- 负数解析出来为 减号和一个大数
- float： 十进制 十六进制
- flex学习链接
- 起始状态的理解:start condition

> 词法分析器当前处于哪一种模式下进行匹配

情况	需要不同规则
普通代码区	识别关键字、标识符、数字等
字符串内	不识别关键字，要识别转义符、结束引号
注释内	不识别代码，直到注释结束符出现

```
%x STRING COMMENT
```

- `%x` 表示 “exclusive start condition”排他，表示进入这个状态后，只有标了这个状态的规则才会匹配。
- `%s` inclusive 普通+该状态

main.c

- 在main函数之前中定义了一些控制输出展示的宏，以及输出控制函数，若超过所设置长度需要截断。
- 在main函数中解析命令行参数，编译选项，输出文件，优化级别，输入文件，出错处理。
- 文件流的打开 `ifstream in(inputFile)  istream* inStream = &in`
- 创建parser对象，调用parseTokens()进行词法分析，按照表格格式输出所有token信息
- 清理资源goto clenup_files并退出 ret =0

parser.h

FE命名空间，frontend

- 成员变量：词法分析器 语法分析器 ast根节点指针
- 成员函数 构造与析构 错误报告
- 私有：词法分析，返回token序列，语法分析，返回AST根节点

```
std::vector<Token> parseToken_impl();
AST::Root* paesrAST_imp();
```

scanner.h

- 疑惑:将yyFlexLexer重命名为Yacc_FlexLexer了，为什么还是可以使用呢在后面类的定义里继承了它!Pasted image 20251013205449.png
- 是宏定义 yyFlexLexer在 `FlexLexer.h` 里被全部替换成了 `Yacc_FlexLexer` 编译器所作的工作。
- scanner类，继承自flex生成的语法分析器，重命名避免冲突
- 成员变量有一个parser对象的引用，用于和语法分析器交互。
- nexttoken函数，返回下一个token

parser.cpp

- `using` 类型别名，简化代码书写
- 可以说type包含了kind,kind所包含信息少，type包含信息多
- 词法分析的实现:
  - 创建一个vector保存所有tokens.
  - 循环调用nexttoken取得下一个token,若遇到结束符跳出循环，
  - 声明一个Token结构体，名字，位置（行列），词素（内置scanner）都可以得到，token为type类型，所包含信息多.
  - 根据kind填充值与类型
  - push_back
  - 返回
- 词法分析根据token类型填充不同属性，语法分析直接调用底层语法分析器

token.h

- 联合类型?
  i. **共享内存**：所有成员共享同一块内存，修改一个成员会影响其他成员。
  ii. **内存大小**：联合的大小至少等于其最大成员的大小，可能会因为内存对齐而更大。
  iii. **同一时间只能使用一个成员**：因为共享内存，所以只能有一个成员存储有效值。

语法例子

[MODE: RESEARCH]

让我帮你分析一下完成词法分析器任务的思考方向和方法。

[4 tools called]

完成词法分析器任务的思考方向

基于我对代码和测试用例的分析，你可以从以下几个维度来思考如何完成任务：

1. 需求分析维度

明确目标

- 从测试用例 `withfloat.lexer` 可以看出，你需要识别的所有 token 类型
- 从 `withfloat.sy` 可以看出需要支持的语法特性
- 从注释中的 TODO 可以看出具体任务

优先级排序

```
高优先级：
1. 运算符识别（+、-、*、/、==、!=、>、<、>=、<=、=、!、&&、|| 等）
2. 浮点数识别（十进制和十六进制）
3. 多行注释处理

中优先级：
4. 长整型常量支持
5. 十六进制/八进制整数

低优先级：
6. 其他高级特性
```

2. 技术实现维度

Token 定义策略

```
// 在 yacc.y 中需要添加的 token
%token <float> FLOAT_CONST
%token <long long> LL_CONST
%token PLUS MINUS STAR SLASH
%token GT LT GE LE EQ NEQ
%token ASSIGN NOT AND OR
%token BITAND BITOR MOD
```

词法规则设计

```
// 浮点数规则（使用已有的转换函数）
[0-9]*\.[0-9]+([eE][+-]?[0-9]+)? {
    float result = convertToFloatDec(yytext);
    return YaccParser::make_FLOAT_CONST(result, loc);
}

// 运算符规则
"+"     { RETT(PLUS, loc) }
"-"     { RETT(MINUS, loc) }
"*"     { RETT(STAR, loc) }
"/"     { RETT(SLASH, loc) }
"=="    { RETT(EQ, loc) }
"!="    { RETT(NEQ, loc) }
// ... 其他运算符
```

3. 测试驱动维度

参考测试用例

- **withfloat.sy**: 包含所有浮点数格式
- **withfloat.lexer**: 期望的输出结果
- **witharray.sy**: 包含数组和运算符使用

验证方法

```
# 编译测试
make

# 运行词法分析测试
./compiler testcase/lexer/withfloat.sy

# 对比期望输出
diff output withfloat.lexer
```

4. 实现步骤维度

第一步：Token 定义

1. 在 `yacc.y` 中添加缺失的 token 定义
2. 确保 token 类型与 AST 定义匹配

第二步：基础运算符

1. 实现单字符运算符（+、-、*、/、=、!、&、|）
2. 实现双字符运算符（==、!=、>=、<=、&&、||）
3. 测试基本运算表达式

第三步：浮点数支持

1. 实现十进制浮点数规则
2. 实现科学计数法支持
3. 实现十六进制浮点数规则
4. 测试各种浮点数格式

第四步：注释处理

1. 实现多行注释 `/* ... */`
2. 处理嵌套注释（如果需要）
3. 确保注释不影响位置跟踪

第五步：完善和测试

1. 启用长整型常量支持
2. 完善错误处理
3. 全面测试所有功能

5. 调试和验证维度

调试策略

```
// 在 lexer.l 中添加调试输出
"+"     {
    std::cout << "识别到 PLUS 运算符" << std::endl;
    RETT(PLUS, loc)
}
```

验证方法

- 对比测试用例的期望输出
- 检查位置信息是否正确
- 验证 token 属性值是否正确

7. 学习建议

理解 Flex 语法

- 正则表达式的优先级
- 最长匹配原则
- 位置跟踪机制

理解编译器架构

- Token 与 AST 的关系
- 词法分析与语法分析的接口
- 错误处理机制

建议的实施顺序：

1. 先实现简单的单字符运算符
2. 再实现浮点数识别
3. 最后处理复杂的多行注释
4. 每个步骤都要测试验证

这样的思考方式能帮助你系统性地完成任务，避免遗漏重要功能，同时确保代码质量。

基于前面的分析，我将为你制定一个完整的可行方案来完成词法分析器的实现。

实施清单

- [x] **修改 yacc.y 文件**：添加所有缺失的 token 定义
- [x] **修改 lexer.l 文件**：添加运算符词法规则
3. **修改 lexer.l 文件**：添加浮点数词法规则
4. **修改 lexer.l 文件**：添加多行注释处理
5. **修改 lexer.l 文件**：添加十六进制/八进制整数规则
6. **修改 lexer.l 文件**：启用长整型常量支持
7. **编译测试**：确保代码能正确编译
8. **功能测试**：使用测试用例验证功能
9. **调试修复**：解决发现的问题
10. **最终验证**：确保所有测试用例通过

其中一项的分析：

- 进入花括号的动作代码时，yytext就是刚刚匹配到的那段原始字符串。
- convert..函数将十进制浮点字符串解析成float数值
- 构造并返回一个带值的float_const token,同时携带当前位置loc
- try/catch:如果解析失败 报告错误消息，返回 err_token,内容是原始字面量。位置还是loc.
- loc的行列更新由宏YY_USER_ACTION自动完成

浮点数

- 第一条 [0-9]+.[0-9]*([eE][+-]?[0-9]+)?：匹配“有小数点”的十进制浮点，整数部分必有，小数部分可空，指数可选。例如：123.456、5.、3.e-10。
- 第二条 .[0-9]+([eE][+-]?[0-9]+)?：匹配“以点开头”的十进制浮点，小数部分必须有，指数可选。例如：.5、.5E+5。
- 第三条 [0-9]+([eE][+-]?[0-9]+)：匹配“只有指数、无小数点”的十进制浮点。例如：1e10、5E-3。

十六进制浮点

- 第一条 `0[xX][0-9A-Fa-f]+(\.[0-9A-Fa-f]*)?[pP][+-]?[0-9]+` ： 要求小数点前至少需要有一位十六进制数字（**有点+无点**）
  - `(\.[...])?` 是可选，所以同时匹配：
  - 无小数点：0x1p5, 0xAp-2
  - 有小数点：0x1.8p0, 0xA.Bp2

- 第二条：匹配以点开头的形式，整数部分为空。
  - 例如：0x.ABp2, 0x.8p0

整数十六进制

> 把匹配到的整数字面值转换为数值，并根据是否溢出Int选择返回INT_CONST或LL_CONST

- `bool isLL=false` :准备一个标志位，表示结果是否超出Int范围
- `long long result = convertInt(yytext,'\0',isLL)`：
  - 解析yytext,即当前匹配到的整数文本，第二个参数表示解析到字符串末尾为止。该函数内部自动判断进制，并在结果超出Int范围时把isLL置为 `true`

单行注释

吞掉本行余下文本 不产生token.
关于行号：这一条不会跨行 所以不需要显式更新loc.lines。

多行注释

使用开始条件。把词法器切换到一个专门处理注释的子模块。核心思路：遇到 `\*` 就进入 `COMMENT` 模式；在 `COMMENT` 里不断吞字符 统计换行；知道见到 `*/` 再回到正模式，若到EOF还没闭合则报错。

```
%x COMMENT   /* 1) 声明注释模式，写在规则表最上方选项附近 */

"/*"            { BEGIN(COMMENT); }   /* 2) 看到开头，进入COMMENT */

<COMMENT>{
  [^*\n]+       { /* 吃普通字符 */ }
  \n            { loc.lines(1); loc.step(); }     /* 3) 统计换行，更新位置 */
  \*+[^*/\n]    { /* 一串*后面不是/，继续吃 */ }
  \*+\/         { BEGIN(INITIAL); }               /* 4) 匹配到 */ 结束，回到初始模式 */
}

/* 5) 若注释未闭合直到EOF，报错 */
<COMMENT><<EOF>> {
  _parser.reportError(loc, "Unterminated block comment");
  return YaccParser::make_ERR_TOKEN(std::string("/*"), loc);
}
```

- `%x COMMENT` 定义一个名为COMMENT的开始条件‘子模式’
- `"/*"{BEGIN(COMMENT);}` :匹配到注释起始，切换到COMMENT模式
- `<COMMENT>{...}` :只有在COMMENT模式时，这些规则才会生效
  - `[^*\n]+` ： 吞掉除*和换行以外的连续字符。
  - `\n` ： 遇换行时，用 `loc.lines(1); loc.step()` ;更新行列。
  - `\*+[^*/\n]` ： 一串 * 后面不是/，说明还没结束，继续吞。
  - `\*+\/` ： 匹配到*/，用BEGIN(INITIAL)回到初始模式（正常扫描）。
- `<COMMENT><<EOF>>` ： 文件结束仍在注释中，报“未闭合注释”并返回错误token。

另外做的事情

- Declare INT and FLOAT tokens in yacc.y
- Add INT/FLOAT keyword and IDENT rules; handle C-style comments in lexer.l
- Populate LL_CONST and FLOAT_CONST properties in parser.cpp token switch