一、内容

一）目标

设计并实现 Lexical Analyzer 中的 C 语言子集的语法制导翻译程序，语义分析及类型检查，并打印分析结果。实现以下功能：

1、

a) 能够实现对整型（int）及布尔变量的类型检查，两类变量不能相互赋值及运算；仅整型变量才能参与算术运算；
b) 实现整数计算器的语义动作定义及语法制导翻译，并绘制出相应语法树；
c) 为表达式“（2+3”生成错误信息“丢失右括号”；
d) 为表达式“2 3”生成错误信息“丢失运算符”；
e) 为表达式“（2+）”生成错误信息“丢失操作数”；

2、

能判断源代码是否符合以下语义假设并给出相应错误具体位置；

a) 过程/函数仅能定义一次、程序中所有变量均不能重名、过程/函数不可嵌套定义；
b) 定位源代码中的错误位置。
c) 能检查结构体中域是否与变量重名，不同结构体中域是否重名；

二）C 语言子集：

数据类型: **int**, 无符号整数, 取值范围 0-9999

```
int a;
int a,b;
int a = 1;
```

算术运算符: **+,-**

```
a = b + 1;
a = b + c;
```

赋值运算符: **=**

```
a = 1;
```

关系运算符: **== ,>,<,<>,>=,<=**

```
a = (b==c);
a = (b>c);
a = (b<c);
```

逻辑运算符: **&&, ||, !**

```
a = (b&&c);
a = (b||c);
a = (!b);
```

```
        条件语句: if
            if(a==b)
            {
            };
        循环语句: while
          while(a==b)
          {

          };
        输入,输出: get,put
        get(a);
        put(a);
        语句结束符:  ;

        条件语句  if else
        if(a==b)
        {

        };
        else
        {
};
```

二、过程或算法（源程序）

1. 语义分析程序总体说明：

能够实现对整型（int）及布尔变量的类型检查，两类变量不能相互赋值及运算；仅整型变量才能参与算术运算；能够检查出未声明的变量。

实现了整数计算器的语义动作定义及语法制导翻译，并绘制出了相应语法树。

能够处理以下错误：

为表达式“（2+3”生成错误信息“丢失右括号”；

为表达式“2 3”生成错误信息“丢失运算符”；

为表达式“（2+）”生成错误信息“丢失操作数”。

能判断源代码是否符合以下语义假设并给出相应错误具体位置：过程/函数仅能定义一次、程序中所有变量均不能重名、过程/函数不可嵌套定义；能够定位源代码中的错误位置。

能检查结构体中域是否与变量重名，不同结构体中域是否重名。

2. **语法制导翻译方法说明：**

采用基于 L 属性的 SDD 和 LL 语法分析的语法制导翻译方法的一种变体。借助栈构建语法分析树，在语法分析树中嵌入语义动作，通过遍历语法分析树实现语义动作。在 Syntax Analyzer 中的文法基础上，定制如下语法翻译方案。

$S \rightarrow T\ S \mid O\ S \mid C\ S \mid W\ S \mid id\ 11\ CC\ S \mid ST\ S \mid ;\ S \mid \varepsilon$
$11 \rightarrow \varepsilon$
$T \rightarrow 15\ int\ decs\ ; \mid 15\ \_Bool\ decs\ ;$
$decs \rightarrow 16\ dec\ dec'$
$dec' \rightarrow 15\ ,decs \mid \varepsilon$
$dec \rightarrow id\ 1\ assign$
$assign \rightarrow \varepsilon \mid\ =\ expr\ 12$
$O \rightarrow (id\ ) = expr;$
$1 \rightarrow \varepsilon$
$expr \rightarrow G\ 5\ expr'\ 6$
$expr' \rightarrow\ \&\&\ G\ 8\ expr'\ 9 \mid \backslash|\backslash|\ G\ 8\ expr'\ 9 \mid \varepsilon\ 10$
$G \rightarrow\ L\ 5\ G'\ 6$
$G' \rightarrow \backslash|\ L\ 8\ G'\ 9 \mid \varepsilon\ 10$
$L \rightarrow M\ 5\ L'\ 6$
$L' \rightarrow \&\ M\ 8\ L'\ 9 \mid \varepsilon\ 10$
$M \rightarrow H\ 5\ M'\ 6$
$M' \rightarrow ==\ H\ 8\ M'\ 9 \mid\ !=\ H\ 8\ M'\ 9 \mid \varepsilon\ 10$
$H \rightarrow I\ 5\ H'\ 6$
$H' \rightarrow >\ I\ 8\ H'\ 9 \mid <\ I\ 8\ H'\ 9 \mid\ >=\ I\ 8\ H'\ 9 \mid <=\ I\ 8\ H'\ 9 \mid \varepsilon\ 10$
$I \rightarrow J\ 5\ I'\ 6$
$I' \rightarrow\ +\ J\ 8\ I'\ 9 \mid -\ J\ 8\ I'\ 9 \mid \varepsilon\ 10$
$J \rightarrow K\ 5\ J'\ 6$
$J' \rightarrow *\ K\ 8\ J'\ 9 \mid\ /\ K\ 8\ J'\ 9 \mid \varepsilon\ 10$
$5 \rightarrow\ \varepsilon$
$6 \rightarrow\ \varepsilon$
$8 \rightarrow\ \varepsilon$
$9 \rightarrow\ \varepsilon$
$7 \rightarrow\ \varepsilon$
$10 \rightarrow\ \varepsilon$
$K \rightarrow\ !\ K\ 4 \mid (expr)\ 4 \mid id\ 3 \mid decimal\ 7 \mid hex\ 4$
$3 \rightarrow\ \varepsilon$
$C \rightarrow if\ (\ expr\ )\ \{\ S\ \}\ ;EL$
$EL \rightarrow \varepsilon \mid else\ \{\ S\ \}\ ;$
$W \rightarrow while\ (\ expr\ )\ \{\ S\ \}\ ;$
$CC \rightarrow (\ expr\_list) \mid\ = expr\ 13\ ;$
$12 \rightarrow \varepsilon$

$13 \rightarrow \varepsilon$
$4 \rightarrow \varepsilon$
$expr\_list \rightarrow expr\ E$
$E \rightarrow ,\ expr\ E \mid \varepsilon$
$ST \rightarrow struct\ id\ 2\ \{\ ST'\ \};$
$2 \rightarrow \backslash e$
$14 \rightarrow \backslash e$
$15 \rightarrow \backslash e$
$16 \rightarrow \backslash e$
$ST' \rightarrow 14\ T\ ST' \mid ST\ ST' \mid 14\ ;\ ST' \mid \varepsilon$

上述文法中，采用数字标号指向对应的语义动作。标号对应的具体语义动作将在后面列出。

属性说明：

属性名	属性类型	属性值(作用)	属性值类型
itype	继承属性	用于传递标识符类型	string
stype	综合属性	用于表示标识符类型	string
ip	继承属性	用于传递位置	pos
sp	综合属性	用于表示位置	pos
ival	继承属性	用于传值	int
sval	综合属性	用于表示值	int
iaddr	继承属性	用于传递地址	int
saddr	综合属性	用于表示地址	int
iname	继承属性	用于传递标识符名字	string
sname	综合属性	用于表示标识符名字	string

上述属性说明中，仅列出了属性名称的含义和类型，因为文法符号太多，不具体列出每个文法符号对应的属性。关于 pos 类型的说明详见 Syntax Analyzer。

文法中标号与语义动作的对应表如下：

标号	语义动作

1	{if(symtable.count(node(1).sname))raiseError else begin
	node(0).stype=node(1).stype;
	node(3).iname = node(1).sname;
	node(3).itype = node(0).stype;
	node(0).saddr = new addr;
	if(symtable.count(node(0).iname)&&symtable[node(0).iname].type==”struct”) begin
	if(struct_table.count(node(1).sname)printf(“Field name existed!”);
	struct_table[node(1).sname].insert(node(0).iname); end else
	symtable[node(1).sname] = symt_attr(node(1).stype,
	node(1).sp,id_val.size()); }

2	{ if(symtable.count(node(2).sname))raiseError else begin node(2).stype=”struct”; symtable[node(2).sname] = symt_attr(node(2).stype, node(2).sp); node(5).iname = node(2).sname; end }

3	{if(symtable.count(node(1).sname))raiseError else begin
	node(1).stype = symtable[node(1).sname].stype node(1).saddr = symtable[node(1).sname].saddr end;
	node(0).stype = node(1).stype;
	node(0).sp = node(1).sp;
	node(0).saddr = node(1).saddr;}

4	{ node(0).sp = node(1).sp}
5	{node(3).itype = node(1).stype; node(3).iaddr =node(1).saddr; node(3).ip = node(1).sp }
6	{node(0).stype = node(3).stype; node(0).saddr= node(3).saddr; node(0).sp = node(1).sp;}
7	{node(0).sval = int(node(1).sval); node(0).sp = node(1).sp; node(0).saddr = node(1).saddr }

8	{if(node(0).itype != node(2).stype)raiseError;
	node(4).itype = operator(1).type;
	node(4).iaddr = new addr;
	id_val[node(4).iaddr] = id_val[node(0).iaddr] operator(1)
	id_val[node(2).saddr]}

9	{node(0).stype = node(4).stype; node(0).saddr = node(4).saddr;}
10	{node(0).stype = node(0).itype; node(0).saddr = node(0).iaddr;}
11	{node(3).iname = node(1).sname; node(3).ip = node(1).sp}
12	{if(symtable[node(0).iname].type!=node(2).stype)raiseError; id_val[symtable[node(0).iname].addr]=id_val[node(2).saddr]; printf("%s = %d\n",node(0).iname,id_val[node(2).saddr]);}

13	{ if(symtable.count(node(0).iname))rasieError else begin
	if(symtable[node(0).iname].type!=node(2).stype)raiseError;
	id_val[symtable[node(0).iname].addr]=id_val[node(2).saddr];
	printf("%s = %d\n",node(0).iname,id_val[node(2).s]); end}

14	{node(2).iname = node(0).iname; node(3).iname = node(0).iname}
15	{node(3).iname = node(0).iname}
16	{node(3).iname = node(0).iname; node(2).iname = node(0).iname}

表中 node()表示取文法符号对应语法分析树中的结点。将产生式中的符号进行编号，产生式头编号为 0，产生式体中的符号从左往右依次编号为 1，2，3…。用 operator(i)表示取结点 i 代表的运算符，并用 operator(i).type 表示该运算结果的类型。id_val 表示模拟地址 addr 指向的空间的 vector。

在文法中，用标号替代语义动作的好处有三：一是语义动作较长，全部嵌入到文法中会使文法看起来很冗长，画出来的语法树中每个结点也会很扁平，用标号替代可以使文法看上去更简洁。二是文法中没有数字这样的符号，但有终结符{和}，直接将语义动作嵌入到文法中，{符号会产生歧义，而数字标号则不会产生歧义。三是在将文法

输入到程序中后，程序可以很方便地根据数字标号跳转到对应的代码语句以完成对应的语义动作，而直接输入嵌入在文法中的语义动作则面临把字符串翻译成可执行代码语句的困难。

在可以采用捷径时采用捷径，如在展开非终结符为产生式时，直接把表示类型的终结符本身赋值给 decs.itype。

在程序中，处理标号时，把它当作非终结符来看待，为不影响 FIRST 集和 FOLLOW 集的正常求取，为每个标号添加推出空语句的产生式。

在实际程序实现时，还存在一种附加在非终结符展开动作上的语义动作，即将非终结符展开成对应产生式体的时候，同时附带执行相应代码。这类动作不在产生式体中，因此它们不能被嵌入到语法产生式中，也不能作为附加结点添加到语法分析树上。这些语义动作对应于非终结符，列举如下：

非终结符	语义动作
T	{decs.itype = 上一个终结符}
decs	{dec.itype = decs.itype; dec'.itype = decs.itype}
dec'	{decs.itype = dec'.itype}
dec	{ symtable[id.sname].type = dec'.itype}

表中为表示方便，使用了自然语言进行描述。

在存在捷径或优化方法时，在代码实现中采用捷径。即能在非终结符展开时完成的动作就在此时完成，而不必通过值传递的方式等到后面完成。

当自顶向下的语法分析栈中的终结符与输入流中的符号匹配时，也可附加执行一些语义动作。列举如下。

终结符	语义动作
id	{node(id).sname=输入流中当前标识符名; node(id).stype=栈中当前终结符类型; }
decimal	{node(decimal).sval = 输入流中当前数值;node(decimal).saddr = new addr; id_val[node(decimal).saddr]=int(decimal); node(decimal).stype = “int"}
任意终结符	{node(该终结符).sp=输入流中当前标识符位置}

表中，node()表示取文法符号对应的语法分析树中的结点。int()表示取对应整数值。

关于计算功能的实现。为语法分析树中每个结点分配 iaddr 和 saddr 属性。addr 表示地址，指向存储该结点值的地址。这里用一个 vector 数组模拟存储空间，地址值就是 vector 中的下标。为每个标识符和数字单独分配空间，为每个表达式也单独分配一个空间。为表达式分配空间是考虑到后续生成汇编码时仅有 32 个寄存器，如果表达式过长，可能存在寄存器不够用的问题，但若对每个运算符都单独分配一个空间存储其中间结果，就可以避免寄存器不够用的问题。

关于结构体域重名的检查。结构体中域与变量的重名按照正常的变量名重名检查过程就可以检查出来。不同结构体中域重名的检查，则需单独建立一个 map 映射结构，它把所有结构体中的域映射到该域所属的所有结构体，后者以字符串集合的形式出现。该结构的声明如下。

```
map<string,set<string> > struct_table;// id name -> struct names
```

每当结构体中声明一个域时，便检查该 struct_table 映射结构，若结构中已存在新声明的域名，说明该域名与其他结构体中的域名重复，进行输出提示。无论新域名是否重复，都需在 map 表项中为新域名添加当前结构体名。

3. **错误处理**

通过向预测分析表中填入对应错误信息来实现错误处理，如为表达式“2 3”生成错误信息“丢失运算符”，需为第一个看到第二个十进制数 3 的非终结符添加表项，使其在看到十进制数时报错。

```
// deal with Error type
vector<string> err(2);
err[0]="Error!";
err[1]="Operator lost!";
ppt[pair<string,string>("7","decimal")]=err;
ppt[pair<string,string>("7","id")]=err;
```

“丢失操作数”这一错误可类似处理。

```
err[1]="Operand lost!";
ppt[pair<string,string>("J",")")]=err;
```

而“丢失右括号”这一错误类型，可直接在检测到输入流中不是预期的右括号时报错。

```
}else if(lct.count(X.first)){// terminal symbal unmatched
    if(X.first==")"){// ')' lost
        cout << "Semantic Error at Ln " << a->p.line <<
        ", Col " << a->p.column << "! ')' lost!" << endl;
        St.pop();
    }else{
```

对整型（int）及布尔变量的类型检查，两类变量不能相互赋值及运算。通过在语法分析树结点中传递变量的类型，并在执行运算操作的语义动作中对两边操作数的类型进行判断，来检查两操作数的类型是否相同，否则报错。

```
case 8:
    if(pare->itype!=gptree[pare->neibs[1]].stype){
        printf("%s:%d:%d: note:  mismatched types '%s' and '%s'\n",
        fn, gptree[pare->neibs[1]].sp.line,gptree[pare->neibs[1]].sp.col
        pare->itype.c_str(),gptree[pare->neibs[1]].stype.c_str());
    }
```

赋值运算的类型检查与之类似。

```
case 12:
    if(symtable[pare->iname].type!=gptree[pare->neibs[1]].stype){
        printf("%s:%d:%d: error: conversion from '%s' to '%s' requested\r
        fn,gptree[pare->neibs[0]].sp.line,gptree[pare->neibs[0]].sp.colum
        gptree[pare->neibs[1]].stype.c_str(),symtable[pare->iname].type.c
    }else{
        id_val[symtable[pare->iname].addr]=id_val[gptree[pare->neibs[1]].
        printf("%s = %d\n",pare->iname.c_str(),id_val[gptree[pare->neibs[
    }
break;
```

```
535  }else{
536      if(symtable[pare->iname].type!=gptree[pare->neibs[1]].stype){
537          printf("%s:%d:%d: error: conversion from '%s' to '%s' requested\n",
538          fn,gptree[pare->neibs[0]].sp.line,gptree[pare->neibs[0]].sp.column,
539          gptree[pare->neibs[1]].stype.c_str(),symtable[pare->iname].type.c_str());
540      }else{
```

仅整型变量才能参与算术运算。对加减乘除等算术运算，额外检查其两边的操作数是否为整型变量。下图中以加法为例。

```
if(gptree[pare->neibs[0]].sname=="+"){
    if(pare->itype!="int"||gptree[pare->neibs[1]].stype!="int"){
        printf("%s:%d:%d: note:  only int operands can participate in arithmetic ope
         fn, gptree[pare->neibs[1]].sp.line,gptree[pare->neibs[1]].sp.column);
    }
    id_val.push_back(id_val[pare->iaddr]+id_val[gptree[pare->neibs[1]].saddr]);
    gptree[pare->neibs[3]].itype = "int";
}
```

检查未声明的变量。在使用变量（标识符）时，首先检查符号表中是否有该表项，若没有，说明变量未声明，报错。

```
case 3:
    if(!symtable.count(gptree[pare->neibs[0]].sname)){
        printf("%s:%d:%d: error: '%s' was not declared in this scope\n",
        fn, gptree[pare->neibs[0]].sp.line,gptree[pare->neibs[0]].sp.column,gptree[p
    }else{
        gptree[pare->neibs[0]].stype = symtable[gptree[pare->neibs[0]].sname].type;
        gptree[pare->neibs[0]].saddr = symtable[gptree[pare->neibs[0]].sname].addr;
    }
    pare->stype = gptree[pare->neibs[0]].stype;
    pare->sp = gptree[pare->neibs[0]].sp;
    pare->saddr = gptree[pare->neibs[0]].saddr;
break;
```

```
case 13:
    if(!symtable.count(pare->iname)){
        printf("%s:%d:%d: error: '%s' was not declared in this scope\n",
         fn, pare->ip.line,pare->ip.column,
         pare->iname.c_str());
    }else{
```

程序中所有变量均不能重名。在执行声明语句时，首先检查声明的标识符是否已在符号表中，若是，则说明此次声明为重复声明，报错。

```
case 1:// check and fill the symbol table, check type
    if(symtable.count(gptree[pare->neibs[0]].sname)){
        printf("%s:%d:%d: error: conflicting declaration '%s %s'\n",
         fn, gptree[pare->neibs[0]].sp.line,
         gptree[pare->neibs[0]].sp.column,
         gptree[pare->neibs[0]].stype.c_str(),
         gptree[pare->neibs[0]].sname.c_str());
    }else{
```

```
case 2:
    gptree[pare->neibs[1]].stype="struct";
    if(symtable.count(gptree[pare->neibs[1]].sname)){
        printf("%s:%d:%d: error: conflicting declaration '%s %s'\n",
         fn, gptree[pare->neibs[1]].sp.line,
         gptree[pare->neibs[1]].sp.column,
         gptree[pare->neibs[1]].stype.c_str(),
         gptree[pare->neibs[1]].sname.c_str());
    }else{
```

因为过程/函数名为标识符，而声明标识符时会检查是否重复声明，因此也就保证了过程/函数仅能定义一次。

因为没有设计定义过程/函数的文法，自然有过程/函数不可嵌套定义。

定位源代码中的错误位置。通过在分析树结点与符号表中传递输入流中携带的位置信息，来实现错误位置显示。涉及此功能的代码太过分散，这里不在展示。

检查结构体中域是否与变量重名。可直接通过查看符号表中是否已有该域名来实现，与检查变量是否重复声明相同。

检查不同结构体中域是否重名。通过检查专门为结构体中的域设计的 map 结构 struct_table，来查看是否已有其它结构体中声明了这个域。

```
if(symtable.count(pare->iname)&&symtable[pare->iname].type=="struct"){
    if(struct_table.count(gptree[pare->neibs[0]].sname)){
        printf("%s:%d:%d: note: conflicting field declaration '%s %s' with struct",
            fn, gptree[pare->neibs[0]].sp.line,
            gptree[pare->neibs[0]].sp.column,
            gptree[pare->neibs[0]].stype.c_str(),
            gptree[pare->neibs[0]].sname.c_str());
        for(auto it=struct_table[gptree[pare->neibs[0]].sname].begin();
                it!=struct_table[gptree[pare->neibs[0]].sname].end();++it){
            cout << ' ' << *it;
            if(it!=--struct_table[gptree[pare->neibs[0]].sname].end())printf(",");
        }printf("!\n");
    }
    struct_table[gptree[pare->neibs[0]].sname].insert(pare->iname);
}else{
```

4. 程序说明：符号表说明、语法分析树构造过程

为了能够判断结构体中域是否与变量重名，以及不同结构体的域是否重名，将构建符号表的工作后移，使之与语法分析、语义分析一同完成。

符号表结构如下：

字段	类型	说明
name	string	标识符的名字
type	string	标识符的类型，缺省值为“unknown”
p	pos	标识符在源代码中出现的位置（行，列）
val	int	标识符的值（如果有）
addr	string	存储标识符值的地址

语法分析树构造过程与 Syntax Analyzer 中相同，只是修改了文法，在文法中嵌入了语义动作。此外，在构造语法分析树时,还附加了一些可以顺便执行的动作，包括产生式展开时的动作和匹配到终结符时的动作，这些动作用于传递某些继承属性以及把输入流中的信息加载到树上。

```
if(X.first=="T"){// shortcut for T
    notp[2].type=notp[1].first;
}
if(X.first=="decs"){// shortcut for X
    notp[1].type=X.type;
    notp[2].type=X.type;
}
if(X.first=="dec'"&&notp.size()>1){
    notp[2].type=X.type;
}
if(X.first=="dec"){
    notp[0].type=X.type;
}
```

上图中为产生式展开时顺带执行的动作。

```
if(X.first==a->type){// terminal symbal matched
    St.pop();
    if(X.first=="id"){
        gptree[X.second].sname = a->val;
        gptree[X.second].stype=X.type;
    }
    if(X.first == "decimal"){
        gptree[X.second].sval = atoi(a->val.c_str());
        gptree[X.second].saddr=id_val.size();// new addr
        gptree[X.second].stype = "int";
        id_val.push_back(gptree[X.second].sval);
    }
    gptree[X.second].sp=a->p;
    ++a;
```

上图中为匹配到终结符时顺带执行的动作。

对于明确用数字标号标出的嵌入到语法树中的语义动作，则是把标号视作特殊的非终结符，并令这些标号推出空串，以免其影响到 FIRST 集和 FOLLOW 集的正常求取，这些标号在语法树中就是一个以数字为名的结点。在生成完语法树后，再专门深度优先遍历一遍语法树，在遇到标号结点时，执行该标号对应的动作语句。

```
void dfs(node& cur){
    int act=atoi(cur.sname.c_str());
    if(act){
        actions(act,cur);
    }
    for(int i=0;i<cur.neibs.size();++i){
        dfs(gptree[cur.neibs[i]]);
    }
}
```

上图中为深度优先遍历语法树。

上图中为语义动作的实现。

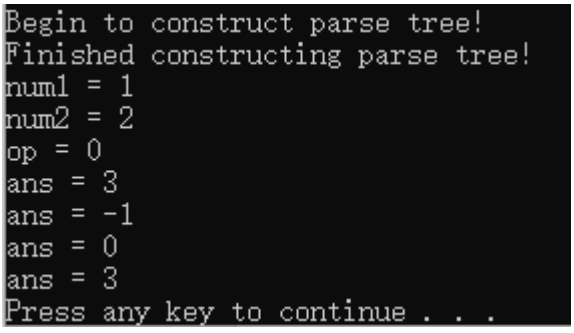
三、结果及分析和源程序调试过程

1、测试样例及结果展示：

1）简易计算器

```
//输入数据 num1,num2,op，根据 op 确定操作进行运算，最后输出运算结果 ans
int num1=1,num2=2,op=0,ans;
//get(num1,num2,op);
if(op==0)
{
    ans = num1 + num2;
};
if(op==1)
{
    ans = num1 - num2;
};
if(op==2)
{
    ans = num1 & num2;
};
if(op==3)
{
    ans = num1 | num2;
};
put(ans);
```

程序运行结果如图所示。语法分析树参见附件中的图片“简易计算器.jpeg”。本语法分析树中嵌入了语义动作，但因为语义动作过长，全部绘制在语法树结点中展示效果不佳，因此用语义动作的标号表示该语义动作对应的结点，可在前面给出的标号与语义动作的对应表中找到相应的语义动作。



图中运算结果全部正确。由于没有实现 get 和 put 函数，也没有实现控制流语句，因此，在声明变量时为它们赋了初值，并且所有 if 块中的语句都被顺序执行。

2) 跑马灯

```
//循环输入 op，改变输出结果 out，输入 0 则结束程序
int num0,num1,out,op;
num1 = 3333;
num2 = 6666;
num3 = 9999;
op = 1;
while(op>0)
{
    if(op==1)
    {
         out = num1;
        };
    if(op==2)
    {
         out = num2;
    };
    if(op==2)
    {
         out = num3;
    };
    put(out);
    get(op);
};
```

程序运行结果如图所示。生成的语法分析树参见附件中的图片“跑马灯.jpeg”。关于树的相关解释可参见前面测试用例中的描述。

```
Begin to construct parse tree!
Finished constructing parse tree!
num1 = 3333
inputs.c:4:1: error: 'num2' was not declared in this scope
inputs.c:5:1: error: 'num3' was not declared in this scope
op = 1
out = 3333
inputs.c:15:16: error: 'num2' was not declared in this scope
inputs.c:15:14: error: conversion from 'unknown' to 'int' requested
inputs.c:19:16: error: 'num3' was not declared in this scope
inputs.c:19:14: error: conversion from 'unknown' to 'int' requested
Press any key to continue . . .
```

图中结果正确。由于没有实现 get 和 put 函数，也没有实现控制流语句，因此所有 if 块均被顺序执行，while 循环只被顺序执行一次。 而 num2 和 num3 在代码中没有声明，因此会报错，并且对应赋值语句不被执行。

修改该测试样例，在第一行中声明 num2，num3，使之不再存在错误。修改后的文件可在附件中找到，名为 inputs_revised.c。修改后样例的测试效果如下。

```
Begin to construct parse tree!
Finished constructing parse tree!
num1 = 3333
num2 = 6666
num3 = 9999
op = 1
out = 3333
out = 6666
out = 9999
Press any key to continue . . .
```

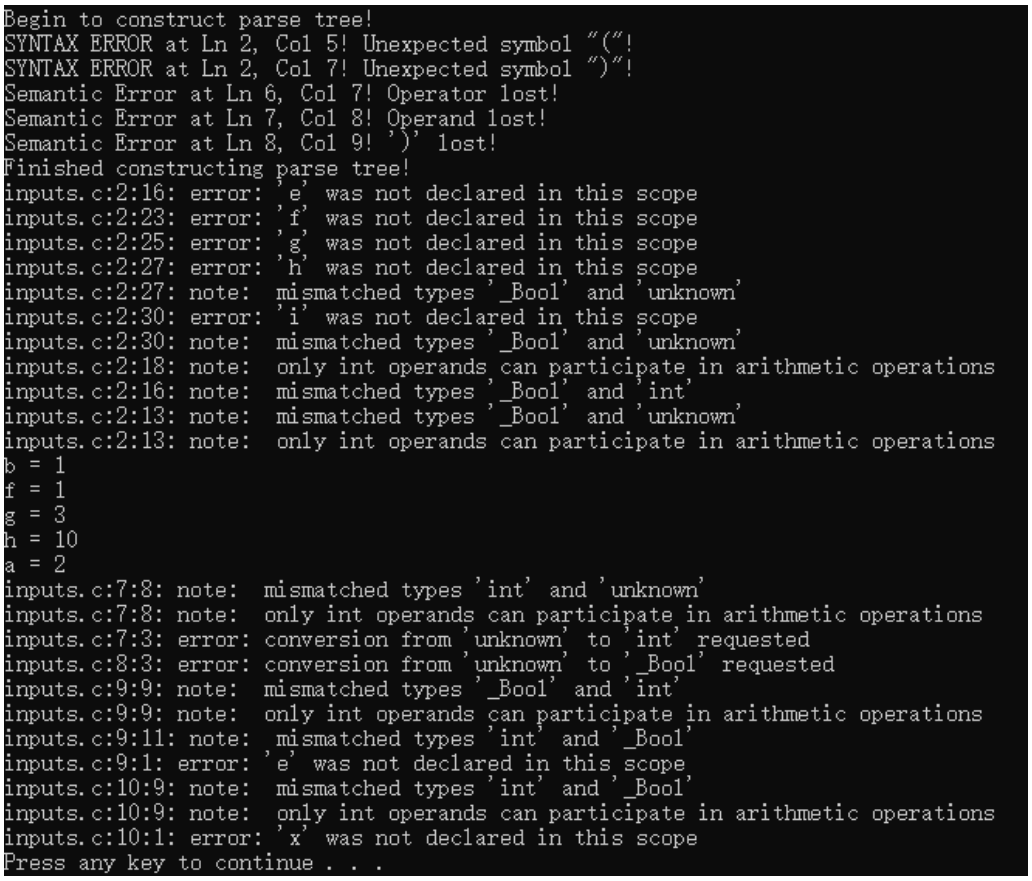
修改后的结果全部正确。修改后的语法分析树图片参见“跑马灯_revised.jpeg”。

2、自定义测试样例(包含自定义的词法错误类型):

样例 1：

```
1   _Bool c,d;
2   int (a),b=c+(d|e*!(c!=f>g&h||i));
3   int f=1;
4   int g=f+2;
5   int h=f+g*4-3;
6   a = 2 3;
7   b = (2+);
8   c = (2+3;
9   e = a+c*b|d;
10  x = 1+b+c;
```

该样例包含了所有词法错误类型，类型检查以及整型计算器的实现。执行结果如下图。



图中报错信息和计算结果全部正确。可以看到，在构建语法分析树时，就检查出了位于 6、7、8 行的三类“丢失”错误。后面又在遍历语法分析树时，通过嵌入语义动作，实现了对未声明变量的检查，两个运算数类型不匹配的检查，参与算数运算的操作数必须为整型数的检查，赋值语句两边的变量类型不一致的检查等。此外，可以看到几个变量的计算与赋值全部正确，其中 b 的值来自布尔运算。而存在错误的语句的执行结果没有意义，可以忽略。

该样例的语法树见“样例 1.jpeg”。也可以根据样例 1 文件夹中的 Tree.txt 在网站 http://www.webgraphviz.com/ 上绘制。

样例 2：

```
1  _Bool m;
2  struct x{
3      int i,j,l,m;
4  };
5  struct y{
6      _Bool k,i,l;
7      struct z{
8          int i,j,k;
9      };
10 };
11 _Bool c=i+j*k-m;
```

该样例主要测试结构体相关检查。包括检查结构体中域是否与变量重名，不同结构体中域是否重名。测试结果如下图。

```
Begin to construct parse tree!
Finished constructing parse tree!
inputs.c:3:16: error: conflicting declaration 'int m'
inputs.c:6:14: note: conflicting field declaration '_Bool i' with struct x!
inputs.c:6:16: note: conflicting field declaration '_Bool l' with struct x!
inputs.c:8:15: note: conflicting field declaration 'int i' with struct x, y!
inputs.c:8:17: note: conflicting field declaration 'int j' with struct x!
inputs.c:8:19: note: conflicting field declaration 'int k' with struct y!
inputs.c:11:9: error: 'i' was not declared in this scope
inputs.c:11:11: error: 'j' was not declared in this scope
inputs.c:11:13: error: 'k' was not declared in this scope
inputs.c:11:13: note:  only int operands can participate in arithmetic operations
inputs.c:11:11: note:  mismatched types 'unknown' and 'int'
inputs.c:11:11: note:  only int operands can participate in arithmetic operations
inputs.c:11:15: note:  mismatched types 'int' and '_Bool'
inputs.c:11:15: note:  only int operands can participate in arithmetic operations
inputs.c:11:8: error: conversion from 'int' to '_Bool' requested
Press any key to continue . . .
```

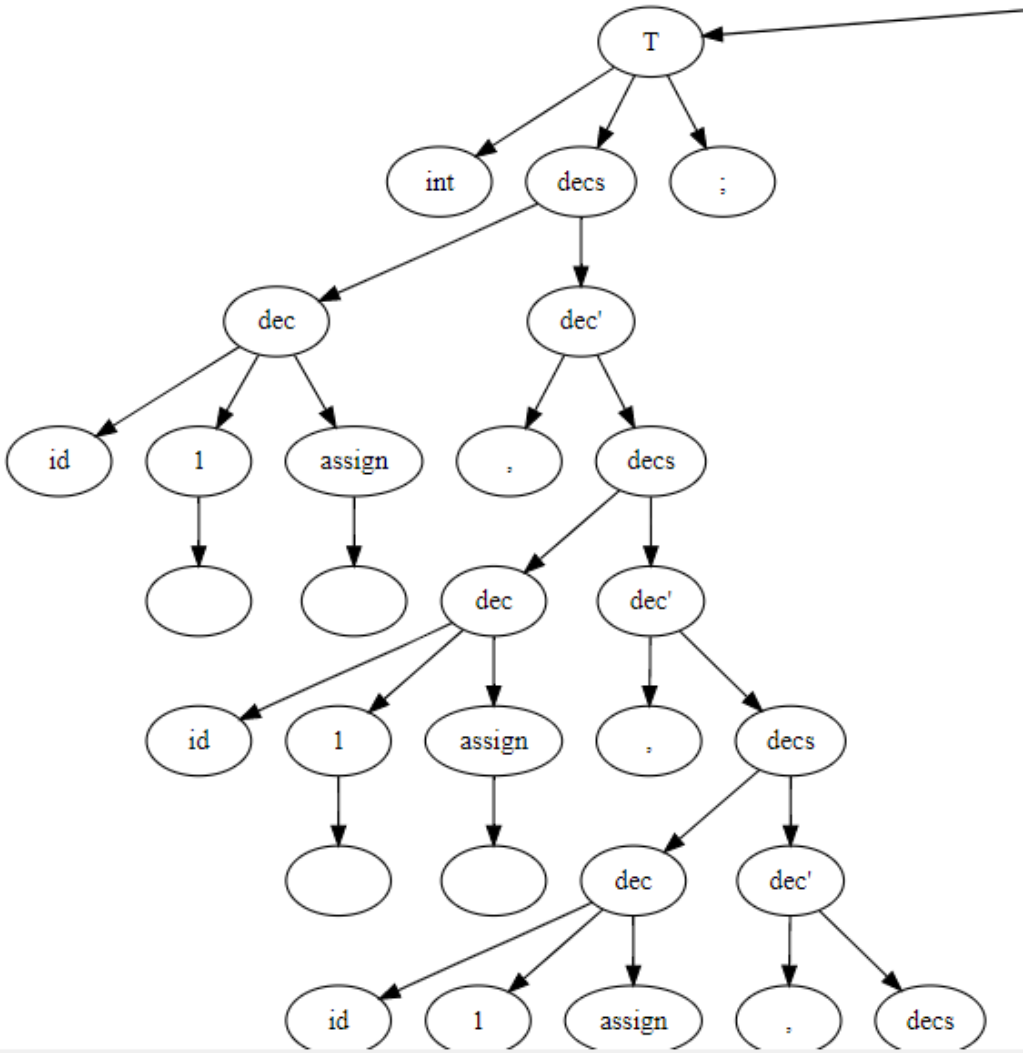
图中报错信息完全符合预期。3 行 16 列处结构体 x 中声明的域名 m 与外部变量 m 冲突。而第 6 行声明的 i、l 等域名则已在结构体 x 中声明过。类似地，第 8 行结构体 z 中存在同样的重复声明，由于 i 在结构体 x、y 中均声明过，因此报错信息把 x、y 两个结构体都列举了出来。由于结构体的域名属于结构体内部的局部变量，在结构体外无法直接使用结构体内的变量，因此第 11 行报 i、j、k 等变量未声明的错。未声明变量默认类型未“unknown”，因此后续运算存在类型不匹配等问题。

该样例的语法树见“样例 2.png”。也可以根据样例 2 文件夹中的 Tree.txt 在网站 http://www.webgraphviz.com/上绘制。

说明：

由于语法分析树过大，附件中分析树的图片不够清晰。如果要查看清晰的图片，烦请复制与分析树图片在同一目录下的 tree.txt 文件中的内容，粘贴到 http://www.webgraphviz.com/网站中的输入框，在线绘制出语法分析树。

下面给出一小部分分析树的截图实例，以说明采用标号表示语义动作的方法。



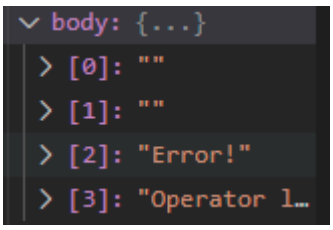
图中，截取的部分树的形式基本与 Syntax Analyzer 中一致，但多了名字为数字的结点，这个数字就是语义动作的标号，如显示为数字 1 的结点，就对应标号为 1 的语义动作。这样做可以避免把冗长的语义动作放到结点中，使得绘制出来的树相对简洁。

四、总结

1、遇到的问题及解决办法

问题 1：检测“丢失运算符”错误时，不能生成应有的报错信息。

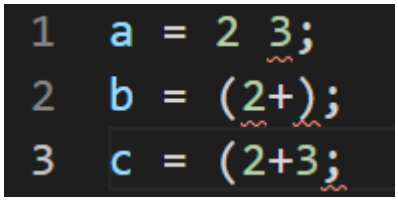
解决方案：检查发现，报错信息添加有误，声明报错信息字符串 vector 时预申请了两个空元素，后续又通过 push_back 添加报错信息，导致报错信息被添加到了 vector 中下标为 2、3 的位置。



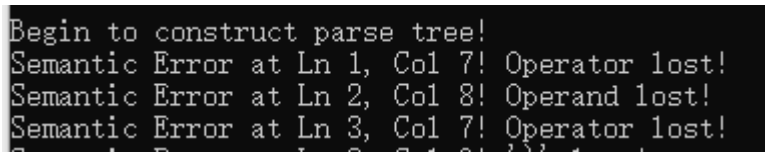
修改后，问题得到解决。

问题 2：出现误报“丢失运算符”错误的问题。

在如下测试样例中，3 行 7 列有运算符‘+’。



而误报“运算符丢失”。



解决方案：

检查发现，词法分析时，把+3 分析成了正 3，导致+3 成为一个十进制数，从而缺失运算符。修改词法，使之不再把正负号和数字合在一起识别，修改后问题解决。

问题 3：误将函数调用时的函数名检测成未声明的标识符。

解决方案：虽然 C 语言标准中函数名确实也需要预先声明，但这里只用到 gets 和 puts 函数，且不支持宏，因此应设计成这两个函数默认已被声明更为合理。

修改文法，单独处理 CC 前面 id 的检查。修改后问题解决。

问题 4：不能检测到表达式中运算符两侧操作数类型不匹配的问题。

解决方案：检查发现，在语义动作 3 中，id 的 type 和 p 应由查符号表得到。而符号表中没能正确存储 id 的相关信息。而这是因为自顶向下生成语法分析树时，没能成功把这些信息保存到 id 对应的结点里。在终结符 id 对应的语义动作中把其类型复制到对应树中结点，修改后问题解决。

问题 5：标识符类型没能沿语法分析树向上传递上去。

```
J'.type= unknown
6
I'.type= unknown
5
I.type=
6
H'.type=
5
H.type=
6
M'.type=
inputs.c:9:28: note:  mismatched types '_Bool' and ''
inputs.c:9:32: error: 'h' was not declared in this scope
```

解决方案：检查发现，从非终结符 I 开始，类型为空。

```
inputs.c:9:21: error: 'e' was not declared in this scope
6
K.type= _Bool
6
J.type= _Bool
6
I.type= _Bool
inputs.c:9:28: error: 'f' was not declared in this scope
6
K.type=
6
J.type=
inputs.c:9:30: error: 'g' was not declared in this scope
6
K.type=
6
J.type=
5
I.type=
6
I.type=
```

进一步检查发现，从非终结符 K 开始类型就为空，据此发现是因为未声明标识符‘f‘的类型为空。将文法符号结构 grmsym 中的 type 初始化为“unknown”,问题解决。

```
inputs.c:9:21: error: 'e' was not declared in this scope
6
K.type= _Bool
6
J.type= _Bool
6
I.type= _Bool
inputs.c:9:28: error: 'f' was not declared in this scope
6
K.type= unknown
6
J.type= unknown
inputs.c:9:30: error: 'g' was not declared in this scope
6
K.type= unknown
6
J.type= unknown
5
I.type= unknown
6
I.type= unknown
5
H.type= unknown
6
H.type= _Bool
inputs.c:9:32: error: 'h' was not declared in this scope
```

问题 6：类型匹配报错位置不对。

```
inputs.c:3:1: error: 'e' was not declared in this scope
inputs.c:1:7: note:  mismatched types '_Bool' and 'int'
inputs.c:2:7: note:  mismatched types 'int' and '_Bool'
inputs.c:2:9: note:  mismatched types 'int' and '_Bool'
inputs.c:4:1: error: 'x' was not declared in this scope
inputs.c:2:7: note:  mismatched types 'int' and '_Bool'
inputs.c:1:7: note:  mismatched types 'unknown' and 'int'
inputs.c:9:11: error: conflicting declaration 'int a'
inputs.c:9:14: error: conflicting declaration 'int b'
```

解决方案：位置被和类型采用完全同样的方式传递，因此最终的位置是变量首次声明的位置，因而出错。修改语义动作 3 中传递位置的语句，使之传递当前产生式体中标识符 id 的位置，而非传递符号表中该 id 的位置。修改后问题解决。

```
inputs.c:3:1: error: 'e' was not declared in this scope
inputs.c:3:9: note:  mismatched types '_Bool' and 'int'
inputs.c:3:7: note:  mismatched types 'int' and '_Bool'
inputs.c:3:11: note:  mismatched types 'int' and '_Bool'
inputs.c:4:1: error: 'x' was not declared in this scope
inputs.c:4:9: note:  mismatched types 'int' and '_Bool'
inputs.c:4:7: note:  mismatched types 'unknown' and 'int'
inputs.c:9:11: error: conflicting declaration 'int a'
inputs.c:9:14: error: conflicting declaration 'int b'
```

问题 7：对表达式进行类型检测时，同级运算符的运算顺序错误地执行为从右向左。

解决方案：语义动作 5 的设置有问题，例如$I' \rightarrow +JI'$ 5，5 检测的是 J 和 I’的类型是否匹配，实际上应该检测 J 和加号前面的变量类型是否匹配。发现检查策略有问题，于是重新设计检查策略，从 SDT 中消除左递归。

问题 8：运算符的类型检查报错中，标识符的类型不对。

```
1  int a,b;
2  _Bool c,d;
3  e = a+c*b|d;
```

图中测试样例的报错如下。

```
inputs.c:3:1: error: 'e' was not declared in this scope
inputs.c:3:9: note:  mismatched types 'unknown' and 'int'
```

_Bool 型变量 c 的类型被误报为“unknown”。

解决方案：检查发现，代码中编号为 8 的语义动作实现有误。

```
case 8:
    if(pare->stype!=gptree[pare->neibs[1]].stype){
        printf("%s:%d:%d: note:  mismatched types '%s' and '%s'\n",
        fn, gptree[pare->neibs[1]].sp.line,gptree[pare->neibs[1]].sp.colu
        pare->stype.c_str(),gptree[pare->neibs[1]].stype.c_str());
    }
```

上图代码中误使用父结点的综合属性。如下修改为继承属性。

```
e 8:
 if(pare->itype!=gptree[pare->neibs[1]].stype){
     printf("%s:%d:%d: note:  mismatched types '%s' and '%s'\n",
     fn, gptree[pare->neibs[1]].sp.line,gptree[pare->neibs[1]].sp.columr
     pare->itype.c_str(),gptree[pare->neibs[1]].stype.c_str());
 }
```

修改后问题仍未解决。即父结点的继承属性没能正确地把类型传递过来。进一步检查发现语义动作 5 的代码实现错误，结点访问错误。

```
case 5:
    gptree[pare->neibs[3]].itype = gptree[pare->neibs[1]].stype;
    gptree[pare->neibs[3]].ival = gptree[pare->neibs[1]].sval;
    gptree[pare->neibs[3]].ip = gptree[pare->neibs[1]].sp;
break;
```

修改如下。

```
case 5:
    gptree[pare->neibs[2]].itype = gptree[pare->neibs[0]].stype;
    gptree[pare->neibs[2]].ival = gptree[pare->neibs[0]].sval;
    gptree[pare->neibs[2]].ip = gptree[pare->neibs[0]].sp;
break;
```

修改后问题解决。

```
inputs.c:3:1: error: 'e' was not declared in this scope
inputs.c:3:9: note:  mismatched types '_Bool' and 'int'
```

问题 9：整型常量类型错误。

```
1  int a,b;
2  _Bool c,d;
3  e = a+c*b|d;
4  x = 1+b+c;
```

上图测试用例的第四行出现如下报错。

```
inputs.c:4:7: note:  mismatched types 'unknown' and 'int'
```

即对整型常量 1 的类型识别出现错误。

```
519             if(X.first == "decimal"){
520                 gptree[X.second].sval = atoi(a->val.c_str());
521                 gptree[X.second].sp=a->p;
522                 gptree[X.second].saddr=id_val.size();
523                 id_val.push_back(gptree[X.second].sval);
524             }
```

检查发现，LL1 检测到整型常量时没有把它的类型初始化为 int。

```
398     case 7:
399         pare->sval = gptree[pare->neibs[0]].sval;
400         pare->sp = gptree[pare->neibs[0]].sp;
401         pare->saddr = gptree[pare->neibs[0]].saddr;
```

且语义动作 7 中没有将整型常量的类型传递给其父结点。修改如下。

```
519             if(X.first == "decimal"){
520                 gptree[X.second].sval = atoi(a->val.c_str());
521                 gptree[X.second].sp=a->p;
522                 gptree[X.second].saddr=id_val.size();
523                 gptree[X.second].stype = "int";
524                 id_val.push_back(gptree[X.second].sval);
525             }
```

```
case 7:
    pare->stype = gptree[pare->neibs[0]].stype;
    pare->sval = gptree[pare->neibs[0]].sval;
    pare->sp = gptree[pare->neibs[0]].sp;
    pare->saddr = gptree[pare->neibs[0]].saddr;
```

修改后问题解决。

```
inputs.c:4:1: error: 'x' was not declared in this scope
inputs.c:4:9: note:  mismatched types 'int' and '_Bool'
inputs.c:4:9: note:  only int operands can participate in arithmetic operations
```

不再出现之前的报错。

问题 10:出现位置不明的报错。

```
inputs.c:0:0: note:  mismatched types 'int' and 'unknown'
inputs.c:0:0: note:  only int operands can participate in arithmetic operations
```

解决方案：位置信息为 0，0，应该是没有被赋值。即存在语义动作 8 中父结点继承属性位置没有被赋值的情况。

```
1   a = 2 3;
2   b = (2+);
3   c = (2+3;
```

修改测试用例，将几类缺失错误的测试放到最前面，发现上述位置不明的报错也相应移到了前面，即这些报错是针对测试缺失错误的测试用例的，而缺失错误语法上本身就有错，因此对其进行语义分析是没有意义的，可以不予理会。在检测操作数丢失的地方把右括号的位置赋给缺失操作数所在的结点。

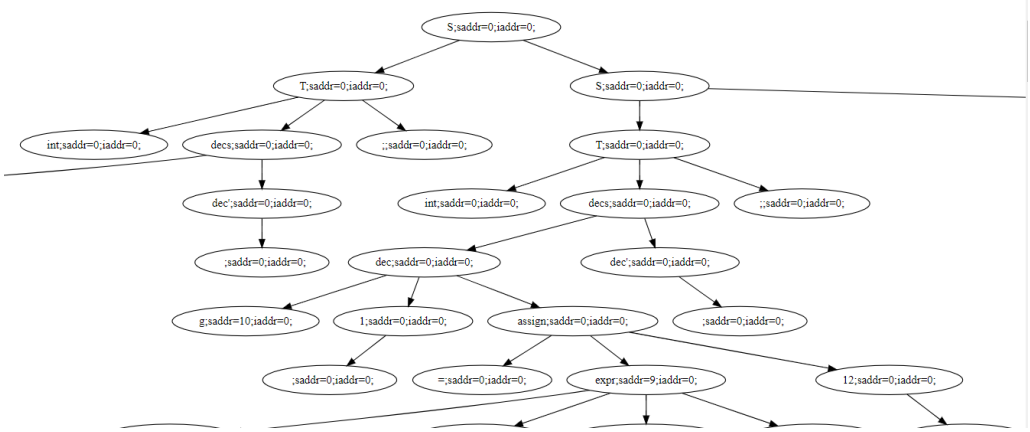
```
if(body[1]=="Operand lost!"){
    gptree[X.second].sp=a->p;
    St.pop();
}
```

修改后问题解决。

```
inputs.c:1:1: error: 'a' was not declared in this scope
inputs.c:2:1: error: 'b' was not declared in this scope
inputs.c:2:8: note:  mismatched types 'int' and 'unknown'
inputs.c:2:8: note:  only int operands can participate in arithmetic operations
inputs.c:3:1: error: 'c' was not declared in this scope
```

问题 11：计算过程中表达式的地址与预期不一致。

解决方案：由于计算器功能涉及的树中结点过多，为便于调试，将每个结点最终的地址值可视化绘制出来，如下图。由于图片过大无法截全，这里只截取了一小部分，意在说明为解决问题所用到的绘图方式。



通过检查上述形式的分析树，发现语义动作 9 和 6 中存在错误，9 中为父结点地址赋值的子结点引用错误，9 和 6 中都错用了子结点的继承属性来给父结点的综合属性赋值。修改后问题解决。

问题 12：位置属性的传递仍然有误，仍有报错信息位置不对。

```
inputs.c:0:0: note:  only int operands can participate in arithmetic operations
```

解决方案：仿照问题 11 中的解决方法，把位置信息可视化绘制出来，以便寻找问题所在。检查发现终结符括号的位置没有成功添加。发现 LL1 中只对 id 和 decimal 添加了位置信息，而忽视了其它终结符。此外，没有处理到非终结符 K 的所有可能的产生式的位置信息传递。修改后问题解决。

```
inputs.c:1:18: note:  only int operands can participate in arithmetic operations
```

问题 13：从报错中发现类型传递存在问题。

测试样例如下。

```
int (a),b=c+(d|e*!(c!=f>g&h||i));
```

```
inputs.c:14:15: error: conversion from 'int' to '' requestedinputs.c:17:7: error: 'x' was not declared in this scope
```

之前声明过 c 为_Bool 型变量，报错信息中的两处类型都不对。

解决方案：仍然采取将类型可视化绘制出来的方式以便检查错误。绘图发现，这是变量重复声明所导致的。为符号表中的 type 设置默认值“unknown”，问题解决。

```
inputs.c:13:15: error: conversion from 'int' to 'unknown' requested
```

图中不再出现‘’这样的空类型，至于类型的合理性则无需考虑，因为重复声明的变量本身就没有意义。