语义分析器实现

课程项目

主要内容：实验内容是为一个小型的类C语言（C--）实现一个编译器。如果你顺利完成了本实验任务，那么不仅你的编程能力将会得到大幅提高，而且你最终会得到一个比较完整的、能将C--源代码转换成MIPS汇编代码的编译器，所得到的汇编代码可以在SPIM Simulator上运行。

课程项目总共分为五个阶段：词法和语法分析、语义分析、中间代码生成、目标代码生成以及中间代码优化。每个阶段的输出是下一个阶段的输入，后一个阶段总是在前一个阶段的基础上完成。其中，目标代码生成以及中间代码优化均基于第三次中间代码生成。

实验助教：

燕言言　　QQ：2214871526

邮箱：yanyanthunder@foxmail.com

何天行　　QQ：976792132

邮箱：976792132@qq.com

提纲

- 实验一
  - 成绩分布
- 实验二
  - 实验内容
  - 实验设计
    - 符号表表示
    - 类型表示
    - 语法树遍历
    - 符号表填充
    - 语义检查
  - 注意事项

实验内容

目标

实验二的任务是在词法分析和语法分析程序的基础上编写一个程序，对C--源代码进行语义分析和类型检查，并打印分析结果。与实验一不同的是，实验二不再借助已有的工具，所有的任务（符号表、数据类型一致性）都必须手写代码来完成

实验二需要设计诸如符号表、变量类型等数据结构，实现正确、高效地实现语义分析的各种功能

实验二依赖于实验一实现的词法语法分析器，后续实验也会用到实验二的代码

实验内容

实验内容（必选项）

- C—假设
  - **语义分析：**主要包含函数、结构体、数组定义使用检查
  - **类型检查：**int、float等基础类型一致性检查，结构体类型等价判断
- 错误类型
  - 17种错误类型检测
- 输入格式
  - 程序的输入是一个包含C--源代码的文本文件，该源代码中可能会有语义错误。程序需要能够接收一个输入文件名作为参数

实验内容

输出格式

- 对于那些没有语义错误的输入文件，程序不需要输出任何内容
- 对于那些存在语义错误的输入文件，程序应当输出相应的错误信息，这些信息包括错误类型、出错的行号以及说明文字

  Error type [错误类型] at Line [行号]: [说明文字].

- 错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准
- 输入文件中可能包含一个或者多个错误（但每行最多只有一个错误），程序需要将它们全部检查出来

如果源程序里有错而程序没有报错或报告的错误类型不对，又或者源程序里没有错但程序却报错，都会影响实验评分

实验内容

- **C—假设（构建符号表和类型系统）**
  - **假设1：**整型（int）变量不能与浮点型（float）变量相互赋值或者相互运算（考虑赋值赋值语句、运算符表达式，记录变量类型）
  - **假设2：**仅有int型变量才能进行逻辑运算或者作为if和while语句的条件；仅有int型和 float型变量才能参与算术运算（考虑条件表达式、运算符表达式中变量类型）
  - **假设3：**任何函数只进行一次定义，无法进行函数声明（函数定义后添加到符号表中）
  - **假设4：**所有变量（包括函数的形参）的作用域都是全局的（变量的作用域范围）

实验内容

- **C—假设（构建符号表和类型系统）**
  - **假设5：**结构体间的类型等价机制采用名等价（Name Equivalence）的方式（类型系统定义以及类型一致性判定）
  - **假设6：**函数无法进行嵌套定义（函数都是全局的，很多语言支持闭包或者内部函数定义，C—不支持这些）
  - **假设7：**结构体中的域不与变量重名，并且不同结构体中的域互不重名（考虑将域添加到符号表中，结构体合法定义必须满足域的名称与符号表中已有的符号名不冲突）

实验内容

- 错误类型（符号相关）
  - **错误类型1：** 变量在使用时未经定义
  - **错误类型2：** 函数在调用时未经定义
  - **错误类型3：** 变量出现重复定义，或变量与前面定义过的结构体名字重复
  - **错误类型4：** 函数出现重复定义
  - **错误类型6：** 赋值号左边出现一个只有右值的表达式

考虑使用符号表以及增加类型一致性规则

实验内容

错误类型（类型相关）

- **错误类型5：**赋值号两边的表达式类型不匹配
- **错误类型7：**操作数类型不匹配或操作数类型与操作符不匹配
- **错误类型8：** return语句的返回类型与函数定义的返回类型不匹配
- **错误类型9：**函数调用时实参与形参的数目或类型不匹配
- **错误类型10：**对非数组型变量使用“[…]”（数组访问）操作符
- **错误类型11：**对普通变量使用“(…)”或“()”（函数调用）操作符
- **错误类型12：**数组访问操作符“[…]”中出现非整数

类型判断需要考虑到类型间一致性，需要制定类型相容性规则。或者类型一致性判断由特定的类型系统或者模块负责，可以动态扩充会更好

实验内容

错误类型（结构体相关）

考虑使用符号表，比如每个结构体可以使用单独的符号表。数组的话要保存维数和基类型

- **错误类型13：**对非结构体型变量使用“.”操作符
- **错误类型14：**访问结构体中未定义过的域
- **错误类型15：**结构体中域名重复定义，或在定义时对域进行初始化（struct P {int age = 18;}）
- **错误类型16：**结构体的名字与前面定义过的结构体或变量的名字重复
- **错误类型17：**直接使用未定义过的结构体来定义变量
  - 关于数组类型的等价机制，同C语言一样，只要数组的基类型和维数相同我们即认为类型是匹配的
  - 允许类型等价的结构体变量之间的直接赋值，对应的域相互赋值
  - 每个匿名的结构体类型我们认为均具有一个独有的隐 藏名字，以此进行名等价判定

实验内容

实验内容（可选内容）

- **修改假设3（函数进行了声明，但没有被定义）**
  - 函数除了在定义之外还可以进行声明，声明一致的情况下可重复出现
  - 函数定义必须出现且不可以重复
  - 错误类型
    - ✓ **错误类型18：**函数进行了声明，但没有被定义
    - ✓ **错误类型19：**函数的多次声明互相冲突（即函数名一致，但返回类型、形参数量 或者形参类型不一致），或者声明与定义之间互相冲突。
    - ✓ 由于C--语言文法中并没有与函数声明相关的产生式，因此需要先对该文法进行适当修改。在修改的时候要留意，你的改动应该以不影响其它错误类型的检查为原则

实验内容

实验内容（可选内容）

- **修改假设4**（所有变量（包括函数的形参）的作用域都是全局的，即程序中所有变量均不能重名）
  - 变量的定义受可嵌套作用域的影响，外层语句块中定义的变量可在内层语句块中重复定义
  - 内层语句块中定义的变量到了外层语句块中就会消亡，不同函数体内 定义的局部变量可以相互重名
  - 错误类型
    - ✓ 在新的假设4下，完成错误类型1至17的检查

实验内容

实验内容（可选内容）

- **修改假设5**（结构体间的类型等价机制采用名等价（Name Equivalence）的方式）
  - 结构体间的类型等价机制采用按结构等价的方式
  - 在结构等价时不要将数组展开来判断
  - 错误类型
    - 在新的假设5下，完成错误类型1至17的检查

两个结构体类型

```
struct a {
  int x; float y;
}
```

和

```
struct b {
  int y; float z;
}
```

仍然是等价的类型

实验内容

- 实验二任务分配

选做分类	队伍编号	选做内容
选做内容一	29, 11, 43, 12, 1, 26, 44, 38, 32, 30, 5, 2, 3, 9, 22	修改假设3，C--支持函数定义和声明，定义不可重复
选做内容二	19, 28, 20, 21, 10, 31, 46, 17, 23, 45, 40, 36, 24, 18, 34, 47	修改假设4，C--变量的定义受可嵌套作用域影响
选做内容三	4, 37, 41, 6, 8, 35, 27, 39, 7, 14, 33, 42, 15, 13, 25, 16	修改假设5，C—结构体类型等价机制改为结构等价

实验设计——符号表表示

从符号表入手

- 左图所示的C--源码是符合文法定义的（实验二_ C– 语义分析器实现必做内容样例一），词法分析与语法分析都正确，那么这段代码是否可以成功运行呢？
  - 答案是否定的。参考答案给出的第四行中变量“j”未定义，而变量“i”是可以正常使用的。这些变量可以看做是程序中的符号，符号“i”是已经正确定义的，类型为int，初始值为0
  - 语义分析过程就是在词法语法分析基础上，遍历程序语法树，遇到ExtDef和Def语法单元时，就将其中包含的变量和函数符号信息添加到符号表

```
1  int main()
2  {
3      int i = 0;
4      j = i + 1;
5  }
```

C—源码

```
<source>: In function 'main':
<source>:4:5: error: 'j' undeclared (first use in this function)
j = i + 1;
^
<source>:4:5: note: each undeclared identifier is reported only once for each
function it appears in
Compiler returned: 1
```

Gcc 7.5.0报错信息

实验设计——符号表表示

从符号表入手

- 符号表中每一项都对应于程序中普通变量、函数、结构体、数组等中的某一个，这些符号的都有自己的符号名、类型等。因此，符号表中每一项都必须包含这些基本项，定义如左下示例所示
  - 符号表上最常见的操作是填表和查表操作，与符号表本身实现的方式无关，可以根据自己的喜好，选择线性链表、平衡二叉树、散列表（**open hashing**）
  - 有了符号表，我们可以为每一个符号名在表中找到一个位置（计算**hash**值）

```
1  // open hashing
2  struct HashNode_{
3      char *name;
4      Type type;
5      FieldList param;
6      struct HashNode_ *next;
7  };
8  HashNode *gTable[1024];
9  HashNode *sTable[1024];
```

符号表定义

```
1  // Time33 hash algorithm
2  int hashFunc(char *key){
3      unsigned int hash = 5381;
4      while(*key){
5          hash += (hash << 5 ) + (*key++);
6      }
7      return (hash & 0x7FFFFFFF) % 1024;
8  }
```

Hash函数

实验设计——符号表表示

- 从符号表入手
  - 符号表构建之后，我们需要频繁的插入及查找符号。符号表的插入操作是先根据符号名计算的**hash**值获取**key**，然后查找**key**所在的位置是否已经有数据，查找操作类似

```
 1  int insert(char *name,Type type){
 2      计算哈希值;
 3      构建新的符号并用name、type初始化;
 4      HashNode *node = gTable[hash];
 5      if(node==NULL){
 6          gTable[hash] = newnode;
 7          return 0;
 8      }else{
 9          循环遍历直到node->next为NULL,
10  插入;
11      }
12  }
```

符号表插入操作

```
 1  int check(char *name){
 2      计算hash值;
 3      for(HashNode *node = gTable[hash];
 4            node!=NULL; node = node->next){
 5          if(strcmp(node->name,name)==0){
 6              return 1;
 7          }
 8      }
 9      return 0;
10  }
```

符号表查询操作

实验设计——类型表示

类型表示

- C--类型包括基本类型（某些语言中称为primitive type）、复合类型（数组）、用户自定义类型（结构体）。因此，我们需要同时表示这几种类型。Kind表示类型种类、u表示类型信息、assign表示符号出现的位置

```
 1  struct Type_{
 2      enum {
 3          BASIC,      // variable
 4          ARRAY,      // array
 5          STRUCTURE,  // structure
 6          FUNCTION    // function
 7      } kind;
 8      union{
 9          int basic; // int(1), float(2)
10          struct {
11              Type elem; int size;
12          } array;
13          Structure structure;
14          Function function;
15      } u;
```

```
16    enum {
17          LEFT,     // left value
18          RIGHT,    // right value
19          BOTH      // left | right value
20      } assign;
21  };
```

类型统一表示Type

实验设计——类型表示

类型表示

- C--基本类型int和float可以简单表示，比如用常数表示，变量类型、数组类型、结构体类型表示更加复杂，这些类型本身可以嵌套其他类型

```
1  // 变量、参数、结构体的域
2  struct  FieldList_{
3      char *name;
4      Type type;
5      FieldList tail;
6  };
7  // 结构体类型，包含名称、域
8  struct Structure_{
9      char *name;
10     FieldList domain;
11 };
```

变量、结构体类型表示

```
1  // 函数类型，包括名称、行号、返回值类型、参数
2  struct Function_{
3          char *name;
4          int line;
5          Type type;
6          FieldList param;
7  };
```

函数类型表示

实验设计——类型表示

类型一致性

- 主要考察基本类型、复合类型、用户自定义类型一致性。基本类型一致需满足同为int或者float；数组一致性需满足对应的元素类型一致，维数不限制；必做要求结构体按名等价

```
 1  int Type_check(Type t1,Type t2){
 2      if (t1 == NULL && t2 == NULL)
 3          return 1; // 都为空一致
 4      if (t1->kind == BASIC)
 5          if (t1->u.basic == t2->u.basic)
 6              return 1; // 基本类型相同
 7          return 0;      // 基本类型不同
 8      if (t1->kind == ARRAY) // 逐个比对数组元素类型
 9          return Type_check(t1->u.array.elem, t2->u.array.elem);
10      if (t2->kind == STRUCTURE)
11          if (t2->kind != STRUCTURE)
12              return 0;
13          return t1->u.structure.name == t2->u.structure.name;
14
15  }
```

类型一致性检查

实验设计——类型表示

函数参数一致性

- 函数参数一致性需要依次提取每一个参数类型，并判断是否类型一致。只要有一对类型不一致，则不匹配

```
 1  int Param_check(FieldList p1,FieldList p2){
 2      while (p1 != NULL && p2 != NULL) {
 3          if (Type_check(p1, p2) == 0)
 4              return 0; // 参数类型不一致
 5          p1 = p1->tail;
 6          p2 = p2->tail; // 获取下一个参数
 7      }
 8      if (p1 == NULL && p2 == NULL)
 9          return 1;    // 检查结束，参数类型一致
10      return 0;        // 默认类型不一致
11  }
12
13
```

函数参数类型一致性检查

实验设计——语法树遍历

- 语义分析器入口
  - 在实验一词法分析、语法分析基础上，我们的分析器可以准确识别出程序中的词法错误、语法错误。对于不包含这两类的错误的源程序，我们需要将其转换成语法树，并用于语义分析

```
1  int main(int argc, char** argv) {
2      if (argc <= 1) {
3          return 1;
4      }
5      FILE* f = fopen(argv[1], "r");
6      if (!f) {
7          perror(argv[1]);    return 1;
8      }
9      yyrestart(f);
10     yyparse(); // 语法分析
11     if (!errorflag) { // 语法树正确构建
12         semantic_check(Root); // 语义分析
13     }
14     return 0;
15 }
```

Main函数定义

```
1  // 语义分析入口函数
2  void semantic_check(struct Node* Root){
3      初始化工作，如全局变量初始化；
4      // 遍历语法树中的ExtDefList节点
5      ExtDefList(Root->firstChild);
6      选做内容实现，如函数声明一致性；
7      善后工作，如内存清理；
8  }
```

语义分析入口

实验设计——语法树遍历

全局语法单元遍历

- C--文法规定ExtDefList产生式为ExtDef ExtDecList | ε 。因此，我们需要遍历ExtDef的语法树和递归遍历ExtDecList，直到语法树全部分析完
- ExtDef包含三个产生式，分别是全局变量定义ExtDecList、结构体定义Speicifier和Specifier FuncDec CompSt。ExtDef函数需要依次判断节点类型并调用对应的处理函数

```
1  void ExtDefList(struct Node *node){
2      判断节点是否为空;
3      // 遍历第一个节点
4      ExtDef(node->firstChild);
5      // 遍历兄弟节点
6      ExtDefList(node->firstChild->Sibc);
7  }
```

遍历ExtDefList单元

```
1  void ExtDef(struct Node* node){
2      判断节点是否为空;
3      获取node的specifier类型t;
4      获取node的其他节点信息sibc;
5      // 如果sibc是全局变量ExtDecList
6      ExtDecList(sibc, t);
7      // 如果sibc是结构体定义,t就是结构体类型
8      // 如果sibc是函数定义
9      提取函数参数，返回值类型t，函数名;
10     声明函数;
```

遍历ExtDef

实验设计——语法树遍历

局部语句块遍历

- C--文法规定CompSt表示由一对花括号括起来的语句块。语句块内部包含变量定义DefList、语句StmtList。StmtList由零个或多个Stmt组成，Stmt可以是表达式、CompStmt、返回语句、if、while等。后续语义分析过程中，需要分析表达式等。因此，我们先遍历CompSt语法树

```
1  void CompSt(struct Node *node, Type ntype){
2      判断节点是否为空;
3      // 遍历第一个节点
4      提取node孩子节点n;
5      if (n是DefList) {
6          遍历DefList(n, 0)
7      } else if (n是StmtList) {
8          遍历StmtList(n, ntype);
9      }
10 }
```

遍历CompSt单元

```
1  void Stmt(struct Node *node, Type ntype){
2      判断节点是否为空;
3      提取node的孩子节点n;
4      if (n是Exp) {
5          Exp(n);
6      } else if (n是CompSt) {
7          CompSt(n, ntype);
8      } else if (n是RETURN) {
9          // 返回值类型
10         Type expType = Exp(n的子节点);
       } else if (n是WHILE) { // if类似
           Exp(n的孩子节点1);
           Stmt(n的孩子节点2)
       }
```

遍历Stmt

实验设计——符号表填充

变量填充

- C--文法规定变量声明文法中包括ID(a)、ID LB INT RB(arr[10])两种形式，我们需要提取出符号名、加上Specifier中得到的类型t，就可以构建出变量的符号信息并将其加入到符号表中

```
 1   FieldList VarDec(struct Node* node, Type type,int structflag){
 2       提取node的孩子节点n;
 3       if (n类型是ID && n的符号名未在符号表中出现) {
 4           构建FieldList变量 f;
 5           初始化f的名称、类型、tail域信息;
 6           n的符号名及类型插入全局符号表中;
 7           返回f;
 8       } else { // 数组定义
 9           构建Type变量arrType;
10           初始化arrType的kind、u.array的size/elem类型;
11           递归调用VarDec(n, arrType)返回FiedList变量f;
12           返回f;
13       }
14   }
```

全局变量及数组的填充

实验设计——符号表填充

结构体填充

- C--文法规定结构体包含STRUCT OptTag LC DefList RC和STRUCT Tag。前者是结构体定义，后者是结构体变量声明，我们需要将这两者都加入到符号表中

```
1   Type StructSpecifier(struct Node *node){
2       提取node的孩子节点n;
3       if (n是结构体定义且名字存在) {
4           构建Type变量sType;
5           初始化sType的name、kind、u.structure, assign域为BOTH;
6           // 遍历结构体中的域
7           DefList(n的子节点, 1)
8           插入符号表中;
9           返回sType
10      } else if (n是结构体但名字不存在) { // 匿名结构体
11          与一般结构体定义类似，但名称可以定义为随机值或者置为NULL;
12          插入符号表并返回结构体类型sType;
13      }
14  }
```

结构体定义的填充

实验设计——符号表填充

函数填充

- C--文法规定包含Specifier FuncDec CompSt（必做内容不支持函数声明，暂不考虑函数声明）。首先，我们获取函数名、行号（用于错误输出）、返回值类型；其次，我们提取函数参数；最后，我们返回函数

```
1  Function FunDec(struct Node *node, Type type){
2      提取node的孩子节点n;
3      构建函数类型变量Fucntion func;
4      初始化func的name、line、type（返回值类型）;
5      if (n包含参数) {
6          调用VarList遍历n的参数并返回FieldList变量f;
7          设置func的param域的值为f;
8      } else { // 无参
9          设置func的param域的值为NULL;
10     }
11     返回func;
12 }
```

函数定义的填充

实验设计——语义检查

错误类型1

- 检查内容：变量在使用时未经定义
- 检查节点：Exp中ID子节点
- 检查过程：先判断是否是使用ID对应的变量，然后查看ID对应的变量是否存在于符号表中，最后根据检查结果输出内容。ID在符号表，则提取类型信息，否则就输出错误类型1

```
1  int main()
2  {
3      int i = 0;
4      j = i + 1;
5  }
```

必做样例1

```
1   Type Exp(struct Node *node){
2       提取node的孩子节点n;
3       if (n是ID) {
4           if (n的兄弟节点为NULL)
5               if (check(n的名称) == 0) // 变量不在符号表中
6                   printError(错误类型1, 行号, 变量名)
7               else
8                   Type type = Type_get(n的名称);
9                   return type;
10      }
11  }
```

错误类型1检测

实验设计——语义检查

错误类型2

- 检查内容：函数在调用时未经定义
- 检查节点：Exp中ID LP Args RP(函数调用形式1)或者ID LP RP（函数调用形式2）
- 检查过程：先判断是否是函数调用形式，然后从符号表中获取函数类型。函数存在，则继续执行调用，否则就输出错误类型2

```
1  int main()
2  {
3      int i = 0;
4      inc(i);
5  }
```

必做样例2

```
1  Type Exp(struct Node *node){
2      提取node的孩子节点n;
3      if (n是ID) {
4          if (n的兄弟节点是Args || n的兄弟节点==NULL) {
5              Type funcType = Type_get(n的名称);
6              if (funcType == NULL)
7                  printError(错误类型2, 行号, 函数名)
8                  return NULL;
9              ...
10         }
11     }
12 }
```

错误类型2检测

实验设计——语义检查

错误类型3

- 检查内容：变量出现重复定义，或变量与前面定义过的结构体名字重复
- 检查节点：VarDec节点
- 检查过程：先判断是否是使用ID对应的变量，然后查看ID对应的变量是否存在于符号表中。ID不在符号表，则添加到符号表，否则输出错误类型3，并返回构造的符号（防止语义检查中断）

```
1  int main()
2  {
3      int i, j;
4      int i;
5  }
```

必做样例3

```
1   FieldList VarDec(struct Node* node, Type type,int structflag){
2       提取node的孩子节点n;
3       if (n是ID) {
4           if (n不是结构体 && n的名称在符号表中存在)
5               printError(错误类型3, 行号, 变量名)
6               构建FieldList变量f;
7               使用n的属性初始化f;
8               返回f;
9       }
10  }
```

错误类型3检测

实验设计——语义检查

错误类型4

- 检查内容：函数出现重复定义
- 检查节点：FunDec节点
- 检查过程：先构造Function变量func并初始化，然后查看func是否存在于符号表中，最后根据检查结果输出内容。func不在符号表，则添加到符号表中，否则就输出错误类型4

```
1  int func(int i)
2  {
3      return i;
4  }
5  int func()
6  {
7      return 0;
8  }
9  int main()
10 {
11 }
```

必做样例4

```
1  Function FunDec(struct Node *node, Type type){
2      提取node的孩子节点n;
3      构造Function变量func并初始化;
4      if (checkfunc(func) != 0) { // 函数已存在符号表中
5          printError(错误类型4, 行号, 变量名)
6          return NULL;
7      }
8  }
```

错误类型4检测

实验设计——语义检查

错误类型5

- 检查内容：赋值号两边的表达式类型不匹配
- 检查节点：Dec节点和Exp中的ASSIGNOP节点
- 检查过程：Dec节点主要判断变量初始化类型和声明类型一致性；ASSIGNOP检查左右操作数类型是否一致

```
1  int main()
2  {
3      int i;
4      i = 3.7;
5  }
```

必做样例5

```
1   FieldList Dec(struct Node *node, Type type,int structflag){
2       提取node的孩子节点n;
3       Type expType = Exp(n的兄弟节点);//初始化类型
4       if (!Type_check(type, expType))
5           printError(错误类型5, 行号, 其他信息);return NULL;
6   }
7   Type Exp(struct Node *node){
8       提取node的孩子节点n;
9       if (n的兄弟节点是ASSIGNOP) {
10          获取赋值运算符左右操作数类型left, right;
11          if (!Type_check(left, right))
            printError(错误类型5, 行号, 其他信息);return NULL;
        }
    }
```

错误类型5检测

实验设计——语义检查

错误类型6

- 检查内容：赋值号左边出现一个只有右值的表达式
- 检查节点：Exp中ASSIGNOP子节点
- 检查过程：先获取左右操作数类型，然后判断左操作数是否是左值。如果是左值，再判断类型是否一致，否则就输出错误类型6

```
1  int main()
2  {
3      int i;
4      10 = i;
5  }
```

必做样例6

```
1   Type Exp(struct Node *node){
2       提取node的孩子节点n;
3       if (n的兄弟节点是ASSIGNOP) {
4           获取赋值运算符左右操作数类型left, right;
5           if (left->assign == RIGHT) // 右值
6           {
7               printError(错误类型6, 行号, 其他信息);
8               return NULL;
9           }
10      }
11  }
```

错误类型6的检测

实验设计——语义检查

错误类型7

- 检查内容：操作数类型不匹配或操作数类型与操作符不匹配
- 检查节点：Exp中一元运算符MINUS及二元运算符
- 检查过程：先判断负数表达式是否是基本类型，不是，则输出错误类型7；然后判断二元运算符两个操作数类型一致性，不一致输出错误类型7

```
1  int main()
2  {
3      float j;
4      10 + j;
5  }
```

必做样例7

```
1   Type Exp(struct Node *node){
2       提取node的孩子节点n;
3       if (n是负数MINUS) {
4           Type type = Type_get(n的兄弟节点);
5           if (type->kind != BASIC)
6               printError(错误类型7, 行号, 其他信息);return NULL;
7       }
8       if (n是二元运算符) {
9           Type left = Type_get(n的节点1);
10          Type right = Type_get(n的节点2);
11          if (Type_check(left, right) == 0)
                printError(错误类型7, 行号, 其他信息);return NULL;
        }
    }
```

错误类型7检测

实验设计——语义检查

错误类型8

- 检查内容：return语句的返回类型与函数定义的返回类型不匹配
- 检查节点：Stmt中的RETURN节点
- 检查过程：先获取return语句表达式类型，然后检测表达式类型和函数声明类型是否一致，不是，则输出错误类型8

```
1  int main()
2  {
3      float j = 1.7;
4      return j;
5  }
```

必做样例8

```
1  void Stmt(struct Node *node, Type ntype){
2      提取node的孩子节点n;
3      if (n是RETURN) {
4          Type expType = Exp(n的兄弟节点);
5          if (Type_check(ntype, expType) == 0)
6              printError(错误类型8, 行号, 变量名)
7      }
8  }
```

错误类型8的检测

实验设计——语义检查

错误类型9

- 检查内容：函数调用时实参与形参的数目或类型不匹配
- 检查节点：Exp中函数调用有参或者无参
- 检查过程：如果有实参，则实参和形参是否一致，不一致输出错误类型9；如果无实参而函数有形参，输出错误类型9

```
1  int func(int i)
2  {
3      return i;
4  }
5
6  int main()
7  {
8      func(1, 2);
9  }
```

必做样例9

```
1   Type Exp(struct Node *node){
2       提取node的孩子节点n;
3       if (n是ID) {
4           if (n的兄弟节点是Args) {
5               Type funcType = Type_get(n的名称);
6               FieldList param = Args(n的兄弟节点);
7               if (!Param_check(param, funcType->u.function->param))
8                   printError(错误类型9, 行号, 函数名)return NULL;
9   } else (n的兄弟节点为NULL)
10              Type funcType = Type_get(n的名称);
11              if (funcType有形参) printError(错误类型9,  行号, 函数名)
12      }
13  }
```

错误类型9检测

实验设计——语义检查

错误类型10

- 检查内容：对非数组型变量使用“[…]”（数组访问）操作符
- 检查节点：Exp中LB节点及其兄弟节点
- 检查过程：先判断当前表达式是否是数组访问操作，是的话，则检查变量是否是数组类型，不是则报告错误类型10

```
1  int main()
2  {
3      int i;
4      i[0];
5  }
```

必做样例10

```
1   Type Exp(struct Node *node){
2       提取node的孩子节点n;
3       if (n的兄弟节点是LB) { // 数组访问操作
4           Type arr = Exp(n);
5           Type idx = Exp(n的兄弟节点);
6           if (arr->kind != ARRAY) {
7               printError(错误类型10, 行号, 变量名);
8               return NULL;
9           }
10      }
11  }
```

错误类型10检测

实验设计——语义检查

错误类型11

- 检查内容：对普通变量使用“(...)”或“()”操作符
- 检查节点：Exp中函数调用有参或者无参
- 检查过程：先判断是否是函数调用，然后判断调用的符号类型是否是函数，不是则输出错误类型11

```
1  int main()
2  {
3      int i;
4      i(10);
5  }
```

必做样例11

```
1   Type Exp(struct Node *node){
2       提取node的孩子节点n;
3       if (n是ID) {
4           if (n的兄弟节点是Args) {
5               Type funcType = Type_get(n的名称);
6               if (funcType->kind != FUNCTION)
7                   printError(错误类型11, 行号, 函数名)return NULL;
8           } else (n的兄弟节点为NULL){
9               Type funcType = Type_get(n的名称);
10              if (funcType->kind!= FUNCTION)
11                  printError(错误类型11,  行号, 函数名)
            }
        }
    }
```

错误类型11检测

实验设计——语义检查

错误类型12

- 检查内容：数组访问操作符“[...]”中出现非整数
- 检查节点：Exp中LB节点及其兄弟节点
- 检查过程：先判断是否是数组访问操作，然后判断下标的类型是否是int类型，不是则输出错误类型1

```
1  int main()
2  {
3      int i[10];
4      i[1.5] = 10;
5  }
```

必做样例12

```
1   Type Exp(struct Node *node){
2       提取node的孩子节点n;
3       if (n的兄弟节点是LB) { // 数组访问操作
4           Type arr = Exp(n);
5           Type idx = Exp(n的兄弟节点);
6           ...
7           if (idx->kind != BASIC || idx->u.basic != 1) {
8               printError(错误类型12, 行号, 变量名);
9               return NULL;
10          }
11      }
    }
```

错误类型12检测

实验设计——语义检查

错误类型13

- 检查内容：对非结构体型变量使用“.”操作符
- 检查节点：Exp中DOT节点
- 检查过程：先判断是否是DOT节点，然后判断DOT左操作数的类型是否是结构体，不是则输出错误类型13

```
1  struct Position
2  {
3      float x, y;
4  };
5
   int main()
   {
       int i;
       i.x;
   }
```

必做样例13

```
1  Type Exp(struct Node *node){
2      提取node的孩子节点n;
3      if (n的兄弟节点是DOT) {
4          Type left = Type_get(n);
5          if (left->kind != STRUCTURE){// 不是结构体类型
6              printError(错误类型13, 行号, 变量名);
7              return NULL;
8      }
9  }
```

错误类型13检测

实验设计——语义检查

错误类型14

- 检查内容：访问结构体中未定义过的域
- 检查节点：Exp中DOT节点
- 检查过程：先判断DOT节点，然后获取调用structure中对应域的类型，不存在输出错误类型14

```
1  struct Position
2  {
3      float x, y;
4  };
5
   int main()
   {
       struct Position p;
       if (p.n == 3.7)
           return 0;
   }
```

必做样例14

```
 1  Type Exp(struct Node *node){
 2      提取node的孩子节点n;
 3      if (n的兄弟节点是DOT) {
 4          Type left = Type_get(n);
 5          Type fType = Type_get_f(left->u.structure-
 6  >domain, n的兄弟节点的名称);
 7          if (fType == NULL){
 8              printError(错误类型14, 行号, 变量名);
 9              return NULL;
10      }
11  }
```

错误类型14检测

实验设计——语义检查

错误类型15

- 检查内容：结构体中域名重复定义，或在定义时对域进行初始化
- 检查节点：Exp中VarDec节点和Dec节点
- 检查过程：先检测是否处于结构体中，然后判断是否引用了不存在的域或者初始化域，是则输出错误类型15

```
1  struct Position
2  {
3      float x, y;
4      int x;
5  };
6
7  int main()
8  {
9  }
```

必做样例15

```
1   FieldList VarDec(struct Node* node, Type type,int structflag){
2       提取node的孩子节点n;
3       if (n是ID) {
4           if (处于结构体中 && check(n的名称)==1) {
5               printError(错误类型15, 行号, 变量名);
6               构造FieldList f存储域的信息; return f;
7           }
8       }
9   }
10  FieldList Dec(struct Node *node, Type type,int structflag){
11      if (处于结构体中 && node孩子节点的兄弟节点存在) // 结构体域初始化
            printError(错误类型15,  行号, "");
    }
```

错误类型15检测

实验设计——语义检查

错误类型16

- 检查内容：结构体的名字与定义过的结构体或变量的名字重复
- 检查节点：StructSpecifier节点
- 检查过程：先判断是否是结构体定义，然后判断结构体名是否已存在于符号表中，是则输出错误类型16

```
1  struct Position
2  {
3      float x;
4  };
5  struct Position
6  {
7      int y;
8  };
9  int main()
10 {
11 }
```

必做样例16

```
1  Type StructSpecifier(struct Node *node){
2      提取node的孩子节点n;
3      if (n的兄弟节点是OptTag) {
4          构造Type type并初始化为结构体;
5          构造Structure s并初始化name，fields;
6          if (s->name && check(s->name))
7              printError(错误类型16, 行号, 变量名); return NULL;
8      }
9  }
```

错误类型16检测

实验设计——语义检查

错误类型17

- 检查内容：直接使用未定义过的结构体来定义变量
- 检查节点：StructSpecifier节点
- 检查过程：先判断是否是使用ID对应的变量，然后查看ID对应的变量是否存在于符号表中，最后根据检查结果输出内容。ID在符号表，则提取类型信息，否则就输出错误类型1

```
1  int main()
2  {
3      struct Position pos;
4  }
```

必做样例17

```
1   Type StructSpecifier(struct Node *node){
2       提取node的孩子节点n;
3       if (声明结构体变量) {
4           Type type = Type_get(n的名称);
5           if (type == NULL) {
6               printError(错误类型17, 行号, 变量名);
7               return NULL;
8           } else {
9               return type;
10          }
11      }
    }
```

错误类型17检测

实验设计——语义检查

选做内容

- 选做内容1：函数声明实现，考虑加入函数声明语法，记录函数的状态，比如使用常数区分函数声明和函数定义；对于函数声明与定义类型不一致，要报告错误类型18；对于只声明未定义的函数要报告错误类型19（函数的类型一致性检测考虑函数返回值、函数名、函数参数列表，这些构成函数签名，判断签名一致性）
- 选做内容2：作用域其那套，可以参考实验指导手册（支持多层作用域的符号表一节），采用Functional style或者Imperative style
- 选做内容3：结构体采用结构等价策略而非按名等价策略；参考前面Type_check和Param_check，获取两个结构体的domain，然后依次比较域类型是否一致即可

谢谢大家