《编译原理Lab2》实验报告

姓名：李诗宇

学号：3210100999

实验步骤

1.符号表实现

我采用**命令式风格**来实现我的符号表，即始终维护同一张符号表，采用栈来控制block中局部变量的作用域。数据结构定义如下：

```
struct bucket {
      string key;   //变量或函数名
      meta info;    //存储变量的数组维度大小或函数参数列表信息
      struct bucket* next;  //采用链表连接
};
#define SIZE 109
struct bucket *table[SIZE];
```

每当语法分析检测到变量或函数定义语句，用哈希函数将其map到`Table`相应的entry，检测是否在该作用域中有重复定义，若无则头插入链表。每条链表的`insert`与`pop`都是先进后出，与栈相同。

```
int check_redef(string key){
      uint index = hash(key)%SIZE;
      struct bucket* b;
      for(b = table[index]; b != NULL&&strcmp(b->key,"*"); b = b->next){
            if(0 == strcmp(b->key, key))
                  return false;
      }
      return true;
}
```

每次进入新的作用域时，对符号表`Table`的每一个entry都插入一个特殊符`“*”`。离开该作用域时则对每一条链表弹栈至特殊符号“*”，清除了该作用域的所有局部变量。

```
void beginscope(){
      for(int i = 0; i < 109; i++){
        struct bucket * special = create("*", NULL, NULL);
        special->next = table[i]->next;
        table[i]->next = special;
    }
}
void endscope(){
      for(int i = 0; i < 109; i++){
            struct bucket * head;
            for(head = table[i]->next; head != NULL&&strcmp(head->key, "*"); ){
                  struct bucket* temp = head->next;
                  free(head);
                  head = temp;
                  table[i]->next = head;
            }
            struct bucket* temp = head->next;
            free(head);
            head = temp;
            table[i]->next = head;
      }
}
```

2.类型检查

我实现的类型检查主要分为：

- 变量定义与使用时数组维数不同，数组访问越界。
- 操作数类型不匹配或操作数类型与操作符不匹配，如整型变量与数组变量相加减
- return 语句的返回类型与函数定义的返回类型不匹配
- 函数调用时实参与形参的数目或类型不匹配; 特别的，函数调用时数组对应维数不匹配
- 对普通变量使用"(...)"或"()"(函数调用)操作符，也可以认为是调用未定义的函数

我对变量与函数分别采用特定数据结构来记录变量定义时的数组维数（普通int为0维）与数组size，函数的参数个数与参数类型，若为数组参数则记录数组维数（普通int为0维）与数组size。在使用是检查是否与定义一致。其余类型检查较为容易，不加赘述。

```
struct BRACKETList{
      int length;
      int *array_size;
}list1;
struct FuncFParam{//Expr
      int type; int length; int *array_size; char *key; int value;
}funparam;
struct ParamList{
      struct YYSTYPE::FuncFParam f[30];
      int length;
}paralist;
```

3.数组初始化列表

本实验最为复杂的部分是数组的列表初始化，通过对文法规则的分析，依次处理初始化列表内的元素, 元素的形式无非就两种可能: 整数, 或者另一个初始化列表，因此设计如下的树状结构**(left child right sibling)**来存储初始化列表

```
struct InitList {
    struct InitList* next;  // 指向下一个节点，值或子列表
    int isValue;            // 是否为值，0代表子列表，1代表值
    union { int value;          // 值节点的值
        struct InitList* child; // 子列表节点
    } data;
} * initlist;
```

然后我采用北京大学编译原理实验指导中的建议，先按照规则对初始化列表补零，将多维数组打平到一维数组(`int * val`)，然后检查是否越界。

```
int zero_fill(struct YYSTYPE::InitList * root, int size, int &index, int * val, int * array_size, int
length){
      struct YYSTYPE::InitList* head = root;
      struct YYSTYPE::InitList* current = root->data.child;
      while(current){
            if(current->isValue){   //遇到整数，直接一次填值
                  val[index++] = current->data.value;
            }
            else {  //遇到子列表节点
                  int t, mod;
                  for(t = 0, mod = 1; t < length; t++){ //检查当前对齐到了哪一个边界，
                        mod *= array_size[t];
                        if(t == length - 1) break;
                        else if(index % mod != 0) break;
                  }
                  int new_index = index + mod / array_size[t];  //然后将当前初始化列表视作这个边界所对应的最长维度的数
                  zero_fill(current, new_index, index, val, array_size, t-1);//组的初始化列表，并递归处理
                  index = new_index;
            }
            current = current->next;
      }
      printf("index : %d, size : %d\n" ,index, size);
      if(index > size) return -1;//数组越界
      else return 0;
}
```

编译

在提交的zip文件中包含了`Makefile`文件，只需进入Makefile所在的目录执行：

```
make
```

`src`文件夹中的源代码即可完成编译，生成执行文件`compiler`，即可进行测试。