

**课程实验报告**

**题目： 编译技术实验——编译器**

**课程名称： 编译技术实验**

**专业班级： 软件工程1804**

**学 号： U201816816**

**姓 名： 杨子越**

**指导教师： 胡雯蔷**

**报告日期： 2020年10月22日**

**软件学院**

**目录**

**1 概述** 1

**2 系统描述** 1

2.1 自定义语言概述 1

2.2 单词文法与语言文法 1

2.3 符号表结构定义 3

2.4 中间代码结构定义 3

2.5 目标代码指令集选择 4

**3 系统设计与实现** 4

3.1 词法分析器 4

3.2 语法分析器 4

3.3 符号表管理 4

3.4 语义检查 4

3.5 编译与开发环境 4

3.6 中间代码生成 5

3.7 机器码生成与程序的执行 5

**4 系统测试与评价** 5

4.1 测试用例 5

4.2 报错功能测试 8

4.3 系统的优点 8

4.4 系统的缺点 8

# 1概述

本次实验是构造一个高级语言的子集的编译器，目标代码是LLVM IR 中间语言。

这里实现了一个精简版 C语言编译期，只有基础的简单的语法。实验的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，提高学生系统软件研发技术。

# 2系统描述

### **2.1自定义语言概述**

这里实现了一个简单的C语言的编译器，仅支持C语言的一个子集。

### **2.2单词文法与语言文法**

%type <ptr> program ExtDefList ExtDef Specifier ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args

%token <type\_int> INT

%token <type\_id> ID RELOP TYPE

%token <type\_float> FLOAT

%token DPLUS LP RP LC RC SEMI COMMA

%token PLUS MINUS STAR DIV MOD ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR SWITCH CASE COLON DEFAULT CONTINUE BREAK

%token EXT\_DEF\_LIST EXT\_VAR\_DEF FUNC\_DEF FUNC\_DEC EXT\_DEC\_LIST PARAM\_LIST PARAM\_DEC VAR\_DEF DEC\_LIST DEF\_LIST COMP\_STM STM\_LIST EXP\_STMT IF\_THEN IF\_THEN\_ELSE

%token FUNC\_CALL ARGS FUNCTION PARAM ARG CALL LABEL GOTO JLT JLE JGT JGE EQ NEQ

%left ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%left MOD

%right UMINUS NOT DPLUS

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

%%

program: ExtDefList { print\_ast\_node($1,0); entrypoint($1);};

ExtDefList: {$$=nullptr;}

| ExtDef ExtDefList {$$=make\_node(EXT\_DEF\_LIST,yylineno,{$1,$2});};

ExtDef: Specifier ExtDecList SEMI {$$=make\_node(EXT\_VAR\_DEF,yylineno,{$1,$2});}

|Specifier FuncDec CompSt {$$=make\_node(FUNC\_DEF,yylineno,{$1,$2,$3});}

| error SEMI {$$=nullptr;};

Specifier: TYPE {$$=make\_node(TYPE,yylineno);$$->data = string($1);$$->type=(string($1) == "float")?FLOAT:INT;} ;

ExtDecList: VarDec {$$=$1;}

| VarDec COMMA ExtDecList {$$=make\_node(EXT\_DEC\_LIST,yylineno,{$1,$3});};

VarDec: ID {$$=make\_node(ID,yylineno);$$->data = $1;};

FuncDec: ID LP VarList RP {$$=make\_node(FUNC\_DEC,yylineno,{$3});$$->data = $1;}

|ID LP RP {$$=make\_node(FUNC\_DEC,yylineno);$$->data = $1;$$->ptr[0]=nullptr;};

VarList: ParamDec {$$=make\_node(PARAM\_LIST,yylineno,{$1});}

| ParamDec COMMA VarList {$$=make\_node(PARAM\_LIST,yylineno,{$1,$3});};

ParamDec: Specifier VarDec {$$=make\_node(PARAM\_DEC,yylineno,{$1,$2});};

CompSt: LC DefList StmList RC {$$=make\_node(COMP\_STM,yylineno,{$2,$3});};

StmList: {$$=nullptr; }

| Stmt StmList {$$=make\_node(STM\_LIST,yylineno,{$1,$2});};

Stmt: Exp SEMI {$$=make\_node(EXP\_STMT,yylineno,{$1});}

| CompSt {$$=$1;}

| RETURN Exp SEMI {$$=make\_node(RETURN,yylineno,{$2});}

| IF LP Exp RP Stmt %prec LOWER\_THEN\_ELSE {$$=make\_node(IF\_THEN,yylineno,{$3,$5});}

| IF LP Exp RP Stmt ELSE Stmt {$$=make\_node(IF\_THEN\_ELSE,yylineno,{$3,$5,$7});}

| WHILE LP Exp RP Stmt {$$=make\_node(WHILE,yylineno,{$3,$5});}

| CONTINUE SEMI {$$=make\_node(CONTINUE,yylineno);}

| BREAK SEMI {$$=make\_node(BREAK,yylineno);};

DefList: {$$=nullptr; }

| Def DefList {$$=make\_node(DEF\_LIST,yylineno,{$1,$2});}

| error SEMI {$$=nullptr;};

Def: Specifier DecList SEMI {$$=make\_node(VAR\_DEF,yylineno,{$1,$2});};

DecList: Dec {$$=make\_node(DEC\_LIST,yylineno,{$1});}

| Dec COMMA DecList {$$=make\_node(DEC\_LIST,yylineno,{$1,$3});};

Dec: VarDec {$$=$1;}

| VarDec ASSIGNOP Exp {$$=make\_node(ASSIGNOP,yylineno,{$1,$3});};

Exp: Exp ASSIGNOP Exp {$$=make\_node(ASSIGNOP,yylineno,{$1,$3});}

| Exp AND Exp {$$=make\_node(AND,yylineno,{$1,$3});}

| Exp OR Exp {$$=make\_node(OR,yylineno,{$1,$3});}

| Exp RELOP Exp {$$=make\_node(RELOP,yylineno,{$1,$3});$$->data = $2;}

| Exp PLUS Exp {$$=make\_node(PLUS,yylineno,{$1,$3});}

| Exp MINUS Exp {$$=make\_node(MINUS,yylineno,{$1,$3});}

| Exp STAR Exp {$$=make\_node(STAR,yylineno,{$1,$3});}

| Exp MOD Exp {$$=make\_node(MOD,yylineno,{$1,$3});}

| Exp DIV Exp {$$=make\_node(DIV,yylineno,{$1,$3});}

| LP Exp RP {$$=$2;}

| MINUS Exp %prec UMINUS {$$=make\_node(UMINUS,yylineno,{$2});}

| NOT Exp {$$=make\_node(NOT,yylineno,{$2});}

| DPLUS Exp {$$=make\_node(DPLUS,yylineno,{$2});}

| Exp DPLUS {$$=make\_node(DPLUS,yylineno,{$1});}

| ID LP Args RP {$$=make\_node(FUNC\_CALL,yylineno,{$3});$$->data = $1;}

| ID LP RP {$$=make\_node(FUNC\_CALL,yylineno);$$->data = $1;}

| ID {$$=make\_node(ID,yylineno);$$->data = $1;}

| INT {$$=make\_node(INT,yylineno);$$->data=$1;$$->type=INT;}

| FLOAT {$$=make\_node(FLOAT,yylineno);$$->data=$1;$$->type=FLOAT;};

Args: Exp COMMA Args {$$=make\_node(ARGS,yylineno,{$1,$3});}

| Exp {$$=make\_node(ARGS,yylineno,{$1});};

%%

### **2.3 符号表结构定义**

class Symbol {

public:

string name;

int level;

int type;

int paramnum;

string alias;

char flag;

int idx;

Symbol();

};

vector<Symbol> symbol\_table;

### **2.4 错误类型码定义**

基于C++ 11的std::exception，遇到报错只需要向外层throw std::exception 即可，可参考

<https://en.cppreference.com/w/cpp/error/exception>

### **2.5中间代码结构定义**

class CodeNode {

public:

int kind;

vector<shared\_ptr<CodeNode>> data;

Operation opn1, opn2, result;

shared\_ptr<CodeNode> next, prev;

CodeNode();

};

### **2.6目标代码指令集选择**

本编译器的产物是LLVM IR，具体可参考 <https://llvm.org/>

# 3系统设计与实现

1. **词法分析器**

词法分析器使用flex 2.6.4 GNU/Linux，通过编写lex.l文件，生成对应的C++程序。

1. **语法分析器**

语法分析器使用bison 3.6.4 GNU/Linux,通过编写parser.y文件，生成对应的C++程序。

1. **符号表管理**

符号表的添加、删除、搜索操作由C++ STL的std::vector<T> 来维护，同时维护一个前缀和，当作用域改变的时候，使用resize方法修改符号表的长度。

1. **语义检查**

通过对语法分析生成的AST进行dfs遍历，同时维护符号表，进而检测变量类型是否一致、函数调用参数是否一致、变量是否预先声明过，等等。同时生成CodeNode 节点，方便下一步的中间代码生成。

1. **编译与开发环境**

本代码开发、编译、测试、运行，环境为 Manjaro ArchLinux 5.8.16-2，编译器使用G++ 10.2.0，使用CMake 3.18.3 辅助编译，使用 GDB 9.2进行调试，开发IDE为 Visual Studio Code，LLVM-config 版本 v10.0.1，CLang++ v10.0.1，x86\_64架构 CPU

1. **中间代码生成**

Dfs遍历上一次生成的 CodeNode，根据kind、operation等利用 LLVM IR 提供的API,生成LLVM IR中间代码。

**3.7 机器码的生成与程序的执行**

由LLVM Backend作为后端，由LLVM IR中间码生成可以运行在 Linux x86\_64平台上的机器码，从而可以执行。

# 4系统测试与评价

**4.1 测试用例**

测试思路：编写一段C语言代码，将编译器编译好的可执行文件运行，通过结果，结合预期与GCC编译期进行相同的操作产生的文件的结果进行对比，如果无异常，则说明通过此Case.

举例：

int add(int a, int b) {

return a + b;

}

int main() {

int a = 12;

float b = 1.2 + 3.6;

if (b > 1.2) {

print\_int(114514);

}

while (a > 0) {

if (add(a, 1) % 2 == 1) {

print\_int(a);

break;

}

a = a - 1;

}

return 0;

}

这段代码生成的LLVM IR如下：

File path = ./test.txt

; ModuleID = 'code'

source\_filename = "code"

@0 = private unnamed\_addr constant [4 x i8] c"%d\0A\00", align 1

declare i32 @printf(i8\*, ...)

define i32 @print\_int(i32 %0) {

%2 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @0, i32 0, i32 0), i32 %0)

ret i32 %2

}

define i32 @add(i32 %v2, i32 %v3) {

entry:

%t1 = add i32 %v2, %v3

ret i32 %t1

}

define i32 @main() {

entry:

%t2 = alloca i32

store i32 12, i32\* %t2

%v5 = alloca i32

%0 = load i32, i32\* %t2

store i32 %0, i32\* %v5

%t3 = alloca float

store float 0x3FF3333340000000, float\* %t3

%t4 = alloca float

store float 0x400CCCCCC0000000, float\* %t4

%1 = load float, float\* %t3

%2 = load float, float\* %t4

%t5 = fadd float %1, %2

%v6 = alloca float

store float %t5, float\* %v6

%t6 = alloca float

store float 0x3FF3333340000000, float\* %t6

%3 = load float, float\* %v6

%4 = load float, float\* %t6

%cmpres = fcmp ogt float %3, %4

br i1 %cmpres, label %l2, label %l1

l2: ; preds = %entry

%t7 = alloca i32

store i32 114514, i32\* %t7

%5 = load i32, i32\* %t7

%t8 = call i32 @print\_int(i32 %5)

br label %l1

l1: ; preds = %entry, %l2

br label %l5

l5: ; preds = %l6, %l1

%t9 = alloca i32

store i32 0, i32\* %t9

%6 = load i32, i32\* %v5

%7 = load i32, i32\* %t9

%cmpres1 = icmp sgt i32 %6, %7

br i1 %cmpres1, label %l4, label %l3

l4: ; preds = %l5

%t10 = alloca i32

store i32 1, i32\* %t10

%8 = load i32, i32\* %v5

%9 = load i32, i32\* %t10

%t11 = call i32 @add(i32 %8, i32 %9)

%t12 = alloca i32

store i32 2, i32\* %t12

%10 = load i32, i32\* %t12

%t13 = srem i32 %t11, %10

%t14 = alloca i32

store i32 1, i32\* %t14

%11 = load i32, i32\* %t14

%cmpres2 = icmp eq i32 %t13, %11

br i1 %cmpres2, label %l7, label %l6

l7: ; preds = %l4

%12 = load i32, i32\* %v5

%t15 = call i32 @print\_int(i32 %12)

br label %l3

l6: ; preds = %l4

%t16 = alloca i32

store i32 1, i32\* %t16

%13 = load i32, i32\* %v5

%14 = load i32, i32\* %t16

%t17 = sub i32 %13, %14

store i32 %t17, i32\* %v5

br label %l5

l3: ; preds = %l5, %l7

%t18 = alloca i32

store i32 0, i32\* %t18

%15 = load i32, i32\* %t18

ret i32 %15

}

这段代码的 运行结果如下：

114514

12

与 GCC 10编译C语言的运行产生的结果一致。

**4.2 报错功能测试**

测试思路：编写一段有错误的代码，看一下是否会产生报错。通过与GCC、预期进行对比，说明程序的行为是正确的。

举例子：下一段代码是有问题的，引用了未声明的符号，执行代码，程序正常抛出错误。

int main() {

print\_int(b);

return 0;

}

产生结果：

File path = ./test.txt

terminate called after throwing an instance of 'std::runtime\_error'

what(): Line 2, Message: b Variable not declared.

fish: “./compiler ./test.txt” terminated by signal SIGABRT (Abort)

**4.3 系统的优点**

可以直接生成LLVM IR源代码，通过 LLVM Backend 可以编译到 ARM/X86/RISC-V/WASM等平台运行。基于C++ 20编写，源代码质量较高。基于C++智能指针来管理内存，减少内存泄漏的情况。

**4.4 系统的缺点**

符号表的管理算法仍然有待优化，代码较多的时候会有性能瓶颈。仅仅支持C语言的一个子集，并不支持很多C11、C18语言特性。

**参考文献**

[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016

[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**