第一部分

毋哲

文法修改

任务：

为了能根据文法，更快，更准确地写出递归子程序来做语法分析，我们要进行文法的修改。文法的修改之所以在词法分析之前进行，是因为大多数同学不是很了解文法修改需要改什么地方，如果我们在写完词法分析之后再进行文法修改，很有可能词法部分的文法也有改动，这样再去修改词法分析程序会变得很麻烦。因此首先去修改文法是很有必要的，这样就要求同学们至少在学完语法分析，在了解完为什么要去修改文法之后，再去进行文法的修改。

文法是否需要完全改成 LL(1)才行？

对于只能使用特定语法分析方法，如递归子程序法或者 LL(1)分析法来说，文法改成 LL(1)是必须的，否则便无法使用上述分析方法来进行语法分析。

但是，对于例如如下的文法规则

<程序> ::= [<常量说明部分>][<变量说明部分>]{<有返回值函数定义部分>|<无返回值函数定义部分>}<主函数>

想将其改写成 LL(1)会变得非常复杂和困难，而且很难保证所改写后的文法与之前文法的等价性。此时，我们便可以思考一下，是否真的需要将文法完全改成 LL(1)文法？

观察下面的文法规则

<常量说明部分> ::= const<常量定义>;{ <常量定义>;}

<常量定义> ::= int<标识符>＝<整数>{,<标识符>＝<整数>} | float<标识符>＝<实数>{,<标识符>＝<实数>} | char<标识符>＝<字符>{,<标识符>＝<字符>}

<声明头部> ::= int<标识符> |float <标识符>|char<标识符>

<变量说明部分> ::= <变量定义>;{<变量定义>;}

<变量定义> ::= <类型标识符><标识符>{,<标识符>}

<类型标识符> ::= int | float | char

<有返回值函数定义部分> ::= <声明头部>'(' <参数> ')' '{' <复合语句> '}'

<无返回值函数定义部分> ::= void<标识符>'(' <参数> ')' '{' <复合语句> '}'

我们很容易发现，只要处理到<程序>时，最多只需要向前多读三个单词，我们便可以区分出<常量说明部分><变量说明部分><有返回函数定义部分><无返回函数定义部分>和<主函数>。如 **int a=** 肯定是常量说明部分，**int a;** 肯定是变量说明部分， **int a (** 肯定是有返回函数定义部分。而在程序中编写向前多读三个单词的程序要比改写文法可靠和简便的多。因此，我们在这里根本不需要将文法改成 LL(1)文法，处理起来反而相对简单一些。

我对文法的修改：

下面是原文法：

<加法运算符> ::= ＋｜－

<乘法运算符> ::= ＊｜／

<关系运算符> ::= <｜<=｜>｜>=｜!=｜==

<字母> ::= _｜a｜...｜z｜A｜...｜Z

<数字> ::= 0｜<非零数字>

<非零数字> ::= 1｜...｜9

<字符>　::= '<加法运算符>丨<乘法运算符>丨<字母>丨<数字>'
<字符串>　::= "｛<合法字符>｝"
　　//字符串中可以出现所有合法的可打印字符集中的字符
<程序>　::= [<常量说明部分>][<变量说明部分>]{<有返回值函数定义部分>|<无返回值函数定义部分>}<主函数>
<常量说明部分> ::= const<常量定义>;{ <常量定义>;}
<常量定义>　::= int<标识符>＝<整数>{,<标识符>＝<整数>}
　　| float<标识符>＝<实数>{,<标识符>＝<实数>}
　　| char<标识符>＝<字符>{,<标识符>＝<字符>}
<整数>　::= [＋丨－]<非零数字>｛<数字>｝丨 0
<实数>　::= [＋丨－]<整数>[.<整数>]
<标识符>　::= <字母>｛<字母>丨<数字>｝
<声明头部>　::= int<标识符> |float <标识符>|char<标识符>
<变量说明部分>　::= <变量定义>;{<变量定义>;}
<变量定义> ::= <类型标识符><标识符>{,<标识符>}
<类型标识符>　　::= int | float | char
<有返回值函数定义部分>　::= <声明头部>'(' <参数> ')' '{' <复合语句> '}'
<无返回值函数定义部分>　::= void<标识符>'(' <参数> ')' '{' <复合语句> '}'
<复合语句>　::= [<常量说明部分>][<变量说明部分>]<语句列>
<参数>　::= <参数表>
<参数表>　::= <类型标识符><标识符>{，<类型标识符><标识符>}| <空>
<主函数>　::= void main '(' ')' '{' <复合语句> '}'
<表达式>　::= [＋丨－]<项>{<加法运算符><项>}
<项>　::= <因子>{<乘法运算符><因子>}
<因子>　::= <标识符>丨'('<表达式>')'丨<整数>丨<有返回值函数调用语句>|<实数>|<字符>
<语句>　::= <条件语句>丨<循环语句>丨'{'<语句列>'}'丨<有返回值函数调用语句>;
　　|<无返回值函数调用语句>；丨<赋值语句>;丨<读语句>;丨<写语句>;丨<空>丨<返回语句>;
<赋值语句>　::= <标识符>＝<表达式>
<条件语句> ::= if '('<条件>')'<语句>[else<语句>]
<条件>　::= <表达式><关系运算符><表达式>丨<表达式> //表达式为 0 条件为假，否则为真
<循环语句>　::= while '('<条件>')'<语句>| for'(' <标识符>＝<表达式>；<条件>；<标识符>＝<标识符>(+|-)<步长>')'<语句>
<步长>　::= <非零数字>｛<数字>｝
<有返回值函数调用语句> ::= <标识符>'('<值参数表>')'
<无返回值函数调用语句> ::= <标识符>'('<值参数表>')'
<值参数表>　::= <表达式>{，<表达式>}丨<空>
<语句列>　::= <语句>｛<语句>｝
<读语句>　::= scanf '('<标识符>{，<标识符>}')'
<写语句>　::= printf '(' <字符串>,<表达式> ')'|printf '(' <字符串> ')'|printf '('<表达式>')'
<返回语句>　::= return['('<表达式>')']

下面是修改后的文法：

〈加法运算符〉 : := ＋丨－

<乘法运算符> ::= * | /

<关系运算符> ::= < | <= | > | >= | != | ==

<字母> ::= _ | a | . . . | z | A | . . . | Z

<数字> ::= 0 | <非零数字>

<非零数字> ::= 1 | . . . | 9

<字符> ::= ’<加法运算符> | <乘法运算符> | <字母> | <数字>’

<字符串> ::= ″{<合法字符>}″//字符串中可以出现所有合法的可打印字符集中的字符

<程序> ::= [<常量说明部分>][<变量说明部分>]{<有返回值函数定义部分>|<无返回值函数定义部分>}<主函数>

<常量说明部分> ::= const<常量定义>;{ <常量定义>;}

<常量定义> ::= **int<标识符>＝[＋ | －]<整数>{,<标识符>＝[＋ | －]<整数>}**

| float<标识符>＝[＋ | －]<实数>{,<标识符>＝[＋ | －]<实数>}

| char<标识符>＝<字符>{,<标识符>＝<字符>}

<整数> ::=<非零数字>｛<数字>｝ | 0

<实数> ::= <整数>[.<整数>] **（修改这里的目的是为了让正负号的处理在语法分析阶段而不是词法分析阶段，这样词法分析阶段少了正负号的判断会简便许多，而语法分析阶段因为是使用递归下降法，所以也不会变得多复杂）**

<标识符> ::= <字母>{<字母> | <数字>}

<声明头部> ::= int<标识符> |float <标识符>|char<标识符>

<变量说明部分> ::= <变量定义>;{<变量定义>;}

<变量定义> ::= <类型标识符><标识符>{,<标识符>}

<类型标识符> ::= int | float | char

<有返回值函数定义部分> ::= <声明头部>‘(’ <参数> ‘)’ ‘{’ <复合语句> ‘}’

<无返回值函数定义部分> ::= void<标识符>‘(’ <参数> ‘)’ ‘{’ <复合语句> ‘}’

<主函数> ::= void main ‘(’ ‘)’ ‘{’ <复合语句> ‘}’

<参数> ::= <参数表>

<参数表> ::= <类型标识符><标识符>{, <类型标识符><标识符>}| <空>

<复合语句> ::= [<常量说明部分>][<变量说明部分>]<语句列>

<语句列> ::= <语句>{<语句>}

<语句>　::=<条件语句>|<循环语句>|‘{’<语句列>‘}’ | <函数调用语句>;|<赋值语句>;|<读语句>;|<写语句>;|;|<返回语句>; **（将两种函数合并为一种，便于处理）**

<赋值语句>　::= <标识符>=<表达式>

<条件语句>　::= if ‘(’<条件>‘)’<语句>[else<语句>]

<条件>　::= <表达式><关系运算符><表达式>|<表达式> //表达式为0条件为假，否则为真

<循环语句>　::= while ‘(’<条件>‘)’<语句>| for ‘(’ <标识符>=<表达式>;<条件>;<标识符>=<标识符>(+|-)<步长>‘)’<语句>

<步长>　::= <非零数字>{<数字>}

<函数调用语句>::=<标识符>‘(’<值参数表>‘)’ **（将两种类型的函数合为一种，便于处理）**

<值参数表>　::=<表达式>{,<表达式>}|<空>

<读语句>　::= scanf ‘(’<标识符>{,<标识符>}‘)’

<写语句>　::=printf ‘(’ <字符串>,<表达式> ‘)’|printf ‘(’ <字符串> ‘)’|printf ‘(’<表达式>‘)’

<返回语句>　::= return[‘(’<表达式>‘)’]

<表达式>　::=[＋|－]<项>{<加法运算符><项>}

<项>　::=<因子>{<乘法运算符><因子>}

<因子>　::=<标识符>|‘(’<表达式>‘)’ | [＋|－]<整数>|<函数调用语句>|[＋|－]<实数>|<字符> **（对应于前面整数和实数的改写，这里的整数和实数也应加正负号）**

注：上文加粗处为文法有改动的地方

词法分析程序

任务：

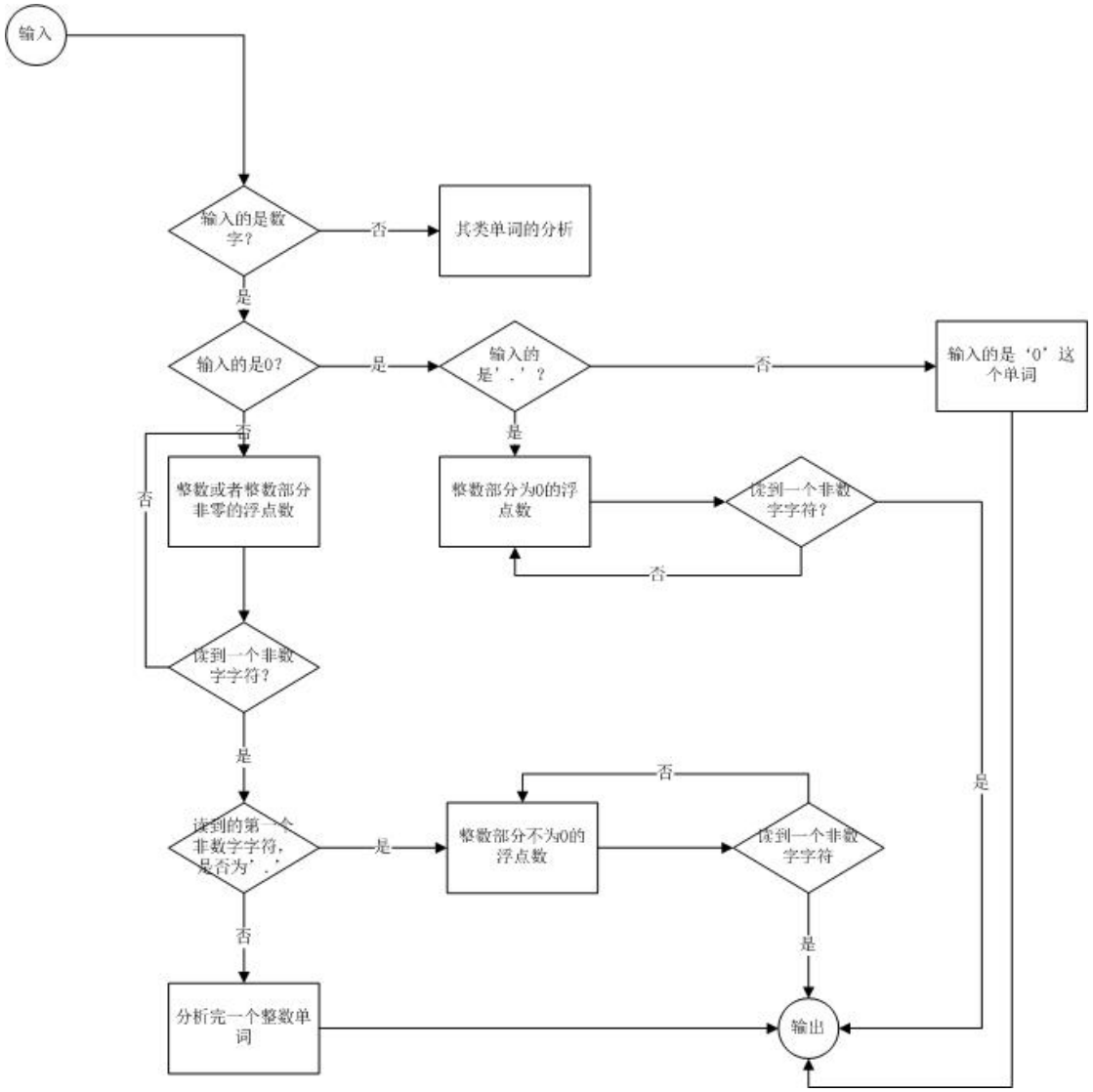
词法分析程序的工作就是要为语法分析程序提供一个个的单词，以便于语法分析程序进行语法的正确性判断。

状态图

状态图的作用主要是帮助我们能更容易地编写出词法分析程序，而且正确性，可靠性相对会比较高。需要注意的是，状态图是根据文法中的一些规则抽象出来的，而究竟那些规则是词法的规则，这就需要我们自己去思考了。

状态图还有一个好处就是能在我们编程的过程中，帮助我们思考问题更全面，什么时候应该加入错误处理，什么时候需要判断是整数还是浮点数，什么时候需要判断是标识符还是保留字，等等一些细节。对于那些思维不是很严谨的同学来说，利用状态图来编写词法分析程序会使程序可靠性很高，而且编写起来也相对容易一些。

例如对于数字的词法分析，我们需要判断是否是整数，是否是浮点数，整数前是否有前导零，浮点数除了 0.x 的情况以外，是否有前导零等等，各种情况，如果我们自己单凭想来解决问题的话，不免困难有些大，这时我们不妨根据文法画出状态图，这样词法分析程序就很好写了。



根据状态图可以方便地写出程序：

```
case ISNUMBER :
    IntNum=c-48;
    cn=fgetc(in);//再读一个，判断是否是'.'
    if (IntNum==0 && cn!='.')
    {
        return  '0'这个单词;
    }
    else
    {
        读入整数或浮点数的整数部分
        if (c=='.')
        {
```

```
                读入的是浮点数的小数部分，直到遇到一个非数字字符后返回这个浮点数
            }
            else
            {
                如果遇到非数字部分，且不是'.'，则返回这个整数
            }
        }
```

词法分析程序返回的单词结构

词法分析程序最终是要向语法分析程序提供单词信息，因此，选择什么样的结构发送给语法分析程序是一个非常重要的问题。

要想设计出一个好的，便于后面程序利用的结构，就需要我们十分清楚词法分析程序的作用。词法分析程序就是要将一个个单词从整个程序中取出来，并将单词的各种信息告诉语法分析程序。类似于这个单词是标识符还是保留字，是数字还是字母，如果是数字的话，它的值又是多少等等。

从语法分析程序的角度来看，我们需要从词法分许程序获得什么呢？首先当然是这个单词的类型，这个单词是保留字，还是标识符，还是整形，还是实型，还是字符类型？这有利于我们立刻能按照文法来判断程序的语法是否错误。因此词法分析程序返回的结构中需要有一个 type 字段保存这些信息。

如果单词是一个整形，语法分析程序希望能立刻知道这个整形的值，而不是再自己通过字符串来转换成值。因此，词法分析程序返回的结构中需要有 intValue 字段来保存整形的值。

类似，我们还需要一个 floatValue 保存浮点数的值，一个 ctr 保存字符值，一个 str 保存字符串的值，一个 name 保存标识符的值。

这里我们发现 str 和 name 字段其实都是字符串，而一个单词不可能同时是字符串和标识符，所以可以将 str 字段和 name 字段合并，就叫做 str，其中保存的可以是标识符的名字或者是字符串。这样，我们便设计出了一个合理的词法分析程序返回的单词结构。

```
struct FeedBack{
    int type;                   //取出词的类型
    char Ctr;                   //若是字符，则存于此
    char Str[STRINGLENGTH];     //若是字符串或者标识符，则存于此
    int IntValue;               //若是整数，则保留与此
    float RealValue;            //若是实数，则保留于此
} WordBack;
```