实验题目：词法分析扫描器的设计实现

实验原理：

本次实验是针对 C 语言的词法进行分析，将源代码文件分割为一个个 Token，并输出 Token 的种类和位置。

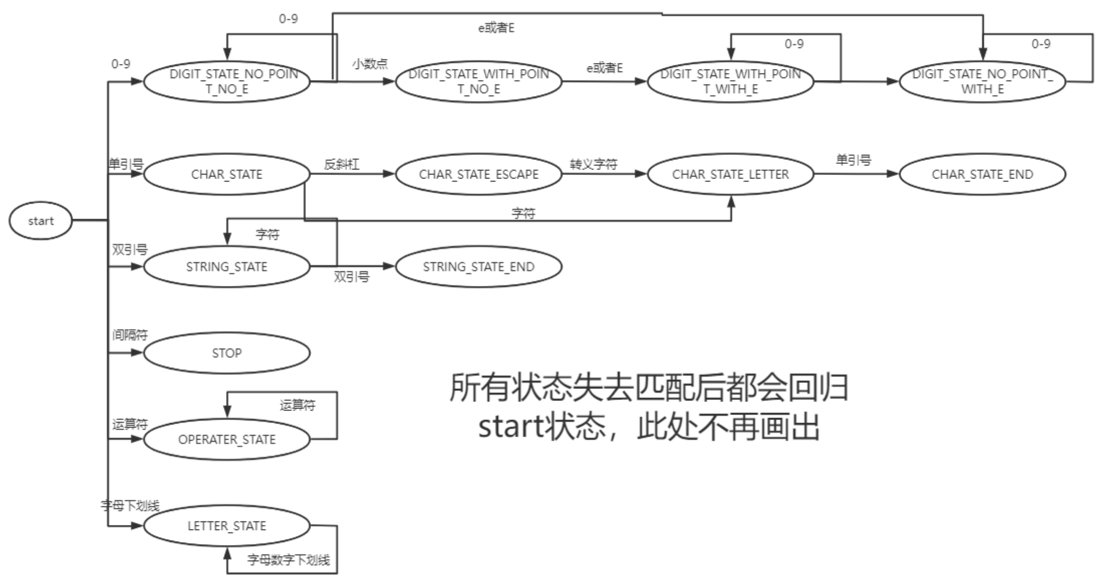
实验过程：

实现的核心是需要实现状态机与划分状态机的转换，在我的设计中，一共有 13 种状态，如下图中代码所示

```
//states
const (
    START = iota
    LETTER_STATE
    DIGIT_STATE_NO_POINT_NO_E
    DIGIT_STATE_WITH_POINT_NO_E
    DIGIT_STATE_WITH_POINT_WITH_E
    DIGIT_STATE_NO_POINT_WITH_E
    CHAR_STATE
    CHAR_STATE_ESCAPE
    CHAR_STATE_LETTER
    CHAR_STATE_END
    STRING_STATE
    STRING_STATE_END
    OPERATER_STATE
    ERROR
    STOP
)
```

分别对应初始态，四个和数字有关的状态，字符三个状态字符串两个状态，符号一个状态，语法错误一个状态，间隔符一个状态，相关状态说明体现在了命名规则中，不再描述。

其中，状态转换图如下



根据此图编写代码。

本实验中，使用了 golang 语言编写词法分析器

第一步：定义 TokenType 类型，包含了 C 语言的关键字，字符，预处理器，括号等等

```go
type TokenType int

const (
    // TokenType
    INCLUDE = iota
    DEFINE
    HEAD_FILE
    BLOCKCOMMENT
    LINECOMMENT
    IDENTIFIER
    STRING
    CHAR
    INT
    FLOAT
    DOUBLE
    VOID
    IF
    ELSE
    FOR
    WHILE
    RETURN
    BREAK
    CONTINUE
    LBRACE
    RBRACE
    LPAREN
    RPAREN
    LBRACKET
    RBRACKET
    SEMICOLON
    COMMA
    ASSIGN
    PLUS
    MINUS
    MUL
    DIV
    MOD
    EQ
    NEQ
    LT
    GT
    LEQ
    GEQ
    AND
    OR
    NOT
    EOF
    DO
    CONST
    STRUCT
    UNION
    ENUM
    TYPEDEF
    EXTERN
    STATIC
    AUTO
    REGISTER
    SIGNED
    UNSIGNED
    SHORT
    LONG
    PLUSASSIGN
    MINUSASSIGN
    MULASSIGN
    DIVASSIGN
    MODASSIGN
    ANDASSIGN
    ORASSIGN
    XORASSIGN
    LSHIFTASSIGN
    RSHIFTASSIGN
    BITAND
    BITOR
    BITXOR
    BITNOT
    BITLSHIFT
    BITRSHIFT
    MACRO
    NUMBER
    UNKNOWN
    BADTOKEN
)
```

第二步：将 C 语言中的运算符与关键字与这些类型对应起来，使用 map 存储

```
var keywords = map[string]TokenType{
    "include":    INCLUDE,
    "define":     DEFINE,
    "head_file":  HEAD_FILE,
    "identifier": IDENTIFIER,
    "string":     STRING,
    "char":       CHAR,
    "int":        INT,
    "float":      FLOAT,
    "double":     DOUBLE,
    "void":       VOID,
    "if":         IF,
    "else":       ELSE,
    "for":        FOR,
    "while":      WHILE,
    "return":     RETURN,
    "break":      BREAK,
    "continue":   CONTINUE,
    "do":         DO,
    "const":      CONST,
    "struct":     STRUCT,
    "union":      UNION,
    "enum":       ENUM,
    "typedef":    TYPEDEF,
```

```go
    "extern":     EXTERN,
    "static":     STATIC,
    "auto":       AUTO,
    "register":   REGISTER,
    "signed":     SIGNED,
    "unsigned":   UNSIGNED,
    "short":      SHORT,
    "long":       LONG,
}
var operaters = map[string]TokenType{
    "+=":  PLUSASSIGN,
    "-=":  MINUSASSIGN,
    "*=":  MULASSIGN,
    "/=":  DIVASSIGN,
    "%=":  MODASSIGN,
    "&=":  ANDASSIGN,
    "|=":  ORASSIGN,
    "^=":  XORASSIGN,
    "<<=": LSHIFTASSIGN,
    ">>=": RSHIFTASSIGN,
    "&":   BITAND,
    "|":   BITOR,
    "^":   BITXOR,
    "~":   BITNOT,
    "<<":  BITLSHIFT,
    ">>":  BITRSHIFT,
    "\"":  STRING,
    "'":   CHAR,
    "#":   MACRO,
    "{":   LBRACE,
    "}":   RBRACE,
    "(":   LPAREN,
    ")":   RPAREN,
    "[":   LBRACKET,
    "]":   RBRACKET,
    ";":   SEMICOLON,
    ",":   COMMA,
    "=":   ASSIGN,
    "+":   PLUS,
    "-":   MINUS,
    "*":   MUL,
    "/":   DIV,
    "%":   MOD,
    "==":  EQ,
    "!=":  NEQ,
    "<":   LT,
    ">":   GT,
    "<=":  LEQ,
    ">=":  GEQ,
    "&&":  AND,
    "||":  OR,
    "!":   NOT,
    "/*":  BLOCKCOMMENT,
    "//":  LINECOMMENT,
}
```

第三步：准备判断字符种类的辅助函数

```
var operatorChars = []byte{'+', '-', '*', '/', '%', '&', '|', '^', '~',
'<', '>', '=', '!', '?', ':', ',', '#', '"', '\'', '\\', '.'}

func isSpace(ch byte) bool {
    return ch == ' ' || ch == '\t' || ch == '\r' || ch == '\n'
}

func isDigit(ch byte) bool {
    return ch >= '0' && ch <= '9'
}

func isAlpha(ch byte) bool {
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')
}
func isEscape(ch byte) bool {
    return ch == '\\'
}
func isAlphaLodashNum(ch byte) bool {
    return isAlphaLodash(ch) || isDigit(ch)
}
func isAlphaLodash(ch byte) bool {
    return isAlpha(ch) || ch == '_'
}

func isStringQuote(ch byte) bool {
    return ch == '"'
}
func isCharQuote(ch byte) bool {
    return ch == '\''
}
func isOperaterChar(ch byte) bool {
```

```go
    for _, c := range operatorChars {
        if c == ch {
            return true
        }
    }
    return false
}

func isOperaterString(s string) bool {
    _, ok := operaters[s]
    return ok
}

func iskeyword(s string) bool {
    _, ok := keywords[s]
    return ok
}
func isStop(ch byte) bool {
    return ch == ',' || ch == ';' || ch == '{' || ch == '}' || ch == '('
|| ch == ')' || ch == '[' || ch == ']'
}
```

第四步：读取 C 语言源文件，初始化数据

```go
func main() {
    //read a cpp file
    data, err := ioutil.ReadFile("../demo.c")
    if err != nil {
        panic(err)
    }
    //init
    var result []Token
    fmt.Println(string(data))
    line, cur := 1, -1
    state := START
    cur_string := ""
```

第五步：开始状态机的匹配步骤，按照上文图中的状态机转换表编写状态机转换代码。

```go
for i := 0; i < len(data); i++ {
        ch := data[i]
        cur++
        if ch == '\n' {
            line++
            cur = -1
        }
        if DEBUG {
            fmt.Print("now char is ", string(ch), "\t")
            fmt.Print("cur_string is ", cur_string, "\t")
            fmt.Println("state is ", stateStrings[state], "\t")
        }
        //state machine
        switch {

        case state == START:
            switch {
            case isSpace(ch):
            case isDigit(ch):
                cur_string += string(ch)
                state = DIGIT_STATE_NO_POINT_NO_E
            case isAlpha(ch):
                cur_string += string(ch)
                state = LETTER_STATE
            case isStringQuote(ch):
                cur_string += string(ch)
                state = STRING_STATE
            case isCharQuote(ch):
                cur_string += string(ch)
                state = CHAR_STATE
            case isOperaterChar(ch):
                cur_string += string(ch)
                state = OPERATER_STATE
            case isStop(ch):
                cur_string += string(ch)
                state = STOP
            }
        case state == DIGIT_STATE_NO_POINT_NO_E:
            switch {
            case isDigit(ch):
                cur_string += string(ch)
            case ch == 'E' || ch == 'e':
                cur_string += string(ch)
                state = DIGIT_STATE_NO_POINT_WITH_E
            case ch == '.':
                cur_string += string(ch)
                state = DIGIT_STATE_WITH_POINT_NO_E
            default:
                result = append(result, Token{NUMBER, cur_string, line, 
cur})
                restart(&state, &cur_string, &i, ch, &cur)
            }
        case state == DIGIT_STATE_WITH_POINT_NO_E:
            switch {
            case isDigit(ch):
                cur_string += string(ch)
            case ch == 'E' || ch == 'e':
                cur_string += string(ch)
                state = DIGIT_STATE_WITH_POINT_WITH_E
            default:
                result = append(result, Token{NUMBER, cur_string, line, 
cur})
                restart(&state, &cur_string, &i, ch, &cur)
            }
        case state == DIGIT_STATE_WITH_POINT_WITH_E:
            switch {
            case isDigit(ch):
                cur_string += string(ch)
            default:
                result = append(result, Token{NUMBER, cur_string, line, 
cur})
                restart(&state, &cur_string, &i, ch, &cur)
            }
        case state == LETTER_STATE:
            switch {
            case isAlphaLodashNum(ch):
                cur_string += string(ch)
            default:
                if iskeyword(cur_string) {
                    result = append(result, Token{keywords[cur_string], 
cur_string, line, cur})
                } else {
                    result = append(result, Token{IDENTIFIER, 
cur_string, line, cur})
                }
                restart(&state, &cur_string, &i, ch, &cur)
            }
        case state == STRING_STATE:
            switch {
            case isStringQuote(ch):
                cur_string += string(ch)
```

```go
                state = STRING_STATE_END
            default:
                cur_string += string(ch)
            }
        case state == CHAR_STATE:
            switch {
            case isEscape(ch):
                cur_string += string(ch)
                state = CHAR_STATE_ESCAPE
            default:
                cur_string += string(ch)
                state = CHAR_STATE_LETTER
            }
        case state == CHAR_STATE_ESCAPE:
            cur_string += string(ch)
            state = CHAR_STATE_LETTER
        case state == CHAR_STATE_LETTER:
            switch {
            case isCharQuote(ch):
                cur_string += string(ch)
                state = CHAR_STATE_END
            default:
                state = ERROR
            }
        case state == CHAR_STATE_END:
            result = append(result, Token{CHAR, cur_string, line, cur})
            restart(&state, &cur_string, &i, ch, &cur)
        case state == OPERATER_STATE:
            switch {
            case isOperaterChar(ch):
                cur_string += string(ch)
                state = OPERATER_STATE
            default:
                if isOperaterString(cur_string) {
                    result = append(result, Token{operaters[cur_string],
cur_string, line, cur})
                } else {
                    result = append(result, Token{UNKNOWN, cur_string,
line, cur})
                }
                restart(&state, &cur_string, &i, ch, &cur)
            }
        case state == ERROR:
            result = append(result, Token{BADTOKEN, cur_string, line,
cur})
            restart(&state, &cur_string, &i, ch, &cur)
        case state == STOP:
            result = append(result, Token{operaters[cur_string],
cur_string, line, cur})
            restart(&state, &cur_string, &i, ch, &cur)
        case state == STRING_STATE_END:
            result = append(result, Token{STRING, cur_string, line,
cur})
            restart(&state, &cur_string, &i, ch, &cur)
        }

    }
```

其中，匹配失败后的重新开始函数为

```go
func restart(state *int, cur_string *string, i *int, ch byte, cur *int)
{
    if ch == ' ' || ch == '\t' || ch == '\r' || ch == '\n' {
        *cur_string = ""
    } else {
        *cur_string = ""
        *i = *i - 1
        *cur = *cur - 1
    }
    *state = START
}
```

在一定的情况下，需要将 i 下标前移，防止错过有用的字符。

第六步：输出结果

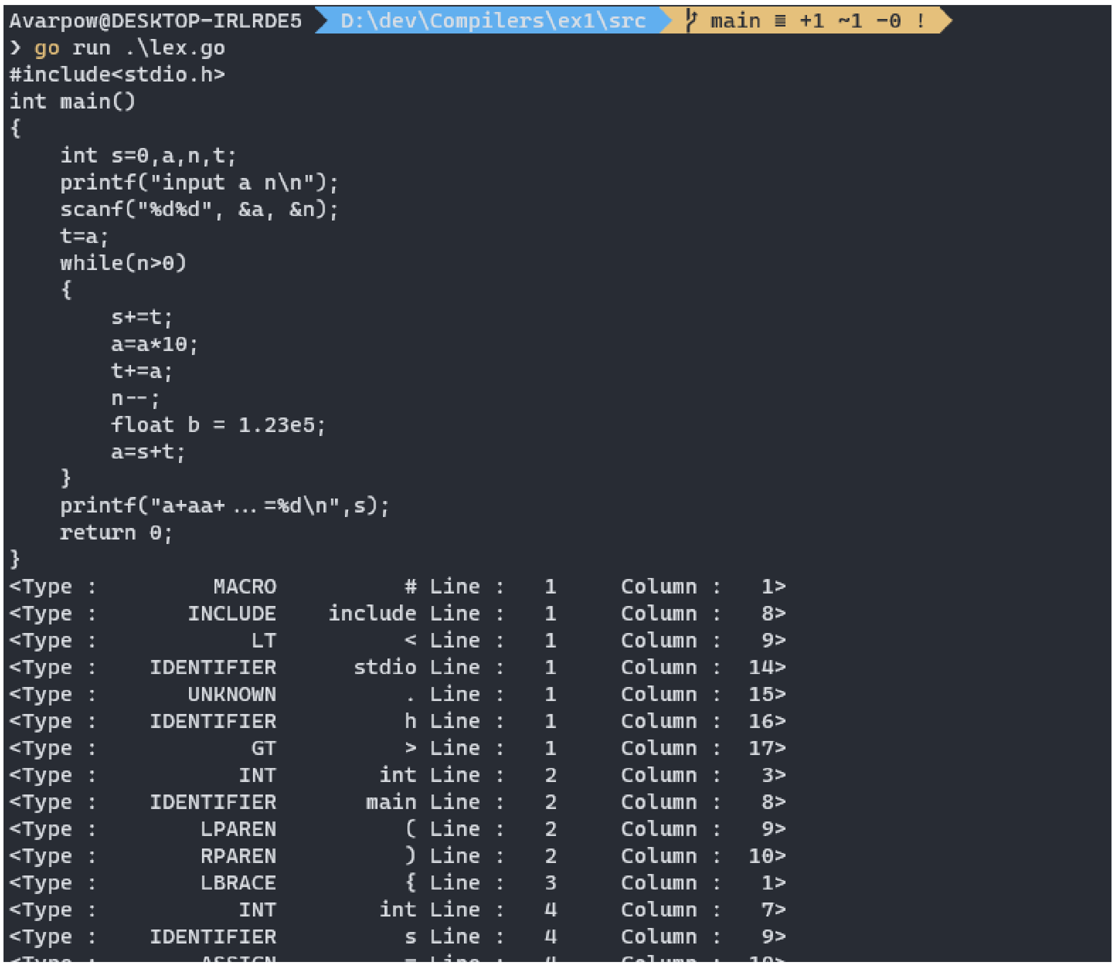
```go
    for _, i := range result {
        fmt.Print(i.String())
    }
    fmt.Printf("Lexical finish get %d tokens\n", len(result))
    //print result to tokens.txt
    file, err := os.Create("tokens.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()
    for _, i := range result {
        file.WriteString(i.String())
    }
```

实验结果：

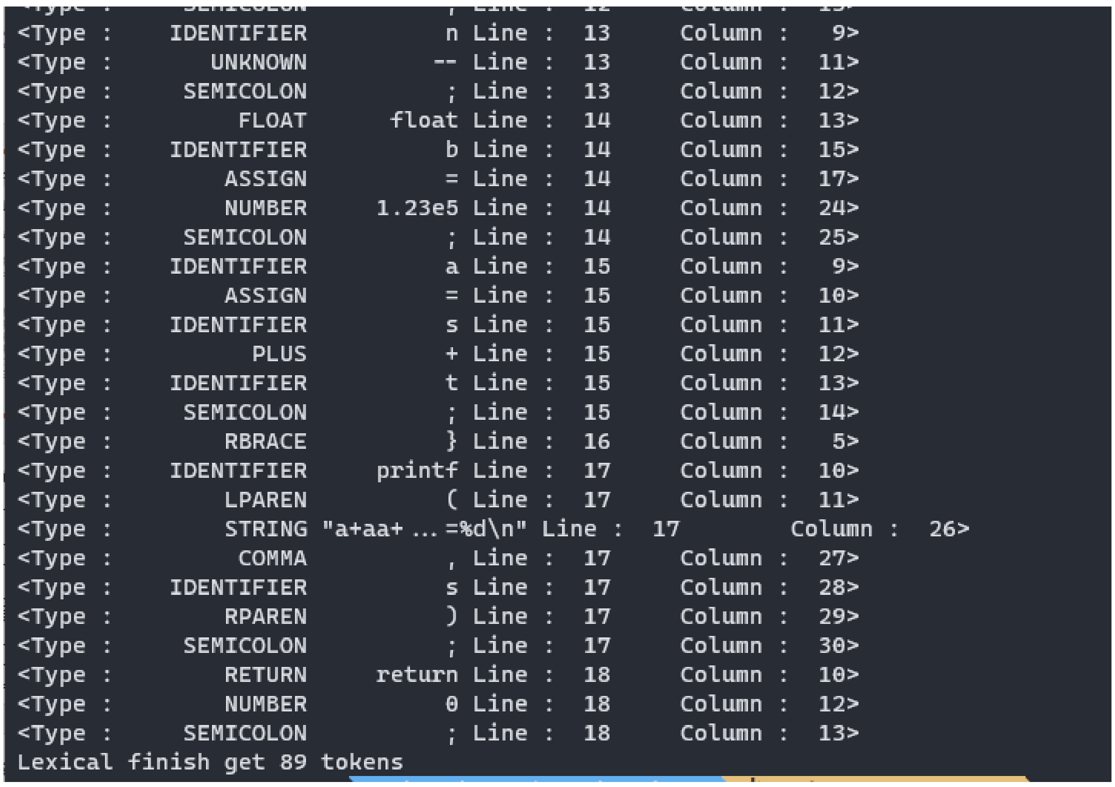
C 语言源程序：

```
#include<stdio.h>
int main()
{
    int s=0,a,n,t;
    printf("input a n\n");
    scanf("%d%d", &a, &n);
    t=a;
    while(n>0)
    {
        s+=t;
        a=a*10;
        t+=a;
        n--;
        float b = 1.23e5;
        a=s+t;
    }
    printf("a+aa+...=%d\n",s);
    return 0;
}
```

运行结果



```
Avarpow@DESKTOP-IRLRDE5  D:\dev\Compilers\ex1\src  ᚴ main ≡ +1 ~1 -0 !
❯ go run .\lex.go
#include<stdio.h>
int main()
{
    int s=0,a,n,t;
    printf("input a n\n");
    scanf("%d%d", &a, &n);
    t=a;
    while(n>0)
    {
        s+=t;
        a=a*10;
        t+=a;
        n--;
        float b = 1.23e5;
        a=s+t;
    }
    printf("a+aa+ ... =%d\n",s);
    return 0;
}
<Type :         MACRO          # Line :   1     Column :   1>
<Type :       INCLUDE    include Line :   1     Column :   8>
<Type :            LT          < Line :   1     Column :   9>
<Type :    IDENTIFIER      stdio Line :   1     Column :  14>
<Type :       UNKNOWN          . Line :   1     Column :  15>
<Type :    IDENTIFIER          h Line :   1     Column :  16>
<Type :            GT          > Line :   1     Column :  17>
<Type :           INT        int Line :   2     Column :   3>
<Type :    IDENTIFIER       main Line :   2     Column :   8>
<Type :        LPAREN          ( Line :   2     Column :   9>
<Type :        RPAREN          ) Line :   2     Column :  10>
<Type :        LBRACE          { Line :   3     Column :   1>
<Type :           INT        int Line :   4     Column :   7>
<Type :    IDENTIFIER          s Line :   4     Column :   9>
<Type :        ASSIGN          = Line :   4     Column :  10>
```



```
<Type :     SEMICOLON          ; Line :  12     Column :  13>
<Type :    IDENTIFIER          n Line :  13     Column :   9>
<Type :       UNKNOWN         -- Line :  13     Column :  11>
<Type :     SEMICOLON          ; Line :  13     Column :  12>
<Type :         FLOAT      float Line :  14     Column :  13>
<Type :    IDENTIFIER          b Line :  14     Column :  15>
<Type :        ASSIGN          = Line :  14     Column :  17>
<Type :        NUMBER     1.23e5 Line :  14     Column :  24>
<Type :     SEMICOLON          ; Line :  14     Column :  25>
<Type :    IDENTIFIER          a Line :  15     Column :   9>
<Type :        ASSIGN          = Line :  15     Column :  10>
<Type :    IDENTIFIER          s Line :  15     Column :  11>
<Type :          PLUS          + Line :  15     Column :  12>
<Type :    IDENTIFIER          t Line :  15     Column :  13>
<Type :     SEMICOLON          ; Line :  15     Column :  14>
<Type :        RBRACE          } Line :  16     Column :   5>
<Type :    IDENTIFIER     printf Line :  17     Column :  10>
<Type :        LPAREN          ( Line :  17     Column :  11>
<Type :        STRING "a+aa+ ... =%d\n" Line :  17          Column :  26>
<Type :         COMMA          , Line :  17     Column :  27>
<Type :    IDENTIFIER          s Line :  17     Column :  28>
<Type :        RPAREN          ) Line :  17     Column :  29>
<Type :     SEMICOLON          ; Line :  17     Column :  30>
<Type :        RETURN     return Line :  18     Column :  10>
<Type :        NUMBER          0 Line :  18     Column :  12>
<Type :     SEMICOLON          ; Line :  18     Column :  13>
Lexical finish get 89 tokens
```

通过转换，我们得到了 C 语言源文件中的每个 Token 的种类，原文，行数和

位置，该文件中，共有 89 个 Token，其中针对字符串，字符，自增，带有小数点的科学计数法数字都能良好的识别,达到了实验的要求。

心得体会:

本次实验是编译器最基础的一次实验，难点在于如何设计状态机使得字符串可以被合理的拆分，在理清楚状态机的结构后，看图说话编写状态机的代码，学会了如何将字符串切分为 Token，为将来进一步完成编译器打下了基础，同时这也是我第一次使用 golang 语言编写较为复杂的程序，通过和 C++语言的对比，golang 语言有编译速度快，编写方便的好处，今后的实验也将继续使用 golang 语言完成。

我的 github 地址为

https://github.com/avarpow/SYSU_Compilers