

**实践课程报告**

**题目： C—编译器的实现**

**课程名称： 编译原理实践**

**专业班级： CS1409**

**学 号： U201414795**

**姓 名： 王卓焱**

**指导教师： 邵志远**

**报告日期： 2017年 1 月 13日**

**计算机科学与技术学院**

**目录**

# 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

本次实验选用的文法是来自实验书的C -- 文法，文法的特点是精简版的C语言。

# 2词法分析和语法分析

本实验的任务是编写一个程序对使用C – 语言书写的代码进行词法和语法分析，并且打印分析结果。实验要求使用词法分析工具GUN Flex 和语法分析工具GUN Bison， 并且使用C语言完成。

2.1单词文法描述

1. **Lex文件中使用的描述正规表达式**

x匹配除换行符以外的任何单个字符

\* 匹配前面表达式的零个或多个拷贝

[]匹配括号中的任意字符的字符类.第一个符号是("^"),表示匹配除括号中的字符以外的任意字符.

^ 作为正则表达式的第一个字符匹配行的开头

$ 作为正则表达式的最后一字符匹配行的结尾

{}指示前面的模式被允许匹配多少次

\ 用于转义字符

+ 匹配前面的正则表达式的一次或多次出现

? 匹配前面的正则表达式的零次或一次出现

| 匹配前面的正则表达式或随后的下面表达式

""引号中的每个字符解释为字面意义

/ 只有在后面跟有指定的正则表达式时才匹配前面的正则表达式

{}将一系列的正则表达式组成一个新的正则表达式

1. **单词文法的定义约定：**

digit [0-9]//即单个数字用digit来表示

number [-]?({digit}+)//定义了int|long型的数字（可以匹配正负号）

letter [a-zA-Z]//匹配了字母A-Z的大写和小写情况

identifier {letter}({digit}|{letter}|\_)\*//根据letter来定义出变量的表达式

blank [\t\x0B\f\r\x20]+//将空格 制表符 换行符等进行匹配（它们地位相同）

notes \{.\*\}//匹配注释

1. **文法构造的规则：**

词法分析器构造的时候，根据的是由高到低的顺序进行规则制定，所谓的由高到低指的是，根据先将注释使用规则匹配掉，然后在剩下的文法中进行关键字匹配，具体的关键字见下表格：

|  |  |  |
| --- | --- | --- |
| 关键字 | 匹配规则 | 关键字的描述 |
| INT | number | 代表一切的整型数字（正负都有） |
| FLOAT | [+-]?({digit}+"."{digit}\*|{digit}\*"."{digit}+) | 代表一切的浮点型数字（正负都有） |
| SEMI | ; | 代表分号 |
| COMMA | , | 代表句号 |
| ASSIGNOP | = | 代表等号 |
| RELOP | >|<|>=|<=|==|!= | 代表各种判断符号 |
| PLUS | + | 四则运算的加号 |
| MINUS | — | 四则运算的减号 |
| STAR | \* | 四则预算的乘号 |
| DIV | / | 四则运算的除号 |
| AND | && | 与运算 |
| OR | || | 或运算 |
| DOT | . | 结构体的点 |
| NOT | ! | 求非 |
| TYPE | int|float | 类型（整型，浮点） |
| LP | ( | 左括号 |
| RP | ) | 右括号 |
| LB | [ | 左中括号 |
| RB | ] | 右中括号 |
| LC | { | 左大括号 |
| RC | } | 右大括号 |
| STRUCT | struct | 结构体标识 |
| RETURN | return | 返回标识 |
| IF | if | 判断语句 |
| ELSE | else | 判断分支 |
| WHILE | while | 循环语句 |

1. **构造的优先级简述：**

构造时候，要根据一定的规则进行匹配，在匹配时候，将int和float分开进行了匹配，然后指定出的优先级从高到底规则如下：

1. 统计换行符

在词法分析阶段最开始统计行数，这个时候统计出来的行数是最准确的，因为文本在最开始时候没有被进行任何修改。统计换行符的个数来确定行数，每一个\n代表一行，在开始时定义一个整型的yycolum=1，每次匹配到一个换行符就累加，可以实现行号的精确计数。

1. 匹配注释

因为注释中可能有任何文字内容，如果这些文字内容是关键字或者数字等的话，会将注释中的内容是被为关键字或者各种符号，导致分析出错。

注释分为两种，一种是//开头的行注释，另一种是/\*注释注释\*/的格式；

第一种注释的匹配方法是直接判断//到行的结尾符号\n,中间可以包含任意字符的※闭包。

第二种注释是匹配/\* 与\*/号，两者之间的所有文字为注释（如果一个/\*后有多个\*/则会有限匹配第一个，在gcc编译器进行常规C语言编译时亦是如此。）

1. 匹配各类型关键字

关键字的匹配理论可以在任何地方进行，但是在实际编写中放到ID（变量符号）之前匹配，可以保证匹配规则的正确。

1. 匹配ID
2. 匹配数字

数分为整数和浮点数，优先匹配浮点数，因为如果先匹配整数的话，会导致浮点数的整数部分和小数部分都被匹配为一个整数，而中间的点被匹配为DOT，其中浮点数的规则为：[+-]?({digit}+"."{digit}\*|{digit}\*"."{digit}+)，整数的匹配规则为：[+-]?{digit}+

1. 匹配各类型符号

将各类型的符号进行匹配，从而得出最后结果

1. 报错处理

如果在上6步之后还存在没有匹配的短语，则该短语不是我们的C--文法能够识别的，从而在上述六步后，匹配到的任何字符都会转入出错处理。

2.2语言文法描述

1. **语法分析的传入参数：**

语法分析的参数是通过词法分析后传入的，所以需要将词法分析中的匹配后行为规则进行进一步的修改，在lexical.l中增加对syntax.tab.h的引用，并且让flex源代码中的规则部分每一条都返回一个相对应的词法单元，如下所示：

%{

#include”syntax.tab.h”

…

}

…

%%

“+” {return PLUS;}

“-” {return MINUS;}

“+24{return ID}”

….

1. **Bison编写时语法规则**

Bison文件的编写和上边词法分析的flex文件的编写格式相同，我们只需要关注语法规则即可

语法规则来自实验参考书附录，主要分为如下六部分：

1. High-level Definitions

这一部分包含了本语言中的高层（全局变量和函数定义）语法。

1. Specifiers

这一部分主要与变量的类型有关

1. Declarators

这一部分的产生式主要与变量和函数的定义有关

1. Statements

这一部分主要和语句有关

1. Local Definitions

这一部分的产生式主要与局部变量的定义有关

1. Exp

主要和表达式有关

1. **冲突处理**

Bison采用的是LR分析法，而且存在一些递归调用，在编写的时候不可避免的会出现一些移进-规约冲突和规约-规约冲突，故在编写时需要注意根据优先级更改结合性，消除直接递归等措施来避免冲突。

1. **结合性处理**

大部分的移进规约冲突和规约规约冲突可以通过修改文法的结合性的优先级来进行避开，例如本实验中的C—文法中的Exp生成部分，是冲突的重灾区，解决的办法就是定义文法的结合性，根据C—文法的定义表，得出结合性如下：

%left <a>LP RP LB RB DOT

%right <a>NOT

%left <a>MINUS DIV PLUS STAR

%left <a>RELOP AND OR

%right <a>ASSIGNOP

其余符号的结合性不需要进行修改，顺序写下即可。

1. **判断语句的移进-规约冲突的解决**

判断语句存在的冲突是通过修改优先级结局不了的，此时需要进行文法的局部修改(增加优先级的判断)，修改前后对照如下：

修改前：

Stmt: IF LP Exp RP Stmt

| IF LP Exp RP Stmt ELSE Stmt

修改后：

%nonassoc NOELSE

%nonassoc ELSE

Stmt: IF LP Exp RP Stmt ELSEFU

ELSEFU: ELSE Stmt|%prec NOELSE;

当语法分析程序读到IF LP Exp RP时候，优先选择移入else 从而避免bison中报告冲突

1. **Bison工作方式**

bison编写程序后，规约的步骤一般如下：

1. 读取/解析命令行选项 Main.getargs(), 打开文件 Files.openfiles()

这些是程序基本的准备工作, 一般不属于算法范畴, 我们略去不述.

2. 读入文法文件 Reader.reader(), 文法文件一般以 .y 为后缀.

读取时建立内存的一些中间数据结构(主要是单链结构), 后面详述.

3. 将第2步读入的数据进行检查, 消除无用的,错误的产生, 建立/转换为适合

计算 LR 的数据结构(主要是单链=>数组结构). 预先计算一些辅助数据.

4. 计算 LR0 状态集, 结果可能是一个非确定的(有冲突的)有限状态机.

5. 转变第 4 步的状态机为确定的 LALR 状态机.

6. 如果第 5 步中有 s/r, r/r 冲突, 则解决冲突.

7. 输出及别的收尾工作. 一般略去不细述了.

1. **语法树的生成**

语法树在语法分析时候是一个重要的组成即代表了整个文法的框架组成，又将各个有用的信息存储到各个节点上，对后边的语义分析起到了至关重要的作用，在语法分析时候，生成一个语法树，可以清晰的看出规约过程是否正确，文法中各部分间的联系等，所以语法树的设计对于整个实验是至关重要的。

语法树的生成方式是利用数据结构中的树的模型，每次扫描都将信息加入到一个节点中，最后生成语法树，遍历语法树。

树中每个节点都是一个子节点结构体，定义如下：

struct ast

{

int line; //行号

char\* name;//语法单元的名字

struct ast \*l;//左孩子

struct ast \*r;//右孩子

union//共用体用来存放ID/TYPE/INTEGER/FLOAT结点的值

{

char\* idtype;

int intgr;

float flt;

};

};

每次生成都会将各个信息填入，然后将节点串联，最后生成一个语法树。

1. **语法树的概述**

语法树是在进行词法分析后，语法分析的时候，一边进行语法分析，一边生成的语法树，每规约一条语句，都将相应的有用信息加入到语法树的相应结点中。叶子节点代表了一个有实际含义的终结符，而中间结点可以看做是规约的过程。

创建结点的方式如下（举例）：

Exp: Exp ASSIGNOP Exp{$$=newest("Exp",3,$1,$2,$3);}

1. **错误恢复（多处报错）**

当输入文件中出现语法错误的时候，Bison总是会让他生成的语法分析程序尽早的报告错误。每当语法分析程序从yylex()得到了一个词法单元，如果当前状态没有针对这个词法单元的动作，那么久认为输入文件里出现了语法错误。

要实现在多个地方报错，首先的目标就是实现“错误恢复”，即在及时已经检测到错误的情况下，还是继续进行语法分析，此时就需要在规则中添加匹配错误信息的规则，添加例子如下：

| error MINUS Exp{yyerrok;yytest\_zy=1;}

例如此例子可实现在减法的第一个操作数出现错误的情况下，讲该错误匹配，然后继续进行后边的匹配，检查后续代码的语法情况，直到遇到error都无法匹配或者结束，然后再进行错误输出，即可实现多处报错。

1. **传入信息到树的结点**

因为如果一个一个数值传入结点的各类信息是很复杂的事情，所以在编写的时候听取了别人的建议，采用了变长列表的方法，将信息通过各个规约语句的所包含符号个数，使用结构体的方式进行变长传值。

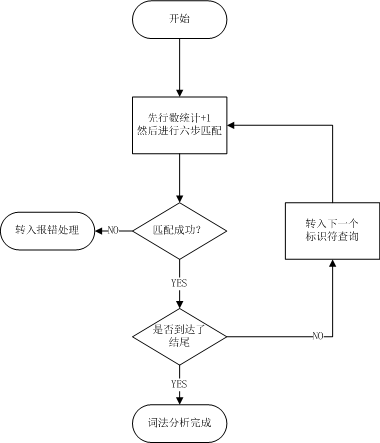
结构体的参数定义如下：

struct ast \*newest(char\* name,int num,...);

其中name为传入的语法单元的名字，num为边长结点的个数，后边跟变长结点。

2.3 词法分析器的设计

词法分析器的结构设计框架图如下：



词法分析器的设计采用flex工具完成，我们只需要进行规则的写入，出错的处理即可，flex的书写规则如下：

{definitions}

%%

{rules}

%%

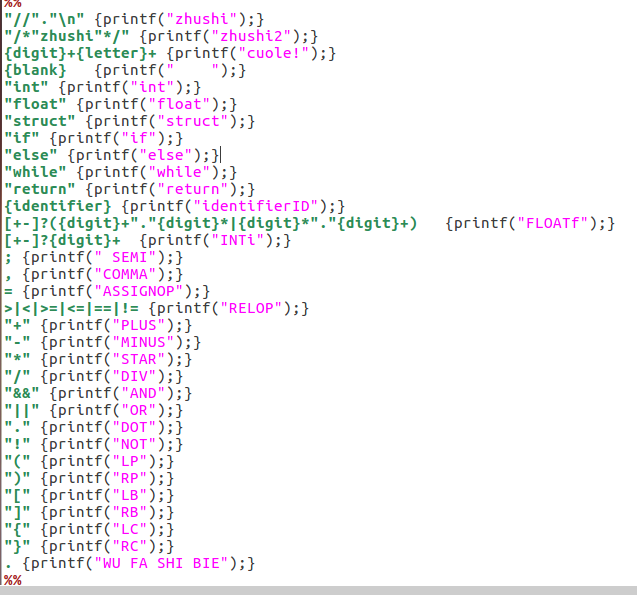
{user subroutines}

书写内容则按照2.2中的描述进行编写。

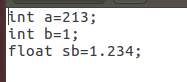
除了int float id需要自己想规则来进行匹配外，其余均在实验指导书上给出，没有难度。

在对C—语法有了了解后，先进行规则查询，使用printf函数输出匹配情况，查看匹配是否成功。

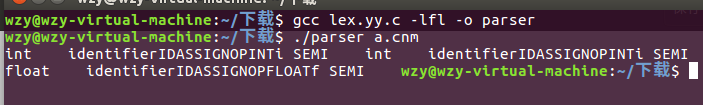
设计代码如下：



尝试输出结果如下：



如果对上述语句进行词法分析，有如下结果：



可以看出，分析成功，则设计方案可行

2.4 语法分析器设计

（1）语法分析器是利用bison直接生成的，采用的是自底向上的上下文无关文法的LALR(1)分析方法，输入的接口来自于上边的词法分析的结果，此时需要将词法分析中的输出到屏幕的匹配信息换成return XXX以便yylex()带出该函数供语法分析使用。

所以词法分析部分代码修改如下：

"int" { yylval.a = newest("DEF\_INT",0,yylineno);return DEF\_INT;}

"float" { yylval.a = newest("DEF\_FLOAT",0,yylineno);return DEF\_FLOAT;}

"struct" { yylval.a = newest("STRUCT",0,yylineno);return STRUCT;}

"if" { yylval.a = newest("IF",0,yylineno);return IF;}

将返回值写为一个个的标识符，通过return带出。

（2）在完成了词法分析器修改后，syntax.y的语法分析文件中应该将相应的返回标识符按照优先级顺序进行列出（根据不同的结合性）（%token %left %right）

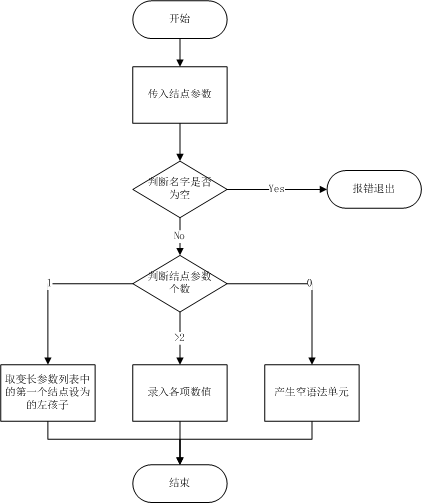
然后将C--文法规则写入到代码段的对应规则部分，每条规则后边加{}备后边生成语法树和语义分析使用。

（3）将错误恢复加入

（4）加入树的节点生成

树的结点生成是通过加入到语义中形成的，每次规约完毕一个句子，则调用节点生成函数。

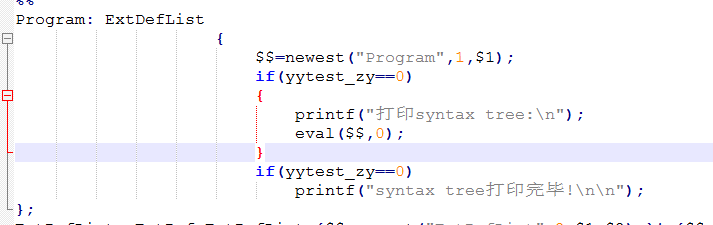
语法生成函数构造方式：



（4）构造完毕后，需要进行语法树的遍历输出，语法树的遍历输出有一个前提条件，就是语法分析必须通过，而语法树是在语法分析的时候同步生成的，所以如果报错，也可能存在语法树的输出（但是不完全），这是我们不希望看到的，所以在语法规则匹配的时候，需要做错误检查。

在这儿错误检查的方法很简单，就是添加一个全局变量error\_flag，如果规则中匹配到了错误恢复，则将error\_flag的值置1，这样，在遍历语法树的时候，就可以查看语法树是否生成完整，如果成功则打印，否则不打印，输出相应错误信息。

输出语法树是在从下到上的语法分析的最后一步分析完成后进行，所以对应的就是规约的开始符，即遍历添加到此处（遍历函数为eval()）：



遍历函数eval递归打印左右结点，输出整个树之后退出。

至此，语法分析完成。

2.5语法分析器实现结果展示

# 3语义分析

3.1语义表示方法描述

本次实验为实验二，是在实验一的词法分析器和语法分析器的基础上进行的增加，对C—的的源代码进行语义分析和类型检查，并且打印出分析结果。

语义分析的工作是在语法分析的基础上，不再借助原有工具，而是通过自己添加代码的方式进行扩展和补充。

本次实验的完成目标是对C—代码的语义分析，其中生成的C—代码要求支持的项目如下：

1. 整型和浮点型变量的定义。
2. 数组的定义（仅限一维）。
3. 整型和浮点型不能相互转换和赋值运算
4. 任何函数只能一次定义，无法进行函数声明
5. 支持部分作用域得限定
6. 结构体采用名等价的方式限定
7. 函数体无法进行嵌套定义
8. 结构体可以嵌套定义
9. 作用域不会重名

主要的分析方式是在语法树的生成过程中，对其中的结点进行归纳和遍历，得出最终的各个符号表，然后将它们打印出来。

语义分析也叫做上下文相关分析，这部分内容可以独立于前变的分析存在，也可以在做语法分析的时候进行分析，但是这样会使得最后分析较为复杂，所以最好还是将语义分析当做语法分析的后即内容。

在词法分析的过程中，我们借用了正则文法；在语法分析中，我们借助了上上下文无关文法；现在到了语义分析部分，使用的方法是上下文有关文法。

目前采用较为多是属性文法，有两种类型的属性，一是综合属性，二是继承属性；在有了属性的定义方法后，我们采用语法制导的翻译模式(SDT)，在这中，将计算的属性值用“{}”括起来（实际上在语法分析的时候生成语法树的时候就已经使用到了。）。

E

T

R

T

{print(‘9’)}

-

9

5

R

{print(‘5’)}

{print(‘-’)}

R

+

T

2

{print(‘2’)}

{print(‘+’)}

ɛ

3.2符号表结构定义

语法分析的各类信息存储都在符号表中，一个号的符号表不仅可以服务于语法分析，更可以服务于后边的中间代码生成和目标代码生成，一个较为实用的符号表中应该包括的内容有：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等。还需要包含一系列的特性信息：上述名字的种类，具体类型，维数，参数个数，数值及目标地址等。

在此处，因为编写的是C——文法，文法本身结构较为简单，为了维护和编写方便，采用较为简单的链表方式进行定义：

采用多条链表的方式进行定义：变量一个表、结构体一个表、数组一个表、函数一个表、形参一个表等。

单个符号表中的所有内容通过链表串起来，插入一个新的符号只需要将该符号放在链表的表头，时间复杂程度是O(1)。在链表中查找查找一个符号需要对这个链表进行遍历，时间复杂度是O(n)，删除一个符号只需要将该符号从链表中去掉。

链表虽然存在效率较低的缺点，但是鉴于文法较为简单，实际上对速度的影响几乎可以忽略不计，所以被最终采纳。

其中的多个链表的定义如下：

1. 符号变量链表

/\*变量符号表链表\*/

struct var{

char\* scope;//作用域

char\* name;//变量名

char\* type;//变量类型

struct var\* next;

}

1. 形参符号链表

/\*形参变量符号表链表\*/

struct xcvar{

char\* scope;//作用域

char\* name;//变量名

char\* type;//变量类型

struct xcvar\* next;

}

1. 实参符号链表

/\*实参符号表列表\*/

struct scvar{

int tag;//1代表变量，２代表数字

char\* type;//传入值的类型

char\* name;//传入变量的名字

float number;//传入的数值

char\* scope;//所属函数

struct scvar\* next;

}

1. 函数符号链表

/\*函数符号表链表\*/

struct func{

int tag;//标志是否定义

char\* name;//函数名

char\* type;//函数类型

char\* rtype;//实际返回值类型

int paramnum;//形参个数

struct func\* next;

}

1. 数组符号链表

/\*数组符号表链表\*/

struct array{

char\* scope;//作用域

char\* name;//数组名

char\* type;//数组类型

int size;

struct array\* next;

}

1. 结构体符号链表

/\*结构体符号表链表\*/

struct struc{

char\* name;//结构体名

char\* type;//结构体类型

struct struc\* next;

}

3.3错误类型码定义

错误类型码和是按数中相同，增加了语法分析的错误分析，如果有语法错误，则将其找出，然后进行语法报错。

|  |  |  |
| --- | --- | --- |
| 错误类型 | 错误原因 | 显示输出 |
| TYPE1 | 操作变量未定义 | Undefined variable “x” |
| TYPE2 | 引用函数未定义 | Undefined function “xx” |
| TYPE3 | 重复定义的变量 | Redfined variable “x” |
| TYPE4 | 重复定义的函数 | Redefined function “xx” |
| TYPE5 | 赋值类型不匹配 | Type mismatched for assignment |
| TYPE6 | 非左值表达式在左边 | Left-hand side error |
| TYPE7 | 操作数类型不匹配 | Type mismatched for operads |
| TYPE8 | 返回值类型不匹配 | Type mismatched fot return |
| TYPE9 | 函数调用时候数目不正确 | Function is not applicable for arguments |
| TYPE10 | 使用[]访问了非数组型变量 | “x” is not a array |
| TYPE11 | 对非函数使用了() | “xx” is not a function |
| TYPE12 | 使用小数访问了数组元素 | “x.x” is not a integer |
| TYPE13 | 对非结构题使用了. | Illegal use of “x” |
| TYPE14 | 访问域出错 | Non-existent field “n” |
| TYPE15 | 结构体中的变量重复定义 | Rdefined filed of “x” |
| TYPE16 | 两个结构体重复定义 | Duplicated name “Position” |
| TYPE17 | 结构体未定义 | Undefined structure “Position” |
| SYNTAX ERROR | 语法出错 | 报错到相应行 |

3.4 语义分析实现技术

本次实验在上次实验的基础上进行，尤其需要的是语法树，因为本次实验仍然需要对语法树的变历。

每当遇到语法单元Extdef或者Def，就说明该节点的子节点包含了变量或者函数的定义信息，这是后话应当将这些信息通过对子节点的遍历提取出来并判断类型，然后将它们插入到符号表中。每当遇到语法单元Exp，说明该节点及其子节点们的遍历提炼出来的是对变量和函数的使用。

结构体定义语句如下：

STRUCT OptTag LC DefList RC

由OptTag节点中获得结构体名称，要求名称不能重定义，当规约到当前节点时说明缓冲链表中的变量与数组为结构体内的域，将其加入到结构体的内部变量数组链表中。

函数调用语句如下：

Exp：ID LP RP|ID LP Args RP

首先判断函数表中，当前函数名是否存在，如果不存在，这判断其他表中是否存在同名项，如果存在则报错函数调用操作符的错用，否则报错为函数未定义。如果函数名在函数表中存在，则进一步判断实参与形参是否匹配，如果匹配，则将生成的Exp节点的类型赋为函数返回值的类型，以进行后续表达式求值的类型匹配。

错误类型的判断：

1. 语义错误：设置一个全局变量flag，如果查找到了语义错误，则该flag值置1，然后停止遍历，进行报错操作。
2. 17中类型的错误：分别在该中表达式的规约语句处进行判断，如果出错，则输出报错语句，而且将flag置1，保证不输出语法树和符号表。

符号表的查重和遍历：

符号表的查重需要对符号表进行遍历，这儿采用的方式是进行从头到尾的遍历查询方式。

定义域的创建方式：

定义域分为两种：结构体内的和非结构体的；如果是结构体内的变量，则吧他们的作用域设定为结构体的名字。如果是结构体外边的变量，则统一默认为全局变量。

对左值表达式的修改：

例如x = 3，是合法的，而3 = x 是不合法的。。。这类型的错误如果在语义分析结点进行查找是较为困难的，所以只在语法分析阶段进行分析：赋值号左边出现的只可以为ID、Exp LB Exp RB以及Exp DOT ID，而不能是其他形式的语法单元组合。

每一个链表都附带包含了成套的匹配函数：

查询是否重复定义函数：

Is\_var \_define:

查询定义域是否定义，如果定义则返回，否则返回空。

查询定义域的函数是

Is\_scope\_define:

查询定义域是否正确。

3.5语义分析结果展示

# 4中间代码生成

### 4.1中间代码格式定义

本次实验为第三次实验，任务是在词法分析、语法分析和语义分析的基础上，将C—源代码翻译为中间代码。中间代码的表示方式是三地址码（线性型）格式。

中间代码的存储格式：由于第三次实验是对语法树的遍历生成的，所以中间代码的存储方式推荐使用哈希表，最后的生成结果就保存在了哈希表中。

表的定义如下：

### 4.2中间代码生成规则定义

中间代码的生成规则采用书中的翻译模式实现，其中的一些特殊约定有：

1. 不会出现注释、八进制或十六进制整型常数、浮点型常数或者变量。
2. 不会出现结构体和高维的数组
3. 任何函数参数只能是简单变量，结构体和数组都不会作为参数传入函数中
4. 没有全局变量的使用，所有变量均不重名
5. 函数不返回结构体或数组类型的值
6. 函数只会进行一次定义
7. 输入文件中不包含任何词法，语法或者语义错误

中间代码的形式和操作规范如下：

|  |  |
| --- | --- |
| 语 法 | 描 述 |
| LABEL X: | 定义标号x |
| FUNCTION F: | 定义函数f |
| x := y | 赋值操作 |
| x := y + z | 加法操作 |
| x := y – z | 减法操作 |
| x := y \* z | 乘法操作 |
| x := y / z | 触发操作 |
| x := &y | 取y的地址赋给x |
| x := \*y | 取以y值为地址的内存单元的内容赋给x |
| \*x := y | 取y值赋给以x值为地址的内存单元 |
| GOTO x | 无条件跳转至标号x |
| IF x [relop] y GOTO z | 如果x与y满足[relop]则跳转至标号z |
| RETURN x | 退出当前函数并且返回x值 |
| DEC x [size] | 内存空间申请，大小为4的倍数 |
| ARG x | 传入实参x |
| x := CALL f | 调用函数，并且将其返回值赋给x |
| PARAM x | 函数参数声明 |
| READ x | 从控制台读取x的值 |
| WRITE x | 向控制台打印x的值 |

输入格式：程序的输入是一个包含C—源代码的文本文件，程序需要能够接受一个输入文件名作为参数。

输出格式：将程序的运行结果输出到文件。

### 4.3 中间代码生成过程

中间代码生成不是一个独立的程序，而是建立在语法分析器框架中的语义动作。可以把编译器比作一个流水线，源程序是源材料，词法分析器是过滤器，源程序经过词法分析器后形成记号流。语法分析器是一系列相互相关的函数，控制流在这些函数中的转移过程就是语法分析的过程。记号流比作精炼过的材料被送到语法分析器这个流水线上流动。如果没有中间代码生成动作，语法分析器就像是只有履带没有加工机的流水线，记号流流过之后没有任何变化。

由上面的比喻可以看出，中间代码生成和语法分析应该构成一遍，以词法分析生成的记号流作为输入，以某种表示程序结构的中间表示（语法树或三地址码）作为输出。生成的中间代码是介于源代码和目标代码中间的一种结构化表示。它的意义在于能够把前端和后端分开，提高编译器的可移植性（因为结构清晰，对于编译器研究者来说也提高了可读性）和易优化性（对中间代码的优化可以独立于机器）。

三地址码的实现方法为四元式。四元式是带有四个域的记录结构，即op，arg1，arg2以及result。其中视三地址码的类型，某些域可能为空。arg1，arg2以及result域的内容正常情况下是指向这些域所代表的名字在符号表表项的指针。这样的话，临时名字在生成时一定要被填入符号表。对于支持作用域规则的语言，生成三地址码时需要有一个符号表栈，栈顶的符号表代表当前作用域。源代码中的Par\_table\_chain结构实现了这个栈及其必须支持的操作。

变量名采用t加上数字的形式，为所有的临时变量名创建一个符号表，记其在符号表中的位置，这个位置是一个在整数v，从0开始，输出时，输出’t’+v,就可以输出t0,t1这种形式，而且有效地避免了变量名的重复。   
方法二：直接将place设置成整数类型，再设置一个标记数组used[]，用了一个整数i，就在标记数组中将used[i]=-1,表示ti已经被用过了。这样一来，创建一个临时变量，相当于设置place的值，然后更改used[]数组的值。

### 4.5 中间代码生成结果展示

# 5目标代码生成

### 5.1基本思路

由于时间紧迫，没有进行目标代码的生成实践，但是对目标代码的生成进行了一定的理论分析：

1. 算术运算指令:

加:add s1,s2,s3: s1=s2+s3;

Addi s1,s2,#i: s1=s2+i;

减:sub s1,s2,s3: s1=s2-s3;

乘:mul s1,s2,s3: s1=s2\*s3;

除:div s2,s3 mflo s1: s1=s2/s3;

1. 赋值指令:

move s1,s2: s1=s2;

3.内存操作指令:

读内存指令:lw s1,off(s2): s1=\*(s2+off);

写内存指令:sw s1,off(s2): \*(s2+off)=s1;

4.立即数指令:

li s1,#i: s1=i;

5.无条件跳转指令:

jr $ra: 跳转到$ra存的地址

jal name: 跳转到标号name处,并保存返回地址.

j label: 无条件跳转到标号label;

1. 条件跳转指令:

beq s1,s2,label: s1==s2时跳转到标签label

bne s1,s2,label: s1!=s2时跳转到label

bgt s1,s2,label: s1>s2时跳转到label

blt s1,s2,label: s1<s2时跳转到label

bge s1,s2,label: s1>=s2时跳转到label

ble s1,s2,label: s1<=s2时跳转到label

如果以后有时间则继续编写。

# 6结束语

6.1 实践课程小结

编译原理实验随着学期的结束而结束了，在本学期的编译原理实验中，主要学到了如下知识：

1. 编译器的构造是通过四个步骤完成的：词法/语法分析、语义分析、中间代码生成和优化、目标代码生成
2. 在词法语法分析时候，利用工具flex和bison进行，完成还是较为顺利，实现了词法语法分析，语法树的生成；最后可以实现错误恢复和顶点报错。
3. 在语义分析时候，利用链表，构造了符号表，通过遍历语法树的方式进行构造，最后生成了多链表式的符号表。最后可以实现嵌套作用域的结构体、函数、数组等等多种语言方式的支持。
4. 在中间代码生成的时候，利用多次遍历语法树的方式，生成了四地址码（三元表达式），可以将中间代码输出到文件，并且进行了细微的优化。
5. 由于最后时间不足，所以没有进行目标代码的生成编写，只是设计了大体思路，如果以后有时间，将会继续完成，做一个完成的C—编译器出来。

6.2自己的亲身体会

本实验是编译原理课程的配套实验，实验内容是设计一个编译器，通过分步的方式进行设计，但遗憾的是由于时间不足，最终只做到了中间代码生成，不过整体实验的趣味性还是很强的，只是由于时间紧迫，没有太多的空闲来仔细研究，例如语义分析，除了课上的8个学时外，可有整整做了二十多个小时才将其完成，中途出现过不计其数的bug，在一步一步的过程中，不仅仅理解了编译器的生成过程和工作原理，更对自己的编程能力有了很强的提升，对于gcc和gdb的调试也有了一定程度的研究。

虽然本次实验对我们的提升很大，但是如果将时间放宽松一点，课时放多一些，相信效果会更好，在对flex和bison研究的时候，完全就是“摸着石头过河”，根本不知道下一步要干啥，这一步干了之后有什么用，仅仅是研究他们的格式和功能就花费了很长的时间，而这些东西如果提前告知的话，完全可以直接拿来用；在进行语义分析的时候，还没有学习符号表，更不用说建立了。而且中间遇到了例如“悬挂指针”导致的“段错误”等等等等。

其实实验中设计了很多，也思考了很多，包括符号表的生成方式，生成内容，怎么进行嵌套引用，怎么判断是否重复等等，但是由于实验时间紧迫，想到什么就添加什么，最后也就没有记录，这可以说是本次实验的一大遗憾，没有将自己的实验过程点点滴滴详细记录。

总之，虽然这次实验还缺最终还差目标代码生成没有完成，但是收获还是很大的，对编译器的理解，对C语言的运用，对结构体的使用，都有了长足的进步，在今后的学习和实验中，我将会把编译原理实验学到的东西运用进去，实现活学活用，让编译成为一门艺术。

**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

**附件：源代码**