**软件学院2024-2025学年度第一学期**

**《软件设计综合实践》实验报告**

|  |  |
| --- | --- |
| **班级：** |  |
| **学号：** |  |
| **姓名：** |  |

**目录**

[1. 实验内容和要求 7](#_Toc183421757)

[2. 斐波拉契数代码 7](#_Toc183421758)

[3. 词法分析程序 7](#_Toc183421759)

[3.1 主要设计和实现思路 7](#_Toc183421760)

[3.2 词法分析程序代码 7](#_Toc183421761)

[3.3 实验演示 7](#_Toc183421762)

[4. 语法分析程序 7](#_Toc183421763)

[4.1 主要设计和实现思路 7](#_Toc183421764)

[4.2 递归下降语法分析程序代码 8](#_Toc183421765)

[4.3 实验演示 8](#_Toc183421766)

[5. 语义分析程序 8](#_Toc183421767)

[5.1 主要设计和实现思路 8](#_Toc183421768)

[5.2 类型检查语义分析程序代码 8](#_Toc183421769)

[5.3 实验演示 8](#_Toc183421770)

[6. 实验总结 8](#_Toc183421771)

## 实验内容和要求

1.用C--语言写一个函数计算第n（从标准输入获得）个斐波那契数，并将计算结果输出到屏幕；

2.用C语言设计和实现C--语言的词法分析器，并对输入的C--代码输出词法分析扫描结果;

3.用C语言设计和实现C--语言的递归下降语法分析器 ，并对输入的C--代码输出语法树。

4.用C语言设计和实现对C--语言进行符号表构造和类型检查的语义分析程序。

## 斐波拉契数代码

int fibonacci(int n)

{

int cnt;

int firstFib;

int secondFib;

int fib;

firstFib = 1;

secondFib = 1;

cnt = 2; /\* n = 1 或 n = 2 时特判 \*/

if (n == 1)

return 1;

else if (n == 2)

return 1;

else

{

while (cnt < n)

{

fib = firstFib + secondFib;

firstFib = secondFib;

secondFib = fib;

cnt = cnt + 1;

}

}

return fib;

}

void main(void)

{

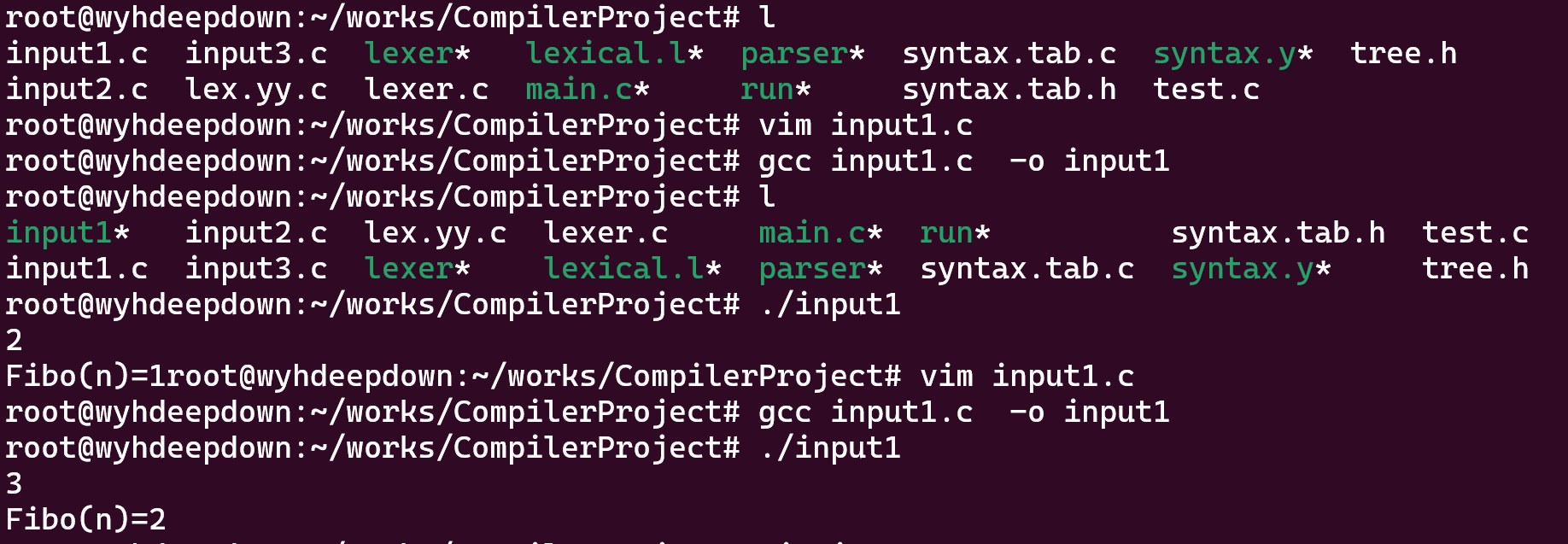
int n;

scanf("%d", &n);

printf("Fibo(n)=%d\n", fibonacci(n));

}

**编译并运行斐波拉契数列求值函数（input1.c）：**

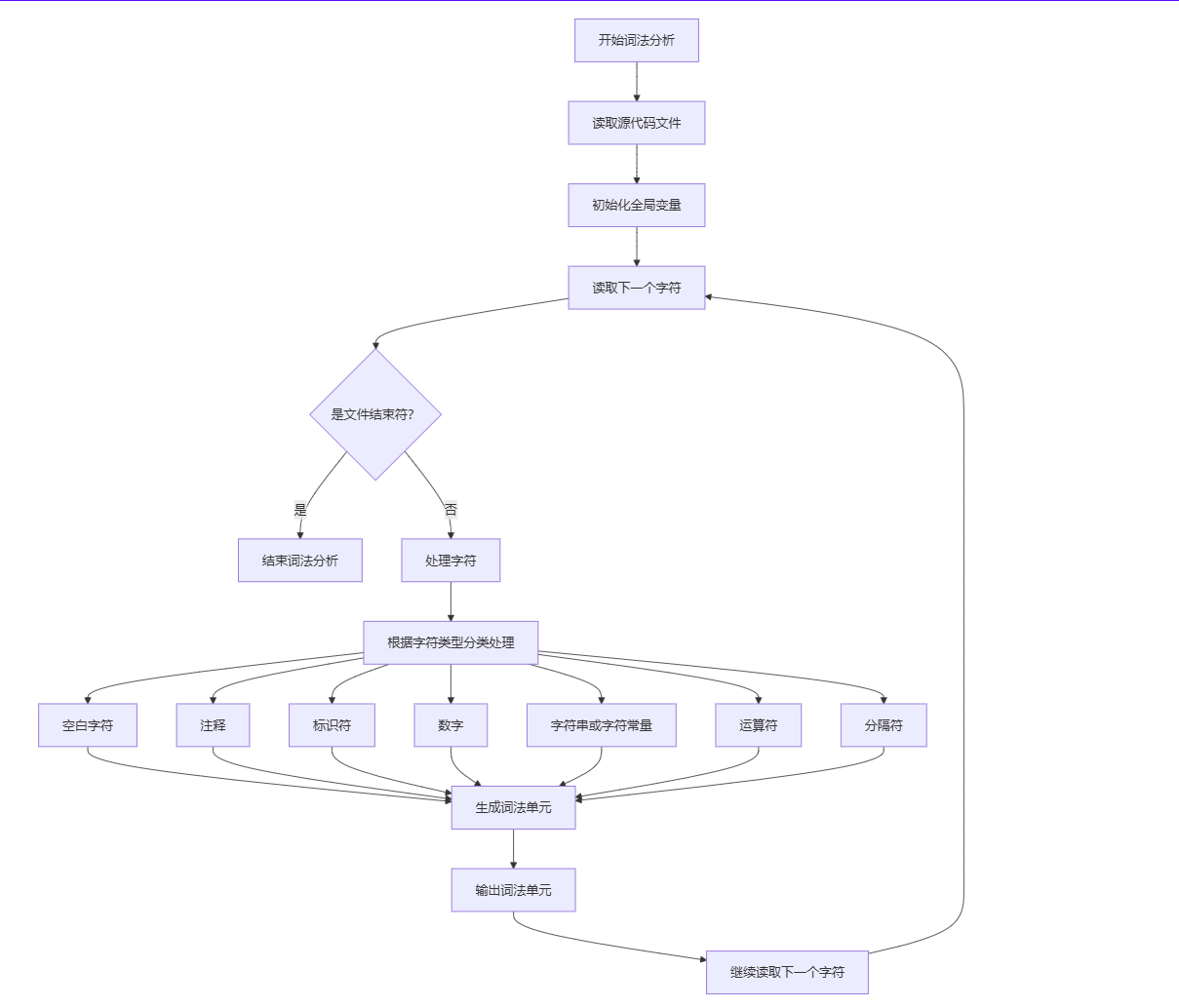


## 词法分析程序

### 主要设计和实现思路

这个词法分析器，用于将输入的源代码分解为一系列的词法单元。其中核心数据结构包括指向源代码的指针src、行号计数器line、当前词法单元类型token、词法单元的值token\_val、数据段指针data，以及符号表symbol\_table。符号表是一个Symbol 结构体数组，用于存储已识别的标识符和关键字，每个条目包含符号名称、哈希值和词法单元类型。符号表指针symbol\_ptr用于跟踪符号表中的条目。符号表通过哈希值快速查找符号，每个符号由名称（name）、哈希值（hash）和词法单元类型（token）组成。符号表的设计采用了一个固定大小的结构体数组，并通过线性探测法处理哈希冲突，确保在有限的存储空间内高效管理符号信息。

**实现思路**:通过枚举类型定义，包括数字（Num）、标识符（Id）、赋值（Assign）、等于（Eq）、自增（Inc）、自减（Dec）等。**主要算法**:词法分析的核心函数是tokenize()，它逐个字符读取源代码并生成词法单元。函数首先处理换行符和注释，然后根据字符类型分别处理标识符、数字、字符串和字符、运算符及分隔符。对于标识符，程序使用简单的哈希算法计算哈希值，并在符号表中查找是否已存在；如果存在，则返回对应的词法单元类型，否则将新标识符添加到符号表中并返回Id类型。对于数字，程序支持十进制、十六进制和八进制的解析，并返回 Num 类型。对于字符串和字符，程序将内容存储到数据段中，并返回 Num 类型或字符串指针。对于运算符和分隔符，程序根据字符和后续字符判断类型，并返回对应的词法单元类型。主函数负责从命令行参数或用户输入获取文件名，读取文件内容到内存中，并初始化数据段。然后调用tokenize()函数逐个生成词法单元，直到源代码结束。最后，程序释放源代码和数据段的内存。在文件读取和内存分配时，程序进行了错误检查，并在出错时输出错误信息。通过简单的哈希算法和线性查找，程序实现了符号表的管理和查找功能，并在识别到词法单元时输出对应的词法单元类型和值。在错误处理方面，程序对文件打开、内存分配等关键操作进行了检查，确保在文件不存在或内存不足时能够优雅地退出并提示错误信息。整体设计兼顾了功能完整性和代码健壮性，为后续的语法分析和语义分析奠定了坚实的基础。通过符号表的高效管理和词法单元的精确生成，该词法分析器能够灵活处理多种编程语言的语法结构，具备较强的可扩展性和实用性。



### 词法分析程序代码

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// 定义全局变量

char \*src; // 源代码指针

int line = 1; // 行号计数器

int token; // 当前词法单元

int token\_val; // 词法单元的值（用于数字或标识符）

char \*data; // 数据段指针（用于存储字符串）

// 符号表结构

typedef struct {

char \*name; // 符号名称

int hash; // 哈希值

int token; // 词法单元类型

} Symbol;

Symbol symbol\_table[100]; // 符号表

int symbol\_ptr = 0; // 符号表指针

// 词法单元类型定义

enum {

Num = 128, Id, Assign, Eq, Inc, Dec, Add, Sub, Mul, Div, Mod, Lt, Gt, Le, Ge, Ne, Shl, Shr, Lor, Land, Or, And, Xor, Brak, Cond

};

// 词法分析函数

void tokenize() {

char \*ch\_ptr;

while ((token = \*src++)) {

if (token == '\n') line++;

else if (token == '#') while (\*src != 0 && \*src != '\n') src++;

else if ((token >= 'a' && token <= 'z') || (token >= 'A' && token <= 'Z') || (token == '\_')) {

ch\_ptr = src - 1;

while ((\*src >= 'a' && \*src <= 'z') || (\*src >= 'A' && \*src <= 'Z') || (\*src >= '0' && \*src <= '9') || (\*src == '\_'))

token = token \* 147 + \*src++;

token = (token << 6) + (src - ch\_ptr);

symbol\_ptr = 0;

while (symbol\_table[symbol\_ptr].token) {

if (token == symbol\_table[symbol\_ptr].hash && !memcmp(symbol\_table[symbol\_ptr].name, ch\_ptr, src - ch\_ptr)) {

token = symbol\_table[symbol\_ptr].token;

return;

}

symbol\_ptr++;

}

symbol\_table[symbol\_ptr].name = ch\_ptr;

symbol\_table[symbol\_ptr].hash = token;

token = symbol\_table[symbol\_ptr].token = Id;

// 输出单词

printf("Token: %.\*s\n", (int)(src - ch\_ptr), ch\_ptr);

return;

}

else if (token >= '0' && token <= '9') {

if ((token\_val = token - '0'))

while (\*src >= '0' && \*src <= '9') token\_val = token\_val \* 10 + \*src++ - '0';

else if (\*src == 'x' || \*src == 'X')

while ((token = \*++src) && ((token >= '0' && token <= '9') || (token >= 'a' && token <= 'f') || (token >= 'A' && token <= 'F')))

token\_val = token\_val \* 16 + (token & 0xF) + (token >= 'A' ? 9 : 0);

else while (\*src >= '0' && \*src <= '7') token\_val = token\_val \* 8 + \*src++ - '0';

token = Num;

printf("Token: %d\n", token\_val); // 输出数字

return;

}

else if (token == '"' || token == '\'') {

ch\_ptr = data;

while (\*src != 0 && \*src != token) {

if ((token\_val = \*src++) == '\\') {

if ((token\_val = \*src++) == 'n') token\_val = '\n';

}

if (token == '"') \*data++ = token\_val;

}

src++;

if (token == '"') token\_val = (int)(long)ch\_ptr; // 使用 long 避免指针到整数的警告

else token = Num;

printf("Token: %.\*s\n", (int)(src - ch\_ptr - 1), ch\_ptr); // 输出字符串或字符

return;

}

else if (token == '/') {

if (\*src == '/') {

while (\*src != 0 && \*src != '\n') src++;

} else {

token = Div;

printf("Token: /\n"); // 输出除号

return;

}

}

else if (token == '=') { if (\*src == '=') { src++; token = Eq; printf("Token: ==\n"); } else { token = Assign; printf("Token: =\n"); } return; }

else if (token == '+') { if (\*src == '+') { src++; token = Inc; printf("Token: ++\n"); } else { token = Add; printf("Token: +\n"); } return; }

else if (token == '-') { if (\*src == '-') { src++; token = Dec; printf("Token: --\n"); } else { token = Sub; printf("Token: -\n"); } return; }

else if (token == '!') { if (\*src == '=') { src++; token = Ne; printf("Token: !=\n"); } else { printf("Token: !\n"); } return; }

else if (token == '<') { if (\*src == '=') { src++; token = Le; printf("Token: <=\n"); } else if (\*src == '<') { src++; token = Shl; printf("Token: <<\n"); } else { token = Lt; printf("Token: <\n"); } return; }

else if (token == '>') { if (\*src == '=') { src++; token = Ge; printf("Token: >=\n"); } else if (\*src == '>') { src++; token = Shr; printf("Token: >>\n"); } else { token = Gt; printf("Token: >\n"); } return; }

else if (token == '|') { if (\*src == '|') { src++; token = Lor; printf("Token: ||\n"); } else { token = Or; printf("Token: |\n"); } return; }

else if (token == '&') { if (\*src == '&') { src++; token = Land; printf("Token: &&\n"); } else { token = And; printf("Token: &\n"); } return; }

else if (token == '^') { token = Xor; printf("Token: ^\n"); return; }

else if (token == '%') { token = Mod; printf("Token: %%\n"); return; }

else if (token == '\*') { token = Mul; printf("Token: \*\n"); return; }

else if (token == '[') { token = Brak; printf("Token: [\n"); return; }

else if (token == '?') { token = Cond; printf("Token: ?\n"); return; }

else if (token == '~' || token == ';' || token == '{' || token == '}' || token == '(' || token == ')' || token == ']' || token == ',' || token == ':') {

printf("Token: %c\n", token); // 输出单个字符

return;

}

}

}

// 主函数

int main(int argc, char \*argv[]) {

char filename[256];

// 如果未通过命令行参数指定文件名，则提示用户输入

if (argc < 2) {

printf("请输入文件名: ");

scanf("%s", filename);

} else {

strcpy(filename, argv[1]);

}

// 读取源代码文件

FILE \*file = fopen(filename, "r");

if (!file) {

perror("无法打开文件");

return 1;

}

fseek(file, 0, SEEK\_END);

long file\_size = ftell(file);

fseek(file, 0, SEEK\_SET);

src = (char \*)malloc(file\_size + 1);

if (!src) {

perror("内存分配失败");

fclose(file);

return 1;

}

size\_t read\_size = fread(src, 1, file\_size, file); // 检查 fread 返回值

if (read\_size != file\_size) {

perror("读取文件失败");

free(src);

fclose(file);

return 1;

}

src[file\_size] = 0; // 添加字符串结束符

fclose(file);

// 初始化数据段

data = (char \*)malloc(1024);

if (!data) {

perror("内存分配失败");

free(src);

return 1;

}

// 词法分析

while (\*src) {

tokenize();

}

// 释放内存

free(src);

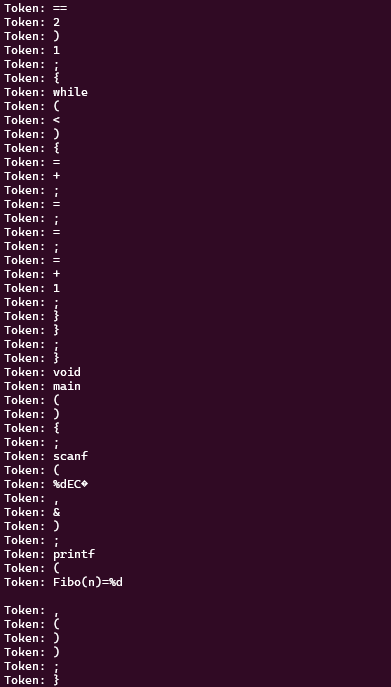
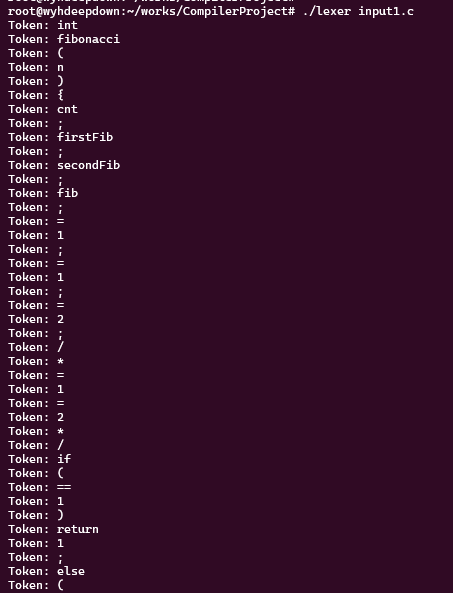
free(data);

return 0;

}

### 实验演示

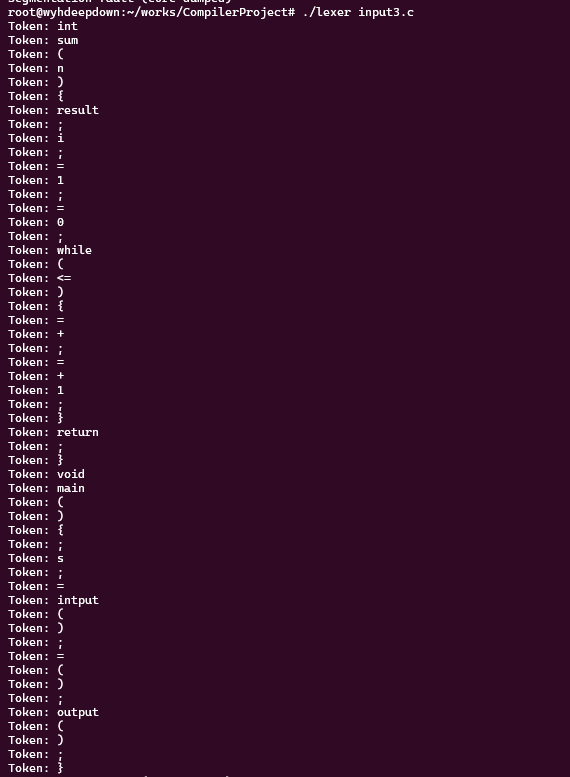
**斐波拉契数:**



**示例1：求三个输入整数中的最大值：**



**示例2：给定N，求1到N之和：**



## 语法分析程序

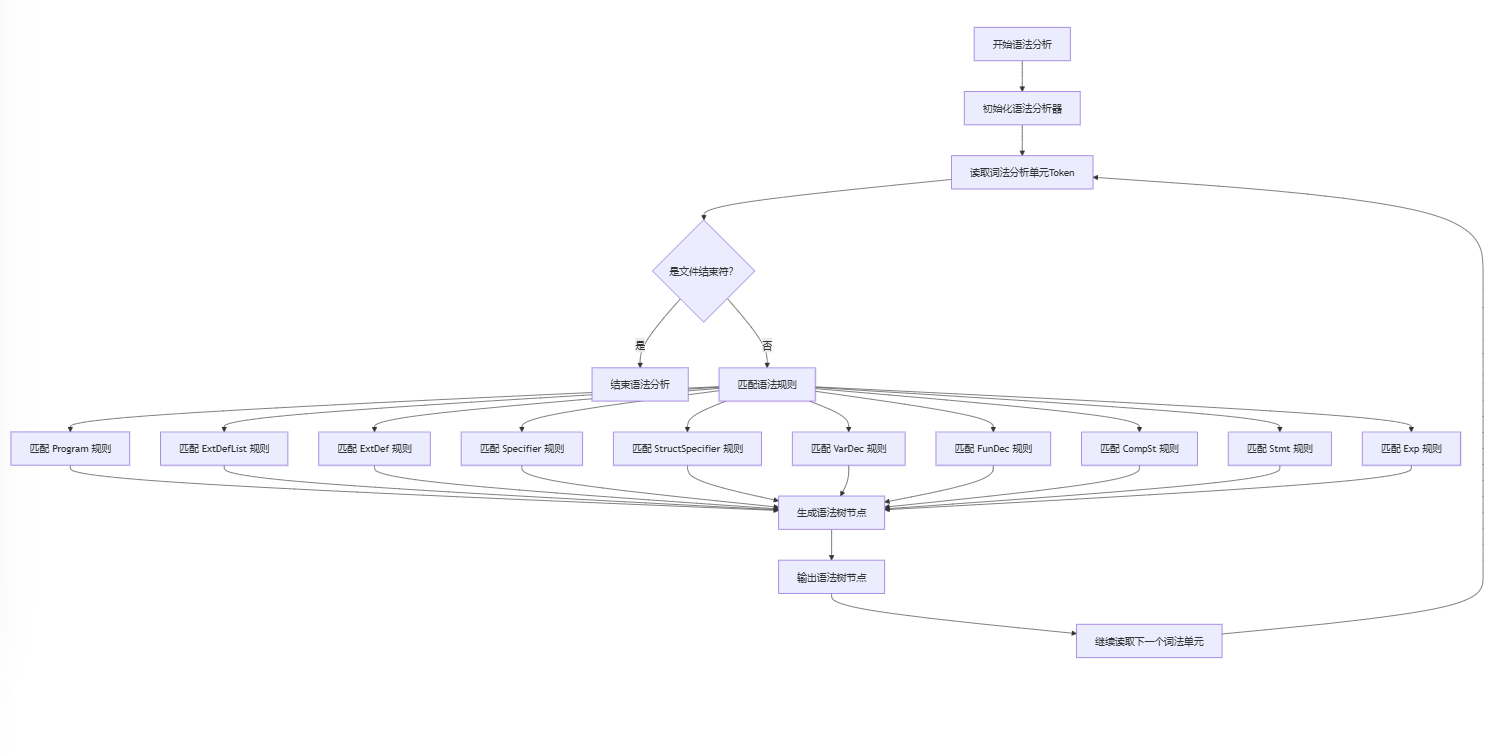
### 主要设计和实现思路

实现了一个cminus编译器前端的设计与实现，核心功能包括词法分析、语法分析以及抽象语法树（AST）的生成与遍历。词法分析器通过 Flex 工具实现，其主要任务是将输入的源代码分解为词法单元（Token），并记录每个词法单元的位置信息（如行号和列号）。在 lexical.l 文件中，词法分析器使用正则表达式定义了各种词法单元的匹配规则，例如标识符（id）、整数（int）、浮点数（float）、关系运算符（relop）以及关键字（如 struct、return、if 等）。对于每个匹配的词法单元，词法分析器会调用 newLeaf 函数生成相应的语法树叶子节点，并将其类型和值存储在节点中。词法分析器还通过 YY\_USER\_ACTION 宏记录每个词法单元的位置信息，确保在后续的语法分析和错误处理中能够准确定位问题。对于无法识别的字符，词法分析器会输出错误信息并记录错误类型，同时通过 err 变量标记错误状态，确保在后续处理中能够及时发现并处理问题。

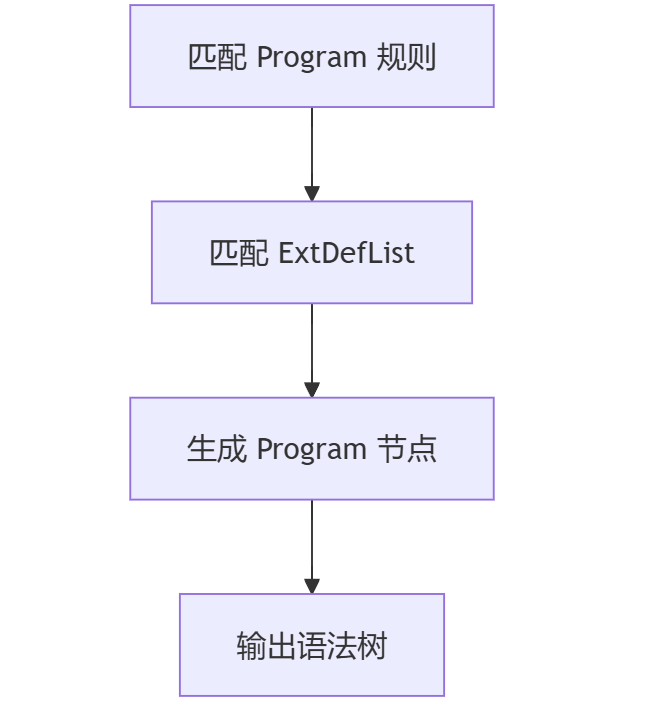
语法分析器通过 Bison 工具实现，其核心功能是根据预定义的语法规则对词法单元序列进行递归下降分析，逐步构建抽象语法树。在 syntax.y 文件中，语法分析器定义了终结符（如 INT、FLOAT、ID 等）和非终结符（如 Program、ExtDef、Exp 等）的类型，并为每个语法规则生成相应的语法树节点。例如，在 Program 规则中，语法分析器会创建一个名为 Program 的节点，并将其子节点设置为 ExtDefList 的语法树节点。语法分析器还通过 yyError 函数处理语法错误，例如在 ExtDecList 规则中，如果遇到错误的词法单元，会输出错误信息并标记错误状态。抽象语法树是编译器前端的核心数据结构，用于表示源代码的语法结构。在 tree.h 文件中，定义了 struct Ast 结构体，每个语法树节点包含节点名称、行号、子节点数量以及子节点数组等信息。通过 newLeaf 函数创建叶子节点表示标识符或常量，通过 newNode 函数创建分支节点表示语法规则中的非终结符，而 newNode0 函数则用于处理语法规则中的可选部分（如空的 ExtDefList）。抽象语法树的遍历通过递归算法实现，printTree 函数能够按层次打印节点信息，便于调试和可视化。例如，对于 Program 节点，printTree 函数会递归打印其子节点 ExtDefList 的信息，并缩进显示层次结构。

错误处理机制贯穿整个编译器前端，词法分析器和语法分析器在遇到错误时会输出详细的错误信息并记录错误类型。例如，在词法分析器中，如果遇到无法识别的字符，会输出 Error type A 并标记错误状态；在语法分析器中，如果遇到语法错误（如缺少分号或括号不匹配），会输出 Error type B 并标记错误状态。主程序负责读取输入文件，初始化词法分析器和语法分析器，生成并打印抽象语法树。在 main.c 文件中，主程序通过 yyrestart 函数初始化词法分析器，并调用 yyparse 函数进行语法分析。如果分析成功，主程序会调用 printTree 函数打印抽象语法树，最终完成编译器前端的整个流程。通过这一系列设计与实现，该代码为后续的语义分析和代码生成奠定了坚实的基础，同时也展示了编译器前端开发中的关键技术与方法。

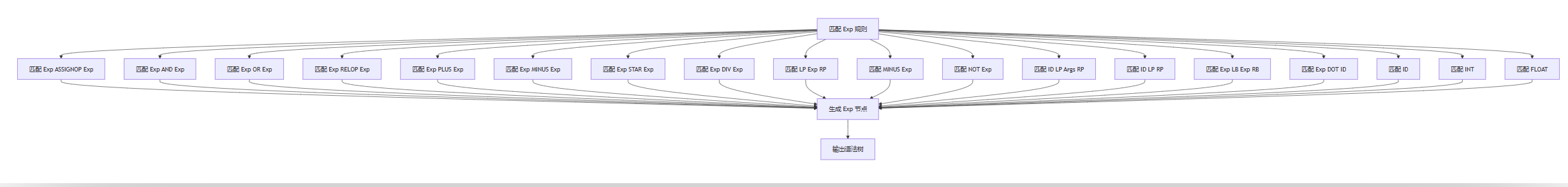
**主要算法：**递归下降算法：算法的实现主要体现在 syntax.y 文件中，通过 Bison 工具将上下文无关文法规则转换为递归下降分析的逻辑。该算法的核心思想是自顶向下地应用文法规则，逐步构建抽象语法树（AST）。具体来说，语法分析器从起始符号 Program 开始，递归地调用各个文法规则，例如 ExtDefList、ExtDef 和 Exp，并为每个规则生成相应的语法树节点。例如，在 Exp : Exp PLUS Exp 规则中，语法分析器会生成一个类型为 Exp 的节点，并将其左右子节点设置为两个 Exp 节点，从而表示加法表达式的语法结构。通过这种递归的方式，语法分析器能够逐步构建完整的抽象语法树，同时处理语法错误并通过 yyError 函数进行错误报告。递归下降分析算法不仅是语法分析的核心，还通过抽象语法树的构建为后续的语义分析和代码生成提供了基础数据结构，是整个编译器前端实现中最为关键的部分。



**语法分析**



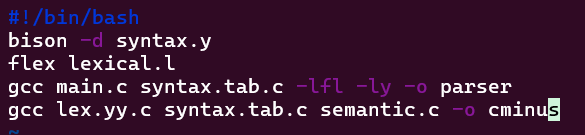
**语法规则(Program规则)匹配**



**语法规则(Exp规则)匹配**

### 递归下降语法分析程序代码

**run文件:**



**lexical.l文件:flex lexical.l生成lex.yy.c**

%{

#include "syntax.tab.h"

extern FILE\* yyin;

int yycolumn = 1;

int err = 0;

struct Ast\* newLeaf(char\* s, int yyline);

#define YY\_USER\_ACTION \

yylloc.first\_line = yylloc.last\_line = yylineno; \

yylloc.first\_column = yycolumn; \

yylloc.last\_column = yycolumn + yyleng - 1; \

yycolumn += yyleng;

%}

%option yylineno

/\*The first section:regular definitions\*/

int [1-9]+[0-9]{0,31}|0

int8 0[0-9]{3}

int16 0x[0-9A-Fa-f]{0,4}

float 0|([1-9]+[0-9]\*)\.[0-9]\*

id [\_a-zA-Z][\_0-9a-zA-Z]\*

relop >|<|>=|<=|==|!=

/\*The second section:roles\*/

%%

\/\\*[^\*]\*\\*+([^/\*][^\*]\*\\*+)\*\/ { }

"//".\* { /\*comment\*/ }

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

return {yylval.a=newLeaf("RETURN",yylineno);return RETURN;}

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

else {yylval.a=newLeaf("ELSE",yylineno);return ELSE;}

whiel {yylval.a=newLeaf("WHILE",yylineno);return WHILE;}

int|float {yylval.a=newLeaf("TYPE",yylineno);return TYPE;}

{int} { yylval.a=newLeaf("INT",yylineno);return INT; }

{float} { yylval.a=newLeaf("FLOAT",yylineno);return FLOAT; }

{id} { yylval.a=newLeaf("ID",yylineno);return ID; }

";" { yylval.a=newLeaf("SEMI",yylineno);return SEMI; }

"," { yylval.a=newLeaf("COMMA",yylineno);return COMMA; }

"=" { yylval.a=newLeaf("ASSIGNOP",yylineno);return ASSIGNOP; }

{relop} {yylval.a=newLeaf("RELOP",yylineno);return RELOP;}

"+" { yylval.a=newLeaf("PLUS",yylineno);return PLUS; }

"-" {yylval.a=newLeaf("MINUS",yylineno);return MINUS;}

"\*" {yylval.a=newLeaf("STAR",yylineno);return STAR;}

"/" {yylval.a=newLeaf("DIV",yylineno);return DIV;}

"&&" {yylval.a=newLeaf("AND",yylineno);return AND;}

"||" {yylval.a=newLeaf("OR",yylineno);return OR;}

"." {yylval.a=newLeaf("DOT",yylineno);return DOT;}

"!" {yylval.a=newLeaf("NOT",yylineno);return NOT;}

"(" {yylval.a=newLeaf("LP",yylineno);return LP;}

")" {yylval.a=newLeaf("RP",yylineno);return RP;}

"[" {yylval.a=newLeaf("LB",yylineno);return LB;}

"]" {yylval.a=newLeaf("RB",yylineno);return RB;}

"{" {yylval.a=newLeaf("LC",yylineno);return LC;}

"}" {yylval.a=newLeaf("RC",yylineno);return RC;}

[ \t\r\f]+ { /\*whitespace\*/ }

"\n" { yycolumn=1; }

. {

fprintf(stderr,"Error type A at Line %d: Mysterious characters \"%s\".\n",yylineno,yytext);

err = 1;

}

%%

//单个文件扫描完毕退出，以免打印语法树相关信息

int yywrap()

{

if(err==1)

exit(0);

return 1;

}

**(主)Syntax.y文件:bison -d syntax.y文件生成syntax.tab.c和syntax.tab.h**

%{

#include "tree.h"

//#define YYERROR\_VERBOSE

void yyError(char\* msg);

void yyerror(const char\* msg);

int ERR = 0;

%}

%union{

struct Ast\* a;

}

/\*tokens 终结符 词法单元 尖括号定义其类型\*/

%token <a> INT FLOAT ID

%token <a> SEMI COMMA ASSIGNOP RELOP

%token <a> PLUS MINUS STAR DIV

%token <a> AND OR

%token <a> DOT NOT TYPE

%token <a> LP RP LB RB LC RC

%token <a> STRUCT RETURN IF ELSE WHILE

/\*语法单元值的类型说明\*/

%type <a> Program ExtDefList ExtDef ExtDecList Specifier StructSpecifier

%type <a> OptTag Tag VarDec FunDec VarList ParamDec CompSt StmtList Stmt DefList Def DecList Dec Exp Args

/\*左结合，右结合，优先级设定\*/

%right ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%right NOT

%left LP COMMA RP LB RB DOT

%nonassoc LOWER\_THAN\_ELSE

%nonassoc ELSE

%%

Program : ExtDefList {struct Ast\* t[1]={$1};$$=newNode("Program",@1.first\_line,1,t);if(ERR==0) printTree($$,0);}

;

ExtDefList : ExtDef ExtDefList {struct Ast\* t[2]={$1,$2};$$=newNode("ExtDefList",@1.first\_line,2,t);}

| {$$=newNode0("ExtDefList",0);}

;

ExtDef : Specifier ExtDecList SEMI {struct Ast\* t[3]={$1,$2,$3};$$=newNode("ExtDef",@1.first\_line,3,t);}

| Specifier SEMI {struct Ast\* t[2]={$1,$2};$$=newNode("ExtDef",@1.first\_line,2,t);}

| Specifier FunDec CompSt {struct Ast\* t[3]={$1,$2,$3};$$=newNode("ExtDef",@1.first\_line,3,t);}

;

ExtDecList : VarDec {struct Ast\* t[1]={$1};$$=newNode("ExtDecList",@1.first\_line,1,t);}

| VarDec COMMA ExtDecList {struct Ast\* t[3]={$1,$2,$3};$$=newNode("ExtDecList",@1.first\_line,3,t);}

| VarDec error ExtDecList {yyError("text");}

;

Specifier : TYPE {struct Ast\* t[1]={$1};$$=newNode("Specifier",@1.first\_line,1,t);}

| StructSpecifier {struct Ast\* t[1]={$1};$$=newNode("Specifier",@1.first\_line,1,t);}

;

StructSpecifier : STRUCT OptTag LC DefList RC {struct Ast\* t[5]={$1,$2,$3,$4,$5};$$=newNode("StructSpecifier",@1.first\_line,5,t);}

| STRUCT Tag {struct Ast\* t[2]={$1,$2};$$=newNode("StructSpecifier",@1.first\_line,2,t);}

;

OptTag : ID {struct Ast\* t[1]={$1};$$=newNode("OptTag",@1.first\_line,1,t);}

| {$$=newNode0("OptTag",0);}

;

Tag : ID {struct Ast\* t[1]={$1};$$=newNode("Tag",@1.first\_line,1,t);}

;

VarDec : ID {struct Ast\* t[1]={$1};$$=newNode("VarDec",@1.first\_line,1,t);}

| VarDec LB INT RB {struct Ast\* t[4]={$1,$2,$3,$4};$$=newNode("VarDec",@1.first\_line,4,t);}

| VarDec LB error RB { yyError("int");}

;

FunDec : ID LP VarList RP {struct Ast\* t[4]={$1,$2,$3,$4};$$=newNode("FunDec",@1.first\_line,4,t);}

| ID LP RP {struct Ast\* t[3]={$1,$2,$3};$$=newNode("FunDec",@1.first\_line,3,t);}

| ID LP error RP { yyError("VarList"); }

;

VarList : ParamDec COMMA VarList {struct Ast\* t[3]={$1,$2,$3};$$=newNode("VarList",@1.first\_line,3,t);}

| ParamDec {struct Ast\* t[1]={$1};$$=newNode("VarList",@1.first\_line,1,t);}

;

ParamDec : Specifier VarDec {struct Ast\* t[2]={$1,$2};$$=newNode("ParamDec",@1.first\_line,2,t);}

;

CompSt : LC DefList StmtList RC {struct Ast\* t[4]={$1,$2,$3,$4};$$=newNode("CompSt",@1.first\_line,4,t);}

;

StmtList : Stmt StmtList {struct Ast\* t[2]={$1,$2};$$=newNode("StmtList",@1.first\_line,2,t);}

| {$$=newNode0("StmtList",0);}

;

Stmt : Exp SEMI {struct Ast\* t[2]={$1,$2};$$=newNode("Stmt",@1.first\_line,2,t);}

//| Exp error SEMI {yyerror("e");}

| CompSt {struct Ast\* t[1]={$1};$$=newNode("Stmt",@1.first\_line,1,t);}

| RETURN Exp SEMI {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Stmt",@1.first\_line,3,t);}

| IF LP Exp RP Stmt %prec LOWER\_THAN\_ELSE {struct Ast\* t[5]={$1,$2,$3,$4,$5};$$=newNode("Stmt",@1.first\_line,5,t);}

| IF LP Exp RP Stmt ELSE Stmt {struct Ast\* t[7]={$1,$2,$3,$4,$5,$6,$7};$$=newNode("Stmt",@1.first\_line,7,t);}

| WHILE LP Exp RP Stmt {struct Ast\* t[5]={$1,$2,$3,$4,$5};$$=newNode("Stmt",@1.first\_line,5,t);}

| Exp error{yyError(" Missing \";\"");}

;

DefList : Def DefList {struct Ast\* t[2]={$1,$2};$$=newNode("DefList",@1.first\_line,2,t);}

| {$$=newNode0("DefList",0);}

;

Def : Specifier DecList SEMI {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Def",@1.first\_line,3,t);}

| Specifier error SEMI {yyError("Syntax error");}

| Specifier DecList error {yyError("missing \";\"");}

;

DecList : Dec {struct Ast\* t[1]={$1};$$=newNode("DecList",@1.first\_line,1,t);}

| Dec COMMA DecList {struct Ast\* t[3]={$1,$2,$3};$$=newNode("DecList",@1.first\_line,3,t);}

;

Dec : VarDec {struct Ast\* t[1]={$1};$$=newNode("Dec",@1.first\_line,1,t);}

| VarDec ASSIGNOP Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Dec",@1.first\_line,3,t);}

;

Exp : Exp ASSIGNOP Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| Exp AND Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| Exp OR Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| Exp RELOP Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| Exp PLUS Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| Exp MINUS Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| Exp STAR Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| Exp DIV Exp {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| LP Exp RP {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| MINUS Exp {struct Ast\* t[2]={$1,$2};$$=newNode("Exp",@1.first\_line,2,t);}

| NOT Exp {struct Ast\* t[2]={$1,$2};$$=newNode("Exp",@1.first\_line,2,t);}

| ID LP Args RP {struct Ast\* t[4]={$1,$2,$3,$4};$$=newNode("Exp",@1.first\_line,4,t);}

| ID LP RP {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| Exp LB Exp RB {struct Ast\* t[4]={$1,$2,$3,$4};$$=newNode("Exp",@1.first\_line,4,t);}

| Exp DOT ID {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Exp",@1.first\_line,3,t);}

| ID {struct Ast\* t[1]={$1};$$=newNode("Exp",@1.first\_line,1,t);}

| INT {struct Ast\* t[1]={$1};$$=newNode("Exp",@1.first\_line,1,t);}

| FLOAT {struct Ast\* t[1]={$1};$$=newNode("Exp",@1.first\_line,1,t);}

| Exp ASSIGNOP error {yyError("a");}

| LP error RP {yyError("Syntax error");}

| ID LP error RP {yyError("Syntax error");}

| Exp LB error RB {yyError("Syntax error");}

| Exp LB Exp error RB{yyError(" Missing \"]\"");}

//| error ';' //{yyError(" Missing\"]\"");}

;

Args : Exp COMMA Args {struct Ast\* t[3]={$1,$2,$3};$$=newNode("Args",@1.first\_line,3,t);}

| Exp {struct Ast\* t[1]={$1};$$=newNode("Args",@1.first\_line,1,t);}

;

%%

void yyerror(const char\* msg)

{

//ERR = 1;

//fprintf(stderr,"Error type B at Line %d:%s\n",yylineno,msg);

}

void yyError(char\* msg)

{

ERR = 1;

fprintf(stderr,"Error type B at Line %d:%s.\n",yylineno,msg);

}

**tree.h:定义Ast及其相关操作**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define MAXCHILD 10

extern int yylineno; /\*[未定义的应用]行号\*/

extern char \*yytext; /\*[未定义的应用]\*/

extern int yylex(void);

/\*grammer Tree node\*/

struct Ast{

int line;

char \*name;

int n; //子节点个数

union{

char\* type;

int i;

int f;

};

struct Ast\* child[MAXCHILD]; //指向子节点的链表

};

struct Ast\* newLeaf(char\* s,int yyline);

struct Ast \*newNode(char \*s,int yyline,int num,struct Ast\* arr[]);

struct Ast \*newNode0(char \*s,int num);

void printTree(struct Ast\* r,int layer);

**mian.c:抽象语法树生成的主要代码**

#include"tree.h"

#include"lex.yy.c"

extern void yyrestart(FILE \*input\_file);

extern int yyparse(void);

extern char \*yytext;

//新建一个叶子节点

struct Ast\* newLeaf(char\* s,int yyline)

{

struct Ast \*l=(struct Ast \*)malloc(sizeof(struct Ast));

l->name=s; //语法单元名字

l->line=yyline;

l->n=0;

if((!strcmp(l->name,"ID"))||(!strcmp(l->name,"TYPE")))

{

char \*t;

t=(char \*)malloc(sizeof(char \*)\*10);

strcpy(t,yytext);

l->type=t;

}

else if(!strcmp(l->name,"INT"))

l->i=atoi(yytext);

else if(!strcmp(l->name,"FLOAT"))

l->f=atof(yytext);

return l;

}

//新建一个语法树分支节点

struct Ast \*newNode(char \*s,int yyline,int num,struct Ast\* arr[])

{

struct Ast \*l=(struct Ast \*)malloc(sizeof(struct Ast));

l->name=s;

l->line=yyline;

l->n=num;

for(int i=0;i<l->n;i++)

l->child[i]=arr[i];

return l;

}

//实现对空的处理

struct Ast \*newNode0(char \*s,int num)

{

struct Ast \*l=(struct Ast \*)malloc(sizeof(struct Ast));

l->name=s;

l->n=0;

l->line=-1;

return l;

}

//遍历语法树

void printTree(struct Ast\* r,int layer)

{

if(r!=NULL && r->line!=-1)

{

for(int i=0;i<layer;i++)

printf(" ");

printf("%s",r->name);//语法单元名字

if((!strcmp(r->name,"ID"))||(!strcmp(r->name,"TYPE")))

printf(": %s",r->type);

else if(!strcmp(r->name,"INT"))

printf(": %d",r->i);

else if(!strcmp(r->name,"FLOAT"))

printf(": %f",r->f);

else if(r->n!=0) //语法单元输出行号

printf(" (%d)",r->line);

printf("\n");

for(int k=0;k<r->n;k++)

printTree(r->child[k],layer+1);

}

}

int main(int argc, char\*\* argv) {

if(argc<=1)

return 1;

FILE\* f=fopen(argv[1],"r");

if(!f)

{

perror(argv[1]);

return 1;

}

yyrestart(f);

yyparse();

return 0;

}

### 实验演示

**斐波拉契数:**

**示例1：求三个输入整数中的最大值：**



**示例2：给定N，求1到N之和：**

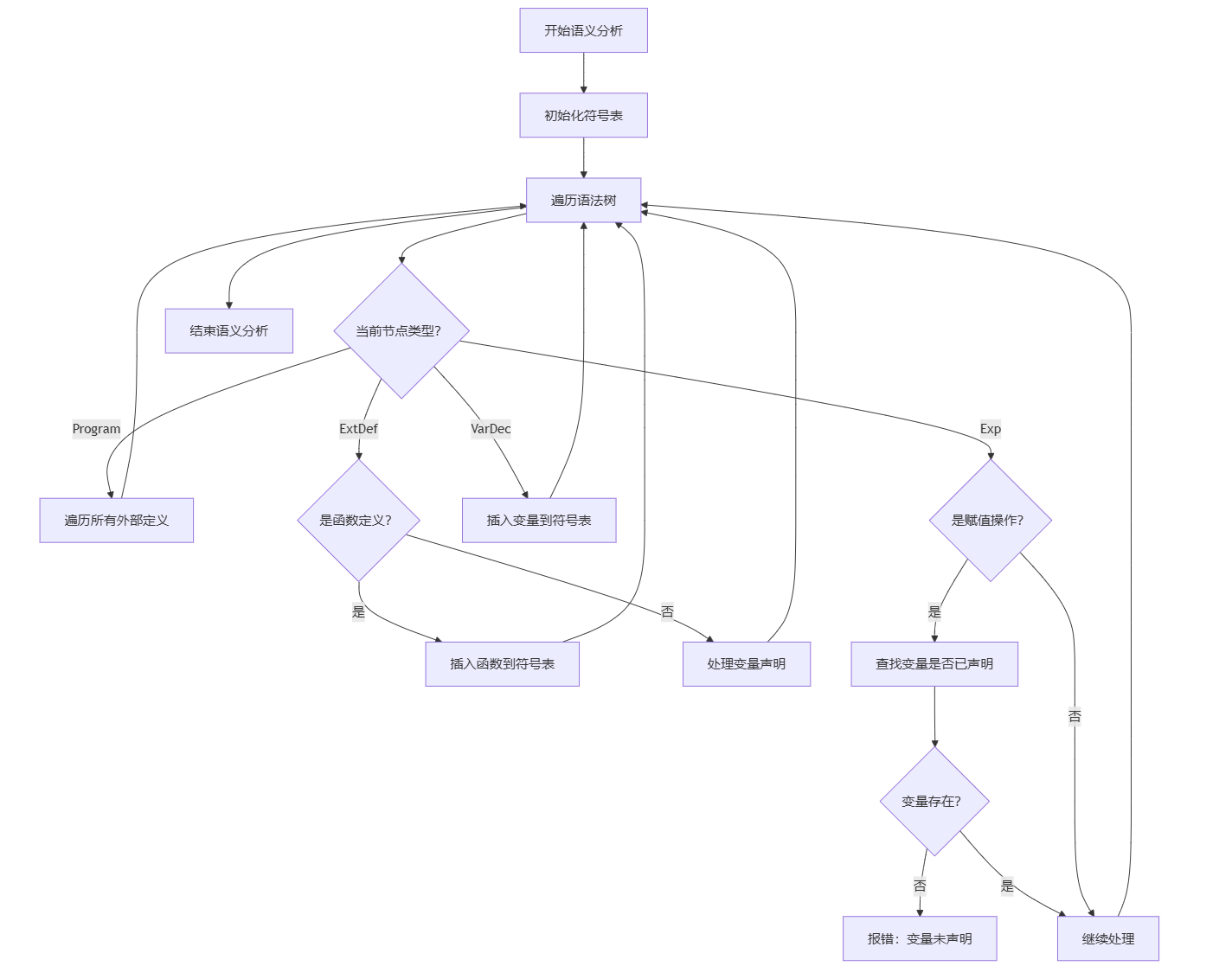


## 语义分析程序

### 主要设计和实现思路

该语义分析程序的设计思路主要围绕构建语法树、符号表以及进行语义分析三个核心部分展开。首先，程序通过语法分析器（如Bison生成的yyparse）解析输入的源代码，生成一棵抽象语法树（AST）。AST的节点结构Ast包含了节点的名称、行号、子节点数量以及一个联合体用于存储不同类型的值（如标识符、整数、浮点数等）。每个节点可以包含多个子节点，形成一个树状结构，用于表示程序的语法结构。在语法树构建完成后，程序创建了一个符号表SymTab，用于存储程序中定义的变量和函数的信息。符号表的每个条目Symbol包含了符号的名称、类型（如变量或函数）、以及一个指向下一个符号的指针。符号表通过链表实现，支持动态插入和查找操作。符号表的插入操作insertSymbol会将新的符号添加到链表的头部，而查找操作lookupSymbol则会遍历链表，查找指定的符号。语义分析的核心函数semanticAnalysis递归遍历语法树，根据不同的语法节点类型执行相应的语义检查。例如，当遇到Program节点时，程序会遍历所有的外部定义；当遇到ExtDef节点时，程序会检查是否为函数定义，并将函数名插入符号表；当遇到VarDec节点时，程序会将变量名插入符号表；当遇到Exp节点时，程序会检查赋值操作中的变量是否已声明。如果发现未声明的变量，程序会输出错误信息。最后，程序通过printSymTab函数打印符号表的内容，展示所有已声明的变量和函数的信息。整个设计思路通过语法树和符号表的结合，实现了对程序语义的检查和分析，确保程序的正确性和一致性。

**主要算法**：该语义分析程序的核心算法和数据结构主要包括**抽象语法树（AST）、符号表**以及**递归遍历算法**。AST 是程序语法结构的树形表示，其节点结构 Ast 包含节点名称、行号、子节点数量以及一个联合体用于存储不同类型的值（如标识符、整数、浮点数等），并通过 child 数组指向子节点，形成多叉树结构。符号表 SymTab 用于存储变量和函数信息，采用链表实现，每个 Symbol 节点包含符号名称、类型和指向下一个符号的指针，支持动态插入和查找操作。语义分析通过递归遍历 AST 实现，根据节点类型执行相应的语义检查，例如在遇到变量声明时将变量插入符号表，在遇到赋值操作时检查变量是否已声明，并在发现未声明变量时输出错误信息。整个设计通过 AST 和符号表的结合，实现了对程序语义的全面分析和检查。这也说明了语法分析使用解析器和上下文无关文法，而语义分析使用符号表和类型检查规则，两者存在区别。



**语法分析**

### 类型检查语义分析程序代码

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// 定义语法树节点结构

#define MAXCHILD 10

struct Ast {

int line;

char \*name;

int n; // 子节点个数

union {

char\* type;

int i;

float f;

};

struct Ast\* child[MAXCHILD]; // 指向子节点的链表

};

// 定义符号表结构

typedef struct Symbol {

char \*name;

char \*type;

int is\_function;

struct Symbol \*next;

} Symbol;

typedef struct SymTab {

Symbol \*head;

} SymTab;

// 全局变量

extern int yyparse(void);

extern int yylineno;

extern char \*yytext;

extern FILE \*yyin;

struct Ast \*root = NULL; // 语法树的根节点

// 函数声明

struct Ast\* newLeaf(char\* s, int yyline);

struct Ast \*newNode(char \*s, int yyline, int num, struct Ast\* arr[]);

struct Ast \*newNode0(char \*s, int num);

void printTree(struct Ast\* r, int layer);

SymTab\* createSymTab();

void insertSymbol(SymTab \*symtab, char \*name, char \*type, int is\_function);

Symbol\* lookupSymbol(SymTab \*symtab, char \*name);

void printSymTab(SymTab \*symtab);

void semanticAnalysis(struct Ast \*root, SymTab \*symtab);

// 创建语法树叶子节点

struct Ast\* newLeaf(char\* s, int yyline) {

struct Ast \*l = (struct Ast \*)malloc(sizeof(struct Ast));

l->name = s; // 语法单元名字

l->line = yyline;

l->n = 0;

if ((!strcmp(l->name, "ID")) || (!strcmp(l->name, "TYPE"))) {

char \*t = (char \*)malloc(sizeof(char) \* 10);

strcpy(t, yytext);

l->type = t;

} else if (!strcmp(l->name, "INT")) {

l->i = atoi(yytext);

} else if (!strcmp(l->name, "FLOAT")) {

l->f = atof(yytext);

}

return l;

}

// 创建语法树分支节点

struct Ast \*newNode(char \*s, int yyline, int num, struct Ast\* arr[]) {

struct Ast \*l = (struct Ast \*)malloc(sizeof(struct Ast));

l->name = s;

l->line = yyline;

l->n = num;

for (int i = 0; i < l->n; i++) {

l->child[i] = arr[i];

}

return l;

}

// 创建空的语法树节点

struct Ast \*newNode0(char \*s, int num) {

struct Ast \*l = (struct Ast \*)malloc(sizeof(struct Ast));

l->name = s;

l->n = 0;

l->line = -1;

return l;

}

// 打印语法树

void printTree(struct Ast\* r, int layer) {

if (r != NULL && r->line != -1) {

for (int i = 0; i < layer; i++) {

printf(" ");

}

printf("%s", r->name); // 语法单元名字

if ((!strcmp(r->name, "ID")) || (!strcmp(r->name, "TYPE"))) {

printf(": %s", r->type);

} else if (!strcmp(r->name, "INT")) {

printf(": %d", r->i);

} else if (!strcmp(r->name, "FLOAT")) {

printf(": %f", r->f);

} else if (r->n != 0) { // 语法单元输出行号

printf(" (%d)", r->line);

}

printf("\n");

for (int k = 0; k < r->n; k++) {

printTree(r->child[k], layer + 1);

}

}

}

// 创建符号表

SymTab\* createSymTab() {

SymTab \*symtab = (SymTab\*)malloc(sizeof(SymTab));

symtab->head = NULL;

return symtab;

}

// 插入符号到符号表

void insertSymbol(SymTab \*symtab, char \*name, char \*type, int is\_function) {

Symbol \*symbol = (Symbol\*)malloc(sizeof(Symbol));

symbol->name = strdup(name);

symbol->type = strdup(type);

symbol->is\_function = is\_function;

symbol->next = symtab->head;

symtab->head = symbol;

}

// 查找符号

Symbol\* lookupSymbol(SymTab \*symtab, char \*name) {

Symbol \*current = symtab->head;

while (current != NULL) {

if (strcmp(current->name, name) == 0) {

return current;

}

current = current->next;

}

return NULL;

}

// 打印符号表

void printSymTab(SymTab \*symtab) {

Symbol \*current = symtab->head;

while (current != NULL) {

printf("Name: %s, Type: %s, Function: %d\n", current->name, current->type, current->is\_function);

current = current->next;

}

}

// 语义分析

void semanticAnalysis(struct Ast \*root, SymTab \*symtab) {

if (root == NULL) return;

if (strcmp(root->name, "Program") == 0) {

// 遍历所有外部定义

for (int i = 0; i < root->n; i++) {

semanticAnalysis(root->child[i], symtab);

}

} else if (strcmp(root->name, "ExtDef") == 0) {

// 处理外部定义

if (root->n == 3 && strcmp(root->child[1]->name, "FunDec") == 0) {

// 函数定义

char \*funcName = root->child[1]->child[0]->type;

insertSymbol(symtab, funcName, "function", 1);

}

} else if (strcmp(root->name, "VarDec") == 0) {

// 变量声明

char \*varName = root->child[0]->type;

insertSymbol(symtab, varName, "int", 0); // 假设所有变量都是int类型

} else if (strcmp(root->name, "Exp") == 0) {

// 表达式

if (root->n == 3 && strcmp(root->child[1]->name, "ASSIGNOP") == 0) {

// 赋值操作

char \*varName = root->child[0]->type;

Symbol \*symbol = lookupSymbol(symtab, varName);

if (symbol == NULL) {

printf("Error: Variable '%s' not declared.\n", varName);

}

}

}

// 递归处理子节点

for (int i = 0; i < root->n; i++) {

semanticAnalysis(root->child[i], symtab);

}

}

// 主函数

int main(int argc, char\*\* argv) {

if (argc <= 1) return 1;

FILE\* f = fopen(argv[1], "r");

if (!f) {

perror(argv[1]);

return 1;

}

yyin = f;

yyparse();

// 创建符号表并进行语义分析

SymTab \*symtab = createSymTab();

semanticAnalysis(root, symtab);

// 打印符号表

printSymTab(symtab);

return 0;

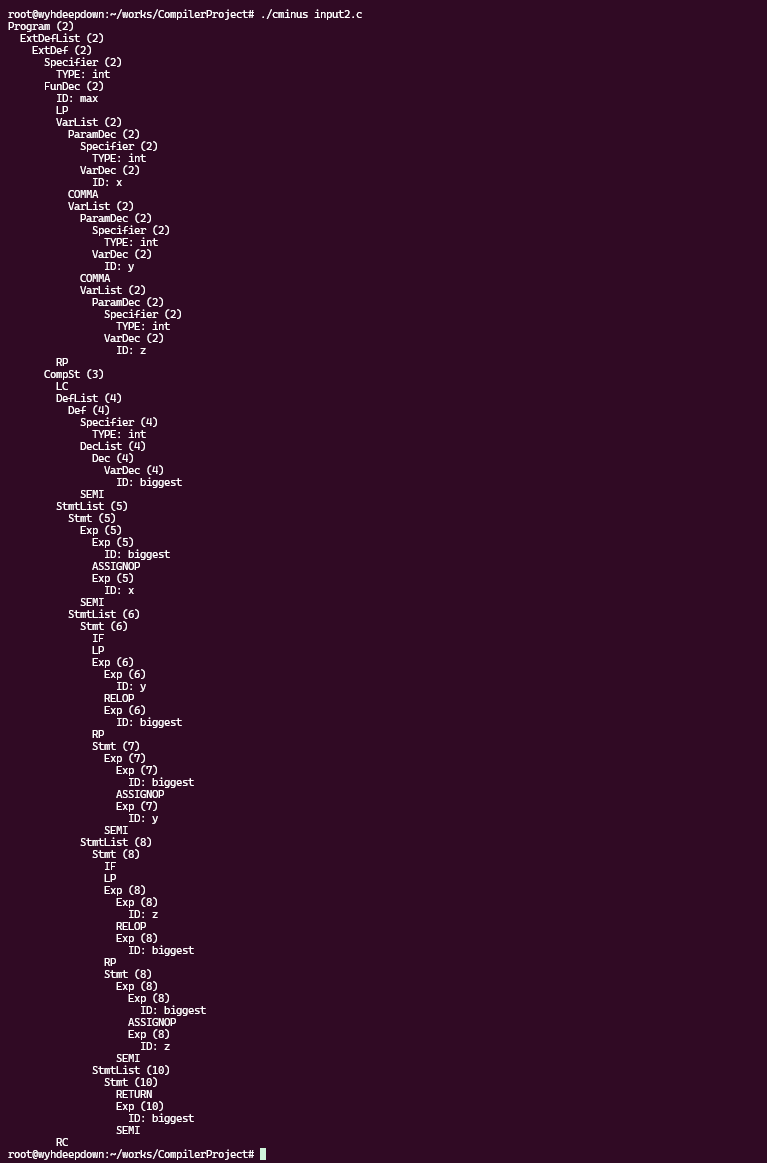
}

### 实验演示

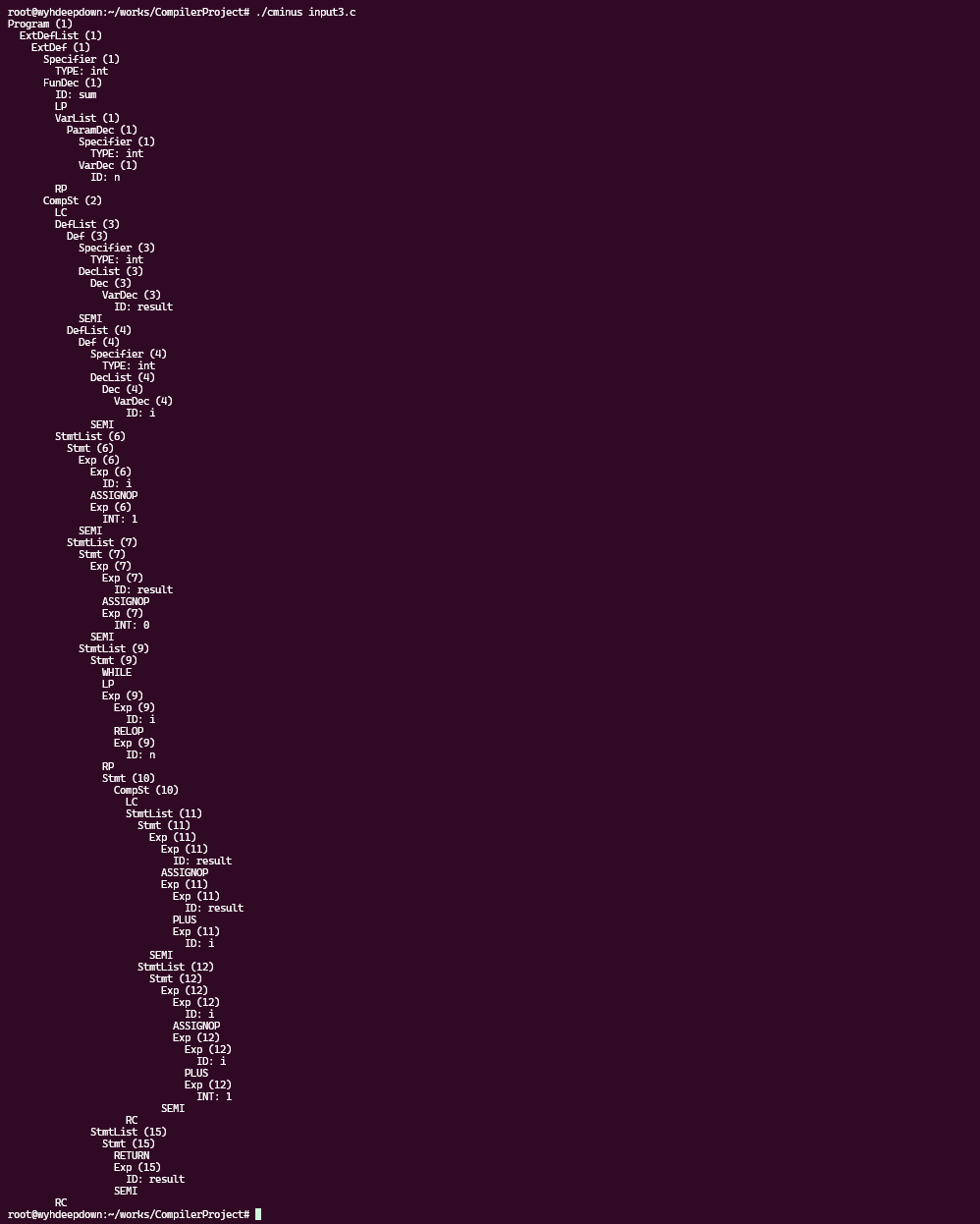
**斐波拉契数:**

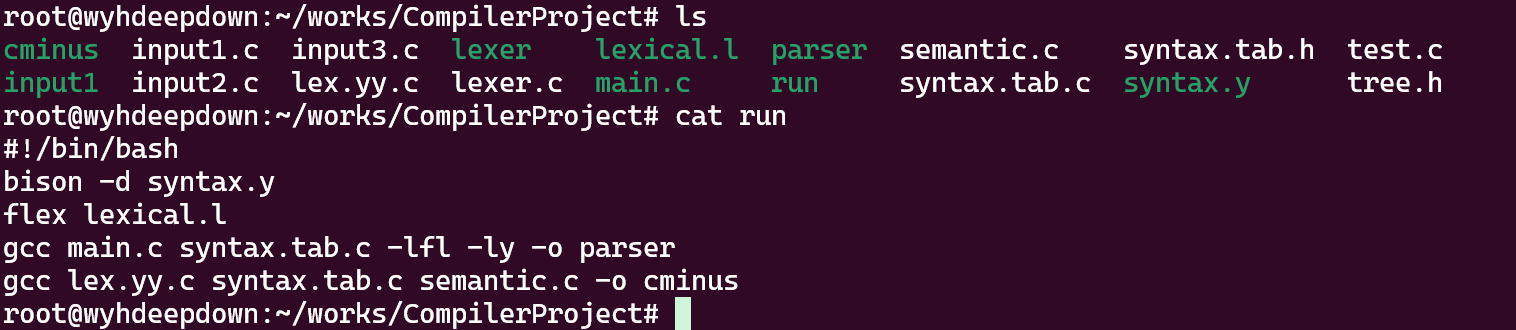
**示例1：求三个输入整数中的最大值：**



**示例2：给定N，求1到N之和：**



## 实验总结



**情况分析和总结：**

本次实验的主要目标是实现一个简单的编译器前端，包括**词法分析**、**语法分析**和**语义分析**三个部分。通过实验，成功构建了一个能够处理简单C语言子集的编译器前端，并生成了抽象语法树（AST）和符号表，完成了对源代码的词法、语法和语义的检查，明白了语法分析是确保代码“写得对”，而语义分析是确保代码“用得对”，两者相辅相成，共同保证程序的正确性。深入理解了词法分析、语法分析和语义分析的核心原理。词法分析器使用Flex工具将源代码转换为词法单元，语法分析器基于Bison工具生成抽象语法树（AST），而语义分析器则通过遍历AST并结合符号表检查程序的语义正确性，成功处理了基本的C语言子集，包括变量声明、函数定义和控制语句，并能够检测词法、语法和语义错误。下一步可以扩展支持更多C语言特性（如指针、结构体），优化错误处理机制，并实现代码生成模块，从而构建一个功能更完善、性能更优的编译器。

**实验收获和改进：**

通过本次实验，深入理解了编译器前端的工作原理，包括词法分析、语法分析和语义分析的具体实现。掌握了Flex和Bison工具的使用，能够基于正则表达式和上下文无关文法构建词法分析器和语法分析器。熟悉了抽象语法树（AST）和符号表的设计与实现，了解了它们在编译器中的重要作用。增强了调试和错误处理的能力，能够快速定位和解决编译过程中的问题。通过实践，将课堂上学到的理论知识应用到实际项目中，加深了对理论的理解。完成一个完整的软件项目，获得了成就感和自信心，增强了对软件开发的兴趣和热情，有助于在未来的学习和工作中保持积极的态度，为未来的职业发展提供了宝贵的经验和能力储备。

下一步需要改进主要包括以下几个方面：首先，因为目前的编译器仅支持C语言的一个子集，功能有限。为了使其能够处理更复杂的程序，需要支持更多的语言特性，如指针、结构体、数组等。这些特性是C语言的核心组成部分，扩展支持它们可以显著提高编译器的实用性和适用性；其次，当前的错误处理机制较为简单，错误信息不够详细，且缺乏错误恢复功能。优化错误处理可以帮助用户更快地定位和修复问题，提升编译器的用户体验。同时，错误恢复机制可以避免因一个错误导致整个编译过程中断，增强编译器的健壮性。因此需要改进优化错误处理机制，增加更详细的错误提示和恢复功能，帮助用户快速定位和修复问题；第三，由于符号表是语义分析的核心数据结构，当前的实现较为基础，仅支持简单的变量和函数信息。通过支持作用域、嵌套函数以及采用哈希表等高效数据结构，可以提高符号表的查找效率，并使其能够处理更复杂的程序结构；第四，实现代码生成模块，将抽象语法树（AST）转换为目标代码（如汇编代码或中间代码），为后续的优化和目标代码生成奠定基础；最后，优化编译器的性能，减少内存占用和提高解析速度，例如通过共享字符串常量、优化AST存储结构等方式，从而构建一个功能更完善、性能更优的编译器。