

# 华中科技大学

## 课程设计报告

题目：基于高级语言源程序格式处理工具

课程名称：程序设计综合课程设计

专业班级：计卓 1801 班

学    号：U201814489

姓    名：刘一鸣

指导教师：许贵平

报告日期：2020.3.30

计算机科学与技术学院

## 任 务 书

### □ 设计内容

- 在计算机科学中,抽象语法树(abstract syntax tree 或者缩写为 AST),是将源代码的语法结构的用树的形式表示,树上的每个结点都表示源程序代码中的一种语法成分。之所以说是“抽象”,是因为在抽象语法树中,忽略了源程序中语法成分的一些细节,突出了其主要语法特征。
- 抽象语法树(Abstract Syntax Tree ,AST)作为程序的一种中间表示形式,在程序分析等诸多领域有广泛的应用.利用抽象语法树可以方便地实现多种源程序处理工具,比如源程序浏览器、智能编辑器、语言翻译器等。
- 在《高级语言源程序格式处理工具》这个题目中,首先需要采用形式化的方式,使用巴克斯(BNF)范式定义高级语言的词法规则(字符组成单词的规则)、语法规则(单词组成语句、程序等的规则)。再利用形式语言自动机的原理,对源程序的文件进行词法分析,识别出所有单词;使用编译技术中的递归下降语法分析法,分析源程序的语法结构,并生成抽象语法树,最后可由抽象语法树生成格式化的源程序。

### □ 设计要求

要求具有如下功能:

#### 1. 语言定义

选定 C 语言的一个子集,要求包含:

- (1) 基本数据类型的变量、常量,以及数组。不包含指针、结构/联合、枚举等;
- (2) 各种双目运算符;
- (3) 函数定义、声明与调用;
- (4) 表达式语句、复合语句、if 语句的 2 种形式、while 语句、for 语句,return 语句、break 语句、continue 语句、外部变量说明语句、局部变量说明语句;
- (5) 编译预处理(宏定义、文件包含)
- (6) 注释(块注释与行注释)

## 2. 单词识别

设计 DFA 的状态转换图，实验时给出 DFA，并解释如何在 zhuan 状态迁移中完成单词识别（每个单词都有一个种类编号和单词的字符串这两个特征值），最终生成单词识别子程序。含后缀常量。

## 3. 语法结构分析

- （1） 外部变量声明；
- （2） 函数声明与定义；
- （3） 局部变量的声明；
- （4） 语句及表达式；
- （5） 生成（1）-（4）（包含编译预处理和注释）的抽象语法树并显示。

## 4. 按缩进编排生成源程序文件

## □ 参考文献

- [1] 王生原，董渊，张素琴，吕映芝等. 编译原理（第3版）. 北京：清华大学出版社. 前4章
- [2] 严蔚敏等.数据结构(C语言版).北京：清华大学出版社

## 目 录

<b>1 引言</b> .....	4
1.1 课题背景与意义 .....	4
1.2 国内外研究现状 .....	4
1.3 课程设计的主要研究工作 .....	4
<b>2 系统需求分析与总体设计</b> .....	5
2.1 系统需求分析 .....	5
2.2 系统总体设计 .....	5
<b>3 系统详细设计</b> .....	8
3.1 有关数据结构的定义 .....	8
3.2 主要算法设计 .....	11
<b>4 系统实现与测试</b> .....	13
4.1 系统实现 .....	13
4.2 系统测试 .....	17
<b>5 总结与展望</b> .....	25
5.1 全文总结 .....	25
5.2 工作展望 .....	25
<b>6 体会</b> .....	26
<b>参考文献</b> .....	27
<b>附录</b> .....	28

# 1 引言

## 1.1 课题背景与意义

在计算机科学中，抽象语法树(Abstract Syntax Tree)是将源代码的语法结构用树形式表示、树上的每个节点都表示源程序代码中的一种语法成分。在 AST 中，忽略了语法成分的一些细节，突出了其主要的语法特征。

作为程序的一种中间表现形式，AST 在程序分析等诸多领域有着广泛的应用。利用 AST 可以方便地实现多种源程序处理工具。

在实际的工程中，尤其是编译器设计中，AST 起着举足轻重的作用。编译器领域的著名项目如 GCC (GNU Compiler Collection) 等都用于生成 AST 和对 AST 进行处理的相关组件。

## 1.2 国内外研究现状

当前，抽象代码树的生成是编译器编写中最重要的一环。其主要涉及到编译原理的相关知识。而编译原理是一门历史悠久的课程。由此，在抽象代码树方面，国内外的研究都以及较为成熟，并且以及发展出了成熟的理论。

目前，抽象代码树的生成算法已经存在现成的算法可以生成，如 GNU 工具链中的 BISON、YACC 等。这些工具可以通过以特定语法编写的 BNF 范式，生成对应的 parser 函数和 lexer 函数。从而实现抽象代码树的生成。

## 1.3 课程设计的主要研究工作

在本次课设中，我们主要进行的工作大体可分为如下三点：

- (1) 通过有限状态机的思想实现一个词法分析器 `Lexer`，将代码拆分为一个一个的 `token`，以便于 AST 生成函数 `Parser` 处理；
- (2) 利用递归下降语法分析法，实现从 `token` 到 AST 的转换工作；
- (3) 对任务书中提供的思路进行优化，从而优化代码，降低内存占用率。

## 2 系统需求分析与总体设计

### 2.1 系统需求分析

对于任务书中的描述，本系统所需要解决的主要问题有如下两个：

- (1) 对于给定的 C 语言代码，将其转化为一棵 AST 树，并通过合理的方式输出；
- (2) 对于给定的 C 语言代码，对其进行合理的格式化工作，使代码格式合乎规范。

而在这两个问题中，问题（1）占主要地位，解决了问题（1），我们即可按照同样的思路解决问题（2）。因此，下文我们将主要围绕问题（1）展开。

### 2.2 系统总体设计

如上文所言，AST 树的生成分为两个主要部分：Lexical Analysis 和 Token Parser。故我们不妨将代码分为拆分为两个主要的工具类——Lexer 类以及 Parser 类。我们可以通过调用这两个类的相关成员函数来实现相应的功能。

#### 2.2.1 Lexer 类的设计

从描述可以知道，Lexer 类需要逐步从目标文件中读取字符，组成一个合乎语法的 token，并识别其类型。进一步而言，我们需要通过 Lexer 类不断地组成 token，直到其读取到目标文件的文件尾。为此，我们对 Lexer 类的设计如下（Code/inc/lexer）：

```
class Lexer {
public:
    typedef std::streampos pos_t;

    Lexer(std::istream &s):
        token_line(0), token_lineChar(0), type(tokenUndefined),
        token(""), cur_line(1), cur_lineChar(0), cur_char('\0'), stream(s)
    { charNext();}
```

```

void next(void);

int get_line(void) { return token_line;}
int get_lineChar(void) { return token_lineChar;}
token_t get_type(void) { return type;}
std::string get_token(void) { return token;}

private:
    /* private part */
};

```

在设计中，Lexer 将从文件流 stream 中逐步读取字符，组成 token，然后停下来。用户可以通过 `get_token()` 和 `get_type()` 两个成员函数来获取当前匹配的 token 的类型以及其字符串。同时，为了便于调试，我们提供了 `get_line()` 以及 `get_lineChar()` 来提供当前 token 的起始位置（行数、列数）。当用户需要查看下一个 token 时，用户可以通过调用 `next()` 函数来实现 token 的更新。

Lexer 类将可以通过 `next()` 函数不断地更新状态，直到其读取到 stream 的文件末尾。此时 `Lexer.get_type()` 将返回 `tokenEOF`（定义在 `Code/inc/lexer` 中）来表示已到达文件末尾。此时调用 `next()` 函数将不会造成 Lexer 的更新。

`Lexer.get_type()` 的返回类型为 `token_t`（定义在 `Code/inc/lexer` 中）。`Token_t` 为枚举类型，其主要分为如下几类：

- （1）**token 型**：包括一些基本的 token 类型，如标识符和各种各样的字面值常量（字符、字符串、数字等），同时也有表示匹配错误的 `tokenUndefined` 和表示文件尾的 `tokenEOF` 两个符号，这一类型的枚举值以 “token” 开头；
- （2）**keyword 型**：包括 C 语言中各种各样的关键字，如 “if” “while” “do” 等等，这一类型的枚举值以 “keyword” 开头；
- （3）**punct 型**：包括各种各样的标点符号，、大、中、小三对括号、以及我们所会用到的各种运算符。这一类的枚举值以 “punct” 开头。

### 2.2.2 Parser 类的设计

Parser 类用于不断地从对于的 Lexre 类对象 `lex` 中读取 token，并通过 token

的特征值对下一步的行为做出决策。例如，当 `lex` 中的 `token` 类型为 `keywordInt` 时，`Parser` 类下一步可能就会开始生成一个代表变量定义的节点，诸如此类。`Parser` 类只有一个公开的成员函数 `parserProgramme()`（声明在 `Code/inc/parser` 中，实现详见 `Code/src/parser.cpp`）。通过该函数，用户将可以建立 AST 树并将其通过输出流 `stream` 输出。

### 2.2.3 关于 AST 树

尽管这很惊人，但实际上我们并没有设计专门的数据结构来用于存储 AST 树的相关数据。事实上，我们将直接通过 `Parser` 类输出我们生成的 AST 树，但我们并没有设计一个将 AST 树存储在内存中的中间过程——“在内存中生成一颗 AST 树并通过先序遍历的方式来访问它以输出用户可以阅读的信息”实现起来过于低效。毕竟我们并不需要像 GCC 编译器那样对生成的 AST 树做许多复杂的处理，本课设所要求的只是简单地输出打印 AST 树！

基于这些考虑，我们将不会在内存中存储并访问 AST 树，取而代之的，我们将通过 `Parser` 类对所有的 `token` 进行一次遍历，并将 AST 树的结构输出。如此一来我们能够节约大量的内存空间和计算时间，并避免了对 AST 树节点的繁琐设计。

值得一提的是，尽管我们不在内存中存储 AST 树，但 `Parser` 类中所使用的思路与各种函数是完全可以在经过改造后用来生成一棵 AST 树（保存在内存中，或是放入某个文件，如果有这方面需求的话）的。

### 2.2.4 Format 类的设计

既然我们不存储 AST 树，那么我们将如何利用 AST 树来对我们的代码进行格式化呢？在具体实现中，`Format` 类是完全仿照 `Parser` 类来实现的。同时考虑到“格式化”这一过程的特殊性——我们并不需要像 `Parser` 类那样对 AST 树中的每一个节点都具体分析，处理节点 `VarDef`（代码中定义变量的部分）和处理节点 `FuncDef`（代码中定义函数的节点）的方式几乎是相同的，如此的情况还有很多。因此，我们在 `Parser` 类的基础上对 `Format` 类进行了大幅度的精简，删除了许多本来在 `Parser` 类中存在的私有目标函数。



## 3 系统详细设计

### 3.1 有关数据结构的定义

本项目中主要的数据结构有如下三个，如下我们给出其对应的代码并通过注释来说明各个成员的作用。

#### 3.1.1 Lexer 类（定义在 Code/inc/lexer 中）

```
class Lexer {
public:
    // 构造函数，用于初始化 Lexer 类对象，主要功能是将 Lexer 对象与
    // 文件输入流相绑定，以便于从文件中读取字符
    Lexer(std::istream &s):
        token_line(0), token_lineChar(0), type(tokenUndefined),
        token(""), cur_line(1), cur_lineChar(0), cur_char('\0'), stream(s)
    { charNext();}
    // 用于更新 Lexer 对象的状态
    void next(void);
    // 用于获取 Lexer 当前所匹配的 token 的各种状态
    int get_line(void) { return token_line;}           // 获取 token 第一个字
符所在行数
    int get_lineChar(void) { return token_lineChar;}  // 获取 token 第一个字
符所在列数
    token_t get_type(void) { return type;}           // 获取 token 的类型
    std::string get_token(void) { return token;}      // 获取 token 本身
private:
    int token_line;           // 记录 token 第一个字符所在行数
    int token_lineChar;      // 记录 token 第一个字符所在列数
    token_t type;            // 记录 token 的类型
    std::string token;       // 记录 token 本身
```

```
int cur_line, cur_lineChar;    // 记录当前在 stream 中的行数、列数
char cur_char;                // 记录当前所读取的字符

std::istream &stream;         // 输入流，用于从中读取字符
// 这一部分涉及试探机制
char charNext(void);          // 获取下一个字符
char eatNext(void);           // 捕获下一个字符
bool eatTryNext(char ch);     // 若下一个字符为 ch，则捕获它
// 专门用于匹配 punct 类型的 token
token_t lexerPunct(void);

};
```

### 3.1.2 Parser 类

```
class Parser {
public:
    Parser(std::ostream &ostrm):
        stream(ostrm) {}

    bool parserProgramme(Lexer &lex, int indent = 0);
private:
    std::ostream &stream;        // 用于输出的输出流

    void println_indent(std::string, int indent);    // 在打印特定内容前打印
    indent 个制表符，用于实现特定的缩进
    void queue_clean(int indent);                    // 清空 queue, queue 是一个
    缓冲区，用于存储已匹配成功的 token
    // 用于处理各式各样的子节点的函数
    bool parserVarDef(Lexer &lex, int indent);
    bool parserStructUnionDef(Lexer &lex, int indent);
    bool parserEnumDef(Lexer &lex, int indent);
    bool parserFuncDef(Lexer &lex, int indent);
};
```

```
bool parserFuncType(Lexer &lex, int indent);
bool parserArgList(Lexer &lex, int indent);
bool parserArg(Lexer &lex, int indent);
bool parserComplex(Lexer &lex, int indent);
bool parserJump(Lexer &lex, int indent);
bool parserIf(Lexer &lex, int indent);
bool parserWhile(Lexer &lex, int indent);
bool parserDoWhile(Lexer &lex, int indent);
bool parserFor(Lexer &lex, int indent);
bool parserExpr(Lexer &lex, int indent);
bool parserUnaryExpr(Lexer &lex, int indent);
bool parserBinaryExpr(Lexer &lex, int indent);
bool parserTriExpr(Lexer &lex, int indent);
bool parserFuncCall(Lexer &lex, int indent);
bool parserUnaryOp(Lexer &lex, int indent);
bool parserBinaryOp(Lexer &lex, int indent);
};
```

### 3.1.3 Format 类

```
class Format {
public:
    Format(std::ostream &o): stream(o) {}
    // 因为 Format 的特殊性, 大多数的 token 都在 formatProgramme()中处理
    bool formatProgramme(Lexer &lex);
private:
    std::ostream &stream;
    // 主要用来处理这些比较特殊的格式
    void formatEnumDef(Lexer &lex);
    void formatIf(Lexer &lex, int indent);
    void formatWhile(Lexer &lex, int indent);
    void formatDoWhile(Lexer &lex, int indent);
    void formatFor(Lexer &lex, int indent);
};
```

## 3.2 主要算法设计

### 3.2.1 基于有限状态机实现的 `Lexer::next()`

建立以从 `stream` 中读取的下一个字符 `ch` 为输入，当前匹配字符串为现态的有限状态机（Finite State Machine）。状态机的状态转换图表示如下：

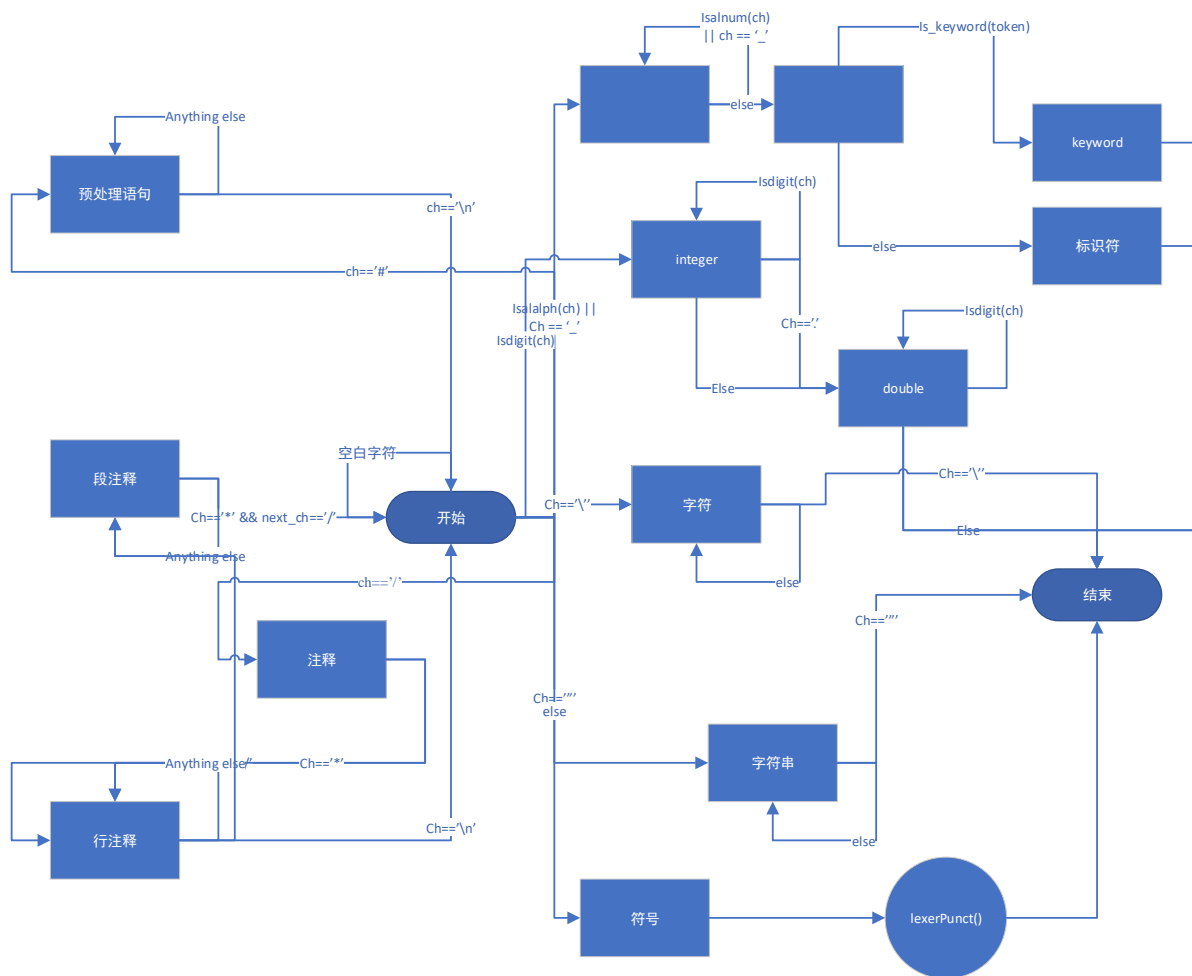


图 3-2-1 Lexer 有限状态机示意图

如图所示，`Lexer` 将通过所读入的字符来判断下一步将出于什么状态。对于预处理语句以及注释，因为其在 `AST` 树中的地位微妙，故 `Lexer` 会跳过他们，并直接开始匹配下一个 `token`。

对于数字常量的匹配，我们将复杂的问题简化为两种：整数（`integer`）和浮点数（`double`）。区分这两者的标志是匹配过程中是否存在 `'.'` 字符。同时，状态机也会通过后缀来判断数字常量的类型，但因为其较为复杂，故在图 3-2-1 中未表示出来。

识别字符/字符串常量的方式很简单：判断 `ch` 是 `'\'` 还是 `''`。同时，匹配时 `Lexer` 考虑到了转义字符的可能性，故 `Lexer` 将可以匹配被转义的字符。

### 3.2.2 使用递归下降子程序法实现 `Parser::parserProgramme()`

利用递归下降法实现 `parserProgramme()` 的思路非常简单。我们只需要在 BNF 范式的基础上编写各个定义项的处理函数即可。我们在 `Code/doc/small_c.txt` 中给出了 C 语言定义的一个子集，由这个子集我们可以编写出各个处理函数来处理程序的不同部分。

但是需要注意的是，在转换的过程中，可能有部分定义拥有相同的开头定义。比如：

```
<var-def> ::= [<storage-type>] <type> [{"*"}] <identifier-list> ";"
<func-def> ::= [<func-type>] <type> <identifier> [{"*"}]
              (" "void" | <arg-list> ")" ( ";" | <complex> )
<identifier-list> ::= <identifier> [{" ", <identifier> }]
```

可以注意到，`<var-def>` 和 `<func-def>` 可能都拥有相同的开头

`<storage-type> <type> <identifier>`

对于这样的情况，当我们才开始匹配 `<storage-type>` 时应该怎么办呢？我们应当如何得知后面的元素如 `<identifier>` 是属于 `<var-def>` 还是属于 `<func-def>` 的呢？一般来说，这里应该引入回溯机制来解决这种问题。但是对于像是文件流这样并不是很适合进行随机访问的数据类型而言，回溯机制的实现是极为繁琐的。而且，在输入流中进行回溯有些时候是不可能的。假如我们使用的输入流并非可重定位的文本输入流而是标准输入流呢（注意，在 `Lexer` 类中，我们只规定了 `stream` 为一般的输入流，而不强求其为一个文件流）？在这样的情况下想要回到输入流的之前某个位置完全是不可能的。因此，我们采用了一种叫做“pre-match”的辅助机制来解决这一问题。关于这一问题详见 4.1.3 节。

关于 `Parser` 的另一个问题是我们如何保证各级节点拥有正确的缩进。注意到任何一个 `parser*()` 函数使用两个参数：一个是用于获取 `token` 的 `Lexer` 对象的引用，而另一个就是当前节点所要求的缩进值了。当一个节点进入下一层节点时，只需要将 `indent+1` 作为下一层的 `indent` 参数传递下去即可。

## 4 系统实现与测试

### 4.1 系统实现

对于本项目，我们采用 ISO C++11 标准进行代码编写，并使用 GNU 工具链进行编译。在代码编写时，关于相关容器和 I/O 的实现均依赖于 C++ 标准库，故用户安装和运行时均不需要安装额外的依赖。

关于代码编写的相关数据结构以及基本思路我们已经在第 3 章中做了总体的介绍，有了编写思路，代码的实现并不复杂，故我们在此不再赘述。如下我们将简要地介绍代码实现中的部分小功能以及技术细节，以进一步完善我们对于系统实现的描述。

#### 4.1.1 系统各组件之间的相互关系

如我们在第 3 章中所描述的那样，本系统主要分为三个部分：Lexer 类、Parser 类以及 Format 类。而在此之上，我们加入了简单的用户交互系统（定义在 Code/src/main.cpp 中）和一套错误处理系统。这些组件之间的调用关系如下图所示

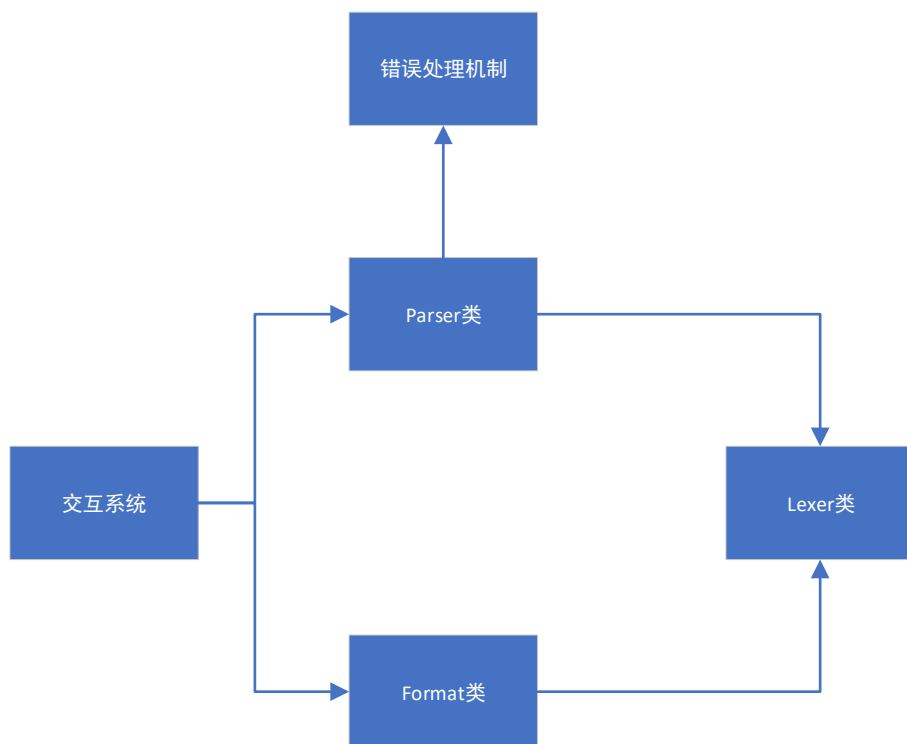


图 4-1-1 各组件之间调用关系图

### 4.1.2 交互系统的实现

虽然没有设计相应的 GUI 供用户使用，但本项目支持一个较为完备的命令输入交互系统。用户可以通过为命令提供各种各样的 args 来实现他们想要的功能。这一交互系统是在 Code/src/main.cpp 中实现的。其使用效果如下：

```
bash-4.4$ build/test
Usage: D:\Documents\z\Code\build\test.exe
-f in_file          assign input file
-o out_file         assign output file
--format            mode = format
--ast               mode = parser
-h                 display this page
bash-4.4$
```

图 4-1-2 交互展示

用户可以在命令行中指定输入与输出文件，并选择是生成 AST 树还是格式化源代码。当未输入“--format”或是“--ast”时，系统将默认使用“--ast” flag。当未指定输出文件时，系统将默认使用标准输出流作为输出文件流。

程序运行时，系统会通过标准错误流将其所匹配到的 token 打印出来，或者逐层打印运行过程中所遭遇的错误（详见 4.1.3 节）。

### 4.1.3 错误处理机制

错误处理机制主要用于在各种 parser\*() 函数中处理未匹配的 token。其主要包含三个全局函数 enterDebug(), exitDebug(), debug\_msg()（均定义在 Code/src/parser.cpp 中）。

enterDebug() 和 exitDebug() 函数维护着一个元素类型为 std::string 的栈容器 s。当程序进入一个 parser\*() 函数时，程序会将当前处理的节点名通过 enterDebug() 函数进行压栈；而当系统退出一个 parser\*() 函数时，函数会调用 exitDebug() 进行退栈。如此一来我们就可以得知当前程序的调用情况以及我们当前所处的是哪一个函数。

当程序在运行过程中发现一个错误时，其会调用 debug\_msg() 函数来向标准错误流发送一条信息。debug\_msg() 在发送这条信息时会将 s 的栈顶元素加在这条信息的前面。

当程序遇到一个较为严重的错误时，比如 `parser*()` 程序匹配到了一个无法满足当前内容规定的 `token` 时，其往往需要通过 `debug_msg()` 发送一条错误信息并退出该函数。我们将该过程简化为了一个宏定义 `debug_error()`。该宏定义会连续执行 `debug_msg()`，`exitDebug()` 以及 `return false` 三条语句，以实现函数的正常退出。而对于上一层的函数，其能够正常运行也取决于其调用的函数能否正常运行，故被调用函数的异常退出将导致上一层的函数也报告错误并退出。如此一来，一旦出现错误，系统表现出来的将会是从被调用的最内层逐层向外报告错误的情景。这显然有利于开发人员追踪 bug。

#### 4.1.4 Parser 中回溯的避免

如我们在之前所提到的，我们在实现 Parser 类的时候，会很自然地想到要使用回溯机制来完成各个模式之间的匹配工作，但我们设法避免了使用回溯。本节将描述我们是如何实现这一功能的。

我们定义了一个名为 `queue` 的向量，用以存放当前已经成功匹配的 `token`。为了便于说明不妨以一段实际的代码为例：

```
static int foo(int a) { /* code here */ }
```

在匹配这一段代码时，系统首先会利用 `Lexer` 类读取第一个 `token` 即“`static`”。其应当属于 `<storage-type>` 或 `<func-type>` 模式。显然，如果我们不继续匹配后续的 `token`，我们是无法得知“`static`”应该属于何种模式的。为解决这一问题，我们不妨将“`static`”先存入 `queue` 中，然后继续读取下一个 `token` “`int`”。同样的，“`int`”同时满足了模式 `<func-def>` 和 `<var-def>`。也将其存入 `queue` 中。如此往复，直到我们读取到 `token` “`(`”。这时我们便可以肯定地指出当前的语句应当是满足 `<func-def>` 而非 `<var-def>` 的。故此时调用用于处理 `<func-def>` 的 `parserFuncDef()` 函数。注意到，此前所匹配的三个 `token` “`static`” “`int`” “`foo`” 也应当是属于 `parserFuncDef()` 的处理范围。故在 `parserFuncDef()` 中会首先遍历 `queue` 中的各个 `token`，对其逐一处理。处理完毕后，因为我们 `queue` 中的 `token` 已经被处理并且不再存在歧义（即它们已经明确地属于 `<func-def>` 这一模式了），我们会清除 `queue` 以便下一次的使用。

注意到，本该属于 `parserFuncDef()` 的处理的三个 `token` “`static`” “`int`” “`foo`” 早在我们进入 `parserFuncDef()` 之前就已经被匹配了的，`parserFuncDef()` 只需要匹



配接下来的 token 并在适当的时候退出即可。故我们称该方法为“pre-match”。对于匹配模式相近的 parser\*() 函数，可以肯定其只有模式开始时的若干 token 是存在歧义的（即可能同时满足若干种模式），故必然存在着一个 token（称该 token 为分歧点）使得从该 token 开始后面的 token 满足且仅满足一个唯一确定的模式。设计函数时，分歧点之前的 token 已经被“pre-match”并保存到 queue 中，函数本身只需要处理分歧点之后的 token。因为分歧点后的 token 不存在歧义，所以处理过程将是单向的。如此一来，我们便避开了使用回溯，程序也就只需要从输入流中单向地读取。

#### 4.1.5 对 Format 成员函数的简化

不同于 Parser 类，在实际使用 Format 类时我们并不关心诸如 <identifier-list> 或是 <expression> 这样的节点下子节点的存在情况，我们更关心的是这个模式之中包含的所有 token。故设计时完全可以放弃创建子节点而直接在当前节点中进行格式化输出。同时，对于大多数节点如 <var-def> 或是以“;”结尾的 <func-def>，其处理形式将是相同的：打印“;”之前的 token 并在打印“;”后换行。事实上，这一处理手法适用于大部分的模式。由此一来，我们便可以大大简化 Format 中的各种成员函数。

而对于各行的缩进问题，在一般情况下，由大括号括起来的部分将会比大括号外面的部分多一个缩进。故我们在计算缩进时主要是通过检测大括号来实现的。

对于一些格式比较特殊的模式，如 <if-block> <do-while-block> 等，我们则按照一般的方法来进行匹配。

#### 4.1.6 系统功能的短板

因为能力有限，我们并没有实现全部的 C 语言功能，而是仅挑选了一些比较有代表性以及常用的功能来加以实现。如下我们将简要介绍本程序在设计上不能满足的 C 语言表述：

- （1）对于单目操作符，本程序仅允许前缀表达，不支持后缀表达。诸如“a++”这样的表达式是不被支持的；
- （2）本程序不允许在定义一个变量的同时对其进行初始化。诸如“int a = 0;”

这样的语句是不被支持的；

- (3) 本程序不支持数组的定义与使用；
- (4) 本程序不支持 `typedef` 或是 `sizeof` 运算符；
- (5) 本程序不支持三元表达式；
- (6) 本程序不支持留空的 `for` 循环语句。诸如“`for (;;)` ”这样的表达是不被支持的；
- (7) 本程序不支持通过“`\`”字符进行换行，这也就意味着字符串常量或宏定义必须定义在一行以内；
- (8) 本程序不支持相邻字符串的自动合并。

诸如此类，因为能力有限，本程序并不适用于所有的 C 语言代码。上文只是叙述了本程序的一些主要的短板。本程序对于 C 语言的支持还有很多问题，在此难以详尽，故使用时还需多加注意。

## 4.2 系统测试

在文件夹 `Code/test` 中，我们编写了一系列的 C 源文件用于测试。不同的源文件有着不同的测试重点，如有的源文件侧重于对函数定义、外部变量、结构体定义等功能测试；而有的源文件则侧重于对复杂表达式的测试。

### 4.2.1 Lexer 功能测试

对于 `Lexer` 功能的测试主要是测试该程序能否顺利识别代码中所出现的所有 `token`。我们主要是使用 `Code/test/lex_test.c` 这一文件作为测试集进行测试。

为了简单起见，我们还专门编写了一段代码 `Code/src/lex_test.cpp` 用于单独测试 `Lexer` 类的功能。用户可以通过在命令行输入“`make lex`”来获取用于测试的程序。下图是测试时的截图片段


```
 /usr/bin/bash --login -i  
20:16: 33: {  
match:21:2: // keyword match  
22:2: 17: auto  
22:7: 22: char  
22:12: 2: a  
22:13: 39: ;  
23:2: 16: const  
23:8: 24: int  
23:12: 2: b1  
23:14: 39: ;  
24:2: 24: int  
24:6: 2: b2  
24:8: 39: ;  
25:2: 18: static  
25:9: 29: double  
25:16: 2: c  
25:17: 39: ;  
26:2: 19: volatile  
26:11: 27: long  
26:16: 2: d  
26:17: 39: ;  
27:2: 26: signed  
27:9: 2: e  
27:10: 39: ;  
28:2: 25: unsigned  
28:11: 2: f  
28:12: 39: ;  
29:2: 28: float  
29:8: 2: g  
29:9: 39: ;  
30:2: 22: char  
30:7: 51: *  
30:8: 2: cptr  
30:12: 39: ;  
31:2: 21: void  
31:7: 51: *  
31:8: 51: *
```

图 4-2-1 Lexer 单独测试（片段）

由此可见，Lexer 成功地匹配了我们所定义所有 token，符合要求。

## 4.2.2 Parser 功能测试

Parser 类的功能测试可以拆分为如下几个部分：（1）测试对于各种外部定义（函数、变量、宏、结构体/联合体、枚举类型的支持；（2）测试对于复杂表达式的支持；（3）条件语句、迭代语句及其相互嵌套的测试。

我们在 Code/test/ 中编写了三个 C 源文件用于测试：test\_extern.c、test\_expression.c、test\_if.c。用户可以通过这三个代码来测试对应的功能。

其测试结果如下：

1	Programme	52	Identifier: a
2	VarDef	53	punctuation: )
3	Keyword: extern	54	punctuation: ;
4	Keyword: int	55	punctuation: }
5	Identifier: a	56	Struct/Union Def
6	punctuation: ;	57	Keyword: struct
7	Func Def	58	Identifier: s
8	Keyword: static	59	punctuation: {
9	Keyword: void	60	VarDef
10	Identifier: fool	61	Keyword: int
11	punctuation: (	62	Identifier: a
12	Arg List	63	punctuation: ;
13	Arg	64	VarDef
14	Keyword: int	65	Keyword: double
15	Identifier: a	66	Identifier: b
16	punctuation: )	67	punctuation: ;
17	punctuation: ;	68	punctuation: }
18	Func Def	69	punctuation: ;
19	Keyword: int	70	Struct/Union Def
20	Identifier: foo2	71	Keyword: union
21	punctuation: (	72	Identifier: u
22	Keyword: void	73	punctuation: {
23	punctuation: )	74	VarDef
24	punctuation: ;	75	Keyword: struct
25	Func Def	76	Identifier: s
26	Keyword: static	77	Identifier: tmp
27	Keyword: void	78	punctuation: ;
28	Identifier: fool	79	VarDef
29	punctuation: (	80	Keyword: int
30	Arg List	81	Identifier: a
31	Arg	82	punctuation: ;
32	Keyword: int	83	punctuation: }
33	Identifier: a	84	punctuation: ;
34	punctuation: )	85	Enum Def
35	Complex	86	Keyword: enum
36	punctuation: {	87	Identifier: e
37	While	88	punctuation: {
38	Keyword: while	89	Identifier: ONE
39	punctuation: (	90	punctuation: ,
40	Enum Def	91	Identifier: TWO

图 4-2-2 测试 test\_extern.c 的输出（部分）

```

79      Identifier: d
80      punctuation: )
81      punctuation: ;
82      Expression
83      Identifier: a
84      punctuation: <=<=
85      Expression
86      punctuation: (
87      Identifier: b
88      punctuation: &
89      Expression
90      Integer: 0x0f
91      punctuation: )
92      punctuation: ;
93      Expression
94      Identifier: a
95      punctuation: =
96      Expression
97      Unary Expression
98      punctuation: ++
99      Expression
100     Identifier: c
101     punctuation: -
102     Expression
103     Unary Expression
104     punctuation: (
105     punctuation: --
106     Expression
107     Identifier: b
108     punctuation: +
109     Expression
110     Unary Expression
111     punctuation: ++
112     Expression
113     Identifier: d
114     punctuation: )
115     punctuation: ;
116     Expression
117     Identifier: a
118     punctuation: ;

```

Normal text file | length: 2,678 |

图 4-2-2 测试 test\_expression.c 的输出（部分）

```

127         punctuation: }
128     DoWhile
129     Keyword: do
130     Complex
131     punctuation: {
132         Expression
133         Identifier: a
134         punctuation: =
135             Expression
136             Identifier: b
137         punctuation: -
138             Expression
139             Integer: 2
140     punctuation: ;
141     Expression
142     Identifier: b
143     punctuation: =
144         Expression
145         Integer: 2
146     punctuation: ;
147     If
148     Keyword: if
149     punctuation: (
150         Expression
151         Identifier: a
152         punctuation: ==
153         Expression
154         Identifier: b
155     punctuation: )
156     Expression
157         Unary Expression
158         punctuation: --
159         Expression
160         Identifier: a
161     punctuation: ;
162     Keyword: else
163     If
164     Keyword: if
165     punctuation: (
166         Expression

```

Normal text file

图 4-2-3 测试 test\_if.c 的测试结果（部分 1）

181	If
182	Keyword: if
183	punctuation: (
184	Expression
185	Identifier: a
186	punctuation: >
187	Expression
188	Identifier: c
189	punctuation: )
190	Expression
191	Function Call
192	Identifier: printf
193	punctuation: (
194	String: "hello world\n"
195	punctuation: )
196	punctuation: ;
197	Keyword: else
198	Complex
199	punctuation: {
200	While
201	Keyword: while
202	punctuation: (
203	Expression
204	Identifier: a
205	punctuation: <
206	Expression
207	Identifier: c
208	punctuation: )
209	Complex
210	punctuation: {
211	Expression
212	Function Call
213	Identifier: printf
214	punctuation: (
215	String: "hello world\n"
216	punctuation: )
217	punctuation: ;
218	For
219	Keyword: for
220	punctuation: /

Normal text file
 length: 5,457 lines: 264

图 4-2-4 测试 test\_if.c 的测试结果（部分 2）

由此可见，我们的代码能够满足测试代码中所要求的各种情况。

### 4.2.3 Format 功能测试

Format 类主要涉及的是程序格式的再处理，为了测试该程序，我们将测试程序 test\_if.c 的段落随意打乱，并命名为 test\_format.c。其内容如下图所示：

```

1  #include <stdio.h>
2  int main(void) { int a, b; a = b = 1;
3      if (a = b) { printf("hello world\n");}
4      else if (a != b) {
5          printf("hello world\n");}
6      while (a != b) {
7          printf("hello world");a = 2;++ b;
8      }
9      do{}while(1);
10     for (int a; b = 0; ++ i) {}
11     do {a = b-2;b = 2;
12     if (a == b)
13         -- a;
14         else if (a > b)
15             ++ a;else {
16         if (a > c)
17             printf("hello world\n");else {
18             while (a < c) {
19                 printf("hello world\n");    for (int a; a < c; ++ a)
20                 printf("world hello\n");}
21             }} while (a != b);return 0;
22     }
23

```

图 4-2-5 格式化之前的 test\_if.c

我们输入命令将其格式化并储存在 output.c 中，处理后的代码如下图所示

```

1  int main ( void ) {
2      int a , b;
3      a = b = 1;
4      if (a=b) {
5          printf ( "hello world\n" );
6      }
7      else if (a!=b) {
8          printf ( "hello world\n" );
9      }
10     while (a!=b) {
11         printf ( "hello world" );
12         a = 2;
13         ++ b;
14     }
15     do {} while (1);
16     for (int a; b = 0; ++ i) {
17     }
18     do {
19         a = b - 2;
20         b = 2;
21         if (a==b)
22             -- a;
23         else if (a>b)
24             ++ a;
25         else {
26             if (a>c)
27                 printf ( "hello world\n" );
28             else {
29                 while (a<c) {
30                     printf ( "hello world\n" );
31                     for (int a; a < c; ++ a)
32                         printf ( "world hello\n" );
33                 }
34             }
35         }
36     } while (a!=b);
37     return 0;
38 }
39

```

图 4-2-6 格式化后的 output.c



可以看到，尽管最终生成的代码还有很多不足，但其效果总体而言是令人满意的。尽管如此我们还是要注意到，预处理语句在格式化的过程中遗失了。这是因为预处理语句本身在 AST 树中不占有任何位置，其地位较为微妙，故基于 AST 树产生的格式化过程无法正确生成预处理语句。

为了克服这一点，用户应当首先使用 GNU 工具链中的 CPP 指令完成对预处理指令的展开，然后再通过本程序进行格式化工作。

当然，这样的作法并不实用。所以在格式化的过程中，程序会将匹配到的预处理语句通过标准错误流输出，用户可以根据这些输出自行补全缺失的预处理指令。

## 5 总结与展望

### 5.1 全文总结

纵观全文，我们实现了一个适用于简单 C 语言的 AST 生成器。并在原有的思路对算法、代码进行了精简以及优化。我们将任务分为三个部分并顺利地完成了这三个部分的代码编写以及测试。

在 Lexer 类的编写过程中，我们利用有限状态机的思想，成功地构建起了关于 token 匹配的有限状态机。由此我们可以将复杂的 C 语言代码拆分成一个一个的 token，大大地便利了后续的分析工作。

在 Parser 类的编写过程中，我们冒险地删去了本应该用于存储 AST 数据的数据结构，并且通过“pre-match”机制避免了回溯问题，使得对源文件“只浏览一次”成为可能。这两种作法大大地降低了代码的时间空间复杂度，使得我们的代码能够更好地利用资源完成任务。

在 Format 类中，我们通过观察，合理利用各个模式之间的共性，精简了代码结构，实现了效率上的提高。

尽管如此，因为个人能力有限，我们的代码不可避免地存在着各种各样的缺陷。很多在 C 语言中的自然而然的表述在程序中都没有得到很好的支持。同时代码目前的功能也过于简单，不足以支持大规模的应用需要；代码格式化的功能也过于机械等等。这些都是我们后续所需要解决的问题。

### 5.2 工作展望

根据目前程序的诸多不足，在未来我们将着重致力于增强程序对 C 语言的支持度，不断改进算法以支持更多更复杂的 C 语言语句。在短期会将尝试加入对于后缀单目运算符的支持、对于结构体/联合体的检查、对于 sizeof 运算符的支持等功能。同时，我们将尝试使用其他的思路实现代码的格式化功能，以克服当前算法所存在的缺陷。

## 6 体会

本次课设虽说从开始到最后上交足足有一个月之久，但真正做起来还是感到手忙脚乱。为了实现任务书中的各种功能，在实现的过程中我尝试了许多不同的思路。本项目的初稿拟用包含若干的 `switch-case` 语句的循环实现 `Lexer` 类中使用的有限状态机。用不同的 `case` 来表示当前状态机的状态，通过循环不断地在各种状态之间跳转，这个思路本身很契合有限状态机的表述。而实际上这一思路也成功了，第一版状态机拥有和现在所使用的状态机几乎相同的功能，并且支持回溯机制。但因为是需要不同的状态之间跳转，所以实现代码显得极为冗长，不便于阅读、维护。同时因为使用的 `while` 循环+`switch` 的结构，代码的效率很低。基于以上的种种原因，该版本最终被放弃。在继续查阅了编译原理领域的相关文献后，我最终选择了使用 `if-else` 结构重写状态机。其实比起状态机，这一版本在结构上更像是行为树。状态之间的跳转大幅减少，代码的可读性也有所上升。

`Parser` 的实现过程也同样是一波三折。最开始在实现是我们利用了 `C++` 的异常处理机制来实现从内层被调用函数到外层函数的回溯。并希望通过回溯机制来使得程序最终匹配到正确的模式。同样的，这样实现的代码很冗长，并且在一些情况下显得多余。因为在每个可能的产生分歧的位置都要编写用于回溯机制的代码，所以代码量变得很大，代码重复率很高。所以当照着这个思路写到一半多一点时，我们放弃了这个思路。并开始思考如何才能减少由回溯机制所带来的代码量。最终，我们在 `github` 上发现了一个值得参考的项目 <https://github.com/Fedjmike/fcc>。并参考这个项目采用了“`pre-match`”机制。同时，也是通过阅读这个项目的代码，我逐渐意识到可以不建立实际的 `AST` 树来实现我们所需的功能。各层函数之间的递归调用本身就构成了一棵 `AST` 树！基于这两个思路，项目中最终所使用的 `Parser` 才得以成型。

所以在完成这个项目的过程中，我再一次体会到了思路对于程序设计的重要性。一个错误的思路能够使代码变得冗长，而一个良好的设计思路能够大大地精简代码结构。同时，这也是我第一次尝试编写这样稍具规模的项目，在查阅资料的过程中也学习到了很多有趣的东西，实在是受益良多。

## 参考文献

- [1] Sam Nipps, Fredjmike's C Compiler, <https://github.com/Fedjmike/fcc>
- [2] Sam Nipps, Dr Strangepack, or: how to write a self-hosting C compiler in 10 hours, <https://github.com/Fedjmike/mini-c>
- [3] 严蔚敏等.数据结构(C语言版).北京:清华大学出版社
- [4] VaibhavRai3, Syntax Directed Translation in Compiler Design, <https://www.geeksforgeeks.org/syntax-directed-translation-in-compiler-design/>
- [5] VaibhavRai3, jishnu3, Introduction of Lexical Analysis, <https://www.geeksforgeeks.org/introduction-of-lexical-analysis/?ref=lbp>

## 附录

```
// Code/inc/parser
/* -*- C++ -*- */

#pragma once

#include <string>
#include <ostream>

#include "lexer"

class Parser {
public:
    Parser(std::ostream &ostrm):
        stream(ostrm) {}

    bool parserProgramme(Lexer &lex, int indent = 0);
private:
    std::ostream &stream;

    void println_indent(std::string, int indent);
    void queue_clean(int indent);

    bool parserVarDef(Lexer &lex, int indent);
    bool parserStructUnionDef(Lexer &lex, int indent);
    bool parserEnumDef(Lexer &lex, int indent);
    bool parserFuncDef(Lexer &lex, int indent);
    bool parserFuncType(Lexer &lex, int indent);
    bool parserArgList(Lexer &lex, int indent);
    bool parserArg(Lexer &lex, int indent);
    bool parserComplex(Lexer &lex, int indent);
```

```
bool parserJump(Lexer &lex, int indent);
bool parserIf(Lexer &lex, int indent);
bool parserWhile(Lexer &lex, int indent);
bool parserDoWhile(Lexer &lex, int indent);
bool parserFor(Lexer &lex, int indent);
bool parserExpr(Lexer &lex, int indent);
bool parserUnaryExpr(Lexer &lex, int indent);
bool parserBinaryExpr(Lexer &lex, int indent);
bool parserTriExpr(Lexer &lex, int indent);
bool parserFuncCall(Lexer &lex, int indent);
bool parserUnaryOp(Lexer &lex, int indent);
bool parserBinaryOp(Lexer &lex, int indent);
};
// Code/inc/lexer
/* -*- C++ -*- */

#pragma once

#include <iostream>
#include <string>

typedef enum {
    tokenUndefined,
    tokenEOF,
    tokenIdent,
    tokenInt,
    tokenDouble,
    tokenChar,
    tokenStr,

    keywordIf, keywordElse, keywordWhile, keywordDo, keywordFor,
    keywordReturn, keywordBreak, keywordContinue,
    keywordSizeof,
    keywordConst, keywordAuto, keywordStatic, keywordVolatile, keywordExtern,
```

keywordVoid, keywordChar, keywordShort, keywordInt,  
keywordUnsigned, keywordSigned, keywordLong,  
keywordFloat, keywordDouble,  
keywordStruct, keywordUnion, keywordEnum,

punctLBrace, punctRBrace,  
punctLParen, punctRParen,  
punctLBracket, punctRBracket,  
punctSemicolon, punctArrow,  
punctComma, punctPeriod,  
punctQuestion, punctColon,

/\* algorithm \*/

punctPlus, punctPlusAssign, punctPlusPlus,  
punctMinus, punctMinusAssign, punctMinusMinus,  
punctTimes, punctTimesAssign,  
punctDivide, punctDivideAssign,  
punctModulo, punctModuloAssign,  
punctAssign,

/\* compare \*/

punctEqual, punctNotEqual,  
punctGreater, punctGreaterEqual,  
punctLess, punctLessEqual,

/\* bitwise \*/

punctBitwiseAnd, punctBitwiseAndAssign,  
punctBitwiseOr, punctBitwiseOrAssign,  
punctBitwiseXor, punctBitwiseXorAssign,  
punctBitwiseNot,  
punctShr, punctShrAssign,  
punctShl, punctShlAssign,

/\* logical \*/

punctLogicalAnd, punctLogicalOr,  
punctLogicalNot,

} token\_t;

```
class Lexer {
public:
    Lexer(std::istream &s):
        token_line(0), token_lineChar(0), type(tokenUndefined),
        token(""), cur_line(1), cur_lineChar(0), cur_char('\0'), stream(s)
    { charNext();}

    void next(void);

    int get_line(void) { return token_line;}
    int get_lineChar(void) { return token_lineChar;}
    token_t get_type(void) { return type;}
    std::string get_token(void) { return token;}
private:
    int token_line;
    int token_lineChar;
    token_t type;
    std::string token;

    int cur_line, cur_lineChar;
    char cur_char;

    std::istream &stream;

    char charNext(void);
    char eatNext(void);
    bool eatTryNext(char ch);

    token_t lexerPunct(void);
};
// Code/inc/format
/* -*- C++ -*- */
```



```
#pragma once

#include <iostream>
#include <string>

#include "lexer"

class Format {
public:
    Format(std::ostream &o): stream(o) {}

    bool formatProgramme(Lexer &lex, int indent = 0);
private:
    std::ostream &stream;

    bool formatFuncDef(Lexer &lex, int indent);
    bool formatComplex(Lexer &lex, int indent);
    bool formatIf(Lexer &lex, int indent);
    bool formatWhile(Lexer &lex, int indent);
    bool formatDoWhile(Lexer &lex, int indent);
    bool formatFor(Lexer &lex, int indent);
    bool formatExpr(Lexer &lex, int indent);
};

// Code/src/lexer.cpp
#include <deque>
#include <string>
#include <iostream>

#include "../inc/parser"

#pragma GCC dependence "../inc/parser"

static std::deque<std::string> s;
```

```
static void enterDebug(std::string str) {
    s.push_back(str);
}

static void debug_msg(std::string str) {
    std::cerr << s.back() << ": "
        << str << std::endl;
}

static void debug_msg(char *str) {
    std::cerr << s.back() << ": "
        << str << std::endl;
}

static void exitDebug(void) {
    s.pop_back();
}

static std::string decoder(token_t type) {
    if (type == tokenUndefined) return "Undefined";
    else if (type == tokenEOF) return "EOF";
    else if (type == tokenIdent) return "Identifier";
    else if (type == tokenInt) return "Integer";
    else if (type == tokenDouble) return "Float number";
    else if (type == tokenChar) return "Character";
    else if (type == tokenStr) return "String";
    else if (type >= keywordIf && type <= keywordEnum)
        return "Keyword";
    else return "punctuation";
}

class token_pair {
public:
    token_t type;
```

```
std::string token;

token_pair(token_t tp, std::string tk):
    type(tp), token(tk) {}
};

static std::deque<token_pair> queue;

static void tokenMatchNext(Lexer &lex) {
    queue.push_back(token_pair(lex.get_type(), lex.get_token()));

    std::cerr << "matched: " <<
        lex.get_line() << ":" << lex.get_lineChar() << ":" <<
        << lex.get_token() << std::endl;
    lex.next();
}

static bool tokenTryMatchNext(Lexer &lex, token_t type) {
    if (type == lex.get_type()) {
        tokenMatchNext(lex);
        return true;
    }
    else return false;
}

static bool storageTypeTryMatchNext(Lexer &lex) {
    return tokenTryMatchNext(lex, keywordConst)
        || tokenTryMatchNext(lex, keywordAuto)
        || tokenTryMatchNext(lex, keywordStatic)
        || tokenTryMatchNext(lex, keywordVolatile)
        || tokenTryMatchNext(lex, keywordExtern);
}

static bool typeTryMatchNext(Lexer &lex) {
```

```

bool ret = tokenTryMatchNext(lex, keywordVoid)
    || tokenTryMatchNext(lex, keywordChar)
    || tokenTryMatchNext(lex, keywordShort)
    || tokenTryMatchNext(lex, keywordInt)
    || tokenTryMatchNext(lex, keywordUnsigned)
    || tokenTryMatchNext(lex, keywordSigned)
    || tokenTryMatchNext(lex, keywordLong)
    || tokenTryMatchNext(lex, keywordFloat)
    || tokenTryMatchNext(lex, keywordDouble);

if (!ret && tokenTryMatchNext(lex, keywordStruct) ||
    tokenTryMatchNext(lex, keywordUnion) ||
    tokenTryMatchNext(lex, keywordEnum))
{
    ret = tokenTryMatchNext(lex, tokenIdent);
}

return ret;
}

void Parser::println_indent(std::string str, int indent) {
    while (indent --)
        stream << "\t";
    stream << str << std::endl;
}

void Parser::queue_clean(int indent) {
    for (token_pair &it : queue)
        println_indent(decoder(it.type) + ": " + it.token, indent);
    queue.clear();
}

// exception handling
#define debug_error() \

```

```
do {\n    debug_msg("invalid token " + lex.get_token()); \n    exitDebug(); \n    return false; \n} while (false)\n\nbool Parser::parserProgramme(Lexer &lex, int indent) {\n    enterDebug("Programme");\n    bool ret;\n\n    println_indent("Programme", indent);\n    while (lex.get_type() != tokenEOF) {\n        // struct-union-def\n        if (tokenTryMatchNext(lex, keywordStruct) ||\n            tokenTryMatchNext(lex, keywordUnion))\n        {\n            ret = parserStructUnionDef(lex, indent + 1);\n        }\n        // enum-def\n        else if (tokenTryMatchNext(lex, keywordEnum))\n            ret = parserEnumDef(lex, indent + 1);\n        // var-def | func-def\n        else if (storageTypeTryMatchNext(lex)\n            || typeTryMatchNext(lex))\n        {\n            typeTryMatchNext(lex);\n            // if pointer\n            while (tokenTryMatchNext(lex, punctTimes))\n                ;\n            if (!tokenTryMatchNext(lex, tokenIdent))\n                debug_error();\n            // func-def\n            if (tokenTryMatchNext(lex, punctLParen))\n                ret = parserFuncDef(lex, indent + 1);\n        }\n    }\n}
```

```
// var-def
else
    ret = parserVarDef(lex, indent + 1);
}
else
    debug_error();
}

exitDebug();
return ret;
}

/* pre-match:
[<storage-type>] <type> [{"*"}] <identifier>
*/

bool Parser::parserVarDef(Lexer &lex, int indent) {
    enterDebug("VarDef");
    println_indent("VarDef", indent);

    queue_clean(indent);
    while (tokenTryMatchNext(lex, punctComma)) {
        if (!tokenTryMatchNext(lex, tokenIdent))
            debug_error();
    }
    if (!tokenTryMatchNext(lex, punctSemicolon))
        debug_error();
    queue_clean(indent);

    exitDebug();
    return true;
}

/* pre-matched:
"struct" | "union"
```

```
*/  
  
bool Parser::parserStructUnionDef(Lexer &lex, int indent) {  
    enterDebug("StructUnionDef");  
    println_indent("Struct/Union Def", indent);  
  
    if (tokenTryMatchNext(lex, tokenIdent) &&  
        tokenTryMatchNext(lex, punctLBrace))  
    {  
        queue_clean(indent);  
        // matching fields  
        while (typeTryMatchNext(lex)) {  
            if (tokenTryMatchNext(lex, tokenIdent) &&  
                parserVarDef(lex, indent + 1))  
            {}  
            else debug_error();  
        }  
  
        if (tokenTryMatchNext(lex, punctRBrace)  
            && tokenTryMatchNext(lex, punctSemicolon))  
            ;  
        else debug_error();  
    }  
    else debug_error();  
    queue_clean(indent);  
  
    exitDebug();  
    return true;  
}  
  
/* pre-matched:  
    "enum"  
*/  
  
bool Parser::parserEnumDef(Lexer &lex, int indent) {  
    enterDebug("EnumDef");
```

```

println_indent("Enum Def", indent);

queue_clean(indent);
if (tokenTryMatchNext(lex, tokenIdent) &&
    tokenTryMatchNext(lex, punctLBrace))
{
    while (tokenTryMatchNext(lex, tokenIdent)) {
        if (!tokenTryMatchNext(lex, punctComma))
            break;
    }
    for (token_pair &it : queue)
        println_indent(decoder(it.type) + ": " + it.token, indent + 1);
    queue.clear();

    if (tokenTryMatchNext(lex, punctRBrace) &&
        tokenTryMatchNext(lex, punctSemicolon))
        ;
    else debug_error();
}
queue_clean(indent);

exitDebug();
return true;
}

/* pre-matched:
    [<func-type>] <type> [{"*"}] <identifier> "("
*/
bool Parser::parserFuncDef(Lexer &lex, int indent) {
    enterDebug("FuncDef");
    println_indent("Func Def", indent);

    queue_clean(indent);
    // "void" is a bit different from other types

```



```
if (tokenTryMatchNext(lex, keywordVoid) &&
    tokenTryMatchNext(lex, punctRParen))
{
    queue_clean(indent);
    if (tokenTryMatchNext(lex, punctSemicolon))
        queue_clean(indent);
    else if (parserComplex(lex, indent + 1))
        ;
    else debug_error();
}
else if (parserArgList(lex, indent + 1) &&
    tokenTryMatchNext(lex, punctRParen))
{
    queue_clean(indent);
    if (tokenTryMatchNext(lex, punctSemicolon))
        queue_clean(indent);
    else if (parserComplex(lex, indent + 1))
        ;
    else debug_error();
}
else debug_error();

exitDebug();
return true;
}

/* pre-matched:
*/

bool Parser::parserArgList(Lexer &lex, int indent) {
    enterDebug("ArgList");
    println_indent("Arg List", indent);

    if (parserArg(lex, indent + 1)) {
        while (tokenTryMatchNext(lex, punctComma)) {
```

```
        queue_clean(indent);
        if (!parserArg(lex, indent + 1))
            debug_error();
    }
}
else debug_error();

exitDebug();
return true;
}

/* pre-matched:
*/

bool Parser::parserArg(Lexer &lex, int indent) {
    enterDebug("Arg");
    println_indent("Arg", indent);

    if (typeTryMatchNext(lex)) {
        // if pointer
        while (tokenTryMatchNext(lex, punctTimes))
            ;
        if (!tokenTryMatchNext(lex, tokenIdent))
            debug_error();
        queue_clean(indent);
    }
    else debug_error();

    exitDebug();
    return true;
}

/* pre-matched:
*/

bool Parser::parserComplex(Lexer &lex, int indent) {
```

```
enterDebug("Complex");
println_indent("Complex", indent);
bool ret = true;

if (!tokenTryMatchNext(lex, punctLBrace))
    debug_error();
queue_clean(indent);
while (ret) {
    if (tokenTryMatchNext(lex, punctRBrace)) {
        queue_clean(indent);
        break;
    }
    // <var-def>
    else if (storageTypeTryMatchNext(lex) ||
             typeTryMatchNext(lex))
    {
        typeTryMatchNext(lex);
        ret = tokenTryMatchNext(lex, tokenIdent) &&
              parserVarDef(lex, indent + 1);
    }
    // <if-block>
    else if (tokenTryMatchNext(lex, keywordIf))
        ret = parserIf(lex, indent + 1);
    // <while-block>
    else if (tokenTryMatchNext(lex, keywordWhile))
        ret = parserWhile(lex, indent + 1);
    // <do-while-block>
    else if (tokenTryMatchNext(lex, keywordDo))
        ret = parserDoWhile(lex, indent + 1);
    // <for-block>
    else if (tokenTryMatchNext(lex, keywordFor))
        ret = parserFor(lex, indent + 1);
    // <jump>
    else if (tokenTryMatchNext(lex, keywordBreak)
```

```
        || tokenTryMatchNext(lex, keywordContinue)
        || tokenTryMatchNext(lex, keywordReturn))
        ret = parserJump(lex, indent + 1);
    // <expression> ";"
    else {
        ret = parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon);
        queue_clean(indent + 1);
    }

}

if (false == ret)
    debug_error();

exitDebug();
return true;
}

/* pre-matched:
    "break" | "continue" | "return"
*/
bool Parser::parserJump(Lexer &lex, int indent) {
    enterDebug("Jump");
    println_indent("Jump", indent);

    queue_clean(indent);
    if (parserExpr(lex, indent + 1) && tokenTryMatchNext(lex, punctSemicolon))
        queue_clean(indent);
    else debug_error();

    exitDebug();
    return true;
}
```

```

/* pre-matched:
   "if"
*/

bool Parser::parserIf(Lexer &lex, int indent) {
    enterDebug("If");
    println_indent("If", indent);

    if (tokenTryMatchNext(lex, punctLParen))
        queue_clean(indent);
    else debug_error();
    if (parserExpr(lex, indent + 1) && tokenTryMatchNext(lex, punctRParen)) {
        queue_clean(indent);
    }
    else debug_error();

    bool ret = true;
    if (lex.get_type() == punctLBrace) {
        ret = parserComplex(lex, indent + 1);
    }
    else {
        ret = parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon);
        queue_clean(indent);
    }
    if (tokenTryMatchNext(lex, keywordElse)) {
        queue_clean(indent);
        if (tokenTryMatchNext(lex, keywordIf))
            ret = parserIf(lex, indent + 1);
        else if (lex.get_type() == punctLBrace)
            ret = parserComplex(lex, indent + 1);
        else {
            ret = parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon);
            queue_clean(indent);

```

```
    }
}

if (false == ret)
    debug_msg("invalid token " + lex.get_token());
exitDebug();
return ret;
}

/* pre-matched:
   "while"
*/

bool Parser::parserWhile(Lexer &lex, int indent) {
    enterDebug("While");
    println_indent("While", indent);

    if (tokenTryMatchNext(lex, punctLParen))
        queue_clean(indent);
    else debug_error();
    if (parserExpr(lex, indent + 1) && tokenTryMatchNext(lex, punctRParen)) {
        queue_clean(indent);
    }
    else debug_error();

    bool ret = true;
    if (lex.get_type() == punctLBrace) {
        ret = parserComplex(lex, indent + 1);
    }
    else {
        ret = parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon);
        queue_clean(indent);
    }
}
```

```
    if (false == ret)
        debug_msg("invalid token " + lex.get_token());
    exitDebug();
    return ret;
}

/* pre-matched:
    "do"
*/

bool Parser::parserDoWhile(Lexer &lex, int indent) {
    enterDebug("DoWhile");
    println_indent("DoWhile", indent);
    queue_clean(indent);

    if (parserComplex(lex, indent + 1) &&
        tokenTryMatchNext(lex, keywordWhile) &&
        tokenTryMatchNext(lex, punctLParen))
    {
        queue_clean(indent);
        if (parserExpr(lex, indent + 1) &&
            tokenTryMatchNext(lex, punctRParen) &&
            tokenTryMatchNext(lex, punctSemicolon))
        {
            queue_clean(indent);
        }
        else debug_error();
    }
    else debug_error();

    exitDebug();
    return true;
}

/* pre-matched:
```

```
"for"
*/
bool Parser::parserFor(Lexer &lex, int indent) {
    enterDebug("For");
    println_indent("For", indent);

    if (tokenTryMatchNext(lex, punctLParen)) {
        queue_clean(indent);
        // <var-def>
        if (storageTypeTryMatchNext(lex) || typeTryMatchNext(lex))
        {
            typeTryMatchNext(lex);
            if (tokenTryMatchNext(lex, tokenIdent) &&
                parserVarDef(lex, indent + 1))
            {}
            else debug_error();
        }

        if (parserExpr(lex, indent + 1) &&
            tokenTryMatchNext(lex, punctSemicolon))
            ;
        else debug_error();
        queue_clean(indent);
        if (parserExpr(lex, indent + 1) &&
            tokenTryMatchNext(lex, punctRParen))
            ;
        else debug_error();
        queue_clean(indent);
        if (lex.get_type() == punctLBrace)
            parserComplex(lex, indent + 1);
        else if (parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon))
            queue_clean(indent);
        else debug_error();
```



```
    }  
    else debug_error();  
  
    exitDebug();  
    return true;  
}  
  
static bool unaryTryMatchNext(Lexer &lex) {  
    return tokenTryMatchNext(lex, punctPlus)  
        || tokenTryMatchNext(lex, punctPlusPlus)  
        || tokenTryMatchNext(lex, punctMinus)  
        || tokenTryMatchNext(lex, punctMinusMinus)  
        || tokenTryMatchNext(lex, punctTimes)  
        || tokenTryMatchNext(lex, punctBitwiseAnd)  
        || tokenTryMatchNext(lex, punctBitwiseNot)  
        || tokenTryMatchNext(lex, punctLogicalNot)  
        || tokenTryMatchNext(lex, keywordSizeof);  
}  
  
static bool binaryTryMatchNext(Lexer &lex) {  
    return tokenTryMatchNext(lex, punctPlus)  
        || tokenTryMatchNext(lex, punctPlusAssign)  
        || tokenTryMatchNext(lex, punctMinus)  
        || tokenTryMatchNext(lex, punctMinusAssign)  
        || tokenTryMatchNext(lex, punctTimes)  
        || tokenTryMatchNext(lex, punctTimesAssign)  
        || tokenTryMatchNext(lex, punctDivide)  
        || tokenTryMatchNext(lex, punctDivideAssign)  
        || tokenTryMatchNext(lex, punctModulo)  
        || tokenTryMatchNext(lex, punctModuloAssign)  
        || tokenTryMatchNext(lex, punctAssign)  
        || tokenTryMatchNext(lex, punctEqual)  
        || tokenTryMatchNext(lex, punctLess)  
        || tokenTryMatchNext(lex, punctLessEqual)
```

```
    || tokenTryMatchNext(lex, punctGreater)
    || tokenTryMatchNext(lex, punctGreaterEqual)
    || tokenTryMatchNext(lex, punctNotEqual)
    || tokenTryMatchNext(lex, punctShr)
    || tokenTryMatchNext(lex, punctShrAssign)
    || tokenTryMatchNext(lex, punctShl)
    || tokenTryMatchNext(lex, punctShlAssign)
    || tokenTryMatchNext(lex, punctBitwiseAnd)
    || tokenTryMatchNext(lex, punctLogicalAnd)
    || tokenTryMatchNext(lex, punctBitwiseAndAssign)
    || tokenTryMatchNext(lex, punctBitwiseOr)
    || tokenTryMatchNext(lex, punctLogicalOr)
    || tokenTryMatchNext(lex, punctBitwiseOrAssign)
    || tokenTryMatchNext(lex, punctBitwiseXor)
    || tokenTryMatchNext(lex, punctBitwiseXorAssign)
    || tokenTryMatchNext(lex, punctComma)
    || tokenTryMatchNext(lex, punctPeriod)
    || tokenTryMatchNext(lex, punctArrow);
}
```

```
static bool literalTryMatchNext(Lexer &lex) {
    return tokenTryMatchNext(lex, tokenInt)
        || tokenTryMatchNext(lex, tokenDouble)
        || tokenTryMatchNext(lex, tokenChar)
        || tokenTryMatchNext(lex, tokenStr);
}
```

```
/* pre-matched:
*/
```

```
bool Parser::parserExpr(Lexer &lex, int indent) {
    enterDebug("Expression");
    println_indent("Expression", indent);
```

```

bool flag_paren = tokenTryMatchNext(lex, punctLParen);
bool ret = true;
// <unary-expr>
if (unaryTryMatchNext(lex))
    ret = parserUnaryExpr(lex, indent + 1);
// <literal>
else if (literalTryMatchNext(lex))
    queue_clean(indent);
// <identfier> or <func-call>
else if (tokenTryMatchNext(lex, tokenIdent)) {
    if (tokenTryMatchNext(lex, punctLParen))
        ret = parserFuncCall(lex, indent + 1);
    else
        queue_clean(indent);
}

// [{ <binary-op> <expression> }]
while (ret && binaryTryMatchNext(lex)) {
    queue_clean(indent);
    ret = parserExpr(lex, indent + 1);
}
// "?" <expression> ":" <expression>
if (ret && tokenTryMatchNext(lex, punctQuestion)) {
    queue_clean(indent);
    ret = parserExpr(lex, indent + 1);
    if (!tokenTryMatchNext(lex, punctColon))
        debug_error();
    queue_clean(indent);
    ret = parserExpr(lex, indent + 1);
}

if (flag_paren) {
    if (tokenTryMatchNext(lex, punctRParen))
        queue_clean(indent);
}

```

```
        else
            debug_error();
    }

    if (false == ret)
        debug_error();
    exitDebug();
    return ret;
}

/* pre-matched:
    <unary-op>
*/
bool Parser::parserUnaryExpr(Lexer &lex, int indent) {
    enterDebug("UnaryExpr");
    println_indent("Unary Expression", indent);

    queue_clean(indent);
    if (!parserExpr(lex, indent + 1))
        debug_error();

    exitDebug();
    return true;
}

/* pre-matched:
    <identifier> "("
*/
bool Parser::parserFuncCall(Lexer &lex, int indent) {
    enterDebug("FuncCall");
    println_indent("Function Call", indent);

    if (tokenTryMatchNext(lex, tokenIdent) ||
        literalTryMatchNext(lex))
```

```
{
    while (tokenTryMatchNext(lex, punctComma)) {
        if (tokenTryMatchNext(lex, tokenIdent) ||
            literalTryMatchNext(lex))
            ;
        else debug_error();
    }
}

if (tokenTryMatchNext(lex, punctRParen))
    queue_clean(indent);
else debug_error();

exitDebug();
return true;
}

// Code/src/lexer_test.cpp
#include <iostream>
#include <string>
#include <fstream>

#include "../inc/lexer"

int main(void) {
    std::string filename;

    std::cout << "filename: ";
    std::cin >> filename;

    std::ifstream ifstrm(filename);
    Lexer lex(ifstrm);

    lex.next();
    while (lex.get_type() != tokenEOF) {
```

```
std::cout << lex.get_line() << ":" << lex.get_lineChar()
    << ": " << lex.get_type() << ": "
    << lex.get_token() << std::endl;
lex.next();
}

return 0;
}
// Code/src/parser.cpp
#include <deque>
#include <string>
#include <iostream>

#include "../inc/parser"

#pragma GCC dependence "../inc/parser"

static std::deque<std::string> s;

static void enterDebug(std::string str) {
    s.push_back(str);
}

static void debug_msg(std::string str) {
    std::cerr << s.back() << ": "
        << str << std::endl;
}

static void debug_msg(char *str) {
    std::cerr << s.back() << ": "
        << str << std::endl;
}

static void exitDebug(void) {
```

```

    s.pop_back();
}

static std::string decoder(token_t type) {
    if (type == tokenUndefined) return "Undefined";
    else if (type == tokenEOF) return "EOF";
    else if (type == tokenIdent) return "Identifier";
    else if (type == tokenInt) return "Integer";
    else if (type == tokenDouble) return "Float number";
    else if (type == tokenChar) return "Character";
    else if (type == tokenStr) return "String";
    else if (type >= keywordIf && type <= keywordEnum)
        return "Keyword";
    else return "punctuation";
}

class token_pair {
public:
    token_t type;
    std::string token;

    token_pair(token_t tp, std::string tk):
        type(tp), token(tk) {}
};

static std::deque<token_pair> queue;

static void tokenMatchNext(Lexer &lex) {
    queue.push_back(token_pair(lex.get_type(), lex.get_token()));

    std::cerr << "matched: " <<
        lex.get_line() << ":" << lex.get_lineChar() << ": "
        << lex.get_token() << std::endl;
    lex.next();
}

```

```
}
```

```
static bool tokenTryMatchNext(Lexer &lex, token_t type) {
    if (type == lex.get_type()) {
        tokenMatchNext(lex);
        return true;
    }
    else return false;
}
```

```
static bool storageTypeTryMatchNext(Lexer &lex) {
    return tokenTryMatchNext(lex, keywordConst)
        || tokenTryMatchNext(lex, keywordAuto)
        || tokenTryMatchNext(lex, keywordStatic)
        || tokenTryMatchNext(lex, keywordVolatile)
        || tokenTryMatchNext(lex, keywordExtern);
}
```

```
static bool typeTryMatchNext(Lexer &lex) {
    bool ret = tokenTryMatchNext(lex, keywordVoid)
        || tokenTryMatchNext(lex, keywordChar)
        || tokenTryMatchNext(lex, keywordShort)
        || tokenTryMatchNext(lex, keywordInt)
        || tokenTryMatchNext(lex, keywordUnsigned)
        || tokenTryMatchNext(lex, keywordSigned)
        || tokenTryMatchNext(lex, keywordLong)
        || tokenTryMatchNext(lex, keywordFloat)
        || tokenTryMatchNext(lex, keywordDouble);

    if (!ret && tokenTryMatchNext(lex, keywordStruct) ||
        tokenTryMatchNext(lex, keywordUnion) ||
        tokenTryMatchNext(lex, keywordEnum))
    {
        ret = tokenTryMatchNext(lex, tokenIdent);
    }
}
```



```
    }

    return ret;
}

void Parser::println_indent(std::string str, int indent) {
    while (indent --)
        stream << '\t';
    stream << str << std::endl;
}

void Parser::queue_clean(int indent) {
    for (token_pair &it : queue)
        println_indent(decoder(it.type) + ": " + it.token, indent);
    queue.clear();
}

// exception handling
#define debug_error() \
    do {\
        debug_msg("invalid token " + lex.get_token()); \
        exitDebug(); \
        return false; \
    } while (false)

bool Parser::parserProgramme(Lexer &lex, int indent) {
    enterDebug("Programme");
    bool ret;

    println_indent("Programme", indent);
    while (lex.get_type() != tokenEOF) {
        // struct-union-def
        if (tokenTryMatchNext(lex, keywordStruct) ||
            tokenTryMatchNext(lex, keywordUnion))
```

```

    {
        ret = parserStructUnionDef(lex, indent + 1);
    }
    // enum-def
    else if (tokenTryMatchNext(lex, keywordEnum))
        ret = parserEnumDef(lex, indent + 1);
    // var-def | func-def
    else if (storageTypeTryMatchNext(lex)
        || typeTryMatchNext(lex))
    {
        typeTryMatchNext(lex);
        // if pointer
        while (tokenTryMatchNext(lex, punctTimes))
            ;
        if (!tokenTryMatchNext(lex, tokenIdent))
            debug_error();
        // func-def
        if (tokenTryMatchNext(lex, punctLParen))
            ret = parserFuncDef(lex, indent + 1);
        // var-def
        else
            ret = parserVarDef(lex, indent + 1);
    }
    else
        debug_error();
}

exitDebug();
return ret;
}

/* pre-match:
    [<storage-type>] <type> [{"*"}] <identifier>
*/

```

```
bool Parser::parserVarDef(Lexer &lex, int indent) {
    enterDebug("VarDef");
    println_indent("VarDef", indent);

    queue_clean(indent);
    while (tokenTryMatchNext(lex, punctComma)) {
        if (!tokenTryMatchNext(lex, tokenIdent))
            debug_error();
    }
    if (!tokenTryMatchNext(lex, punctSemicolon))
        debug_error();
    queue_clean(indent);

    exitDebug();
    return true;
}

/* pre-matched:
   "struct" | "union"
*/

bool Parser::parserStructUnionDef(Lexer &lex, int indent) {
    enterDebug("StructUnionDef");
    println_indent("Struct/Union Def", indent);

    if (tokenTryMatchNext(lex, tokenIdent) &&
        tokenTryMatchNext(lex, punctLBrace))
    {
        queue_clean(indent);
        // matching fields
        while (typeTryMatchNext(lex)) {
            if (tokenTryMatchNext(lex, tokenIdent) &&
                parserVarDef(lex, indent + 1))
            {}
            else debug_error();
        }
    }
}
```

```
    }

    if (tokenTryMatchNext(lex, punctRBrace)
        && tokenTryMatchNext(lex, punctSemicolon))
        ;
    else debug_error();
}
else debug_error();
queue_clean(indent);

exitDebug();
return true;
}

/* pre-matched:
   "enum"
*/

bool Parser::parserEnumDef(Lexer &lex, int indent) {
    enterDebug("EnumDef");
    println_indent("Enum Def", indent);

    queue_clean(indent);
    if (tokenTryMatchNext(lex, tokenIdent) &&
        tokenTryMatchNext(lex, punctLBrace))
    {
        while (tokenTryMatchNext(lex, tokenIdent)) {
            if (!tokenTryMatchNext(lex, punctComma))
                break;
        }
        for (token_pair &it : queue)
            println_indent(decoder(it.type) + ": " + it.token, indent + 1);
        queue.clear();

        if (tokenTryMatchNext(lex, punctRBrace) &&
```

```

        tokenTryMatchNext(lex, punctSemicolon))
        ;
    else debug_error();
}
queue_clean(indent);

exitDebug();
return true;
}

/* pre-matched:
    [<func-type>] <type> [{"*"}] <identifier> "("
*/
bool Parser::parserFuncDef(Lexer &lex, int indent) {
    enterDebug("FuncDef");
    println_indent("Func Def", indent);

    queue_clean(indent);
    // "void" is a bit different from other types
    if (tokenTryMatchNext(lex, keywordVoid) &&
        tokenTryMatchNext(lex, punctRParen))
    {
        queue_clean(indent);
        if (tokenTryMatchNext(lex, punctSemicolon))
            queue_clean(indent);
        else if (parserComplex(lex, indent + 1))
            ;
        else debug_error();
    }
    else if (parserArgList(lex, indent + 1) &&
        tokenTryMatchNext(lex, punctRParen))
    {
        queue_clean(indent);
        if (tokenTryMatchNext(lex, punctSemicolon))

```

```
        queue_clean(indent);
    else if (parserComplex(lex, indent + 1))
        ;
    else debug_error();
}
else debug_error();

exitDebug();
return true;
}

/* pre-matched:
*/

bool Parser::parserArgList(Lexer &lex, int indent) {
    enterDebug("ArgList");
    println_indent("Arg List", indent);

    if (parserArg(lex, indent + 1)) {
        while (tokenTryMatchNext(lex, punctComma)) {
            queue_clean(indent);
            if (!parserArg(lex, indent + 1))
                debug_error();
        }
    }
    else debug_error();

    exitDebug();
    return true;
}

/* pre-matched:
*/

bool Parser::parserArg(Lexer &lex, int indent) {
    enterDebug("Arg");
```

```

println_indent("Arg", indent);

if (typeTryMatchNext.lex)) {
    // if pointer
    while (tokenTryMatchNext.lex, punctTimes))
        ;
    if (!tokenTryMatchNext.lex, tokenId))
        debug_error();
    queue_clean(indent);
}
else debug_error();

exitDebug();
return true;
}

/* pre-matched:
*/

bool Parser::parserComplex(Lexer &lex, int indent) {
    enterDebug("Complex");
    println_indent("Complex", indent);
    bool ret = true;

    if (!tokenTryMatchNext.lex, punctLBrace))
        debug_error();
    queue_clean(indent);
    while (ret) {
        if (tokenTryMatchNext.lex, punctRBrace)) {
            queue_clean(indent);
            break;
        }
        // <var-def>
        else if (storageTypeTryMatchNext.lex) ||
            typeTryMatchNext.lex))

```

```

    {
        typeTryMatchNext(lex);
        ret = tokenTryMatchNext(lex, tokenIdent) &&
            parserVarDef(lex, indent + 1);
    }
    // <if-block>
    else if (tokenTryMatchNext(lex, keywordIf))
        ret = parserIf(lex, indent + 1);
    // <while-block>
    else if (tokenTryMatchNext(lex, keywordWhile))
        ret = parserWhile(lex, indent + 1);
    // <do-while-block>
    else if (tokenTryMatchNext(lex, keywordDo))
        ret = parserDoWhile(lex, indent + 1);
    // <for-block>
    else if (tokenTryMatchNext(lex, keywordFor))
        ret = parserFor(lex, indent + 1);
    // <jump>
    else if (tokenTryMatchNext(lex, keywordBreak)
        || tokenTryMatchNext(lex, keywordContinue)
        || tokenTryMatchNext(lex, keywordReturn))
        ret = parserJump(lex, indent + 1);
    // <expression> ";"
    else {
        ret = parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon);
        queue_clean(indent + 1);
    }

}

if (false == ret)
    debug_error();

exitDebug();

```



```
    return true;
}

/* pre-matched:
    "break" | "continue" | "return"
*/

bool Parser::parserJump(Lexer &lex, int indent) {
    enterDebug("Jump");
    println_indent("Jump", indent);

    queue_clean(indent);
    if (parserExpr(lex, indent + 1) && tokenTryMatchNext(lex, punctSemicolon))
        queue_clean(indent);
    else debug_error();

    exitDebug();
    return true;
}

/* pre-matched:
    "if"
*/

bool Parser::parserIf(Lexer &lex, int indent) {
    enterDebug("If");
    println_indent("If", indent);

    if (tokenTryMatchNext(lex, punctLParen))
        queue_clean(indent);
    else debug_error();
    if (parserExpr(lex, indent + 1) && tokenTryMatchNext(lex, punctRParen)) {
        queue_clean(indent);
    }
    else debug_error();
}
```

```

    bool ret = true;
    if (lex.get_type() == punctLBrace) {
        ret = parserComplex(lex, indent + 1);
    }
    else {
        ret = parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon);
        queue_clean(indent);
    }
    if (tokenTryMatchNext(lex, keywordElse)) {
        queue_clean(indent);
        if (tokenTryMatchNext(lex, keywordIf))
            ret = parserIf(lex, indent + 1);
        else if (lex.get_type() == punctLBrace)
            ret = parserComplex(lex, indent + 1);
        else {
            ret = parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon);
            queue_clean(indent);
        }
    }

    if (false == ret)
        debug_msg("invalid token " + lex.get_token());
    exitDebug();
    return ret;
}

/* pre-matched:
   "while"
*/

bool Parser::parserWhile(Lexer &lex, int indent) {
    enterDebug("While");
    println_indent("While", indent);

```

```
if (tokenTryMatchNext(lex, punctLParen))
    queue_clean(indent);
else debug_error();
if (parserExpr(lex, indent + 1) && tokenTryMatchNext(lex, punctRParen)) {
    queue_clean(indent);
}
else debug_error();

bool ret = true;
if (lex.get_type() == punctLBrace) {
    ret = parserComplex(lex, indent + 1);
}
else {
    ret = parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon);
    queue_clean(indent);
}

if (false == ret)
    debug_msg("invalid token " + lex.get_token());
exitDebug();
return ret;
}

/* pre-matched:
    "do"
*/

bool Parser::parserDoWhile(Lexer &lex, int indent) {
    enterDebug("DoWhile");
    println_indent("DoWhile", indent);
    queue_clean(indent);

    if (parserComplex(lex, indent + 1) &&
```

```

        tokenTryMatchNext(lex, keywordWhile) &&
        tokenTryMatchNext(lex, punctLParen))
    {
        queue_clean(indent);
        if (parserExpr(lex, indent + 1) &&
            tokenTryMatchNext(lex, punctRParen) &&
            tokenTryMatchNext(lex, punctSemicolon))
        {
            queue_clean(indent);
        }
        else debug_error();
    }
    else debug_error();

    exitDebug();
    return true;
}

/* pre-matched:
   "for"
*/

bool Parser::parserFor(Lexer &lex, int indent) {
    enterDebug("For");
    println_indent("For", indent);

    if (tokenTryMatchNext(lex, punctLParen)) {
        queue_clean(indent);
        // <var-def>
        if (storageTypeTryMatchNext(lex) || typeTryMatchNext(lex))
        {
            typeTryMatchNext(lex);
            if (tokenTryMatchNext(lex, tokenIdent) &&
                parserVarDef(lex, indent + 1))
            {}
        }
    }
}

```

```

        else debug_error();
    }

    if (parserExpr(lex, indent + 1) &&
        tokenTryMatchNext(lex, punctSemicolon))
        ;
    else debug_error();
    queue_clean(indent);
    if (parserExpr(lex, indent + 1) &&
        tokenTryMatchNext(lex, punctRParen))
        ;
    else debug_error();
    queue_clean(indent);
    if (lex.get_type() == punctLBrace)
        parserComplex(lex, indent + 1);
    else if (parserExpr(lex, indent + 1) && tokenTryMatchNext(lex,
punctSemicolon))
        queue_clean(indent);
    else debug_error();
}
else debug_error();

exitDebug();
return true;
}

static bool unaryTryMatchNext(Lexer &lex) {
    return tokenTryMatchNext(lex, punctPlus)
        || tokenTryMatchNext(lex, punctPlusPlus)
        || tokenTryMatchNext(lex, punctMinus)
        || tokenTryMatchNext(lex, punctMinusMinus)
        || tokenTryMatchNext(lex, punctTimes)
        || tokenTryMatchNext(lex, punctBitwiseAnd)
        || tokenTryMatchNext(lex, punctBitwiseNot)

```

```
    || tokenTryMatchNext(lex, punctLogicalNot)
    || tokenTryMatchNext(lex, keywordSizeof);
}

static bool binaryTryMatchNext(Lexer &lex) {
    return tokenTryMatchNext(lex, punctPlus)
        || tokenTryMatchNext(lex, punctPlusAssign)
        || tokenTryMatchNext(lex, punctMinus)
        || tokenTryMatchNext(lex, punctMinusAssign)
        || tokenTryMatchNext(lex, punctTimes)
        || tokenTryMatchNext(lex, punctTimesAssign)
        || tokenTryMatchNext(lex, punctDivide)
        || tokenTryMatchNext(lex, punctDivideAssign)
        || tokenTryMatchNext(lex, punctModulo)
        || tokenTryMatchNext(lex, punctModuloAssign)
        || tokenTryMatchNext(lex, punctAssign)
        || tokenTryMatchNext(lex, punctEqual)
        || tokenTryMatchNext(lex, punctLess)
        || tokenTryMatchNext(lex, punctLessEqual)
        || tokenTryMatchNext(lex, punctGreater)
        || tokenTryMatchNext(lex, punctGreaterEqual)
        || tokenTryMatchNext(lex, punctNotEqual)
        || tokenTryMatchNext(lex, punctShr)
        || tokenTryMatchNext(lex, punctShrAssign)
        || tokenTryMatchNext(lex, punctShl)
        || tokenTryMatchNext(lex, punctShlAssign)
        || tokenTryMatchNext(lex, punctBitwiseAnd)
        || tokenTryMatchNext(lex, punctLogicalAnd)
        || tokenTryMatchNext(lex, punctBitwiseAndAssign)
        || tokenTryMatchNext(lex, punctBitwiseOr)
        || tokenTryMatchNext(lex, punctLogicalOr)
        || tokenTryMatchNext(lex, punctBitwiseOrAssign)
        || tokenTryMatchNext(lex, punctBitwiseXor)
        || tokenTryMatchNext(lex, punctBitwiseXorAssign)
```

```
    || tokenTryMatchNext(lex, punctComma)
    || tokenTryMatchNext(lex, punctPeriod)
    || tokenTryMatchNext(lex, punctArrow);
}

static bool literalTryMatchNext(Lexer &lex) {
    return tokenTryMatchNext(lex, tokenInt)
        || tokenTryMatchNext(lex, tokenDouble)
        || tokenTryMatchNext(lex, tokenChar)
        || tokenTryMatchNext(lex, tokenStr);
}

/* pre-matched:
*/

bool Parser::parserExpr(Lexer &lex, int indent) {
    enterDebug("Expression");
    println_indent("Expression", indent);

    bool flag_paren = tokenTryMatchNext(lex, punctLParen);
    bool ret = true;
    // <unary-expr>
    if (unaryTryMatchNext(lex))
        ret = parserUnaryExpr(lex, indent + 1);
    // <literal>
    else if (literalTryMatchNext(lex))
        queue_clean(indent);
    // <identfier> or <func-call>
    else if (tokenTryMatchNext(lex, tokenIdent)) {
        if (tokenTryMatchNext(lex, punctLParen))
            ret = parserFuncCall(lex, indent + 1);
        else
            queue_clean(indent);
    }
}
```

```

// [{ <binary-op> <expression> }]
while (ret && binaryTryMatchNext(lex)) {
    queue_clean(indent);
    ret = parserExpr(lex, indent + 1);
}
// "?" <expression> ":" <expression>
if (ret && tokenTryMatchNext(lex, punctQuestion)) {
    queue_clean(indent);
    ret = parserExpr(lex, indent + 1);
    if (!tokenTryMatchNext(lex, punctColon))
        debug_error();
    queue_clean(indent);
    ret = parserExpr(lex, indent + 1);
}

if (flag_paren) {
    if (tokenTryMatchNext(lex, punctRParen))
        queue_clean(indent);
    else
        debug_error();
}

if (false == ret)
    debug_error();
exitDebug();
return ret;
}

/* pre-matched:
    <unary-op>
*/

bool Parser::parserUnaryExpr(Lexer &lex, int indent) {
    enterDebug("UnaryExpr");

```



```
println_indent("Unary Expression", indent);

queue_clean(indent);
if (!parserExpr(lex, indent + 1))
    debug_error();

exitDebug();
return true;
}

/* pre-matched:
   <identifier> "("
*/
bool Parser::parserFuncCall(Lexer &lex, int indent) {
    enterDebug("FuncCall");
    println_indent("Function Call", indent);

    if (tokenTryMatchNext(lex, tokenIdent) ||
        literalTryMatchNext(lex))
    {
        while (tokenTryMatchNext(lex, punctComma)) {
            if (tokenTryMatchNext(lex, tokenIdent) ||
                literalTryMatchNext(lex))
                ;
            else debug_error();
        }
    }

    if (tokenTryMatchNext(lex, punctRParen))
        queue_clean(indent);
    else debug_error();

    exitDebug();
    return true;
}
```

```
}  
// Code/src/format.cpp  
#include <deque>  
#include <string>  
  
#include "../inc/format"  
  
#pragma GCC dependence "../inc/format"  
  
static std::deque<std::string> s;  
  
static void enterDebug(std::string str) {  
    s.push_back(str);  
}  
  
static void debug_msg(std::string str) {  
    std::cerr << s.back() << ": "  
        << str << std::endl;  
}  
  
static void debug_msg(char *str) {  
    std::cerr << s.back() << ": "  
        << str << std::endl;  
}  
  
static void exitDebug(void) {  
    s.pop_back();  
}  
  
#define debug_error() \  
do { \  
    debug_msg("invalid token " + lex.get_token()); \  
    exitDebug(); \  
    return false; \  
}
```

```
    } while (false)

class token_pair {
public:
    token_t type;
    std::string token;

    token_pair(token_t tp, std::string tk):
        type(tp), token(tk) {}
};

static std::deque<token_pair> queue;

static void tokenMatchNext(Lexer &lex) {
    queue.push_back(token_pair(lex.get_type(), lex.get_token()));
    lex.next();
}

static bool tokenTryMatchNext(Lexer &lex, token_t type) {
    if (type == lex.get_type()) {
        tokenMatchNext(lex);
        return true;
    }
    else return false;
}

static bool storageTypeTryMatchNext(Lexer &lex) {
    return tokenTryMatchNext(lex, keywordConst)
        || tokenTryMatchNext(lex, keywordAuto)
        || tokenTryMatchNext(lex, keywordStatic)
        || tokenTryMatchNext(lex, keywordVolatile)
        || tokenTryMatchNext(lex, keywordExtern);
}
```

```
static bool typeTryMatchNext(Lexer &lex) {
    bool ret = tokenTryMatchNext(lex, keywordVoid)
        || tokenTryMatchNext(lex, keywordChar)
        || tokenTryMatchNext(lex, keywordShort)
        || tokenTryMatchNext(lex, keywordInt)
        || tokenTryMatchNext(lex, keywordUnsigned)
        || tokenTryMatchNext(lex, keywordSigned)
        || tokenTryMatchNext(lex, keywordLong)
        || tokenTryMatchNext(lex, keywordFloat)
        || tokenTryMatchNext(lex, keywordDouble);

    if (!ret && tokenTryMatchNext(lex, keywordStruct) ||
        tokenTryMatchNext(lex, keywordUnion) ||
        tokenTryMatchNext(lex, keywordEnum))
    {
        ret = tokenTryMatchNext(lex, tokenIdent);
    }

    return ret;
}

bool Format::formatProgramme(Lexer &lex, int indent) {
    enterDebug("Programme");
    bool ret;

    while (lex.get_type() != tokenEOF) {
        // struct-union-def
        if (lex.get_type() == keywordStruct ||
            lex.get_type() == keywordUnion)
        {
            stream << lex.get_token() << " ";
            lex.next();
            if (lex.get_type() != punctLBrace)
                debug_error();
        }
    }
}
```

```
stream << lex.get_token();
lex.next();

while (lex.get_type() != punctRBrace) {
    for (int i = 0; i < indent + 1; i++)
        stream << '\t';
    while (true) {
        stream << lex.get_token();
        lex.next();
        if (lex.get_type() != punctSemicolon)
            stream << " ";
        else {
            stream << lex.get_token() << std::endl;
            lex.next();
            break;
        }
    }
}
stream << lex.get_token();
lex.next();
if (lex.get_type() != punctSemicolon)
    debug_error();
else {
    stream << ";\n\n";
    lex.next();
}
}

// enum-def
else if (lex.get_type() == keywordEnum) {
    lex.next();
    stream << "enum " << lex.get_token();
    lex.next();
    if (lex.get_type() != punctLBrace)
        debug_error();
}
```

```

        stream << "{\n";
        lex.next();
        while (lex.get_type() != punctRBrace) {
            for (int i = 0; i < indent + 1; i++)
                stream << "\t";
            while (true) {
                stream << lex.get_token();
                lex.next();
                if (lex.get_type() != punctComma)
                    stream << ' ';
                else {
                    stream << ",\n";
                    lex.next();
                    break;
                }
            }
        }
    }
}

// var-def | func-def
else if (storageTypeTryMatchNext(lex)
        || typeTryMatchNext(lex))
{
    typeTryMatchNext(lex);

    while (lex.get_type() != punctSemicolon &&
            lex.get_type() != punctLBrace)
    {
        tokenMatchNext(lex);
    }
    for (token_pair &it : queue)
        stream << it.token << " ";
    queue.clear();
    if (lex.get_type() == punctLBrace) {
        stream << lex.get_token() << std::endl;
    }
}

```

```
        lex.next();
        ret = formatComplex(lex, indent + 1);
        stream << "\n\n";
    }
    else {
        stream << lex.get_token() << std::endl;
        lex.next();
        ret = true;
    }
}

else debug_error();
}

exitDebug();
return ret;
}

/* pre-print:
   "{"
*/
bool Format::formatComplex(Lexer &lex, int indent) {
    enterDebug("Complex");

    bool ret = true;

    while (lex.get_type() != punctRBrace) {
        for (int i = 0; i < indent; i++)
            stream << '\t';
        switch (lex.get_type()) {
            case keywordIf:
                ret = formatIf(lex, indent + 1);
                break;
            case keywordWhile:
                ret = formatWhile(lex, indent + 1);
```

```
        break;
    case keywordDo:
        ret = formatDoWhile(lex, indent + 1);
        break;
    case keywordFor:
        ret = formatFor(lex, indent + 1);
        break;
    default:
        while (true) {
            stream << lex.get_token();
            lex.next();
            if (lex.get_type() != punctSemicolon)
                stream << ' ';
            else {
                stream << ";\n";
                lex.next();
                break;
            }
        }
        break;
    }
}

for (int i = 0; i < indent - 1; i++)
    stream << '\t';
stream << '}';
lex.next();

if (false == ret) debug_error();

exitDebug();
return ret;
}
```

```
bool Format::formatIf(Lexer &lex, int indent) {
```



```
enterDebug("If");

bool ret = true;

stream << "if (";
lex.next(); lex.next();

int cnt = 1;
while (cnt > 0) {
    if (lex.get_type() == punctLParen) cnt ++;
    else if (lex.get_type() == punctRParen) cnt --;
    stream << lex.get_token();
    lex.next();
}

if (lex.get_type() == punctLBrace) {
    stream << "{\n";
    lex.next();
    ret = formatComplex(lex, indent);
    stream << std::endl;
}
else {
    stream << std::endl;
    ret = formatExpr(lex, indent);
}

if (lex.get_type() == keywordElse) {
    for (int i = 0; i < indent - 1; i ++)
        stream << "\t";
    stream << "else ";
    lex.next();
    if (lex.get_type() == punctLBrace) {
        stream << "{\n";
        lex.next();
```

```
        ret = formatComplex(lex, indent);
        stream << std::endl;
    }
    else if (lex.get_type() == keywordIf)
        ret = formatIf(lex, indent);
    else {
        stream << std::endl;
        ret = formatExpr(lex, indent);
    }
}

if (false == ret)
    debug_error();

exitDebug();
return ret;
}

bool Format::formatWhile(Lexer &lex, int indent) {
    enterDebug("While");

    bool ret = true;

    stream << "while (";
    lex.next(); lex.next();

    int cnt = 1;
    while (cnt > 0) {
        if (lex.get_type() == punctLParen) cnt ++;
        else if (lex.get_type() == punctRParen) cnt --;
        stream << lex.get_token();
        lex.next();
    }
}
```

```
if (lex.get_type() == punctLBrace) {
    stream << " {\n";
    lex.next();
    ret = formatComplex(lex, indent);
    stream << std::endl;
}
else {
    stream << std::endl;
    ret = formatExpr(lex, indent);
}

if (false == ret)
    debug_error();

exitDebug();
return ret;
}

bool Format::formatDoWhile(Lexer &lex, int indent) {
    enterDebug("DoWhile");

    stream << "do {";
    lex.next(); lex.next();

    if (lex.get_type() == punctRBrace) {
        stream << "} while (";
        lex.next();lex.next();lex.next();
    }
    else {
        stream << std::endl;
        formatComplex(lex, indent);
        stream << " while (";
        lex.next(); lex.next();
    }
}
```

```
while (lex.get_type() != punctSemicolon) {
    stream << lex.get_token();
    lex.next();
}
stream << ";\n";
lex.next();

exitDebug();
return true;
}

bool Format::formatFor(Lexer &lex, int indent) {
    enterDebug("For");

    stream << "for (";
    lex.next(); lex.next();

    while (true) {
        stream << lex.get_token();
        lex.next();
        if (lex.get_type() != punctSemicolon)
            stream << ' ';
        else {
            stream << "; ";
            lex.next();
            break;
        }
    }
}

while (true) {
    stream << lex.get_token();
    lex.next();
    if (lex.get_type() != punctSemicolon)
        stream << ' ';
```

```
        else {
            stream << "; ";
            lex.next();
            break;
        }
    }
    while (true) {
        stream << lex.get_token();
        lex.next();
        if (lex.get_type() != punctRParen)
            stream << ' ';
        else {
            stream << ") ";
            lex.next();
            break;
        }
    }

    if (lex.get_type() == punctLBrace) {
        stream << "{\n";
        lex.next();
        formatComplex(lex, indent);
        stream << std::endl;
    }
    else {
        stream << std::endl;
        formatExpr(lex, indent);
    }

    exitDebug();
    return true;
}
```

```
bool Format::formatExpr(Lexer &lex, int indent) {
```

```
enterDebug("Expression");

for (int i = 0; i < indent; i++)
    stream << '\t';
while (true) {
    stream << lex.get_token();
    lex.next();
    if (lex.get_type() != punctSemicolon)
        stream << ' ';
    else {
        stream << ";\n";
        lex.next();
        break;
    }
}

exitDebug();
return true;
}

// Code/src/main.cpp
#include <fstream>
#include <iostream>
#include <cstring>

#include "../inc/parser"
#include "../inc/lexer"
#include "../inc/format"

#define STR_EQU(pstr1, pstr2) \
    !strcmp(pstr1, pstr2)

int main(int argc, char *argv[]) {
    if (1 == argc) { // display usage
        std::cout << "Usage: " << argv[0] << "\n"
```

```
<< " -f in_file" << "\t\tassign input file\n"
<< " -o out_file" << "\t\tassign output file\n"
<< " --format" << "\t\tmode = format\n"
<< " --ast" << "\t\tmode = parser\n"
<< " -h" << "\t\tdisplay this page" << std::endl;

return 0;
}

std::string infile = "";
std::string outfile = "";
bool mode = true;          // mode = true, parser; mode = false, format

for (int i = 1; i < argc; i++) {
    if (STR_EQU(argv[i], "-f"))
        infile = argv[++i];
    else if (STR_EQU(argv[i], "-o"))
        outfile = argv[++i];
    else if (STR_EQU(argv[i], "--format"))
        mode = false;
    else if (STR_EQU(argv[i], "--ast"))
        mode = true;
    else if (STR_EQU(argv[i], "-h")) {
        std::cout << "Usage: " << argv[0] << "\n"
            << " -f in_file" << "\t\tassign input file\n"
            << " -o out_file" << "\t\tassign output file\n"
            << " --format" << "\t\tmode = format\n"
            << " --ast" << "\t\tmode = parser\n"
            << " -h" << "\t\tdisplay this page" << std::endl;
        return 0;
    }
    else {
        std::cerr << "invalid flag: " << argv[i]
            << std::endl;
        return -1;
    }
}
```

```
    }  
}  
  
std::ifstream istrm(infile);  
if (!istrm.is_open()) {  
    std::cerr << "warning: invalid input file"  
    << std::endl;  
    return -1;  
}  
  
Lexer lex(istrm);  
lex.next();  
std::ofstream ostrm(outfile);  
if (ostrm.is_open()) {  
    if (mode) {  
        Parser p(ostrm);  
        p.parserProgramme(lex);  
    }  
    else {  
        Format f(ostrm);  
        f.formatProgramme(lex);  
    }  
}  
else {  
    if (mode) {  
        Parser p(ostrm);  
        p.parserProgramme(lex);  
    }  
    else {  
        Format f(ostrm);  
        f.formatProgramme(lex);  
    }  
}  
return 0;
```



}