# 编译原理实验报告

实验二：语义分析

高亦远 151160014

## 一、 实验目的

设计、编写一个语义分析程序，实现 c—语言的语义分析功能，加深对语法翻译的理解。

## 二、 实验要求

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。的程序需要对输入文件进行语义分析（输入文件中可能包含函数、结构体、一维和高维数组）并检查如下类型的错误：

1) 错误类型 1：变量在使用时未经定义。
2) 错误类型 2：函数在调用时未经定义。
3) 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。
4) 错误类型 4：函数出现重复定义（即同样的函数名出现了不止一次定义）。
5) 错误类型 5：赋值号两边的表达式类型不匹配。
6) 错误类型 6：赋值号左边出现一个只有右值的表达式。
7) 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（例如整型变量与数组变量相加减，或数组（或结构体）变量与数组（或结构体）结构体变量相加减）。
8) 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。
9) 错误类型 9：函数调用时实参与形参的数目或类型不匹配。
10) 错误类型 10：对非数组型变量使用“[…]”（数组访问）操作符。
11) 错误类型 11：对普通变量使用“(…)”或“()”（函数调用）操作符。
12) 错误类型 12：数组访问操作符“[…]”中出现非整数（例如 a[1.5]）。
13) 错误类型 13：对非结构体型变量使用“.”操作符。
14) 错误类型 14：访问结构体中未定义过的域。
15) 错误类型 15：结构体中域名重复定义（指同一结构体中），或在定义时对域进行初始化（例如 struct A { int a = 0; }）。
16) 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。
17) 错误类型 17：直接使用未定义过的结构体来定义变量。

实验二要求通过标准输出打印程序的运行结果。对于那些没有语义错误的输入文件，你的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，你的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，其格式为：
Error type [错误类型] at Line [行号]: [说明文字].

### 3.1 程序结构

实验一的程序的源文件由词法分析文件 lexical.l、语法分析文件 syntax.y 及程序入口 main.c 组成。

实验二在实验一的基础上，新增了语义分析文件 semantic.c 及其对应头文件 semantic.h。

此外，为使代码结构明晰且调用方便，将语法树的结构定义和语法树的相关操作函数从 lexical.l 和 syntax.y 中分离出来，单独成为一组文件 tree.c 和 tree.h。

使用 makefile 可以快速进行编译。经过编译后生成中间文件 lex.yy.c、syntax.tab.c、syntax.tab.h 及执行程序 complier。

测试时在根目录下使用 make 命令进行编译，使用./complier test1.cmm 来对测试文件进行测试。

### 3.2 程序功能

本程序实现了 c--语言的语义分析功能。当发现语义错误（如使用未定义的变量、变量重复定义等）时，会输出“Error type [错误类型] at Line [行号]: [说明文字]”格式的；当发现语法错误时，会输出“Error: type B at line …: ……（说明文字）”；当代码中不含有词法和语法错误时，则会输出语法树信息。

例：对于源代码：

```
int main()
{
    int i;
    int i;
    a=5;
}
```

会输出错误信息：

```
Error: type 1 at line 4: Redefined variable “i”.
Error: type 3 at line 5: Redefined variable “a”.
```

对于语义信息正确的代码则不会报错。

## 3.3 功能实现

#### 3.3.1 符号表的结构

为实现语义分析功能，需要用到符号表。符号表用来记录源程序中各种名字及其特性信息，前者如如变量名、函数名等；后者则包括上述名字的类型、维数、参数个数、数值等。

符号表可以采用线性表、平衡二叉树、散列表等结构实现。线性链表表结构简单，但查找起来效率并不优秀；二叉树过于复杂，实现难度过大；而散列表虽然空间复杂度较高（需要申请一个大数组），但查找效率十分优秀，代码结构上也比较简单，容易实现。综上，选择了散列表来作为符号表的基础结构。

本程序使用了一个 65536 项的大数组作为符号表的主体。每一项为名为 FeildList_的结构体类型的指针 FeildList。（即“FeildList”是“FeildList_”的指针）结构体 FeildList_定义如下：

```
typedef struct FieldList_ {
    char *name;        //变量/函数/结构名
    TypePtr type;    //类型
    FieldList tail;   //指向结构体或链表的下一维度
    int collision;   //hash 值是否冲突
}FieldList_;
```

其中：

TypePtr 是一个结构体类型 Type_的指针，该结构体中包含了符号的信息，例如符号是

basic 类型还是数组、结构体或是函数；basic 类型的话是 int 还是 float；数组的话还包括数组的大小和类型（以 Type_嵌套定义）；函数的话则包括其参数、数量以及函数本身的返回值类型等。总之，TypePtr 包含了每个符号的详细详细。

FeildList_指针类型的 tail 指向了某个域的下一个域，用于定义多维数组和结构体。以多维数组为例，tail 指向其下一个维度，如 int[][]的 tail 指向 int[]类型。

collision 用于在极少数情况下出现的 hash 值冲突的情况，其实际上表示大数组中下标的偏移量，当两项出现下标相同的冲突时，后一项的实际下标会加上 collision 值。由于冲突情况十分少见，collision 一般为 0，偶尔可能为 1。

Hash 函数采用了实验指导中给出的 P.J.Weinberger 函数。

3.3.2 符号表的操作

符号表包含一些基础的操作，如查找，插入等。当需要插入新项时，根据变量（或函数、结构体）名利用 hash 函数计算出下标 key，然后 hashTable[key]指向该项对应的 FeildList_结构。查找同理。

3.3.3 语义分析过程

语义分析的主要输入是实验一所生成的语法树。语义分析采用语法树相关代码中的 tranverse()函数来遍历整个语法树，然后对语法树中的每一节点进行相关分析和操作。例如：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，此时通过遍历将这些信息提取出来并插入到符号表里；每当遇到语法单元 Exp，说明该结点及其子结点们会对变量或者函数进行使用，这个时候则查符号表以确认这些变量或者函数是否存在以及它们的类型，如果发生错误则打印错误信息

语义分析的入口函数为 ExtDefList(Node *root)，这也是语法树的根节点名称。对于不同语法单元的处理函数大多以其节点名称命名，如 CompSt(Node *root,TypePtr funcType)、DefList(Node *root)，这样命名比较方便且清晰。

3.3.4 左值错误

语义分析中的大部分操作只涉及到查表与类型操作，不过有一个错误例外，那就是有关左值的错误。即赋值号左边只能出现地址（左值）。而这只有三种情况：有 ID、Exp LB Exp RB 以及 Exp DOT ID。这可以通过语法检查实现。当出现等号（ASSIGNOP）时，检查等号左边是否为以上三项，若不是则报错。