华北电力大学

课程设计报告

(2021 -- 2022 年度第 2 学期)

名　　称：编译技术课程设计

题　　目：词法分析器设计

自上而下语法分析器设计

简单赋值语句的语法制导翻译程序设计

院　　系：计算机系

班　　级：网络 1902

学　　号：220191090418

学生姓名：穆文翰

指导教师：黄建才

设计周数：1 周

成　　绩：

日期：2022　年 6 月　12 日

《编译技术》课程设计

任　务　书

一、 目的与要求

通过设计、编写和调试词法分析程序，了解词法扫描器的组成结构、不同种类单词的识别方法，掌握由单词的词法规则出发，构建识别单词的状态转换图，从而实现词法扫描器的方法。通过设计、编写和调试语法分析程序，了解语法分析器的组成结构以及对文法的要求，掌握基于简单程序设计语言文法的语法分析程序的实现方法。通过设计、编写和调试语法制导翻译程序，掌握构造语义子程序，实现基于语法结构的语法制导翻译方法。

二、 主要内容

设计并实现一个简单语言的编译程序，包括词法分析、语法分析和语义分析阶段，实现方法参考书中算法描述。

1. 词法分析器的设计与实现

设计并实现词法扫描器，其输入是源程序字符串，输出是二元式（种别编码，单词的属性值）形式的单词。同时需要创建并填写符号表和常数表。

2. 语法分析器的设计与实现

程序设计语言所涉及的大部分语法单位，都可以采用递归下降分析法或预测分析分析法。注意，为了构造不带有回溯的自上而下的语法分析，首先需要通过形式变换将文法转化为 LL(1)文法。

3. 语法制导翻译程序

对语法分析正确的程序在其语法分析的基础上，进行语义翻译工作。每当分析出某语法单位时，就调用对应产生式的语义子程序，完成相应的翻译工作。例如，说明语句的语义动作为填写符号表，执行语句将生成相应的四元式。

三、 进度计划

序号	设计内容	完成时间	备注
1	词法分析器的设计与实现	4 学时	
2	语法分析器的设计与实现	8 学时	
3	语法制导翻译器的实现　撰写实验报告	4 学时	

四、 设计（实验）成果要求

1. 词法分析器：能够接收用户录入的一段源程序，通过词法分析，输出此段程序中所包含的所有单词的编码和属性，以及相应符号表和常数表。
2. 语法分析器：对词法分析结果进行语法分析，能够判断单词串是否某文法的句子，识别其所包含的语法单位，语法正确情况下，在语法分析成分分析结束前，输出当前语法成分的名字。

3. 语义分析器：对输入源语言程序进行语法制导翻译工作。每当分析出某语法单位时，则完成相应的语义分析动作，包括产生中间代码，填写符号表等。

五、 考核方式

课程的考核通过实验平台的作业成绩以及考核成绩和实验报告进行综合评判。其中，平台的作业30 分，考核成绩加实验报告 70 分，考核成绩中词法 20 分，语法 30 分，语义 20。最终成绩采用五分制。实验结束后必须上交实验报告。

学生姓名：

指导教师：黄建才　岳燕

2022 年 5 月 23 日

六、 附录

程序设计语言的文法如下(其中，id 代表标识符，num 代表整型常数)：

<程序> → main(){<程序体>}

<程序体> → <语句组>

<语句组> → <单语句><语句组>|<单语句>

<单语句> → <说明语句> | <赋值语句> | <if 语句> | <while 语句>

<说明语句> → <类型><变量列表>;

<变量列表> → id, <变量列表> | id

<类型> → int | float

<赋值语句> → id = <表达式>;

<if 语句> → if(<布尔表达式>)<语句块>

| if(<布尔表达式>)<语句块>else<语句块>

<while 语句> → while(<布尔表达式>)<语句块>

<语句块> → <单语句> | {<语句组>}

<表达式> → <项>+<表达式> |<项> -<表达式> | <项>

<项> → <因子>*<项> | <因子>/<项> | <因子>

<因子> → id | (<表达式>)|num

<布尔式> → <布尔项>|| <布尔式>| <布尔项>

<布尔项> → <布尔因子>&&<布尔项> | <布尔因子>

<布尔因子> → !<关系式> | <关系式>

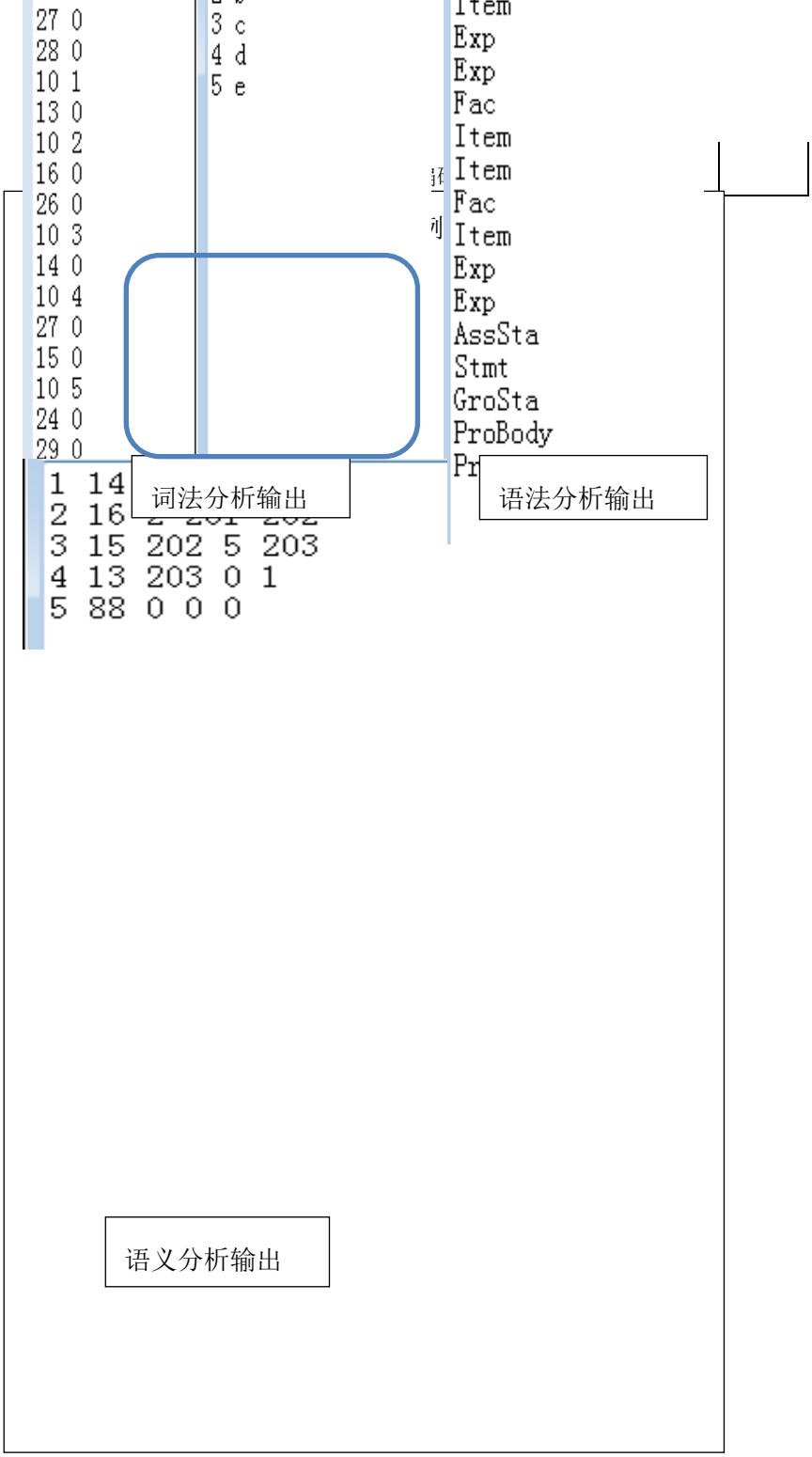
<关系式> → id <关系符> id

<关系符> → < | <= | > | >= | == | !=

单词	编码	单词	编码	
main	1	*	16	
int	2	/	17	
float	3	<	18	
if	4	<=	19	
else	5	>	20	
While	6	>=	21	
||	7	==	22	
&&	8	!=	23	
!	9	;	24	
标识符	10	,	25	
整型常数	11	(	26	
实型常数	12	)	27	
=	13	{	28	
+	14	}	29	
-	15			

中间代码运算符编码	
j>运算的编码与>的编码一致，类推； 无条件跳转编码为 30；	

语法单位	英文名称
<程序>	Program
<程序体>	ProBody
<语句组>	GroSta
<单语句>	Stmt
<说明语句>	DecSta
<变量列表>	ListVar
<类型>	BType
<赋值语句>	AssSta
<if 语句>	IfSta
<while 语句>	WhileSta
<语句块>	Block
<表达式>	Exp
<项>	Item
<因子>	Fac
<布尔式>	BoolExp
<布尔项>	BoolItem
<布尔因子>	BoolFac
<关系式>	RelExp
<关系符>	Rel



输出说明：

1）词法分析输出说明：输出包括两个文件，单词二元式序列文件和符号表文件

2）语法分析输出说明：识别出语法成分且语法正确的情况下，在语法分析成分分析结束前，输出当前语法成分的名字（名字使用上面表格中的英文名称，注意大小写），对于为使文法变为 LL（1）文法而增加的非终结符，无需输出该语法成分的名字。输出内容为各语法成分的英文名称。

3）语义分析输出说明：输出一个文件，文件内容为四元式序列，其中临时变量从 201 开始。

编译技术课程设计报告正文

一、课程设计的目的与要求

1　词法分析器设计的目的与要求

1.1　词法分析器设计的目的

词法分析是编译程序的第一阶段，作为即将编译的源程序和编译器后续阶段的接口，其功能是将输入程序字符串扫描分解，识别出各个单词，输出单词的内部表示。词法分析器的设计目的即为实现上述词法分析的功能，对输入程序进行拆分识别并生成存储单词的 token 表和存储标识符和常数的 symble 表。

1.2　词法分析器设计的要求

能够正确接收并读取用户录入的一段源程序，通过词法分析，输出此段程序中所包含的所有单词的编码和属性，以及相应符号表和常数表。

2　自上而下语法分析器设计的目的与要求

2.1　语法分析器设计的目的

语法分析是编译过程的核心部分，其本质是句型分析，设计语法分析器目的是识别一个符号串是否是所设定的文法的句型，它是对推导和语法分析树构造的过程。

2.2　语法分析器设计的要求

利用自上而下语法分析思想以及递归下降法对词法分析结果进行语法分析，能够判断单词串是否某文法的句子，识别其所包含的语法单位，语法正确情况下，在语法分析成分分析结束前，输出当前语法成分的名字。

3　语义分析器设计的目的和要求

3.1　语义分析器设计的目的

编译程序的目的是将源程序翻译为语义相同的目标程序，设计语法分析器可以使编译程序更加便于设计和实现，能够提高代码的执行效率。

3.2　语义分析器设计的要求

对输入源语言程序进行语法制导翻译工作。每当分析出某语法单位时，则完成相应的语义分析动作，包括产生中间代码，填写符号表等。在本次实验中为主要针对简单算术表达式构成的赋值语句的翻译。

二、课程设计正文

1　词法分析器设计

1.1. 数据结构

1) 结构体类型 word：用于表示从源程序中读入的单词，其中包含 name（string 类型，代表单词的值），code（int 类型，代表种别编码）两个属性；

2) 结构体类型 wordsymble：用于代表 symble 表中每一项的数据结构，其中包含 seq（int 类型，表示在 symble 表中的序号），value（string 类型，存储常数或者标识符的实际值）；

3) 四类匹配表：word 类型的数字，分别存储了关键字、界符、单字符运算符[1]以及双字符运算符与其种别编码的匹配关系[2]；

4) 标识符表和常数表：均为 wordsymble 类型的数组，共同构成了 symble 表的结构，为了程序的编写的简便，将这两部分分开。

1.2 流程图

本次词法分析的设计思想是逐字符的从源程序中读入并拼接，对于每个单词的首字符判断其类型，是否为字母和数字，若是字母，则调用判断是否为关键字和标识符的分析函数；若是数字，则调用判断是否为常数的分析程序；此外读入的字符还可能为单字符运算符或多字符运算符的一部分以及界符，在整体流程图中将其表示为其他单词的分析程序。整体分析流程图见图 1[3]。

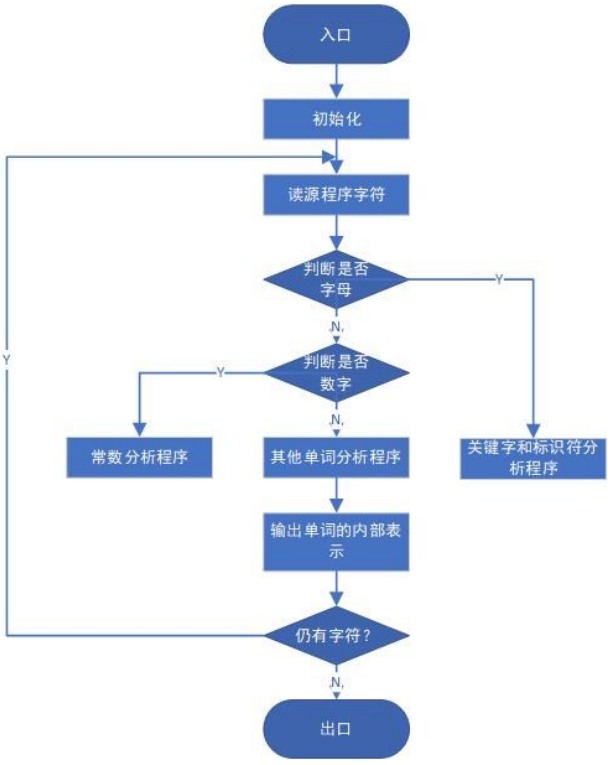


图 1 词法分析的总体流程图

[1] 为了简化程序的编写、便于最长匹配这里的设计包含了部分构成双字符运算符的非运算符部分。

[2] 这一部分借助了映射的思想，由于测试平台的限制没有直接采用键值对方式。

[3] 这一部分设计流程参照了课本 P34-P35 转换图以及流程图，程序设计严格按照了此流程。

1.3 词法分析器的实现

1) 数据结构的实现

对于 1.1 节提到的两个结构体类型的数据结构，这里做简单的声明：

```
typedef struct
{
    string name;//单词名称
    int code;//对应编码
} word;
```

```
typedef struct //symble 表数据结构
{
    int seq;//序号
    string value;//标识符值
} wordsymble;
```

四类匹配表的实现方式如下：

```
word keywords[keywordnum]={{"main",1},{"int",2},{"float",3},{"if",4},{"else",5},{"while",6}};
word delimiters[delimiternum]={{";",24},{",",25},{"(",26},{")",27},{"{",28},{"}",29}};
word unioperators[unioperatornum]={{"!",9},{"=",13},{"+",14},{"-",15},{"*",16},{"/",17},{"<",18},{">",20}};
word binoperators[binoperatornum]={{"||",7},{"&&",8},{"<=",19},{">=",21},{"==",22},{"!=",23}};
```

2) 四类首字符的判断函数（字母、数字、点号、字符 e[4]）

利用 if 语句结构，对当前读入的字符进行判断，判断给字符是否为数字或字母或点号、对数运算符 e，前两者在首字符判断时调用，后两者在实数判断时调用。四种函数均为 bool 类型，若匹配成果则返回 true，无法匹配则返回 false；

a) 判断字符是否为字母：

```
bool isletter(char ch)
{
    if((ch>='a' && ch<='z')||(ch>='A' && ch<='Z'))
    {
        return true;
    }
    return false;
}
```

[4] 这一部分的点号以及字符 e 的判断主要目的是为了匹配浮点数的两种表示方式。

b) 判断字符是否为数字

```
bool isdigit(char ch)
{
    if(ch>='0' && ch<='9')
    {
        return true;
    }
    return false;
}
```

c) 判断字符是否为小数点

```
bool isdot(char ch)
{
    if(ch=='.')
    {
        return true;
    }
    return false;
}
```

d) 判断字符是否为 e（或 E）

```
bool isletterE(char ch)
{
    if (ch=='e'||ch=='E')
    {
        return true;
    }
    return false;
}
```

3) 关键字判断函数 iskeyword、界符判断函数 isdelimiter、单字符运算符判断函数 isunioperator 以及双字符判断函数 isbinoperator

在 1.1 节数据结构部分提到，我们使用类似映射的关系存储了种别编码与单词的对应关系，通过循环结构查找存储对应关系的数组，若匹配，则返回该单词的种别编码，否则返回 0 表示未查到，以供调用函数做进一步的错误处理；

这四个函数的操作过程和匹配方式都极为相似，因此在此次仅用其中一个作为代表来展示具体的程序编写思路。

```
int isdelimiter(char ch)
{
    string tempstring="";
    tempstring.append(1,ch);
    for (int i=0; i<delimiternum; i++)
    {
        if(tempstring == delimiters[i].name)
        {
            return delimiters[i].code;
         }
     }
     return 0;
}
```

4) 查找符号表中是否存在的判断函数 judgeexist

通过传入表示标识符或实数表、带判断单词以及全局长度变量（表示当前已保存了的标识符或实数数量），利用循环结构，查找当前的符号表中是否以及存在该实数或者标识符，若存在，返回其在符号表中的位置[5]，否则返回-1；

```
int judgeexist(wordsymble s[],string str,int len)
{
    for (int i = 0; i < len; i++)
    {
        if(s[i].value == str)
        {
            return i;
        }
    }
     return -1;
}
```

5) 标识符或关键字匹配函数

当判断了读入的首字符为字母时，调用此函数。

在函数中，用临时变量存储暂时独到的单词（初始时为首字符），然后不断读入下一个字符，判断其是否为字母或数字，直到读入的下一个字符并非这二者，完成对完整单词的读入。

随后，判断该单词是否为关键词，调用前述 iskeyword 函数，若是，则表示该单词为关

[5] 在程序中，应为在数组中的位置，从 0 开始。

键词，则输出单词的种别编码及其属性值（0）到 token 文件中。

若不是，则说明该单词为标识符，首先查找符号表中是否以及存在此单词，调用 judgeexist 函数，传入标识符表、该单词以及 identnum（表示已经存入的标识符数量）。判断其返回值，若不为-1，说明已经存在，则利用返回的位置信息读取该标识符的信息，输出到 token 文件中。

若返回值为-1，说明此标识符第一次出现，将 identnum+1 一同与该单词的值存入标识符表中。并将相关信息输出到 token 和 symble 文件中。

```
1. void judge_iden_or_key(char &ch)
2. //ch 为当前读入的首字符，判断为字母执行关键字和标识符判别程序
3. {
4.     string tempstring="";
5.     tempstring.append(1,ch);
6.     infile >> ch;
7.     while(isdigit(ch)||isletter(ch))
8.     {
9.         tempstring.append(1,ch);
10.         infile >> ch;
11.     }
12.     int code=iskeyword(tempstring);
13.     if(code != 0)
14.     {
15.         ofile << code <<" " << 0 <<endl;
16.     }
17.     else
18.     {
19.         int k=judgeexist(identifier,tempstring,identnum);
20.         if (k==-1)//标识符表里尚不存在此标识符
21.         {
22.             identifier[identnum].seq=identnum+1;
23.             identifier[identnum].value=tempstring;
24.             ofile << 10 <<" " << identnum+1 << endl;
25.             ofile_2 << identnum+1 << " " << tempstring << endl;
26.             identnum++;
27.         }
28.         else//标识符已经存在，只需输出 token 表即可
29.         {
30.             ofile << 10 <<" " << identifier[k].seq << endl;
31.         }
32.     }
33. }
```

6)　整数或实数匹配函数

当判断了读入的首字符为数字时，调用此函数。

若读入的下一个字符为数字，则将该字符拼接到临时字符串上，循环测过程；

直到读入了非数字字符：点号或科学计数法标志 E（e），进行如下操作。

若读入的字符为 E（e），下一个读入的字符满足规则的可以为-号或+号，也可以直接为数字，当读入数字时，进行循环读入并拼接，直到读入的字符不是数字为止，单词的完整读入过程结束；

若读入的字符为点号，读入下一个字符，当读入数字时，进行循环读入并拼接，直到读入的字符不是数字为止，单词的完整读入过程结束；

这个过程中，利用 bool 变量表示是否为实数。

接下来调用 judgeexist 函数判断是否在符号表中存在。后续过程与 5) 标识符或关键字匹配函数对应部分类似，这里不再做过多赘述。唯一不同的地方是，这里在输出 token 文件时，需要利用前述 bool 类型变量判断是否为浮点数，以输出对应的种别编码。

```
1. void judge_int_or_float(char &ch)//首字符已经读入为数字
2. {
3.     bool dot=false;
4.     bool letterE=false;
5.     string tempstring="";
6.     tempstring.append(1,ch);
7.     infile >> ch;
8.     while(isdigit(ch))
9.     {
10.         tempstring.append(1,ch);
11.         infile >> ch;
12.     }
13.     if(isletterE(ch))
14.     {
15.         letterE=true;
16.         tempstring.append(1,ch);
17.         infile >> ch;
18.         if(ch== '-' || ch=='+')
19.         {
20.             tempstring.append(1,ch);
21.             infile >> ch;
22.         }
23.         if(isdigit(ch))
24.         {
25.             tempstring.append(1,ch);
```

```
            infile >> ch;
            while(isdigit(ch))
            {
                tempstring.append(1,ch);
                infile >> ch;
            }
        }
    }
    else if(isdot(ch))
    {
        dot=true;
        tempstring.append(1,ch);
        infile >> ch;
        while(isdigit(ch))
        {
            tempstring.append(1,ch);
            infile >> ch;
        }
    }
    int k=judgeexist(constant,tempstring,constnum);
    }
     /*
      ……
       查完符号表后的输出过程与在判断标识符和或者关键词的过程中相似，此处省略。根据 dot、letterE 的值判断是否为为整型常数或实型常数，分别输出对应的编码和值
     */
}
```

7) 运算符匹配函数

当读入首字符属于运算符或者运算符一部分时，调用此函数

为了实现最长匹配运算符，读入下一个字符进行拼接，调用双字符判断函数 isbinoperator，若匹配成功，则输出相关结果。若不成功，调用单字符运算符判断函数 isunioperator，判断首字符是否为运算符，若成功输出相关结果。不成功则需转入错误处理。

```
void judgeoperator(char &ch)//以及识别到 ch 为单字符运算符
{
    string tempstring="";
    tempstring.append(1,ch);
    if(ch!='|'&& ch!='&')
    {
        char ch_2=ch;
        infile >> ch;
```

```
        if(ch == '=')
        {
            tempstring.append(1,ch);
            int code = isbinoperator(tempstring);
            if(code!=0)//是一个双字符运算符
            {
                ofile << code << " " << 0 << endl;
            }
            infile >> ch;
        }
        else
        {
            ofile << isunioperator(ch_2) << " " << 0 << endl;
        }
    }
    else
    {
        char ch_2;
        ch_2 = ch;
        infile >> ch;
        tempstring.append(1,ch);
        if(isbinoperator(tempstring)!=0)
        {
            ofile << isbinoperator(tempstring) << " " << 0 << en
dl;
            infile >> ch;
        }
        else
        {
            ofile<< isunioperator(ch_2) << " " << 0 << endl;//这
里转换格式也可以继续用 append
        }
    }
}
```

8) 界符匹配函数

调用界符判断函数 isdelimiter 即可。

2 词法分析器设计

2.1 数据结构

1) 结构体类型 token：用于表示词法分析的输出结果 token 表中的单词，其中包含 name（string 类型，代表单词的值），code（int 类型，代表种别编码）两个属性；

2) 结构体类型 symble：用于存储从 symble 文件中读取的内容，包括 seq（int 类型，表示在 symble 表中的序列号），name（string 类型，存储实际值）两个属性。

3) 变长数组（vector）类型 token1、symble1：存储从 token 表和 symble 中读取到的全部内容。

2.2 流程图

由于语法分析使用的分析方法为递归下降法，需要对每一个非终结符构造一个函数，然后根据文法中对应产生式调用其中非终结符对应的函数过程，这样的函数过程有 26 个，限于篇幅，此处不再展示完整的语法分析流程，仅将其进行抽象概括表示，如图 2 所示。

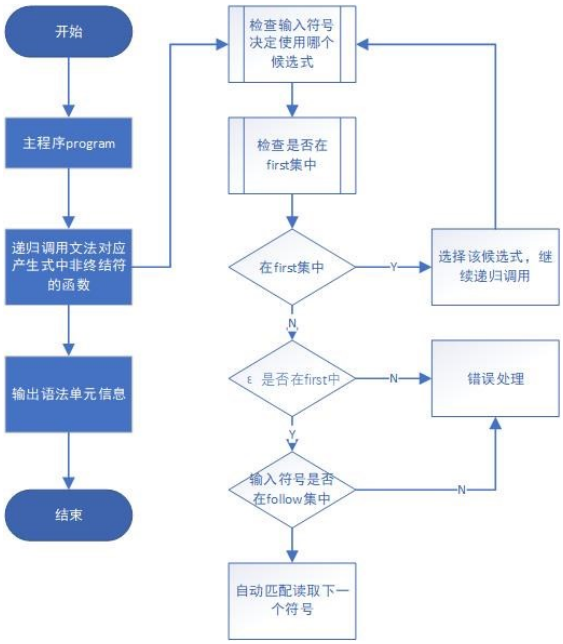


图 2 语法分析的流程（概况）

2.3 语法分析器的实现

1) 提左因子，消除左递归转换为 LL（1）型文法

由于我们选择的是递归下降法进行语法分析，所以首先第一步将题目所给的文法转换成为 LL（1）文法，转换过程与转换后的文法请见附录 1；

2) 计算 first 集合与 follow 集合，在后面编写函数时使用

计算出 first 集和 follow 集合才能进行下一步的分析构造，对于 first 和 follow 集的提取结果请见附录 2；

3) 数据结构的实现

对于 2.1 节提到的两个结构体类型的数据结构，这里做简单的声明：

```
1. typedef struct {
```

```
2.     int code;//种别编码
3.     int value;//属性
4. }token;
```

```
1. typedef struct
2. {
3.     int seq;
4.     string name;
5. }symble;
```

4) 根据上述 LL（1）型文法写出每个非终结符的递归下降函数

```
1. void Program();
2. void ProBody();
3. void GroSta();
4. void Stmt();
5. void Sg();
6. void DecSta();
7. void Asssta();
8. void Ifsta();
9. void WhileSta();
10.void BType();
11.void ListVar();
12.void V();
13.void Exp();
14.void BoolExp();
15.void Block();
16.void Else();
17.void Item();
18.void E();
19.void Fac();
20.void I();
21.void BoolItem();
22.void Be();
23.void BoolFac();
24.void Bi();
25.void RelExp();
26.void Rel();
```

5) 递归下降函数的设计思路

用程序的方式模拟产生式语言产生的过程，或者说是语法树的生成/遍历过程，分析程序由一组递归子程序组成，每个子程序都对应一个非终结符，分析过程从调用文法的开始符号的子程序出发，直到所有的非终结符都展开为终结符并得到匹配为止，表明分析成功，否则，表面由语法错误。

对于每个终结符及其函数，检查输入符号，根据 first 集选择调用哪个候选式对应的函数。也可能 first 集中不存在该符号，但存在空串，则判断是否在 follow 集当中，若在，空串将被使用。在当前的函数下不做其他的操作。

同时，本题目的语法树遍历过程应当是自下而上，因此应当是使用后序遍历的思想（其实也是一个深度优先遍历的方式），在整个函数的末尾才最后输出该节点的语法信息。

由于程序书写了 26 个非终结符对应的函数，每个函数都有较强的相似性，因此我们在这里选择其中的一部分来表现递归下降分析法思想。

a) 主函数入口：

```
1. void Program()
2. {
3.     // Gramnames.push("Program");
4.     if(token1[tokennum].code == 1)
5.     {
6.         getnexttoken();
7.         if(token1[tokennum].code == 26)
8.         {
9.             getnexttoken();
10.             if(token1[tokennum].code == 27)
11.             {
12.                 getnexttoken();
13.                 if(token1[tokennum].code == 28)
14.                 {
15.                     getnexttoken();
16.                     ProBody();
17.                     if(token1[tokennum].code == 29)
18.                     {
19.                         success=true;
20.                     }
21.                 }
22.                 else
23.                 {
24.                     error();
25.                 }
26.             }
27.             else
```

```
            {
                error();
            }
        }
        else
        {
            error();
        }
    }
    else
    {
        error();//预留错误接口
        // Gramnames.pop();
    }
    ofile << "Program" << endl;
}
```

b) 以语句组对应的文法为例，展示每个递归函数的大致过程以及因变换文法而新产生的语法单元的操作过程（主要是自动匹配）

```
void ProBody()
{
    // Gramnames.push("ProBody");
    GroSta();
    ofile << "ProBody" << endl;
}

void GroSta()
{
    int tempcode=token1[tokennum].code;
    // Gramnames.push("GroSta");
    if(tempcode == 2||tempcode == 3||tempcode ==10||tempcode ==4
||tempcode ==6)
    {
        Stmt();
        Sg();
    }
    ofile << "GroSta" << endl;
}
```

3. 语义分析器设计

3.1 数据结构

1) 结构体类型 quadruple，分别含有四个 int 类型的属性 op,arg1,arg2,result，对应于四元式的四个内容。

2) 结构体类型 node：用于表示非终结符节点。设计为结构体类型的目的是，本题中仅涉及到简单运算符构成的赋值语句的翻译过程，对于一个非终结符节点仅需要其 place 属性。但如果要使语法分析器同时能够执行对说明语句、控制语句等的翻译，那么则需要对节点的属性进行扩充，如 quad、type 属性等。使用结构体类型时仅需对其定义部分进行扩充即可，便于语义分析器功能的扩展。

3.2 流程图

语义分析的实现过程是在语法分析的过程是添加了相关属性的计算，由于语法分析的子程序颇多，因此我们主要展示的是每个子程序所进行的大致流程，如图 3 所示。

图 3 语义分析中每个子程序的执行过程

3.3 语法分析器的实现

1) 数据结构的实现

对于 3.1 节提到的两个结构体类型的数据结构，这里做简单的声明：

```
typedef struct
{
    int op;
    int arg1;
    int arg2;
    int result;
}quadruple;//四元式结构体
```

```
typedef struct
{
    int place;
}node;//非终结符节点[6]
```

2) 辅助函数的实现

对于赋值语句的翻译，需要两个关键的过程，即 emit（产生四元式）和 newtemp（产生临时变量）。

这两个函数的设计如下所示：

```
void emit(int op, int arg1, int arg2, int result)
{
    quadruple q;
    q.op=op;
    q.arg1=arg1;
    q.arg2=arg2;
    q.result=result;
    quadtable.push_back(q);
}
```

```
int newtemp()
{
    tempvarnum++;
    return tempvarnum+200;//临时变量编码从 201 开始
}
```

3) 赋值语句语义分析的总体思路

语义分析的实现，是在语法分析的框架上搭建起来的，个人的理解是，在遍历语法树的过程增加中计算相关的属性。

在语法分析的部分，无需进行属性的计算，因此我们将函数的返回值类型设为了 void，在语义分析的部分，需要返回一些必须的属性值，因此用 node 类型来作为返回值类型，并在每个函数被调用的过程中生成一个新的 node 以表示当前的节点，并作为返回值返回。

要注意的是，在本题目中，所需要调用 emit 的过程就是执行计算的过程，对于简单赋值语句而言，就是出现+、-、*、/，四种运算的时候以及在最后程序执行完毕的末尾。

语义分析建立在语法分析的基础上，对于赋值语句的语义分析在程序中仅做了与赋值语句相关的改动，其余部分和语法分析基本相同。在此次仅展示做出修改的部分，同样以其中

[6] 对于简单赋值语句的翻译不涉及其他属性的计算过程，因此本报告中不显示对其他属性的定义，必要时可以快速拓展。

的一些函数代表全部修改。

```
node Exp()
{
    // Gramnames.push("Exp");
    node s;
    s=Item();
    s=E(s);
    // ofile << "Exp" << endl;
    return s;

}

node E(node e)
{
    node s=e;
    if(token1[tokennum].code == 14)
    {
        getnexttoken();
        node i=Exp();
        s.place=newtemp();
        emit(14,e.place,i.place,s.place);
    }
    else if(token1[tokennum].code == 15)
    {
        getnexttoken();
        node i=Exp();
        s.place=newtemp();
        emit(15,e.place,i.place,s.place);
    }
    else if(token1[tokennum].code == 27 || token1[tokennum].code
 == 24)
    {
        //空
    }
    return s;
}
```

三、课程设计总结或结论

本次编译技术实验说实话难度还是比较大的，虽然说对于课本上的知识掌握还算比牢固，但是一当把理论知识转换为实践操作时，却又显得不那么如意。

首先是词法分析，这是第一个实验，还算比较简单，通过简单的分析与相关资料的查询，很快就完成了第一个实验。但是一当我开始第二个实验的时候，我便开始显得手足无措，无从下手了。

在语法分析的时候我就碰到了一个很大的问题，我到底该选择什么样的语法分析方法去实现它，选择预测分析法的话，由于实验所给文法较长，那么就需要构造一张比较大的预测分析表，所花费的时间成本较大。所以在思考一段时间之后，再加上同学的帮助，我选择使用递归下降法去实现它。好在文法仅存在一些左递归和回溯，没有间接左递归，但是消除之后的文法还是比较长的，这就意味着后面的非终结符函数会有很多。

在消除回溯和左递归的过程以及求 first 和 follow 集的过程中，需要极大的耐心和细心，仅 first 和 follow 集的求解过程中，我就前后经历了五次，总是少看了一个小标点或者是看错了文法，这样的教训告诉我，凡事不能急于求成，应当保持耐心和细心。

之后通过仔细检查文法，确认无误之后，我开始编写每一个非终结符的递归函数，由于文法的长度，所以这个过程是十分漫长而枯燥的，并且我不能够一次性将所有语句完成，我必须使用增量的方式去编写，在编写完一部分之后就进行测试，确保无误后再继续编写下一部分。在完成所有的编写之后我还进行了一个全面的测试，但是可能不知道是编写时哪里出了问题，测试没有通过，所以后来花了很长的时间去一步一步调试，最终才解决了问题。

最后就是语义分析，由于本次实验所给的文法是类 C 语言的文法，而我们平时课程中所学习到的都是 L 语言的文法，所以在何处生成四元式，以及在何处返回什么样的值都令我思考了很长时间，不过好在最终解决了问题。

完成本次实验的过程中，面临期末、各科的课设以及夏令营的填报、考核，熬了不少的夜，虽有疲惫，但是收获颇丰，在心态上得到了充足的历练，同时我也相信这是对自己一种能力的提升与突破，也让自己对于编译技术这门又重要却又抽象的课程有了更深刻的印象。

最后，非常感谢黄建才老师在实验过程中对我们不断地鞭策、提醒与鼓励，在老师的指导和帮助下，我们才能得以顺利的完成此次实验。我对您的付出表示崇高的敬意和感激！

四、参考文献

[1]鲁斌，李继荣，黄建才，等. 编译原理与实践[M]. 2020 年 1 月. 北京市:北京邮电大学出版社，2020.

[2]陈意云，张昱. 编译原理[M]. 2020 年 12 月版. 北京市:高等教育出版社，2014 年 9 月

附录 1：消除递归和回溯的文法

<程序> Program <程序体> ProBody <语句组> GroSta <单语句> Stmt
<说明语句> DecSta <变量列表> ListVar <类型> BType
<赋值语句> AssSta <if 语句> IfSta <while 语句> WhileSta <语句块> Block
<表达式> Exp <项> Item
<因子> Fac <布尔式> BoolExp <布尔项> BoolItem
<布尔因子> BoolFac <关系式> RelExp <关系符> Rel

消除左递归和回溯的文法：
（注意空格不代表任何含义只是为了容易辨识相邻终结符或非终结符）

程序：Program → main(){ProBody}
程序体：ProBody → GroSta

//语句组 GroSta → Stmt GroSta | Stmt :消除回溯
GroSta → Stmt Sg
Sg→GroSta | ε

单语句：Stmt → DecSta | AssSta | IfSta | WhileSta

说明语句：
DecSta → BType ListVar ;

//变量列表：ListVar → id , ListVar | id :消除回溯
ListVar → id V
V → , ListVar | ε

类型： BType → int | float

赋值语句：
AssSta → id = Exp ;

If 语句：
//IfSta → if (BoolExp)Block | if (BoolExp)Block else Block :消除回溯
IfSta → if (BoolExp)Block Else
Else → else Block | ε

While 语句：
WhileSta → while(BoolExp)Block

语句块： Block → Stmt | { GroSta }

//表达式： Exp → Item+Exp | Item-Exp | Item 消除回溯
Exp → Item E
E → +Exp | -Exp | ε

<程序> Program <程序体> ProBody <语句组> GroSta <单语句> Stmt
<说明语句> DecSta <变量列表> ListVar <类型> BType
<赋值语句> AssSta <if 语句> IfSta <while 语句> WhileSta <语句块> Block
<表达式> Exp <项> Item

<因子> Fac　<布尔式> BoolExp　<布尔项> BoolItem
<布尔因子> BoolFac　<关系式> RelExp <关系符> Rel

//项:　Item → Fac * Item | Fac / Item | Fac 消除回溯
Item → Fac I
I → * Item | / Item | ε

因子:
Fac → id | (Exp) | num

//布尔式: BoolExp → BoolItem || BoolExp | BoolItem 消除回溯
BoolExp → BoolItem Be
Be → || BoolExp | ε

//布尔项 BoolItem → BoolFac && BoolItem | BoolFac 消除回溯
BoolItem → BoolFac Bi
Bi → &&BoolItem | ε

布尔因子:
BoolFac → ! RelExp | RelExp

关系式:
RelExp → id Rel id

关系符:
Rel → < | <= | > | >= | == | !=

附录 2：first 集和 follow 集

First 集:
//Program:　main
ProBody　int float id if while
GroSta　int float id if while
Sg　int float id if while ε
Stmt　int float id if while
DecSta　int float
ListVar　id
V　, ε
BType　int float
AssSta　id
IfSta　if
Else　else ε
WhileSta　while

Block	{ int float id if while
Exp	id (num
E	+ - ε
Item	id (num
I	* / ε
Fac	id (num
BoolExp	! id
Be	|| ε
BoolItem	! id
Bi	ε &&
BoolFac	! id
RelExp	id
Rel	< <= > >= == !=