWorld.c
    # Disassemble HelloWorld.o and save the output to
HelloWorld2.dump
    objdump -M intel -d HelloWorld.o > HelloWorld2.dump
```

cal.l

This option tells flex not to generate the 'yywrap' function, which is used for input file handling.

Since we are using Bison, we don't need 'yywrap'.

%option noyywrap

This block of code is copied verbatim into the generated C file.

It includes the header file generated by Bison, which contains token definitions and other necessary declarations.

%{

  #include"cal.tab.h"

%}

```
# Defines a pattern named 'delim' that matches a space or a tab character.
delim[ \t]
# Defines a pattern named 'ws' that matches one or more whitespace characters (spaces or tabs).
ws ({delim}+)
# Defines a pattern named 'digit' that matches any single digit (0-9).
digit [0-9]
# Defines a pattern named 'digits' that matches one or more digits.
digits({digit}+)
# Defines a pattern named 'letter' that matches any single letter (uppercase or lowercase).
letter[a-zA-Z]
# Defines a pattern named 'letters' that matches one or more letters.
letters({letter}+)
# Defines a pattern named 'us' that matches an underscore character.
us[_]
# Defines a pattern named 'identifier' that matches a sequence starting with a letter or underscore,
# Followed by zero or more letters, underscores, or digits.
identifier ({letter|us})({letter|us|digit}*)
# Marks the beginning of the rules section in the flex file.
%%
# Matches whitespace characters and ignores them (no action is taken).
{ws} {}
# Matches a sequence of digits, converts it to an integer using 'atoi',
# Assigns it to 'yylval', and returns the token 'NUM'.
{digits} {yylval=atoi(yytext); return (NUM);}
# Matches the '+' character and returns the token 'ADD'.
"+" {return (ADD);}
```

Matches the '-' character and returns the token 'SUB'.

"-" {return (SUB);}

%%

The main function calls 'yylex' to start the lexical analysis process and then returns 0.

This is not needed if we are using Bison, as Bison will generate its own main function.

int main ()

{

   yylex();

   return 0;

}

cal.y

```
%{
    // Include standard I/O library for input/output functions
    #include<stdio.h>
    // Declare the error handling function
    void yyerror(char *s);
    // Declare the lexical analyzer function
    int yylex();
/*
Define tokens for numbers and operators
%token NUM ADD SUB
Define the start symbol for the grammar
%start cal
Define operator precedence and associativity
%left ADD SUB
*/

%}

%token NUM ADD SUB
%start cal
%left ADD SUB

%%
```

```
// Grammar rules section
// Define the program as a sequence of statements
program: statements
    ;
// Define statements as an if statement with an expression and a block of code
statements: IF LP exp RP LB id_declare RB
    ;
// Define an identifier declaration as an assignment of an expression to an
identifier
id id_declare: ID ASSIGN exp SEMICOLON
    ;
// Define expressions with addition, subtraction, or a number
exp: exp ADD number
    | exp SUB number
    | number
// Define the cal rule to evaluate an expression and print the result
cal: exp {$$=$1; printf("exp = cal %d\n",$$);}
    ;
// Define the exp rule to handle addition, subtraction, and numbers
exp: exp ADD NUM {$$=$1+$3; printf("exp: exp SUM NUM %d\n",$$);}
    | exp SUB NUM {$$=$1-$3; printf("exp: exp SUB NUM %d\n",$$);}
    | NUM {$$=$1; printf("exp = NUM %d\n",$$);}
    ;

%%
// The main function calls 'yyparse' to start the parsing process and then prints
a message
int main()
{
    yyparse();
    printf("Parsing Finished\n");
}
// The error handling function prints an error message to stderr
void yyerror(char *s)
{
    fprintf(stderr, "error: %s\n", s);
}
```

Makefile:

```
input=input.txt

output=output.txt

main: cal.l
    flex cal.l
    gcc lex.yy.c
    a < $(input) > $(output)

main2: cal.y cal.l
    bison -d cal.y
    flex cal.l
    gcc cal.tab.c lex.yy.c
    a < $(input) > $(output)
    @cat $(output)
```