Hcalc.l

```
%{
#include <stdio.h>
#include <stdlib.h> // for strtol
#include "hcalc.tab.h"  // Include the parser h
generated by Bison
%}

%%
[ \t]  ; // Skip whitespace
\n     return EOL; // End of line
"+"    return ADD;
"-"    return SUB;
"*"    return MUL;
"/"    return DIV;
"|"    return ABS;
0[xX][0-9a-fA-F]+ { // Hexadecimal number
   yylval = (int)strtol(yytext + 2, NULL,
16);
// Convert hex string to integer
   return NUMBER;
}
[0-9]+  { // Decimal number
   yylval = atoi(yytext); // Convert decimal str
integer
   return NUMBER;
}

.      { fprintf(stderr, "Invalid character:
%s\n", yytext); } // Handle invalid input

%%
int yywrap() {
   return 1;
}
```

Hcalc.y

```
%{
#include <stdio.h>
#include <stdlib.h> // for strtol
extern int yylex();
extern int yyerror(char *s);
%}

/* Declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL

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

exp: factor      { $$ = $1; }
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 ;

factor: term      { $$ = $1; }
 | factor MUL term { $$ = $1 * $3; }
 | factor DIV term { $$ = $1 / $3; }
 ;

term: NUMBER  { $$ = $1; }
 | ABS term   { $$ = $2 >= 0? $2 : -$2; }
;

%%
int main() {
   printf("Enter arithmetic expressions
in hexadecimal(i.e. 0xa + 0x8):\n");
   yyparse(); // Call the parser
   return 0;
}
int yyerror(char *s) {
   fprintf(stderr, "Error: %s\n", s);
   return 0;
}
```

Calc.l

```
%{
#include "calc.tab.h"  // Include the Bison-gen
header
#include <stdlib.h>
%}

%%

[0-9]+        { yylval = atoi(yytext); return NUM
"+"           { return PLUS; }
"-"           { return MINUS; }
"*"           { return MULT; }
"/"           { return DIV; }
"quit"        { return QUIT; }  // Return QUIT tok
[ \t]         ;  // Ignore spaces and tabs
\n            { return NEWLINE; }  // Return NEW
.             { return yytext[0]; }  // Return any oth

%%

int yywrap() {
   return 1;
}
```

Calc.y

```
%{
#include <stdio.h>
extern int yylex();
extern int yyerror(char *s);
%}

%token NUMBER PLUS MINUS MULT
DIV QUIT NEWLINE

%left PLUS MINUS    // Lower precedence
%left MULT DIV      // Higher precedence

%%

program:
   /* Empty */              { /* Start empty */ }
   | program statement      { /* Process multipl
*/ }
   ;

statement:
   NEWLINE
 { /* Empty line, do nothing */ }
   | expression NEWLINE
 { printf("Result: %d\n", $1); }
   | QUIT NEWLINE           { return 0; }
   | error NEWLINE
{ printf("Invalid expression, try again
\n"); yyclearin; }
   ;

expression:
   NUMBER                   { $$ = $1; }
   | expression PLUS expression
{ $$ = $1 + $3; }
   | expression MINUS expression
{ $$ = $1 - $3; }
   | expression MULT expression
{ $$ = $1 * $3; }
   | expression DIV expression    {
      if ($3 == 0) {
         yyerror("Division by zero");
         $$ = 0;
      } else {
         $$ = $1 / $3;
      }
   }
   ;

%%

int yyerror(char *s) {
   fprintf(stderr, "Error: %s\n", s);
   return 0;
}

int main() {
   printf("Enter arithmetic expressions
(type 'quit' to exit):\n");
   while (1) {
      yyparse();  // Keep parsing until
QUIT is encountered
   }
   return 0;
}
```

Word count

```
/* just like Unix wc */
%{
#include<stdio.h>
#include<string.h>
int chars = 0;
int words = 0;
int lines = 0;
%}

%%

[a-zA-Z]+  { words++; chars+=strlen(yytext); }
\n         { chars++; lines++; }
.          { chars++; }

%%

int main()
{
  printf("Enter input: ");
  yylex();
  printf("Lines: %8d Words: %8d Characters: %8
words, chars);
}

int yywrap()
{
        return 1;
}
```

Vowel count

```
%{
#include<stdio.h>
int v=0, c=0;
%}

%%
[ \t\n]+ ;
[aeiouAEIOU] {v++;}
[^aeiouAEIOU] {c++;}

%%

int main()
{
        printf("Enter input:\n");
        yylex();
        printf("Number of vowels: %d\n", v);
        printf("Number of consonents: %d\n", c);

}

int yywrap()
{
        return 1;
}
```

Recognize

```
%{
#include<stdio.h>
%}

%%
if |
else |
printf {printf("\n%s is a keyword", yytext);}
[0-9]+ {printf("\n%s is a number", yytext);}
[a-zA-Z]+ {printf("\n%s is a word", yytext);}
.|\n {ECHO;}
%%

int main()
{
        printf("\nEnter input: ");
        yylex();
}

int yywrap()
{
        return 1;
}
```

Recognize octal

```
%{
#include <stdio.h>
#include <stdlib.h>
int count = 0;
%}

%%

0[0-7]+   {
   // Convert octal string to decimal
   long decimal = strtol(yytext, NULL, 8);  // Base 8 for
octal
   if (decimal > 250) {
      printf("Found %s\n", yytext);
      count++;
   }
}

[ \t\n]    ;  // Ignore whitespace and newlines
.          ;  // Ignore any other characters

%%

int yywrap() {
   return 1;
}

int main() {
   printf("Enter octal integers (end with Ctrl+D or
Ctrl+Z):\n");
   yylex();
   printf("Total %d\n", count);
   return 0;
}
```

Recognize binary string

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int count = 0;
%}

%%
```

```
b?[oi]+    {
   // Remove 'b' if present and process only 'O'

   char* binary = yytext;
   if (yytext[0] == 'b') {
      binary++;  // Skip 'b' if present
   }

   // Count length of binary string after 'b' (if a
   int len = strlen(binary);
   long decimal = 0;

   // Convert binary (o=0, I=1) to decimal
   for (int i = 0; i < len; i++) {
      decimal = decimal * 2 + (binary[i] ==
'i' ? 1 : 0);
   }

   // Print if valid binary and decimal value is
int range
   printf("Found %s (Decimal value =
%ld)\n", yytext, decimal);
   count++;

}

[ \t]    ;  // Ignore whitespace and newlines
[\n]  {return 0;}
.         ;  // Ignore any other characters

%%

int yywrap() {
   return 1;
}

int main() {
   printf("Enter binary strings
(end with Ctrl+D or Ctrl+Z):\n");
   yylex();
   printf("Total %d\n", count);
   return 0;
}
```

Alphanumariccalc.l

```
%{
#include <stdio.h>
#include "alphanumariccalc.tab.h"
// Include the parser header file generated by
%}

%%

[a-zA-Z]   { yylval = yytext[0]; return
LETTER; } // Match a single letter
[0-9]+     { yylval = atoi(yytext);
return DIGITS; } // Match one or more digits
[+]        return '+'; // Match the addition operato
[ \t]      ; // Skip whitespace
\n         return '\n'; // End of line
.        { fprintf(stderr, "Invalid character: %s\n'
// Handle invalid input

%%

int yywrap() {
   return 1;
}
```

Alphanumariccalc.y

```
%{
#include <stdio.h>
#include <stdlib.h> // for atoi
extern int yylex();
extern int yyerror(char *s);
%}


/* Declare tokens */
%token LETTER DIGITS

%%

input: /* nothing */
 | input expr '\n' { printf("Result = %c\n", $2); }
result character
 ;

expr: LETTER '+' DIGITS {
      int letter_ascii = $1;
      int sum_digits = $3;
      int result_ascii = letter_ascii +
sum_digits;

      // Handle cycling for lowercase letters (AS
      if (letter_ascii >= 'a' && letter_ascii
<= 'z') {
         result_ascii = 'a' + (result_ascii -
'a') % 26;
      }
      // Handle cycling for uppercase
letters (ASCII 'A' to 'Z')
      else if (letter_ascii >= 'A' &&
letter_ascii <= 'Z') {
         result_ascii = 'A' + (result_ascii -
'A') % 26;
      }

      $$ = result_ascii; // Set the result
   }
 | expr '+' DIGITS {
      int current_result = $1;
      int sum_digits = $3;
      int result_ascii = current_result +
sum_digits;

      // Handle cycling for lowercase
letters (ASCII 'a' to 'z')
      if (current_result >= 'a' &&
current_result <= 'z') {
         result_ascii = 'a' + (result_ascii -
'a') % 26;
      }
      // Handle cycling for uppercase
letters (ASCII 'A' to 'Z')
      else if (current_result >= 'A' &&
current_result <= 'Z') {
         result_ascii = 'A' + (result_ascii -
'A') % 26;
      }

      $$ = result_ascii; // Set the result
   }
 ;

%%

int main(int argc, char **argv) {
   yyparse(); // Call the parser
   return 0;
}

int yyerror(char *s) {
   fprintf(stderr, "Error: %s\n", s);
   return 0;
}
```

Calc_bitwise.l

```
%{
#include <stdio.h>
#include "calc_bitwise.tab.h" // Include the pa
file generated by Bison
%}

%%

[0-9]+     { yylval = atoi(yytext); return NUMBE
numbers
"+"        return ADD; // Addition
"-"        return SUB; // Subtraction
"*"        return MUL; // Multiplication
"/"        return DIV; // Division
"&"        return AND; // Bitwise AND
"|"        return OR;  // Bitwise OR
[ \t]      ; // Skip whitespace
\n         return EOL; // End of line
.        { fprintf(stderr, "Invalid character: %s\n'
// Handle invalid input

%%

int yywrap() {
   return 1;
}
```

Calc_bitwise.y

```
%{
#include <stdio.h>
extern int yylex();
extern int yyerror(char *s);

%}

/* Declare tokens */
%token NUMBER
%token ADD SUB MUL DIV
%token AND OR
%token EOL

%%

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

exp: factor      { $$ = $1; }
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 | exp OR factor  { $$ = $1 | $3; } // Bitwise OR
 ;

factor: term      { $$ = $1; }
 | factor MUL term { $$ = $1 * $3; }
 | factor DIV term { $$ = $1 / $3; }
 | factor AND term { $$ = $1 & $3; } // Bitwise AND
 ;

term: NUMBER  { $$ = $1; }
 ;

%%
int main(int argc, char **argv) {
   yyparse(); // Call the parser
   return 0;
}
int yyerror(char *s) {
   fprintf(stderr, "error: %s\n", s);
   return 0;
}
```

Float calculator.l

```
%{
#include <stdlib.h>
#include "calc_float.tab.h" // Include the parser header file
generated by Bison
%}

%%

[0-9]+([.][0-9]*)?([eE][+-]?[0-9]+)? {
   yylval.dval = atof(yytext); // Convert matched text to
double
   return NUMBER;
}

"+"       return ADD; // Addition
"-"       return SUB; // Subtraction
"*"       return MUL; // Multiplication
"/"       return DIV; // Division
"|"       return ABS; // Absolute value (unary)
[ \t]     ; // Skip whitespace
\n        return EOL; // End of line
.       { fprintf(stderr, "Invalid character: %s\n", yytext); }
// Handle invalid input

%%

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

Float calculator.y

```
%{
#include <stdio.h>
extern int yylex();
extern int yyerror(char *s);

%}

/* Declare tokens */

%token ADD SUB MUL DIV ABS
%token EOL

/* Declare the type of yylval */
%union {
   double dval; // Use double for floating-point numbers
}

%token <dval> NUMBER
%type <dval> exp factor term

%%

calclist: /* nothing */
 | calclist exp EOL { printf("= %g\n", $2); } // Print result as
a floating-point number
 ;

exp: factor      { $$ = $1; }
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 ;

factor: term      { $$ = $1; }
 | factor MUL term { $$ = $1 * $3; }
 | factor DIV term { $$ = $1 / $3; }
 ;

term: NUMBER  { $$ = $1; }
 | ABS term   { $$ = $2 >= 0 ? $2 : -$2; } // Absolute value
 ;

%%

int main(int argc, char **argv) {
   printf("Enter arithmetic expressions (type 'quit' to
exit):\n");
   yyparse(); // Call the parser
   return 0;
}

int yywrap() {
   return 1;
}
```

Symbol table

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TABLE_SIZE 100
// Symbol Table Entry
typedef struct Symbol {
  char name[50];
  char type[20];
  int scope;
  struct Symbol* next;
} Symbol;

// Symbol Table (hash table)
Symbol* symbolTable[TABLE_SIZE];
// Hash function to calculate index
unsigned int hash(char* name) {
  unsigned int hashValue = 0;
  for (int i = 0; name[i] != \0 ; i++) {
    hashValue = 31 * hashValue + name[i];
  }
  return hashValue % TABLE_SIZE;
}

// Insert into the symbol table
void insert(char* name, char* type, int scope)
  unsigned int index = hash(name);
  Symbol* newSymbol = (Symbol*)
   malloc(sizeof(Symbol));
  strcpy(newSymbol->name, name);
  strcpy(newSymbol->type, type);
  newSymbol->scope = scope;
  newSymbol->next = symbolTable[index];
  symbolTable[index] = newSymbol;
}

// Lookup a symbol in the table
Symbol* lookup(char* name) {
  unsigned int index = hash(name);
  Symbol* current = symbolTable[index];
  while (current != NULL) {
    if (strcmp(current->name, name) == 0)
    {
      return current;
    }
    current = current->next;
  }
  return NULL;
}

int main() {
  insert("x", "int", 0);
  insert("y", "int", 1);
  Symbol* s = lookup("x");
  if (s != NULL) {
    printf("Found %s of type %s in scope
     %d\n", s->name, s->type, s->scope);
  }
  else {
    printf("Symbol not found\n");
  }
  return 0;
}

#define TABLE_SIZE 100
// Symbol Table Entry
typedef struct Symbol {
  char name[50];
  char type[20];
  int scope;
  struct Symbol* next; // For collision
  handling (linked list)
} Symbol;
// Hash function and symbol table
Symbol* symbolTable[TABLE_SIZE];
unsigned int hash(char* name);
void insert(char* name, char* type, int scope)
Symbol* lookup(char* name);

void insert(char* name, char* type, int scope)
  unsigned int index = hash(name);
  Symbol* newSymbol = (Symbol*)
   malloc(sizeof(Symbol));
  strcpy(newSymbol->name, name);
  strcpy(newSymbol->type, type);
  newSymbol->scope = scope;
  newSymbol->next = symbolTable[index];
  symbolTable[index] = newSymbol;
}

Symbol* lookup(char* name) {
  unsigned int index = hash(name);
  Symbol* current = symbolTable[index];
  while (current != NULL) {
    if (strcmp(current->name, name) == 0){
      return current;
    }
    current = current->next;
  }
  return NULL;
}
```

Recursive Descent parser

```
E -> TE
E_p -> +TE_p | e
T -> FTp
Tp -> *FTp | e
F -> (E) | id

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
// Global variable for the input expression and p
current character
const char *input;
char lookahead;
// Function declarations for recursive parsing
void E();
void E_prime();
void T();
void T_prime();
void F();

// Helper function to match and move forward i
void match(char expected) {
  if (lookahead == expected){
    lookahead = *++input; // Move to the next cha
  }
  else{
    printf("Syntax Error: Expected '%c', found '%c
expected, lookahead);
    exit(1);
  }
}

// Function to parse Expression: E-> T F'
void E(){
  printf("E-> T E'\n");
  T();
  E_prime();
}

// Function to parse E'-> + T E' |
void E_prime() {
  if (lookahead == '+'){
    printf("E'-> + T E'\n");
    match('+');
    T();
    E_prime();
  }
  else{
    printf("E'-> \n"); //
  }
}

// Function to parse Term: T-> F T'
void T(){
  printf("T-> F T'\n");
  F();
  T_prime();
}
// Function to parse T'-> * F T' |
void T_prime(){
  if (lookahead == '*'){
    printf("T'-> * F T'\n");
    match('*');
    F();
    T_prime();
  }
  else{
    printf("T'-> \n"); //
  }
}

// Function to parse Factor: F-> (E) | id
void F(){
  if (lookahead == '('){
    printf("F-> (E)\n");
    match('(');
    E();
    match(')');
  }
  else if (isalnum(lookahead)){
    printf("F-> id\n");
    match(lookahead); // Match identifier/numbe
  }
  else{
    printf("Syntax Error: Unexpected character '%
lookahead);
    exit(1);
  }
}

// Main function to start parsing
int main(){
  // Input arithmetic expression
  // input = "(2++3)*5";
  input = (char *)malloc(100 * sizeof(char));
  printf("Input an arithmetic expression\n");
  while (scanf("%s", input) != EOF){
    lookahead = *input; // Initialize lookahead
    printf("Parsing input: %s\n", input);
    E(); // Start parsing from the start symbol E
    if (lookahead == '\0'){
      printf("Parsing successful!\n");
    }
    else{
      printf("Syntax Error: Unexpected input after
parsing.\n");
    }
    printf("Input an arithmetic expression\n");
  }
  return 0;
}
```