编译原理 Lab2

202220013 徐简

项目环境:

1. GNU LINUX Realease: Ubuntu
2. GCC version: 7.5.0
3. GNU Flex version:2.6.4
4. GNU Bison version: 3.0.4

实现的功能

- 完成了必做的语义分析
- 完成了选做3——结构体间的类型等价机制由名等价改为结构等价

运行截图如下：

```
int main()
{
int i, j;
int i;
}
```

```
./parser ../Test/test3.cmm
Error type 3 at Line 4: Redefined variable "i".
```

下面简单地介绍一下基本的实现逻辑

符号表

- 采取实验手册推荐的方法，用哈希表来实现符号表

```
struct TableList_ {
    char* name;
    Type* type;
    unsigned size;
    TableList* next;
};
TableList* hashTable[HASHSIZE + 1];
```

- 其中Type为变量的类型，由整型、浮点、数组、结构体、函数组成

```
struct Type_ {
    enum { INT, FLOAT, ARRAY, STRUCTURE,  FUNC, WRONGFUNC } kind;
    union {
```

```
        //基本类型
        int basic;
        //数组类型信息包括元素类型与数组大小
        struct
        {
            Type* elem;
            int size;
        } array;
        //结构体类型信息是一个链表
        FieldList* structure;
        //FieldList* structure;
        //函数类型信息
        struct
        {
            Type* ret;
            int argc;
            FieldList* args;
        } function;
    } u;
};
```

- 产生冲突时，插入在链表头部，这样便于实现作用域的嵌套，查表所得即是最内层的定义（不过还是没有完成选做2 doge）

```
void insert(TableList* item)
{
    unsigned index = hash_pjw(item->name);
    item->next = hashTable[index];
    hashTable[index] = item;
}
```

- 针对选做三，设计了结构体的比较函数, 递归地对结构体链表进行比较。

```
int cmp_structure(FieldList* s1, FieldList* s2)
{
    int res = 0;
    while (s1 && s2) {
        if (s1 == s2)
            return 1;
        else if (s1->type->kind != s2->type->kind) {
            return 0;
        } else if (s1->type->kind == STRUCTURE || s1->type->kind == STRUCTURE) {
            res = cmp_structure(s1->type->u.structure, s2->type->u.structure);
            if (!res)
                return 0;
        } else if (s1->type->kind == ARRAY) {
            res = cmp_array(s1->type, s2->type);
            if (!res)
```

```
                return 0;
            } else {
                res = 1;
            }
            s1 = s1->tail;
            s2 = s2->tail;
        }
        if (s1 || s2)
            res = 0;
        return res;
}
```

语义分析

- 基于实验一实现的语法树，Bision构造语法树，语义的分析由函数`semantic_check`完成
- 从根节点开始，每当遇到语法单元ExtDef或者Def，就说明该结点的子结点们包含了变量或者函数的定义 信息，将变量函数的信息插入维护的哈希表中。

```
void semantic_check(Node *node);
void ExtDefList(Node *node);
void ExtDef(Node *node);
```

- 而当遇到语 法单元Exp，说明该结点及其子结点们会对变量或者函数进行使用，这个时候应当查符号表，检查语义是否符合C--的要求，并将错误信息输出到终端。

```
void Exp(Node* node);
```

使用方法

在terminal中输入相应指令，完成编译、运行、测试：

- make编译
- make test运行测试文件
- 在makefile中可以修改要使用的测试样例

待改进的地方

- 没有完成选做1,2
- 代码冗长，很多地方没有设计好
- 没有完成选做1,2，对语法分析造成了一些干扰，某些测试样例会输出一些奇怪的语法错误，也可能是实验1的语法分析器本身就有问题

实验思考

- 在不确定第一阶段的语法分析器有没有问题的情况下，debug太困难了
- 时间规划不够好，本来没打算完成选做，后来发现选做1并不是很困难(增加一个bool型变量标记是否定义)，但时间原因改了一半放弃了