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 15 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：

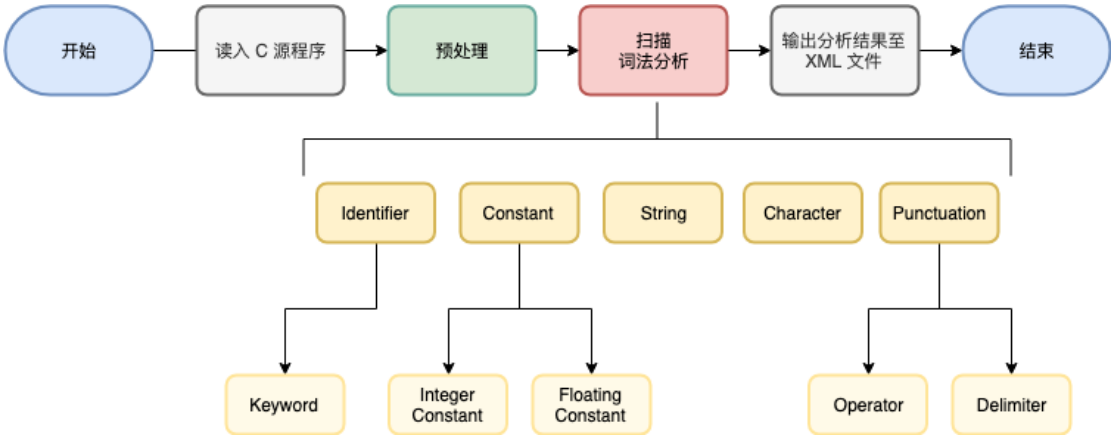

图 1: 词法分析器的大致流程

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。

```
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：

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

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

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

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

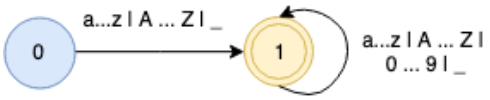

图 2: 识别标识符的状态机

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：

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

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

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

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

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

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

3.2.4 字符 Character、字符串 String 的判断

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

4 实验结果

5 实验心得体会