实验七 对算数表达式构造递归下降翻译器

一、 实验目的

1. 对算数表达式做递归下降分析，同时将其翻译为中间代码

二、 实验要求

1. 对实验四的程序进行升级改造，使得程序对于输入的任意一个算术表达式，在对其做递归下降分析的同时，生成等价的中间代码，一遍完成
2. 基础文法同实验四
3. 语法分析沿用实验四的程序框架，去掉语法树的部分
4. 语义处理时生成四元式序列
5. 一遍处理要求：
    a) 语法分析走到哪里的时候应该执行语义动作
    b) 语义动作应该怎么做
6. 为简化问题，不考虑输入有误的情况，不考虑语义检查

三、 实验设计

1. 代码复用

a) 首先本次实验沿用前面实验设计的词法分析模块，将输入的表达式切分为 token 存储在 word_list 中。字符编码表沿

用实验三中的编码表：

表 1 字符编码表

单词符号	编码	单词符号	编码	单词符号	编码
main	0	-	22	<	30
if	1	*	23	<=	31
else	2	/	24	(	32
while	3	=	25	)	33
int	4	==	26	{	34
id	10	!=	27	}	35
num	20	>	28	;	36
+	21	>=	29	,	37

b) 本次实验的文法与实验四相同，则将其改造为消除递归的无二义性文法的方法相同，First 集和 Follow 集相同。

```
<Expr> → <Term> <Expr1>
<Expr1> → <AddOp> <Term> <Expr1> | empty
<Term> → <Factor> <Term1>
<Term1> → <MulOp> <Factor> <Term1> | empty
<Factor> → id |number | ( <Expr> )
<AddOp> → + | -
<MulOp> → * | /
```

图 1 基础文法

```
将文法改写：
E -> T E1
E1 -> + T E1 | - T E1 | C
T -> F T1
T1 -> * F T1 | / F T1 | C
F -> ID | NUMBER | (E)
```

图 2 文法改造

```
FIRST(E) = {ID,NUMBER, ( }
FIRST(E1) = {+,-,C }
FIRST(T) = {ID,NUMBER, ( }
FIRST(T1) = {*,/,C}
FIRST(F) = {ID,NUMBER,( }

FOLLOW(E) = {#, )}
FOLLOW(E1) = {#, )}
FOLLOW(T) = {#, ), +, -}
FOLLOW(T1) = {#, ), +, -}
FOLLOW(F) = {#, ), *, /, +, -}
```

图 3 First 集和 Follow 集的求解

2. 递归下降翻译器的构造原理

思想：对递归下降的语法分析器进行扩展——在适当的位置加入语义子程序，使之能够实现翻译方案。

方法：为每一个非终结符 A 构造函数 A：

- 函数 A 的形式参数是非终结符 A 的继承属性
- 函数 A 的返回值是非终结符 A 的综合属性的集合
- 在 A 的函数体中，用局部变量来保存所有的中间值

当处理 A 的右部符号串时：

- 对于带有综合属性 x 的终结符 X，把 x 的值存入为 X.x 设置的变量中，然后产生一个匹配 X 的调用，并继续输入
- 对于每个非终结符 B，产生一个右部带有函数调用的赋值语句 c=B(b1,b2…bn),其中 b1,b2…bn 是 B 的继承属性设置的变量，c 是为 B 的综合属性设置的变量
- 对于语义动作，把动作的代码抄入语法分析器中，并把对属性的引用改为对相应变量的引用

3. 一遍扫描的处理方法

指的是在语法分析的同时计算属性值，处理完一个语法单位就执行

与之相关的语义处理。计算语义规则，完成有关语义分析和代码生成动作的时机：

- **自上而下分析中一个产生式匹配输入串成功时**
- 自下而上分析中一个产生式被用于进行规约时

4. 四元式中间代码

一个四元式是一个带有四个域的记录结构：op,arg1,arg2,result。它实际上就是一条三地址的指令，用四元式表达更为清晰。

```
op arg1 arg2 result
```

图 4 四元式的格式

5. 算数表达式构造递归下降翻译器的基本思路

a) 这里还是先考虑一下利用语法树进行翻译的情况：

对于生成的语法树，每个结点表示一个非终结符或者终结符，在结点中存储该符号的综合属性和继承属性，生成好语法树后，**对于每个非终结符符号多次重复计算所有能够计算的继承属性，最后计算所有能够计算的综合属性**。这里存在多次遍历的问题，效率不高，而对于算数表达式来说可以将其改造为 L-属性文法，可以利用一遍扫描处理的方法，在语法分析的同时生成中间代码。

b) 然后是一遍处理的情况：

首先我们可以根据《编译原理》课程中所学的内容，写出算术表达式对应的语义动作、属性计算的方法，然后按照上述的一遍处理的基本思路，对于实验四中不含语法树生成的算数表达式的各非终结符函数进行改造，使其适用于递归下降翻译。

图 5 加入语义动作后的文法

实现时需要注意以下几点：

- 标识符在此实验中不进行取值操作。在后续包含赋值语句时，将其所对应的值在符号表记录，并通过 getValue 进行取值
- 利用 genCode 函数进行四元式的生成
- 由于生成的是四元式，所以需要中间变量 res 存储结果，并将结果进行传递
- 这里的四元式有两种表现形式：一种是针对算数表达式的将计算的值作为 res，则此时需要将 string 类型的 token 转换为可计算的整型；另一种是模拟汇编中利用寄存器存储结果，利用 getRx 自动生成寄存器的编号。两种方法在后面的实验中都有实现。

四、 重要代码分析

这里对重要的代码进行简单分析。

1. 以 E1 为例在递归下降中加入语义动作(两种方式)

首先我们得到不含语义动作的递归下降的 E1 函数：

```
void E1()//E1 -> +TE1 | -TE1 | C
{
    if(word_list[index].s=="+")
    {
        index++;
        T();
        E1();
    }
    else if(word_list[index].s=="-")
    {
        index++;
        T();
        E1();
    }
    else if(word_list[index].s==")"||word_list[index].s=="#")
        return;
    else
    {
        error();
    }
}
```

图 6 不含语义动作的递归下降分析

然后利用第一种算出结果的方法来生成四元式的思路来加入语义动作：

```
int E1(int x)
{
    if(word_list[index].s=="+")
    {
        index++;
        int y = T();//E1.i = T.s
        int res = x+y;//进行计算
        genCode("+", x, y, res);//四元式生成
        return E1(res);//这里实现了综合属性的向上传递，以及继承属性向下传递
    }
    else if(word_list[index].s=="-")
    {
        index++;
        int y = T();
        int res = x-y;
        genCode("-", x, y, res);
        return E1(res);
    }
    else if(word_list[index].s==")"||word_list[index].s=="#")
        return x;//E1.s = E1.i
    else
        error(word_list[index].s);//否则报错
    return -1;
}
```

图 7 加入语义动作的递归下降翻译代码

这里以 x 作为 E1 的继承属性，以 y 作为 T 的综合属性，对于加号匹配成功的情况，加法的两个参数分别为 E1 的继承属性和 T 的综合属性，得到的结果 res 需要往后传递给 E1’，继续进行加法；对于空的情况，说明后面没有加法了，则将前面传递的 res(也就是这里的

x)直接 return 回传即可；然后将最终的总和保存在 E1’的综合属性中向上传递给 E1。由于结果可计算，所以 res 可以直接由 x+y 得到。

利用第二种类汇编的方式，用寄存器编号替代 res：

```
string E1(string x)
{
    if(word_list[index].s=="+")
    {
        index++;
        string y = T();
        string res = getRx();//由函数getRx生成寄存器编号存储结果
        genCode("+", x, y, res);
        return E1(res);//将该结果用于后续的计算
    }
    else if(word_list[index].s=="-")
    {
        index++;
        string y = T();
        string res = getRx();
        genCode("-", x, y, res);
        return E1(res);
    }
    else if(word_list[index].s==")"||word_list[index].s=="#")
        return x;
    else
        error(word_list[index].s);
    return "";
}
```

图 8 加入语义动作的递归下降翻译代码

和第一种思路相似，但是参数的类型需要发生变化，同时结果不由 x 和 y 直接产生，而是通过 getRx 进行生成并传递。同时在该种思路下，无需将字符串类型的 id 或 number 转换为整型。

2. 函数 str2int

通过该函数实现 id 和 number 的整型值转换：

```
int str2int(struct word w)
{
    if(w.type==10)//id
    {
        //处理方式为按位相加
        int res = 0;
        for(unsigned int i=0;i<w.s.length();i++)
            res+=(w.s[i]-64);
        return res;
    }
    else if(w.type==20)//num
    {
        return atoi(w.s.c_str());//将string转为char*再转为int
    }
    else
        error(w.s);
    return -1;
}
```

图 9 函数 str2int

- 由于不存在赋值语句，所以标识符的值不确定，这里人为规定将其 ASCII 码值按位相加作为该标识符的整型值。后续加入赋值语句后可以通过查表获取其值。
- 对于 number 常量来说，利用函数 atoi 和方法 c_str 可以将 string 类型的字符串转为整型

3. 函数 getRx

```
//自动生成寄存器编号
int number = 0;//寄存器编号
string getRx()
{
    number++;
    char str[10];
    itoa(number, str, 10);//将编号转换为char*类型
    string temp = str;
    string res = "$T"+temp;//进行内容拼接
    return res;
}
```

图 10 函数 getRx

4. 函数 genCode

```
void genCode(string op, int x, int y, int res)
{
    cout << op << " " << x << " "<< y << " "<< res <<endl;
}
void genCode(string op, string x, string y, string res)
{
    cout << op << " " << x << " "<< y << " "<< res <<endl;
}
```

图 11 函数 genCode

这里针对两种方式编写了不同的 genCode，采用直接输出的方式，如果考虑后续优化的问题，可以用结构体将四元式进行存储。

五、 实验结果

这里对于错误的处理不考虑语义上的处理，仅考虑词法和语法上的错误，对于语法或词法上的错误，将错误的具体字符给出，并终止当前程序。在后续优化中会考虑语义错误的处理（比如不能除 0 等）

以下测试通过两种方法进行分别验证：

测试用例 1：纯常量进行运算

测试用例 2：标识符和常量混合运算

六、 实验总结

1. 学习了递归下降翻译器的构造原理
2. 对算数表达式进行了中间代码四元式序列的生成进行编码实现，对四元式的产生和形式表达进行了讨论和分析
3. 语义翻译是在语法分析的基础上进行的，需要瞻前顾后：利用前面写好的代码进行复用，减少工作量；需要考虑后面整个程序的中间代码翻译的实现和设计，使得前后工作能够衔接

七、 附录