北京郵電大學

实验报告

题目：NaiveLex

——一款朴素的 C 语言词法分析器的实现

班　　级：2018211312

学　　号：马伟健

姓　　名：2018211611

学　　院：计算机学院

2020 年 9 月 7 日

一、 摘要。

NaiveLex 是一款基于面向对象的程序设计思想开发的 C 语言词法分析器，其只需一遍扫描，就可以完成输入程序中所有单词符号的分析，同时找出其中的错误。分析器依照的 C 语言规范为针对 C17 最新的 ISO C N2176 草案[1]。

二、 简介。

2.1 实验内容

NaïveLex 实现的功能具体如下：

- 识别出 C 语言编写的源程序中的每一个单词符号，并以记号形式输出他们。
- 跳过原程序中的注释。
- 检查程序中存在的词法错误，并报告错误的位置。
- 通过对错误进行适当恢复，从而达到只需要一次扫描就可以报告程序中存在的全部错误的目标。

2.2 实验环境。

操作系统：Windows 10
编辑器：Visual Studio Code 1.49.3 + C/C++ Intellisense v1.0.1
编译器：MinGW 6.3.0
编程语言：C++ 17

三、 详细设计方案。

通常而言，用 C/C++手工实现的词法分析器采用根据语言规范，手工实现对应的自动机的方法。即通过自动机识别 token，再将 token 归类为不同的类别并记录。有关 C 语言中对于 token 的完整定义，可在 ISO C N2176 草案[1]的 6.4 节中找到。

3.1 本实验对于 N2176 草案的处理。

本实验并没有完全按照 ISO C N2176 草案进行实现，而是参考了课程内容以及主流开源 C 编译器的实现，对于标准内的定义进行了不改变主要功能的修改以及简化。

主要变动如下所示：

A. 对于预处理器相关的内容，不做具体解析。N2176 草案中，对于预处理相关的 token 的处理方式为将其细分为不同的类别.（*The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories. N2176 pp.42*）

但是事实上，不论是课程内容，还是主流 C 编译器都将词法分析与语法分析

/预处理分成了两个不同的阶段。严格来说对于这一级的类别检测并非词法分析器的内容，由此，Naïve Lex 将与预处理相关的部分统一构成一个名为 PreprocessingDirective 的 Token，而将 token 拆分、替换的过程留待后续处理进行完成。

B. 处理标识符（Identifier）时，不对通用字符名（Universal Character Names，即通常所说的 Unicode，其被 C11 标准所采用）编码方式的字符进行识别。

C. 将常数中的浮点常数（FloatingConstant）与整型常数（IntegerConstant）进行合并，NaiveLex 中统一将其用 NumericConstant 表示，这样既不过度影响程序功能，又使得判断常数的过程可以直接调用 C++库函数，从而简化程序。

D. 对于字符串的处理不考虑字符串内部的换行。

3.2 对于错误的处理方案。

事实上，词法分析能够识别的错误十分有限。常见的可以识别的错误仅有不合法的标识符以及含有不可识别的字符的情况。

若是出现含有不合法的标识符的情况，其可分为两种，第一种为变量名不以字母或下划线开头，此时我们将分为两种情况。

A. 变量名以数字开头，此时我们将试图将其理解成一个数字常量(NumericConstant)，若是理解失败，我们将会把它理解成一个错误的数字常量(NumericConstantWithError)，同时尽量将错误的字符吞掉。

B. 变量名以不可识别的字符开头。此时我们将直接将不可识别字符单独生成一个名为 Unknown 的 token。

不合法的标识符的另一种情况为标识符内出现了不可识别的字符，我们此时将该字符单独识别为一个 Unknown Token，同时将合法字符另外进行识别。

同样的，对于直接出现的不可识别的字符，我们也是直接将其生成一个 Unknown Token。

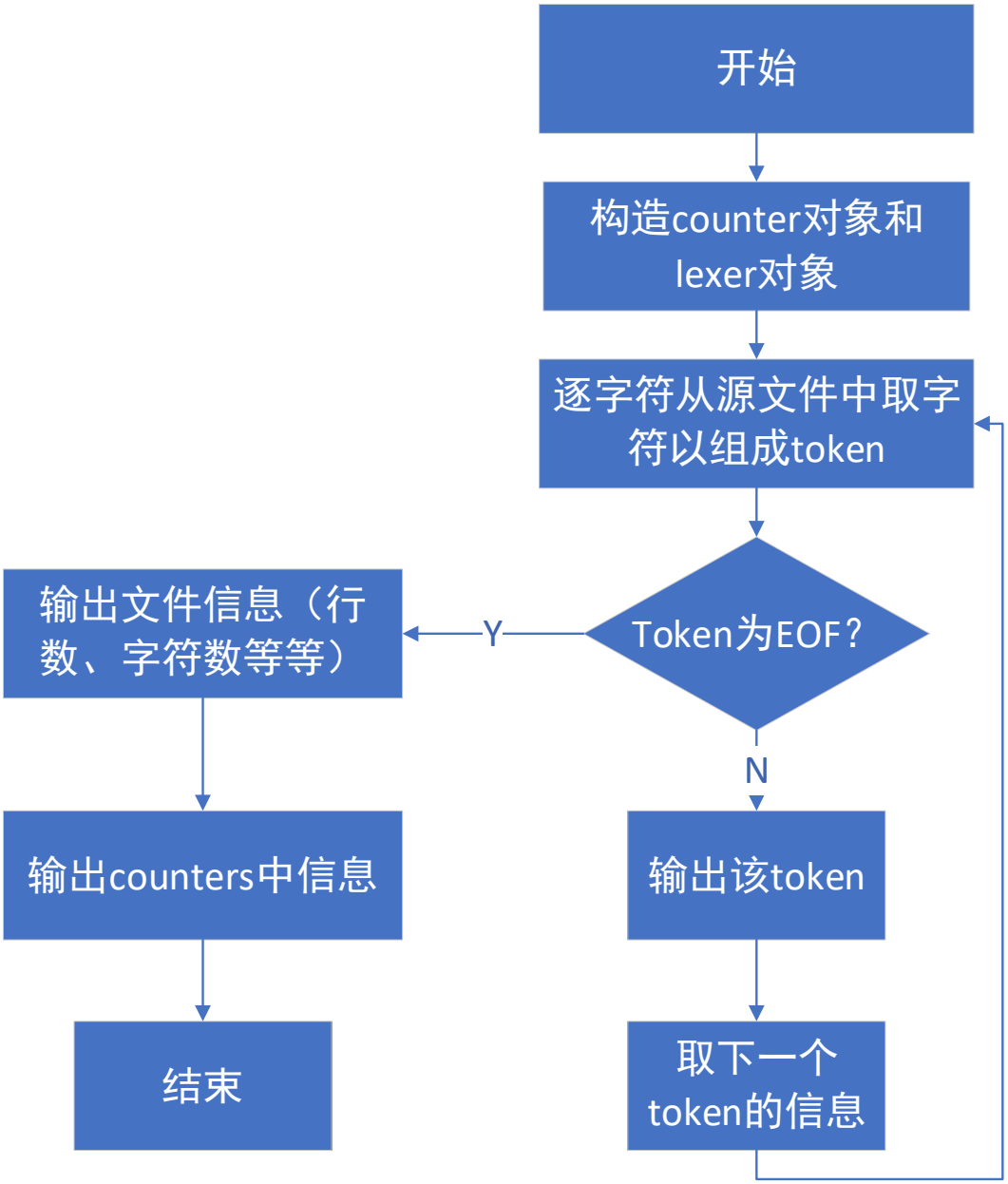
从实际编程经验看，一般导致词法分析层面的错误的，通常为误插入的非法字符，或是出现在标识符首位的非下划线及字母。这些我们通常都可以使用 Unknown token 与 NumericConstantWithError 将其快速检测出，并做兼顾高效性与合理性的纠正。

3.3 程序整体流程图。

NaïveLex 程序整体流程如下图所示：

可以看到，NaiveLex 程序的主体对象为 lexer 和 counter 两个。此二者作用分别如下：

Counter 对象负责维护整个待分析程序中各个对象的数目。而 lexer 对象则负责不断从源文件中读取并生成不同类型的 token，同时将这些 token 输出到指定位置。



3.4 程序模块划分

3.4.1 naiveLex 对象

该对象为整个词法分析器的主要部分，这部分从源程序输入流中不断读取字符，通过一个大型的自动机，将其匹配为标识符、标点、数值常量、字符串等等的多种类别，并将其输出。该类定义如下：

```
class naiveLex {
    void skip_line_comment();
    void skip_block_comment();
    Token get_next_numeric_token();
    Token get_next_string_literial_token();
    Token get_next_character_constant_token();
```

```
    Token get_next_identifier_token();
    Token get_next_preprocessing_directive_token();
    FileWrapper fileWrapper_;
    TokenLoc currentLocation_;
    std::string tokenBuffer_;
public:
    explicit naiveLex(const std::string& fileName) : fileWrapper_{fileN
ame} {}
    Token getNextToken();
    size_t getCount(){          return fileWrapper_.getCount();
}
    size_t getLineCount(){      return fileWrapper_.getLineCount();
}
    std::string getSrcName(){   return fileWrapper_.getName();
}
};
```

其中，getNextToken()为词法分析器的核心函数。它的主体为一个由 switch case 组成的大型自动机。其返回一个 token 类，作为从文件输入流中读出的 token。函数中的自动机通过每一个符号的首字符来判断这个符号可能的类别，同时对于特定符号专门设置的辅助函数，来对整个输入文件进行超前读取，从而判断输入符号的属性。

3.4.2 token 对象。

该对象为词法分析器的基本单位，用于承载每一个输入组成的基本对象。其类定义如下所示：

```
class Token {
    TokenType token_type_;
    TokenLoc token_loc_;

    KeyWord keyword_;
    Punctuator punctuator_;
    std::string  literalValue_;
public:
    Token(TokenType token_type, TokenLoc token_loc, KeyWord keyword, Pu
nctuator punctuator, std::string literalValue)
    {
        token_type_ = token_type;
        token_loc_ = std::move(token_loc);
        keyword_ = keyword;
        punctuator_ = punctuator;
        literalValue_ = std::move(literalValue);
    }
```

```
    Token(TokenType token_type, const TokenLoc& token_loc, KeyWord keyw
ord) :
            Token(token_type, token_loc, keyword, Punctuator::None, "")
{
        assert(token_type == TokenType::Keyword);
    }

    Token(TokenType token_type, TokenLoc token_loc, Punctuator punctuat
or) :
            Token(token_type, std::move(token_loc), KeyWord::None, punc
tuator, "") {
        assert(token_type == TokenType::Punctuator);
    }

    Token(TokenType token_type, TokenLoc token_loc, std::string literal
Value) :
            Token(token_type, std::move(token_loc), KeyWord::None, Punc
tuator::None, std::move(literalValue)){
        assert(token_type != TokenType::Keyword && token_type != TokenT
ype::Punctuator);
    }

    Token() : Token(TokenType::Unknown, TokenLoc(), KeyWord::None, Punc
tuator::None, "") {}

    TokenType getTokenType()            const {     return token_type_;
        }
    TokenLoc getTokenLoc()              const {     return token_loc_;
   }
    KeyWord getKeyWord()                const {     return keyword_;
       }
    Punctuator getPunctuator()          const {     return punctuator_;
       }
    std::string getLiteralValue()       const {     return literalValue
_;     }

    std::string toStr() const;
    friend std::ostream& operator<<(std::ostream& os, const Token & tok
en);
};
```

其中，数据成员 tokenType，keyword，punctuator 分别为源自 N2176 的 C 语言中符号类型、关键字、符号的形式，以枚举常量形式存放在源代码中。TokenLoc 为存放 Token 变量的位置的对象，另外还有一个存放字符串常量的 std:: string

类型的缓冲区。

函数成员主要为针对不同输入数据成员的构造函数，以及分别将其转化成字符串类型的 to_str 和输出运算符<<重载。

3.4.3 Tokenloc 对象。

如上所述，该对象存放记号的位置。具体如下所示：

```
class TokenLoc {
    std::string fileName;
    size_t colCount;
    size_t lineCount;
public:
    TokenLoc()
    {
        fileName = "UNKNOWN";
        colCount = 0;
        lineCount = 1;
    }
    TokenLoc(std::string file, size_t col, size_t line)
    {
        fileName = std::move(file);
        colCount = col;
        lineCount = line;
    }
    size_t getColCount()    const {    return colCount;    }
    std::string toStr()  const {    return fileName + ":" + std::to_str
ing(lineCount) + ":" + std::to_string(colCount);   }
};
```

3.4.4 Counter 对象。

如前所述，该对象维护输入程序的统计信息，私有数据成员存放着各类符号类型的个数，同时也存放着输出以及获取各类私有成员的接口。具体定义如下所示：

```
class Counter {
    size_t count_keyword;
    size_t count_identifier;
    size_t count_numeric_constant;
    size_t count_numeric_constant_with_error;
    size_t count_character_constant;
    size_t count_string_literal;
    size_t count_punctuator;
    size_t count_preprocessing_directive;
    size_t count_EOF;
    size_t count_unknown;
public:
```

```
    void update(const Token& token);
    friend std::ostream& operator<<(std::ostream& os, const Counter & c
ounter);
    size_t get_keyword() const                      {   return  count_k
eyword; }
    size_t get_numeric_constant() const             {   return  count_n
umeric_constant; }
    size_t get_identifier() const                   {   return  count_i
dentifier; }
    size_t get_numeric_constantWithError() const    {   return  count_n
umeric_constant_with_error; }
    size_t get_character_constant() const           {   return  count_c
haracter_constant; }
    size_t get_string_literal() const               {   return  count_s
tring_literal; }
    size_t get_punctuator()  const                  {   return  count_p
unctuator; }
    size_t get_preprocessing_directive() const      {   return  count_p
reprocessing_directive; }
    size_t get_EOF() const                          {   return  count_E
OF; }
    size_t get_unknown()  const                     {   return  count_u
nknown; }
};
```

3.4.5 FileWrapper 对象

该对象对于输入文件进行预处理，使得后续的 token 读取变得简单。

该对象提供了我们所需要的字符的位置、查看后续指定个数个字符、跳过后续指定个数个字符等等的功能。其主要数据结构为一个 ifstream 和一个由字符组成的双端对列 buffer_，类定义如下所示：

```
class FileWrapper {
private:
    std::string fileName_;
    std::ifstream sourceFile_;
    std::deque<char> buffer_;
    bool eof_;
    size_t line_;
    size_t column_;
    size_t count_;

void read()
{
    ……
}
```

```
public:
FileWrapper(std::string fileName)
{
    ……
}

char getNextChar()
{
    ……
}

char peekChar(size_t const offset)
{
    ……
}

void eatChars(size_t const num)
{
    ……
}

bool eof()              {    return eof_ && buffer_.empty(); }

size_t getCount()       {   return count_; }
size_t getColumn()      {   return column_; }
size_t getLineCount()   {   return line_; }
std::string getName()   {   return fileName_; }

friend std::ostream& operator<<(std::ostream& os, FileWrapper & fileWra
pper);
TokenLoc getLocation();
};
```

各个函数功能如下所示：

FileWrapper(std::string fileName) 类构造函数，打开文件并将其重定向至一个输入流。

char getNextChar() 获取下一个字符并更新文件对应位置

char peekChar(size_t const offset) 查看后续第 offset 个字符

void eatChars(size_t const num) 跳过 num 个字符。

bool eof() 判断文件是否结束。

void read() 从输入流读取一个字符到队列内。

3.4.6 main

程序的对外接口，串联起各个模块同时进行输入输出。具体实现如下：

```
int main(int argc, char** argv) {
```

```
    if(argc < 2) {
        usage();
        return 0;
    }
    auto counter = Counter();
    auto lex_op = naiveLex(argv[1]);
    std::cout << "Tokens:" << std::endl;
    Token tok = lex_op.getNextToken();
    while(tok.getTokenType() != TokenType::EndOfFile) {
        std::cout << tok << std::endl;
        counter.update(tok);
        tok = lex_op.getNextToken();
    }
    std::cout << "File info:" << std::endl;
    std::cout << "File name:\t" << lex_op.getSrcName() << std::endl;
    std::cout << "Total chars:\t" << lex_op.getCount() << std::endl;
    std::cout << "Total lines:\t" << lex_op.getLineCount() << std::endl
;
    std::cout << counter;
    return 0;
}
```

可以看到，main 函数的主要实现逻辑如流程图所示，为获得 token——判断类别——输出 token——更新 counter——获得 token 的循环。

四、 程序的编译与使用。

4.1 程序的编译方法。

本程序编译方法很简单，只需要在装有 gcc 的控制台中键入以下命令：

```
gcc main.cpp token.cpp counter.cpp file.cpp lexer.cpp lexer.h file.h token.h -o naiveLex
```

一个可以运行的 naiveLex 就被生成了。

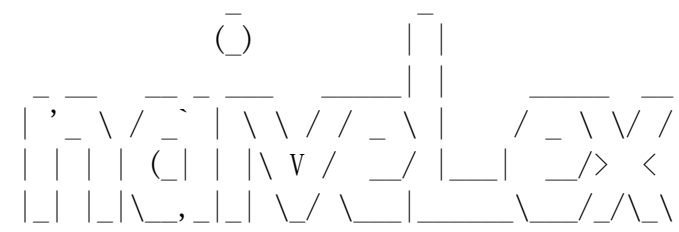
4.2 程序的使用方法。

本程序基于命令行 CLI 进行运行，程序的默认输出为标准输出。如需将输出导入文件，可以在命令行中使用>>进行重定向。

一个典型的 naiveLex 的使用方法如下：

```
naiveLex <input file name> [output file name]
```

其中输入文件为必填，输出文件为选填。若不填输出文件，那么结果将会在标准输出流中输出。而若填写输出文件，结果则会被重定向到输出文件。其如下所示：



```
Welcome to naiveLex.
The output file has been redirected to correct.pch
```

4.2 程序测试用例。

A. 首先我们测试一个正确的程序，它名叫 correct.c 其如下所示：

```c
#include <stdio.h>

# define FOO(A, B, ..., format, ...) \
if(X == Y) { \
    fprintf(stderr, format __W32API_MAJOR_VERSION(,) __ARGS__); \
}

// Main Function
int main(int argc, char *argv[])
{
    /* This
     * is a comment. */
    unsigned long long a = 114514Ull;
    double b = .24e+10f;
    double c = 0xab.cdp+18f;
    __uint128_t c = ++a;
    a >> 4;
    return 0;
}

./naiveLex correct.c
```

其分析结果如下所示：

```
Tokens:
Type: PreprocessingDirective            Location: correct.c:1:1
 Value: include <stdio.h>
Type: PreprocessingDirective            Location: correct.c:3:1
 Value:  define FOO(A, B, ..., format, ...) \
if(X == Y) { \
    fprintf(stderr, format __W32API_MAJOR_VERSION(,) __ARGS__); \
}
Type: KeyWord           Location: correct.c:9:1         Value: int
```

```
Type: Identifier                Location: correct.c:9:5         Value:
main
Type: Punctuator                Location: correct.c:9:9         Value:
(
Type: KeyWord           Location: correct.c:9:10                Value:
int
Type: Identifier                Location: correct.c:9:14
 Value: argc
Type: Punctuator                Location: correct.c:9:18
 Value: ,
Type: KeyWord           Location: correct.c:9:20                Value:
char
Type: Punctuator                Location: correct.c:9:25
 Value: *
Type: Identifier                Location: correct.c:9:26
 Value: argv
Type: Punctuator                Location: correct.c:9:30
 Value: [
Type: Punctuator                Location: correct.c:9:31
 Value: ]
Type: Punctuator                Location: correct.c:9:32
 Value: )
Type: Punctuator                Location: correct.c:10:1
 Value: {
Type: KeyWord           Location: correct.c:13:5                Value:
unsigned
Type: KeyWord           Location: correct.c:13:14               Value:
long
Type: KeyWord           Location: correct.c:13:19               Value:
long
Type: Identifier                Location: correct.c:13:24
 Value: a
Type: Punctuator                Location: correct.c:13:26
 Value: =
Type: NumericConstant           Location: correct.c:13:28
 Value: 114514Ull
Type: Punctuator                Location: correct.c:13:37
 Value: ;
Type: KeyWord           Location: correct.c:14:5                Value:
double
Type: Identifier                Location: correct.c:14:12
 Value: b
Type: Punctuator                Location: correct.c:14:14
 Value: =
```

```
Type: NumericConstant           Location: correct.c:14:16
 Value: .24e+10f
Type: Punctuator                Location: correct.c:14:24
 Value: ;
Type: KeyWord           Location: correct.c:15:5                Value:
double
Type: Identifier                Location: correct.c:15:12
 Value: c
Type: Punctuator                Location: correct.c:15:14
 Value: =
Type: NumericConstant           Location: correct.c:15:16
 Value: 0xab.cdp+18f
Type: Punctuator                Location: correct.c:15:28
 Value: ;
Type: Identifier                Location: correct.c:16:5
 Value: __uint128_t
Type: Identifier                Location: correct.c:16:17
 Value: c
Type: Punctuator                Location: correct.c:16:19
 Value: =
Type: Punctuator                Location: correct.c:16:21
 Value: ++
Type: Identifier                Location: correct.c:16:23
 Value: a
Type: Punctuator                Location: correct.c:16:24
 Value: ;
Type: Identifier                Location: correct.c:17:5
 Value: a
Type: Punctuator                Location: correct.c:17:7
 Value: >>
Type: NumericConstant           Location: correct.c:17:10
 Value: 4
Type: Punctuator                Location: correct.c:17:11
 Value: ;
Type: NumericConstant           Location: correct.c:18:12
 Value: 0
Type: Punctuator                Location: correct.c:18:13
 Value: ;
Type: Punctuator                Location: correct.c:19:1
 Value: }
File info:
File name:      correct.c
Total chars:    376
Total lines:    19
```

```
KeyWord:        9
Identifier:     10
NumericConstant:        5
NumericConstant(Error Detected):        0
CharacterConstant:      0
StringLiteral:  0
Punctuator:     20
PreprocessingDirective: 2
EndOfFile:      0
Unknown:        0
```

我们可以发现，除了因为不属于 C++11 标准草案中的__uint128_t 关键字被识别成了标识符之外，其余识别均正确。同时对于文件内容的统计信息也是正确的。由于__uint128_t 为 gcc 等编译器自行添加的关键字，不属于 N2176 标准，因此我们不予识别。

由此，我们可以认为 naiveLex 出色地完成了对于符合 N2176 标准的 C 语言源程序的词法分析。

B. 随后我们测试一个在词法层面有错误的程序，其名为 faulty.c，内容如下所示：

```c
#include <no/such/header.h>

int main(bool argc, char argv, float foo)
{
    int 1caz;
    double xsw2;

    double d = 0xabcdesf.achp+ef;
    return -1;
}
```

我们可以发现，在词法分析层面，可以检测出来的错误仅限于不符合命名方法的标识符 1qaz，以及非法的常量 0xabcdesf.achp+ef。将其输入 naiveLex，得到的结果如下：

```
Tokens:
Type: PreprocessingDirective            Location: fualty.c:1:1
 Value: include <no/such/header.h>
Type: KeyWord           Location: fualty.c:3:1          Value: int
Type: Identifier                Location: fualty.c:3:5          Value:
main
Type: Punctuator                Location: fualty.c:3:9          Value:
(
Type: Identifier                Location: fualty.c:3:10         Value:
bool
Type: Identifier                Location: fualty.c:3:15         Value:
argc
```

```
Type: Punctuator                Location: fualty.c:3:19         Value:
,
Type: KeyWord           Location: fualty.c:3:21         Value: char
Type: Identifier                Location: fualty.c:3:26         Value:
argv
Type: Punctuator                Location: fualty.c:3:30         Value:
,
Type: KeyWord           Location: fualty.c:3:32         Value: float
Type: Identifier                Location: fualty.c:3:38         Value:
foo
Type: Punctuator                Location: fualty.c:3:41         Value:
)
Type: Punctuator                Location: fualty.c:4:1          Value:
{
Type: KeyWord           Location: fualty.c:5:5          Value: int
Type: NumericConstantWithError          Location: fualty.c:5:9
 Value: 1ca
Type: Identifier                Location: fualty.c:5:12         Value:
a
Type: Punctuator                Location: fualty.c:5:13         Value:
;
Type: KeyWord           Location: fualty.c:6:5          Value: double
Type: Identifier                Location: fualty.c:6:12         Value:
xsw2
Type: Punctuator                Location: fualty.c:6:16         Value:
;
Type: KeyWord           Location: fualty.c:8:5          Value: double
Type: Identifier                Location: fualty.c:8:12         Value:
d
Type: Punctuator                Location: fualty.c:8:14         Value:
=
Type: NumericConstant           Location: fualty.c:8:16         Value:
0xabcde
Type: Identifier                Location: fualty.c:8:23         Value:
sf
Type: Punctuator                Location: fualty.c:8:25         Value:
.
Type: Identifier                Location: fualty.c:8:26         Value:
acho
Type: Punctuator                Location: fualty.c:8:30         Value:
+
Type: Identifier                Location: fualty.c:8:31         Value:
ef
```

```
Type: Punctuator                Location: fualty.c:8:33         Value:
;
Type: KeyWord           Location: fualty.c:9:5          Value: return
Type: Punctuator                Location: fualty.c:9:12         Value:
-
Type: NumericConstant           Location: fualty.c:9:13         Value:
1
Type: Punctuator                Location: fualty.c:9:14         Value:
;
Type: Punctuator                Location: fualty.c:10:1         Value:
}
File info:
File name:      fualty.c
Total chars:    157
Total lines:    12
KeyWord:        7
Identifier:     11
NumericConstant:        2
NumericConstant(Error Detected):        1
CharacterConstant:      0
StringLiteral:  0
Punctuator:     14
PreprocessingDirective: 1
EndOfFile:      0
Unknown:        0
```

可以看到，这里对于这两个错误的处理分别如下：

对于 int 1caz，由于标识符开头不为数字，因此我们不将它识别为标识符。随后，naiveLex 根据我们处理数字中尽可能吞掉更多的字符的原则，将 ca 认为是 16 进制，同时由于 1ca 前面没有标志 16 进制的前导 0x，因此我们不认为它是正确的数字常量，将其归类为 NumericConstant(Error Detected)，最终将剩余的 z 认为是一个单独的标识符。这符合 naiveLex 认为词法分析错误大多是由于误插入所致的原则。

对于 double d = 0xabcdesf.achp+ef，我们遵循数字常量的原则对其进行处理。首先 naiveLex 识别了 0xabcde，随后遇到不可识别的 s，将其与后面的 f 一并识别为标识符，对于“.”，“achp”，“+”，“ef”，其分别单独识别为一个 punctuaor，identifier，punctuator 和 identifier。也符合 naiveLex 的处理原则。

综上，我们可以认为 naiveLex 很好地完成了错误的识别，同时也按照既定的逻辑对于错误进行了处理和恢复。

五、总结。

事实上，等到整个词法分析器编写完成后，我发现最复杂的地方并不是程序的架构和自动机的设计，而是手工将一个个规则置入程序。这也是老师上课时所

讲的手工编写词法分析器的最大的困难所在。而在编写过程中，通过面向对象的设计方式，我也很好地将输入文件、token、token 的处理以及统计信息进行了分离，使得程序的逻辑变得清晰。此外，通过词法分析器的编写，也让我对于课上所说的词法分析的过程有了更深的了解。

六、参考资料

【 1 】 ISO/IEC 9899:2017 N2176 Programming languages — C https://teaching.csse.uwa.edu.au/units/CITS2002/resources/n2176.pdf