CS205 C/C++ Programming - Lab Project 2 - Simple Arithmetic Calculator

Name: FANG Jiawei

SID 12110804

Part 0 - Project Link

IskXCr/CS205-Project-2-Calculator

Part 1 - Analysis

The project asks the student to write a calculator that is much better than that in Project 1.

The whole project is written in `C`.

The following features are implemented to illustrate the need:

1. Interactive direct calculation from expressions involving

   a. *numbers in* ***arbitrary precision***

   b. ***math functions***, some in arbitrary precision

   c. ***variable assignments***

   d. ***comparison operators*** (evaluates to 1 if ***true***, 0 if ***false***

2. History. Press Up/Down to select a history to enter.
3. Efficient algorithms to speed up the calculation.

The following algorithms and libraries are **implemented** to speed up and ensure the correctness and efficiency of the program.

1. `number.h`: An whole arbitrary precision number **library** that supports

   a. Add, Subtraction, Multiplication, Division, Sqrt, Power to any precision

   b. sin, cos, arctan, ln, exp to limited precision (at around). Major reference: `GNU bc`

   c. Memory management: reuse of existing structure storages, linked list to reuse existing instances, manual garbage collection, ***built-in reference count (which is a part of GC)*** that avoids frequent `malloc()`, `realloc()` and `free()` operations.

2. Karatsuba's Multiplication. Reduce multiplication time complexity to at around.

   $$O(n^{1.58})$$

   See reference Karatsuba algorithm - Wikipedia.

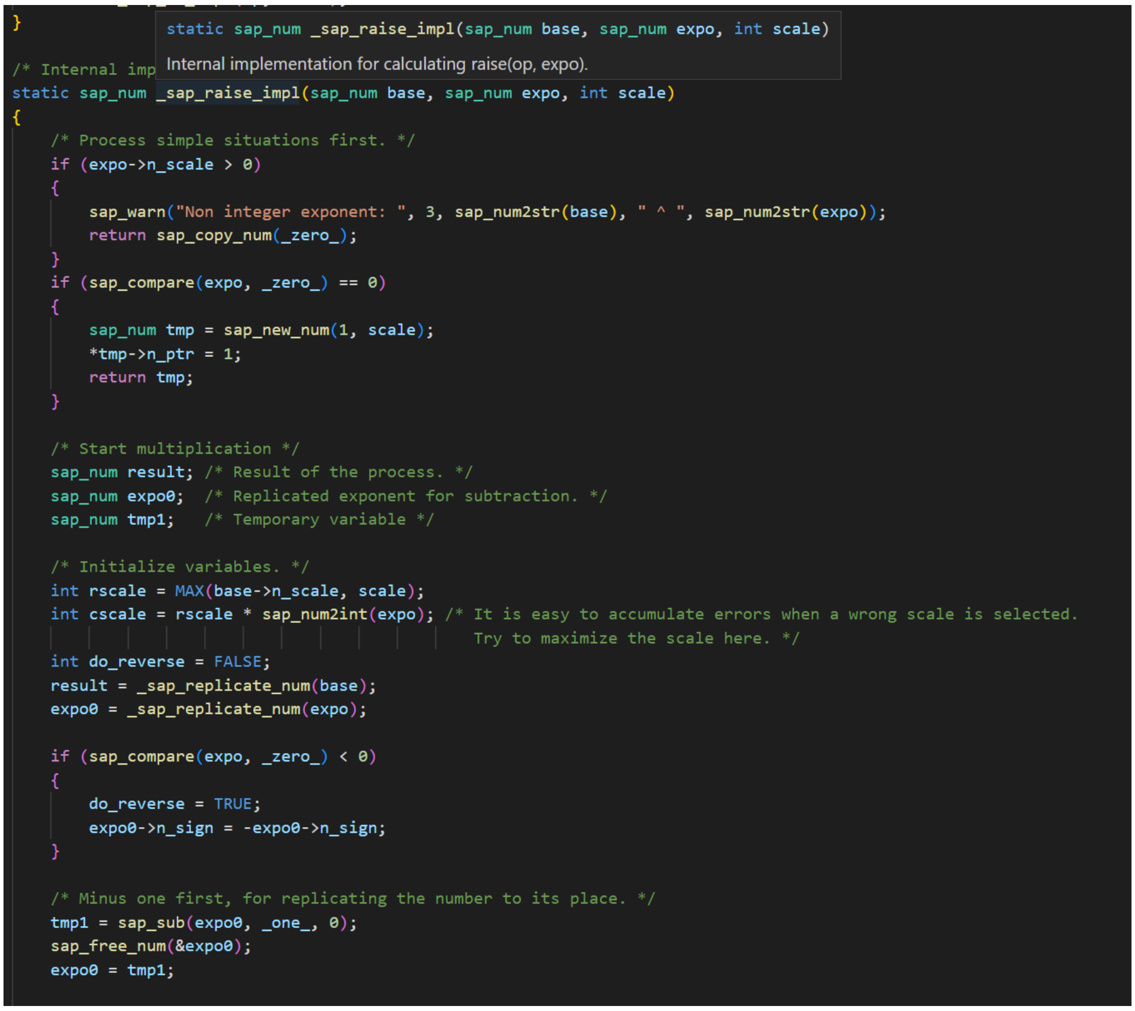
3. Infix-to-postfix algorithm. See reference: Expression Parsing, Infix to Postfix using different Precedence Values for In-Stack and Out-Stack - GeeksforGeeks.
4. Memory management: Every custom structure has corresponding constructor/destructor, and recursive resource freeing algorithms are implemented to free resources efficiently.
5. **Clear Structure**:

a. Every source file (except for main) has its corresponding header file that *declares exposed function prototypes, i.e. APIs, defines constants, and uses* `#ifndef` *to handle duplicated header inclusions*.

b. Every function is commented along with detailed explanations on algorithms and different test branches.

c. Low coupling, high-cohesion. Each library can be modified separately in an easy manner (as long as the exposed interface doesn't change).

d. Functions are written in such a way that some branches are ignored and explicitly stated in the comment of the function itself to reduce branches. For example, `NULL` checks won't be performed if the function already states that it won't accept such. ***However, all scenarios that are not explicitly stated to be avoided are considered in the best possible way***.



```
}                  static sap_num _sap_raise_impl(sap_num base, sap_num expo, int scale)

/* Internal imp    Internal implementation for calculating raise(op, expo).
static sap_num _sap_raise_impl(sap_num base, sap_num expo, int scale)
{
    /* Process simple situations first. */
    if (expo->n_scale > 0)
    {
        sap_warn("Non integer exponent: ", 3, sap_num2str(base), " ^ ", sap_num2str(expo));
        return sap_copy_num(_zero_);
    }
    if (sap_compare(expo, _zero_) == 0)
    {
        sap_num tmp = sap_new_num(1, scale);
        *tmp->n_ptr = 1;
        return tmp;
    }

    /* Start multiplication */
    sap_num result; /* Result of the process. */
    sap_num expo0;  /* Replicated exponent for subtraction. */
    sap_num tmp1;   /* Temporary variable */

    /* Initialize variables. */
    int rscale = MAX(base->n_scale, scale);
    int cscale = rscale * sap_num2int(expo); /* It is easy to accumulate errors when a wrong scale is selected.
                                                Try to maximize the scale here. */
    int do_reverse = FALSE;
    result = _sap_replicate_num(base);
    expo0 = _sap_replicate_num(expo);

    if (sap_compare(expo, _zero_) < 0)
    {
        do_reverse = TRUE;
        expo0->n_sign = -expo0->n_sign;
    }

    /* Minus one first, for replicating the number to its place. */
    tmp1 = sap_sub(expo0, _one_, 0);
    sap_free_num(&expo0);
    expo0 = tmp1;
```

```c
    /* Since the first derivative of x^2 is positive and increasing in the first quadrant,
       we can evaluate this with newton's iteration. */
    /* From calculation we know that this iteration must work. */

    /* Major reference for calculating scales: See `number.c` in the source code of GNU bc. */
    int rscale = MAX(op->n_scale, scale);                        /* The final real scale used */
    int cscale = (sap_compare(op, _one_) > 0) ? 3 : op->n_scale; /* Current scale for arithmetic operations */

    int done = FALSE;

    sap_num cguess = NULL;   /* Current guess */
    sap_num nguess = NULL;   /* Next guess */
    sap_num diff = NULL;     /* For evaluating difference to control the precision */
    sap_num one_half = NULL; /* For division by 2 */
    sap_num tmp1 = NULL;
    sap_num tmp2 = NULL;

    one_half = sap_new_num(1, 1);
    one_half->n_val[1] = 5; /* Assign it +0.5 */

    /* Place the initial guess */
    cguess = sap_copy_num(_one_);

    while (!done)
    {
        // todo: fix
        tmp1 = sap_div(op, cguess, cscale);
        tmp2 = sap_add(tmp1, cguess, cscale);
        nguess = sap_mul(tmp2, one_half, cscale);
        diff = sap_sub(cguess, nguess, cscale + 1);
        if (sap_is_near_zero(diff, cscale))
        {
            if (cscale < rscale + 1)                      /* If the precision is not enough */
                cscale = MIN(cscale * 3, rscale + 1); /* Adjusting the precision. */
            else
                done = TRUE;
        }
        sap_free_num(&tmp1);
        sap_free_num(&tmp2);
        sap_free_num(&cguess);
        sap_free_num(&diff);
        cguess = nguess;
    }
    _sap_truncate(cguess, scale, TRUE);
    return cguess;
}
```

Part 2 - Code

```
├── include
|   ├── global.h
|   ├── lut.h
|   ├── number.h
|   ├── opt.h
|   ├── parser.h
|   ├── sapdefs.h
|   ├── sap.h
|   ├── stack.h
|   ├── test.h
|   └── utils.h
├── project2.pdf
├── readme.md
├── report.md
├── src
|   ├── lut.c
|   ├── main.c
|   ├── number.c
|   ├── parser.c
|   ├── sap.c
|   ├── stack.c
|   ├── test.c
|   └── utils.c
└── test
```

Header file for `number.h`

```
/* Header file for arbitrary precision number library */

#ifndef _NUMBER_H
#define _NUMBER_H


/* Definitions */
#ifdef TRUE
#undef TRUE
#endif
#define TRUE 1

#ifdef FALSE
#undef FALSE
#endif
#define FALSE 0

/* Defining the precision of transcendental functions */
#ifndef _TRANS_FUNC_PREC
#define _TRANS_FUNC_PREC 22
#endif

/* Struct declarations */

typedef enum
{
    POS = 255,
    NEG = -255
} sign;

typedef struct sap_struct *sap_num;

/* Struct for holding properties and pointers to storage of a sap_number */
typedef struct sap_struct
{
    sign n_sign;  /* For the sign of the number. To specify, zero has positive
sign. */
    int n_refs;   /* For counting how many references are pointed to this
number.
                     If 0, the structure will be appended to the available
resource list. */

    struct sap_struct *n_next; /* For storing the next node when in the
sap_free_list */

    int n_len;    /* For number of digits before the decimal point */
    int n_scale;  /* For number of digits after the decimal point */

    /* The structure of the storage is shown as follows:
        |---(MSB)---Integral---(LSB)---|---(MSB)---Fractional---(LSB)---|
      */
    char *n_ptr;  /* For internal storage. This may be NULL to indicate that
                     the value actually points to the storage in another number.
*/

    char *n_val;  /* For pointer to actual value. */
} sap_struct;


/* Global constants */

extern sap_num _zero_;
extern sap_num _one_;
extern sap_num _two_;
extern sap_num _e_;
extern sap_num _pi_;


/* Function prototypes */

void sap_init_number_lib(void);

sap_num sap_new_num(int length, int scale);

void sap_free_num(sap_num *op);

sap_num sap_copy_num(sap_num src);

void sap_init_num(sap_num *op);

sap_num sap_str2num(char *ptr);

sap_num sap_double2num(double val);

sap_num sap_int2num(int val);

char *sap_num2str(sap_num op);

double sap_num2double(sap_num op);

int sap_num2int(sap_num op);

int sap_is_zero(sap_num op);

int sap_is_near_zero(sap_num op, int scale);

int sap_is_neg(sap_num op);

int sap_compare(sap_num op1, sap_num op2);

void sap_negate(sap_num op);

sap_num sap_add(sap_num op1, sap_num op2, int scale_min);
sap_num sap_sub(sap_num op1, sap_num op2, int scale_min);

sap_num sap_mul(sap_num op1, sap_num op2, int scale);

sap_num sap_div(sap_num dividend, sap_num divisor, int scale);

sap_num sap_mod(sap_num dividend, sap_num divisor, int scale);

void sap_divmod(sap_num dividend, sap_num divisor, sap_num *quotient, sap_num
*remainder, int scale);

sap_num sap_sqrt(sap_num op, int scale);

sap_num sap_sin(sap_num op, int scale);

sap_num sap_cos(sap_num op, int scale);

sap_num sap_arctan(sap_num op, int scale);

sap_num sap_ln(sap_num op, int scale);

sap_num sap_raise(sap_num base, sap_num expo, int scale);

sap_num sap_exp(sap_num expo, int scale);

#endif
```

Header file for `parser.h`

```
/* Header file for declarations of parser functions */
#ifndef _PARSER_H
#define _PARSER_H


/* Include libraries */
#include "number.h"


/* Definitions */

/* Function token definitions */

#define _SAP_TEXT_FUNC_SIN    "sin"
#define _SAP_TEXT_FUNC_COS    "cos"
#define _SAP_TEXT_FUNC_SQRT   "sqrt"
#define _SAP_TEXT_FUNC_ARCTAN "atan"
#define _SAP_TEXT_FUNC_LN     "ln"
#define _SAP_TEXT_FUNC_EXP    "exp"


/* Enum declarations */

/* Defines token types for the parser. */
typedef enum sap_token_type
{
    _SAP_STACK_SENTINEL = -1, /* Sentinel of the tokens */
    _SAP_END_OF_STMT = 1,     /* End of statement */

    _SAP_LESS,    /* less */
    _SAP_GREATER, /* greater */
    _SAP_EQ,      /* Equal */
    _SAP_LEQ,     /* less or equal */
    _SAP_GEQ,     /* greater or equal */
    _SAP_ASSIGN,  /* Assignment */

    _SAP_ADD,      /* Add */
    _SAP_MINUS,    /* Minus */
    _SAP_MULTIPLY, /* Multiply */
    _SAP_DIVIDE,   /* Divide */
    _SAP_POWER,    /* Power */

    _SAP_SQRT,   /* Sqrt */
    _SAP_SIN,    /* Sin */
    _SAP_COS,    /* Cos */
    _SAP_ARCTAN, /* Arctangent */
    _SAP_LN,     /* Natural Logarithm */
    _SAP_EXP,    /* exp */

    _SAP_PAREN_L, /* Left parentheses */
    _SAP_PAREN_R, /* Right parentheses */

    _SAP_VAR_NAME, /* Variable name */
    _SAP_NUMBER,   /* Number */

    _SAP_FUNC_CALL /* Function calls */
} sap_token_type;


/* Struct declarations */

typedef struct sap_token_struct *sap_token;

/* Structure used to store the actual token. */
typedef struct sap_token_struct
{
    sap_token_type type; /* Type of this token. */
    char *name;          /* If it is a variable or function call, stores its
name. Else it is NULL. Must be copied when using. */
    sap_num val;         /* If it is a number, stores the value, and hen use
this value ensure that it is copied. Else it is NULL. */

    /* If it is a function, stores the array of tokens of arguments.
       Else, it is NULL. Array must end with _SAP_END_OF_STMT.
       The struct has complete control over the array, and _sap_free_token will
free the array when necessary. */
    struct sap_token_struct **arg_tokens;
} sap_token_struct;


/* Function prototypes */
```

```c
int sap_get_prec(sap_token_type type);

int sap_get_token_arr_length(sap_token *array);

sap_token *sap_parse_expr(char *src);

void sap_free_tokens(sap_token **array);


/* Debug function prototypes. They are forbidden to use in production
environments. */

char *_sap_debug_token2text(sap_token token);

#endif
```

Header file for `lut.h`

```c
/* Header file for the symbol lookup table. This library supports arbitrary
number of LUTs in the same instance of a program. */

#ifndef _LUT_H
#define _LUT_H

#include "number.h"
#include <stddef.h>


/* Definitions */

/* This shows that the implementation of LUT is ordered. */
#define _LUT_UNORDERED_IMPL

#define _LUT_DEFAULT_CAPACITY 1000


/* Struct declarations */

typedef struct lut_node_struct *lut_node;

/* Struct for holding properties of a node of linked list. */
typedef struct lut_node_struct
{
    char *key;                  /* The keyword for the entry */
    sap_num val;                /* The value for the entry */
    struct lut_node_struct *next; /* Point to the next element in case of Hash
Collision */
} lut_node_struct;

typedef struct lut_table_struct *lut_table;

/* Struct for holding properties of a hashtable. */
typedef struct lut_table_struct
{
    lut_node *entries; /* Pointer to the start of array of pointer to entries */
    size_t capacity;   /* Capacity of this lut_table */
} lut_table_struct;


/* Function prototypes */

lut_table lut_new_table(void);

void lut_free_table(lut_table *table);

sap_num lut_find(lut_table table, char *key);

void lut_insert(lut_table table, char *key, sap_num val);

void lut_delete(lut_table table, char *key);

void lut_reset_all(lut_table table);

#endif
```

`main.c`. Simplicity is preferred.

```
/*  This file is the main program of the Simple Arithmetic Program, acting
    as a bridge to connect the user to the interface of the SAP virtual machine.
 *******************************************************************/


#include "sapdefs.h"
#include "sap.h"
#include "global.h"
#include "utils.h"
#include "opt.h"

#define DEBUG

#ifdef DEBUG
#include "test.h"
#endif

/* Definition of constants */
int quiet = FALSE;
int debug = FALSE;

#define OPT_CNT 4
static option options[OPT_CNT] = {
    {'h', "help"},
    {'q', "quiet"},
    {'v', "version"},
    {'d', "debug"}};

static void
usage(const char *progname)
{
    printf("usage: %s [options] [file ...]\n%s%s%s", progname,
           "  -h  --help     print this usage and exit\n",
```

```c
           "  -q  --quiet    don't print initial banner\n",
           "  -v  --version  print version information and exit\n");
}

static void
show_version()
{
    printf("Simple Arithmetic Program, aka SAP. VERSION: " VERSION "\n"
           "Engineering sample. Interactive mode only.\n");
}

static void
show_instruction()
{
    printf("Enter \"quit\" to exit.\n");
}

static void
show_debug()
{
    printf("==========>CAUTION: DEBUG mode enabled.\n");
}

/* Process argument to apply the settings. */
static void
_process_arg_abbr(char arg, char *progname)
{
    switch (arg)
    {
    case 'h':
        usage(progname);
        exit(0);
    case 'q':
        quiet = TRUE;
        break;
    case 'v':
        show_version();
        exit(0);
    case 'd':
        show_debug();
        debug = TRUE;
    default:
        usage(progname);
        exit(1);
    }
}

static void
_parse_args_full(char *arg, char *progname)
{
    if (arg == NULL)
    {
        usage(progname);
        exit(1);
    }
    int index;
    for (index = 0; index < OPT_CNT; ++index)
        if (strcmp(options[index].full, arg) == 0)
            break;
    if (index == OPT_CNT)
    {
        usage(progname);
        exit(1);
    }
    _process_arg_abbr(options[index].abbr, progname);
}

/* Parse arguments from the command line. */
static void
parse_args(int argc, char **argv)
{
    int opt;
    char **p = argv;
    while (--argc > 0)
    {
        ++p;
        switch (opt = **p)
        {
        case '-':
            int c = *(++*p);
            if (c == '-')
            {
                char *arg = fetch_token(++*p);
                // parse arg
                _parse_args_full(arg, argv[0]);
            }
            else
                while (c != '\0')
                {
                    _process_arg_abbr(c, argv[0]);
                    c = *(++*p);
                }
            break;
        default:
            usage(argv[0]);
            exit(1);
        }
    }
}

int main(int argc, char **argv)
{
    /* Parse arguments first. */
    parse_args(argc, argv);
    if (quiet != TRUE)
    {
        show_version();
        show_instruction();
    }
    /* Init libraries */
    sap_init_lib();

    /* Start executing */
    sap_num result = NULL;
    char *buf = NULL;
    size_t size = 0;

    while (getline(&buf, &size, stdin) > 0)
    {
        if (strcmp(buf, "quit") == 0)
            exit(0);

        char **stmts = fetch_expr(buf); /* Will later be freed. */
        char **ptr = stmts;             /* Pointer to current statement */

        for (; *ptr != NULL; ptr++)
        {
            result = sap_execute(*ptr);
            printf("%s\n", sap_num2str(result));
            sap_free_num(&result);
        }

        /* Clean up */
        free_expr_array(&stmts);
        free(buf);
        buf = NULL;
    }
}
```

Part of the library `number.c`. The following excerpt illustrates how multiplication is done.

```c
/* Left shift the number in 10's base. Negative shift value indicates to do right
shift. */
static sap_num _sap_shift(sap_num op, int shift)
{
    int len = op->n_len;
    int scale = op->n_scale;

    if (shift == 0)
        return sap_copy_num(op);
    else if (shift < 0)
    {
        /* Process the length */
        shift = -shift;
        len -= shift;
        if (len <= 0)
            len = 1;
        scale += shift;

        /* Copying the number and return. */
        sap_num tmp = sap_new_num(len, scale);
        tmp->n_sign = op->n_sign;
        for (int i = 0; i < op->n_len + op->n_scale; ++i)
```

```
            *(tmp->n_val + tmp->n_len + tmp->n_scale - i - 1) = *(op->n_val +
op->n_len + op->n_scale - i - 1);
        return tmp;
    }
    else
    {
        /* Process the length */
        scale -= shift;
        if (scale <= 0)
            scale = 0;
        len += shift;

        /* Copying the number and return. */
        sap_num tmp = sap_new_num(len, scale);
        for (int i = 0; i < op->n_len + op->n_scale; ++i)
            *(tmp->n_val + i) = *(op->n_val + i);
        return tmp;
    }
}

/* Internal simple multiplication for handling small numbers. Both of the
operands are assumed positive integers. */
static sap_num _sap_simple_mul(sap_num op1, sap_num op2)
{
    int len = op1->n_len + op2->n_len;
    sap_num result = sap_new_num(len, 0);

    /* Simulate hand multiplication. */
    for (int i = 0; i < op2->n_len; ++i)
    {
        for (int j = 0; j < op1->n_len; ++j)
        {
            char *p = result->n_val + result->n_len - i - j - 1; /* The target
position */
            char *vp = op2->n_val + op2->n_len - i - 1;
            char *vs = op1->n_val + op1->n_len - j - 1;
            *p += *vp * *vs;
            if (*p >= 10)
            {
                *(p - 1) += *p / 10;
                *p %= 10;
            }
        }
    }
    _sap_normalize(result);
    return result;
}

/* Decompose a positive integer into the form x1 * B^m + x0.  (^ denotes power
here) */
static void _sap_karatsuba_decomp(sap_num op1, sap_num *x1, sap_num *x0, int m)
{
    int llen = op1->n_len - m; /* The length of x1. */
    *x1 = sap_new_num(llen, 0);
    *x0 = sap_new_num(m, 0);
```

```c
    memcpy((*x1)->n_val, op1->n_val, llen);
    memcpy((*x0)->n_val, op1->n_val + llen, m);
}

#define _KARATSUBA_THRESHOLD 2

/* Internal simple multiplication for recursive Karatsuba's multiplication
method.
   Both of the operands are assumed positive integers. */
static sap_num _sap_rec_mul(sap_num op1, sap_num op2)
{
    // todo: implement this recursive multiplication
    if (op1->n_len <= _KARATSUBA_THRESHOLD || op2->n_len <=
_KARATSUBA_THRESHOLD)
        return _sap_simple_mul(op1, op2);

    /* Karatsuba's method: x = x1*B^m + x0, y = y1*B^m + y0, xy = z2*B^(2m) + z1
* B^m + z0. */
    sap_num result, x1, x0, y1, y0, z2, z1, z0;
    int shift;

    shift = MIN(op1->n_len / 2, op2->n_len / 2);
    _sap_karatsuba_decomp(op1, &x1, &x0, shift);
    _sap_karatsuba_decomp(op2, &y1, &y0, shift);

    z2 = _sap_rec_mul(x1, y1);
    z0 = _sap_rec_mul(x0, y0);

    /* tmp1 = x1 + x0, tmp2 = y1 + y0, tmp3 = z2 + z0, tmp4 = tmp1 * tmp2 */
    /* z1 = tmp4 - tmp3 */
    sap_num tmp1, tmp2, tmp3, tmp4;
    tmp1 = sap_add(x1, x0, 0);
    tmp2 = sap_add(y1, y0, 0);
    tmp3 = sap_add(z2, z0, 0);
    tmp4 = _sap_rec_mul(tmp1, tmp2);
    z1 = sap_sub(tmp4, tmp3, 0);

    /* Free the intermediate variables first. */
    sap_free_num(&x1);
    sap_free_num(&x0);
    sap_free_num(&y1);
    sap_free_num(&y0);
    sap_free_num(&tmp1);
    sap_free_num(&tmp2);
    sap_free_num(&tmp3);
    sap_free_num(&tmp4);

    sap_num tmp5, tmp6, tmp7;
    tmp5 = _sap_shift(z2, shift * 2);
    tmp6 = _sap_shift(z1, shift);
    tmp7 = sap_add(tmp5, tmp6, 0);
    result = sap_add(tmp7, z0, 0);

    /* Then free the intermediate variables generated afterwards. */
    sap_free_num(&tmp5);
    sap_free_num(&tmp6);
    sap_free_num(&tmp7);
    sap_free_num(&z2);
    sap_free_num(&z1);
    sap_free_num(&z0);

    return result;
}

/* Internal implementation for multiplying two numbers. */
static sap_num _sap_mul_impl(sap_num op1, sap_num op2, int scale)
{
    sap_num tmp1, tmp2, result0, result;

    /* If the numbers have fractional parts, first convert them to integer, then
perform the multiplication. */
    tmp1 = _sap_shift(op1, op1->n_scale);
    tmp2 = _sap_shift(op2, op2->n_scale);
    result0 = _sap_rec_mul(tmp1, tmp2);
    result = _sap_shift(result0, -(op1->n_scale + op2->n_scale));
    _sap_truncate(result, scale, FALSE);
   /* Truncate the number (only the fractional part) to meet scale requirements.
*/
    result->n_sign = (op1->n_sign == POS) ? op2->n_sign : _sap_negate(op1-
>n_sign); /* Negate the sign when op1 is negative. */

    sap_free_num(&tmp1);
    sap_free_num(&tmp2);
    sap_free_num(&result0);
    return result;
}

/* Multiply two numbers. The fractional part will be truncated to the size. (Must
be larger than 0).
   Return a new number as the result. */
sap_num sap_mul(sap_num op1, sap_num op2, int scale)
{
    return _sap_mul_impl(op1, op2, scale);
}
```

`parser.c`: Parser for expressions:

```c
/* Internal implementation for parsing an expression to array of tokens. */
static sap_token *sap_parse_expr_impl(char *src)
{
    /* Perform initial allocation for array. */
    int len = _SAP_TOKEN_ARR_SIZE;
    sap_token *arr = (sap_token *)malloc(len * sizeof(sap_token));
    sap_token *newarr;    /* In case we need to realloc more memory. */
    sap_token *ptr = arr; /* Next available position. */

    if (arr == NULL)
        out_of_memory();

    sap_token next = NULL;
    do
    {
        /* If there is no available slot left */
        if (ptr - arr == len)
        {
            newarr = (sap_token *)realloc(arr, (len + _SAP_TOKEN_ARR_SIZE) *
sizeof(sap_token));
            if (newarr == NULL)
                out_of_memory();
            len += _SAP_TOKEN_ARR_SIZE;
            arr = newarr;
            ptr = arr + len;
        }

        /* Fetch next token */
        next = _sap_parse_next_token(&src);
        *ptr++ = next;
    } while (next->type != _SAP_END_OF_STMT);

    return arr;
}
```

Part 3 - Result and Verification

Test Case #1

```
Input: 1 * 2 + 3
Output: 5
```

Test Case #2

```
Input: sqrt(3)
Output: 1
```

Test Case #3

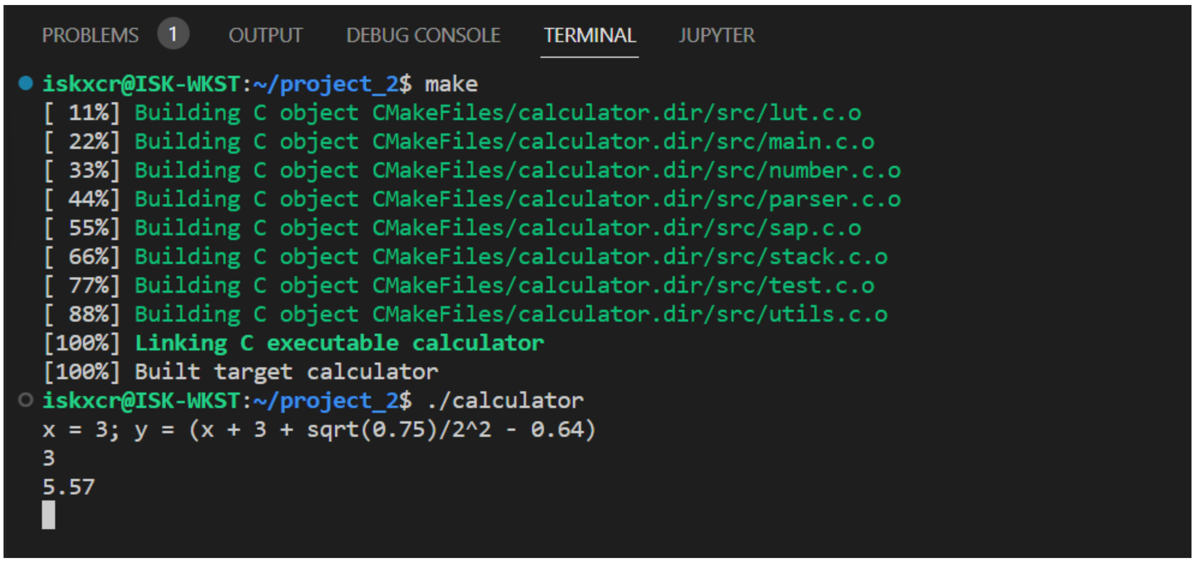
```
Input: sqrt(3.0)
Output: 1.7
```

Test Case #4

```
Input: x = 3.5; y = 7.5
Output: 3.5
7.5
```

Test Case #5

```
Input: x = 3; y = (x + 3 + sqrt(0.75)/2^2 - 0.64)
Output: 3
5.57
```



```
iskxcr@ISK-WKST:~/project_2$ make
[ 11%] Building C object CMakeFiles/calculator.dir/src/lut.c.o
[ 22%] Building C object CMakeFiles/calculator.dir/src/main.c.o
[ 33%] Building C object CMakeFiles/calculator.dir/src/number.c.o
[ 44%] Building C object CMakeFiles/calculator.dir/src/parser.c.o
[ 55%] Building C object CMakeFiles/calculator.dir/src/sap.c.o
[ 66%] Building C object CMakeFiles/calculator.dir/src/stack.c.o
[ 77%] Building C object CMakeFiles/calculator.dir/src/test.c.o
[ 88%] Building C object CMakeFiles/calculator.dir/src/utils.c.o
[100%] Linking C executable calculator
[100%] Built target calculator
iskxcr@ISK-WKST:~/project_2$ ./calculator
x = 3; y = (x + 3 + sqrt(0.75)/2^2 - 0.64)
3
5.57
```

Test Case #6

```
Input: ...(Too many long digits)
Output: (stderr) critical error: out of memory.\n
```

This is only supposed to happen on machines with little memory available.

Test Case #7

```
(Input in radians)
Input: sin(1.20)
Output: 0.93
```

Other test cases for different libraries are included in `test.c` *file, which contains some common test routines.* Call `test()` in the main function to see the result.

Part 4 - Difficulties & Solutions

1. How to ensure the precision of the functions?

   Using newton's iteration method or Taylor series the error must be estimated and correctly controlled. A good initial guess is required.

2. How to efficiently do multiplication, division, sqrt() and other math functions?

   Karatsuba's Algorithm, and other algorithm in Knuth's books. Since time is limited, only some relatively simple implementations are realized.

3. How to efficiently optimize simple operations in `number.c` such that those non-primitive operations (operations except for *addition* and *subtraction*) could be faster?

   Operate on small numbers, convert additions and subtractions to shifts may be faster.

4. How to efficiently and safely manage memory such that we can better reuse the resource already allocated and avoid frequent malloc()/realloc()/free() calls (which are very expensive operations) ?

   First, ***always write malloc() with free(), constructor with proper deconstructor***. Second, reuse small resources to avoid frequent call to memory management functions.

5. How to parse the expression efficiently?

Infix expression -> Postfix expression -> Stack Evaluation. Tokenization is performed on a custom structure, and we need to efficiently parse a token tree.

6. How to implement the lookup table (LUT) for variables ?

   Hash Table. Pick a good hash function is very important (see source code for detailed explanation on hash function chosen.)

7. How to write the library so that they can later on be reused without significant modification? Is it serious enough?

   Correctly define the header file. ***Write the header file with function prototypes first*** to ensure a clear big picture on how to write the entire library for use. Detailed explanations and comments are necessary. Also, API exposed to user must be simple and reasonable. (From my shallow experience. Welcome to correct any mistakes)

8. Keep the comments and documents in detail so that people *won't be confused* when examining the source code.

9. Dividing the program into multiple libraries and corresponding header files that contain *definitions for structures, constants and global variables* and *prototypes* for functions.

10. Hide the implementation detail from the caller to ensure that when API changes, no significant change needs to be made in the source library. Also, it would be easier to change the internal implementation afterwards ***without affecting the calling procedure***.

11. Possible concurrency: The original number won't be modified during operations (unless it is being processed by internal *refactor functions*). However, it is ***Undefined Bahavior*** to modify an argument during processing.

12. Multiple include files: possible nested duplicated includes. Solution: `#ifndef` and `#define_HEADER_H`.

13. How to handle exceptions during process if there is any? (***Speaking of pure C***)

    My approach: ***function pointers*** that process such exceptions.

14. Portability. Using `<ctype.h>` and other library routines to ensure portability of the program. (For example, application on a non ASCII coding for characters, though very rarely to see).

Arithmetic Implementations

Multiplication

Supports arbitrary precision

Karatsuba's Method

Division, Modulus and Sqrt(), raise, ln

Supports arbitrary precision

Simple iterative division, Newton's Iteration, Lookup Table. Recursive divide-and-conquer algorithms.

Sin, Cos, Exp

Supports limited precision

LUT (Lookup Table)/Math.h