

# 华中科技大学

## 实践课程报告

题目： C--语言编译器

课程名称： 编译原理实践

专业班级： CS1409

学 号： U201414797

姓 名： 张丹朱

指导教师： 邵志远

报告日期： 2017 年 1 月 14 日

计算机科学与技术学院

# 目录

<b>1 选题背景.....</b>	<b>1</b>
1.1 任务.....	1
1.2 目标.....	1
1.3 源语言定义.....	1
<b>2 实验一 词法分析和语法分析.....</b>	<b>2</b>
2.1 单词文法描述.....	2
2.2 语言文法描述.....	3
2.3 词法分析器的设计.....	3
2.4 语法分析器设计.....	4
2.5 语法分析器实现结果展示.....	5
<b>3 实验二 语义分析.....</b>	<b>7</b>
3.1 语义表示方法描述.....	7
3.2 符号表结构定义.....	7
3.3 错误类型码定义.....	10
3.4 语义分析实现技术.....	10
3.5 语义分析结果展示.....	13
<b>4 结束语.....</b>	<b>16</b>
4.1 实践课程小结.....	16
4.2 自己的亲身体会.....	16
<b>参考文献.....</b>	<b>18</b>

# 1 选题背景

## 1.1 任务

主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生对系统软件编写的能力。

## 1.2 目标

本次课程实践目标是构造一个高级语言的子集的编译器，目标代码是汇编语言。按照任务书，实现的方案可以有很多种选择。

## 1.3 源语言定义

可以根据自己对编程语言的喜好选择实现。建议大家选用 C 语言的简单集合 C--语言或教材中的 Decaf 语言。

本次实验选择 C--语言作为源语言。

## 2 实验一 词法分析和语法分析

### 2.1 单词文法描述

使用正则表达式来进行单词文法的描述，在 C--中基本的单词及其正则表达式如下表 1.1 所示。

（其中，附加了对八进制和十六进制数以及指数形式的浮点数的识别。

表 1.1 单词正则表达式表

单词	正则表达式
INT	$(([1-9][0-9]^*)0) \mid ((0x[1-9a-fA-F][0-9a-fA-F]^*)0) \mid ((0x[1-9a-fA-F][0-9a-fA-F]^*)0)$
FLOAT	$([0-9]^*\.[0-9]^+ \mid [0-9]^+\.[0-9]^+)((E e)[+-]?[0-9]^+)?$
ID	$[a-zA-Z]([0-9] \mid [_a-zA-Z])\{0,31\}$
SEMI	;
COMMA	,
ASSIGNOP	=
RELOP	> < >= <= == !=
PLUS	+
MINUS	-
STAR	*
DIV	/
AND	&&
OR	
DOT	.
NOT	!
TYPE	int float
LP	(
RP	)
LB	[
RB	]
LC	{
RC	}
STRUCT	struct
RETURN	return
IF	if
ELSE	else
WHILE	while

另外，对于单行和多行注释也进行了识别，将注释内容全部忽略。

## 2.2 语言文法描述

采用上下文无关文法（LL（1）文法）描述语言，具体产生式见编译原理实验书 P116 至 117,在此不多加赘述。

除了明确基本的产生式外，还需要给基本的算符类单词定义结合型和优先级，由于使用了 `bison` 工具，故可以很方便地为各词法单元添加结合性和优先级（越前面的优先级越高）。

```
%right ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right NOT
%left LP RP LB RB LC RC DOT
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
```

## 2.3 词法分析器的设计

词法分析器主要利用 `flex` 工具直接生成，利用正则表达式识别词法单元，在后面跟大括号扩起来的动作，`flex` 会直接帮助生成词法分析的相关程序。

其主要利用的仍是有限状态机的思想，通过逐个字符的读入并与已经写好的正则表达式进行匹配，根据读入的字符与当前的状态决定接下来的状态，若是最后能够达到某个终态，则其识别为对应的词法单元并执行相关的动作。

以去掉多行注释为例，代码片段如下：

```
"/" { BEGIN COMMENT;          /* 切换到注释模式 */ }
<COMMENT>. |
```

```
<COMMENT>\n ;      /* 抛弃注释文字 */
<COMMENT>"*/" { BEGIN INITIAL;      /* 回到正常状态 */ }
```

可见,当读到“\*/”后便进入状态 COMMENT,期间读到的所有字符全部忽略,直到读到“\*/”则返回初始状态继续进行审查。

在词法分析器识别出一个词法单元之后,构造语法树的叶子结点并返回该词法单元对应的类型码用于在语法分析器中进行使用。例如:

```
";" { yyval.gt = Create_Node(yylineno, "SEMI", 0); return SEMI;}
```

## 2.4 语法分析器设计

语法分析器采用自底向上的分析方法,当栈中产生可归约前缀时进行归约并执行相应的动作。

语法分析的目标是要能够获得一棵语法树供后面的语义分析进行使用,故在产生式进行归约的时候就需要生成相应的语法树结点,树结点的结构定义如下:

```
// 语法树结点
typedef struct gramtree {
    int unit_type;          // 类型（终结符或非终结符，0 为终结符，-1 为产生
                            // 空的语法单元，主要用于语法树输出时判断是否要额外信息）
    int line;              // 行号
    int type0;             // 类型（0:int, 1:float, 2:array, 3:func, 4:struct）
    symtable *type;        // 具体类型
    int leftval;           // 标记是否为左值
    char* lex_unit_name;   // 词法单元名
    struct gramtree* l_child; // 左孩子结点
    struct gramtree* r_child; // 右孩子结点
    // 联合体用于存放额外打印信息
    union {
        char *id_type_name; // 标识符对应词素或具体类型
        int int_val;         // int 型的值
        float float_val;     // float 型的值
    };
};
```

```
}gramtree;
```

对于树的相关操作主要是树的创建与遍历，树采用孩子兄弟表示法（即左孩子为当前结点的孩子，右孩子为当前结点的兄弟）。在创建一个新的结点时将其第一个孩子结点挂在左子树，将后面的孩子全部挂在左孩子的右子树。

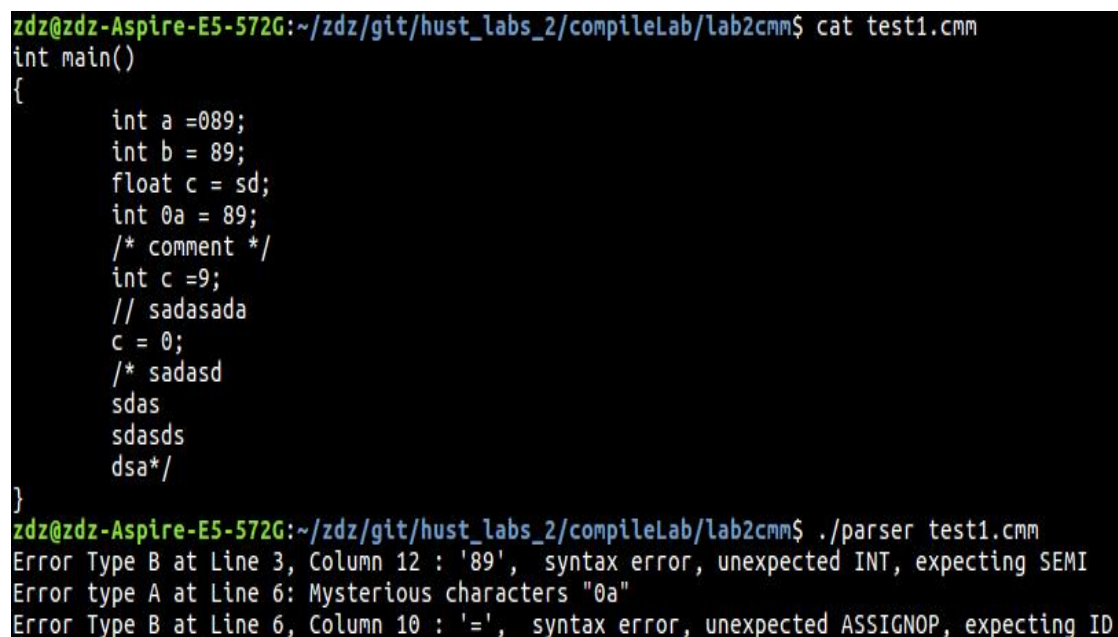
树的遍历采用先序遍历的方式，先看本身，再看其孩子，最后看其兄弟。

语法分析器完成后需要能够根据词法分析器生成一棵语法树。

## 2.5 语法分析器实现结果展示

语法分析器完成后，具备查错与生成语法树的功能。

当有词法错误或是语法错误的时候，语法分析器指出其所在的行、列，并将错误原因给出。如下图 2.1 所示。



```
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_labs_2/compileLab/lab2cmm$ cat test1.cmm
int main()
{
    int a =089;
    int b = 89;
    float c = sd;
    int 0a = 89;
    /* comment */
    int c =9;
    // sadasada
    c = 0;
    /* sadasd
    sdas
    sdasds
    dsa*/
}
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_labs_2/compileLab/lab2cmm$ ./parser test1.cmm
Error Type B at Line 3, Column 12 : '89', syntax error, unexpected INT, expecting SEMI
Error type A at Line 6: Mysterious characters "0a"
Error Type B at Line 6, Column 10 : '=', syntax error, unexpected ASSIGNOP, expecting ID
```

图 2.1 语法分析器报错图

如图，以前导 0 开头的数为八进制数，不能有 8、9，故此处报错，显示出错的行、列和单词，并给出出错原因。

若是词法和语法都没有问题则生成相应的语法树，如下图 2.2 所示。

```

RC
SEMI
ExtDefList (6)
  ExtDef (6)
    Specifier (6)
      TYPE: int
    FunDec (6)
      ID: main
      LP
      RP
    CompSt (7)
      LC
      DefList (8)
        Def (8)
          Specifier (8)
            TYPE: float
          Declist (8)
            Dec (8)
              VarDec (8)
                ID: b
                ASSIGNOP
                Exp (8)
                  FLOAT: 78.900002
          SEMI
        DefList (9)
          Def (9)
            Specifier (9)
              TYPE: int
            Declist (9)
              Dec (9)
                VarDec (9)
                  ID: c
                  ASSIGNOP
                  Exp (9)
                    INT: 16
              COMMA
            Declist (9)
              Dec (9)
                VarDec (9)
                  ID: d
                  ASSIGNOP
                  Exp (9)

```

图 2.1 语法分析器生成的语法树

语法树在打印时使用缩进代表树的层数，对于 ID，同时打印出变量名，对于 INT 和 FLOAT，同时打印出其数值（上图中的浮点数据为指数形式的浮点数转换得到，整型数为八进制和十六进制转换得到，均无误）。



## 3 实验二 语义分析

### 3.1 语义表示方法描述

在实验一中生成的语法树的基础上为各结点增加相应的属性,主要是增加类型。各语法单元类型的确定是在语法树生成之后采用自顶向下的方式确定语法树各结点、符号表各变量的类型和相关数据。

然后利用符号表和语法树结点本身的属性,对相应的短语中的单元进行类型的匹配。若类型不匹配则报语义错误。

### 3.2 符号表结构定义

采用基于十字链表和 open hashing 散列表的符号表,支持多层作用域。

其结构如下图 3.1 所示。

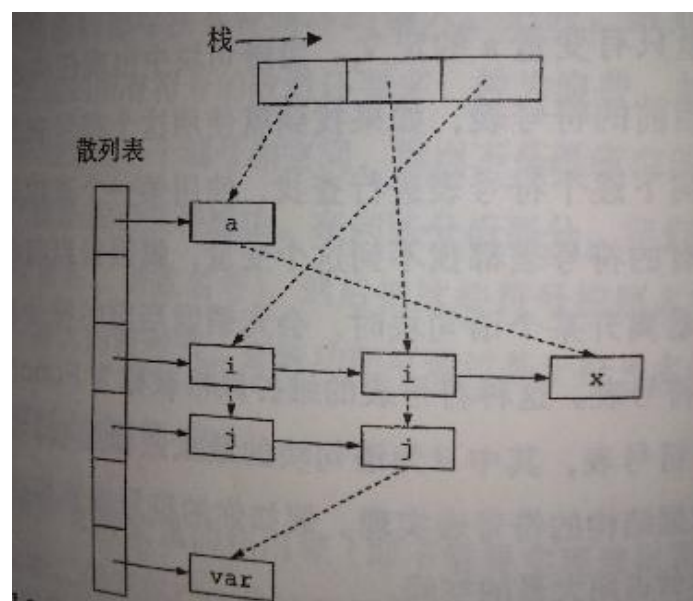


图 3.1 符号表结构图

如图，符号表主体采用 open hashing 散列表，方便查找与插入，另外使用一个栈（具体做实验的时候使用了数组进行了近似实现）对符号表的作用域进行管理。

每次向散列表中插入元素时，总是将新插入的元素放到该槽下挂的链表以及该层所对应的链表的表头。每次查表时如果定位到某个槽，则按顺序遍历槽并返回第一个查找到的。此时该返回值必定是最内层定义。

每次进入一格语句块时为这层语句块新建一个链表来串联该层中新定义的变量；每次离开一个语句块时需要顺链表将所有本层定义变量全部删除。

符号表中变量（语法树中同样如此）的类型采用链表的结构，链表每个结点又是一个代表类型的结构体，对于高维数组，其类型结构如下图 3.2 所示。

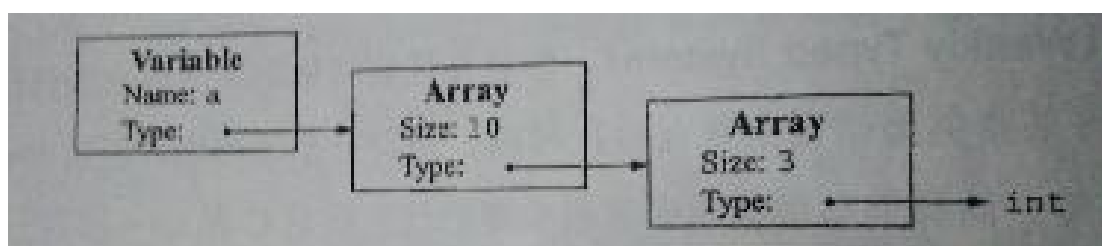


图 3.2 高维数组链表表示

对于结构体变量，其类型表示如下图 3.3 所示。

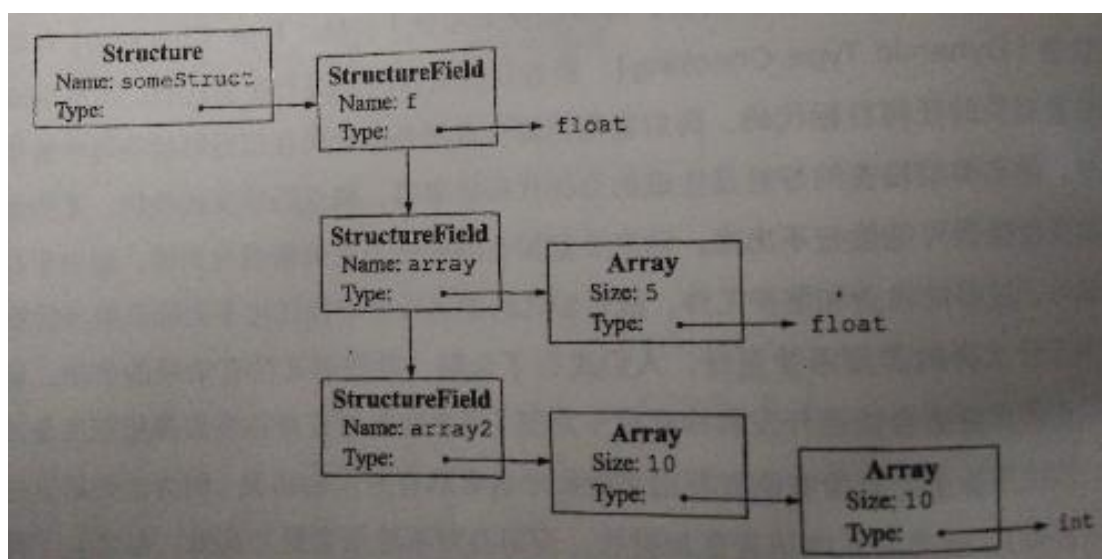


图 3.3 结构体的链表表示

由以上分析可以构造符号表结点的结构体如下表示：

```
// 符号表结点
typedef struct symtable {
    char *name;                // 符号名
    int type0;                 // 变量类型 0:int,1:float,2:array,3:func, 4:struct
    int size;                  // 维度或参数数目
    int kind;                  // 表示是否为结构体类型声明，1 是
    struct symtable *type;     // 具体类型或返回类型
    struct symtable *args;     // 参数列表
    // 联合体用于存放变量的值
    union {
        int int_val;
        float float_val;
    };
    struct symtable *next;     // 指向下一个符号表结点
    struct symtable *pre;      // 指向上一个符号表结点，方便删除当前层
                                // 时调整 openhash 的链
    struct symtable *snext;    // 指向当前层的下一个符号表结点
} symtable;
```

按照设计思路，将符号表中每个结点存储一个变量，可以是 int 或 float，也可以是数组、函数、结构体，将所有的变量全部放在一张符号表中统一进行管理。

对于每个结点，其自身有一个代表类型的整型变量 type0, 另外还有一个指针指向具体类型。具体类型仍然使用符号表结点，以它的 type0 和 type 指针指向的类型来描述变量的具体类型。如此以来，变量本身和变量类型使用相同的结构体，方便进行各类查询与其他操作。

由于要记录当前层，故需增加一个全局变量 SCOPE 来表示当前层数。由于我是用数组代替了栈，所以直接用 SCOPE 便可以查找到当前层，获得层号，方便查找与删除。

### 3.3 错误类型码定义

本次实验，除了完成实验书上给出的所有错误类型外，另外增加错误类型 18 和 19。最后所有错误类型的定义如下表 3.1 所示

表 3.1 语义错误类型定义表

错误类型	错误原因
1	变量在使用时未经定义
2	函数在调用时未经定义
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	函数出现重复定义
5	赋值号两边的表达式类型不匹配
6	赋值号左边出现一个只有右值的表达式
7	操作数类型不匹配或操作数类型与操作符不匹配
8	return 语句的返回类型与函数定义的返回类型不匹配
9	函数调用时实参与形参的数目或类型不匹配
10	对非数组型变量使用“[...]”操作符
11	对普通变量使用“( ... )”或“( )”操作符
12	数组访问操作符“[...]”中出现非整数
13	对非结构体型变量使用“.”操作符
14	访问结构体中未定义的域
15	结构体中域名重复定义，或在定义时对域进行初始化
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	直接使用未定义过的结构体来定义变量
18	没有 main 函数
19	if 和 while 语句的条件类型不为 int

### 3.4 语义分析实现技术

语义分析采用自顶向下的分析方式，对于生成的语法树，从根结点开始进行先根遍历，遇到特定的语法单元时转入相应的处理子程序，计算变量类型，进行类型的检查。

以下挑选主要的处理子程序进行分析。

## 1. 处理 ExtDef

对于 ExtDef 的处理主要根据参数的数量分为三种情况，

一种是直接 Specifier 后跟 SEMI 的,主要处理方式是调用处理 Specifier 的子程序将类型确定下来，根据文法，一般只有结构体的声明是有意义的，其余皆为无意义的声明。

第二种是变量定义，对于这种，处理完 Specifier 获得类型之后，在将获得的类型作为参数传递给处理 ExtDecList 的函数，将相应变量的类型赋为获得的类型。

第三种是对函数定义的处理,将从 Specifier 中获得的类型作为函数的返回值类型，再处理参数列表，参数列表处理完后，将获得的变量挂上符号表当前作用域，同时将参数列表挂到符号表相应函数结点上。最后处理函数体。

## 2. 处理 Exp

对于表达式的处理，首先是分清表达式的类别，如加法，赋值等，然后根据相应的规则进行类型的检查。

一般流程为：递归处理产生式右部的 Exp，获得其类型，然后根据当前表达式的类别，应用不同的规则判断类型是否匹配，若不匹配则转错误输出。若产生式右部各单元类型均符合要求，则按照表达式的不同，产生相应的类型赋给当前 Exp 结点。

## 3. 处理 Specifier

对于 Specifier 的处理，主要是要获得类型并返回。

Specifier 对应的产生式有两种，一种是 Specifier<-TYPE，对于此种类型的直接生成一个新的符号表结点(因为类型和符号用的是同一种结构体)，根据 TYPE

的类型决定新生成结点的 type0。

对于生成 StructSpecifier 的情况,要判断 struct 到底是定义还是已经有了可以直接作为类型。若是定义的语句,则需要在符号表中插入一个新的结点,其作为声明存放在符号表中(为了防止同名变量无法定义,所有 struct 的声明其结点中 kind 是为 1 的,以普通方式查找符号表的结点时会跳过此类声明),接着处理该 struct 的 DefList,处理完后从符号表上将这些域形成的链表剥离,当前 struct 结点的域指针指向该链,最后返回该符号表结点。

对于 STRUCT Tag 形式的,则要从符号表中查找到 Tag 对应的结构体声明,取其 type 链并将其作为返回值。

#### **4. 处理 VarDec**

若其左孩子结点为 ID,则在符号表中新建一结点并插入,同时将 type 指向传进来的类型,再根据 type 确定该新增变量的 type0 的值。

若其孩子结点组成数组形式的,则新增一个描述类型的结点,该结点为数组类型,size 由产生式中的 INT 决定,该结点的类型为传进来的类型,同时将该结点作为参数传递下去,递归调用处理 VarDec 的函数处理当前 VarDec 的左孩子结点。如此一级级链下去最终便可以获得数组类型的 type。

#### **5. 处理 If、while**

先处理条件表达式,获得类型后判断其是否符合标准,然后 SCOPE++,开始新的作用域,处理 Stmt。

#### **6. 比较两个类型是否相等**

若两个类型的 type0 不同则返回 0,若相同,对于 type0 为 0 或 1 的情况则可以直接返回 1;对于 type0 为 2(数组类型),则继续比较 type,递归调用该判断类



型相等的函数，返回递归调用获得的返回值;对于 type0 为 3（结构体）的情况，循环比较其域的类型，同样是递归调用该比较函数。（结构体采用结构等价而不是名等价）

## 7. 其他

对于其他语法单元的处理与上面已经列出的处理子程序思路相似，更多的是递归调用自身和调用上面的主要函数，在此不多赘述。

## 3.5 语义分析结果展示

语义分析有错时会将出错的位置和原因打印出来，如下图 3.4 所示

```
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_labs_2/compileLab/lab2cmm$ ./parser testyuyi.cmm
Error type 3 at Line 12: Redefined variable 'b'.
Error type 3 at Line 14: Redefined variable 'a'.
Error type 3 at Line 23: Redefined variable 'aa'.
Error type 3 at Line 24: Redefined variable 'b'.
Error type 6 at Line 26: The left-hand side of an assignment must be a variable.
Error type 5 at Line 27: Type mismatched for assignment.
Error type 8 at Line 28: Type mismatched for return.
Error type 4 at Line 31: Redefined function 'func1'.
Error type 5 at Line 43: Type mismatched for assignment.
Error type 13 at Line 44: Illegal use of '.'.
Error type 5 at Line 44: Type mismatched for assignment.
Error type 1 at Line 45: Undefined variable 'mmm'.
Error type 5 at Line 45: Type mismatched for assignment.
Error type 6 at Line 47: The left-hand side of an assignment must be a variable.
Error type 5 at Line 48: Type mismatched for assignment.
Error type 5 at Line 49: Type mismatched for assignment.
Error type 10 at Line 51: Illegal use of '['.
Error type 5 at Line 51: Type mismatched for assignment.
Error type 5 at Line 52: Type mismatched for assignment.
Error type 12 at Line 53: Between the '[' must be an integer.
Error type 5 at Line 53: Type mismatched for assignment.
Error type 5 at Line 54: Type mismatched for assignment.
Error type 5 at Line 56: Type mismatched for assignment.
Error type 10 at Line 58: Illegal use of '['.
Error type 10 at Line 58: Illegal use of '['.
Error type 10 at Line 58: Illegal use of '['.
Error type 5 at Line 58: Type mismatched for assignment.
```

图 3.4 语法分析报错截图

如图所示，当代码中出现语义错误的时候，语义分析程序输出错误信息，包括错误类型，错误发生所在行以及错误说明。

经过测试，各类型的错误均可以正确识别，符合实验预期。

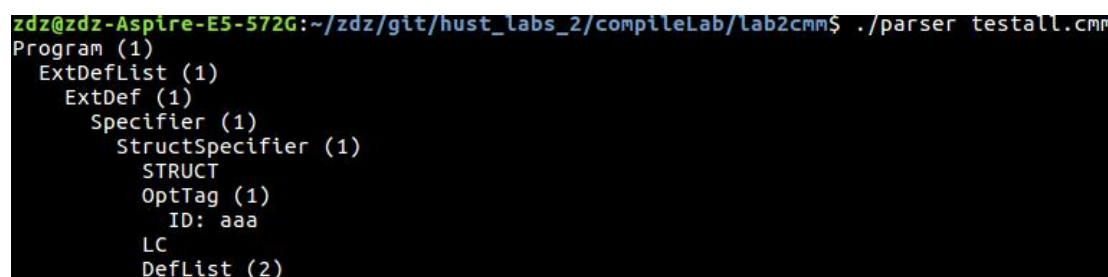
当代码中无语法错误时不报错，而是将简化的符号表输出到 symbol\_table.txt

以备查验。下面是测试的例子之一。

源程序为：

```
struct aaa{
    int a;
    struct v{
        int a;
    }b;
}b;
int main()
{
    int a;
    a = b.a;
    a = b.b.a;
    return 0;
}
```

此时，进行语义分析过后得到如下图 3.5 所示界面信息：



```
zdz@zdz-Aspire-E5-572G:~/zdz/git/hust_labs_2/compileLab/lab2cmm$ ./parser testall.cmm
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        StructSpecifier (1)
          STRUCT
          OptTag (1)
            ID: aaa
          LC
          DefList (2)
```

图 3.5 正确的语义分析结果

没有错误，打印了语法分析阶段生成的语法树。

查看 symbol\_table.c,获得如下信息：

第 1 层作用域：

name: a type: 0

第 2 层作用域：

name: a type: 0

第 1 层作用域：



name: b type: 4

第 0 层作用域:

name: b type: 4

第 1 层作用域:

name: a type: 0

第 0 层作用域:

name: main type: 3

简单地记录了变量的层数，变量名和变量类型。经检查，作用域和类型的判断均正确。符合实验预期。

## 4 结束语

### 4.1 实践课程小结

本次实验主要完成了如下几点工作：

- (1) 确定了所有单词的正则表达式，实现基本单词识别，附加实现八进制和十六进制数的识别、指数形式的浮点数的识别和单行、多行注释的识别。
- (2) 确定了语法树的结构，生成了语法树。
- (3) 完成了对实验一的调试与测试工作。
- (4) 确定了符号表的结构，实现了符号表的相关操作函数，实现了对整棵语法树的语义分析。
- (5) 完成了对语义分析的调试与测试工作。

### 4.2 自己的亲身体会

总的来说，本次实验任务设计的比较合理，若是能够把整个编译器完全实现，那么对于理解编译原理相关理论知识会有极大的帮助。唯一不足的是实验的时间安排并不合理，安排的时间太少了，跟不上进度。

对于词法和语法分析，使用了 flex 和 bison 工具之后节省了大量的时间，有利于后面工作的展开，但同时失去了一次对于有限状态机和分析表的构造与使用的练习机会，好在利用工具时可以看到其生成的源程序，能够帮助我理解和体会词法分析与语法分析的过程。

在语法分析构造语法树的时候构造了一棵孩子兄弟表示法的二叉树，使用的

时候发现很不方便，构造成多叉树会好一些。

在语义分析的时候动态维护符号表，一出作用域就将该作用域的相关符号信息从符号表中删去，这样在语义分析的时候降低了空间的消耗，方便了对作用域所处层数的判断，使每层的作用域清楚得区分开来，在语义分析过程中有很大帮助。但是在后续生成中间代码的过程中这样的符号表使得没法将生成中间代码的程序单独一个文件，只能在语法分析的同时进行词法分析，有诸多不便，下次在设计符号表的时候需要好好考虑如何使得符号表能够被高效地利用。

另外，像错误类型、语法单元等数据都可以用枚举，方便使用，做实验的时候考虑不周，比较是否是某种类型时较为复杂。。

总之，虽然没有完整地完成实验，但是掌握了 flex 和 bison 的使用，熟悉了语法树和符号表的构造及使用，练习了树的生成与遍历、链表的创建和查找，对于自顶向下的语义分析也有了较深的体会，收获颇丰。

## 参考文献

- [1] 吕映芝等. 编译原理(第二版). 北京: 清华大学出版社, 2005
- [2] 胡伦俊等. 编译原理(第二版). 北京: 电子工业出版社, 2005
- [3] 王元珍等. 80X86 汇编语言程序设计. 武汉: 华中科技大学出版社, 2005
- [4] 王雷等. 编译原理课程设计. 北京: 机械工业出版社, 2005
- [5] 曹计昌等. C 语言程序设计. 北京: 科学出版社, 2008