第 19 卷 第 3 期 华侨大学学报(自然科学版) Vol. 19 No. 3
1998 年 7 月 Journal of Huaqiao University (Natural Science) Jul. 1998

基于代码驱动的词法分析器的改进*

张全伙 陈飞舟

(华侨大学计算机科学系, 泉州 362011)

摘要 介绍两种基本的词法分析器的实现, 就代码驱动的词法分析器的不足之处提出改进方法, 并用 PASCAL 语言给出了实现的程序段.

关键词 词法分析, 表格驱动, 代码驱动

分类号 TP 314

随着形式语言与自动机理论的提出, 计算机编译理论有了突破性的进展, 特别是词法分析和语法分析理论已发展得相当成熟. 在词法分析的实现上, 主要有基于表格驱动和代码驱动两种方法. 表格驱动可通过工具软件生成驱动表格, 简单高效, 但其存储较复杂; 而代码驱动一般采用手工编程实现, 适合简单的词法分析, 但代码量相对较大, 效率有时不如表格驱动方法高. 关于词法分析的讨论, 以往只注重于理论上的论述, 对具体代码实现的实用和效率较少涉及. 本文正是基于这种考虑, 针对代码驱动的词法分析器的实现中存在的问题, 如字母数字字符的识别、超前字符处理和注释的过滤处理等, 提出新的解决方法, 并用 PASCAL 语言给出相应的程序段.

1 基本词法分析器的实现

1.1 基于表格驱动的词法分析器

基于表格驱动的词法分析器[1]的实现方法是: 将词法自动机的状态作为驱动表格的行, 输入字符作为表格的列. 词法分析程序从初始状态开始, 根据输入字符查表进入下一状态, 然后再从该状态出发, 根据输入字符查表进入下一状态, 如此直至最长匹配终态, 返回该终态值作为词码. 例如, 简单实数的正规式为 "数字(数字)* · 数字(数字)* ", 它的识别自动机如图 1 所示. 据此, 可得到附表的驱动表格(其中空白表示错误).

附表 驱动表格

状态	·	0	1	2	3	4	5	6	7	8	9	…
1		2	2	2	2	2	2	2	2	2	2	
2	3	2	2	2	2	2	2	2	2	2	2	
3		4	4	4	4	4	4	4	4	4	4	
4		4	4	4	4	4	4	4	4	4	4	

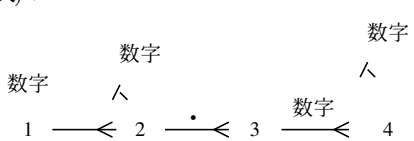

图1 简单实数识别自动机

* 本文 1998-02-19 收到; 福建省自然科学基金资助项目

表格驱动程序可用 PASCAL 描述为

```
CurrentState= 1;
Read(f, ch);
while not eof (f) do begin
  NextState: = Table[Currentstate][ch];
  if NextState= ErrorState then break; {退出循环}
  CurrentState: = NextState;
  Read(f, ch);
end;
if IsFinalState(CurrentState) then begin {如果该状态是终态}
  {返回相应词码}
end else begin
  {词法分析错误处理}
end;
```

从上述例子可以看出, 表格驱动程序非常简单, 只要将自动机的类树型结构利用表格化为线性处理方式, 以数据的复杂性换取实现的简单和高效. 由于驱动表格庞大, 在实际应用中常采用压缩表的存储结构(如 Hash 表). 它的不足之处是, 词法分析类的小修改将带来驱动表格的大变动, 因此驱动表格通常用工具软件来生成, 把类产生式语言转化为驱动表格. 典型的是 ScanGen 和 Lex 两个软件, 当 ScanGen 到达终态时, 返回定义的主次词码, 而当 Lex 到达终态时, 执行相应的子程序返回定义的词码.

1.2 基于代码驱动的词法分析器[2]

基于代码驱动的实现方法是将词法自动机的每个状态对应一小段代码, 根据输入字符执行相应的代码分支, 直至该代码执行完毕, 返回相应的词码. 例如, 标识符的正规式是: 字母(字母|字母)*, 它的识别自动机如图 2 所示, 对应的代码段用 PASCAL 描述如下:

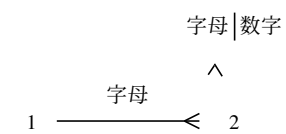

图2 标识符识别自动机

```
Read(f, ch);
if IsAlpha(ch) chen begin
   Token: = '';
   while IsAlpha(ch) or IsNumber(ch) do begin
     Token: = Token+ ch;
     Read(f, ch);
   end;
   Unread(f, ch); {将当前字符回退到文件 f 中}
end;
```

2 代码实现的改进

目前的教科书中, 基本上都采用基于代码驱动方法对词法分析进行描述, 但对于具体实现

上存在的低效,及难定位错误等问题一般不予考虑. 我们从分析问题入手,对代码实现加以改进.

2.1 字母、数字及特殊字符的识别

判断一个字符是否字母或数字时,一般采用 IsAlpha 或 IsNumber 等函数(或宏)来识别. 我们以 IsAlpha 的实现为例:

```
Function IsAlpha(ch: char): boolean;
Begin
  if((ch> = ´a´) and(ch< = ´z´) or((ch> = ´A´) and(ch< = ´Z´));
    then return TRUE
    else return FALSE;
End;
```

从上面程序段可看出,判断一个字符为字母需作两次或四次判断,显然执行效率很低,而采用字符转义表可明显提高效率. 字符转义表是,为每一字符在表中的相应位置赋予该字符所属类型. 例如,charTable[´a´] = chalpha 表示´a´属于字母,可用以下代码段建立转义表,即

```
Type CharType= (chNone, chAplha, chNumber, …);
Var CharTable: array char of CharType;
for ch: = chr(0) to chr(255) do {初始化}
  CharTable[ch]: = chNone;
for ch: = ´a´ to ´z´ do
  CharTable[ch]: = chAlpha;
for ch: = ´A´ to ´Z´ do
  CharTable[ch]: = chAlpha;
for ch: = ´0´ to ´9´ do
  CharTable[ch]: = chNumber;
```

这样,只要对要处理的字符进行一次查表,即可知道字符所属类型. 据此,可将前面代码驱动的程序段改写为

```
Read(f, ch);
if CharTable [ch]= chAlpha then begin
  Token: = '';
  while (CHarTable[ch]= chAlpha) or (CHarTable[ch]ch= chNumber) do begin
    Token: = Token+ ch;
    Read(f, ch);
  end;
  UnRead(f, ch); {将当前字符回退到文件 f 中}
end;
```

2.2 超前字符处理

在词法分析中,对于有双重含义的字符往往采用超前读入下一字符来确定其含义. 例如,‘>’字符可能是大于号,也可能是大于等于号的前一个字符. 此时,如果读入的字符没用,就

需将该字符回送到文件,以便下次读入.当该字符是一行的最后一个字符时,文件指针已指向下一行,若将该字符简单加到行首,则该字符所属的行数是错误的,这会给错误定位带来麻烦.一种解决方案是将该字符缓冲而不回送,但这种方案在词法分析的实现中较为困难.究其原因是,由于只有当前字符可见,使得为了看到下一字符,要使下一字符成为当前字符(即读入下一字符).如果读入字符时,当前字符和下一字符均可见,问题便迎刃而解.为此,读入字符函数可修改为

```
Function GetChar (var CurrentChar, NextChar char): Boolean;
Begin
  while (CurrentPos> Length(CurrentLine) do begin {读入下一非空行}
    if eof(SrcFile) then begin
      CurrentChar:= chr(26);{End of Program};
      NextChar:= chr(26);
      return FALSE;
    end;
    Readln(SrcFile, CurrentLine);
    CurrentLine [Length( CurrentLine) + 1]:= # 0;
    CurrentPos:= 1; {设置当前字符列数为一}
    Lines: = Lines+ 1; {行数加一}
  end;
  CurrentChar:= CurrentLine[CurrentPos];
  CurrentPos: = CurrentPos+ 1;
  NextChar:= CurrentLine[CurrentPos];
  return TRUE;
End;{GetChar}
```

因此,处理'>','<'的代码如下所示:

```
GetChar(CurrentChar, NextChar);
case CurrentChar of
  ⋮
  '> ':begin
    if NextChar= '= 'then begin
      GetChar(CurrentChar, NextChar);{读入'= '字符}
      return tkLargerEqual; {返回大于等于号}
    end else return tkLarger;{返回大于号}
  end;{'> '};
  ⋮
end:{case}
```

值得一提的是,这种方法对于 Pascal, C++ 这类只需预读 1 到 2 个字符的语言特别有效,但不适用于象 Fortran 这类需预读多个(超过 3 个)字符的语言.目前开发出来的新语言大部

分可用这种方法处理.

2.3　注释的过滤处理[3,4]

在程序设计中,适当使用注释可以增强程序的可读性,以及暂时屏蔽不需要的代码. 注释对语法分析并无用处,也不参加编译,因此在词法分析阶段就应该将注释过滤掉. 在各类程序设计语言中,注释可分为单行注释和整段注释,在 C++ 语言中,单行注释用 // 开头,直到该行结尾;整段注释用/* 开头,以* /结尾. 通常的注释需要预读,使过滤处理变得麻烦. 我们采用改写后的读入字符函数,使过滤处理变得甚为简单,其代码为

```
GetChar(CurrentChar, NextChar);
case CurrentChar of
    ⋮
  '/': begin
    if NestChar= '/' then begin {单行注释}
      while NestChar< > # 0 do GetChar(CurrentChar, NextChar);
    end else if NextChar= '* ' then begin {多行注释}
      GetChar(CurrentChar, NextChar); {读入'* '字符}
      repeat;
        GetChar(CurrentChar, NextChar);
      until((CurrentChar= '* ') and (NExtChar= '/')
      GetChar(CurrentChar, NextChar); {读入'/'字符}
    end else return tkDiv; {返回除法的词码}
  end;{'/'};
    ⋮
end; {case}
```

这个代码段可用于处理普通的注释,但对于嵌套注释则无能为力,对此,可考虑用以下程序代码的处理.

```
/*  下面的代码被注释,暂时不执行
for (i= 0; i< 10; i+ + ) {
  /*  输出 A[i]的值以供测试 * /
  printf("A[%d]= %d\n", i, A[i]);
}
* /
```

注释的开始符号'/* '碰到第一个'* /',处理便告结束,碰到应该与之配对的'* /'时会出现'* /'符号无'/* '匹配的错误. 这个问题很容易解决,只要在程序中增设一个注释层次计数器,当碰到'/* '时,该计数器加一;当碰到'* /'时,计数器减一,如此直至计数器的值为零时退出. 据此程序段改写为

```
GetChar(CurrentChar, NextChar);
case CurrentChar of
    ⋮
```

```
'/': begin
  if NestChar= '/' then begin {单行注释}
    while NestChar< > = 0 do GetChar(CurrentChar, NextChar);
  end else if NextChar= '* ' then begin {多行注释}
    GetChar(CurrentChar, NextChar){读入 '* ' 字符}
    RemNum:= 1;
    while RemNum> 0 do begin
      GetChar(CurrentChar, NextChar);
    if ((CurrentChar= '* ') and (NextChar= '/') then RemNum
    := RemNum- 1;
    else if ((CurrentChar= '/') and (NextChar= '* ') then RemNum
    := RemNum+ 1;
  end; {while}
    GetChar(CurrentChar, NextChar); {读入 '/' 字符}
  end else return tkDiv; {返回除法的词码}
end;{'/'}
  ⋮
end; {case}
```

参 考 文 献

1 陈火旺, 钱家骅, 孙永强. 程序设计语言编译原理. 北京: 国防工业出版社, 1984. 29～50

2 丘玉圃, 刘春年, 刘建丽. 编译程序构造方法. 北京: 科学出版社, 1991. 4～90

3 缪淮扣. 语言处理程序. 北京: 清华大学出版社, 1993. 3～55

4 谷惠英, 刘甲耀, 严桂兰. 程序设计图形化的集约环境. 华侨大学学报(自然科学版), 1996, 18(2): 213～218

Improvement of Code-Driven Lexical Analyzer

Zhang Quanhuo Chen Feizhou

(Dept. of Comput. Sci., Huaqiao Univ., 362011, Quanzhou)

Abstract The implementation of two basic lexical analyzers is briefed. As to the inadequacy of code-driven lexical analyzer, a method is advanced for its improvement; and the implemented program segment is given in PASCAL language.

Keywords lexical analysis, table driven, code driven