北京理工大學

词法分析器实验

姓 名：徐恒达
学 号：1120151811
班 级：07111504

目录

1 实验目的

利用高级语言设计并实现一个词法分析程序，加深对词法分析原理的理解。并在实验中掌握扫描器的工作原理，以及对输入的程序设计语言源代码进行扫描过程中将其分解为各类不同属性的单词的词法分析方法。

2 实验内容

以 C 语言作为源语言，构建 C 语言的词法分析器，对于给定的测试程序，输出 XML 格式的属性字符流。词法分析器的构建按照 C 语言的词法规则进行。

3 实验过程

本次实验中的 C 语言词法分析器基于现行的 ISO C11 标准（ISO/IEC 9899:2011 Programming languages C）。标准中将 C 语言中的词法元素分为 5 类，分别是关键字（keyword）、标识符（identifier)、常量（constant）、字符串字面量（string literal）和分隔符（punctuator），本次实验也严格按这 5 类进行识别，唯一的不同就是实验中把 ISO 标准中的分隔符进一步分为了限界符（delimiter）和运算符（operator），限界符包含了只有分隔作用没有运算作用的 6 个符号(,),[,],{,},.,->，其余符号均识别位运算符，以方便后续的处理。

本次实验中的词法分析器几乎实现了 ISO C11 标准中词法元素的全集，只有通用字符名（universal character name）和多字节字符（multibyte character）这两类使用频率非常低的词法元素没有进行识别，其余所有的符合 ISO C11 定义的词法元素均能有效识别，并且有健全的出错处理机制，准确划分出非法单词的范围，不会因为某个非法单词而导致其后的识别过程发生混乱。

实验中对 ISO 标准中的 5 类单词重新进行了一些划分，使总的识别任务分为几个任务量均不太大的子任务。将标识符和关键字的识别放在一起，因为关键字本身就是符合标识符定义的特殊单词；因为常量包含的单词范围很大，规则各异，故将常量的识别分为整型常量、浮点型常量和字符常量的识别，但识别后均标注为常量类型；最后是把分隔符分为了限界符和运算符，识别之后仍分别标注限界符和运算符。在词法分析器输出的属性字流中共有 7 种类型，分别是关键

字（keyword）、标识符（identifier）、常量（constant），字符串（string），限界符（delimiter），运算符（operator）和非法（illegal）。词法分析器的设计和实现也针对这 7 种类型展开。

对于每一种要识别的单词，首先根据 ISO C11 标准中给出的词法定义写出其对应的正规式，然后将正规式转化为等价的 DFA，最终将每一类单词的 DFA 和并在一起并用程序实现。本次实验中采用 Java 语言以面向程序的方法实现自动机，即对自动机中的每一个状态定义一个决定其后继状态的状态转换函数。如图1所示。

图 1: 词法分析器实现流程

3.1 标识符和关键字

ISO C11 标准中对于标识符和关键字的词法定义如下。

identifer:
nondigit
identifer nondigit
identifer digit

nondigit: one of

```
_
a b c d e f g h i j k l m n
o p q r s t u v w x y z
A B C D E F G H I J K L M N
O P Q R S T U V W X Y Z
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

keyword: one of

```
auto break case char const continue default do
double else enum extern float for gotoif inline
int long register restrict return short signed
sizeof static struct switch typedef union
```

```
unsigned void volatile while _Alignas _Alignof
_Atomic _Bool _Complex _Generic _Imaginary
_Noreturn _Static_assert _Thread_local
```

标识符的定义简而言之就是由字母、数字和下划线组成且开头不能为数字的单词。转化为等价的正规式如下。

```
identifer = (letter|_) (letter|digit|_)*
letter = [0-9]
digit = [a-zA-Z]
```

关键字完全符合标识符的定义，可以理解为是标识符的特例。在实际识别过程中不需要对关键字设置单独的识别规则，在识别出标识符后，判断如果是关键字集合中的元素就标记为关键字，否则标记为标识符。

识别标识符的 DFA 如图2所示。

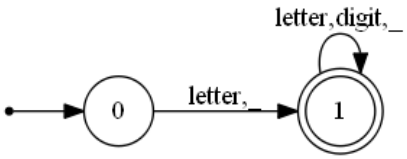

图 2: 识别标识符的 DFA

在实际编程中，程序处在 1 状态后，并不知道此时应该直接识别单词还是继续等待下一个字符，只有当读入一个不是字母数字和下划线的字符后，程序才会知道到当前字符的前一个字符为止是一个完整的标识符。因此在程序中 1 状态并不是一个接受态，而从 1 状态读入一个其他字符后进入的状态才是真正的接受态。要到达此状态后，程序将当前字符之前的单词识别为标识符，并将当前字符放回缓冲区，重新回到 0 状态开始下一轮的识别。实际上在识别过程中大部分单词都像这样需要一个额外的字符才能确定识别。为了突出问题核心，此处以及下文中并没有将实际的接受态画出，但会对每个接受态进行说明。

3.2 整数常量

ISO C11 中的常量分为整数常量、浮点数常量、字符串常量、枚举常量和字符常量。其中枚举常量就是标识符，这里不再单独处理，下面分别对其余 4 种进行识别。首先识别整数常量。

整数常量分为十进制整数常量、八进制整数常量和十六进制整数常量，每一种都可以有整数后缀或者没有。

integer-constant:
decimal-constant
octal-constant
hexconstant
decimal-constant integer-suffix
octal-constant integer-suffix
hex-constant integer-suffix

首先讨论不带整数后缀的常量。十进制整数常量的定义如下所示。

decimal-constant:
nonzero-digit
decimal-constant digit

digit: one of
`0 1 2 3 4 5 6 7 8 9`

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

用文字描述可以为“以 1 到 9 之间的数字开头，其后可以跟任意数字”。用正规式描述如下，其中 * 表示符号可以出现零次或多次，? 表示可以出现零次或一次。

```
decimal-constant= nonzero-digit digit* integer-suffix?
```

八进制整数常量的定义如下所示。

octal-constant:
`0`
octal-constant octal-digit

octal-digit: one of
`0 1 2 3 4 5 6 7`

用文字描述可以为“必须以 0 开头，其后可以跟 0 到 7 之间的任意数字”。正规式描述如下。

```
octal-constant = 0 octal-digit* integer-suffix?
```

十六进制整数常量如下所示。

hex-constant:
hex-prefix hex-digit
hex-constant hex-digit

hex-prefix: one of
`0x 0X`

hex-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

用文字描述可以为“以前缀 0x 或 0X 开头，至少有一位十六进制数字”。正规式描述如下。

```
hex-constant = 0(x|X) hex-digit+ integer-suffix?
```

值得注意的是十进制和八进制常量开头的标志性数字也是其值的一部分，所以后面的数字可以有一个或多个，正规式中表现为数字后面加 *，而十六进制的前缀标识 0x 并不作为其值的一部分，所以要求后面至少有一位十六进制数字，在正规式中表现为数字后面带 +。

不带后缀的三种常量全部讨论完毕了，将这三类常量的正规式转化为等价的 DFA 如图3所示。其中状态 2 表示识别整型常量，状态 3 表示识别八进制常量，状态 13 表示十六进制常量。这三个接受状态再编程时均需要再读入一个额外字符后才能结束识别。

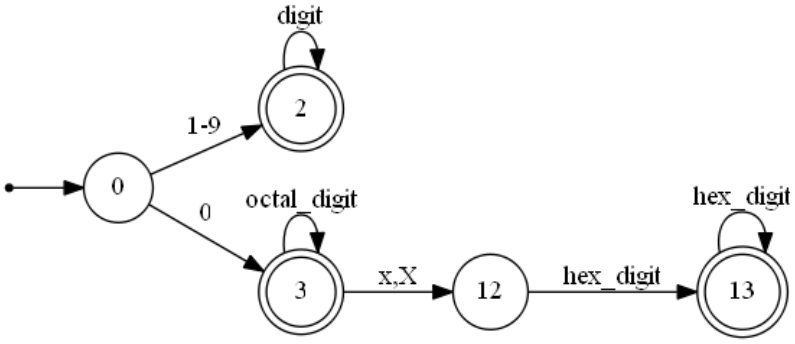

图 3: 识别整数常量的 DFA

最后讨论整数后缀。定义如下。

integer-suffix:
unsigned-suffix
unsigned-suffix long-suffix
unsigned-suffix long-long-suffix
long-suffix
long-suffix unsigned-suffix
long-long-suffix
long-long-suffix unsigned-suffix

unsigned-suffix: one of
`u | U`

long-suffix: one of
`l | L`

long-long-suffix: one of
`ll | LL`

从形式上看略有复杂，概括起来，后缀分为两部分，一部分是 u，另一部分是 l 或 ll，大小写均可。两部分可以只取一种，也可以全有，而且没有顺序限制。可以看出灵活性很大。用正规式描述如下。

```
integer-suffix =
    (u|U) |
    (u|U) (l|L|ll|LL) |
    (l|L|ll|LL) |
    (l|L|ll|LL) (u|U)
```

转化为等价 DFA 如图4所示。图中第一个状态 x 可以是图中的状态 2、3 或 13，即完整的 DFA 中状态 2、3 和 13 之后都要接一个图中的自动机。自动机中每个状态都是接受态，因为后缀可以在任何一个位置停止。每一个接受态都是再读到一个额外字符后完成判断。

3.3 浮点数常量

浮点数常量分为十进制浮点数常量和八进制浮点数常量，没有八进制浮点数常量。每一种浮点数都可以加浮点数后缀或者不加。

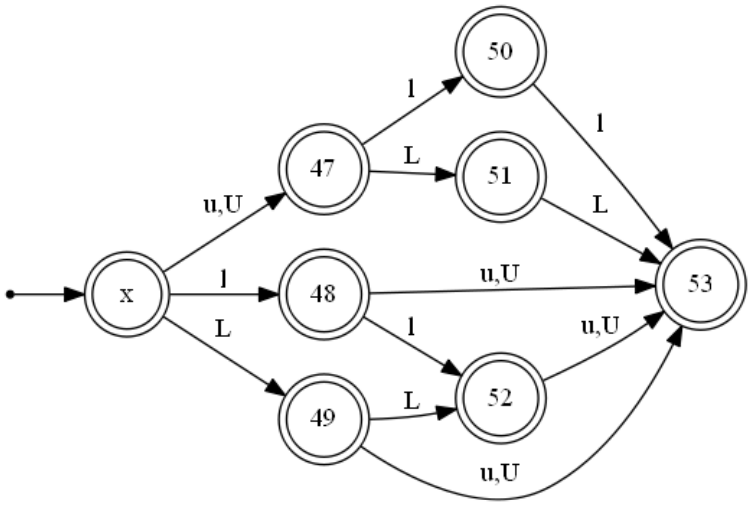

图 4: 识别整数后缀的 DFA

floating-constant:
decimal-float-constant
hex-float-constant
decimal-float-constant float-suffix
hex-float-constant float-suffix

float-suffix: one of
`f l F L`

因为浮点数的后缀只有 f、l、F 和 L 四个单字符，比较简单，故不再单独描述，编程时仿照整数后缀加载每种浮点数的接受态之后即可。下面分别讨论两种浮点数常量。

十进制浮点数的定义如下。

decimal-float-constant:
fractional-constant
fractional-constant exponent-part
digit-sequence exponent-part

fractional-constant:
digit-sequence .
digit-sequence . digit-sequence
. digit-sequence

exponent-part:
exp-flag digit-sequence
exp-flag sign digit-sequence

exp-flag: one of
`e E`

sign: one of
`+ -`

digit-sequence:
digit
digit-sequence digit

值得注意的是十进制浮点数的数字序列不再区分前缀，也就是无论一个数以 0 开头还是以 1 到 9 中的数字开头，只要有小数点或者指数标志 E，均识别为十进制浮点数。也可以理解为将八进制浮点数也作为的十进制浮点数。

将词法转化为正规式如下。

```
decimal-float-constant =
   fractional-constant ((e|E) (+|-)? digit+)? (f|F|l|L
      )? |
   digit+ (e|E) (+|-)? digit+ (f|F|l|L)?
fractional-constant -> digit+ . (digit+)? | . digit+
```

转化为等价的 DFA 如图5所示。其中每一个接受状态均需要读入一个额外字符后才能完成识别。

十六进制浮点数常量的定义如下所示。

hex-floating-constant:
hex-prefix hex-fractional-constant binary-exponent-part
hex-prefix hex-digit-sequence binary-exponent-part

hex-fractional-constant:
hex-digit-sequence . hex-digit-sequence
hex-digit-sequence .
. hex-digit-sequence

binary-exponent-part:

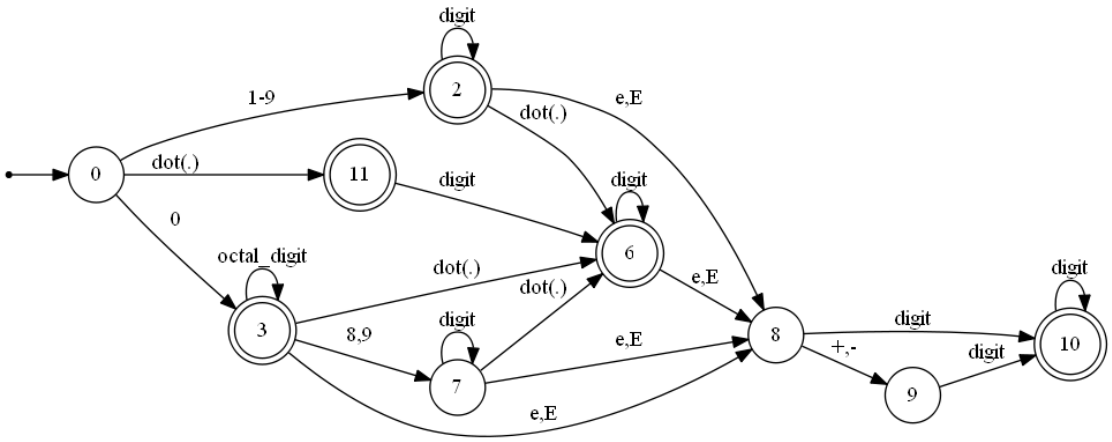

图 5: 识别十进制浮点数的 DFA

bin-exp-flag digit-sequence
bin-exp-flag sign digit-sequence

hex-digit-sequence:
hex-digit
hex-digit-sequence hex-digit

hex-prefix: one of
`0x 0X`

从定义中可以看出，十六进制浮点数必须有指数标志 P，且指数部分的数字为十进制数。

转化为正规式如下所示。

```
hex-float-constant =
   0(x|X) hex-fractional-constant (p|P) (+|-)? digit+
      (f|F|l|L)? |
   0(x|X) hex-digit+ (p|P) (+|-)? digit+ (f|F|l|L)?
hex-fractional-constant ->
   hex-digit+ . (hex-digit+)? |
   . hex-digit+
```

将其转化为等价的 DFA 如图6所示。

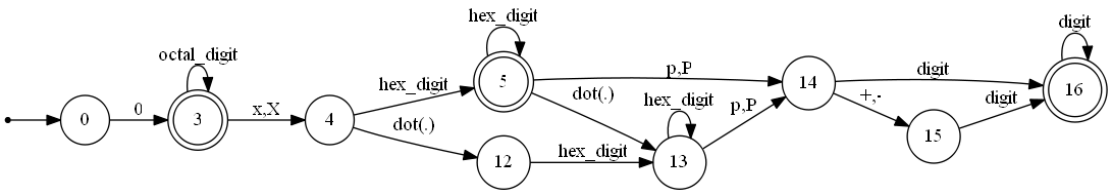

图 6: 识别十六进制浮点数的 DFA

3.4 字符常量

字符常量的词法定义如下所示。ISO C11 标准中也定义多字节字符（multibyte character），即两个单引号之间允许有多个字符。本次实验中没有定义多字节字符。

character-constant:
’ c-char ’
char-prefix ’ c-char ’

char-prefix: one of
`L u U`

c-char:
any character except ’, new-line
escape-sequence

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hex-escape-sequence

simple-escape-sequence: one of
`\' \" \? \\ \a \b \f \n \r \t \v`

octal-escape-sequence:
octal-digit
octal-digit octal-digit
octal-digit octal-digit octal-digit

hex-escape-sequence:

x hex-digit
hex-escape-sequence hex-digit

字符常量整体上的定义非常简单，两个单引号之间包含一个字符。但从定义中可以看出，只是转义序列的处理比较麻烦。为了不因为细节而把核心问题淹没，先将只包含简单转义序列的词法转化为正规式，如下所示。

```
character-constant = (L|u|U| ) ' normal-char | \ esc-
   char '
esc-char = '|"|?|a|b|f|n|r|t|v
```

等价的 DFA 如图7所示。其中接受态 20 就是真正的接受态，程序到达状态 20 后就可以直接识别为字符常量。

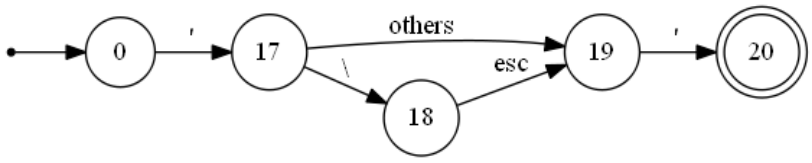

图 7: 识别基本字符常量的 DFA

考虑八进制转义序列和十六进制转义序列的后的正规式如下所示。

```
character-constant = (L|u|U| ) ' normal-char | \ esc-
   char | \ octal-digit (octal-digit octal-digit?)? |
   \x hex-digit+ '
esc-char = '|"|?|a|b|f|n|r|t|v
```

转化为等价的 DFA 如图8所示。

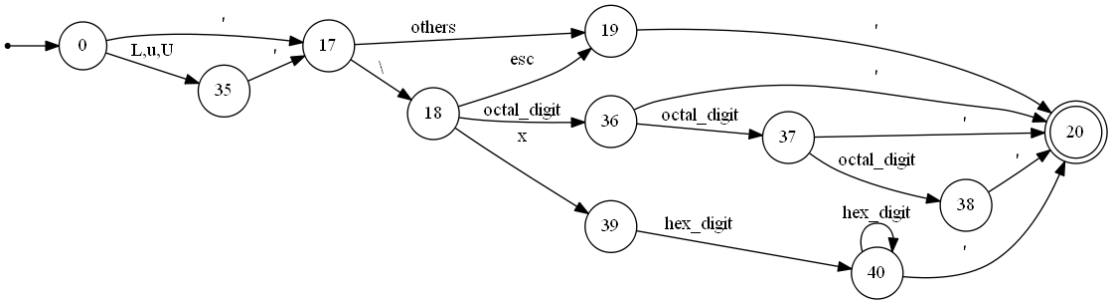

图 8: 识别完整字符常量的 DFA

3.5 字符串

字符串的词法定义如下所示。

string:
" s-char-sequence "
encoding-prefix " s-char-sequence "

encoding-prefix: one of
`u8 u U L`

s-char-sequence:
ϵ
s-char-sequence s-char

s-char:
any character except ",
, new-line
escape-sequence

同字符常量，只考虑简单转义序列的正规式如下。

```
string = u8|u|U|L| " (normal-char | \ esc-char)* "
```

等价的 DFA 如图9所示。接受态 23 是真正的接受态，不需要读入额外的字符。

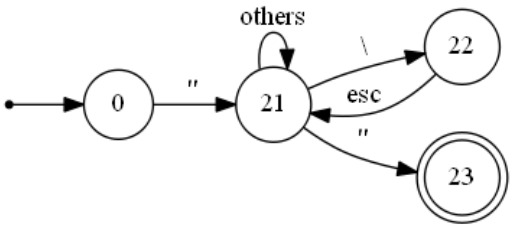

图 9: 识别基本字符串的 DFA

考虑全部转义序列的正规式如下。

```
string = u8|u|U|L| " (normal-char | \ esc-char | \
   octal-digit (octal-digit octal-digit?)? | \x hex-
   digit+)* "
```

将其转化为等价的 DFA 如图10所示。因为实际的转换关系繁多，图中还是略去了一部分边，42、43、44 和 46 状态，每个状态都省略了遇到普通字符回到 21 状态的边、遇到反斜线到 22 号状态的边和遇到非法字符到出错处理状态的边。

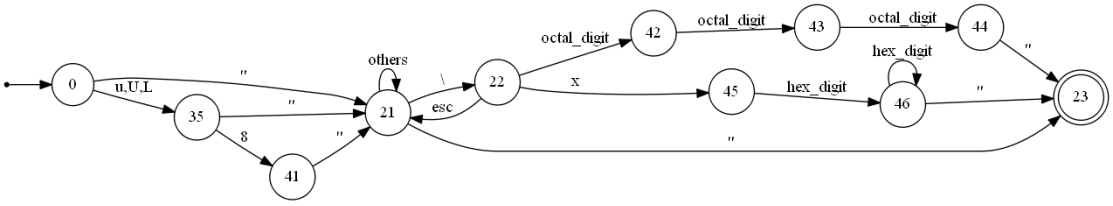

图 10: 识别字符串的 DFA

3.6 分隔符

将 ISO C11 中定义的分隔符分为限界符（delimiter）和运算符（operator）。如下所示。

delimiter: one of

```
( ) [ ] { } ; . ->
```

operator: one of

```
, ? :
+ += ++ - -= -- & &= && | |= ||
< <= << <<= > >= >> >>=
*= / /= % %=^= ! != = ==
```

限界符比较简单，基本上遇到限界符直接识别即可。而运算符比较复杂。根据第一个字符的区别将其分为以下几类。

`( ) [ ] { } ; . ->`: 限界符。

`, ? :` : 这四个运算符不是任何运算符的前缀，遇到后直接识别即可。

`-`: 遇到-后可能是-> 限界符，也有可能是运算符。这是唯一一处限界符和运算符有交集的地方。

`+ - & |`: 这四个运算符的模式相同，可以单独作为运算符，也可以跟 =，也可以自身重复一次。

`< >`: 可以单独作为运算符，可以跟 =，可以重复自己，可以重复自己再跟等号

=。

*** / % ! =**: 最普通的运算符。可以单独，也可以后跟等号 =。

直接给出 DFA 如图11所示。

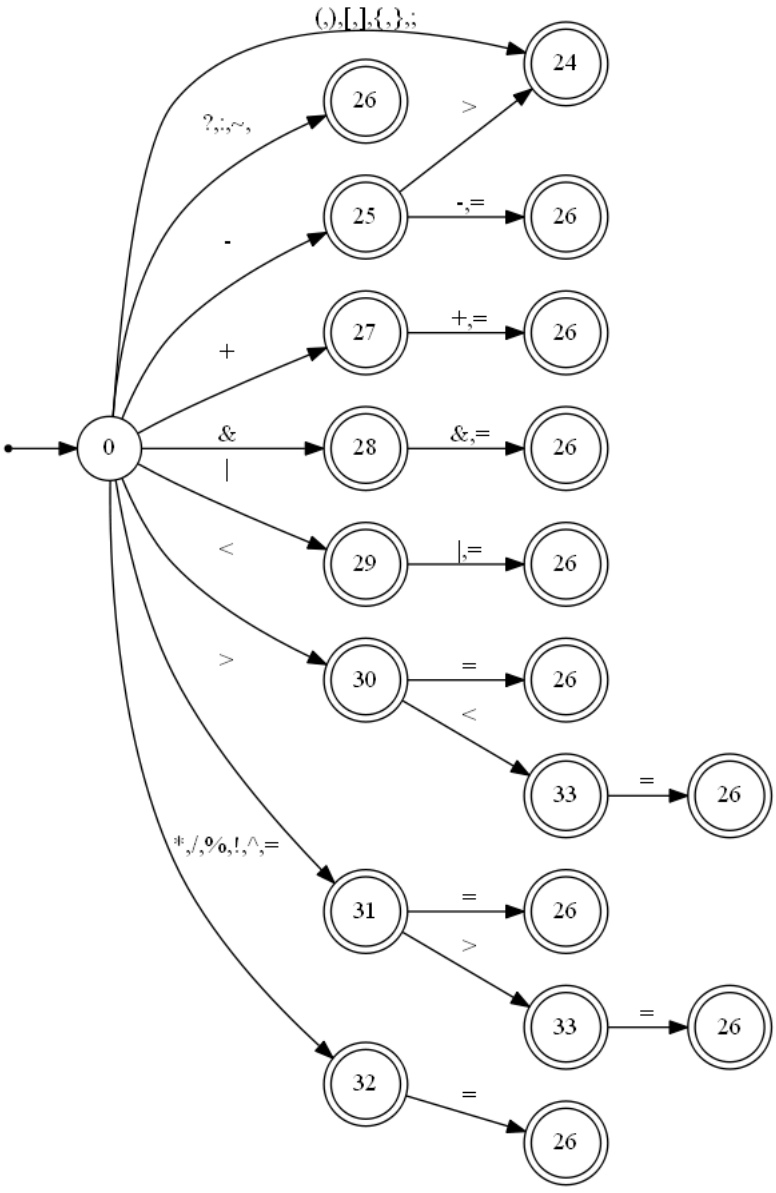

图 11: 识别分隔符的 DFA

3.7 出错处理

出错处理也是词法分析过程中非常重要的一个部分，不可能编译的代码都是完全正确的，而一个好的出错处理机制可以准确地划分出不符合词法的部分，以保证代码的其他部分仍然能够正常识别，而出错处理设计不好时可能以为一处错误导致其他合法部分无法识别或将另外一些非法部分识别为合法。

实际上几乎自动机的每个状态节点要与出错处理机制相关联，因为几乎没有一个状态在接受任意字符后都有一个合法的后继状态定义，而对一个状态而言，接收到的所有合法字符之外的符号都是此状态的非法字符，都将转换到出错处理状态。

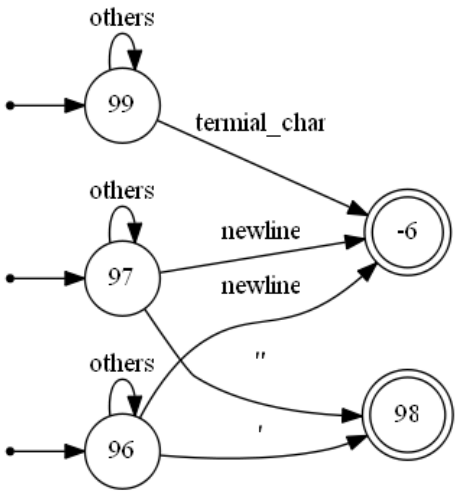

图 12: 负责出错处理的 DFA

实验中与出错处理有关的状态定义了 5 个，如图12所示。一般情况下，遇到非法字符后将进入 99 状态，这个状态并不是接受状态，也就是遇到非法字符时并不会立即结束，而是继续读完整个非法单词，当遇到能够结束一个单词的符号，比如运算符或分隔符时，进入-6 状态，识别非法单词结束，同时将这个终结符放回缓冲区，回到 0 状态开始新一轮的识别。

当在识别字符串时，也就是处理过双引号" 后，遇到非法字符时进入 97 状态，不断接受字符知道字符串结束。非法字符串在两种情况下会结束，遇到回车换行符时进入-6 状态，识别非法字符串，将回车换行符放回缓冲区，重新开始新一轮识别；遇到双引号" 时，进入 98 状态，立即结束识别，双引号作为非法字符串的最后一个字符被识别，不再送回缓冲区。类似的，在识别字符常量时遇到

非法字符将进入 96 状态，遇到单引号或回车换行符时结束识别。

3.8 最终 DFA

将前文所有设计的 DFA 合并到一起称为最终的 DFA，如图13所示。介于绘图中节点布局的因素，部分状态和边没有画出，包括整数和浮点数常量的后缀识别，字符常量和字符串的八进制转义序列和十六进制转义序列的识别，每个节点的出错处理部分和程序中真正的接受状态。但在实际程序中全部实现了。每个状态的编号已经事先调整好了，上文中分部分介绍中的编号就是最终合并调整后的编号，没有编号冲突。

3.9 程序实现

程序使用 Java 语言实现，使用以程序为中心的方法，对自动机中的每个状态都定义一个转换函数，以确定当前状态接收不同字符后的后继状态。主程序的伪代码如下。其中输入缓冲区为 buffer，buffer.next() 表示从缓冲区读取一个字符，buffer.push(c) 表示将字符 c 放回缓冲区。接受状态集合为 Accept，其中有一些接受状态是读取了一个额外字符后才识别单词的，处理这些状态时需要向缓冲区中放回一个字符，这些状态包含在回退接受态集合 Back 中。

Algorithm 1 Scanner

Input:
输入缓冲区 buffer
接受状态集 $Accept$
回退接受状态集 $Back$
Output:
属性字流 $(type_i, word_i)$

```
1:  状态 state ← 0
2:  当前单词 word ← {}
3:  c ← buffer.next()
4:  while c ≠ EOF do
5:      word += c
6:      f ← 状态 state 的转换函数
7:      state ← f(c)
8:      if state ∈ Accept then
9:          if state ∈ Back then
10:             buffer.push(c)
11:             word -= c
12:         end if
13:         type ← attr(state)
14:         output (type_i, word_i)
15:         state ← 0
16:         word ← {}
17:     end if
18:     c ← buffer.next()
19: end while
```

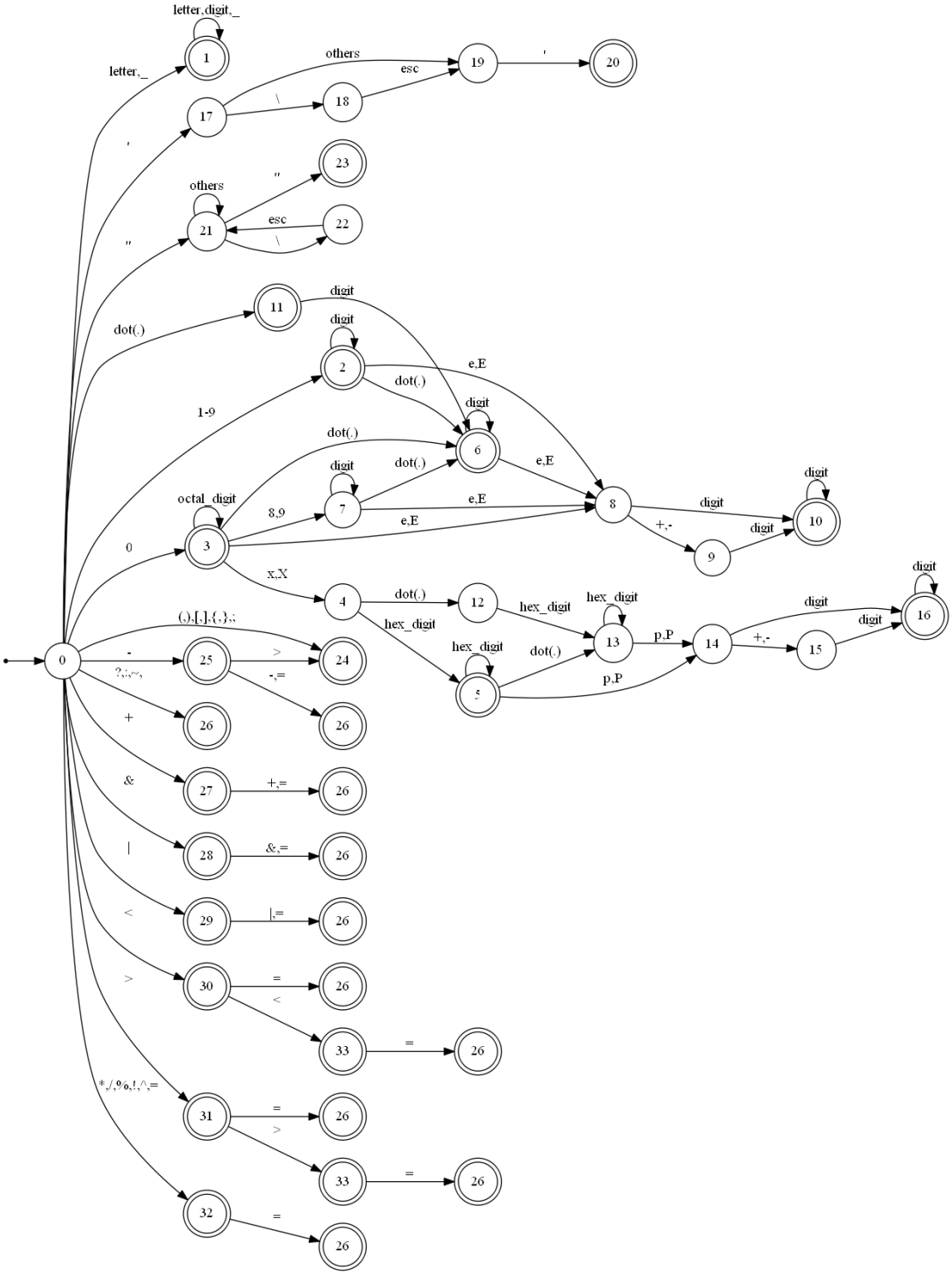
letter,digit,_
letter,_
others
esc
'
\
"
dot(.)
1-9
digit
e,E
octal_digit
8,9
0
x,X
hex_digit
p,P
+,-
(,),[,],{,},;
?,:,~,
+
&
|
<
>
*,/,%,!,^,=
-,=
+,=
&,=
|,=
=

图 13: 合并后的 DFA

核心的 Java 实现代码如下所示。每一个状态的转换函数名为“state+ 状态序号”，程序中的 getMethod() 和 invoke() 函数是 Java 中的反射技术，根据当前的状态值动态地获取对应的状态转换函数来执行。

```
public void run(String inputFile, String outputFile) {
    // Build buffer "reader" from "inputFile"
    int state = 0;
    StringBuilder word = new StringBuilder();
    Class<?> cl = this.getClass();
    for (int r = reader.read(); r != -1; r = reader.
        read()) {
        char c = (char)r;
        word.append(c);
        Method action = cl.getMethod("state" + state,
            char.class);
        state = (int) action.invoke(null, c);
        if (isAcceptState(state)) {
            if (isBackState(state)) {
                reader.unread(c);
                word.setLength(word.length() - 1);
            }
            dealAcceptState(state, word.toString());
            word.setLength(0);
            state = 0;
        }
    }
    // Write to "outpuFile"
}
```

4 实验结果

src 文件夹中又三个 Java 源程序文件，分别实现了三个 Java 类，BITMiniCC、MiniCCPreProcesser 和 Scanner。

其中 BITMinicCC 是主类，是编译器的框架，里面我将原有的框架做了一些修改，但对外的特性完全没有变，修改的具体内容在报告的最后一部分给予了说明。

MiniCCPreProcesser 是对框架附带的预处理 jar 包中的 class 文件反编译后的结果，并作了稍许修改。因为时间的原因，我没有自己实现预处理器。但是框架自带的预处理器功能不完整，而且 bug 颇多，我先临时借来使用，以保证程序流程完整执行。但也可以直接拿预处理过的文件或者不带预处理内容的文件进行词法分析测试。

Scanner 是词法分析器，所有的词法分析功能在 Scanner 类中实现。Scanner 中定义了 main 方法，可以单独运行，单独运行时需要在命令行中给出要处理的文件名。同时 Scanner 也定义了 run 方法，供 BITMiniCC 框架调用。单独使用 Scanner 时，在生成单词属性流 XML 文件的同时，在控制台中也会以简洁的方式打印识别出的每个单词的行号，内容和属性，便于查看。当使用框架调用时，不会在控制台输出属性流内容。

向词法分析器输入预处理后的 C 语言代码如下所示。

```
int main()
{
    printf("Hello\n");
    int a = 1 + 012 * 0xf5;
    double b=++a*0x/08.2E-3;
    char *p = "\12\xff"
    return 0;
}
```

输出的属性字流如下所示。程序最终将其转换为规定的 XML 文件保存。

```
int       keyword                 (         delimiter
main      identifier              "Hello\n" string
(         delimiter               )         delimiter
)         delimiter               ;         delimiter
{         delimiter               int       keyword
printf    identifier              a         identifier
```

=	operator	0x	illegal
1	constant	/	operator
+	operator	08.2E-3	constant
012	constant	;	delimiter
*	operator	char	keyword
0xf5	constant	*	operator
;	delimiter	p	identifier
double	keyword	=	operator
b	identifier	"\12\xff"	string
=	operator	return	keyword
++	operator	0	constant
a	identifier	;	delimiter
*	operator	}	delimiter

国际 C 语言混乱代码大赛（IOCCC）2013 年的获奖作品 misaka 的代码如图14所示。

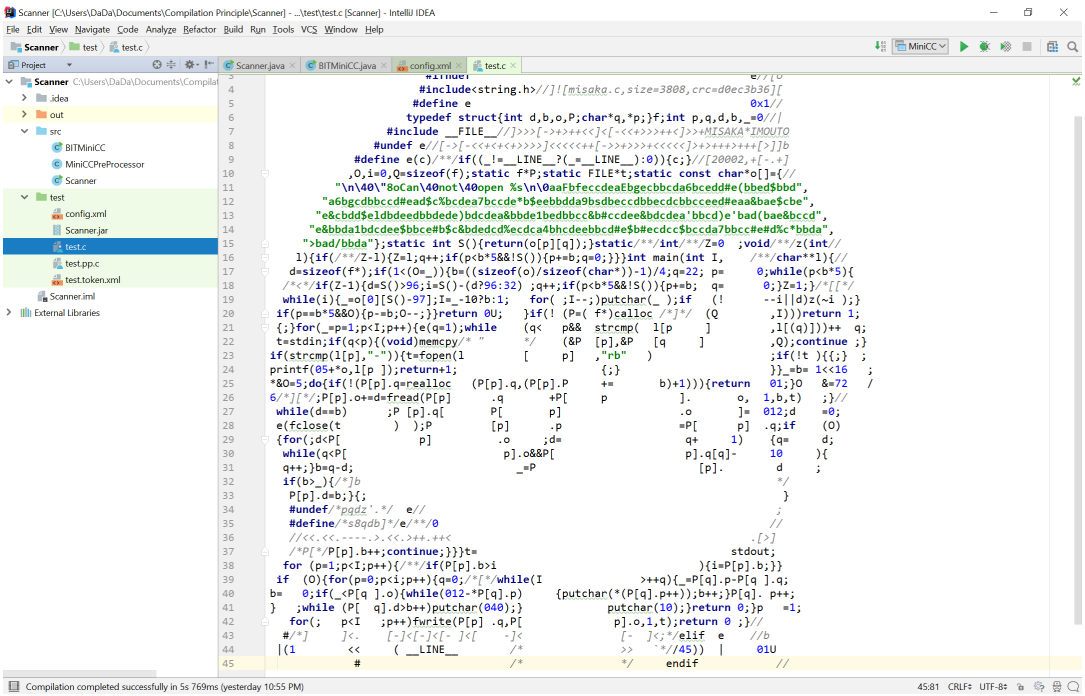

图 14: misaka.c 源文件

将其预处理后输入词法分析其，得到的包含属性字流的 XML 文件如图15所示。

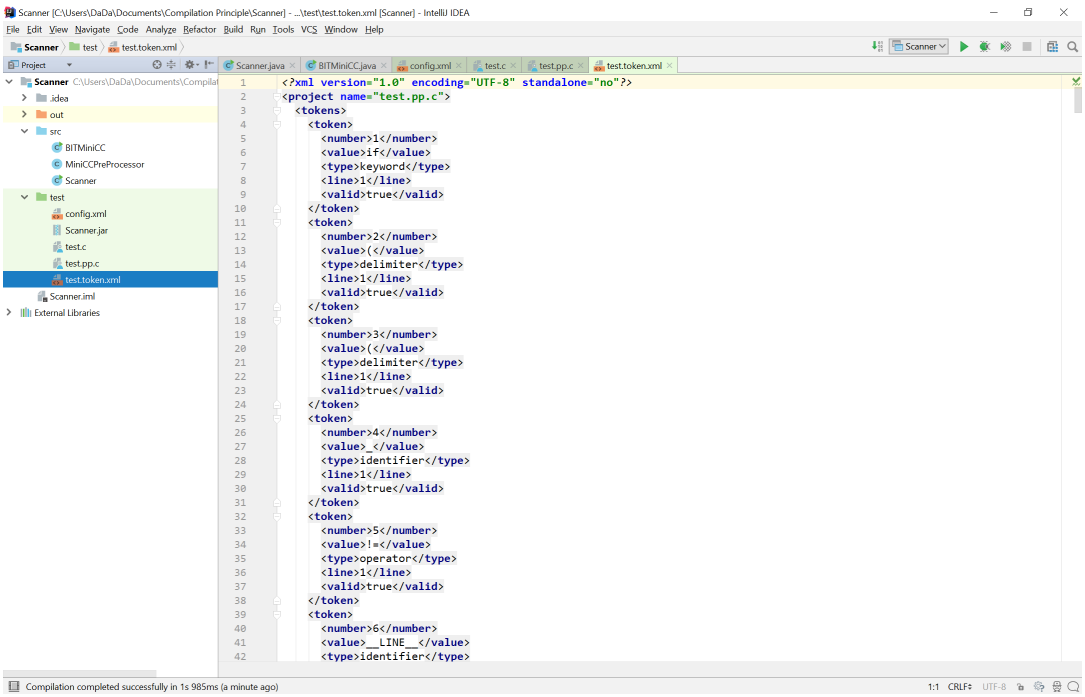

图 15: 词法分析后的 misaka.token.xml 文件

5 心得体会

通过本次实验，我基本掌握了词法分析的基本知识，在有限自动机的设计和简化上有了很多的练习，并且在代码实现的过程中，进一步提高了对 Java 语言的理解和熟练度。通过使用老师提供给我们的编译器框架，我非常清晰地看到了一个编译器的组成，并且更加熟悉了一个代码在从我们平日里点击“编译”之后，整个编译流程所需要进行的各个步骤。

在实验中感触最深的就是，在设计有限自动机的时候，由于之前设计大型 DFA、NFA 比较少，导致在遇到较为复杂的情况时，比如遇到浮点数识别或者转义字符识别时，需要反复修改 NFA，逐渐学习掌握消除无关状态和等价状态的相关方法的用法。

次外也切实体会到了 Java 语言面向对象设计的强大，可以为开发大型工程时提供极大的便利。

6 框架修改

原有的 BITMiniCC 框架的代码有些过于冗长，其实现的功能非常简单，只是根据配置文件 config.xml 中的内容依次调用编译各阶段的程序，将上一个程序的输出文件作为当前程序的输入文件，仅此而已。而原有的框架把这种只需要循环就可以完成的工作复制粘贴了好几份，顿时使得本来简单的代码变得过于冗长、难于阅读和难于修改。我将原代码中功能相同的代码合并到了一个循环中，并优化了部分部分代码的写法，同时完全兼容原有的 config.xml 中的配置，希望可以对老师今后的教学活动有所帮助，也希望能帮助其他同学更好更快地理解和使用框架。

此外，我对老师提供的配置文件 config.xml 的格式也有一个小建议。文件的根元素是一个 config 标签，核心信息是对应 8 个阶段的 phase 标签，但是根标签与核心标签之间却套了两层额外的标签 phases 和 phase，如下所示。这两层额外的框架标签给阅读和解析文件的内容增加了负担，而且外层的 phase 标签与核心的 phase 标签重名，更是给解析过程带来了混乱，也不利于文件框架的清晰定义。所以，我建议老师去掉中间的两层 phases 和 phase 标签。

```
<?xml version="1.0" encoding="UTF-8"?>
<config name="config.xml">
  <phases>
    <phase>
      <phase skip="false" type="java" path="" name="pp"
          />
      <phase skip="false" type="java" path="" name="
         scanning"/>
      ...
```

再者，原有框架中将编译过程每一步的输入文件输出文件后缀名在代码中固定了，同学们必须严格按照规定的后缀名给文件命名，并且如果不阅读源代码往往不容易明确得到这些文件名的定义，这也给同学们理解和使用框架带来了一定的阻力，并且破坏了设计框架的初衷，方便同学，为同学们搭建完整编译器提供帮助，而不是再给同学们出一道阅读代码的题目。所以，我建议在每一个核心 phase 标签上再加一个属性，用来让同学们指定自己的文件后缀名，当然，老师可以提供默认的建议后缀名。

最后一个建议，整个框架中程序间信息的交换完全基于 XML 格式文件，鉴于 XML 格式文件有繁多的标签属性，信息密度并不大，且可阅读性还是稍微差一些，而且同学们要完成整个编译器不得不学习 XML 文件的读取和创建的技术，这无形中又给同学们增加了额外的并不在教学目标中的负担。老师可以考虑一下近来非常流行，也非常轻量级且更易于阅读和自动化处理的 json 格式的文件，功能于 XML 完全相同，但更加方便。比如配置文件可以是这样。

```
{
  "phases": [
    {
      "skip": false,
      "type": "java",
      "path": "",
      "name": "pp",
      "suffix": ".pp.c"
    },
    {
      "skip": false,
      "type": "java",
      "path": "",
      "name": "scanning",
      "suffix": ".token.json"
    },
    ...
  ]
}
```

词法分析器输出的单词属性流可以是这样。

```
{
  "tokens": [
    {
      "number": 1,
      "value": "int",
      "type": "keyword",
```

```
      "line": 1,
      "valid": true
    },
    {
      "number": 2,
      "value": "main",
      "type": "identifier",
      "line": 1,
      "valid": true
    },
    ...
  ]
}
```

把多年来的 XML 文件替换为 json 文件可能会涉及很多地方的修改，并且由于思维惯性的原因，突然改变的难度很大，我也是仅此向老师说出我的想法，希望我们的 BITMiniCC 框架能越来越好，成为北理工编译原理课程代表性的一个特色。