第二讲　词法分析

2025-09

1. 词法分析概述

1.1 词法分析的任务

词法分析的核心任务是从左到右读入源程序的字符流，识别出一个个的单词。所识别的每一个**单词**，是下一个有意义的词法元素，如标识符或整数。在识别出下一单词，同时也验证了其词法正确性之后，词法分析程序就会产生一个单词记录，传递给后续阶段使用。

例如，在处理一个 Pascal 程序文本的某个时刻，假定词法分析程序开始扫描下面的一段源程序字符流：

```
position := initial + rate * 60 ;
```

词法分析程序从左至右逐个扫描字符，根据词法规则，所识别出的下一个单词是‘position’，它是一个标识符。这样，词法分析程序就会产生一个对应于这个标识符的单词符号，它会被传递给编译的后续阶段。然后，词法分析程序继续扫描后面的字符，逐个识别出单词 ‘:=’， ‘initial’， ‘+’， ‘rate’， ‘*’， ‘60’ ，‘;’，等等。

词法分析程序所产生的**单词记录**通常由两部分信息组成：一个是**单词符号**（token），对应某个特定意义的**词法单元**，如标识符、整常数等；另一部分是单词的**属性值**（attribute）。

程序设计语言中有各种类别的单词，常见的如：

- 保留字，也称关键字，如 C 语言中的 if、while、struct、int、typedef 等。
- 标识符，用来表示各种名字，如常量名、变量名、函数/过程名、类名等。
- 各种类型的常数，如整数 25，浮点数 3.1415，串常数“ABC”等。
- 运算符，如 +，*，<= 等。
- 界符，如逗点，分号，括号等。

在程序设计语言中，某些单词类别的所有单词都是作为相同意义的词法单元出现在的语法描述（如文法）中的，因而拥有相同的单词符号。比如，所有的标识符单词都拥有同一个代表标识符的单词符号；所有的整数单词都拥有同一个代表标识符的单词符号。对于这样的单词，至少需要附加一个属性值来表明它是相应单词集合中的哪一个。例如，对于单词‘position’，其单词符号对应于词法单元 ‘标识符’，而其属性值会附加一个字符串常量“position”，代表该标识符的名字；对于单词 ‘60’ ，其单词符号对应于词法单位 ‘整常数’，而属性值则会附加数值 60。

像保留字、运算符、界符等这些类别的每一个单词，通常是作为独立意义的词法单元出现在的语法描述中，因而拥有各自的单词符号。对于这些单词，就没有必要附加任何属性值。

在实践中，通常是将单词记录设计为二元组的形式：<单词符号，单词属性值>，或

<token，attribute>。单词符号是语法分析阶段需要的信息，单词的属性值则是其它阶段需要的信息。单词符号用于区别不同的词法单元，实践中通常将各个单词符号对应到不同的整数。对于单词属性值，或者直接给出这个值，或者是给出一个指向该单词所有属性值的指针。后一种情形是由于有多个附加属性，或者是需要数据格式对齐的时候采用。

例如，在处理上面的 Pascal 程序文本后，词法分析程序所识别的单词序列见表 1，其中列举了每个单词的类别和值（单词值省略掉了某些附加信息，例如在源程序中的位置等）。

词法单元（与单词符号对应）	单词属性值
标识符	position
赋值算符 (:=)	
标识符	initial
加算符 (+)	
标识符	rate
乘算符 (*)	
整数常量	60
分号 (;)	

表 1　识别单词（得到单词类别/单词值等信息）

又如，对于以下 MiniDecaf 程序片断：

```
int main() {
    int a = 2021;
    return a;
}
```

词法分析程序所识别的单词序列参见表 2。

除了识别有词法意义的单词，验证词法正确性，以及产生单词记录这些基本任务外，词法分析程序通常也会完成几个小任务。具体来说，词法分析程序通常会与符号表处理程序进行紧密与频繁的协作，会在需要时在符号表中存储和查找标识符。此外，它还会执行一些简单的任务以简化源程序文本，如大小写转换，或注释和空白等多余段行的去除等。

1.2 词法分析在编译程序中的组织

关于词法分析在编译程序中的组织，较常用的方式是用语法分析程序调用词法分析程序。当从语法分析程序接到下一个单词记录的请求时，词法分析器从左到右读入源程序的字符流，以识别下一单词 —— 即下一个有意义的词法单元。在识别出下一单词同时验证其词法正确性之后，词法分析器产生一个单词记录，并发送至语法分析程序以满足其请求。若在单词识别过程中发现词法错误，则返回出错信息。参见图 1。

设计中还要考虑的一点就是对于单词记录中的属性值的处理，比如可能需要将标识符的名字、类型等信息后续会存放于符号表中。

词法单元（与单词符号对应）	单词属性值
保留字 int	
标识符	main
分隔符 (	
分隔符)	
分隔符 {	
保留字 int	
标识符	a
操作符 =	
整数常量	2021
分隔符 ；	
保留字 return	
标识符	a
分隔符 ；	
分隔符 }	

表 2　单词识别结果（单词类别/单词值等信息）

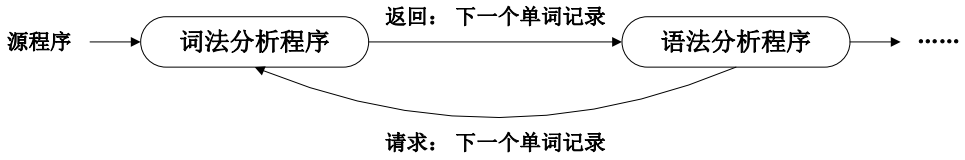

图 1　语法分析调用词法分析

词法分析程序也可以作为单独的一遍，但这种组织方式并不常用。如图 2 所示，词法分析程序从左到右读入源程序的字符流，识别每一个单词后产生相应的单词记录，单词记录形式的源程序作为语法分析程序的输入。

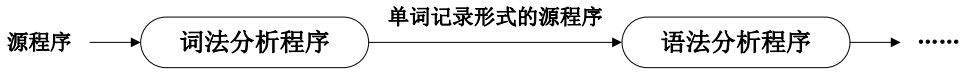

图 2　单独作为一遍的词法分析

在识别出一个标识符后，可以在词法分析阶段或是在语法分析阶段加入符号表。当然，建立和维护符号表的工作也可以放在后续阶段（如课程主体实验的编译器）。假设是在词法分析阶段负责将标识符加入当前符号表（当该标识符已在表中时查找到相应表项），那么就可以将该标识符相应表项的指针作为属性值返回给语法分析程序。因为符号表与作用域密切相关，需要有后续阶段的配合，所以如图 1 所示的单遍词法分析在一定程度上不方便伴随有符号表维护的工作。

2. 词法分析程序的设计和实现

2.1 词法分析程序中如何识别单词

词法分析中识别下一个单词的过程，简单来看就是逐个读取字符，然后将它们拼在一起的过程。因此，本质的问题是在这个拼单词的过程中如何去匹配下一个有意义的词法单元。

要识别出有意义的词法单元，主要是依据程序设计语言的词法规则描述。描述一个语言的词法规则，通常需要借助形式化或半形式化的描述工具，以确信没有歧义性。常见的可用于词法规则描述的工具如：状态转换图，扩展巴克斯范式（EBNF），有限状态自动机，正规表达式等。

在下面一的小节中，我们将会看到用状态转换图和 EBNF 描述词法规则以及手工实现词法分析程序的实际例子。

有限状态自动机和正规表达式是适合于正规语言的描述及处理的形式模型。在现实程序设计语言中，几乎任何一种有意义的词法单元的所有单词集合都是正规语言，这些模型的表达能力足以描述任何程序设计语言的词法规则。另一方面，基于这些语言模型处理正规语言的方法已经非常成熟。因此，以这些模型为基础来设计词法分析程序的自动构造工具是人们目前普遍采取的途径。

我们将在第3节介绍有关词法分析程序自动构造的一般原理和方法。

2.2 词法分析程序的设计与实现

本小节以PL/0词法分析为背景，介绍一个词法分析程序的设计实例，借此读者可进一步了解词法分析程序构造的一些细节。

PL/0 语言是 Pascal 语言的一个简单子集。PL/0 编译程序项目（简称 PL/0 项目），在我们往届学生的实验中占据了重要地位。97级之前，我们的实验只有PL/0 项目；98~02级是在PL/0 项目和Decaf项目之间任选（Decaf 35分，PL/0 20分）；在03~04级包括计50班，PL/0项目必做，但视为平时成绩的一部分只占5分，Decaf项目成为必做的课程实验主体占35分；05之后，取消了PL/0项目，以Decaf/Mind项目为主（18级开始启用MiniDecaf项目）。

可以看出，PL/0项目逐渐淡出了我们的课程实验。但是，有关PL/0 编译程序的内容一直没有真正离开过这门课程，原因包括：（1）编译原理课程是面向一般的程序设计语言实现的，原则上不应该倾向某一类语言；（2）根据 PL/0 编译程序解读全局符号表结构，递归下降分析程序结构，以及含嵌套子程序定义的运行时栈结构等很有帮助，而这些技术在后来的Decaf项目（以及现在的MiniDecaf项目）框架中没有体现；（3）PL/0 项目中提供的编译程序框架只有1000行出头的代码量，同学在课程开始不久便可读完，对后续课程内容的理解很有帮助，同时课堂上也方便举例；（4）PL/0 项目框架中使用的低级中间码虚拟机（类P-code解释程序实现的类P-code虚拟机），符合现代语言（如Java,C#）设计潮流；（5）PL/0项目框架也是容易扩充的，比如增加对面向对象语言以及函数式语言实现机制的支持。

对于本学期的课程，如果有同学感兴趣“手工词法语法分析器”，可以根据需要自行阅读PL/0 项目框架中语法分析部分的源码，而其词法分析部分的源码可作为本讲“手工实现词法分析程序”内容的课外参考阅读材料。

PL/0 项目框架的源码最初是由 Wirth（Pascal 语言的设计者）给出了Pascal语言实现的版本。在以前年级的实验中使用的是我们的助教（蔡锐、梁英毅、龚珩、高崇南和王曦等）维护的 C 和 Java 版。我们会在近日将这些代码提供给大家（见网络学堂），供参考。

下面我们结合PL/0 编译程序中的词法分析过程来介绍词法分析程序的一般设计和实现方法（手工实现）。**为淡化语言本身，我们在课堂讲稿（ppt）中只称为某语言，而非 PL/0 语言。**

<无符号整数> ::= <数字>{<数字>}
<标识符> ::= <字母>{<字母>|<数字>}
<字母> ::= *a* | *b* | … | *X* | *Y* | *Z*
<数字> ::= 0 | 1 | 2 | … | 8 | 9
<保留字> ::= const | var | procedur | begin | end | odd | if | then | call | while | do | read | write
<运算符> ::= + | - | * | / | = | # | < | <= | > | >= | :=
<界符> ::= (|) | , | ; | .

表 3 PL/0 语言词法的 EBNF 描述

表3是 PL/0 语言词法类别的一种 EBNF 描述。可以看出，PL/0的单词可分为五个类别：保留字、运算符、标识符、无符号整数、界符。保留字有13个。运算符有11个，分别是4个整型运算符号，6个比较运算符号，以及赋值运算符。还有括号、分号等界符。无符号整数是由一个或多个数字组成的序列，标识符是字母开头的字母数字序列。

下面，如不特别申明，PL/0 编译程序是指 Java 语言版。

PL/0 编译程序设计了31个词法单元：标识符1个，无符号整数1个，保留字13个，运算符11个，以及界符5个。这31个词法单元，对应31个单词符号：

```
public enum Symbol {
    nul,        ident,      number,     plus,       minus,
    times,      slash,      oddsym,     eql,        neq,
    lss,        leq,        gtr,        geq,        lparen,
    rparen,     comma,      semicolon,  period,     becomes,
    beginsym,   endsym,     ifsym,      thensym,    whilesym,
    writesym,   readsym,    dosym,      callsym,    constsym,
    varsym,     procsym,
}
```

其中，nul 不属于31个单词符号，只是出于实现技术的考虑，代表“不能识别的符号”。

例如，在 PL/0 词法分析程序扫描下列语句

```
position := initial + rate * 60 ;
```

之后，所生成的单词符号形式的语句由如下单词符号序列组成：

ident becomes ident plus ident times number semicolon

PL/0 语法分析程序在需要读取下一个单词时，就调用如下函数：

public void getsym()

getsym() 返回下一个单词的单词记录。单词记录包含单词符号和属性值两个部分。

PL/0 词法分析程序定义了如下变来传递单词符号和单词的属性值：

- 通过变量 sym 传递单词符号

  public Symbol sym;

- 通过变量 id 传递标识符单词的属性值，即标识符的名字

  public String id;

- 通过变量 num 传递无符号整数单词的属性值，即它的整数数值

  public int num;

比如，在识别出标识符 position 之后，变量 sym 的值被置为 ident，id 的值被置为“position”；在识别出无符号整数 60 之后，变量 sym 的值被置为 number，num 的值被置为整数值 60。

getsym() 逐个读取下面的字符，然后将它们拼成下一个有意义的词法单元。

根据表3中的EBNF描述，我们可以直接写出单词识别过程。有时，为方便也经常使用状态转换图更加直观地刻画词法规则，辅助识别过程的实现。例如，图3是描述 PL/0 词法规则的一个状态转换图。

附录 A 列出了一个 PL/0 词法分析程序（Java 版）的完整代码。大家可以通过阅读了解一下拼标识符或保留字matchKeywordOrIdentifier()，拼数 matchNumber()，拼运算符 matchOperator()等函数的设计方法。

值得注意的是，在识别字母数字串的词法单位后，如何来区分标识符和保留字。常采取的方法是预设一个保留字表，通过查表来确定是否保留字。

另外，在拼双符号运算符之类的单词时，要注意到可能需要进行字符“退还”。如，在读取字符‘<’后，如果下一字符是‘=’，则所识别的单词是小于等于号‘<=’；否则，识别的单词是小于号‘<’，但此时要注意退还已经读到的一个非‘=’字符，即需要保证下一次读到的字符仍然是那个非‘=’字符。图3 所给出的状态转换图中，我们对识别到“小于号”和“大于号”的状态表识中特别标以“*”，以示区别此类情形。

函数 getsym( )在需要取下一字符时所调用的函数是 getch( )，其功能是漏掉空格，读取一个字符。值得注意的是，为减少磁盘I/O次数，每次是从文件中读取一行。

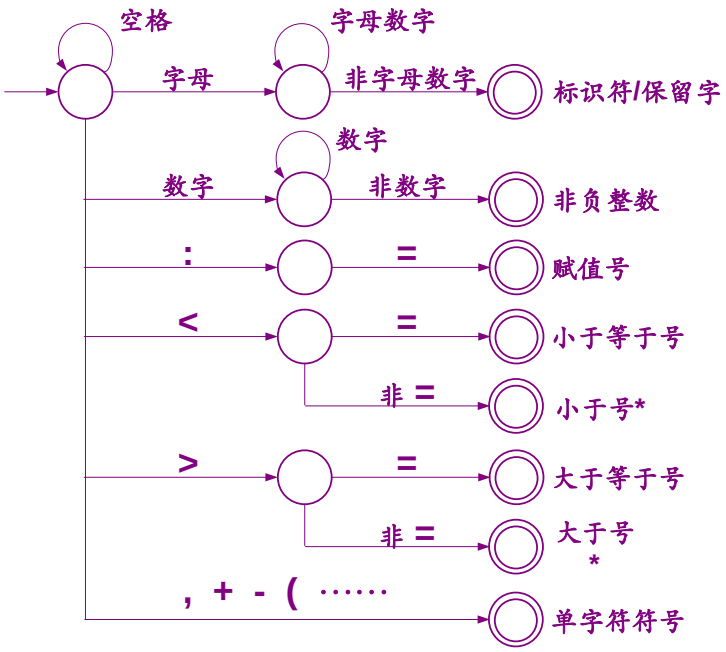

图 3　PL/0 词法规则的状态转换图

函数 getsym( )在需要取下一字符时所调用的函数是 getch( )，其功能是漏掉空格，读取一个字符。值得注意的是，为减少磁盘I/O次数，每次是从文件中读取一行。

3. 词法分析程序自动构造

在有限自动机和正规表达式基础上容易实现词法分析程序的自动构造，典型的过程可能是：

（1） 使用者用正规表达式作为词法规则的形式描述，每一类词法单元都对应一个正规表达式，所有正规表达式以文本方式作为自动构造工具的输入；

（2） 自动构造工具将每一个正规表达式转换成有限自动机的形式，比如使用 Thompson 构造法（参见形式语言与自动机课程）将正规表达式转换成 ε-NFA ；

（3） 可选：增加一个新的开始状态，从该状态引一条 ε-转移边到上述每一个 *ε-NFA* 的初态，得到一个新的 *ε-NFA*；

（4） 必要时，自动构造工具会将这些 *ε-NFA* 确定化，比如使用子集构造法（参见形式语言与自动机课程）得到等价的 *DFA*；

（5） 必要时，自动构造工具会将有限自动机最小化化，比如使用形式语言与自动机课程中介绍的方法得到等价拥有状态数目最少的 *DFA*；

（6） 若执行过第 3 步，那么就模拟单个完整的自动机。否则，自动构造工具按照一定的控制策略生成词法分析程序中扫描程序的代码，该扫描程序可以选择对每一类词法单元所对应的有限自动机进行模拟运行，并从当前输入符号序列中识别下一个单词，然后返回相应的单词记录；

通常，每一类词法单元的单词记录数据结构也需要由使用者来给定，连同每一类词法单

元对应的正规表达式一同作为自动构造工具的输入；单词记录中的单词符号一般会由使用者预先设定。另外，一些工具会按照描述的先后次序，以及可识别单词的最大长度等来确定内部控制策略，这些约定通常也要明确告知使用者。

上述步骤（1）要求给出每一类词法单元的一个正规表达式。有时，直接设计正规表达式是比较困难的。例如：假若想要为 Java 程序中所允许的“注释”给出正规表达式，这类注释以”/*”开始，以”*/” 结束，在”/*”和”*/”之间，除了”*/”序列外，可以出现任意字符。对此类注释，某些同学可能会认为构造 DFA 比构造正规表达式更容易，因而可先构造 DFA，然后再转换为正规表达式。

练习

1 针对表 3 中 EBNF 描述的各单词类别，设计相应的正规表达式。（非书面作业）

2 针对第 3 节中所描述的“注释”单词类别，设计相应的正规表达式。你可以先试着直接设计这个正规表达式，若觉得复杂，就采用先构造 DFA 的方法进行设计。如果有兴趣，还可以在第一阶段实验中测试你的结果。（非书面作业）

3 阅读并分析 PL/0 编译器的词法分析程序（附录 A）。（非书面作业，可为有兴趣“手工实现词法分析程序”的同学提供参考）

附录 A PL/0 编译程序中词法分析程序的完整代码（Java 版）

作者：梁英毅

```
import java.io.BufferedReader;
import java.io.IOException;


/**  词法分析器负责的工作是从源代码里面读取文法符号，这是 PL/0 编译器的主要组成部分之一。*/
public class Scanner {
    /**
     * 刚刚读入的字符
     */
    private char ch = ' ';

    /**
     * 当前读入的行
     */
    private char[] line;

    /**
     * 当前行的长度（line length）
```

```
 */
public int ll = 0;

/**
 * 当前字符在当前行中的位置（character counter）
 */
public int cc = 0;

/**
 * 当前读入的符号
 */
public Symbol sym;

/**
 * 保留字列表（注意保留字的存放顺序）
 */
private String[] word;

/**
 * 保留字对应的符号值
 */
private Symbol[] wsym;

/**
 * 单字符的符号值
 */
private Symbol[] ssym;

// 输入流
private BufferedReader in;

/**
 * 标识符名字（如果当前符号是标识符的话）
 * @see Parser
 * @see Table#enter
 */
public String id;

/**
 * 数值大小（如果当前符号是数字的话）
 * @see Parser
 * @see Table#enter
 */
public int num;
```

```
/**
 * 初始化词法分析器
 * @param input PL/0 源文件输入流
 */
public Scanner(BufferedReader input) {
	in = input;

	// 设置单字符符号
	ssym = new Symbol[256];
	java.util.Arrays.fill(ssym, Symbol.nul);
	ssym['+'] = Symbol.plus;
	ssym['-'] = Symbol.minus;
	ssym['*'] = Symbol.times;
	ssym['/'] = Symbol.slash;
	ssym['('] = Symbol.lparen;
	ssym[')'] = Symbol.rparen;
	ssym['='] = Symbol.eql;
	ssym[','] = Symbol.comma;
	ssym['.'] = Symbol.period;
	ssym['#'] = Symbol.neq;
	ssym[';'] = Symbol.semicolon;

	// 设置保留字名字,按照字母顺序，便于折半查找
	word = new String[] {"begin", "call", "const", "do", "end", "if",
		"odd", "procedure", "read", "then", "var", "while", "write"};

	// 设置保留字符号
	wsym = new Symbol[PL0.norw];
	wsym[0] = Symbol.beginsym;
	wsym[1] = Symbol.callsym;
	wsym[2] = Symbol.constsym;
	wsym[3] = Symbol.dosym;
	wsym[4] = Symbol.endsym;
	wsym[5] = Symbol.ifsym;
	wsym[6] = Symbol.oddsym;
	wsym[7] = Symbol.procsym;
	wsym[8] = Symbol.readsym;
	wsym[9] = Symbol.thensym;
	wsym[10] = Symbol.varsym;
	wsym[11] = Symbol.whilesym;
	wsym[12] = Symbol.writesym;
}
```

```
/**
 * 读取一个字符，为减少磁盘 I/O 次数，每次读取一行
 */
void getch() {
	String l = "";
	try {
		if (cc == ll) {
			while (l.equals(""))
				l = in.readLine().toLowerCase() + "\n";
			ll = l.length();
			cc = 0;
			line = l.toCharArray();
			System.out.println(PL0.interp.cx + " " + l);
			PL0.fa1.println(PL0.interp.cx + " " + l);
		}
	} catch (IOException e) {
		throw new Error("program imcomplete");
	}
	ch = line[cc];
	cc ++;
}

/**
 * 词法分析，获取一个词法符号，是词法分析器的重点
 */
public void getsym() {
	// Wirth 的 PL/0 编译器使用一系列的 if...else...来处理
	// 但是你的助教认为下面的写法能够更加清楚地看出这个函数的处理逻辑
	while (Character.isWhitespace(ch))		// 跳过所有空白字符
		getch();
	if (ch >= 'a' && ch <= 'z') {
		// 关键字或者一般标识符
		matchKeywordOrIdentifier();
	} else if (ch >= '0' && ch <= '9') {
		// 数字
		matchNumber();
	} else {
		// 操作符
		matchOperator();
	}
}

/**
 * 分析关键字或者一般标识符
```

```
 */
void matchKeywordOrIdentifier() {
	int i;
	StringBuilder sb = new StringBuilder(PL0.al);
	// 首先把整个单词读出来
	do {
		sb.append(ch);
		getch();
	} while (ch >= 'a' && ch <= 'z' || ch >= '0' && ch <= '9');
	id = sb.toString();

	// 然后搜索是不是保留字（请注意使用的是什么搜索方法）
	i = java.util.Arrays.binarySearch(word, id);

	// 最后形成符号信息
	if (i < 0) {
		// 一般标识符
		sym = Symbol.ident;
	} else {
		// 关键字
		sym = wsym[i];
	}
}

/**
 * 分析数字
 */
void matchNumber() {
	int k = 0;
	sym = Symbol.number;
	num = 0;
	do {
		num = 10*num + Character.digit(ch, 10);
		k++;
		getch();
	} while (ch>='0' && ch<='9');			// 获取数字的值
	k--;
	if (k > PL0.nmax)
		Err.report(30);
}

/**
 * 分析操作符
 */
```

```
    void matchOperator() {
        // 请注意这里的写法跟 Wirth 的有点不同
        switch (ch) {
        case ':':           // 赋值符号
            getch();
            if (ch == '=') {
                sym = Symbol.becomes;
                getch();
            } else {
                // 不能识别的符号
                sym = Symbol.nul;
            }
            break;
        case '<':           // 小于或者小于等于
            getch();
            if (ch == '=') {
                sym = Symbol.leq;
                getch();
            } else {
                sym = Symbol.lss;
            }
            break;
        case '>':           // 大于或者大于等于
            getch();
            if (ch == '=') {
                sym = Symbol.geq;
                getch();
            } else {
                sym = Symbol.gtr;
            }
            break;
        default:            // 其他为单字符操作符（如果符号非法则返回 nil）
            sym = ssym[ch];
            if (sym != Symbol.period)
                getch();
            break;
        }
    }
}
```