代码翻译

周闯

320170922001

2018计算机二班

实验八 - 对算术表达式构造递归下降翻译器

实现代码见文件夹3.1

基本文法：

本实验使用到的文法与实验5一致，这里将其文法和First集，Follow集再次列出。

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

给定的文法是无二义性的，且没有左递归。因此，可以对其进行直接分析；

First集&&Follow集

根据给定的文法，对每个非终结符构造First集和Follow集，结果如下所示。并且可以看出对于文法的任一非终结符，其规则右部的多个选择所推导的First集与非终结符的Follow集（当存在A→ empty）的情况下不相交。

	First	Follow
Expr	id，number，(	)，；
Expr1	+，-	)，；
Term	id，number，(	+，，)，；
Term1	*，/	+，，)，；
Factor	id，number，(	*，/，+，，)，；
AddOp	+，-	id，number，(
MulOp	*，/	id，number，(

属性文法

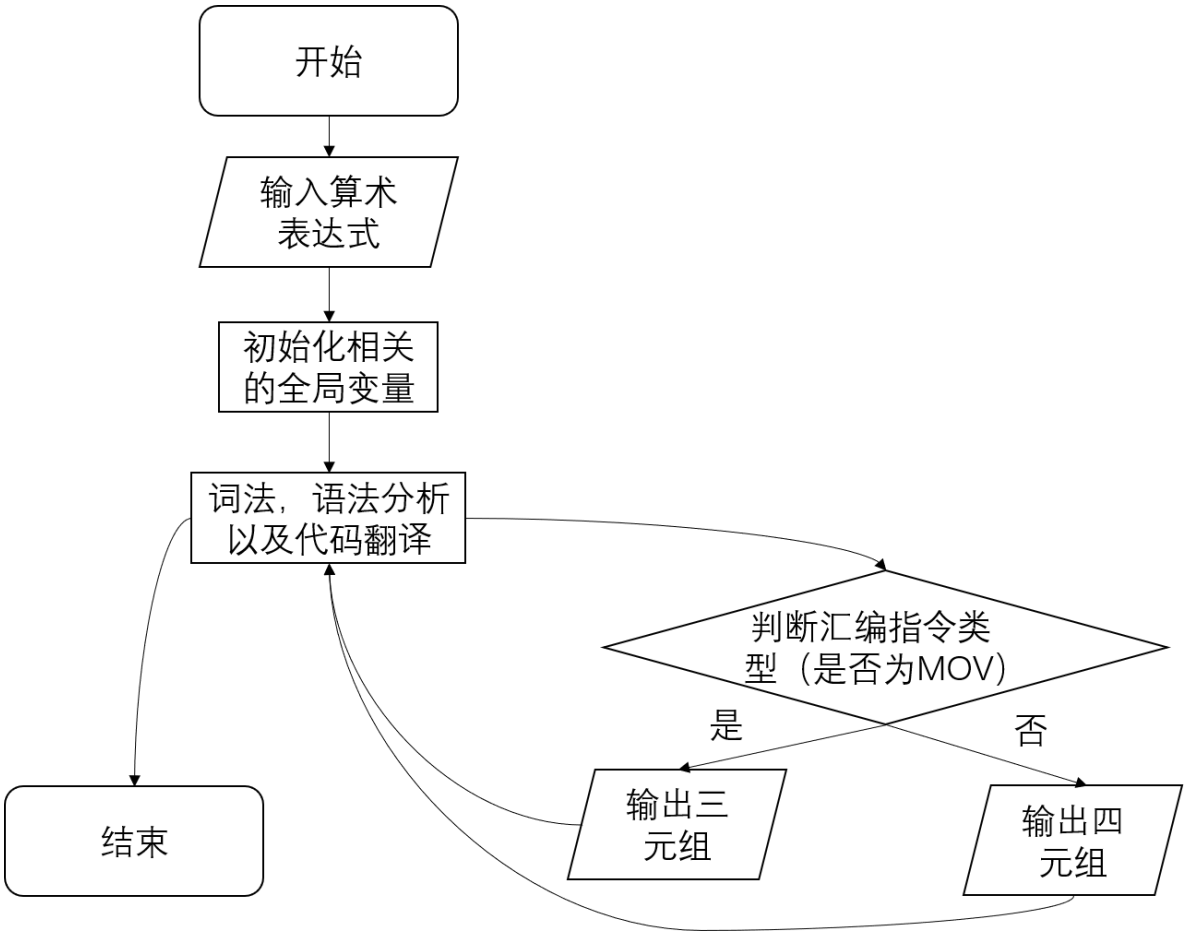
属性文法是在上下文无关文法的基础上为每个文法符号（终结符或非终结符）配备若干个相关的 值（称为 属性 ）。这些属性代表与文法符号相关的信息例如它的类型、值、代码序列 、符号表内容等等。属性和变量一样，可以进行计算和传递。属性一般分为两类：
综合属性 用于 自下而上 传递信息，
继承属性 用于 自上而下 传递信息。

综合属性(synthesized attribute)：在语法树中，一个结点的综合属性的值由其子结点 或其自身 的某些 属性值确定。通常使用自 下而 上的方法在每一个结点处使用语义规则计算综合属性的值。仅仅使用综合属性的属性文法称 S 属性文法。

继承属性(inherited attribute)：在语法树中一个结点的继承属性 值 由 该 结点的父结点 、 兄弟结点和其自身 的某些属性 值 确定 。通常用 继承属性来表示程序语言结构中的上下文依赖关系。

接下来的实验之中将要利用属性文法的综合、继承属性来实现最后的结果。

实验流程图



数据结构，全局变量，函数作用分析

```
struct ele//每个节点的属性
{
    int i; //继承
    int s; //综合
    //如果为寄存器则记录地址，如为立即数则记录数值
    int ireg=-1;   //继承的为寄存器（1）还是立即数（0），初始化为未知类型（-1）
    int sreg=-1;   //综合的为寄存器（1）还是立即数（0），初始化为未知类型（-1）
};
```

结构体ele记录每个节点的继承，综合属性，以及两个标志位表示上述两个属性来自立即数或寄存器。

函数功能说明：

`int reg[200]`： 寄存器
`int r=0`： 分配存储器位置的指针

`int rid`： 记录存储着id内容的寄存器，小于等于rid的寄存器都存储着id的内容

`int ID[130]`： 根据ASCII值定位某个ID所在存储器的位置
`char str[maxlen]`： 存储算术表达式
`int f1=0`： 指向当前分析输入字符

`int inreg(char a)`： 判断ID是否存储在寄存器中
`void error()`： 报错并退出程序
`ele code(char op, ele e1, ele e2)`： 对数据进行运算
`int check(char c)`： 判断传入字符的类型
`bool AddOp(char c)`： 判断是否为加减运算符
`bool MulOp(char c)`： 判断是否为乘除运算符
`void Input()`： 输入必要的数据
`void Output()`： 输出相关结果

`ele Expr(ele e) & ele Expr1(ele e) & ele Term(ele e) & ele Term1(ele e) & ele Factor(ele e)`： 利用递归下降的方法对其进行语法分析，同时进行代码翻译

关键代码分析

接下来一节只展示重要代码，具体细节可以在源码中注释中查看

下面代码展示了code函数的功能，该函数需要输入一个运算符以及两个节点ele；其中第一个节点使用其综合属性，第二个节点使用其继承属性。

一般来说最后结果回存储到e2节点对应的寄存器中，但是实际操作过程中会遇到一些问题：

e2可能传递的是一个立即数，或者e2对应的是一个变量的寄存器；在这些情况下，编译器不能直接将计算的结果存入；对于立即数，我们将其进行一个MOV r1 i1 的指令，将该立即数存储到一个新的寄存器中，使对应节点的继承属性转变成寄存器的地址。

可能有一个令人纠结的点是，为何不在Factor函数中读入立即数的时候就将每一个立即数分配一个寄存器，这样这个问题就可以简单的解决。但是需要注意的是，在汇编指令中，是允许对立即数进行操作的，比如 ADD reg1 3；将reg1存储的内容加3再后存入reg1。因此为了更好体现汇编指令的实际操作，我们需要在code函数中对不同情况进行分类后进行不同的处理。同时，这也是为什么需要在于变量对应的寄存器冲突的时候加一步MOV指令将变量的值移植到新寄存器中；因为实际的汇编指令并没有 ADD reg1 reg2->reg3 将两个寄存器计算的结果存储到第三个寄存器。具体细节，请详见下列代码：

```
ele code(char op, ele e1, ele e2)
{
    if(e2.ireg==0)  //e2作为最后运算结果存放的位置需要有一个寄存器来存储结果，如果是一个立即数则需要将立即数放入到寄存器中
             {
                  reg[r]=e2.i;    //将立即数放到寄存器中
                  e2.ireg=1;      //修改e2继承属性标志为寄存器
                  e2.i=r;         //将e2的继承的值改为寄存器的位置
                  cout<<"MOV\treg"<<r<<"\t"<<reg[r]<<"\n";    //打印汇编指令，形成三元组
                  r++;
             }
    else if(e2.i<=rid)  //如果e2是存储着id内容的寄存器，在算术表达式中，id的内容不改变，所以需要赋值到一个新的寄存器中
            {
                  reg[r]=reg[e2.i];    //将id的值放到新寄存器中
                  int temp=e2.i;
                  e2.i=r;               //将e2的继承的值改为新寄存器的位置
```

```
                cout<<"MOV\treg"<<r<<"\treg"<<temp<<"\n";    //打印汇编指令，形成三元组
                r++;
            }

    switch(op)  //根据不同的操作符有不同的操作
    {
    case '+':
             if(e1.sreg==0)    //立即数
             {
                 reg[e2.i]=reg[e2.i]+e1.s;  //计算结果，并存到目标寄存器中
                 cout<<"ADD\treg"<<e2.i<<"\t"<<e1.s<<"\treg"<<e2.i<<"\n";//打印汇编指令，形成四元组
             }
             else   //寄存器
             {
                 reg[e2.i]=reg[e2.i]+reg[e1.s];//计算结果，并存到目标寄存器中
                 cout<<"ADD\treg"<<e2.i<<"\treg"<<e1.s<<"\treg"<<e2.i<<"\n";//打印汇编指令，形成四元组
             }
         return e2;
    case '-':
            if(e1.sreg==0)    //立即数
             {
                 reg[e2.i]=reg[e2.i]-e1.s;//计算结果，并存到目标寄存器中
                 cout<<"SUB\treg"<<e2.i<<"\t"<<e1.s<<"\treg"<<e2.i<<"\n";//打印汇编指令，形成四元组
             }
             else   //寄存器
             {
                 reg[e2.i]=reg[e2.i]-reg[e1.s];//计算结果，并存到目标寄存器中
                 cout<<"SUB\treg"<<e2.i<<"\treg"<<e1.s<<"\treg"<<e2.i<<"\n";//打印汇编指令，形成四元组
             }
         return e2;
    case '*':
            if(e1.sreg==0)    //立即数
             {
                 reg[e2.i]=reg[e2.i]*e1.s;//计算结果，并存到目标寄存器中
                 cout<<"MUL\treg"<<e2.i<<"\t"<<e1.s<<"\treg"<<e2.i<<"\n";//打印汇编指令，形成四元组
             }
             else   //寄存器
             {
                 reg[e2.i]=reg[e2.i]*reg[e1.s];//计算结果，并存到目标寄存器中
                 cout<<"MUL\treg"<<e2.i<<"\treg"<<e1.s<<"\treg"<<e2.i<<"\n";//打印汇编指令，形成四元组
             }
         return e2;
    case '/':
            if(e1.sreg==0)    //立即数
             {
                 reg[e2.i]=reg[e2.i]/e1.s;//计算结果，并存到目标寄存器中
                 cout<<"DIV\treg"<<e2.i<<"\t"<<e1.s<<"\treg"<<e2.i<<"\n";//打印汇编指令，形成四元组
             }
             else   //寄存器
             {
```

```
                reg[e2.i]=reg[e2.i]/reg[e1.s];//计算结果，并存到目标寄存器中
                cout<<"DIV\treg"<<e2.i<<"\treg"<<e1.s<<"\treg"<<e2.i<<"\n";//打印汇编指令，形成四元组
            }
        return e2;
    default:
        cout<<"运算符出错";
        error();
        return e2;
    }
}
```

语法分析以及代码翻译只展示Expr,Expr1,Factor部分，Term和Term与上述比较类似

其中值得注意的是：如果整个式子只有一个单独的数的话，编译器需要帮助他单独分配一个寄存器存储结果，为以后的处理以及统一输出最终结果做准备，具体可见下面代码

```
ele Expr(ele e)
{
        ele T,E1,E;
        E=e;    //E获得继承的属性
        T=Term(T);  //T获得综合属性
        E1.i=T.s;   //E1获得继承属性
        E1.ireg=T.sreg;
        E1=Expr1(E1);
        E.s=E1.s;   //E获得综合属性
        E.sreg=E1.sreg;
        if(E.sreg==0)   //为式子中只有一个单独的数的情况下所准备的
        {
            reg[r]=E.s;
            E.sreg=1;
            E.s=r;
            cout<<"MOV\treg"<<r<<"\t"<<reg[r]<<"\n";
            r++;
        }
        return E;
}



ele Expr1(ele e)
{
    if(AddOp(str[f1])){     //match('+','-')
        ele E1,T,E2;
        E1=e;
        char addop=str[f1];
        ++f1;
        T=Term(T);  //综合的结果
        E1=code(addop,T,E1);    //进行加减计算
        E2.i=E1.i;  //继承
        E2.ireg=E1.ireg;
        E2=Expr1(E2);
        E1.s=E2.s;  //综合
        E1.sreg=E2.sreg;
```

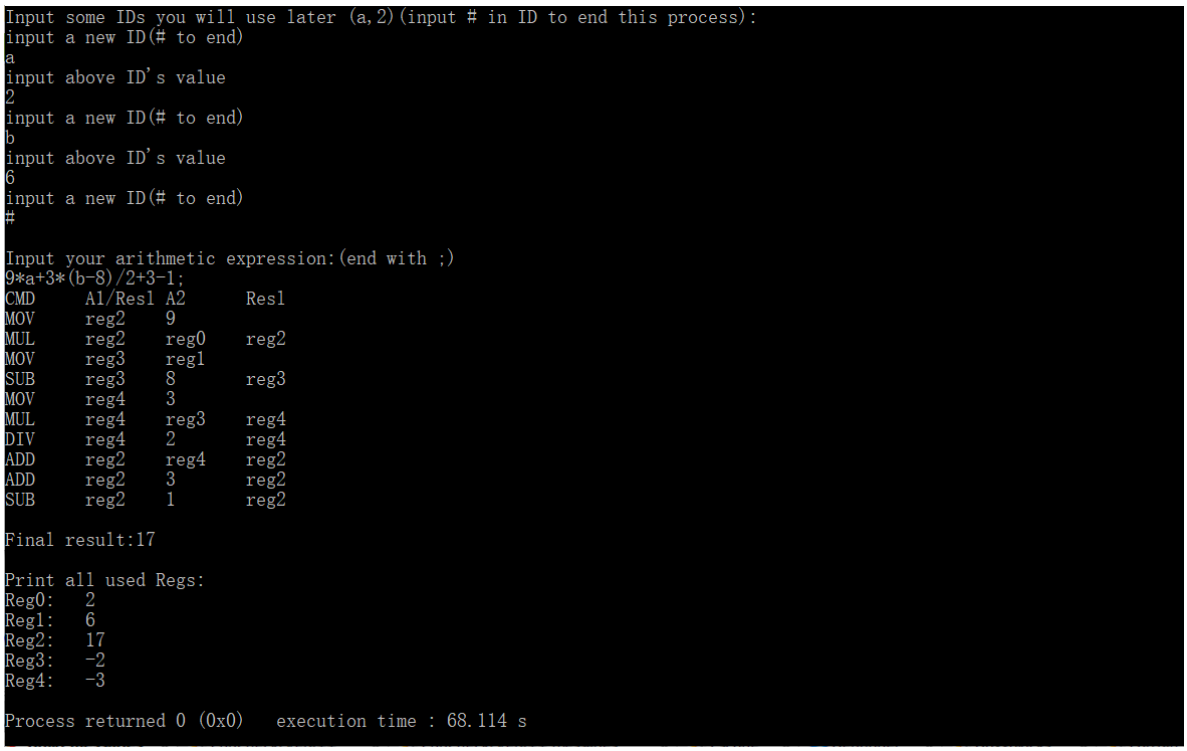
```
        return E1;
    }
    else if(str[f1]==')'||str[f1]==';'){    //E1->空  match Follow set
        ele E1=e;
        E1.s=E1.i;  //综合得到自身的继承属性
        E1.sreg=E1.ireg;
        return E1;
    }
    else{     //error
        cout<<"算术表达式格式错误";
        error();

    }
}
```

```
ele Factor(ele e){
    int b=check(str[f1]);
    if(b==1){       //id
        char id=str[f1];
         ++f1;              //match(id)
         ele F=e;
         int rreg=inreg(id);
        if(rreg!=-1)
        {
            F.s= rreg;
            F.sreg=1;     //综合的为一个寄存器
            return F;
        }
        else
        {
            cout<<"变量未提前定义! ";
            error();
        }
    }
    else if(b==2)       //number
    {
        char num =str[f1];
        ++f1;       //match(number)
        ele F=e;
        F.s= num-'0';
        F.sreg=0;      //综合的为一个立即数
        return F;
    }
    else if(str[f1]=='('){
        ++f1;              //match '('
        ele F,E;
        F=e;
        E=Expr(E);
        F.s=E.s;
        F.sreg=E.sreg;
        if(str[f1]==')'){ //match(')')
            ++f1;

        }
        else{ //error
            cout<<"算术表达式格式错误(括号不完整)";
            error();
```

```
            }
            return F;
        }
        else{     //Error
            cout<<"算术表达式格式错误";
            error();

        }
    }
```

实验结果

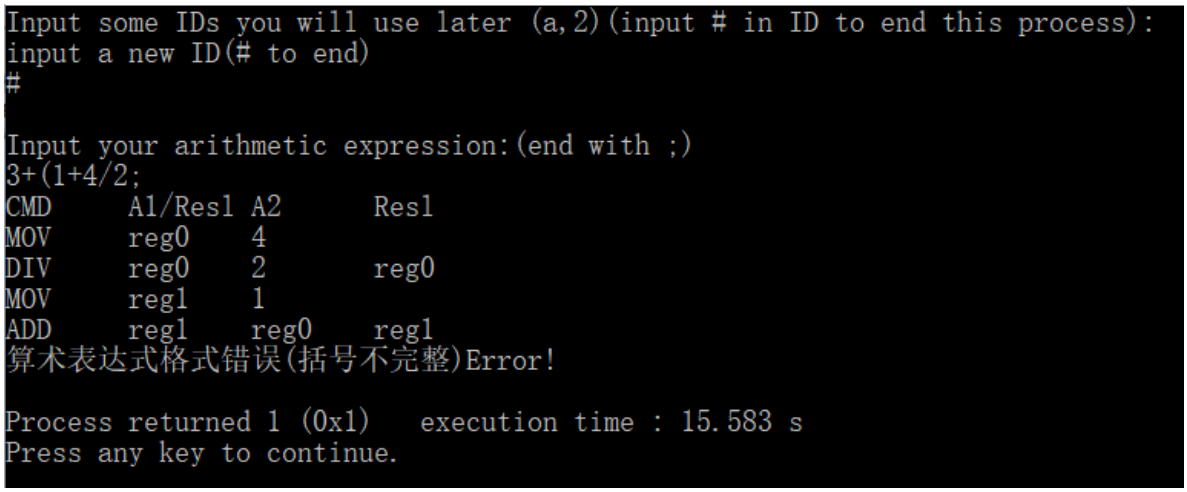
接下来这一节将会介绍实验的最终结果以及对结果的一些分析，具体结果可见下图：



```
Input some IDs you will use later (a,2)(input # in ID to end this process):
input a new ID(# to end)
a
input above ID's value
2
input a new ID(# to end)
b
input above ID's value
6
input a new ID(# to end)
#

Input your arithmetic expression:(end with ;)
9*a+3*(b-8)/2+3-1;
CMD     A1/Res1 A2      Res1
MOV     reg2    9
MUL     reg2    reg0    reg2
MOV     reg3    reg1
SUB     reg3    8       reg3
MOV     reg4    3
MUL     reg4    reg3    reg4
DIV     reg4    2       reg4
ADD     reg2    reg4    reg2
ADD     reg2    3       reg2
SUB     reg2    1       reg2

Final result:17

Print all used Regs:
Reg0:   2
Reg1:   6
Reg2:   17
Reg3:   -2
Reg4:   -3

Process returned 0 (0x0)   execution time : 68.114 s
```

首先，程序会要求你输入接下来需要用到的变量id及其对应的值；通过程序提示，在id部分输入#退出id的键入。随后，需要正确输入算数表达式，否则系统会进行报错并退出程序

错误示例：



```
Input some IDs you will use later (a,2)(input # in ID to end this process):
input a new ID(# to end)
#

Input your arithmetic expression:(end with ;)
3+(1+4/2;
CMD     A1/Res1 A2      Res1
MOV     reg0    4
DIV     reg0    2       reg0
MOV     reg1    1
ADD     reg1    reg0    reg1
算术表达式格式错误(括号不完整)Error!

Process returned 1 (0x1)   execution time : 15.583 s
Press any key to continue.
```

同时，如果需要在算术表达式中使用变量id的话，必须要在之前声明，否则同样也会报错并退出程序。

错误示例：

```
Input some IDs you will use later (a,2)(input # in ID to end this process):
input a new ID(# to end)
a
input above ID's value
1
input a new ID(# to end)
#

Input your arithmetic expression:(end with ;)
a+b;
CMD     A1/Res1 A2      Res1
变量未提前定义！Error!

Process returned 1 (0x1)   execution time : 11.664 s
Press any key to continue.
```

如果正确输入变量以及算术表达式后，可以得到正确的输出，接下来将解释输出的内容：

CMD　　A1/Resl　　A2　　Resl

指令类型 操作数1/结果存储地址　　操作数2　　结果存储地址

对于MOV指令，因为只涉及移动功能所以该程序会将其转变成三元组，第一个操作元素表示操作指令为移动指令，第二个元素为移动目的寄存器，第三个元素为待移动的立即数或待移动的寄存器，如果是立即数则直接将立即数装入目标寄存器，如果是寄存器，则将源寄存器中的值装入到目标寄存器之中。

对于ADD SUB MUL DIV指令，该程序会将其转变成四元组，第一个操作元素表示操作指令的指令类型，第二个，第三个元素表示就算需要用到的操作数，A1只能为寄存器，A2可以为寄存器或立即数；第四个元素为最终运算结果存储地址。通常来说，最终存储地址都与A1的地址相同。

将各个步骤的汇编指令，三/四元组 打印之后，程序输出该算术表达式的最终计算结果。

最后，该程序将所有使用了的寄存器及其内容打印出来，将寄存器的内容打印出来可以供我们对比分析。

实验九-对多条执行语句构造递归下降翻译器&实验十-构造能处理完整程序的递归下降翻译器

由于实验九，实验十内容重复度较高，故将其合并一起完成，实现代码见文件夹3.2

基本文法

本实验使用到的文法与实验6一致，这里将其文法和First集，Follow集再次列出。

```
PROG→int main ( ) BLOCK
BLOCK→{ DECLS STMTS }

DECLS -> DECLS1
DECLS1 -> DECL DECLS1 | empty

DECL→TYPE NAMES ;
TYPE→int

NAMES -> NAME NAMES1
NAMES1 -> , NAME NAMES1 | empty

NAME→id

STMTS -> STMTS1
STMTS1 -> STMT STMTS1 | empty
```

```
STMT→id = EXPR ;
STMT→if ( BOOL ) STMT

STMT→if ( BOOL ) STMT else STMT
STMT→while ( BOOL ) STMT
STMT→BLOCK

STMT→ return int ;

BOOL→EXPR ROP EXPR
ROP→ > | >= | < | <= | == | !=

EXPR→TERM EXPR1

EXPR1→ADDOP TERM EXPR1| empty

TERM→FACTOR TERM1

TERM1→MULOP FACTOR TERM1 | empty

FACTOR→id | int |(EXPR)

ADDOP→+ | -

MULOP→* | /
```

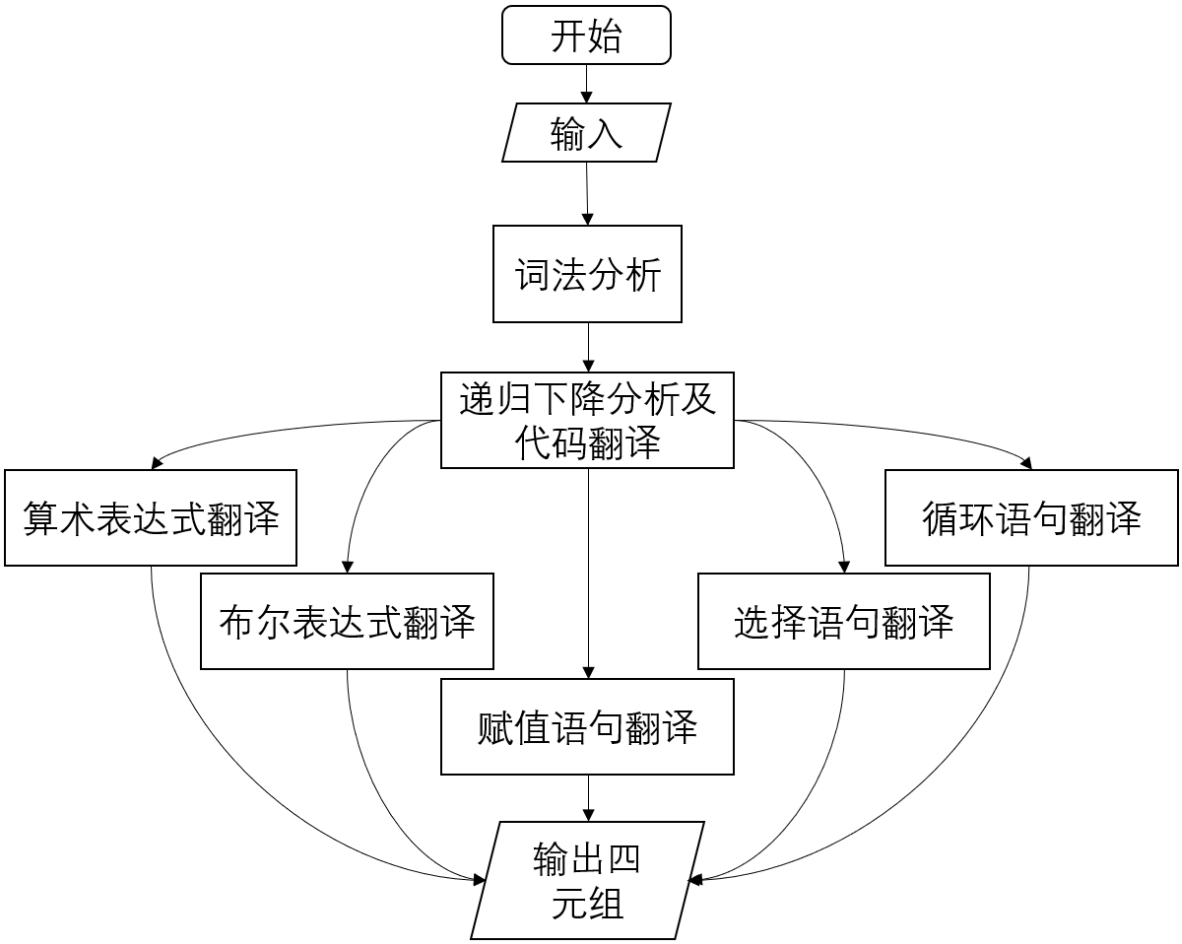
给定的文法是无二义性的，且没有左递归。因此，可以对其进行直接分析；

First集&&Follow集

根据给定的文法，对每个非终结符构造First集和Follow集，结果如下所示。并且可以看出对于文法的任一非终结符，其规则右部的多个选择所推导的First集与非终结符的Follow集（当存在A→ empty）的情况下不相交。

列1	列2	列3
	First	Follow
PROG	int	{
BLOCK	{	id，if，while，return,{，}，#
DECLS	int，empty	id，if，while，return, {
DECLS1	int，empty	id，if，while，return,{
DECL	int	int，id，if，while，return,{, }
NAMES	id	;
NAMES1	,，empty	;
NAME	id	,，;
TYPE	int	id
STMTS	id，if，while，{，empty	}
STMTS1	id，if，while，{，empty	}
STMT	id，if，while，{	id，if，while，return, {，}
BOOL	id，int, (	)
ROP	>, <, ==, !=, >=, <=	id，int, (
EXPR	id，number，(	), ;, >, <, ==, !=, >=, <=
EXPR1	+, -	), ;, >, <, ==, !=, >=, <=
TERM	id, int, (	+, -,), ;, >, <, ==, !=, >=, <=
TERM1	*, /	+, -,), ;, >, <, ==, !=, >=, <=
FACTOR	id, int, (	*, /, +, -,), ;, >, <, ==, !=, >=, <=
ADDOP	-,+	id，int, (
MULOP	* /	id, int，(

实验流程图



数据结构，全局变量，函数作用分析

由于实验八&六中的所有数据结构，全局变量以及函数都在实验九&十中运用到了，接下来将只关注一些新增的部分

`int line;` 存储汇编指令的行数

```
struct ID       //记录id存储的寄存器
{
    string name;
    int reg_id;
}id[50];
```

`int fid;` 统计共存储了多少个id

```
struct JumpStorage  //跳转指令存储位置
{
    string name;
    int line;
}jumps[50];
```

`int js;` 存储跳转指令的个数

关键代码分析

赋值语句

```
void NAME(){
    int x=ele[cnt].type;            //x记录当前单词的类别
    string y=ele[cnt].value;
```

```
    if(x==23){                    //id
        cnt++;
        id[fid].name=y;
        id[fid].reg_id=r;
        cout<<"(给"<<y<<"分配reg"<<r<<")\n";
        r++;
        fid++;

    }
    else{       //Error
        error=1;
    }
}
```

在对变量赋值之前，需要先对变量进行声明；在此过程中，编译器给其分配寄存器，并将id与其对应的寄存器保存起来（本实验保存在结构体数组id中）。

注：此段代码存在于STMT()中

```
    int x=ele[cnt].type;           //x记录当前单词的类别
    string y=ele[cnt].value;
if(x==23){                    //类别(id),赋值语句
        ++cnt;
        if(ele[cnt].type!=6){
            error=1;
            return;
        }
        ++cnt;                   //match('=')
        Ele E;
        E=EXPR(E);

        if(error==1) return;      //如果出现错误，直接return
        if(ele[cnt].type==19){     //match(';')
            ++cnt;
        }
        else{
            error=1;
        }
        int idreg=Findid(y);
        if(idreg!=-1)
        {
        reg[idreg]=reg[E.s];
        id[idreg].name=y;
         cout<<line<<":\tMOV\treg"<<idreg<<"\treg"<<E.s<<"\n";    //打印汇编指令，形成三元组
        line++;
        }
        else
        cout<<"变量未提前声明\n";

    }
```

在赋值表达始中，首先找到对应id存储的寄存器，在通过EXPR返回的值，将最终结果通过MOV指令放到id对应的寄存器之中

布尔表达式

```
void BOOL(int t){
    int x=ele[cnt].type;            //x记录当前单词的类别
    Ele e1,e2;
    e1=EXPR(e1);
    if(error==1) exit(1);
    int rop = ROP();
    if(error==1) exit(1);
    e2=EXPR(e2);
    if(error==1) exit(1);

    switch(rop)
    {
    case 11:    //>
         cout<<line<<":\tCMP\treg"<<e1.s<<"\treg"<<e2.s<<"\n";    //打印汇编指令，形
成三元组
         line++;
         if(t==1)//if
            {

            cout<<line<<":\tJLE\tIFNEXT\n";
            line++;

            }
         else if(t==3)//while
            {

            cout<<line<<":\tJLE\tWHILENEXT\n";
            line++;

            }
            break;
    case 12:    //<
        cout<<line<<":\tCMP\treg"<<e1.s<<"\treg"<<e2.s<<"\n";    //打印汇编指令，形
成三元组
        line++;
         if(t==1)//if
            {

            cout<<line<<":\tJGE\tIFNEXT\n";
            line++;

            }
         else if(t==3)//while
            {

            cout<<line<<":\tJGE\tWHILENEXT\n";
            line++;

            }
break;
    case 13:    //==
        cout<<line<<":\tCMP\treg"<<e1.s<<"\treg"<<e2.s<<"\n";    //打印汇编指令，形
成三元组
        line++;
         if(t==1)//if
            {
```

```
            cout<<line<<":\tJNZ\tIFNEXT\n";
            line++;

            }
         else if(t==3)//while
            {

            cout<<line<<":\tJNZ\tWHILENEXT\n";
            line++;

            }
            break;
    case 14:    // !=
        cout<<line<<":\tCMP\treg"<<e1.s<<"\treg"<<e2.s<<"\n";   //打印汇编指令，形
成三元组
        line++;
         if(t==1)//if
            {

            cout<<line<<":\tJZ\tIFNEXT\n";
            line++;

            }
         else if(t==3)//while
            {

            cout<<line<<":\tJZ\tWHILENEXT\n";
            line++;

            }
            break;
    case 15:    //>=
        cout<<line<<":\tCMP\treg"<<e1.s<<"\treg"<<e2.s<<"\n";   //打印汇编指令，形
成三元组
        line++;
         if(t==1)//if
            {

            cout<<line<<":\tJL\tIFNEXT\n";
            line++;

            }
         else if(t==3)//while
            {

            cout<<line<<":\tJL\tWHILENEXT\n";
            line++;

            }
            break;
    case 16:    //<=
        cout<<line<<":\tCMP\treg"<<e1.s<<"\treg"<<e2.s<<"\n";   //打印汇编指令，形
成三元组
        line++;
         if(t==1)//if
            {

            cout<<line<<":\tJG\tIFNEXT\n";
```

```
            line++;

            }
         else if(t==3)//while
            {

            cout<<line<<":\tJG\tWHILENEXT\n";
            line++;

            }

    }
}
```

此实验中布尔表达式只实现了两个EXPR的大小比较；通过两个EXPR式子返回的结果，进行6中不同的比较方式(< > == != <= >=),对两个值进行CMP指令比较，根据结果进行跳转指令。

以a>b比较方式作为例子；CMP指令会比较a b两个数的大小，实际上是进行一次a-b运算，如果a>b,标志位中就会存储大于零的结果，反之则存储小于零的结果；如果相等则存储等于零的结果。

接下来，会判断结果是否于之前bool条件相反，在此例子中则判断a>=b?；如果a>=b,则会跳过下列代码，其中，通过传入的实参t判断接下来的代码是while还是if；整体功能写成代码则为JLE IFNEXT/WHILENEXT

选择语句

注：此段代码存在于STMT()中

```
   int x=ele[cnt].type;            //x记录当前单词的类别
   string y=ele[cnt].value;

   ......

else if(x==1){              //类别('if')
       ++cnt;
       if(ele[cnt].type!=21){     //match(
           error=1;
           return;
       }
       ++cnt;
       BOOL(x);
       if(error==1) return;       //如果出现错误，直接return
       if(ele[cnt].type!=22){     //match(')')
           error=1;
           return;
       }
       ++cnt;
       STMT();
       if(error==1) return;       //如果出现错误，直接return
       if(ele[cnt].type==2)     //展望一步是否为else
       {
           cout<<line<<":\tJMP\tIFFINISH\n";
           line++;
           cout<<line<<":\tIFNEXT:\n";
           jumps[js].name="IFNEXT";
           jumps[js].line=line;
```

```
            js++;
            line++;
            ++cnt;
            STMT();
            cout<<line<<":\tIFFINISH:\n";
            jumps[js].name="IFFINISH";
            jumps[js].line=line;
            js++;
            line++;
        }
        else
            {cout<<line<<":\tIFNEXT:\n";
            jumps[js].name="WHILEBEGIN";
            jumps[js].line=line;
            js++;
            line++;}
    }
```

选择语句与循环语句一般配合着布尔表达式进行分析；在布尔表达式完成了CMP以及跳转指令后，将第一个STMT翻译成汇编语言；接下来判断是否有else语句；如果没有则直接输出跳转地址IFNEXT并保存，而如果有else语句，则先输出JMP IFFINISH，令执行了第一个STMT的情况下（布尔表达式为真时）不再执行第二个STMT的内容；最后输出IFNEXT的跳转地址并保存，并执行第二个STMT语句（布尔表达式为假时），最后使出IFFINISH的跳转地址并保存

循环语句

注：此段代码存在于STMT()中

```
    int x=ele[cnt].type;            //x记录当前单词的类别
    string y=ele[cnt].value;

    ......


else if(x==3){              //类别('while')
        ++cnt;
        if(ele[cnt].type!=21){     //match('(')
            error=1;
            return;
        }
        ++cnt;
        cout<<line<<":\tWHILEBEGIN:\n";
        jumps[js].name="WHILEBEGIN";
        jumps[js].line=line;
        js++;
        line++;
        BOOL(x);
        if(error==1) return;       //如果出现错误，直接return
        if(ele[cnt].type!=22){     //match(')')
            error=1;
            return;
        }
        ++cnt;
        STMT();
        cout<<line<<":\tJMP\tWHILEBEGIN\n";
        line++;
```

```
        cout<<line<<":\tWHILENEXT:\n";
        jumps[js].name="WHILENEXT";
        jumps[js].line=line;
        js++;
        line++;

    }
```

循环语句的实现与选择语句类似；需要区别的是：循环语句不需判断是否还有else类似的语句，同时，在STMT执行完毕之后，输出JMP WHILEBEGIN跳转指令到循环的开头，再次判断是否布尔表达式为假，为假的话跳转到循环的结尾。

算术表达式

```
Ele EXPR(Ele e){
    //cout<<"进入EXPR\n";
    int x=ele[cnt].type;            //x记录当前单词的类别
        Ele T,E1,E;
        E=e;    //E获得继承的属性
        T=TERM(T);  //T获得综合属性
        E1.i=T.s;   //E1获得继承属性
        E1.ireg=T.sreg;
        //if(error==1) exit(1);      //如果出现Error，直接return
        E1=EXPR1(E1);
        E.s=E1.s;   //E获得综合属性
        E.sreg=E1.sreg;
        if(E.sreg==0)
        {
            reg[r]=E.s;
            E.sreg=1;
            E.s=r;
            cout<<line<<":\tMOV\treg"<<r<<"\t"<<reg[r]<<"\n";
            line++;
            r++;
        }
        return E;
}

Ele EXPR1(Ele e){
    int x=ele[cnt].type;        //x记录当前单词的类别
    if(x==7||x==8){         //ADDOP
        ++cnt;               //match('+' or '-')
        Ele E1,T,E2;
        E1=e;
        T=TERM(T);
        E1=code(x,T,E1);    //进行加减计算
        E2.i=E1.i;  //继承
        E2.ireg=E1.ireg;
        //if(error==1) exit(1);   //如果出现Error，直接return
        E2=EXPR1(E2);
        E1.s=E2.s;  //综合
        E1.sreg=E2.sreg;
        return E1;
    }
    else if(x==22||x==19||(x>=11&&x<=16)){
        Ele E1=e;
        E1.s=E1.i;  //综合得到自身的继承属性
```

```
        E1.sreg=E1.ireg;
        return E1;
    }
    else{     //Error
        error=1;
    }
}

Ele TERM(Ele e){
    //cout<<"进入TERM\n";
    Ele T,F,T1;
    T=e;

    F=FACTOR(F);
    T1.i=F.s;   //继承来自兄弟的属性
    T1.ireg=F.sreg;
        //if(error==1) exit(1);   //如果出现Error，直接return
    T1=TERM1(T1);
    T.s=T1.s;   //综合
    T.sreg=T1.sreg;
    return T;

}

Ele TERM1(Ele e){

    int x=ele[cnt].type;        //x记录当前单词的类别
    //cout<<"进入TERM1且x="<<x<<"\n";
    if(x==9||x==10){         //MULOP
        ++cnt;               //match('*' or '/')
        Ele T1,F,T2;
        T1=e;
        F=FACTOR(F);
        T1=code(x,F,T1);    //进行乘除计算
        T2.i=T1.i;  //继承
        T2.ireg=T1.ireg;
        //if(error==1)
            //exit(1);   //如果出现Error，直接return
        T2=TERM1(T2);
        T1.s=T2.s;  //综合
        T1.sreg=T2.sreg;
        return T1;
    }
    else if((x>=7&&x<=8)||x==22||x==19||(x>=11&&x<=16)){
                Ele T1=e;
                T1.s=T1.i;//综合得到自身的继承属性
                T1.sreg=T1.ireg;
                return T1;     //match follow set
    }
    else{     //Error
        error=1;
    }
}

Ele FACTOR(Ele e){

    int x=ele[cnt].type;      //x记录当前单词的类别
    string y=ele[cnt].value;
```

```
    //cout<<"进入FACTOR且x，y为\n"<<x<<"\t"<<y<<"\n";
    if(x==23){          //类别('id')

        ++cnt;              //match(id)
        Ele F=e;
        int idreg=Findid(y);
        if(idreg!=-1)
        {
        F.s=idreg;
        F.sreg=1;
        return F;
        }
        else
        {
           cout<<"ID未定义";
           exit(1);
        }

    }
    else if(x==0)  //(int)
    {
        ++cnt;
        Ele F=e;
        F.s= atoi(y.c_str());
        //cout<<"F.S="<<F.s<<"\n";
        F.sreg=0;       //综合的为一个立即数
        return F;
    }
    else if(x==21){
        ++cnt;              //match('(')
        Ele F,E;
        F=e;
        E=EXPR(E);
        F.s=E.s;
        F.sreg=E.sreg;

        if(error==1) exit(1); //如果出现Error，直接return
        if(ele[cnt].type==22){//match(')')
            ++cnt;
        }
        else{
            error=1;
        }
    }
    else{     //Error
        error=1;
    }
}

Ele code(int op, Ele e1, Ele e2)
{
    if(e2.ireg==0)  //e2作为最后运算结果存放的位置需要有一个寄存器来存储结果，如果是一个立即数则需要将立即数放入到寄存器中
             {
                 reg[r]=e2.i;    //将立即数放到寄存器中
                 e2.ireg=1;      //修改e2继承属性标志为寄存器
                 e2.i=r;         //将e2的继承的值改为寄存器的位置
```

```
                cout<<line<<":\tMOV\treg"<<r<<"\t"<<reg[r]<<"\n";    //打印汇编指令，形成三元组
                line++;
                r++;
            }
            else if (IfId(e2.i))//如果e2是存储着id内容的寄存器，在算术表达式中，id的内容不改变，所以需要赋值到一个新的寄存器中
            {
                reg[r]=reg[e2.i];    //将id的值放到新寄存器中
                int temp=e2.i;
                e2.i=r;                //将e2的继承的值改为新寄存器的位置
                cout<<line<<":\tMOV\treg"<<r<<"\treg"<<temp<<"\n";    //打印汇编指令，形成三元组
                line++;
                r++;
            }


    switch(op)  //根据不同的操作符有不同的操作
    {
    case 7:
            if(e1.sreg==0)    //立即数
            {
                reg[e2.i]=reg[e2.i]+e1.s;  //计算结果，并存到目标寄存器中
                cout<<line<<":\tADD\treg"<<e2.i<<"\t"<<e1.s<<"\n";//打印汇编指令，形成四元组
                line++;
            }
            else   //寄存器
            {
                reg[e2.i]=reg[e2.i]+reg[e1.s];//计算结果，并存到目标寄存器中
                cout<<line<<":\tADD\treg"<<e2.i<<"\treg"<<e1.s<<"\n";//打印汇编指令，形成四元组
                line++;
            }
        return e2;
    case 8:
            if(e1.sreg==0)    //立即数
            {
                reg[e2.i]=reg[e2.i]-e1.s;//计算结果，并存到目标寄存器中
                cout<<line<<":\tSUB\treg"<<e2.i<<"\t"<<e1.s<<"\n";//打印汇编指令，形成四元组
                line++;
            }
            else   //寄存器
            {
                reg[e2.i]=reg[e2.i]-reg[e1.s];//计算结果，并存到目标寄存器中
                cout<<line<<":\tSUB\treg"<<e2.i<<"\treg"<<e1.s<<"\n";//打印汇编指令，形成四元组
                line++;
            }
        return e2;
    case 9:
            if(e1.sreg==0)    //立即数
            {
                reg[e2.i]=reg[e2.i]*e1.s;//计算结果，并存到目标寄存器中
                cout<<line<<":\tMUL\treg"<<e2.i<<"\t"<<e1.s<<"\n";//打印汇编指令，形成四元组
```

```
                line++;
            }
            else   //寄存器
            {
                reg[e2.i]=reg[e2.i]*reg[e1.s];//计算结果，并存到目标寄存器中
                cout<<line<<":\tMUL\treg"<<e2.i<<"\treg"<<e1.s<<"\n";//打印汇编指
令，形成四元组
                line++;
            }
        return e2;
    case 10:
            if(e1.sreg==0)    //立即数
            {
                reg[e2.i]=reg[e2.i]/e1.s;//计算结果，并存到目标寄存器中
                cout<<line<<":\tDIV\treg"<<e2.i<<"\t"<<e1.s<<"\n";//打印汇编指
令，形成四元组
                line++;
            }
            else   //寄存器
            {
                reg[e2.i]=reg[e2.i]/reg[e1.s];//计算结果，并存到目标寄存器中
                cout<<line<<":\tDIV\treg"<<e2.i<<"\treg"<<e1.s<<"\n";//打印汇编指
令，形成四元组
                line++;
            }
        return e2;
    default:
        cout<<"运算符出错";
        exit(1);
        return e2;
    }
}
```

算数表达式与实验八大体类似，在实验九&十中，对其first follow集进行了补充，具体细节请看上述代码。

实验结果

输入内容：

```
int main ()
{
int a,b;
a=1+1;
b=4*(3-1)/2;
while (a<5)
{
a=a+1;
}
if(b>a)
{
b=a;
}
else
{
a=b;
}
```

```
return 0;
}
```

输出结果

```
line    CMD     A1/Res1 A2
(给a分配reg0)
(给b分配reg1)
0:      MOV     reg2    1
1:      ADD     reg2    1
2:      MOV     reg0    reg2
3:      MOV     reg3    3
4:      SUB     reg3    1
5:      MOV     reg4    4
6:      MUL     reg4    reg0
7:      DIV     reg4    2
8:      MOV     reg1    reg4
9:      WHILEBEGIN:
10:     MOV     reg5    5
11:     CMP     reg0    reg5
12:     JNZ     WHILENEXT
13:     MOV     reg6    reg0
14:     ADD     reg6    1
15:     MOV     reg0    reg6
16:     JMP     WHILEBEGIN
17:     WHILENEXT:
18:     CMP     reg1    reg0
19:     JLE     IFNEXT
20:     MOV     reg1    reg0
21:     JMP     IFFINISH
22:     IFNEXT:
23:     MOV     reg0    reg1
24:     IFFINISH:

Print all storage of jump instruction:
WHILEBEGIN:   line9
WHILENEXT:   line17
IFNEXT:   line22
IFFINISH:   line24

Process returned 0 (0x0)   execution time : 3.843 s
Press any key to continue.
```

根据输入的C语言代码，该程序将其翻译成对应的8086汇编代码，并且记录行数，方便进行回填操作。