

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 编译原理 |
| 姓 名： | 范源颢（3180103574）  宋天泽（3180105221）  钱隆 （3180106068） |
| 学 院： | 计算机科学与技术学院 |
| 系： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3180103574、3180105221 3180106068 |
| 指导教师： | 冯雁 陈纯 |

目录

[序言 4](#_Toc73908653)

[0.1 功能描述 4](#_Toc73908654)

[0.2 文件说明 4](#_Toc73908655)

[0.3运行环境 4](#_Toc73908656)

[0.4 分工 5](#_Toc73908657)

[第一章 词法分析 5](#_Toc73908658)

[1.1 约定 5](#_Toc73908659)

[关键字 5](#_Toc73908660)

[运算符 6](#_Toc73908661)

[标识符和字面量 6](#_Toc73908662)

[注释 6](#_Toc73908663)

[1.2 正规表达式 6](#_Toc73908664)

[1.3 lex实现原理和方法 7](#_Toc73908665)

[第二章 语法分析 8](#_Toc73908666)

[2.1 约定 8](#_Toc73908667)

[2.2 上下文无关文法 9](#_Toc73908668)

[2.2.1面向过程 10](#_Toc73908669)

[2.2.2变量定义 11](#_Toc73908670)

[2.2.3 函数定义 11](#_Toc73908671)

[2.2.3 复合语句结构 12](#_Toc73908672)

[2.2.4 语句 12](#_Toc73908673)

[2.2.4 表达式 13](#_Toc73908674)

[2.2.5 函数调用 14](#_Toc73908675)

[2.3 语法树结点定义 14](#_Toc73908676)

[**2.4 Yacc实现原理和方法** 15](#_Toc73908677)

[**2.5 语法树的遍历和打印** 15](#_Toc73908678)

[**2.6 测试** 16](#_Toc73908679)

[**2.6.1简易测试和冲突处理** 16](#_Toc73908680)

[**2.6.2一个粗暴的手段：更换Yacc的匹配算法** 17](#_Toc73908681)

[**2.6.3 打印语法树的效果** 17](#_Toc73908682)

[第三章 符号表设计 21](#_Toc73908683)

[数据结构设计 21](#_Toc73908684)

[符号表操作 21](#_Toc73908685)

[作用域 21](#_Toc73908686)

[第四章 优化考虑 21](#_Toc73908687)

[4.1 词法设计优化 21](#_Toc73908688)

[4.2 语法设计优化 22](#_Toc73908689)

[4.3 llvm优化 24](#_Toc73908690)

[第五章 代码生成 25](#_Toc73908691)

[5.1 llvm模块和函数 25](#_Toc73908692)

[5.1.1 Module 25](#_Toc73908693)

[5.1.2 Function 25](#_Toc73908694)

[5.1.3 BasicBlock 25](#_Toc73908695)

[5.1.4 IRBuilder 25](#_Toc73908696)

[5.1.5 Value 26](#_Toc73908697)

[5.2 环境维护 26](#_Toc73908698)

[5.2.1 变量信息的维护 26](#_Toc73908699)

[5.2.2函数信息的维护 26](#_Toc73908700)

[5.3 llvm 类型生成 26](#_Toc73908701)

[5.4 llvm中间代码生成 27](#_Toc73908702)

[5.4.1表达式生成 27](#_Toc73908703)

[5.4.2语句生成 27](#_Toc73908704)

[5.4.3 函数生成 28](#_Toc73908705)

[第六章 测试案例 29](#_Toc73908706)

[**6.1 构建工程** 29](#_Toc73908707)

[**6.2 各语句测试** 30](#_Toc73908708)

[**6.3 通过gcc与其他目标代码进行链接的测试** 34](#_Toc73908709)

# 序言

## 功能描述

本报告是编译原理期末大作业小组实验的实验报告，此次编译原理期末大作业的目标是实现一个编译器，可使用的工具包括lex、Yacc、llvm等等，具体的实现形式可以是一个类C、类Pascal或其他程序语言的词法、语法分析器，需要生成语法树和中间代码，语法树需要可视化展示，但具体展示形式不限，中间代码的形式并无规定，从中间代码到目标机器的目标代码的转化为选做模块。

这次编译原理实验，我们完成了一个使用C++语言，利用lex、Yacc、llvm工具的类C（mytinyC）语言的编译器，这种类C语言在文法上基本参照C语言文法，在功能上去除了指针、结构体、联合体等C语言的特性，此外对于函数和语句的定义有比起C语言更加严格的限制，对于这种类C语言，我们设计的编译器可以有效的实现词法分析、语法分析、语义分析、代码生成以及代码优化，最终可以展示语法树、llvm中间代码和目标机器的目标代码以及可执行文件。

## 文件说明

ASTNodes.cpp 实验中编写，用于实现构建和展示语法树功能的C++语言代码文件；

ASTNodes.hpp 实验中编写，用于实现构建和展示语法树功能的C++语言头文件；

compile.sh 实验中编写，用于在Ubuntu 16.04+系统中运行完整运行编译器的批处理脚本文件；

lexer.l  实验中编写，用于对类C语言进行词法分析的lex文件

parser.y 实验中编写，用于对类C语言进行语法分析的yacc文件

lex.yy.c flex根据lexer.l生成的，可用于实现词法分析的C源文件

parser.tab.c bison根据parser.y生成的，可用于实现语法分析的C源文件

parser.tab.h bison根据parser.y生成的，可用于实现语法分析的C头文件

our\_syntax.md 实验中编写，用于提示我们设计的类C语言语法规则的Markdown文件，不参与编译器工程的构建。

## 0.3运行环境

使用cmake构建，依赖包括：

* Linux 16.04+
* gcc5.4.0+， g++ 5.4.0+
* flex 2.6.0+
* bison 2.6.0+, (推荐3.7.6+)
* llvm 3.8.0+
* clang 3.8.0+

(其中‘+’表示当前版本及以上)

## 0.4 分工

范源颢：词法分析 语法分析

宋天泽：语法分析（语法树构建） 语义分析

钱隆：中间代码和目标代生成

# 词法分析

## 1.1 约定

我们实现的类C语言mytinyC对于C语言的语法做了一些简化，因此我们有必要约定简化的内容，如下所述：

### 关键字

mytinyC关键字如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| int | float | char | double | bool |
| void | if | else | for | while |
| return | extern |  |  |  |

其中bool用于声明mytinyC中的布尔类型的变量，bool类型仅包含true和false两种，分别可以被声明为true/True /TRUE和false/False/FALSE，运算规则和C++一致。其余的关键字和标准C语言中的关键字意义相同。

于是，我们的mytinyC支持的基本类型包括：整形int、浮点数float、字符char、布尔类型bool、双精度浮点数double。在函数的返回类型方面，我们支持虚返回void。我们的mytinyC语法支持的控制语句包括if-else分支语句， while/for循环语句，函数返回语句return。

注意比起标准C语言，我们在关键字便不再支持结构体（struct）、联合体（union）； unsigned、signed等复杂类型不再支持，我们不支持使用typedef自定义类型，也不支持使用const、static约束变量和函数的作用域。在流程控制方面，我们没有实现switch case 语法，也不支持使用break关键字跳出循环。在底层构建方面，我们也不支持使用sizeof语法查看数据类型的长度。

我们保留了extern关键字，这是为了在使用llvm的过程中，利用extern可以直接llvm中实现的函数，例如，假如在mytinyC文件中按照如下语句声明了输出函数printf：

extern int printf(char[] str)

我们就可以在测试在mytinyC文件中调用printf进行输出。

### 运算符

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| = | += | -= | \*= | /= |
| %= | |= | &= | ^= | >>= |
| <<= | || | && | | | & |
| ^(位抑或) | > | < | >= | <= |
| == | != | << | >> | + |
| - | \* | / | % | ! |
| ~（位取反） | ++（仅后缀） | --（仅后缀） | ( | ) |
| { | } | [ | ] | , |
| ; | ‘’ |  |  |  |

基本上涵盖了所有的C语言操作符。

mytinyC约定++和—只有作为后缀表达式出现，同时，我们没有指针运算，因此我们将\*仅理解为乘法，&理解为位与。

### 标识符和字面量

我们对于标识符的约定和C语言基本相同，用户自定义标识符需满足：字母（大小写均可）或者下划线开头，之后包含若干字母、数字和下划线的组合，推荐总长度在20字符以下。mytinyC不得将true/True/TRUE/false/False/False作为标识符。

我们对于字面量的约定和C语言也基本相同，开头或有字符-（负号），之后仅有数字组成的字符串我们认为是整形数字；开头或有负号，之后含有若干数字，在此之后包含小数点.，小数点之后若有字符则全为数字的字符串我们认为是浮点数（注意若小数点之后没有字符，也认为是浮点数，比如4.0和4.均为浮点数，而4为整形数）；‘’中包含的单个字符是C语言的char类型；三种不同大小写风格的true和三种不同大小写风格的false是布尔类型的字面量。

### 注释

我们实现了C语言风格的注释，包括//开头的单行注释，以及/\*和\*/之间的多行注释。

## 1.2 正规表达式

对于C语言关键字、操作符以及字面量false、true，我们都可以使用穷举的方式列出他们的正规表达式，例如，true的正规表达式可以写成：

true|True|TRUE

标识符和整形数、浮点数的正规表达式亦不难写出，我们给出整形、浮点型和标识符的正规表达式，如下：

int  [\-]?[0-9]+

float [\-]?[0-9]+\.[0-9]\*

identifier [a-zA-Z\_][a-zA-Z0-9\_]\*

注意，我们没有使用正规表达式的思路进行注释的词法分析，因为我们知道多行注释的正规表达式将比较复杂，为此，我们将使用下文中提到的lex的另一特性实现C语言的注释。

## 1.3 lex实现原理和方法

lex主要是通过有限状态机的思路实现词法分析的，我们不在赘述有限状态机的知识，而将注意力集中在如何通过书写lexer.l实现代码的分析。

在.l文件的第一部分（第一个%%之前）我们首先通过正规表达式定义一些语词：

 /\*空白符\*/

macro  #.\*

whitespace [ \t\r\n]+

linecomment \/\/.\*\n

 /\*字面量\*/

int  [\-]?[0-9]+

float [\-]?[0-9]+\.[0-9]\*

identifier [a-zA-Z\_][a-zA-Z0-9\_]\*

char  \'.\'

true  true|True|TRUE

false false|False|FALSE

之后，我们在第二部分定义lex在遇到这些语词的过程中需要执行的操作，若分析的结论是关键字和运算符，直接返回token即可：

"if"        return IF;

"else"      return ELSE;

"return"    return RETURN;

"for"       return FOR;

"while"     return WHILE;

"||"    return LOR;

"&&"    return LAND;

若是类型名称，我们需要保存内容，并且通过token表示其类型，这样将有助于日后的类型检查。

"int"  {

    strcpy(yylval.type\_id, yytext);

    return INT\_TYPE;

}

若是字面量，则我们需要保存字面量的值，并且通过token表示其类型。

{int} {

    yylval.type\_int = atoi(yytext);

    return INT;

}

{float} {

    yylval.type\_float = atof(yytext);

    return FLOAT;

}

{true} {

    yylval.type\_bool = 1;

    return TRUE;

}

{identifier} {

    strcpy(yylval.type\_id, yytext);

    return ID;

}

最后我们讲述注释的处理，如上所述，一旦匹配到//，则此行的内容均视为注释。（其后若出现/\*，标准C也不认为是多行注释的开始，下一行除非再有注释符号，否则即使上一行有未闭合的/\*，下一行的语句不会被编译器作为注释跳过）因此，我们可以直接用正则表达式匹配单行注释。

多行注释我们直接利用lex作为有限自动机的机制进行实现，lex在.l文件的第一部分支持直接定义状态，我们可以使用语句：

%x C\_COMMENT

定义一个名为C\_COMMENT的状态，

在.l文件的第二部分，我们定义一旦遇到字符/\*，使用lex定义的宏BEGIN进入C\_COMMENT状态：

"/\*" BEGIN(C\_COMMENT);

处于C\_COMMENT状态中时，无论遇到何种字符均进行空操作，

<C\_COMMENT>.  {}

注意lex使用尖括号标识遇到字符串时有限状态机的状态，如无标识，则推定此时是有限状态机处于lex内置的初始状态INITIAL中。

最后，如果处于C\_COMMENT状态中遇到\*/，则有限状态机应当返回初始状态INITIAL：

<C\_COMMENT>"\*/" BEGIN(INITIAL);

注意书写时，<C\_COMMENT>"\*/"语句应该先于<C\_COMMENT>.，因为lex优先匹配文件中先行的规则，如若带有.的规则先行，则之后的规则便失去意义，因为之后规则中的字符都会被.优先匹配掉。

# 语法分析

## 2.1 约定

我们在设计mytinyC的语法的时候亦有一些语法上的限制：

1. 我们在大括号中语句需要严格满足所有局部变量的定义先于其他语句；我们不支持for语句的小括号内第一个分句直接定义域内变量

2. 我们支持基本变量和一维数组在定义时初始化，但是不支持多维数组的初始化。

3. 同类型变量的声明支持使用逗号分隔，但所谓的“同类型”仅包括基本类型，不包括等复杂类型数组，例如： int a,b,c;视为合法，int a, b[10];视为不合法。

4. 和标准C一样，多维数组作为参数调用的时候，除标识符最右侧的方括号内可以为空之外，其余括号均需指明数值，例如一下的函数k,i,j中，仅有j是正确的定义。

void k(int a[5][]);

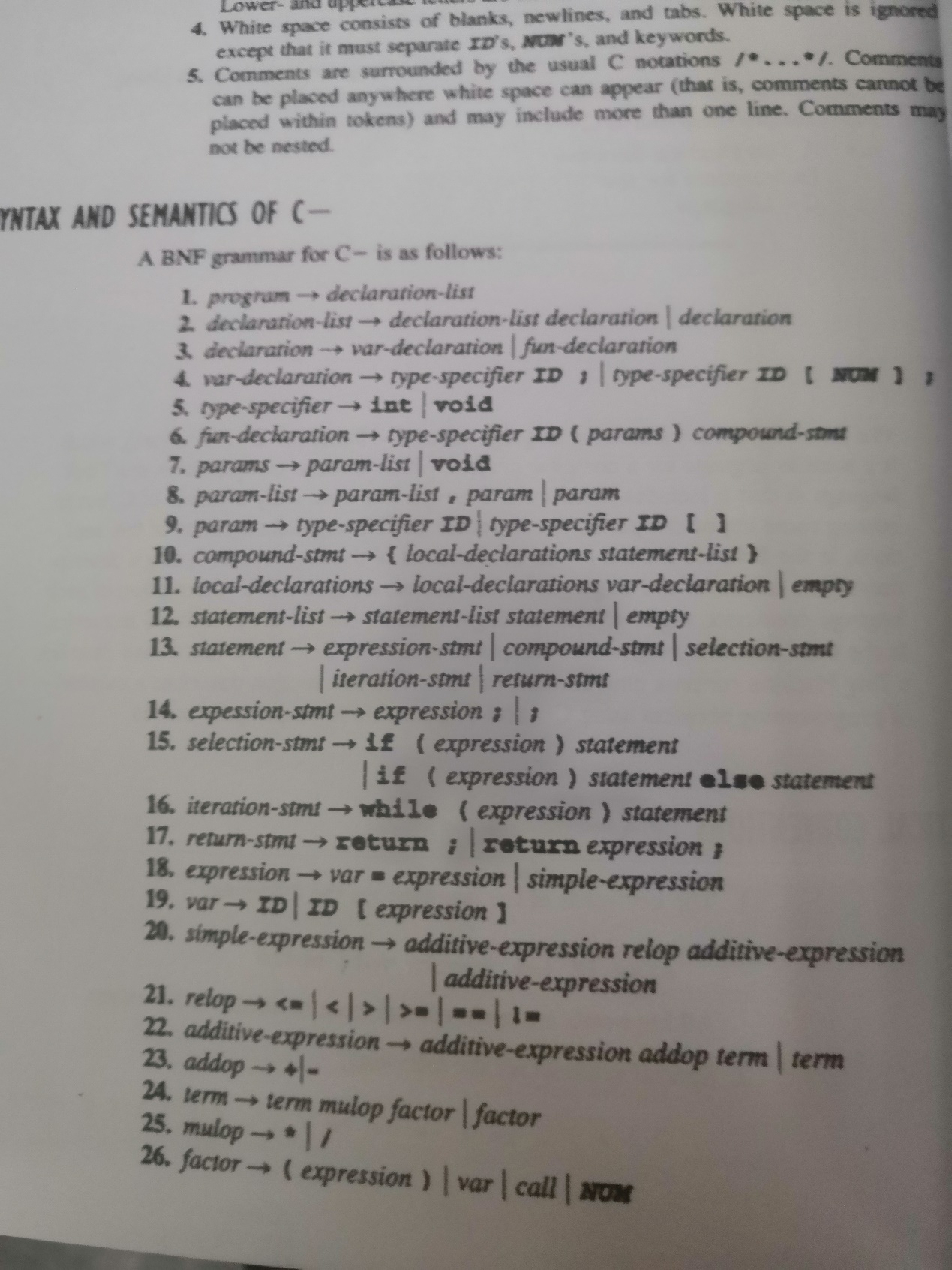
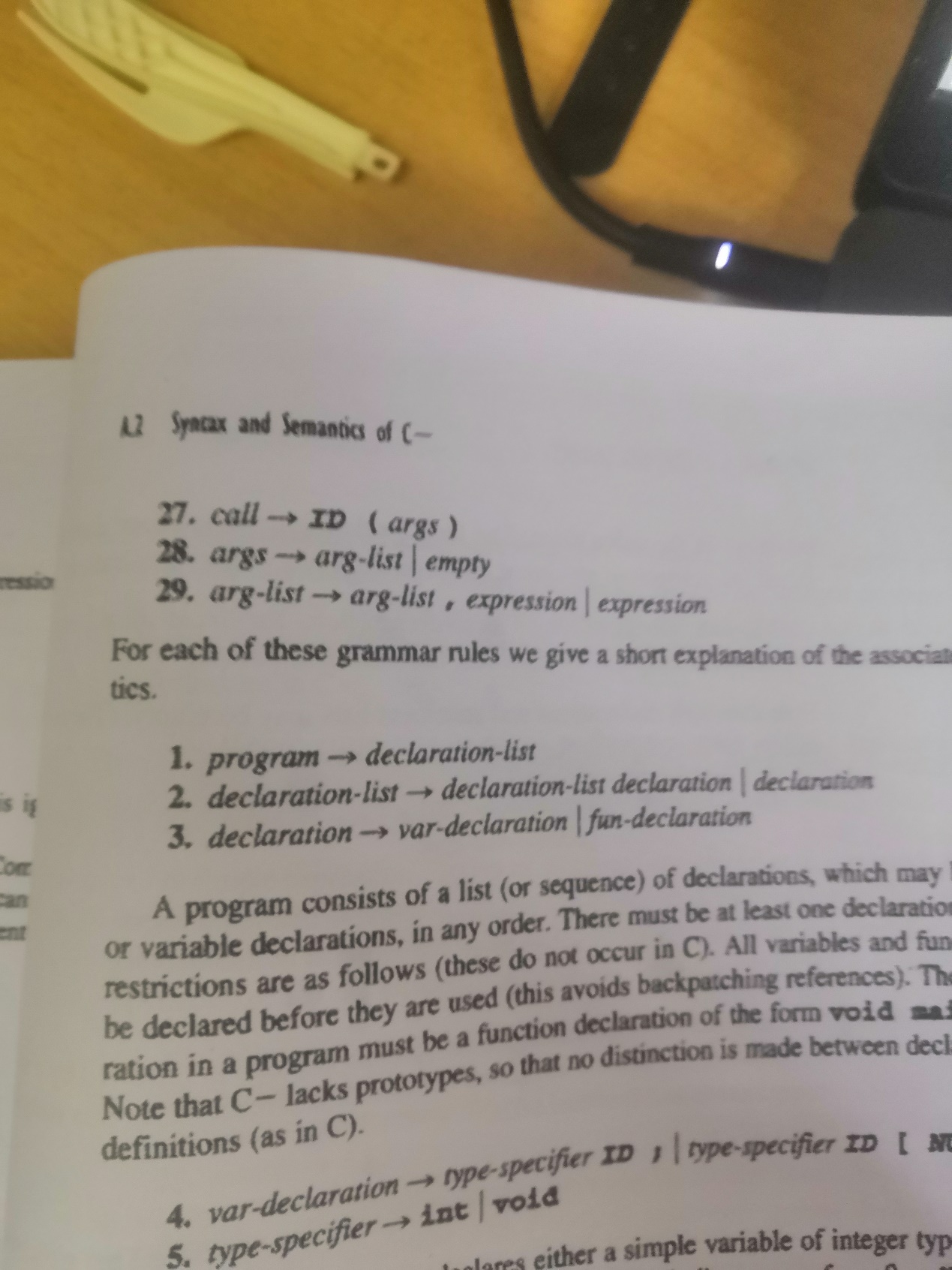
void i(int a[][]);

void j(int a[][5]);

## 2.2 上下文无关文法

我们使用的上下文无关文法主要基于课本附录中C-minus的文法，但是为了实现更多的功能，我们设法进行了一些优化。

课本的C-minus文法如下[[1]](#endnote-1)：

 图中斜体字表示非终继符，粗体正字表示终继符。

不难看出，这些语法规则可以作如下分类：

### 2.2.1面向过程

前三条规则反映了C语言面向过程的特性，我们把C语言的所有程序看做一系列的定义语句，包括函数定义（用户自定义的函数和main函数）和全局变量的定义。

1. program *→* declaration-list
2. declaration-list → declaration-list declaration | declaration
3. declaration → var-declaration | fun-declaration

### 2.2.2变量定义

第4第5条规则规定了变量的定义的语法：

1. var-declaration → type-specifier **ID ;** | type-specifier **ID [ NUM ] ;**
2. type-specifier → **int** | **void**

这两条规则实际上限定了C-只能定义整形和一维整形数组的变量，这不是我们想要的特性，我们对他们进行一系列扩充。

1. var-declaration → base-type id-list **;** | base-type **ID** array-post **;** | base-type **ID** array-post **=** **{** array-const-list **}** **;**
2. id-list → id-list **,** **ID** | id-list **,** **ID** **=** expression | **ID** | **ID** **=** expression
3. array-post → **[** **NUM** **]** | \*array-post **[** **NUM** **]**
4. array-const-list → array-const-list **,** single | single
5. base-type → **int** | **double** | **float** | **char** | **bool**

首先，通过引入了非终继符base-type，实现了整形、双精度浮点数、浮点数、字符和布尔类型的声明，大大提高了语言的灵活性。

其次，通过定义id-list 实现了基本类型的多变量定义，而且每个变量都可以单独初始化，语句int a,b,c;int a=2,b,c;int a=1,b=2,c=3;均合法，但是如前所述，不支持int a,b,c[20];因为我们不认为此时的c和a,b是同种类型

再次，为了实现多维数组的功能，我们需要非终继符array-post, 使其能够产生一对对括号。

最后，我们为了实现以为数组的初始化，设计了array-const-list，用于产生由逗号分割的数字列表，我们之所以不实现多维数组的初始化将在后文讲述。

### 2.2.3 函数定义

课本第6-9条规范了函数的定义：

1. fun-declaration → type-specifier **ID** **(** params **)** compound-stmt
2. params → param-list | **void**
3. param-list → param-list **,** param | param
4. param → type-specifier **ID** | type-specifier **ID** **[ ]**

通过params和param-list可以生成仅含有void，或者有包括一维数组在内的参数列表。

我们将其改进为：

1. fun-declaration → base-type **ID** **(** params **)** compound-stmt | **void** **ID** **(** params **)** compound-stmt | **extern** base-type **ID** **(** params **)** | **extern** **void** **ID** **(** params **)**
2. params → param-list | **void**
3. param-list → param-list **,** param | param
4. param → base-type **ID** | base-type **ID** array-post-param
5. array-post-param → **[** **]** | **[** **NUM** **]** | array-post-param **[** **NUM** **]**

这样我们在定义是可以指明外部函数，参数列表可以包含各个维度的数组，和标准C一样，除标识符最右侧的方括号内可以为空之外，其余括号均需指明数值。

### 2.2.3 复合语句结构

课本的语句实现为：

1. compount-stmt → **{** local-declarations statement-list **}**
2. local-declarations → local-declarations var-declaration | ***ϵ***
3. statement-list → statement-list statement | ***ϵ***

我们将其改进为：

1. *compound-stmt* → **{** *local-declarations statement-list* **}** | **{** *local-declarations* **}** | **{** *statement-list* **}** | **{** **}**
2. *local-declarations* → *local-declarations* *var-declaration* | *var-declaration*
3. *statement-list* → *statement-list* *statement* | *statement*

我们继承了定义语句严格先于其他语句这点，不过，比起课本的书写方式，我们有意回避在表达式中出现空白符，这和yacc的特性有关，我们将在下文详述，这里只需注意两套文化并在识别的语言上没有本质区别即可。

### 2.2.4 语句

课本的定义如下：

1. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt*
2. *expression-stmt* → *expression* **;**  | **;**
3. *selection-stmt* → **if** **(** *expression* **)** *statement* | **if** **(** *expression* **)** *statement* **else** *statement*
4. *iteration-stmt* →  **while** **(** *expression* **)** *statement*
5. *return-stmt* → **return ;**  | **return** *expression* **;**

这些语法指定了单独的语句可以成为的语句类型，比如赋值、分支、迭代和返回，但是注意这里将所有迭代语句都作为while语句，可以进行改进：

1. *statement-list* → *statement-list* *statement* | *statement*
2. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt*
3. *expression-stmt* → *expression* **;** | **;**
4. *selection-stmt* → **if** **(** *expression* **)** *statement* | **if** **(** *expression* **)** *statement* **else** *statement*
5. *iteration-stmt* → *while-stmt* | *for-stmt*
6. *while-stmt* → **while** **(** *expression* **)** *statement*
7. *for-stmt* → **for** **(** *expression* **;** *expression* **;** *expression* **)** *statement*
8. *return-stmt* → **return ;** | **return** *expression* **;**

### 2.2.4 表达式

课本的表达式定义：

1. *expression* → *var* **=** *expression* | *simple-expression*
2. *var* → **ID** | **ID** **[** *expression* **]**
3. *simple-expression* → *additive-expression* *relop* *additive-expression* | *additive-expression*
4. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
5. *additive-expression* → *additive-expression* *addop* *term* | *term*
6. *addop* → **+**|**-**
7. *term* → *term* *mulop* *factor* | *factor*
8. *mulop* → **\***| **/**
9. *factor* → **(** *expression* **)** | *var* | *call* | **NUM**

其对于运算符的支持比较少，我们需要支持更多的运算符，但是考虑到更多的运算符如果将优先级写入规则的话，语法的数量将会非常多，于是我们将优先级/结合性和语法分离，语法仅写作如下：

1. *expression* → *var* *assop* *expression* | *operand*
2. *assop* → **=** | **+=** | **-=** | **\*=** | **/=**| **%=**|**^=** | **&=** | **|=** | **<<=** | **>>=**
3. *var* → **ID** | **ID** *array-post*
4. *operand* → *operand* **||** *operand* | *operand* **&&** *operand* | *operand* **|** *operand* |*operand* **^** *operand* | *operand* **&** *operand* | *operand* **=** *operand* | *operand* **!=** *operand*| *operand* **<** *operand* | *operand* **>** *operand* | *operand* **<=** *operand* | *operand* **>=** *operand* | *operand* **<<** *operand* | *operand* **>>** *operand* | *operand* **+** *operand* | *operand* **-** *operand* | *operand* **\*** *operand* | *operand* **/** *operand* | *operand* **%** *operand* | *prefix* **(** *operand* **)** | **(** *operand* **)** | *prefix* *single* | *single*
5. *prefix* → **!** | **~** | **-**
6. *single* → *var* | *call* | **NUM**| **DOUBLE** | **CHAR** | **TURE** | **FALSE**

对于优先级的实现见下文。

### 2.2.5 函数调用

1. *call* → **ID** ( *args* )
2. *args* → *arg-list* | ***ϵ***
3. *arg-list* → *arg-list* **,** *expression* | *expression*

我们和课本采用完全一致的语法。

## 2.3 语法树结点定义

根据上文提到的文法，我们设计了语法树的节点，节点中储存了从代码中分析出的信息，语法树的结构则表现了代码的结构。各个节点的列表如下：

|  |  |
| --- | --- |
| **类名** | **解释** |
| Node | 所有节点的基类 |
| DeclarationNode | 定义 |
| VarDeclarationNode | 变量定义 |
| IdListNode | 定义的变量列表 |
| FunDeclarationNode | 函数定义 |
| ParamNode | 函数参数 |
| CompoundStmtNode | 复合语句 |
| StatementNode | 各类语句中的一个 |
| SelectionStmtNode | 分支语句 |
| IterationStmtNode | 循环语句 |
| WhileStmtNode | while语句 |
| ForStmtNode | for语句 |
| ReturnStmtNode | 返回值 |
| ExpressionNode | 赋值/运算且赋值/OperandNode中的操作 |
| VarNode | 变量（作为左值） |
| OperandNode | 算数/逻辑操作 |
| SingleNode | 变量/常量/函数返回值 |
| CallNode | 函数调用 |
| IntNode | 整数常量 |
| FloatNode | 浮点型常量 |
| CharNode | 字符类型常量 |
| BoolNode | 布尔类型常量 |

**2.4 Yacc实现原理和方法**

以call为例：

call: ID LP args RP {

$$ = new CallNode(new string($1), $3);

};

当程序解析到ID LP args RP的结构时，就可以新建一个CallNode节点，其中构造函数的第一个参数是string\*，需要将ID（char指针）转化一下，第二个参数是args传递上来的其它Node。新建好这个节点之后，将其赋给$$，就可以让其他地方出现的call给当前的语法提供CallNode了。

其他生成式也是类似的，只不过新建的是其他类型的节点，当有多种生成式时，利用参数是否为nullptr，或者利用构造函数重载的特性进行区分。这样当再去遍历树来打印各个节点或生成代码时，也还保留着原来reduce时的信息。

**2.5 语法树的遍历和打印**

以CallNode为例：

void CallNode::printNode(int layer) {

    printPrefix(layer);

    cout << "Call" << endl;

    if(id) {

        printPrefix(layer + 1);

        cout << \*id << endl;

    }

    if(args) {

        for (auto p : \*args) {

            p->printNode(layer + 1);

        }

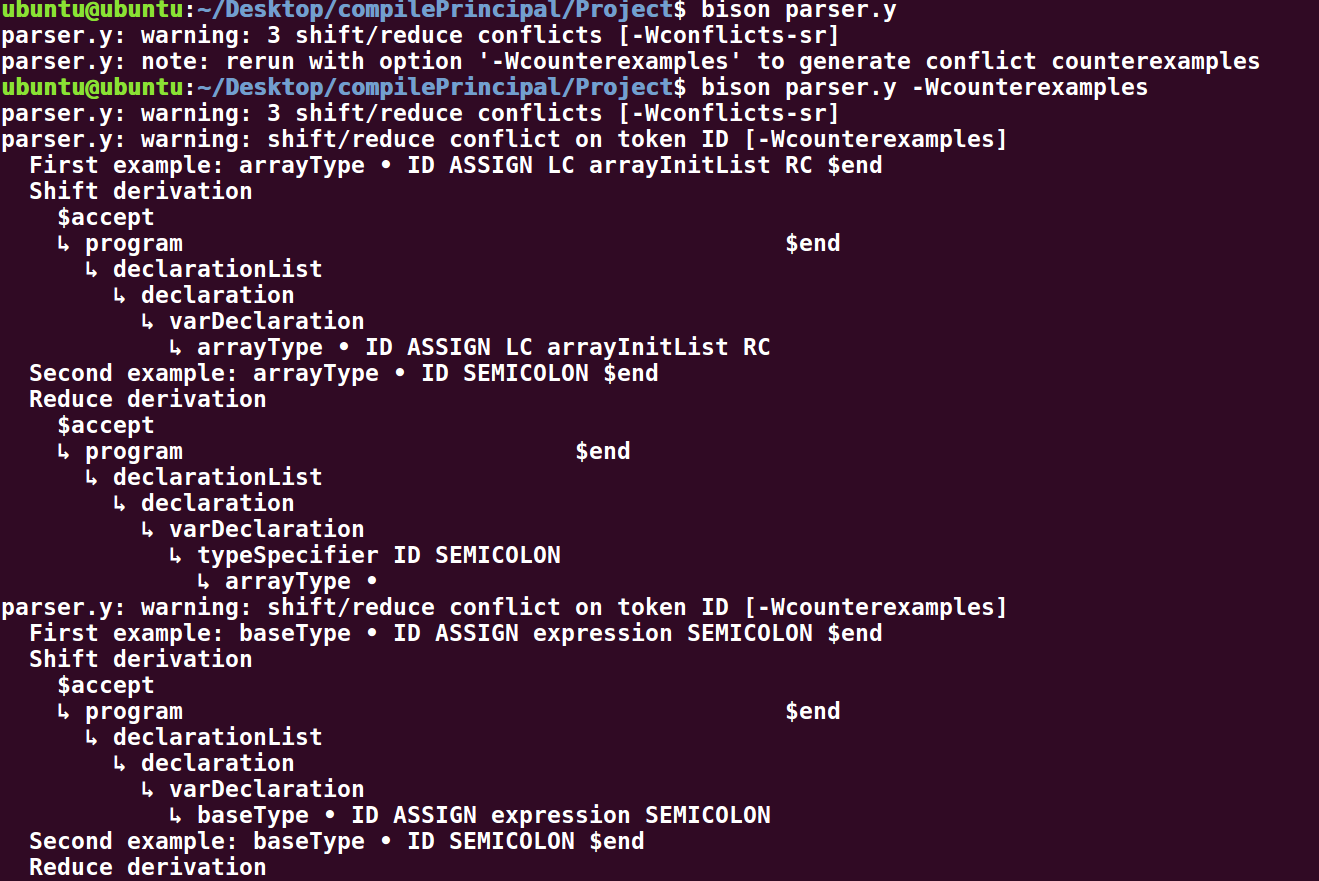
    }

}

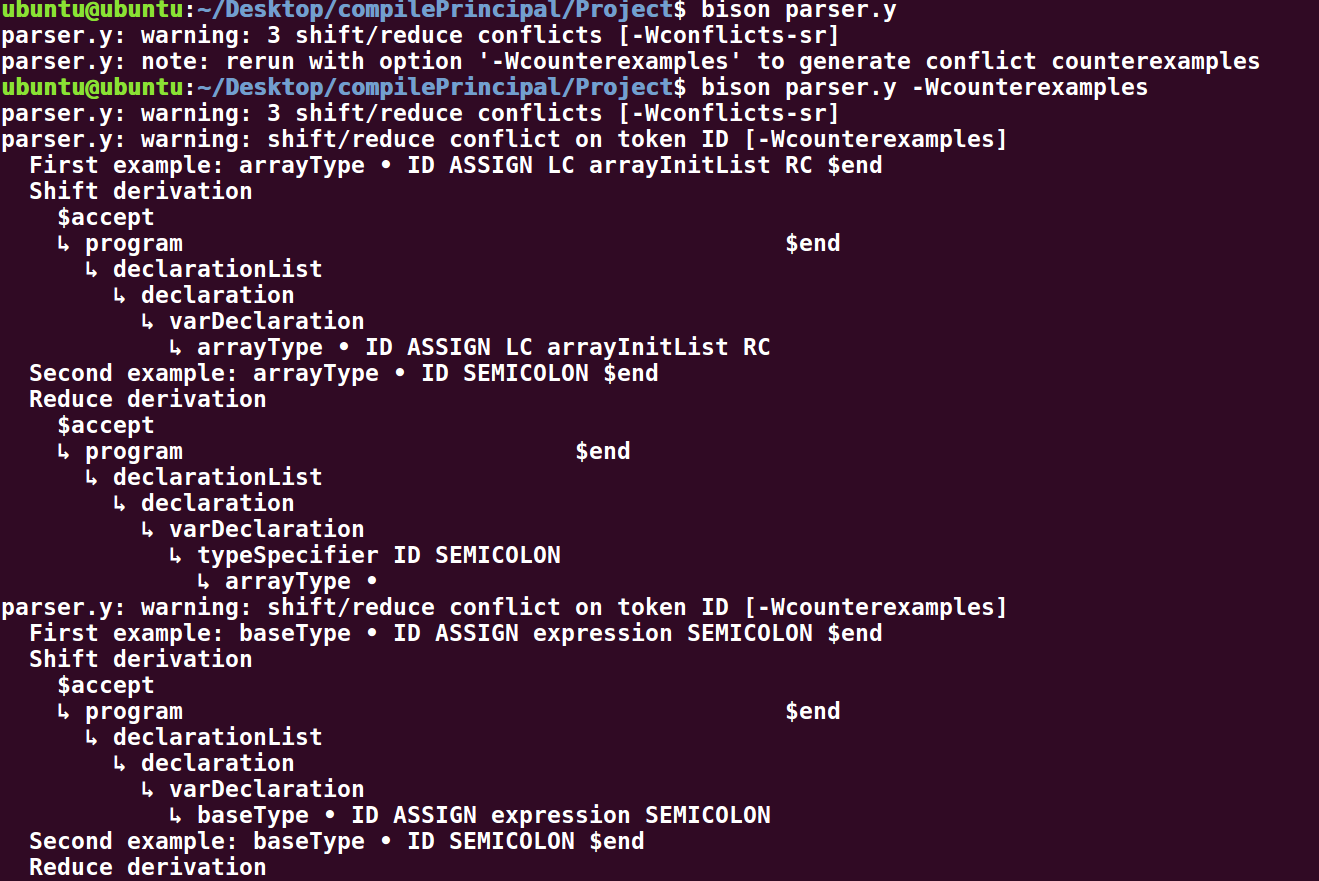
基类Node定义了一个虚函数printNode(int layer)，不同的node对其又有不同的实现。只需要先打印出层的前缀（即树枝部分）和当前节点的名字，然后对当前节点的成员进行展示即可。打印成员的时候对于基本类型可以使用cout输出，对于Node则又可以调用它的printNode函数。这样递归地调用下去，就可以得到完整的语法树。

**2.6 测试**

**2.6.1简易测试和冲突处理**

 我们在完成parser.y之后可以首先尝试使用bison编译出parser.tab.h和parser.tab.c文件，在修改掉所有语法错误之后，我们得到了几个warning，基本上都是shift/reduce conflicts[移进/规约冲突]，在较低版本的bison中，我们可能需要自己发现冲突所在的规则，但是在3.7.6及以上的版本中，bison增加了自动显示冲突例子的内容，而且会在提示用户存在移进/规约冲突之后使用相关的命令显示例。如下：

按照提示，我们使用命令-Wcounterexamples重新编译，可以得到具体的冲突实例：

我们注意，第一次出现语句warning：shift/reduce conflict on token ID [-Wcounterexample]是指，第一个可能出现冲突的实例出现在遇到标识符ID的情况下，其中Fisrst example[例一]和second example[例二]都显示了栈中和输入的情况，通过间隔号·分隔，左边为栈中，右边为等待输入，栈中只含有arrayType，假如读取的标记是ID，则会出现冲突，例一是指，首先移进ID ASSIGN LC arrayInitLit RC，之后在将这四者连同arrayType规约为varDeclaration的过程，是移进优先，而例二是指，首先将arrayType规约为typeSpecifier，之后再和移进ID SEMICOLON进行规约，是规约优先的选择，二者构成了冲突。

结合语义，我们成功通过修改规则消灭了除dangling-else之外的所有移进/规约问题，在语法设计已经没有大问题了。

**2.6.2一个粗暴的手段：更换Yacc的匹配算法**

Bison 默认的LR 算法是LALR，如果说发现设计的语法部分元素需要LR(1) 算法进行识别，那么LALR 算法是不太够的。在.y 文件开头设置开启LR(1) 算法，可以解决部分冲突。

%define lr.type canonical-lr

**2.6.3 打印语法树的效果**

int arr[12] = {1, 4, 5};

extern void printf(char str[]);

int main(void) {

    int a = 10;

    int b = 0;

    int i;

    for (i = 0; i < a; i += 1) {

        if( i % 2 == 0) {

            b += 1;

        }

    }

    a += 2;

    return a;

}

输出语法树如下：

Program

|---Declaration

| |---VarDeclaration

| | |---INT\_TYPE

| | |---IdList

| | | |---arr

| | |---ArrayPost

| | | |---12

| | |---Single

| | | |---Int: 1

| | |---Single

| | | |---Int: 4

| | |---Single

| | | |---Int: 5

|---Declaration

| |---FunDeclaration

| | |---void

| | |---printf

| | |---Param

| | | |---CHAR\_TYPE

| | | |---str

|---Declaration

| |---FunDeclaration

| | |---INT\_TYPE

| | |---main

| | |---CompoundStmt

| | | |---VarDeclaration

| | | | |---INT\_TYPE

| | | | |---IdList

| | | | | |---a

| | | | | |---Expression

| | | | | | |---Operand

| | | | | | | |---Single

| | | | | | | | |---Int: 10

| | | |---VarDeclaration

| | | | |---INT\_TYPE

| | | | |---IdList

| | | | | |---b

| | | | | |---Expression

| | | | | | |---Operand

| | | | | | | |---Single

| | | | | | | | |---Int: 0

| | | |---VarDeclaration

| | | | |---INT\_TYPE

| | | | |---IdList

| | | | | |---i

| | | |---Statement

| | | | |---IterationStmt

| | | | | |---ForStmt

| | | | | | |---Expression

| | | | | | | |---ASSIGN

| | | | | | | |---Var

| | | | | | | | |---i

| | | | | | | |---Expression

| | | | | | | | |---Operand

| | | | | | | | | |---Single

| | | | | | | | | | |---Int: 0

| | | | | | |---Expression

| | | | | | | |---Operand

| | | | | | | | |---LT

| | | | | | | | |---Operand

| | | | | | | | | |---Single

| | | | | | | | | | |---Var

| | | | | | | | | | | |---i

| | | | | | | | |---Operand

| | | | | | | | | |---Single

| | | | | | | | | | |---Var

| | | | | | | | | | | |---a

| | | | | | |---Expression

| | | | | | | |---PLUSASSIGN

| | | | | | | |---Var

| | | | | | | | |---i

| | | | | | | |---Expression

| | | | | | | | |---Operand

| | | | | | | | | |---Single

| | | | | | | | | | |---Int: 1

| | | | | | |---Statement

| | | | | | | |---CompoundStmt

| | | | | | | | |---Statement

| | | | | | | | | |---SelectionStmt

| | | | | | | | | | |---Expression

| | | | | | | | | | | |---Operand

| | | | | | | | | | | | |---EQ

| | | | | | | | | | | | |---Operand

| | | | | | | | | | | | | |---MOD

| | | | | | | | | | | | | |---Operand

| | | | | | | | | | | | | | |---Single

| | | | | | | | | | | | | | | |---Var

| | | | | | | | | | | | | | | | |---i

| | | | | | | | | | | | | |---Operand

| | | | | | | | | | | | | | |---Single

| | | | | | | | | | | | | | | |---Int: 2

| | | | | | | | | | | | |---Operand

| | | | | | | | | | | | | |---Single

| | | | | | | | | | | | | | |---Int: 0

| | | | | | | | | | |---Statement

| | | | | | | | | | | |---CompoundStmt

| | | | | | | | | | | | |---Statement

| | | | | | | | | | | | | |---Expression

| | | | | | | | | | | | | | |---PLUSASSIGN

| | | | | | | | | | | | | | |---Var

| | | | | | | | | | | | | | | |---b

| | | | | | | | | | | | | | |---Expression

| | | | | | | | | | | | | | | |---Operand

| | | | | | | | | | | | | | | | |---Single

| | | | | | | | | | | | | | | | | |---Int: 1

| | | | | | | | | | |---Statement

| | | | | | | | | | | |---CompoundStmt

| | | | | | | | | | | | |---Statement

| | | | | | | | | | | | | |---Expression

| | | | | | | | | | | | | | |---PLUSASSIGN

| | | | | | | | | | | | | | |---Var

| | | | | | | | | | | | | | | |---b

| | | | | | | | | | | | | | |---Expression

| | | | | | | | | | | | | | | |---Operand

| | | | | | | | | | | | | | | | |---Single

| | | | | | | | | | | | | | | | | |---Int: 1

| | | |---Statement

| | | | |---Expression

| | | | | |---PLUSASSIGN

| | | | | |---Var

| | | | | | |---a

| | | | | |---Expression

| | | | | | |---Operand

| | | | | | | |---Single

| | | | | | | | |---Int: 2

| | | |---Statement

| | | | |---ReturnStmt

| | | | | |---Expression

| | | | | | |---Operand

| | | | | | | |---Single

| | | | | | | | |---Var

| | | | | | | | | |---a

我们的测试代码中包含了数组定义、函数定义、局部变量定义、for循环、if分支等操作。而从我们生成的语法树里面也可以看到与之对应的所有结构，如标红的for循环的三个表达式和一个语句块、if语句的条件以及分支。经过验证，该语法树与源代码是对应的。

# 符号表设计

### 数据结构设计

我们采用了C++中自带的vector、map和pair三种数据结构来设计符号表。符号表的定义语句为：

static vector<map<string, pair<Value \*, vector<int> > > > varTable;

每次进入一个由花括号包裹的复合语句，程序都会向varTable中添加新的map成员，也即添加属于该作用域的符号表。如果所定义的变量为数组变量，则在map中插入键值对{变量名:<变量指针,数组形状>}，其中数组形状由vector<int>类型的对象来储存；而如果所定义的变量为非数组变量，则新建一个空vector填入与数组形状对应的参数。

### 符号表操作

进行中间代码生成时，每次读取到一个变量定义，就将其填入位于符号表vector尾部的map中，如果该变量是数组类型，则相应地填入该数组的形状。每次读取到一个变量名引用，就从符号表中查出其指针，而后通过llvm::CreateLoad方法返回其值。如果是数组类型的变量，则应额外根据下标以及其形状来计算出所需要读取的元素的地址，而后取值并进行引用。

### 作用域

每次进入一个复合语句，或者每次进入函数体，程序都会向符号表vector中插入一个空的map对象，该map对象即为该作用域下的符号表。随后，每次程序读取到一个变量定义，程序就会将该变量的变量名和相关信息记录到该map对象中。

# 优化考虑

## 4.1 词法设计优化

如前所述，我们曾经设法将C语言的多行注释用正则表达式表达出来，但是其内容比较复杂，而且验证的复杂性比较高，因此，我们通过研究lex的语法设计了新的有限状态机状态<­C\_COMMENT>来实现对C语言注释的词法分析，大大降低了复杂度，同时也学习了lex通过新定义状态进行操作的功能。

## 4.2 语法设计优化

首先，bison默认使用LALR的方法解析语言，因此，我们可以在书写规则时偏好使用左递归，如课程所述，左递归可以避免长语句在分析时的爆栈。

其次，我们看之前提到的表达式的优先级和结合性的问题，如果将优先级写死在语法里面，那么我们的语法将会如下所述：

26. *simple-expression* → *simple-expression* *logop* *logic-expression* | *logic-expression*

1. *logop* → **||** | **&&**
2. *logic-expression* → *logic-expression* *bitop* *bit-expression* | *bit-expression*
3. *bitop* → **|** | **^** | **&**
4. *bit-expression* → *bit-expression* *relop* *shift-expression* | *shift-expression*
5. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
6. *shift-expression* → *shift-expression* *shiop* *additive-expression* | *additive-expression*
7. *shiop*→ **<<** | **>>**
8. *additive-expression* → *additive-expression* *addop* *term* | *term*
9. *addop* → **+**|**-**
10. *term* → *term* *mulop* *factor* | *factor*
11. *mulop* → **\***| **/** | **%**
12. *factor* → **!** *factor* | **~** *factor* | **-** *factor* | *incre*
13. *incre* → *incre***--** | *incre***++** | *terminal*
14. *terminal* → **(** *expression* **)** | *var* | *call* | **NUM**| **DOUBLE** | **CHAR** | **TURE** | **FALSE**

其内容将会非常复杂，我们可以直接将不含优先级的语法直接翻译成yacc的代码，通过排列或号之间各个子句的顺序，得到子句的优先级，这是yacc支持的，而结合性，则可以使用yacc内置的伪变量%left，%right指定。

%right ASSIGN PLUSASSIGN MINUSASSIGN MULTASSIGN DIVASSIGN MODASSIGN BORASSIGN BXORASSIGN BANDASSIGN SRASSIGN SLASSIGN

%left LOR

%left LAND

%left BOR

%left BXOR

%left BAND

%left EQ NEQ

%left LT GT LTE GTE

%left SL SR

%left PLUS MINUS

%left MULT DIV MOD

%right LNOT BNOT

......

%%

......

expression:         var assop expression { $$ = new ExpressionNode($2, $1, $3, nullptr); }

                    | operand { $$ = new ExpressionNode(nullptr, nullptr, nullptr, $1); }

                    ;

assop:              ASSIGN { $$ = new string("ASSIGN"); }

                    | PLUSASSIGN { $$ = new string("PLUSASSIGN"); }

                    | MINUSASSIGN { $$ = new string("MINUSASSIGN"); }

                    | MULTASSIGN { $$ = new string("MULTASSIGN"); }

                    | ......

                    ;

var:                ID { $$ = new VarNode(new string($1), nullptr); }

                    | ID arrayPost { $$ = new VarNode(new string($1), $2); }

                    ;

operand:            operand LOR operand { $$ = new OperandNode(new string("LOR"), $1, $3, nullptr); }

                    | operand LAND operand { $$ = new OperandNode(new string("LAND"), $1, $3, nullptr); }

                    | operand BOR operand { $$ = new OperandNode(new string("BOR"), $1, $3, nullptr); }

                    | operand BXOR operand { $$ = new OperandNode(new string("BXOR"), $1, $3, nullptr); }

                    | operand BAND operand { $$ = new OperandNode(new string("BAND"), $1, $3, nullptr); }

                    | operand EQ operand { $$ = new OperandNode(new string("EQ"), $1, $3, nullptr); }

                    |

 ......

再次，我们考虑尽可能减少语法中出现的empty语句，以免shift-reduce冲突，实践中，我们通过将含有空字符串的语法重新设计去除了很多移进/规约冲突的警告。

还有，孤悬-否则问题（dangling-else problem）如前所述，不需要我们解决，yacc在面临shift-reduce冲突的过程中，总会选择优先shift，这和我们设计if-else语句时，偏好将else和最靠近的无配对if配对是一致的，于是我们将允许这样的warning存在。

最后，我们解释一下为何我们不再支持使用上下文无关语法实现多维数组的初始化，因为我们发现，如若我们使用语句：

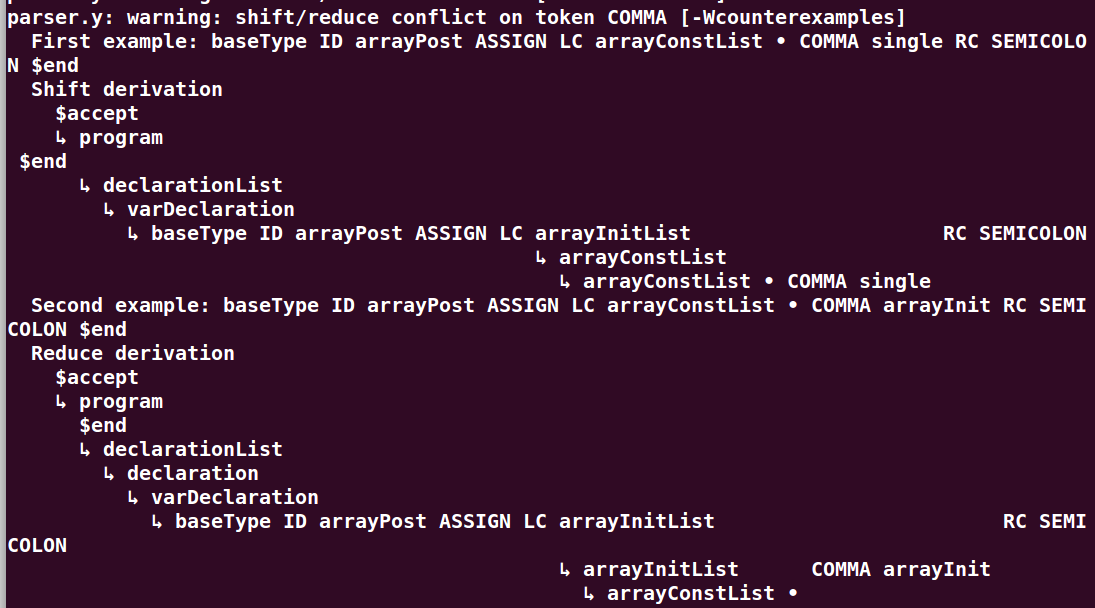
arrayInitList:      arrayInitList COMMA arrayInit {}

                    | arrayInit {}

| arrayConstList {}

;

arrayInit:          LC arrayInitList RC {};

 那么我们将会产生shift-reduce冲突，如下：

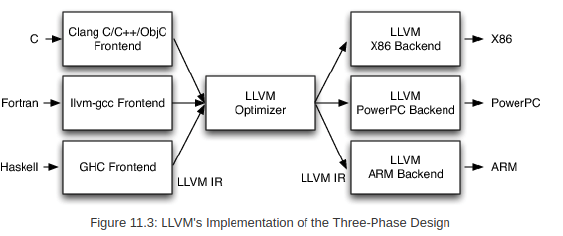
arrayConstList是一系列用逗号分割的初始值的列表，现在的问题是，当我们读入了这样一个列表之后，如果遇到逗号，我们无法分辨这个逗号是需要在这个初始值列表之后新加入一个初始值元素，还是已经表示逗号之后需要新加入一个包含大括号的，新的一列初始值列表，这是LR(1)无法解决的问题，可能需要2位前瞻才可以做到。也许重新设计语法会消除这个问题，但是我们最终因为时间有限放弃了多维数组的初始化。

## 4.3 llvm优化

llvm本身也会对中间代码进行一些优化，如常数直接运算直接在编译时计算。更高级的优化方式可以通过PassManager进行调配。

# 代码生成

## 5.1 llvm模块和函数



调用<llvm/IR/……>中的文件内容，我们可以使用llvm现成的模块，有利于中间代码的生成。

### 5.1.1 Module

llvm::Module模块将会包括所有的函数和全局变量，而作为顶层结构，该模块也会拥有所有中间代码的信息。

### 5.1.2 Function

llvm::Function模块将会通过llvm::FunctionType类型的子对象来记录函数返回值类型、函数参数类型、链接方法等函数原型信息。而得到正确的llvm::FunctionType对象后，就可以通过Function::Create方法来创建函数，并将相关信息注册到llvm::Module模块中。

### 5.1.3 BasicBlock

BasicBlock 是代码块写入和管理的基本单位，诸如跳转、函数定义等功能均需要使用BasicBlock对象来进行分块与规划，而IRBuilder所写入的代码位置也常常需要通过特定的BasicBlock来指定。

### 5.1.4 IRBuilder

IRBuilder是代码生成和插入的基本管理模块，基本的数学运算、变量生成和流程控制对应的中间代码均需要通过IRBuilder的对应方法来进行生成。

### 5.1.5 Value

Value是基本的数据管理类，局部变量、全局变量、常量、运算结果、代码量等数据量均可以直接或者简介地转化成Value类型。同时，Value类型也包含有变量类型、变量名等大量信息。

## 5.2 环境维护

### 5.2.1 变量信息的维护

变量信息主要通过符号表来进行维护。本程序中，符号表用一个模板类型为map<string, Value \*>的vector对象来进行维护。每次进入函数体或者进入由花括号包围的复合语句时，程序都会将一个空的map对象添加进入vector，从而创建一个新的定义域。读取到变量定义时，程序会将其添加进vector对象当前的最后一个map中，也即添加到当前定义域的符号表中。读到变量引用时，程序会倒序读取vector中的map，一旦发现与当前引用变量同名的变量，就将其Value指针和维度信息返回，进而进行进一步调用。

### 5.2.2函数信息的维护

我们没有专门使用一个单独的符号表来维护函数信息，因为我们需要的函数信息都已在llvm::Module模块中得到储存。

### 5.3 llvm 类型生成

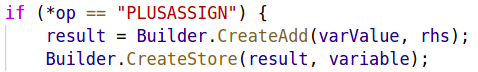
本项目中使用的数据类型和llvm中的数据类型的对应关系如下表所示。

|  |  |
| --- | --- |
| C语言类型 | llvm类型 |
| bool | llvm::Type::getInt1Ty |
| char | llvm::Type::getInt8Ty |
| int | llvm::Type::getInt32Ty |
| float | llvm::Type::getFloatTy |
| void | llvm::Type::getVoidTy |

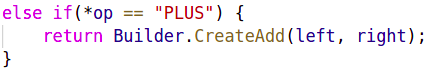
## 5.4 llvm中间代码生成

### 5.4.1表达式生成

表达式生成中主要涉及的两类节点为expression和operand。其中，expression节点主要是为了嵌套地生成赋值语句而设计的，所以只需首先调用expression的codeGen方法获取其值，而后通过CreateStore方法将所得到的结果存入目标变量即可。例如，下图是当前的赋值符号为”PLUSASSIGN”，也即”+=”时，程序所调用的IRBuilder代码片段。



而operand节点主要用于生成常规单目、双目或者括号预算式。例如，下图是当前的运算符号为”PLUS”，也即”+”时，程序所调用的IRBuilder的代码片段：

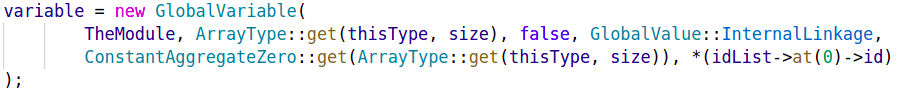


### 5.4.2语句生成

#### 5.4.2.1 变量声明语句

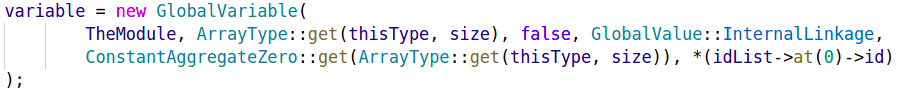
生成变量生成语句的大致过程为：通过节点内保存的变量类型字符串和数组类型相关信息，生成所需的函数类型FunctionType变量，而后调用IRBuilder下的CreateAlloca方法进行空间分配，并进行初始化。这里需要注意的主要有以下几点：

1. 如果所定义的变量时数组变量，则应该使用ArrayType::get方法所返回的结果来作为变量类型传入中间代码生成相关模块，如下图：

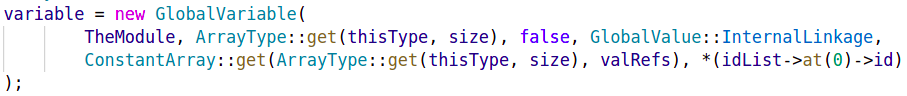


图中，当我们需要声明一个基础类型为thisType的数组时，我们需要先用ArrayType::get(thisType, size)来获取相应的Type \*变量，而后作为参数传入GlobalVariable的构造函数进行初始化。

1. 当所需要定义的为局部变量时，可以使用CreateStore方法来给变量进行初始化，但当所需要定义的为全局变量时，就应在用new方法创建一个Global Variable变量时传入特定的参数进行初始化。如，以数组初始化为例，对全局数组变量进行全零初始化的方法为：



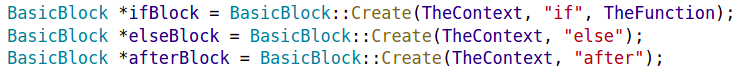
其中ConstantAggregateZero::get方法承担了初始化的功能。而采用常量初始化的方法为:



其中valRefs为ArrayRef<Constant \*>类型的对象变量，我们使用ConstantArray::get方法让其加载到当前正在初始化的variable变量中去。

#### 5.4.2.2控制流语句（if/for/while）

生成控制流语句的关键点在于建立多个BasicBlock并根据其对应关系分别规划其跳转路径和条件。如，在生成if语句块时，总共需要三个BasicBlock来进行管理：



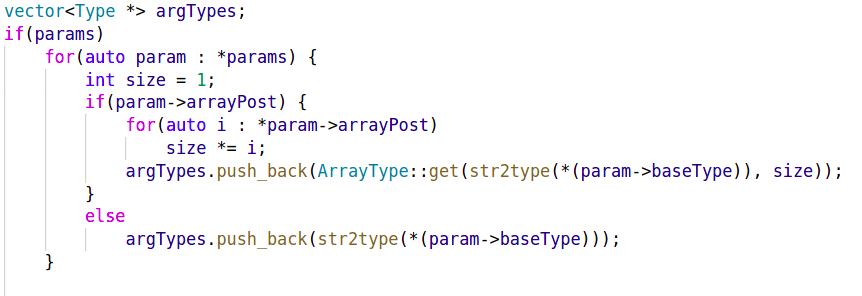
其中，ifBlock主要负责记录条件为真时要执行的代码内容，elseBlock主要负责记录条件为假时要执行的代码内容，而afterBlock主要负责记录if语句块之后需要执行的代码内容。而后，我们需要通过CreateCondBr的方法来进行连接：



如上的代码意为：当cond为真时，跳转执行ifBlock块中的代码内容；而当cond为假时，跳转执行elseBlock中的代码内容。通过多次跳转和主体写入BasicBlock块的修改，我们可以实现if-else、while、for等控制流语句。

### 5.4.3 函数生成

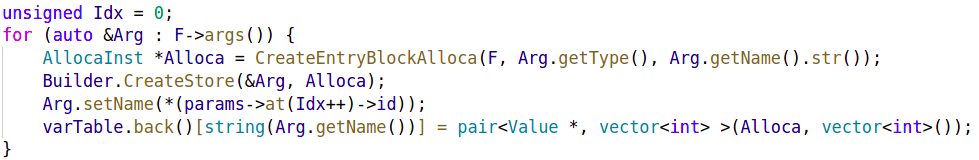
生成函数对应中间代码的主要流程为：首先，我们需要将函数的参数类型以vector<Type \*>的形式进行储存：



而后，我们根据函数的返回类型和参数类型创建函数并链接到当前Module：



最后，我们为函数参数分别声明变量并分配空间，插入符号表：



至此，我们就完成了自定义函数声明的全部过程。

# 测试案例

（每个语句成分的测试案例， 至少两个复合语句组合后的测试案例）

**6.1 构建工程**

shell编译

bison -d parser.y

flex lexer.l

clang++ -g -O3 lex.yy.c parser.tab.c ASTNodes.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core orcjit native` -w -DLLVM\_DISABLE\_ABI\_BREAKING\_CHECKS\_ENFORCING=1

或CMakeList.txt（在CLion中使用）

cmake\_minimum\_required(VERSION 3.19)

project(compile)

set(CMAKE\_CXX\_STANDARD 20)

set(CMAKE\_SYSROOT "${SYSROOT\_PATH}")

set(CMAKE\_FIND\_ROOT\_PATH "${SYSROOT\_PATH}" "${CMAKE\_PREFIX\_PATH}" "${TOOLCHAIN\_PATH}")

set(CMAKE\_FIND\_ROOT\_PATH\_MODE\_PROGRAM NEVER)

set(CMAKE\_FIND\_ROOT\_PATH\_MODE\_LIBRARY ONLY)

set(CMAKE\_FIND\_ROOT\_PATH\_MODE\_INCLUDE ONLY)

set(CMAKE\_FIND\_ROOT\_PATH\_MODE\_PACKAGE ONLY)

find\_package(LLVM REQUIRED CONFIG)

message(STATUS "Found LLVM ${LLVM\_PACKAGE\_VERSION}")

message(STATUS "Using LLVMConfig.cmake in: ${LLVM\_DIR}")

message("LLVM\_INCLUDE\_DIRS=${LLVM\_INCLUDE\_DIRS}")

message("LLVM\_DEFINITIONS=${LLVM\_DEFINITIONS}")

include\_directories(${LLVM\_INCLUDE\_DIRS})

separate\_arguments(LLVM\_DEFINITIONS\_LIST NATIVE\_COMMAND ${LLVM\_DEFINITIONS})

add\_definitions(${LLVM\_DEFINITIONS\_LIST})

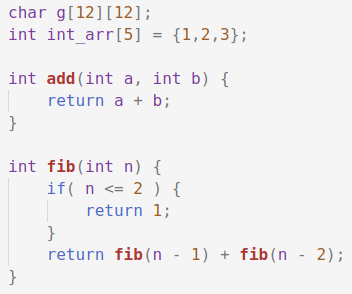
add\_executable(compile ASTNodes.cpp lex.yy.cpp parser.tab.cpp)

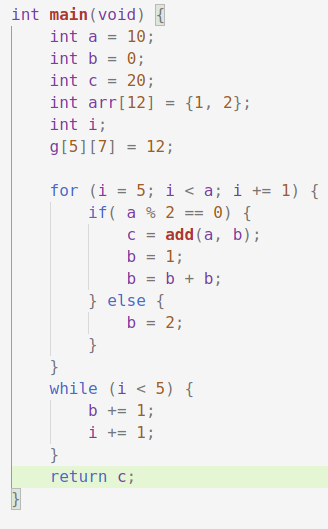
llvm\_map\_components\_to\_libnames(llvm\_libs support core irreader)

target\_link\_libraries(compile ${llvm\_libs})

**6.2 各语句测试**

测试代码，关键内容包括全局变量定义和初始化、多维数组、用户自定义函数、调用函数、局部变量定义和初始化、for循环、while循环、循环中嵌套if语句：





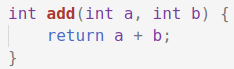
生成的中间代码：

; ModuleID = 'Module'  
source\_filename = "Module"  
  
@g = internal global [144 x i8] zeroinitializer  
@int\_arr = internal global [5 x i32] [i32 1, i32 2, i32 3]  
  
define i32 @add(i32 %a, i32 %b) {  
intoFunction:  
 %0 = alloca i32, align 4  
 %1 = alloca i32, align 4  
 store i32 %a, i32\* %1, align 4  
 store i32 %b, i32\* %0, align 4  
 %a1 = load i32, i32\* %1, align 4  
 %b2 = load i32, i32\* %0, align 4  
 %2 = add i32 %a1, %b2  
 ret i32 %2  
}  
  
define i32 @fib(i32 %n) {  
intoFunction:  
 %0 = alloca i32, align 4  
 store i32 %n, i32\* %0, align 4  
 %n1 = load i32, i32\* %0, align 4  
 %1 = icmp sle i32 %n1, 2  
 br i1 %1, label %if, label %else  
  
if: ; preds = %intoFunction  
  
intoBlock: ; No predecessors!  
 ret i32 1  
 br label %after  
  
else: ; preds = %intoFunction  
 br label %after  
  
after: ; preds = %else, %intoBlock  
 %n2 = load i32, i32\* %0, align 4  
 %2 = sub i32 %n2, 1  
 %calltmp = call i32 @fib(i32 %2)  
 %n3 = load i32, i32\* %0, align 4  
 %3 = sub i32 %n3, 2  
 %calltmp4 = call i32 @fib(i32 %3)  
 %4 = add i32 %calltmp, %calltmp4  
 ret i32 %4  
}  
  
define i32 @main() {  
intoFunction:  
 %i = alloca i32, align 4  
 %arr = alloca [12 x i32], align 4  
 %c = alloca i32, align 4  
 %b = alloca i32, align 4  
 %a = alloca i32, align 4  
 store i32 10, i32\* %a, align 4  
 store i32 0, i32\* %b, align 4  
 store i32 20, i32\* %c, align 4  
 %0 = getelementptr [12 x i32], [12 x i32]\* %arr, i32 0  
 store i32 1, [12 x i32]\* %0, align 4  
 %1 = getelementptr [12 x i32], [12 x i32]\* %arr, i32 1  
 store i32 2, [12 x i32]\* %1, align 4  
 %g = load [144 x i8], [144 x i8]\* getelementptr ([144 x i8], [144 x i8]\* @g, i32 288), align 1  
 store i0 12, [144 x i8]\* getelementptr ([144 x i8], [144 x i8]\* @g, i32 288), align 1  
 %i1 = load i32, i32\* %i, align 4  
 store i32 5, i32\* %i, align 4  
 br label %forLoop  
  
forLoop: ; preds = %after, %intoFunction  
 %i2 = load i32, i32\* %i, align 4  
 %a3 = load i32, i32\* %a, align 4  
 %2 = icmp slt i32 %i2, %a3  
 br i1 %2, label %actualLoop, label %afterFor  
  
actualLoop: ; preds = %forLoop  
  
intoBlock: ; No predecessors!  
 %a4 = load i32, i32\* %a, align 4  
 %3 = srem i32 %a4, 2  
 %4 = icmp eq i32 %3, 0  
 br i1 %4, label %if, label %else  
  
if: ; preds = %intoBlock  
  
intoBlock5: ; No predecessors!  
 %a6 = load i32, i32\* %a, align 4  
 %b7 = load i32, i32\* %b, align 4  
 %calltmp = call i32 @add(i32 %a6, i32 %b7)  
 %c8 = load i32, i32\* %c, align 4  
 store i32 %calltmp, i32\* %c, align 4  
 %b9 = load i32, i32\* %b, align 4  
 store i32 1, i32\* %b, align 4  
 %b10 = load i32, i32\* %b, align 4  
 %b11 = load i32, i32\* %b, align 4  
 %5 = add i32 %b10, %b11  
 %b12 = load i32, i32\* %b, align 4  
 store i32 %5, i32\* %b, align 4  
 br label %after  
  
else: ; preds = %intoBlock  
  
intoBlock13: ; No predecessors!  
 %b14 = load i32, i32\* %b, align 4  
 store i32 2, i32\* %b, align 4  
 br label %after  
  
after: ; preds = %intoBlock13, %intoBlock5  
 %i15 = load i32, i32\* %i, align 4  
 %6 = add i32 %i15, 1  
 store i32 %6, i32\* %i, align 4  
 br label %forLoop  
  
afterFor: ; preds = %forLoop  
 br label %whileLoop  
  
whileLoop: ; preds = %intoBlock18, %afterFor  
 %i16 = load i32, i32\* %i, align 4  
 %7 = icmp slt i32 %i16, 5  
 br i1 %7, label %actualLoop17, label %afterFor21  
  
actualLoop17: ; preds = %whileLoop  
  
intoBlock18: ; No predecessors!  
 %b19 = load i32, i32\* %b, align 4  
 %8 = add i32 %b19, 1  
 store i32 %8, i32\* %b, align 4  
 %i20 = load i32, i32\* %i, align 4  
 %9 = add i32 %i20, 1  
 store i32 %9, i32\* %i, align 4  
 br label %whileLoop  
  
afterFor21: ; preds = %whileLoop  
 %c22 = load i32, i32\* %c, align 4  
 ret i32 %c22  
}

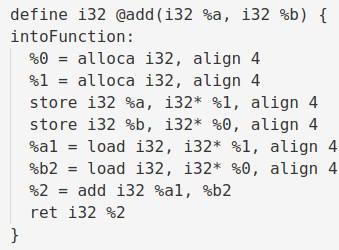
上述中间代码的关键部分已经标红，对应了上文我提到的关键内容，从中间代码的内容和控制流结构看，中间代码的生成也正确。

**6.3 通过gcc与其他目标代码进行链接的测试**

编写简单的加法代码：



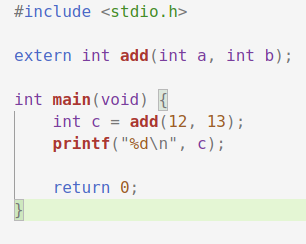
通过我们的编译器生成中间代码：



运行如下指令，生产目标代码：

llc -relocation-model=pic -filetype=obj add.ir -o add.o

编写代码，用extern的方式声明add函数，这样在链接时就可以调用上面的目标代码。



在命令行进行链接，并生成可执行文件，执行的结果正确。



1. compiler construction principles and practice P492 [↑](#endnote-ref-1)