词法分析器

## 一、需求分析

词法分析器是编译器的一个重要组成部分，也是编译器的第一个主要模块，词法分析器需要完成以下任务：

1. 从左至右逐个字符对源程序进行扫描，产生一个个单词符号；
2. 把作为字符串的源程序改造成为单词符号串的中间程序；
3. 输入源程序，输出单词符号及其对应编码

## 二、总体框架

根据需求分析可以知道，词法分析器在实际工作过程中需要完成以下内容：

1. 单词符号编码

为需要识别的单词符号进行编码，使得不同的单词符号有不同的识别标识，编码最好有一定的通用性（每一大类单词符号便于识别），扩展性（当有新的需要识别的单词符号添加时便于添加）等。

1. 预处理

源代码中可能有很多对单词符号识别没有明显意义的东西比如注释部分，多余的回车换行符，制表符，空格等，需要先对其进行归一化处理。

1. 单词符号识别

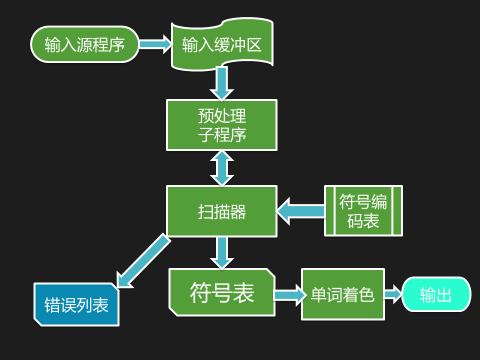
单词符号识别是词法分析器的主要部分也是关键和难点部分，需要识别出每一个单词符号及其对应的类别。

1. 错误处理

输入的源程序中很有可能存在单词符号的拼写上不满足词法要求的单词符号，需要对其进行很好的识别，并指出错误所在。

在以上处理的基础上利用最终的结果还可以对单词进行着色，常见的代码编辑工具都有这个功能，着色后的代码便于阅读且美观。

根据以上分析，我设计的词法分析器的框架如下：



## 三、关键部分实现

根据代码框架的分析和设计，词法分析器中有以下关键部分需要处理：

1. 单词符号编码

词法分析中，单词符号分以下几类：

1. 关键字：int,char,short,for,while,do,break,…
2. 标识符：变量名，数组名，函数名……
3. 运算符：+, -, \*, /, ++, --, <<, >>, <, >, !=, <=, >=, &,…
4. 常量：整型，浮点型，字符型，文本型……
5. 界符：(,),{,},[,],,,……

对于以上五种类型，编码形式如下：

//界符

#define XBO\_OPENP A 0x00000001 //(

#define XBO\_CLOSEPA 0x00000002 //)

//运算符

#define XOP\_PLUS 0x10000001 //+

#define XOP\_MINUS 0x10000002 //-

//关键字

#define XKEY\_AUTO 0x20000001 // auto

#define XKEY\_BREAK 0x20000002 // break

#define XKEY\_CASE 0x20000003 // case

//常量

#define XCON\_CHAR 0x30000001

#define XCON\_INT 0x30000002

//标识符

#define XIDEN 0x40000000

按照以上格式编码主要考虑最高位4比特标识类型，这样在类型识别时只需进行简单的与运算或者移位运算就可以区分大的类别。

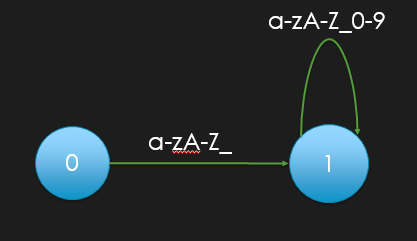
1. 特殊格式单词符号识别

特殊字符例如变量名称，整数等的识别由于需要满足特殊的格式，所以需要根据DFA的相关原理进行处理，处理过程如下：

1. 标识符的识别，标识符满足这样的格式：

[a-zA-Z\_]+[a-zA-Z0-9]\*

所以DFA状态转换图如下：

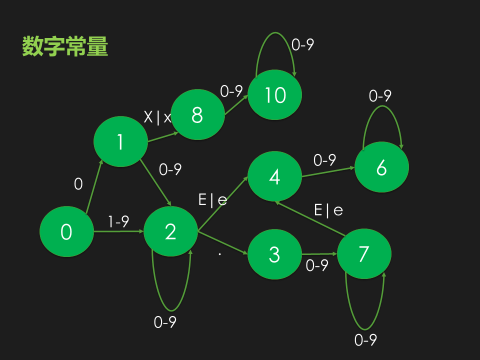


1. 数值型识别

数值型是比较难处理的一类主要有以下几类：整数、浮点数、科学计数，十六进制数：

（0[xX][0-9]+|Ɛ）|[0-9]+(.[0-9]+| Ɛ)([Ee]0-9+)

DFA如下：

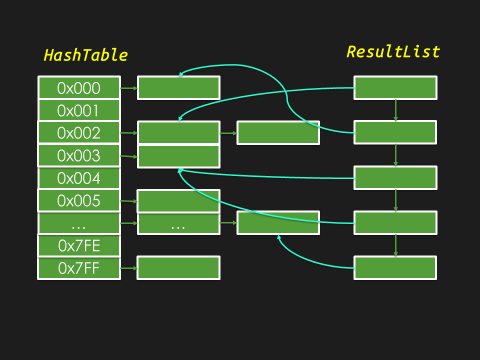


1. 数据结构和存储结构

词法分析器实现过程中的几个关键数据结构如下：

|  |
| --- |
| typedef struct dynamicstr {  int len;  char\* str;  } dynamicstr;  //动态字符串，主要用来实现存储单词符号串，使用这种结构的好处是便于存储，长度的指明可以有效防止移除 |
| typedef struct wordhash {  unsigned int tokencode;  struct wordhash \* next;  dynamicstr \*word;  }wordhash;  //哈希表，主要用来存储每一个单词符号及其对应的hash值，tokencode是该单词符号的编码值，next指针是用来处理hash冲突的。 |
| typedef struct wordlist {  wordhash\* hashkey;  struct wordlist \*next;  }wordlist;  //单词列表，词法分析最终的结构是存储在这样的一个链表中的，hashkey是指向hash表中的结点的 |
| typedef struct errorlist {  wordhash\* hashkey;  int linenum;  struct errorlist\* next;  }errorlist;  //错误列表，用来存储词法分析过程中出错的单词符号，其中linenum用来指明出错的具体位置 |

根据以上的数据结构，最终的存储结构如下：



1. 哈希算法

Hash表的设计很好的提高了算法的时间效率和空间效率，而hash算法的选择是基于以下几点：

1. 算法计算过程简单，计算开销小
2. 适合字符串的处理
3. hash值冲突率比较低

根据网上对常见字符串算法的测评：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hash函数 | 数据1 | 数据2 | 数据3 | 数据4 | 数据1得分 | 数据2得分 | 数据3得分 | 数据4得分 | 平均分 |
| BKDRHash | 2 | 0 | 4774 | 481 | 96.55 | 100 | 90.95 | 82.05 | 92.64 |
| APHash | 2 | 3 | 4754 | 493 | 96.55 | 88.46 | 100 | 51.28 | 86.28 |
| DJBHash | 2 | 2 | 4975 | 474 | 96.55 | 92.31 | 0 | 100 | 83.43 |
| JSHash | 1 | 4 | 4761 | 506 | 100 | 84.62 | 96.83 | 17.95 | 81.94 |
| RSHash | 1 | 0 | 4861 | 505 | 100 | 100 | 51.58 | 20.51 | 75.96 |
| SDBMHash | 3 | 2 | 4849 | 504 | 93.1 | 92.31 | 57.01 | 23.08 | 72.41 |
| PJWHash | 30 | 26 | 4878 | 513 | 0 | 0 | 43.89 | 0 | 21.95 |
| ELFHash | 30 | 26 | 4878 | 513 | 0 | 0 | 43.89 | 0 | 21.95 |

显然BKDRHash算法的表现更加优秀，BKDRHash算法的实现如下：

|  |
| --- |
| #define MAXKEY 4096  unsigned int BKDRHash(dynamicstr \*str)  {  unsigned int seed = 131;  unsigned int hash = 0;  for (int i = 0; i < str->len;i++)  {  hash = hash \* seed + str->str[i];  }  return (hash & 0x7FFFFFFF) % MAXKEY;  } |

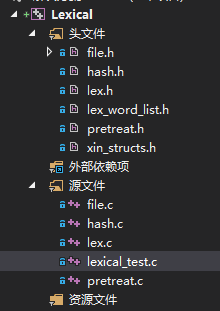
其中MAXKEY是hash表的大小，我选择4096。

1. 单词着色

单词着色的目的是是代码显示出来的效果更加美观，可读性更强，也是词法分析后水到渠成的一件事，单词着色的主要依据就是对不同类型的单词符号选择不同的着色方案，具体实现如下：

|  |
| --- |
| void colorwords(wordlist\* list) {  int token;  HANDLE hd = GetStdHandle(STD\_OUTPUT\_HANDLE);    while (list != NULL) {  token = list->hashkey->tokencode;  token = token >> 28;  switch (token) {  case 0:  SetConsoleTextAttribute(hd, FOREGROUND\_GREEN);  break;  case 1:  SetConsoleTextAttribute(hd, FOREGROUND\_RED);  break;  case 2:  SetConsoleTextAttribute(hd, FOREGROUND\_RED | FOREGROUND\_GREEN);  break;  case 3:  SetConsoleTextAttribute(hd, FOREGROUND\_RED | FOREGROUND\_BLUE);  break;  case 4:  SetConsoleTextAttribute(hd, FOREGROUND\_BLUE | FOREGROUND\_GREEN);  break;  default:  SetConsoleTextAttribute(hd, FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);  break;  }  printf("%s ", list->hashkey->word->str);  list = list->next;  }  SetConsoleTextAttribute(hd, FOREGROUND\_RED | FOREGROUND\_GREEN | FOREGROUND\_BLUE);  } |

以上就是关键部分的实现，最终的代码结构如下：



## 四、复杂度分析

时间开销主要是在以下几个方面：

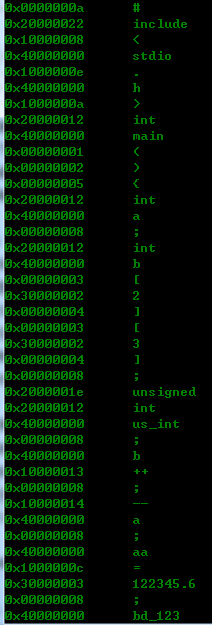
1. 预处理，预处理过程中只需要对源程序进行一遍扫描即可，所以时间复杂度是O(n);
2. 单词符号分离，将源程序分离为一个个独立的单词符号，这个过程也是的输入的源程序进行一遍扫描即可，所以相应的时间复杂度也是O(n);
3. 单词符类型识别，识别出每一个单词符号对应的类别时，由于hash表结构的设计只需要查找相应hash值即可，所以时间复杂度为O(1)。

## 五、样例测试

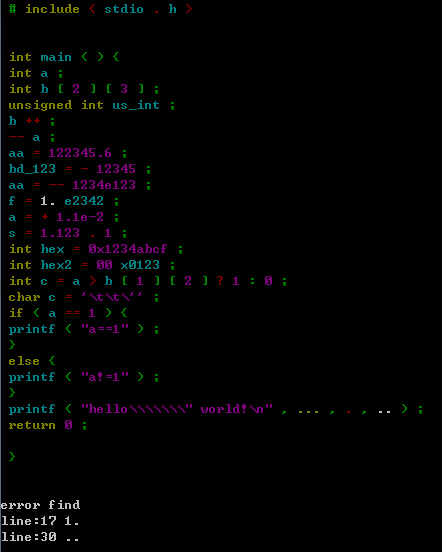
测试样例如下：

|  |
| --- |
| /\*  Xin Complier;  test file    \*/  #include <stdio.h>  int main(){  int a;  int b[2][3];  unsigned int us\_int;  b++;  --a;  aa=122345.6;  bd\_123=-12345;  aa=--1234e123;  f=1.e2342;  a=+1.1e-2;  s=1.123.1;  int hex=0x1234abcf;  int hex2=00x0123;  int c=a>b[1][2]? 1:0;  char c='\t\t\'';  if(a==1){  printf("a==1");  }  else{  printf("a!=1");  }  printf("hello\\\\\\\" world!\n"/\*\*/,...,.,..); //printf "hello world"  return 0;  } |

单词符号识别的结果如下：



单词着色的结果如下：



## 六、参考文献

1.《自己动手写编译器、连接器》王博俊 张宇 清华大学出版社

2.各种常见hash函数比较 ,https://www.byvoid.com/blog/string-hash-compare