Lab1 词法分析

需要阅读的代码：

- **include/symtab.h**:定义了符号表相关类，本次实验你只需要根据注释提示阅读一小段即可
- **utils/symtab.cc**:符号表相关类的具体实现
- **utils/lexer_out.cc**:定义了词法分析结果的输出函数
- **target/main.cc**:主函数，你需要了解框架的整体流程以及全局变量，后续每次实验都需要进行阅读或者编写
- **parser/SysY_parser.y**:只需要阅读开头%token的定义即可，你在词法分析中需要return的枚举类型均来自于该文件开头定义的%token

需要阅读并编写的代码:

- **lexer/SysY_lexer.l**:编写你想实现的词法正则表达式及对应处理函数

Codes Reading

utils/lexer_out.cc

`ALIGNED_FORMAT_OUTPUT_HEAD(STR, CISU, PROP, STR3, STR4)` 定义了一个函数，功能是：输出表头，也就是output中最上面一行的东西

`ALIGNED_FORMAT_OUTPUT(STR, CISU, PROP)` 就是对应输出后面的内容，后面两项就是 输出 `line_number` 和 `cur_col_number`，也就是我们的行号和列号

```
extern int line_number;
extern int col_number;
extern int cur_col_number;
```

然后定义了三个外部变量，分别为行号、初始列号和修改后的列号

```
extern int yylex();  //下一个识别到的字符串
extern YYSTYPE yylval;  //定义了当前识别出的字符串的值
extern char *yytext;  //存储当前识别的词法单元
```

定义了yylex，yylval和yytext

```
void PrintLexerResult(std::ostream &s, char *yytext, YYSTYPE yylval, int token) {
    std::setfill(' ');
    switch (token) {
    case INT:
        ALIGNED_FORMAT_OUTPUT("INT", yytext, "");
        break;
    case INT_CONST:
        ALIGNED_FORMAT_OUTPUT("INT_CONST", yytext, yylval.int_token);
        break;
    case FLOAT:
        ALIGNED_FORMAT_OUTPUT("FLOAT", yytext, "");
        break;
```

```
    case STR_CONST:
        ALIGNED_FORMAT_OUTPUT("STR_CONST", yytext, yylval.symbol_token-
>get_string());
        break;
    case ERROR:
        ALIGNED_FORMAT_OUTPUT("ERROR", yytext, yylval.error_msg);
        break;
```

然后，后面就是定义了我们的词法分析输出的print函数，也就是输出函数。参数为yytext，yylval和token

举几个例子，识别到INT_CONST的情况，我们就调用ALIGNED_FORMAT_OUTPUT函数，输出对应的类型，yytext的文本和对应转化为十进制的值

STR_CONST的情况，我们就输出yytext文本值，然后去获得string的值

ERROR的情况，我们就输出yytext和报错信息，存储在yylval.error_msg里

case的值为int，在sysy_parser.tab.h中定义了yytokentype

```
  enum yytokentype
  {
    YYEMPTY = -2,
    YYEOF = 0,                     /* "end of file"  */
    YYerror = 256,                 /* error  */
    YYUNDEF = 257,                 /* "invalid token"  */
    STR_CONST = 258,               /* STR_CONST  */
    IDENT = 259,                   /* IDENT  */
    FLOAT_CONST = 260,             /* FLOAT_CONST  */
    INT_CONST = 261,               /* INT_CONST  */
    LEQ = 262,                     /* LEQ  */
    GEQ = 263,                     /* GEQ  */
    EQ = 264,                      /* EQ  */
    NE = 265,                      /* NE  */
    AND = 266,                     /* AND  */
    OR = 267,                      /* OR  */
    CONST = 268,                   /* CONST  */
    IF = 269,                      /* IF  */
    ELSE = 270,                    /* ELSE  */
    WHILE = 271,                   /* WHILE  */
    NONE_TYPE = 272,               /* NONE_TYPE  */
    INT = 273,                     /* INT  */
    FLOAT = 274,                   /* FLOAT  */
    FOR = 275,                     /* FOR  */
    RETURN = 276,                  /* RETURN  */
    BREAK = 277,                   /* BREAK  */
    CONTINUE = 278,                /* CONTINUE  */
    ERROR = 279,                   /* ERROR  */
    TODO = 280,                    /* TODO  */
    THEN = 281,                     /* THEN  */
    ADDASSIGN = 282,
    SUBASSIGN = 283,
    MULASSIGN = 284,
    DIVASSIGN = 285,
    MODASSIGN = 286
```

```
};
```

target/main.cc

该部分为主函数

前面两个函数跟out文件中的是一样的，就不说了

```
extern LLVMIR llvmIR;
extern Program ast_root;
extern FILE *yyin;
extern int error_num;
int line_number = 0;
int col_number = 0;
int cur_col_number = 0;
std::ofstream fout;
IdTable id_table;
extern int yylex();
extern YYSTYPE yylval;
extern char *yytext;
extern std::vector<std::string> error_msgs;
void PrintLexerResult(std::ostream &s, char *yytext, YYSTYPE yylval, int token);
```

全局变量llvmIR，存储与IR有关的代码；ast_root存储抽象语法树的根节点；yyin是一个指向 `FILE` 结构的指针，通常用于指定词法分析器的输入流；然后初始化了三个变量，分别为行号，初始列号和修改后的列号，均为0。定义id符号表，定义yylex，yylval和yytext，error_msgs。然后说明了lexer的输出函数，前面已经声明过了

```
enum Target { ARMV7 = 1, RV64GC = 2 } target;
bool optimize_flag = false;
```

第一行，表示我们选用的目标架构，此处我们选择2

第二行，表示我们是否启用优化，此处目前不作考虑

```
int main(int argc, char **argv) {
    target = RV64GC;

    FILE *fin = fopen(argv[file_in], "r");
    if (fin == NULL) {
        std::cerr << "Could not open input file " << argv[file_in] << std::endl;
        exit(1);
    }
    yyin = fin;
    fout.open(argv[file_out]);
    line_number = 1;

    if (strcmp(argv[step_tag], "-lexer") == 0) {
        int token;
        ALIGNED_FORMAT_OUTPUT_HEAD("Token", "Lexeme", "Property", "Line",
"Column");
        while ((token = yylex()) != 0) {
```

```
            PrintLexerResult(fout, yytext, yylval, token);
        }
        fout.close();
        return 0;
    }
    yyparse();

    if (error_num > 0) {
        fout << "Parser error" << std::endl;
        fout.close();
        return 0;
    }

    if (strcmp(argv[step_tag], "-parser") == 0) {

        ast_root->printAST(fout, 0);
        fout.close();
        return 0;
    }

    ast_root->TypeCheck();
    if (error_msgs.size() > 0) {
        for (auto msg : error_msgs) {
            fout << msg << std::endl;
        }
        fout.close();
        return 0;
    }

    if (strcmp(argv[step_tag], "-semant") == 0) {
        ast_root->printAST(fout, 0);
        return 0;
    }

    ast_root->codeIR();

    optimize_flag = (argc == 6 && (strcmp(argv[optimize_tag], "-O1") == 0));

    if (optimize_flag) {
        DomAnalysis dom(&llvmIR);
        dom.Execute();    // 对于AnalysisPass后续应该由TransformPass更新信息，维护
Analysis的正确性
        (Mem2RegPass(&llvmIR, &dom)).Execute();

        // TODO: add more passes
    }

    if (strcmp(argv[step_tag], "-llvm") == 0) {
        llvmIR.printIR(fout);
        fout.close();
        return 0;
    }
    if (strcmp(argv[step_tag], "-S") == 0) {
        MachineUnit *m_unit = new RiscV64Unit();
        RiscV64RegisterAllocTools regs;
        RiscV64Spiller spiller;
```

```
        RiscV64Selector(m_unit, &llvmIR).SelectInstructionAndBuildCFG();
        RiscV64LowerFrame(m_unit).Execute();

        FastLinearScan(m_unit, &regs, &spiller).Execute();
        RiscV64LowerStack(m_unit).Execute();

        RiscV64Printer(fout, m_unit).emit();
    }
    if (strcmp(argv[step_tag], "-select") == 0) {
        MachineUnit *m_unit = new RiscV64Unit();

        RiscV64Selector(m_unit, &llvmIR).SelectInstructionAndBuildCFG();
        RiscV64LowerFrame(m_unit).Execute();

        RiscV64Printer(fout, m_unit).emit();
    }
    fout.close();
    return 0;
}
```

main函数，主要完成我们的编译任务

首先，选择架构为riscv，读取目标文件，如果fin为空指针，也就是没有读取到对应的文件，那么报出`Could not open input file`

然后将读取到的文件赋值给yyin，表示输入，打开编译器准备编译，初始化行号为1

此处我们只需要看，如果和lexer匹配上了，就是进入词法分析环节

直接输出第一行，然后打印词法分析的结果即可

parser/SysY_parser.y

```
//declare the terminals
%token <symbol_token> STR_CONST IDENT
%token <float_token> FLOAT_CONST
%token <int_token> INT_CONST
%token LEQ GEQ EQ NE // <=   >=   ==   !=
%token AND OR // &&    ||
%token CONST IF ELSE WHILE NONE_TYPE INT FLOAT FOR
%token RETURN BREAK CONTINUE ERROR TODO
```

定义了终结符，包含了很多类型

均在lexer.l中实现了

sysy_parser.tab.h中定义了YYSTYPE和YYLTYPE

```
union YYSTYPE
{
```

```
#line 14 "parser/SysY_parser.y"

    char* error_msg;
    Symbol symbol_token;
    double float_token; // 对于SysY的浮点常量，我们需要先以double类型计算，再在语法树节点
创建的时候转为float
    int int_token;
    Program program;
    CompUnit comp_unit;  std::vector<CompUnit>* comps;
    Decl decl;
    Def def;  std::vector<Def>* defs;
    FuncDef func_def;
    Expression expression;  std::vector<Expression>* expressions;
    Stmt stmt;
    Block block;
    InitVal initval;  std::vector<InitVal>* initvals;
    FuncFParam formal;   std::vector<FuncFParam>* formals;
    BlockItem block_item;   std::vector<BlockItem>* block_items;

#line 108 "SysY_parser.tab.h"

};

extern YYSTYPE yylval;
```

yylval就是YYSTYPE类型的，可以有int型，double型等等

```
typedef struct YYLTYPE YYLTYPE;
struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
};

extern YYLTYPE yylloc;
```

Questions

你是如何解析-2147483648 这个语句的。结合代码简要讲解

lexer应该是将其分为-和2147483648进行识别，虽然该大负数没有超出int的范围，但是该正数超出了范围，会溢出，输出的值就不对了

但是问题不大，我们拿他做一下说明

我们需要识别其为-2147483648，但是该数为

对于int 表示范围应该为-2147483648（1000...000 1+31个0）~2147483647（0+31个1）

对于unsigned_int表示范围为0~2147483647（0+31个1） ~2147483648（1+31个0） ~4,294,967,295（32个1）

我们的yytval.int_const是一个int类型的量 对于2147483648会溢出成负的 但其实存的是1+31个0

在具体计算值过程中，10000000 00000000 00000000 00000000 会遇上前面的负号

也就是会全部取反+1 变成 01111111 11111111 11111111 11111111+1 还是为10000000 00000000 00000000 00000000 ，这是一个int值，会被识别为负的，所以这时候正好 识别为-2147483648，是正确值

你在词法分析中是否使用到了yylval,yytext 变量，简要说明它们的类型及作用

使用了yytext和yylval变量。

yylval类型在sysy_parser.tab.h中定义了

```
union YYSTYPE
{
#line 14 "parser/SysY_parser.y"

    char* error_msg;
    Symbol symbol_token;
    double float_token; // 对于SysY的浮点常量，我们需要先以double类型计算，再在语法树节点创建的时候转为float
    int int_token;
    Program program;
    CompUnit comp_unit;  std::vector<CompUnit>* comps;
    Decl decl;
    Def def;  std::vector<Def>* defs;
    FuncDef func_def;
    Expression expression;  std::vector<Expression>* expressions;
    Stmt stmt;
    Block block;
    InitVal initval;  std::vector<InitVal>* initvals;
    FuncFParam formal;   std::vector<FuncFParam>* formals;
    BlockItem block_item;   std::vector<BlockItem>* block_items;

#line 108 "SysY_parser.tab.h"

};

extern YYSTYPE yylval;
```

yylval的作用就是定义了当前识别出的字符串的值，可以是int型，也可以是其他类型，在YYSTYPE中均得到了定义

比如说识别十六进制的时候，我们就需要把读取到的十六进制转为十进制的int型，然后输出int值，就是存储在yylval中完成的

yylval.int_token、yylval.symbol_token、yylval.error_msg等等

yytext类型`char *yytext`，是一个指向数组的指针，yytext读取到的就是我们的字符串

```
[\.\+\-\*\/\=\<\!\%\>] {
    cur_col_number=col_number;
    col_number=col_number+strlen(yytext);
    return yytext[0];
    }
```

比如说这个，我们就返回了yytext数组的第一个值

你在词法分析中return的枚举类型是在哪被定义的，这些return的返回值在哪里被使用到了。

定义的位置在SysY_parser.tab.h

返回值在yytokentype中被识别为了数字

```
  enum yytokentype
  {
    YYEMPTY = -2,
    YYEOF = 0,                     /* "end of file"  */
    YYerror = 256,                 /* error  */
    YYUNDEF = 257,                 /* "invalid token"  */
    STR_CONST = 258,               /* STR_CONST  */
    IDENT = 259,                   /* IDENT  */
    FLOAT_CONST = 260,             /* FLOAT_CONST  */
    INT_CONST = 261,               /* INT_CONST  */
    LEQ = 262,                     /* LEQ  */
    GEQ = 263,                     /* GEQ  */
    EQ = 264,                      /* EQ  */
    NE = 265,                      /* NE  */
    AND = 266,                     /* AND  */
    OR = 267,                      /* OR  */
    CONST = 268,                   /* CONST  */
    IF = 269,                      /* IF  */
    ELSE = 270,                    /* ELSE  */
    WHILE = 271,                   /* WHILE  */
    NONE_TYPE = 272,               /* NONE_TYPE  */
    INT = 273,                     /* INT  */
    FLOAT = 274,                   /* FLOAT  */
    FOR = 275,                     /* FOR  */
    RETURN = 276,                  /* RETURN  */
    BREAK = 277,                   /* BREAK  */
    CONTINUE = 278,                /* CONTINUE  */
    ERROR = 279,                   /* ERROR  */
    TODO = 280,                    /* TODO  */
    THEN = 281,                     /* THEN  */
    ADDASSIGN = 282,
    SUBASSIGN = 283,
    MULASSIGN = 284,
    DIVASSIGN = 285,
```

```
    MODASSIGN = 286

};
```

用于去匹配我们的token，再进行相应的输出即可

路径：return给yylex，yylex在YY_DECL中

把类型给token

token作为out函数的参数，对应去识别switch

再去打印每一行

以 0x.AP-3 为例描述十六进制浮点数格式，并说明你如何处理该浮点常量的。

识别到P，然后分为0x.A和-3，分别进行识别

按照程序走即可

关键字和标识符的处理可不可以在SysY_lexer.l 文件中交换书写顺序，说明理由。

关键字和标识符的书写顺序在词法分析中是有影响的，因为lexer按照它们出现的顺序进行匹配。通常，关键字应该在标识符之前定义，因为关键字是特殊的标识符，它们有特定的意义，而普通的标识符则没有。

如果你先定义了标识符，然后定义了关键字，那么当词法分析器遇到一个词时，它可能会错误地将其识别为标识符而不是关键字，因为标识符的规则会先匹配到这个词。

框架或者你是如何将输入文本传给flex进行处理的，用到了什么变量或者函数。

怎么处理?

match匹配，匹配完之后，去找对应的action，顺便更新状态，然后do action，里面使用了switch匹配规则

具体是如何处理的?

`while ( /*CONSTCOND*/1 )`表示，进入了yylex读取字符的循环

yy_cp保存了yy_c_buf_p，yy_c_buf_p为当前字符缓冲区的值

将yy_hold_char赋值给了yy_cp指向的字符，yy_hold_char为上一个识别出的字符串

将`yy_cp`的值赋给`yy_bp`。`yy_bp`指向当前运行的开始位置。

`yy_start`是扫描器的初始状态，`yy_current_state`用于跟踪扫描器的当前状态

```
    while ( /*CONSTCOND*/1 )        /* loops until end-of-file is reached */
        {
        yy_cp = (yy_c_buf_p);

        /* Support of yytext. */
        *yy_cp = (yy_hold_char);

        /* yy_bp points to the position in yy_ch_buf of the start of
         * the current run.
         */
        yy_bp = yy_cp;

        yy_current_state = (yy_start);
```

yy_match：它定义了`yy_match`标签，用于匹配输入文本中的词法单元。这个循环会一直执行，直到找到一个匹配的词法单元或者到达文件结束。

```
yy_match:
        do
            {
            YY_CHAR yy_c = yy_ec[YY_SC_TO_UI(*yy_cp)] ;
            if ( yy_accept[yy_current_state] )
                {
                (yy_last_accepting_state) = yy_current_state;
                (yy_last_accepting_cpos) = yy_cp;
                }
            while ( yy_chk[yy_base[yy_current_state] + yy_c] != yy_current_state
)
                {
                yy_current_state = (int) yy_def[yy_current_state];
                if ( yy_current_state >= 119 )
                    yy_c = yy_meta[yy_c];
                }
            yy_current_state = yy_nxt[yy_base[yy_current_state] + yy_c];
            ++yy_cp;
            }
        while ( yy_base[yy_current_state] != 220 );
```

1. `yy_match:`：这是一个标签，可以在代码中通过`goto yy_match;`跳转到这个位置。
2. `do {`：开始了一个`do-while`循环，这个循环会尝试匹配输入文本。
3. `YY_CHAR yy_c = yy_ec[YY_SC_TO_UI(*yy_cp)];`：这行代码读取`yy_cp`当前指向的字符，并使用`yy_ec`数组来转换它。`YY_SC_TO_UI`是一个宏，用于将字符的内部表示转换为`unsigned int`类型。`yy_c`是转换后的字符。
4. `if ( yy_accept[yy_current_state] ) {`：这行代码检查当前状态是否是一个接受状态，即是否匹配了一个词法单元。
5. `(yy_last_accepting_state) = yy_current_state;`：如果当前状态是接受状态，那么记录这个状态和当前字符的位置。`yy_last_accepting_state`和`yy_last_accepting_cpos`分别记录最后一个接受状态和对应的字符位置。
6. `}`：`if`语句结束。

7. `while ( yy_chk[yy_base[yy_current_state] + yy_c] != yy_current_state ) {`：这是一个`while`循环，用于查找下一个状态。`yy_chk`和`yy_base`数组用于确定下一个状态，基于当前状态和读取的字符。
8. `yy_current_state = (int) yy_def[yy_current_state];`：如果`yy_chk`数组中没有找到匹配的下一个状态，就使用`yy_def`数组中的默认状态。
9. `if ( yy_current_state >= 119 ) yy_c = yy_meta[yy_c];`：如果当前状态大于或等于119，那么使用`yy_meta`数组来转换字符`yy_c`。
10. `}`：`while`循环结束。
11. `yy_current_state = yy_nxt[yy_base[yy_current_state] + yy_c];`：更新`yy_current_state`为下一个状态。
12. `++yy_cp;`：移动到下一个字符。
13. `}`：`do`循环结束。
14. `while ( yy_base[yy_current_state] != 220 );`：`do-while`循环会继续执行，直到`yy_base[yy_current_state]`的值不等于220，这通常表示到达了文件的末尾或者没有找到匹配的词法单元。

```
yy_find_action:
        yy_act = yy_accept[yy_current_state];
        if ( yy_act == 0 )
            { /* have to back up */
            yy_cp = (yy_last_accepting_cpos);
            yy_current_state = (yy_last_accepting_state);
            yy_act = yy_accept[yy_current_state];
            }

        YY_DO_BEFORE_ACTION;
```

1. `yy_find_action:`：这是一个标签，可以在代码中通过`goto yy_find_action;`跳转到这个位置。
2. `yy_act = yy_accept[yy_current_state];`：这行代码从`yy_accept`数组中获取当前状态对应的动作编号。`yy_accept`数组存储了每个状态可能接受的动作编号。
3. `if ( yy_act == 0 ) {`：这行代码检查获取的动作编号是否为0。如果为0，意味着当前状态没有接受任何动作，需要回溯到最近的接受状态。
4. `yy_cp = (yy_last_accepting_cpos);`：如果需要回溯，这行代码将`yy_cp`（当前字符指针）回溯到`yy_last_accepting_cpos`记录的位置，即上一个接受状态的字符位置。
5. `yy_current_state = (yy_last_accepting_state);`：同时，这行代码将`yy_current_state`回溯到`yy_last_accepting_state`记录的状态，即上一个接受状态。
6. `yy_act = yy_accept[yy_current_state];`：回溯后，重新从`yy_accept`数组中获取当前状态对应的动作编号。
7. `}`：`if`语句结束。
8. `YY_DO_BEFORE_ACTION;`：这是一个宏，用于在执行动作之前执行一些必要的操作。这个宏的具体内容在Flex生成的扫描器代码中定义，可能包括更新`yytext`指针、发送标记到语法分析器等。

```
do_action:  /* This label is used only to access EOF actions. */

        switch ( yy_act )
    { /* beginning of action switch */
            case 0: /* must back up */
            /* undo the effects of YY_DO_BEFORE_ACTION */
            *yy_cp = (yy_hold_char);
            yy_cp = (yy_last_accepting_cpos);
            yy_current_state = (yy_last_accepting_state);
            goto yy_find_action;
```

1. `do_action:`：这是一个标签，用于在扫描器识别出一个词法单元后跳转到这个位置来执行相应的动作。
2. `/* This label is used only to access EOF actions. */`：这是一个注释，说明`do_action`这个标签只用于访问EOF（文件结束）时的动作。
3. `switch ( yy_act ) {`：这是一个`switch`语句，它根据`yy_act`变量的值来选择执行哪个动作。`yy_act`是在`yy_find_action`部分确定的，代表当前状态对应的动作编号。
4. `/* beginning of action switch */`：这是一个注释，表示动作切换的开始。
5. `case 0:`：这是`switch`语句中的一个`case`标签，用于处理`yy_act`值为0的情况。值为0通常意味着当前状态没有特定的动作与之关联，或者需要回溯到最近的接受状态。
6. `/* must back up */`：这是一个注释，说明在这种情况下需要回溯。
7. `/* undo the effects of YY_DO_BEFORE_ACTION */`：这是一个注释，说明接下来的代码将撤销`YY_DO_BEFORE_ACTION`宏的效果。`YY_DO_BEFORE_ACTION`宏可能已经改变了`yytext`指针和`yy_hold_char`的值。
8. `*yy_cp = (yy_hold_char);`：这行代码将`yy_hold_char`的值写回到`yy_cp`指向的位置，这样可以撤销之前读取的字符。
9. `yy_cp = (yy_last_accepting_cpos);`：这行代码将`yy_cp`指针回溯到`yy_last_accepting_cpos`记录的位置，即上一个接受状态的字符位置。
10. `yy_current_state = (yy_last_accepting_state);`：这行代码将`yy_current_state`回溯到`yy_last_accepting_state`记录的状态，即上一个接受状态。
11. `goto yy_find_action;`：这行代码使用`goto`语句跳转到`yy_find_action`标签，以便重新确定动作。这是因为在回溯之后，扫描器需要重新找到一个新的动作来执行。

然后 后面就是有很多种类型的case，自己选择去执行，返回对应的值即可

如何处理?

使用fin传入文本进行读取，然后yylex就是对其进行分词读取，根据不同的case去返回不同的return

答辩提问

1.你是怎么处理-2147483648这个数的?

2.解释一下符号表相关的类型？（symbol_int）

3.解释一下yylex的调用情况，是怎么识别到最后一个字符的？（EOF的token为0，退出）