

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 16 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：

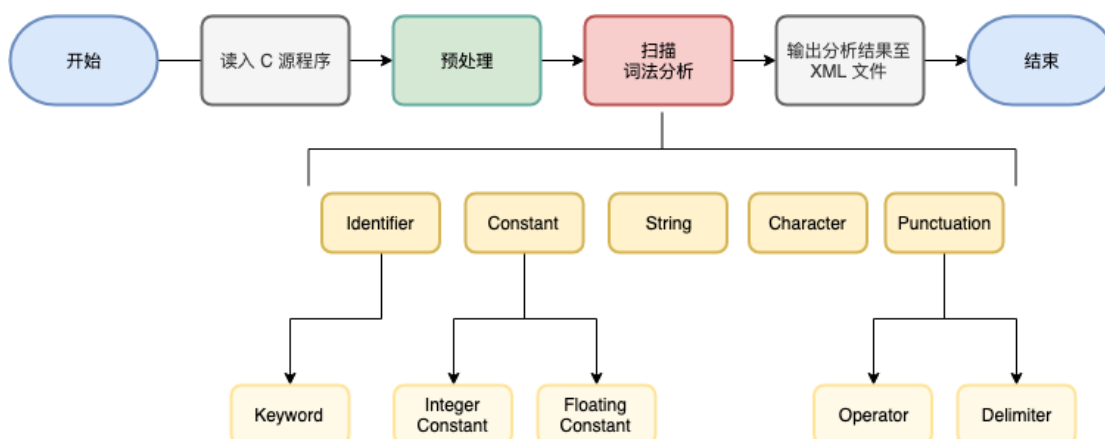


图 1: 词法分析器的大致流程

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。

```
1 def main():
2     # Print usage if arguments are not legal
3     if len(sys.argv) < 2:
4         print('[Usage] ./scan.py <C source file path>')
5         sys.exit(0)
6
7     # Read file from file path taken from command line arguments
8     filePath = sys.argv[1]
9     with open(filePath, 'r', encoding='utf-8') as f:
10        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。

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

```
3  # 预处理函数
4  def preProcess(content):
5      code = ''
6      # Trim leading white space 去掉每行最前面的空白
7      for line in content:
8          if line != '\n':
9              code = code + line.lstrip()
10         else:
11             code = code + line
12     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：

```
1  # 指针查找位置
2  index = 0
3
4  # Token 属性
5  codeNum = 1
6  codeType = ''
7  codeLine = 1
8  codeValue = ''
9  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：

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

```

7  global codeLine
8
9  # 识别到词语的类别
10 global codeType
11 codeType = ''
12 # 识别到的词语
13 global codeValue
14 codeValue = ''
15 # 当前识别字符
16 character = code[index]
17 index = index + 1
18
19 # Ignore white space
20 while character == ' ':
21     character = code[index]
22     index = index + 1
23 ...

```

在主函数 `main()` 中，我通过这样的方式调用扫描器：

```

1  # Start scanning!
2  global codeNum
3  while index <= len(code) - 1:
4      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：

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

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

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

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

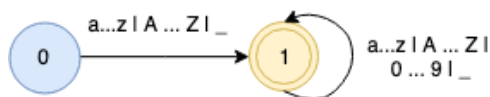


图 2: 识别标识符的状态机

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：

```

1  # Identifier!
2  if character.isalpha() or character == '_':
3      while character.isalpha() or character.isdigit() or character == '_':
4          codeValue = codeValue + character
5          character = code[index]
6          index = index + 1
7      codeType = 'identifier'
8      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：

```

1  # Keyword!
2  for keyword in cKeywords:
3      if codeValue == keyword:
4          codeType = 'keyword'
5          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：

$$\begin{aligned} integer - constant \rightarrow & decimal - constant \mid integer - suffix \\ & \mid octal - constant \mid integer - suffix \end{aligned} \quad (5)$$

$$\mid hexadecimal - constant \mid integer - suffix$$

$$decimal - constant \rightarrow 1...9 \mid decimal - constant digit \quad (6)$$

$$octal - constant \rightarrow 0 \mid octal - constant octal - digit \quad (7)$$

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

$$hexadecimal - prefix \rightarrow 0x \mid 0X \quad (9)$$

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

浮点型常量 Floating Constant 的文法可以这样描述：

$$floating - constant \rightarrow decimal - floating - constant \mid hexadecimal - floating - constant \quad (11)$$

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

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

$$\begin{aligned} & binary - exp floating - suffix \\ exp \rightarrow & e digit - seq \mid E digit - seq \end{aligned} \quad (14)$$

$$sign \rightarrow + \mid - \quad (15)$$

$$digit - seq \rightarrow digit \mid digit - seq digit \quad (16)$$

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

$$binary - exp \rightarrow p digit - seq \mid P digit - seq \quad (18)$$

$$\begin{aligned} hexadecimal - digit - seq \rightarrow & hexadecimal - digit - seq \mid \\ hexadecimal - digit - seq & hexadecimal - digit \end{aligned} \quad (19)$$

$$floating - suffix \rightarrow f \mid F \mid l \mid L \quad (20)$$

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，我们可以构造如下的自动机来识别整形和浮点型常量。

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：

$$char - constant \rightarrow 'c - char - seq' \quad (21)$$

$$c - char - seq \rightarrow c - char \mid c - char - seq \ c - char \quad (22)$$

$$c - char \rightarrow all\ symbols\ other\ than\ ',\ \ and\ \ n \mid esc - seq \quad (23)$$

$$esc - seq \rightarrow \backslash' \mid \backslash" \mid \backslash? \mid \backslash\backslash \mid \backslash a \mid \backslash b \mid \backslash f \mid \backslash n \mid \backslash r \mid \backslash t \mid \backslash v \quad (24)$$

字符串常量同样可以用文法表示如下：

$$string - literal \rightarrow "s - char - seq" \quad (25)$$

$$s - char - seq \rightarrow s - char \mid s - char - seq \ s - char \quad (26)$$

$$s - char \rightarrow all\ symbols\ other\ than\ ',\ \ and\ \ n \mid esc - seq \quad (27)$$

$$esc - seq \rightarrow \backslash' \mid \backslash" \mid \backslash? \mid \backslash\backslash \mid \backslash a \mid \backslash b \mid \backslash f \mid \backslash n \mid \backslash r \mid \backslash t \mid \backslash v \quad (28)$$

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：

4 实验结果

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。