编译原理

词法分析实验实验报告

班级：07111603
学号：1120161730
姓名：武上博
2019 年 4 月 18 日

目录

1 实验目的

1. 熟悉 C 语言的词法规则，了解编译器词法分析器的主要功能
2. 掌握典型词法分析器构造的相关技术和方法，设计并实现 C 语言词法分析器
3. 掌握编译器从前端到后端各个模块的工作原理，词法分析模块与其他模块之间的交互过程

2 实验内容

根据 C 语言的词法规则，设计并识别 C 语言所有单词类的词法分析器的确定有限状态自动机，并使用 Java、C/C++、Python 其中的任意一种语言，采用程序中心法或者数据中心法设计并实现词法分析器。词法分析器的输入为 C 语言源程序，输出为属性字流。

3 实验的具体过程步骤

3.1 程序实现的大致思路

为了和接下来语法分析模块相配合，本次实现的词法分析器接受 C 语言源程序作为输入，利用 XML 作为格式进行输出分析的词法内容。同时，为了和 BIT-MiniCC 进行更好的整合，本次实验我决定使用 Python 作为主语言进行各个模块的实现。

经过分析，我觉得本次实验中词法分析器是如下图 1 的大致构造：

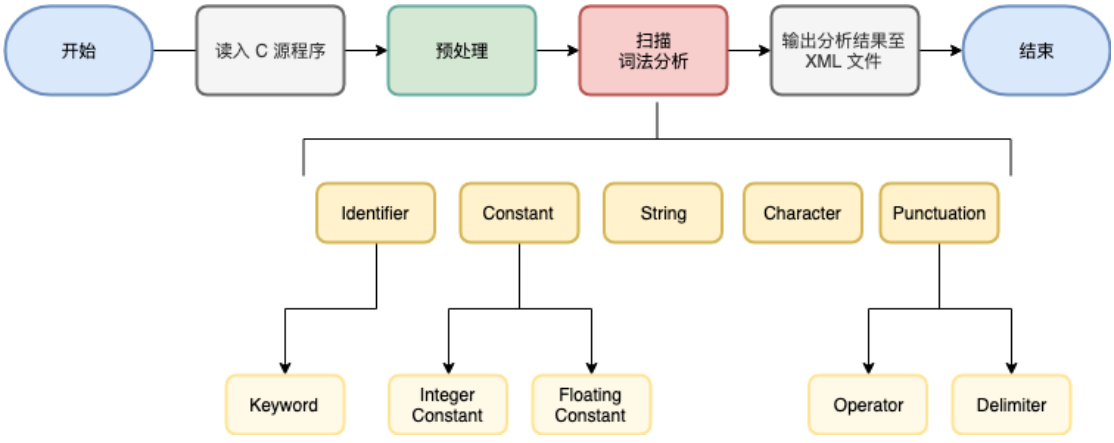


图 1: 词法分析器的大致流程

也就是说，我们本次需要实现的模块有：

1. 文件读入
2. 程序预处理模块（去除每行前部分空格、注释等）

3. 进行词法分析，依次识别：
    - 标识符 Identifier
    - 常量 Constant
        - 整数型常量 Integer Constant
        - 点型常量 Floating Constant
    - 字符 Char
    - 字符串 String
    - 算符 Punctuation
        - 运算符 Operator
        - 界限符 Delimiter
4. 输出 XML 文件

3.2 具体模块的实现

接下来，我们分别对各个模块相应的具体实现方法进行介绍。

3.2.1 程序输入和预处理

本次实验中的输入是一个 C 语言程序的源文件。我们从命令行读入需要处理的文件路径，处理文件内容。

```
def main():
  # Print usage if arguments are not legal
  if len(sys.argv) < 2:
    print('[Usage] ./scan.py <C source file path>')
    sys.exit(0)

  # Read file from file path taken from command line arguments
  filePath = sys.argv[1]
  with open(filePath, 'r', encoding='utf-8') as f:
    content = f.readlines()
```

在读文件时，我使用了 `readlines()` 函数为了逐行读入文件。我们得到 `content`，也就是代码的基本内容。之后，我们对读入的内容进行预处理。

```
# 主函数内的内容，预处理代码内容
code = preProcess(content)
```

```
# 预处理函数
def preProcess(content):
  code = ''
  # Trim leading white space 去掉每行最前面的空白
  for line in content:
    if line != '\n':
      code = code + line.lstrip()
    else:
      code = code + line
  return code
```

我首先定义代码变量 code，之后按行处理代码内容，对于每一行代码，如果代码不是空行，那么我就将这一行的代码和前面定义的 code 相连接，之后我们只需要处理 code 缓冲区内的代码内容即可。

接下来，我们利用自动机对输入字符串进行匹配来判断其输入类型。首先，我定义了下面一个指针（即当前读入字符位置）和五个识别类型：

```
# 指针查找位置
index = 0

# Token 属性
codeNum = 1
codeType = ''
codeLine = 1
codeValue = ''
codeValid = 0
```

我维护指针 index 用来遍历输入代码串，利用 codeNum、codeType、codeLine、codeValue、codeValid 来分别标识：当前识别的 Token 数量、当前识别 Token 的种类、当前读到代码行数、当前识别 Token 的内容以及当前识别 Token 是否合法。

之后，我构造 scanner() 来对输入串进行扫描识别处理：

```
def scanner(code):
  # 当前扫描代码位置
  global index
  # 当前识别符数
  global codeNum
  # 当前代码行
```

```
  global codeLine

  # 识别到词语的类别
  global codeType
  codeType = ''
  # 识别到的词语
  global codeValue
  codeValue = ''
  # 当前识别字符
  character = code[index]
  index = index + 1

  # Ignore white space
  while character == ' ':
    character = code[index]
    index = index + 1
  ...
```

在主函数 `main()` 中，我通过这样的方式调用扫描器：

```
# Start scanning!
global codeNum
while index <= len(code) - 1:
  scanner(code)
```

接下来，我构建了五个自动机，分别对标识符、常量、字符、字符串和算符进行了识别。

3.2.2 标识符 Identifier 的判断

标识符 Identifier 是由字母、数字或下划线“_”组成的，具体的定义大致是这样的：

$$\begin{aligned} identifier \rightarrow\ & identifier - nondigit \\ & \mid identifier\ identifier - nondigit \\ & \mid identifier\ digit \end{aligned} \tag{1}$$

$$identifier - nondigit \rightarrow\ nondigit \mid universal - character \tag{2}$$

$$nondigit \rightarrow \_ \mid a...z \mid A...Z \tag{3}$$

$$digit \rightarrow 0...9 \tag{4}$$

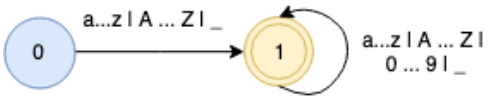


图 2: 识别标识符的状态机

为了识别标识符，我确定如图 2 的状态机。

之后，我们就可以实现对标识符的识别：

```
# Identifier!
if character.isalpha() or character == '_':
  while character.isalpha() or character.isdigit() or character == '_':
    codeValue = codeValue + character
    character = code[index]
    index = index + 1
  codeType = 'identifier'
  index = index - 1
```

在识别了标识符之后，我们可以直接继续判断这个标识符是不是 C 语言中的关键词（Keyword）之一。我们本次实验需要判断的是 C 语言的子集，需要进行识别的关键词有这些：

表 1: C 语言关键词表

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	inline	int	long	register
restrict	return	short	signed	sizeof
static	struct	switch	typedef	union
unsigned	void	volatile	while	

于是，我们维护一个关键词列表 `cKeywords`，之后通过字符串匹配的方式识别标识符是否为关键词：

```
# Keyword!
for keyword in cKeywords:
  if codeValue == keyword:
    codeType = 'keyword'
    break
```

这样我们就可以成功的识别 Token 中的标识符和关键词。

3.2.3 常量（整形常量 Integer Constant 和浮点型常量 Floating Constant）的判断

在 C 语言的语法中，常量 Constant 有整形和浮点型两种需要进行识别。整形常量 Integer Constant 的文法可以这样描述：

$$\begin{aligned} integer-constant \rightarrow\ & decimal-constant\ integer-suffix \\ & \mid octal-constant\ integer-suffix \\ & \mid hexadecimal-constant\ integer-suffix \end{aligned} \tag{5}$$

$$decimal-constant \rightarrow 1...9 \mid decimal-constant\ digit \tag{6}$$

$$octal-constant \rightarrow 0 \mid octal-constant\ octal-digit \tag{7}$$

$$\begin{aligned} hexadecimal-constant \rightarrow\ & hexadecimal-prefix\ hexadecimal-digit \\ & \mid hexadecimal-constant\ hexadecimal-digit \end{aligned} \tag{8}$$

$$hexadecimal-prefix \rightarrow 0x \mid 0X \tag{9}$$

$$\begin{aligned} integer-suffix \rightarrow\ & unsigned\ long \mid unsigned\ long-long \\ & \mid long\ unsigned \mid long-long\ unsigned \end{aligned} \tag{10}$$

浮点型常量 Floating Constant 的文法可以这样描述：

$$floating-constant \rightarrow\ decimal-floating-constant \mid hexadecimal-floating-constant \tag{11}$$

$$\begin{aligned} decimal-floating-constant \rightarrow\ & fractional-constant\ exp\ floating-suffix \\ & \mid digital-seq\ exp\ floating-suffix \end{aligned} \tag{12}$$

$$\begin{aligned} hexadecimal-floating-constant \rightarrow\ & hexadecimal-prefix \\ & hexadecimal-frac-constant\ binary-exp\ floating-suffix \\ & \mid hexadecimal-prefix\ hexadecimal-digit-seq \\ & binary-exp\ floating-suffix \end{aligned} \tag{13}$$

$$exp \rightarrow\ e\ digit-seq \mid E\ digit-seq \tag{14}$$

$$sign \rightarrow\ + \mid - \tag{15}$$

$$digit-seq \rightarrow\ digit \mid digit-seq\ digit \tag{16}$$

$$\begin{aligned} hexadecimal-frac-constant \rightarrow\ & hexadecimal-digit-seq\ . \\ & hexadecimal-digit-seq \mid hexadecimal-digit-seq\ . \end{aligned} \tag{17}$$

$$binary-exp \rightarrow\ p\ digit-seq \mid P\ digit-seq \tag{18}$$

$$\begin{aligned} hexadecimal-digit-seq \rightarrow\ & hexadecimal-digit-seq \mid \\ & hexadecimal-digit-seq\ hexadecimal-digit \end{aligned} \tag{19}$$

$$floating-suffix \rightarrow\ f \mid F \mid l \mid L \tag{20}$$

其中我们需要特别注意的是：

- 整数常量中有十进制 Decimal、八进制 Octal 和十六进制 Hexadecimal 三种常量需要考虑，其中以 `0` 开头的是八进制数字，以 `0x` 或 `0X` 开头的是十六进制数字，其他我们都直接认为是十进制数字
- 浮点型常量中需要考虑科学计数法，比如 `1.5e-4` 就是一个合法的浮点型常量
- 整数常量中的后缀字符有 u、U 表示无符号整形 unsigned 和 l、L 表示长整型 long 或 long long，也就是说比如 `512ull` 这样的无符号长整型就是合法的十进制整型常量
- 浮点型常量中后缀字符有 f、F、l、L，其中 f、F 表示 float 类型的浮点型常量，没有 f、F 后缀的浮点型常量我们认为是 double 类型的；l、L 表示长浮点型常量，即比如 `1.5F`、`4.9L` 这样的浮点数是合法的

综合整形常量和浮点型常量的文法表示和注意事项，我们可以构造如下图 3 的自动机来识别整形和浮点型常量：

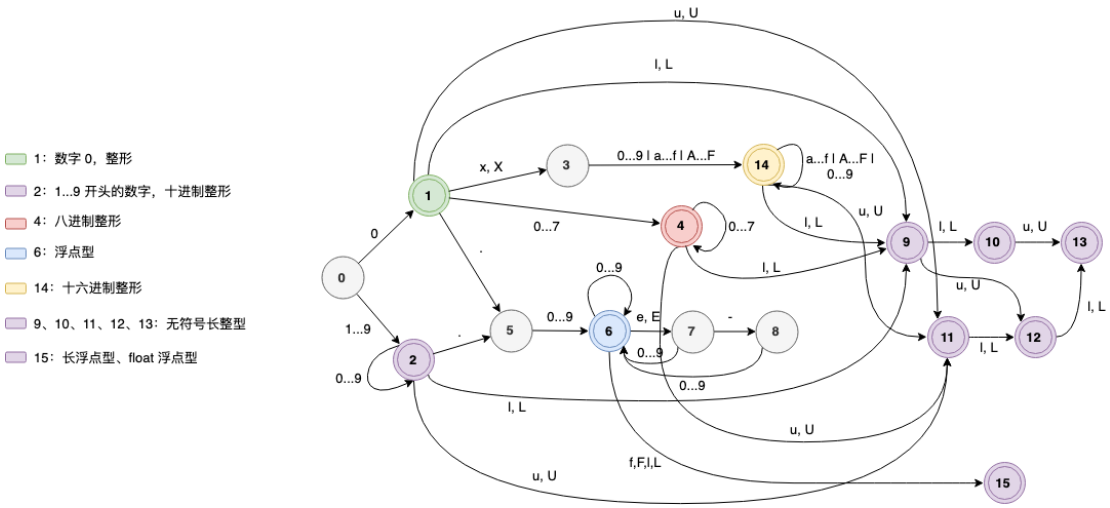


图 3: 整形和浮点型常量识别自动机

之后，我们就可以按照状态机的进行代码实现了。

```
# Integer and float constants
elif character.isdigit():
  global constantState
  while character.isdigit() or character in '-.xXeEaAbBcCdDfFuUlL':
    codeValue = codeValue + character
    # 进入自动机，进行串匹配
    if constantState == 0:
      if character == '0':
        constantState = 1
    ...
    elif constantState == 13 or constantState == 15:
      if character:
```

```
            constantState = -1
        character = code[index]
        index = index + 1

    index = index - 1
    # 接受整形常量
    if constantState in (1, 2, 4, 9, 10, 11, 12, 13, 14):
        codeType = 'integer constant'
        constantState = 0
    # 接受浮点型常量
    elif constantState == 6 or constantState == 15:
        codeType = 'floating constant'
        constantState = 0
    # 被拒绝或未识别的常量
    else:
        codeType = 'illegal constant'
        constantState = 0
```

这样，我们就基本实现了对整形和浮点型常量的识别。

3.2.4 字符 Character、字符串 String 的判断

字符和字符串我们需要分别进行判断。我在这一步骤的判断是基于界限符单引号 ' 和双引号 " 来判断 Token 属于字符还是字符串。

我们首先利用文法描述字符常量：

$$char - constant \rightarrow \ 'c - char - seq' \tag{21}$$

$$c - char - seq \rightarrow \ c - char \mid c - char - seq \ c - char \tag{22}$$

$$c - char \rightarrow \ all \ symbols \ other \ than \ ', \ \backslash \ and \ \backslash n \mid esc - seq \tag{23}$$

$$esc - seq \rightarrow \ \backslash' \mid \backslash" \mid \backslash? \mid \backslash\backslash \mid \backslash a \mid \backslash b \mid \backslash f \mid \backslash n \mid \backslash r \mid \backslash t \mid \backslash v \tag{24}$$

字符串常量同样可以用文法表示如下：

$$string - literal \rightarrow \ "s - char - seq" \tag{25}$$

$$s - char - seq \rightarrow \ s - char \mid s - char - seq \ s - char \tag{26}$$

$$s - char \rightarrow \ all \ symbols \ other \ than \ ', \ \backslash \ and \ \backslash n \mid esc - seq \tag{27}$$

$$esc - seq \rightarrow \ \backslash' \mid \backslash" \mid \backslash? \mid \backslash\backslash \mid \backslash a \mid \backslash b \mid \backslash f \mid \backslash n \mid \backslash r \mid \backslash t \mid \backslash v \tag{28}$$

根据上面的文法描述，我们可以画出识别字符和字符串的自动机如下图 4 所示：

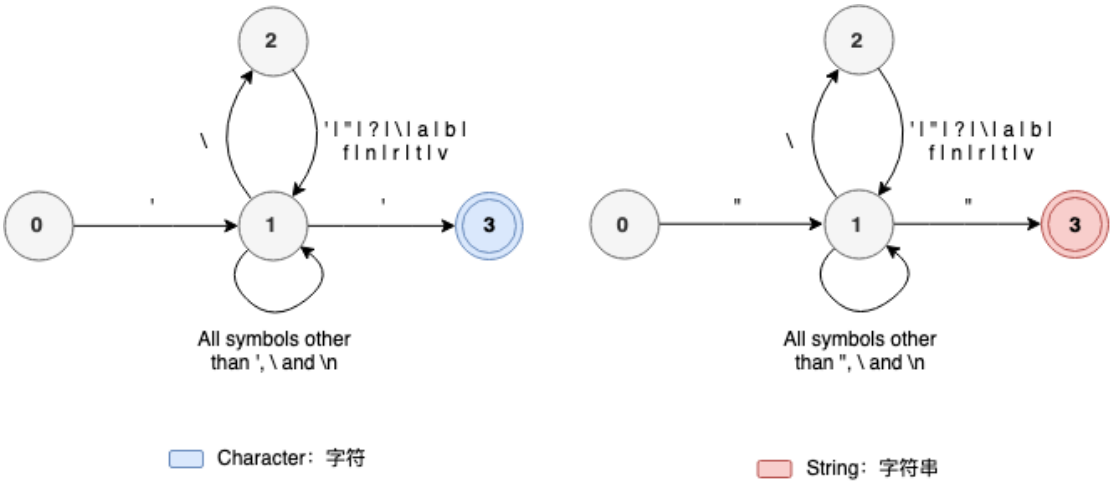


图 4: 识别字符 Character 和字符串 String Literal 的自动机

可以看到，我们在识别字符和字符串的时候，需要特别识别转义字符，因此我们首先构造一个转义字符列表：

```
# 转义字符
cEscSequence = ['\'', '"', '?', '\\', 'a', 'b', 'f', 'n', 'r', 't', 'v']
```

之后，我们就可以根据上面自动机构建识别字符和字符串的代码。

```
# String!
elif character == '"':
  global stringState
  while index < len(code):
    codeValue = codeValue + character
    if stringState == 0:
      if character == '"':
        stringState = 1
    elif stringState == 1:
      if character == '\\':
        stringState = 3
      elif character == '"':
        stringState = 2
        break
    elif stringState == 2:
      break
    elif stringState == 3:
      if character in cEscSequence:
```

```
        stringState = 1
    character = code[index]
    index = index + 1
  if stringState == 2:
    codeType = 'string'
    stringState = 0
  else:
    print('Illegal string.')
    stringState = 0

# Char!
elif character == '\'':
  global charState
  while index < len(code):
    codeValue = codeValue + character
    if charState == 0:
      ...
    character = code[index]
    index = index + 1
  if charState == 2:
    codeType = 'character'
    charState = 0
  else:
    codeType = 'illegal char'
    charState = 0
```

3.2.5 算符（包括运算符 Operator 和界限符 Delimiter）的判断

在 C 语言中，能够被识别的算符有：

! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~

这些算符通过组合可以成为单目或多目运算符，一些单独的算符也属于界限符。通过查阅资料，我找到了 C 语言定义了以下界限符：

表 2: C 语言定义的界限符 Delimiter

[	]	(	)	{	}	'	"	,	;	\

C 语言中合法的运算符有这些：

表 3: C 语言定义的运算符 Operator

.	->	++	–	&	*	+	-	~	!
/	%	«	»	<	>	<=	>=	==	!=
^	|	&&	||	?	:	...	=	*=	/=
%=	+=	-=	«=	»=	&=	^=	|=	#	##
<:	:>	<%	%>	%:					

因此，我们可以根据算符能否作为单目运算符来将其如下区分：

表 4: 能构成多目运算符的算符分类

运算符首字符	对应的多目运算符				
+	++	+=			
-	-=	-=			
<	«	<=	«=	<:	<%
>	»	>=	»=		
=	==				
!	!=				
&	&&	& =			
|	||	|=			
*	*=				
/	/=				
%	%=	%>	%:		
^	^=				
:	:>				
#	##				

那么剩下的算符就是无法直接构成多目运算符的算符了。

具体实现中，我们首先定义运算符列表、可作为多目运算符的算符列表和界限符列表这三个列表：

```
# 运算符
cOperator = ['+', '-', '&', '*', '~', '!', '/',
             '^', '%', '=', '.', ':', '?', '#', '<', '>', '|', '`']
# 可作为二元运算符首字符的算符
cBinaryOp = ['+', '-', '>', '<', '=', '!',
             '&', '|', '*', '/', '%', '^', '#', ':', '.']
# 界限符
cDelimiter = ['[', ']', '(', ')', '{', '}', '\'', '"', ',', ';', '\\']
```

对于界限符，由于在词法分析这一步骤我们尚不需要对括号匹配等内容进行识别，因此我们只需要匹配输入 Token 是否为界限符即可。具体实现如下：

```
  # Delimiters
  elif character in cDelimiter:
    codeValue = codeValue + character
    codeType = 'delimiter'
```

对于操作符，由于我们需要考虑单目和多目运算符进行匹配工作，因此这里我们根据上面表 4 进行实际的代码实现，构造自动机进行 Token 识别。这里涉及到的自动机非常复杂，也远远超过了我们本次实验报告的内容，这里我就不具体进行介绍了。下面是大致的代码实现：

```
# Operators
elif character in cOperator:
  global operatorState
  while character in cOperator:
    codeValue = codeValue + character
    if operatorState == 0:
      if not character in cBinaryOp:
        operatorState = 20
        break
      else:
        ...

    character = code[index]
    index = index + 1
  if operatorState >= 2 and operatorState <= 18:
    index = index - 1
    codeType = 'Unary operator'
    operatorState = 0
  elif operatorState == 20:
    codeType = 'Unary operator'
    operatorState = 0
  elif operatorState == 1:
    codeType = 'Multicast operator'
    operatorState = 0
  else:
    index = index - 1
    codeType = 'Illegal operator'
    operatorState = 0
```

我们到这里基本实现了所有 Token 类型的识别。

3.2.6 输出 XML 格式的 Token 识别结果至文件

最后，我们需要将识别成功的 Token 添加至 XML 树中，并将其输出到指定文件内。为了实现 XML 树的构建，我使用了 `ElementTree` 的 Python 内置 XML 操作库进行处理，建立根节点 `project`、Token 节点 `tokens` 和每个 Token 的属性节点。属性节点大致构建如下：

```
<token>
  <numbers>1</numbers>
  <value>int</value>
  <keyword>keyword</keyword>
  <line>1</line>
  <true>true</true>
</token>
```

为了将项目名作为根节点 `project` 的 `name` 属性，以及确立 XML 输出文件名称，我们首先处理输入文件的名称：

```
# C source file name
fileName = os.path.basename(filePath)
# XML output file name
xmlFileName = os.path.splitext(fileName)[0] + '.token.xml'
```

之后，我们创建 XML 树：

```
# Create XML tree
xmlTree = ElementTree.Element('project', {
    'name': fileName
})
xmlTokens = ElementTree.SubElement(xmlTree, 'tokens')
```

接下来，我们将每次识别的 Token 属性依次赋给每棵子树的节点：

```
while index <= len(code) - 1:
  ...
  if codeType != '':
```

```
    xmlToken = ElementTree.SubElement(xmlTokens, 'token')

    xmlNumber = ElementTree.SubElement(xmlToken, 'numbers')
    xmlValue = ElementTree.SubElement(xmlToken, 'value')
    xmlType = ElementTree.SubElement(xmlToken, 'keyword')
    xmlLine = ElementTree.SubElement(xmlToken, 'line')
    xmlValid = ElementTree.SubElement(xmlToken, 'true')

    xmlNumber.text = str(codeNum)
    xmlValue.text = str(codeValue)
    xmlType.text = codeType.lower()
    xmlLine.text = str(codeLine)
    if not 'illegal' in codeType.lower():
      xmlValid.text = 'true'
    else:
      xmlValid.text = 'false'
    ...
    codeNum = codeNum + 1
```

这样，我们就成功的将 Token 的识别结果输出至树 `xmlTree`，构建了 XML 树。最后，我们将 XML 树的内容格式化（Prettify）并赋予 `utf-8` 的文件编码格式，就可以输出至文件了。

```
# Turn XML tree to string
xmlString = ElementTree.tostring(xmlTree)
# Set XML indent and encoding (This returns a byte for the file to read)
xml = minidom.parseString(xmlString).toprettyxml(indent='  ', encoding='utf-8')
# Write XML to file
with open(xmlFileName, 'wb') as f:
  f.write(xml)
print('Written XML token processing results to file:', xmlFileName)
```

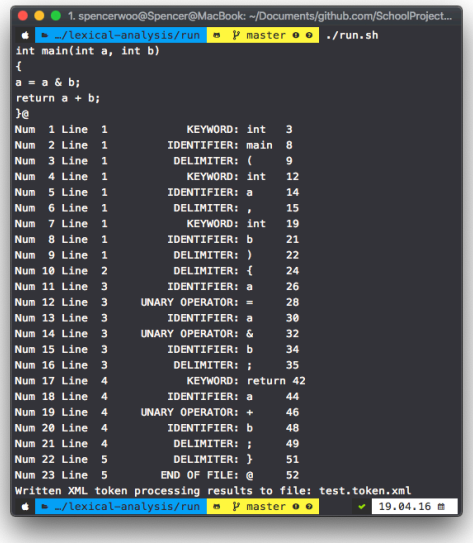
4 实验结果

4.1 Token 的识别结果

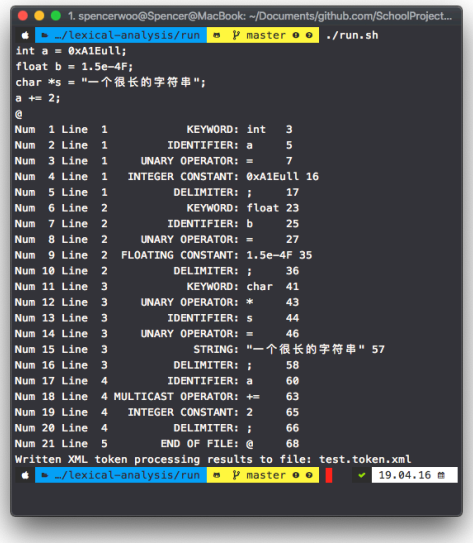
我们在 Token 识别的过程中，将识别到的 Token 属性依次输出，以下面这段 C 语言代码段为例子：

```
int main(int a, int b)
{
  a = a & b;
  return a + b;
}
```

我们可以得到如下图 5a 的输出结果：



(a) 识别过程输出的 Token 属性



(b) 对于不常用 Token 的识别处理

图 5: 词法分析器对 Token 的识别结果

为了更加准确的测试我本次实验实现的词法分析器，我准备了一个如下的 C 语言代码段：

```
int a = 0xA1Eull;
float b = 1.5e-4F;
char *s = "一个很长的字符串";
a += 2;
```

我使用我本次实现的词法分析器进行识别，得到了如上图 5b 的识别结果，可以看到对于复杂的输入串，我们的词法分析器也是准确的。

4.2 XML 文件的输出结果

接下来，我们将识别号的 Token 各个属性进行 XML 格式化，并输出至文件中。我成功生成了 XML 文件，并将之和 `BIT-MiniCC` 词法分析步骤的 XML 文件进行比对，得到了

如下图 6 的 XML 识别结果。

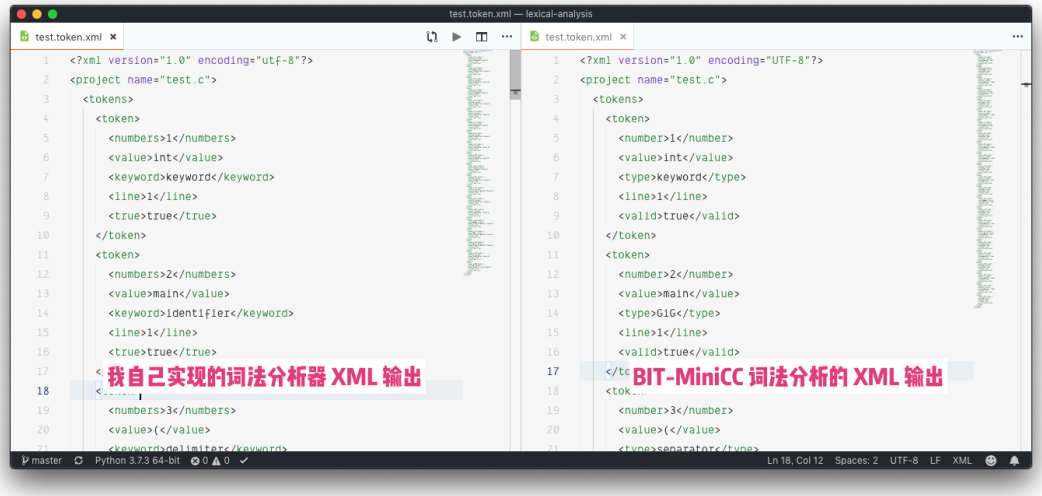


图 6: XML 文件的输出结果和 BIT-MiniCC 输出 XML 对比

可以看到，我们本次实验输出的 XML 结果和 `BIT-MiniCC` 的输出结果大致相同，可以作为下一次语法分析的输入，实验结果符合预期。

5 实验心得体会

通过本次实验，我重新认识了 C 语言编写的程序在编译过程中词法分析的具体方法。我不仅更加了解了标识符、常量、字符和算符的具体文法描述，还实际的利用了自动机对这些 Token 进行匹配，大致实现了一个简单的词法识别器，能够对一段 C 语言代码进行分析，得到每个识别 Token 的位置、属性和是否合法等性质。

在本次实验的词法分析中，虽然每种 Token 的文法描述都非常复杂，但是我发现，只要认真仔细的写好自动机的具体识别过程，利用 Python 来实现这个自动机还是相对简单。

同时，我也在本次实验中领略到各种不常用的 C 语言合法的语法，比如 `0777`、`0xAFull` 等八进制、十六进制整数，比如 `1.5e-4`、`0.5F` 等各种形式的浮点数，比如 `<%`、`|=`、`...` 等不常见的算符。这让我重新了解了 C 语言的语法表示，也让我对自动机理论有了新的认识。