实验手册

一、实验目的

1. 通过动手实践，使学生理解编译的基本过运用各个编译阶段的功能，能够利用LEX(FLEX)和YACC(BISON)等经典工具，设计并实现给定语言的词法分析、语法分析、语义分析等功能。
2. 通过分析和修改PL语言的编译程序和解释程序，获得综合利用编译程序设计的知识进行分析、设计、实现和维护编译程序的能力。
3. 通过课程实验，提高利用理论知识和自动化工具解决复杂问题的能力，加强从语言翻译和表示变换的角度对计算的理解。

二、实验内容及要求

(一) 词法分析程序要求

1. 编写一个程序对TEST语言书写的源代码进行词法分析，并打印分析结果。
2. 程序能够检查源代码中可能包含的词法错误：
    1. 能够识别词法中未定义的字符；
    2. 能够识别注释。

3. 词法分析能够定位错误位置。

(二) 语法分析程序

1. 结合上个实验完成的词法分析程序，编写一个程序：能够对TEST语言源代码进行语法分析，并打印分析结果。
2. 程序要求:
    1. 建立抽象语法树并输出；
    2. 定位错误信息。

(三) 语义分析

1. 结合上面两个实验完成的词法、语法分析程序，编写一个程序对TEST语言书写的源代码进行语义分析以及类型检查，并打印分析结果。
2. 程序要求：
    1. 实现对整型（int）和浮点型（float）变量的类型检查，两类变量不能相互赋值及运算；仅整型及浮点型变量才能参与算术运算；
    2. 判断源代码是否符合以下语义假设并给出相应错误具体位置：函数仅能定义一次、程序中所有变量均不能重名、函数不可嵌套定义。
3. 能够定位错误位置。

(四) 中间代码生成

1. 结合已完成的词法、语法、语义分析程序，编写程序将TEST语言的源代码翻译为中间代码，并翻译结果。
2. 程序要求：能够输出抽象语法树及四元式的中间代码，至少包含以下代码类型：赋值语句、算术运算操作、跳转语句、分支与循环语句。

三、实验原理、实验方法

(一) 词法分析实验原理

> 本次实验采用flex词法分析生成器进行词法分析

1. **flex简介**
   在编程语言中，词法分析器（Lexical Analyzer）是将输入的源代码分割成有意义的单元（Token）序列的程序。Flex是一种常用的词法分析器生成器，它可以根据用户提供的正则表达式规则来生成词法分析器。Flex生成的词法分析器可以用于解析各种编程语言的源代码，如C、C++、Java等。
2. **flex工作原理**
   Flex的工作流程是：先将用户提供的正则表达式规则转换成有限状态自动机（Finite State Automaton, FSA），然后根据该自动机生成词法分析器。词法分析器会读取输入的源代码字符流，并根据自动机中的状态转换规则，识别出源代码中的各个单元。生成的词法分析器通常会将识别出的单元作为Token返回给语法分析器（Parser），由语法分析器进一步处理。

(二) 语法分析/语义分析实验原理

> 本次实验采用了flex+bison实现语法分析

1. **bison简介**
   Bison是一个自由软件，是GNU软件开发工具中的一员，它是一个用于生成语法分析器的工具。Bison可以根据用户提供的语法规则文件，自动生成语法分析器的C源代码。Bison生成的语法分析器通常用于解析各种编程语言的源代码，如C、C++、Java等。
2. **bison工作原理**
   Bison的工作流程是：先将用户提供的语法规则转换成上下文无关文法（Context-Free Grammar, CFG），然后根据该文法生成语法分析器。语法分析器会读取词法分析器返回的Token序列，并根据文法规则进行语法分析。如果输入的源代码符合文法规则，则语法分析器会生成一棵抽象语法树（Abstract Syntax Tree, AST），该树可以用于进一步的语义分析和代码生成。

四、方案设计实现

(一）词法分析设计实现

flex文件编写

Flex文件（通常以.l为扩展名）包含了词法分析器的规则，也就是正则表达式和对应的处理代码。Flex文件通常分为三个部分：定义部分、规则部分和处理代码部分。

```
定义:definition
%%
规则:rules
%%
处理代码:code
```

1.1 定义部分

> 定义又可以分为四部分：添加头文件、变量声明、设置flex属性、正则表达式定义

1. 头文件和变量声明一般包括在括号%{...%}中

```
/*第一部分头文件和变量*/
%{
   #include<string.h>
   int statue=0;
   char e[30][30];
   int eLine[30];
   int cnt=0;
%}
```

2. 设置flex属性

> flex的选项影响最终生成的词法分析器的属性和行为。这些选项可以在运行flex命令时在终端输入，也可以在.l文件中使用%option指定。
> 本次实验中需要使用到当前分析的词所在的行数。

```
/*flex属性,记录符号所在行号*/
%option yylineno
```

3. 正则表达式定义

> 正则表达式声明方式：`{表达式名称} 正则表达式`

```
/*正则表达式定义代码节选*/
ALPHA [a-zA-Z]
DIGIT [0-9]
ID  {ALPHA}+[a-zA-Z0-9_]*
VALUE [1-9]+{DIGIT}*|{DIGIT}+\.{DIGIT}+
OP  \+|-|\*|<=|<|==|=|>=|>|>>|<<|;|\(|\)
...
```

1.2 规则：规则部分需要包括在两对由两个百分号组成分隔号 %% 中

> 编写规则时，一个规则一行，每行有两部分构成：正则表达式 {动作函数}

```
...
/*规则部分代码节选*/
{ID}   {
            if(statue==1){
            }else{
                printf("line%d:",yylineno);
                printf("(ID,%s)\n",yytext);
            }
        }
{VALUE} {
            if(statue==1){
            }else{
                printf("line%d:",yylineno);
                printf("(VALUE,%s)\n",yytext);
            }
        }
...
```

1.3 处理代码

> 该部分必须定义main函数，用于指定需要扫描分析的文件，可以内部指定，也可以通过控制台传入参数。

```
int main(int argc, char **argv)
{
    /*文件读入，结果输出设置，过程代码略*/
  if (argc>1 && argc<=2){...}/*读入一个文件过程在控制台输出*/
  if (argc>2){...}/*读入若干个文件，将结果输出到最后一个文件*/

  yylex();/*由lex创建的扫描程序的入口点yylex()，调用ylex()启动或者重新开始扫描。*/
  yywrap();/*yywrap()函数用于在输入文件的末尾执行一些必要的清理工作。*/

  if(cnt>=1){...}/*输出错误信息及其位置*/
}
int yywrap(){return 1;}/*返回1，使得扫描程序就返回报告文件结尾*/
```

(二）语法分析设计实现

1. 综述

本次语法分析共编写了4个代码文件：`syntax_tree.l`、`syntax_tree.y`、`syntax_tree.h`、`syntax_tree.c`。其中`syntax_tree.l`文件识别定义的Token，并返回给`syntax_tree.y`文件完成语法分析。同时，为了实现相关功能，还需调用`syntax_tree.h`文件，该文件定义了一些结构体、全局变量、函数声明；以及`syntax_tree.c`文件，该文件实现了上个文件中定义的函数，以及主函数`main`。

2. `syntax_tree.y` 文件编写(flex)

> 该文件是Flex文件格式，定义了词法分析器的规则，和实验一不同的是定义部分和action部分，同时由于flex只是被bison调用，所以无需在该文件中定义main函数。

1. 定义部分

```
/*第一部分 定义部分*/
%{
    #include <stdlib.h>
    #include <stdio.h>
    #include "syntax_tree.h"
    #include "syntax_tree.tab.h"
%}
/*正则表达式略*/
...
```

2. 规则部分

这个文件识别到的Token在语法树中为叶子节点，故该文件需要先将识别到的Token封装成一个节点方便其他文件处理。

```
/*代码节选*/
%%
 /*跳过空白和注释*/
{SPACE} {}
{COMMENT} {}
 /*关键字*/
{TYPE} {yylval.type_tnode=newAst("TYPE",0,yylineno);return TYPE;}
{IF} {yylval.type_tnode=newAst("IF",0,yylineno);return IF;}
{ELSE} {yylval.type_tnode=newAst("ELSE",0,yylineno); return ELSE;}
...
%%
```

3. 处理代码(无)

> flex只是被bison调用，所以无需在该文件中定义main函数。也无需处理代码。

3. `syntax_tree.y` 文件编写(bison)

1. bison文件简要说明

> 一个标准的Bison文件（后缀为.y）通常包含以下几个部分：
>
> 1. 宏定义部分：定义了一些常量和宏，如YYSTYPE、YYLTYPE、YYDEBUG等。
> 2. 语法规则部分：定义了文法的各个产生式规则，使用BNF或EBNF等形式表示。

> 3. 语法动作部分：与每个产生式规则相关联的代码块，用于在语法分析期间执行特定的操作。这些动作通常是在语法分析期间生成抽象语法树的关键步骤。
> 4. 辅助函数部分：包含一些辅助函数，用于在语法动作中执行常见的操作，如内存分配、错误处理等。
> 5. 语法错误处理部分：定义了一些函数，用于在语法分析期间处理语法错误。
> 6. 主函数部分：定义了主函数，用于初始化解析器和执行语法分析。

本次编写的bison文件（即`syntax_tree.y`）将辅助函数部分、语法错误处理部分、主函数部分都编写在c文件（即`syntax_tree.c`）中，故只包含宏定义部分、语法规则部分、语法动作部分。又由于语法规则与语法动作部分的代码结合在一起，故两者将一起说明。

综上，`yntax_tree.y`将分为两部分说明：宏定义部分、语法规则和语法动作部分。

2. 宏规则部分代码（节选）

```
/*宏规则代码节选*/
%{
.../*其他头文件略*/
#include "syntax_tree.h"
%}

%union{
    tnode type_tnode;
    double d;// 这里声明double是为了防止出现指针错误（segmentation fault）
}

/*声明记号*//*节选*/
...
%token <type_tnode> INT FLOAT
%token <type_tnode> SEMI ";" COMMA "," ASSIGNOP "=" PLUS "+" MINUS "-" STAR
"*" DIV "/" AND "&&" OR  DOT "." NOT "!" LP "(" RP ")" LB "[" RB "]" LC "{"
RC "}"
...

/*优先级定义*//*节选*/
...
%left RELOP/* 关系运算符 */
%left PLUS MINUS /* + - */
%left STAR DIV /* * / */
%right NOT  /* ! */

%nonassoc LOWER_THAN_ELSE /*标记为‘nonassoc’的tokon不能同时出现 这里指的
是‘LOWER_THAN_ELSE’与‘ELSE’ 不能同时出现*/
%nonassoc ELSE
```

2. 1 `union{...}`部分代码说明
   在`%union`中我们需要定义除了int外的类型，因为在声明记号时，该记号的类型决定了我们用`yyval`传值的类型。
3. 2 声明记号部分代码说明
   在之前的`.l`文件中我将叶节点(tnode)类型作为传值的内容，因此我在这里将`Token`的类型声明为`tnode`（`%token <type_tnode> ....`）。非终结符同理，用来构造语法分析树。

4. 3 优先级定义部分代码说明
   为了避免语法中的二义性、移进规约冲突，我定义了运算符的优先级、嵌套if-else的优先级。%left表示左结合，%right表示右结合，%nonassoc表示不可结合，解决了运算、if-else嵌套的冲突。

3. 语法规则和语法动作部分代码（节选）

```
%%
/*High-level Definitions*//*外部定义*/
...
/*Specifire*//*内部定义*/
...
/*Declarators*//*内部声明*/
...
/*Statement*//*语句语法规则*/
...

/*Expressions*//*表达式*/
Exp:Exp "=" Exp{
        $$=newAst("Exp",3,$1,$2,$3);
        // 当有一边变量是未定义时，不进行处理
        if($1->type==NULL || $3->type==NULL)
            ...
    |Exp "&&" Exp{$$=newAst("Exp",3,$1,$2,$3); }
    |...
...
%%
```

   3. 1 语法规则部分
      语法规则部分包含若干BNF范式表达式，在上述给出的节选代码中为

```
Exp:Exp "=" Exp
    |Exp "&&" Exp
    |Exp OR Exp
    |Exp RELOP Exp
    |...
```

      即去掉`{ }`包含代码的部分。bison根据所给BNF范式以及flex文件传出的Token识别输入文件的语法规则。

   4. 2 语法动作部分
      语法动作部分紧跟着语法规则，在每条语法规则后`{ }`包含着的就是语法动作。表示在识别到这条规则后，bison文件将执行的动作。
      在上述给出的节选代码中，语法的动作的具体含义为：为所识别的语法规则新建一个语法树节点。其中`$$`表示BNF范式的左部，`$1、$2...`表示BNF范式右部的第1、2...个表达式。
      以`Exp:Exp "=" Exp{$$=newAst("Exp",3,$1,$2,$3);}`为例，该行代码表示为：建立一个新节点，该节点的名称为'Exp'，将识别到的`Exp`、`=`、`Exp`表达式（Token，同时也是一个节点！）设置该新建节点的子节点，同时将这个新节点赋值给`Exp`Token，并将其压入栈中。

4. `syntax_tree.h` 文件编写

该文件定义了一些结构体、全局变量、函数声明。

1. 头文件（节选）

```
#include <stdarg.h> // 变长参数函数 头文件
...
```

由于分析树采用的是孩子兄弟表示法，不清楚每个节点有几个子节点，因此采用变长参数，需要引入头文件 `stdarg.h`。

2. bison相关的全局变量、函数

```
extern int yylineno;// 行数
extern char *yytext;// 文本
void yyerror(char *msg);// 错误处理
```

bison语法规定的变量、函数。

3. 数据结构：抽象语法树（多叉树）

```
// 抽象语法树
typedef struct treeNode
{
    int line; // 行数
    char *name;// Token类型
    int tag;// 1变量 2函数 3常数 4数组 5结构体

    struct treeNode *cld[10];// 使用孩子数组表示法
    int ncld;

    char *content;// 语义值
    char *type; // 数据类型 int 或 float
    float value;// 变量的值

} * Ast, *tnode;
```

4. 主要函数：构造抽象语法树、遍历语法树

```
// 构造抽象语法树(节点)
Ast newAst(char *name, int num, ...);
// 先序遍历语法树
void Preorder(Ast ast, int level);
```

5. 全局变量

```
// 所有节点数量
int nodeNum;
// 存放所有节点
tnode nodeList[5000];
int nodeIsChild[5000];
// bison是否有词法语法错误
int hasFault;
```

6. 辅助函数

```
// 设置节点打印状态
void setChildTag(tnode node);
```

5. `syntax_tree.c` 文件编写

> 该文件实现了syntax_tree.h中定义的函数以及主函数main。

1. 树节点构造函数（代码略）

```
Ast newAst(char *name, int num, ...){...}
```

思路：Ast构造函数采用变长参数，对于构建叶节点、构建父节点能够进行不同的处理，同时判断Token类型。如果有多个子节点，则进行子节点的添加。兄弟孩子表示法，左子树为第一个孩子节点，右子树为该节点的兄弟节点。父节点的大部分参数从第一个子节点处获得。

2. 语法树遍历函数（代码略）

```
void Preorder(Ast ast, int level){...}
```

思路：采用先序遍历，先遍历根节点，再遍历左子树、右子树，并且对不同的节点类型进行不同内容的输出。

3. 辅助函数：设置节点类型：根节点/其他节点 （代码略、思路略）

```
void setChildTag(tnode node){...}
```

4. 主函数（省略部分代码）

```
// 主函数 扫描文件并且分析
// 为bison会自己调用yylex()，所以在main函数中不需要再调用它了
// bison使用yyparse()进行语法分析，所以需要我们在main函数中调用yyparse()和
yyrestart()
int main(int argc, char **argv)
{
    /*此处省略未知错误处理*/
    ...
    for (i = 1; i < argc; i++)
    {
        /*此处省略初始化节点记录列表*/
        ...
        /*此处省略文件相关操作*/
        ...

        /*使用下面两个函数，调用bison进行语法分析*/
        yyrestart(f);
        yyparse();

        fclose(f);/*关闭文件*/

        if (hasFault) continue;

        // 遍历所有非子节点的节点，打印语法树
```

```
        for (j = 0; j < nodeNum; j++)
            if (nodeIsChild[j] != 1)
                Preorder(nodeList[j], 0);
    }
}
```

(三）语义分析设计实现

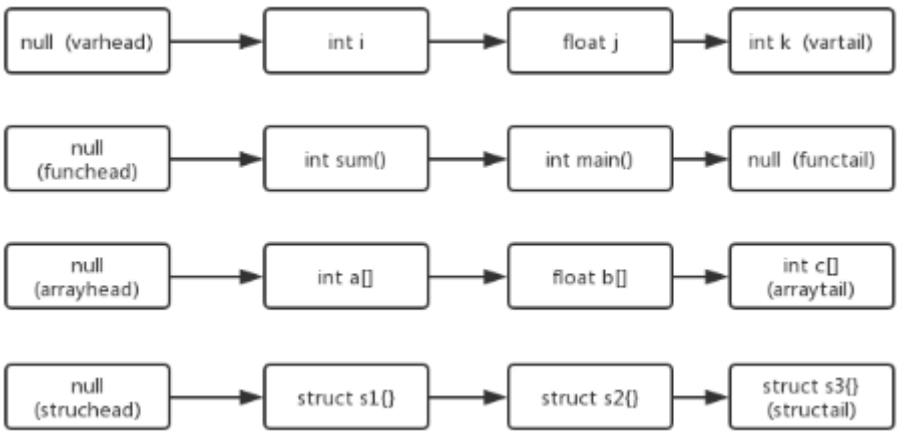
1. 综述

1. 本次实验假定TEST语言有如下特性：
   假设1:整型（int）变量不能与浮点型（float）变量相互赋值或者相互运算。
   假设2:仅有int型变量才能进行逻辑运算或者作为if和 while语句的条件；仅有int型和 float型变量才能参与算术运算。
   假设3:任何函数只进行一次定义，无法进行函数声明。
   假设4:所有变量（包括函数的形参）的作用域都是全局的，即程序中所有变量均不能重名
   假设5:结构体间的类型等价机制采用名等价（ Name Equivalence）的方式
   假设6:函数无法进行嵌套定义。
   假设7:结构体中的域不与变量重名，并且不同结构体中的域互不重名。

   并由如下错误类型：
   错误类型1:变量在使用时未经定义。
   错误类型2:赋值号两边的表达式类型不匹配。
   错误类型3:赋值号左边出现一个只有右值的表达式。
   错误类型4:对普通变量使用“(...)”或“()”（函数调用）操作符。
   错误类型5:函数在调用时未经定义。
   错误类型6:操作数类型不匹配或操作数类型与操作符不匹配。
2. 本次实验实验使用线性链表实现符号表

2. 符号表的实现

> 采用用线性链表实现了符号表，将不同种类的符号组织成不同的表，维护每张表的表头和表尾，从表尾插入，从表头开始遍历。



1. 变量符号表（`syntax_tree.h`）

```
// 变量符号表的结点
typedef struct var_
{
    char *name;
    char *type;
    // 是否为结构体域
```

```
    int inStruc;
    // 所属的结构体编号
    int strucNum;
    struct var_ *next;
}var;
var  *varhead, *vartail;
// 建立变量符号
void newvar(int num,...);
// 变量是否已经定义
int  findvar(tnode val);
// 变量类型
char* typevar(tnode val);
// 这样赋值号左边仅能出现ID、Exp LB Exp RB 以及 Exp DOT ID
int checkleft(tnode val);
```

实现了创建遍历符号、变量类型检查、查看变量是否已经定义、检查赋值号左边变量类型的功能函数。为了实现检查结构体域的功能，在符号表节点中设置了`inStruc`和`strucNum`变量，用于标注该变量属于哪一个结构体。

2. 变量符号表（`syntax_tree.c`省略部分代码）

```
// 建立变量符号
void newvar(int num, ...)
{
    va_list valist;
    va_start(valist, num);

    var *res = (var *)malloc(sizeof(var));
    tnode temp = (tnode)malloc(sizeof(tnode));

    if (inStruc && LCnum){...}// 是结构体的域
    else{...}

    // 变量声明 int i
    temp = va_arg(valist, tnode);
    res->type = temp->content;
    temp = va_arg(valist, tnode);
    res->name = temp->content;

    vartail->next = res;
    vartail = res;
}
// 查找变量
int findvar(tnode val)
{
    var *temp = (var *)malloc(sizeof(var *));
    temp = varhead->next;
    while (temp != NULL)
    {
        if (!strcmp(temp->name, val->content)){
            if (inStruc && LCnum){ // 当前变量是结构体域
                if (!temp->inStruc){...} // 结构体域与变量重名
                else if (temp->inStruc && temp->strucNum != strucNum){...}//
不同结构体中的域重名
                else{...}// 同一结构体中域名重复
            }else {// 当前变量是全局变量
```

```
                if (temp->inStruc){...}// 变量与结构体域重名
                else{...}// 变量与变量重名，即重复定义
            }
        }
        temp = temp->next;
    }
    return 0;
}
```

由于设置了相关属性值，可以实现在寻找变量的同时判断错误类型9和10，这两种错误类型只做提醒，不影响变量的定义以及变量符号表的插入操作。
符号表的插入和遍历操作大致相同，这里展示变量符号表的相关操作，其他符号表只说明不同的地方。

```
// 赋值号左边只能出现ID、Exp LB Exp RB 以及 Exp DOT ID
int checkleft(tnode val)
{
    /*此处省略若干if-else语句实现check功能*/
    ...
}
```

在赋值操作中，左值表示地址，这说明表达式“x = 3”是正确的，而“x + 2 = 3”是错误的。这样赋值号左边仅能出现ID、Exp LB Exp RB 以及 Exp DOT ID，因此需要检查赋值号左边的语法分析树节点的子节点是否符合上述三种情况，用来判断是否为对应的错误类型

3. 函数符号表(`syntax_tree.h`)

```
// 函数符号表的结点
typedef struct func_
{
    int tag; //0表示未定义，1表示定义
    char *name;
    char *type;
    // 是否为结构体域
    int inStruc;
    // 所属的结构体编号
    int strucNum;
    char *rtype; //声明返回值类型
    int va_num;  //记录函数形参个数
    char *va_type[10];
    struct func_ *next;
}func;
func *funchead,*functail;
// 记录函数实参
int va_num;
char* va_type[10];
void getdetype(tnode val);//定义的参数
void getretype(tnode val);//实际的参数
void getargs(tnode Args);//获取实参
int checkrtype(tnode ID,tnode Args);//检查形参与实参是否一致
// 建立函数符号
void newfunc(int num, ...);
// 函数是否已经定义
int findfunc(tnode val);
```

```
// 函数类型
char *typefunc(tnode val);
// 函数的形参个数
int numfunc(tnode val);
// 函数实际返回值类型
char *rtype[10];
int rnum;
void getrtype(tnode val);
```

4. 函数符号表(`syntax_tree.c`)

```
// 创建函数符号
void newfunc(int num, ...)
{
    int i;
    va_list valist;
    va_start(valist, num);

    tnode temp = (tnode)malloc(sizeof(struct treeNode));

    switch {
    case 1://有函数返回值
        if (inStruc && LCnum){...}// 是结构体的域
        else{/*设置为0*/}
        //设置函数返回值类型
        temp = va_arg(valist, tnode);
        functail->rtype = temp->content;
        functail->type = temp->type;

        for (i = 0; i < rnum; i++)
            if (rtype[i] == NULL || strcmp(rtype[i], functail->rtype)){...}//
函数返回值错误

        functail->tag = 1; //标志为已定义
        func *new = (func *)malloc(sizeof(func));
        functail->next = new; //尾指针指向下一个空结点
        functail = new;
        break;
    case 2://无函数返回值
        //记录函数名
        temp = va_arg(valist, tnode);
        functail->name = temp->content;
        //设置函数声明时的参数
        temp = va_arg(valist, tnode);
        functail->va_num = 0;
        getdetype(temp);
        break;
    default:break;
    }
}
```

本次实验中，我对函数的声明和定义分别作了处理（进行`FunDec`和`ExtDef`规约时作不同处理）。在进行`ExtDef`规约时可以获取函数的返回值类型，并且同时判断函数的声明返回类型、实际返回值类型是否相同。在`FunDec`规约时检测函数是否已定义并且设置函数的形参和函数名称。

```
//定义的参数
void getdetype(tnode val)
{
    int i;
    if (val != NULL){
        if (!strcmp(val->name, "ParamDec")){
            functail->va_type[functail->va_num] = val->cld[0]->content;
            functail->va_num++;
            return;
        }
        for (i = 0; i < val->ncld; ++i)
            getdetype((val->cld)[i]);
    }else return;
}
//实际的参数
void getretype(tnode val)
{
    int i;
    if (val != NULL){
        if (!strcmp(val->name, "Exp")){
            va_type[va_num] = val->type;
            va_num++;
            return;
        }
        for (i = 0; i < val->ncld; ++i)
            getretype((val->cld)[i]);
    }else return;
}
//检查形参与实参是否一致,没有错误返回0
int checkrtype(tnode ID, tnode Args)
{
    int i;
    va_num = 0;
    getretype(Args);
    func *temp = (func *)malloc(sizeof(func *));
    temp = funchead->next;
    while (temp != NULL && temp->name != NULL && temp->tag == 1){
        if (!strcmp(temp->name, ID->content))
            break;
        temp = temp->next;
    }
    /*省略未知错误处理*/
    ...
    return 0;
}
```

在Bison代码中，对于函数形参以及实参规约的节点都是父节点，需要用先序遍历获取树结构中所有的参数节点，并进行存储，以便于检测相关类型错误。

5. 数组符号表/结构体符号表（与变量符号表基本一致，故略）

```
...
```

3. 语法分析树的实现

1. 语法树结构体（同语法分析，故略）

```
...
```

2. 语法树实现方式

```
// 表示当前节点不是终结符号，还有子节点
if (num > 0)
{
    father->ncld = num;
    // 第一个孩子节点
    temp = va_arg(list, tnode);
    father->cld[0] = temp;
    setChildTag(temp);
    // 父节点行号为第一个孩子节点的行号
    father->line = temp->line;

    if (num == 1)
    {
        //父节点的语义值等于左孩子的语义值
        father->content = temp->content;
        father->tag = temp->tag;
    }
    else
    {
        for (i = 1; i < num; i++)
        {
            temp = va_arg(list, tnode);
            (father->cld)[i] = temp;
            // 该节点为其他节点的子节点
            setChildTag(temp);
        }
    }
}
```

父节点的类型、语义值可以从子节点处获得。

4. 各个错误类型的处理

> 需要实现在语法分析的同时进行语义分析，因此在规约时对某些类型的节点需要进行符号表插入、遍历以及错误类型的检查工作。

为方便编写、说明定义如下错误类型：

错误类型1:变量在使用时未经定义。

错误类型2:赋值号两边的表达式类型不匹配。

错误类型3:赋值号左边出现一个只有右值的表达式。

错误类型4:对普通变量使用“(...)”或“()”（函数调用）操作符。

错误类型5:函数在调用时未经定义。

错误类型6: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）。

错误类型7:变量出现重复定义。

错误类型8:函数出现重复定义（即同样的函数名出现了不止一次定义）。

错误类型9:结构体中域与变量重名。

错误类型10:不同结构体中的域重名（类型9和10不一定是错误，但需要检查是否重名并进行提示）。

错误类型11:结构体的名字与前面定义过的结构体或变量的名字重复。

错误类型12:函数声明的返回类型和实际返回值类型不一致。

错误类型13:函数定义的形参与调用的实参数量或类型不一致。

1. 错误类型1

```
Exp:ID {
        $$=newAst("Exp",1,$1);
        // 错误类型1:变量在使用时未经定义
        if(!findvar($1)&&!findarray($1))
            printf("Error type 1 at Line %d:undefined variable
%s\n",yylineno,$1->content);
        else $$->type=typevar($1);
    }
```

操作数ID会被规约为Exp，ID就是变量的名称，因此需要在这里检测变量是否已经定义。

2. 错误类型2、3

```
Exp:Exp ASSIGNOP Exp{
        $$=newAst("Exp",3,$1,$2,$3);
        // 当有一边变量是未定义时，不进行处理
        if($1->type==NULL || $3->type==NULL)return;
        // 错误类型2:赋值号两边的表达式类型不匹配
        if(strcmp($1->type,$3->type))
            printf("Error type 2 at Line %d:Type mismatched for assignment.\n
",yylineno);
        // 错误类型3:赋值号左边出现一个只有右值的表达式
        if(!checkleft($1))
            printf("Error type 3 at Line %d:The left-hand side of an
assignment must be a variable.\n ",yylineno);
    }
```

在进行语法分析树建立时修改了节点构造函数，使得子节点类型能够传递给父节点，在检测赋值号两端表达式类型时就可以直接检测表达式节点的type属性是否相同。这里判断了type属性是否为NULL(表达式是否未定义)，如果不进行相关处理，碰到未定义的表达式，其type属性为NULL，直接对NULL使用strcmp函数会报出段错误。错误类型3和错误类型2都是在规约到ASSIGNOP符号时进行检测

3. 错误类型4、5、13

```
Exp:ID "(" Args ")" {
        $$=newAst("Exp",4,$1,$2,$3,$4);
        // 错误类型4:对普通变量使用“(...)”或“()”（函数调用）操作符
        if(!findfunc($1) && (findvar($1)||findarray($1)))
            printf("Error type 4 at Line %d:'%s' is not a function.\n
",yylineno,$1->content);
        // 错误类型5:函数在调用时未经定义
        else if(!findfunc($1))
            printf("Error type 5 at Line %d:Undefined function %s\n
",yylineno,$1->content);
```

```
        // 函数实参和形参类型不一致
        else if(checkrtype($1,$3)){
            printf("Error type 13 at Line %d:Function parameter type error.\n
",yylineno);
        }else{}
    }
    |ID "(" ")" {
        $$=newAst("Exp",3,$1,$2,$3);
        // 错误类型4:对普通变量使用“(...)”或“()”（函数调用）操作符
        if(!findfunc($1) && (findvar($1)||findarray($1)))
            printf("Error type 4 at Line %d:'%s' is not a function.\n
",yylineno,$1->content);
        // 错误类型5:函数在调用时未经定义
        else if(!findfunc($1))
            printf("Error type 5 at Line %d:Undefined function %s\n
",yylineno,$1->content);
        else {}
    }
```

在检测到ID LP ... RP时进行函数调用错误类型的检测，处理方式是当函数已经定义时就不需要检测是否对普通变量进行()操作。另外对于(Args)类型的函数定义规约，需要检测实参和形参类型是否一致。

4. 错误类型6

```
Exp:Exp "+" Exp{
        $$=newAst("Exp",3,$1,$2,$3);
        // 错误类型6:操作数类型不匹配或操作数类型与操作符不匹配
        if(strcmp($1->type,$3->type))
            printf("Error type 6 at Line %d:Type mismatched for operands.\n
",yylineno);
    }
    |Exp "-" Exp{...}/*其他处理方式与"+"类似，故略*/
...
}
```

5. 错误类型7

```
ParamDec:Specifire VarDec {
        $$=newAst("ParamDec",2,$1,$2);
        // 错误类型7:变量出现重复定义
        if(findvar($2)||findarray($2))
            printf("Error type 7 at Line %d:Redefined Variable
'%s'\n",yylineno,$2->content);
        else if($2->tag==4)
            newarray(2,$1,$2);
        else
            newvar(2,$1,$2);
    }
    ;
Def:Specifire DecList SEMI {
    /*处理方式类似，故略*/
    ...
    }
    ;
```

本次实验中假设函数形参也是全局变量，因此在对变量类型错误检测以及变量符号表插入时，需要在两个地方添加代码，分别是变量定义、函数定义时。

6. 错误类型8

```
FunDec:ID "(" VarList ")" {
        $$=newAst("FunDec",4,$1,$2,$3,$4); $$->content=$1->content;
        // 错误类型8:函数出现重复定义（即同样的函数名出现了不止一次定义）
        if(findfunc($1))
            printf("Error type 8 at Line %d:Redefined Function
'%s'\n",yylineno,$1->content);
        // 设置函数名称以及参数列表
        else newfunc(2,$1,$3);
    }
    |ID "(" ")" {
        /*处理方式类似，故略*/
        ...
    }
    ;
```

对函数声明和函数定义（有函数体时）分别进行了处理，函数定义时检测是否重复定义，并且设置函数名称以及参数列表。

7. 错误类型9、10

> 为了实现结构体域的检测，设置了相关的全局变量。在词法分析过程中，当遇到STRUCT TOKEN时，表示有结构体定义，这时将inStruc置1并且`strucNum++`。
>
> 同时为了区分结构体域和结构体声明之外的其他全局变量，我通过两种方式来进行检测，一种是在进行`StructSpecifire:STRUCT OptTag LC DefList RC`规约时，将`inStruc`置0；
>
> 另一种是在词法分析器遇到LC TOKEN时判断`inStruc`，如果是在结构体声明中，则`LCnum++`，如果词法分析器遇到RC并且也在结构体声明中，则`LCnum--`
>
> 当`inStruc`为1且`LCnum`不为0时定义的变量就是结构体的域，此时会调用`findvar`和`findarray`函数进行相关错误类型的判断。

8. 错误类型11

```
StructSpecifire:STRUCT OptTag LC DefList RC {
        // 结构体定义完成，当前在结构体定义外部
        inStruc = 0;
        $$=newAst("StructSpecifire",5,$1,$2,$3,$4,$5);
        // 错误类型11:结构体的名字与前面定义过的结构体或变量的名字重复
        if(findstruc($2))
            printf("Error type 11 at Line %d:Duplicated name
'%s'\n",yylineno,$2->content);
        else newstruc(1,$2);
    }
```

在结构体声明时检测是否有重复定义。

9. 错误类型12

```
Stmt:Exp SEMI {$$=newAst("Stmt",2,$1,$2); }
    |Compst {$$=newAst("Stmt",1,$1); }
    |RETURN Exp SEMI {
        $$=newAst("Stmt",3,$1,$2,$3);
        getrtype($2);
    }
```

在检测到return语句时将返回值类型存入到全局遍历数组中。

```
ExtDef:Specifire FunDec Compst  {
            $$=newAst("ExtDef",3,$1,$2,$3);
            // 设置函数声明的返回值类型并检查返回类型错误
            newfunc(1,$1);
        }
    ;
```

在规约完函数声明以及函数体后可以进行返回类型错误检测，调用`newfunc`函数，传入参数1，`newfunc`函数根据当前函数符号表中存储的返回类型值和返回值类型全局数组存储的类型值做对比，检测返回值类型不匹配的错误。

五、测试、运行结果

(一)词法分析

1. 测试代码

```
34yu
/*这是条注释*/
int a=10
iNT READ rEAd
10+10<=100
10.4+10.5=20.9
@@
{
}
```

结果：

```
D:\MY\Desktop\complier_program\0.04>a in.txt
line3:(KEY,int)
line3:(ID,a)
line3:(=,=)
line3:(VALUE,10)
line4:(KEY,iNT)
line4:(KEY,READ)
line4:(KEY,rEAd)
line5:(VALUE,10)
line5:(+,+)
line5:(VALUE,10)
line5:(<=,<=)
line5:(VALUE,100)
line6:(VALUE,10.4)
line6:(+,+)
line6:(VALUE,10.5)
line6:(=,=)
line6:(VALUE,20.9)
line8:({,{)
line9:(},})
ERROR:
        第1行出现不可识别字符:34yu
        第7行出现不可识别字符:@
        第7行出现不可识别字符:@
2220214386 杨洋
```

(二)语法分析

1. 测试代码1

```c
struct Complex
{
  float real, image;
};

int main()
{
  struct Complex x;
  x.image = 1.5;
}
```

结果：

```
D:\MY\Desktop\complier_program\11.0>parser in.txt
start analysis
Program(1)
 ExtDefList(1)
  ExtDef(1)
   Specifire(1)
    StructSpecifire(1)
     STRUCT(1)
     OptTag(1)
      ID: Complex
     LC(2)
     DefList(3)
      Def(3)
       Specifire(3)
        TYPE: float
       DecList(3)
        Dec(3)
         VarDec(3)
          ID: real
        COMMA(3)
        DecList(3)
         Dec(3)
          VarDec(3)
           ID: image
       SEMI(3)

     RC(4)
   SEMI(4)
  ExtDefList(6)
   ExtDef(6)
    Specifire(6)
     TYPE: int
    FunDec(6)
     ID: main
     LP(6)
     RP(6)
    Compst(7)
     LC(7)
     DefList(8)
      Def(8)
       Specifire(8)
        StructSpecifire(8)
         STRUCT(8)
         Tag(8)
          ID: Complex
       DecList(8)
        Dec(8)
         VarDec(8)
          ID: x
       SEMI(8)

     StmtList(9)
      Stmt(9)
       Exp(9)
        Exp(9)
         Exp(9)
          ID: x
         DOT(9)
         ID: image
        ASSIGNOP(9)
        Exp(9)
         FLOAT: 1.500000
       SEMI(9)

     RC(10)
```

没有词法、语法错误，输出了完整的语法分析树，并且能够识别出数字的值。

2. 测试代码2

```
struct Complex
```

```
{
  float real, image;
};

int main()
{
  struct Complex x;
  float a[10] = 1.5;
  int i = 100;
  x.image = ~i;

  if (a[1][2] == 0) i =1 else i =0;
}
```

结果：

```
D:\MY\Desktop\complier_program\11.0>parser in.txt
start analysis
Error type A at line 11: Mystirious charachter '~'
Error type B at Line 13, syntax error ,before else

D:\MY\Desktop\complier_program\11.0>
```

能够识别词法错误、语法错误，不输出语法分析树。

(三)语义分析

1. 测试代码1

```
int main()
{
    int i = 0;
    j = i +1;
}
```

结果：

```
Error type 1 at Line 4:undefined variable j
```

】

2. 测试代码2

```
int main()
{
    int i;
    i = 3.7;
}
```

结果：

```
Error type 2 at Line 4:Type mismatched for assignment.
```

3. 测试代码3

```
struct Position
{
  float x;
};
struct Number
{
  float x;
};

int main()
{
}
```

结果：

```
Error type 10 at Line 7:Struct Fields use the same name.
```

4. 测试代码4

```
int sum(int a,int b)
{
}
int main(){
    int i;
    float j;
    sum(i,j);
}
```

结果：

```
D:\MY\Desktop\complier_program\2.02>parser in.txt
Error type 13 at Line 7:Function parameter type error.
```

六、实验总结

1. 本次实验扩充了：
    1. 语法分析中结构体语句、while语句；
    2. 语义分析中，返回值函数，带参函数；
    3. 同时定义多个变量；
    4. 数组、结构体，浮点数类型。
2. 实验心得体会

    通过使用flex和bison工具实现词法分析、语法分析、语义分析等实验，我深入理解了编译原理中的基本概念和原理，掌握了如何将这些概念和原理应用到实际的编程工作中。在实验过程中，我需要不断地调试和修改代码，这对我的编程能力和调试能力都是一次很好的锻炼。同时，实验中也可能会遇到一些难以预料的问题，需要自己思考和解决，这对我的问题解决能力和创新能力也是一次很好的提升。最终成功实现了词法分析、语法分析、语义分析的功能，让我感到非常有成就感和满足感，同时也让我更加自信和有信心去面对更加复杂的编程任务。