

**本科实验报告**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**序言**

**概述**

编译器就是将“高级语言”翻译为“机器语言（低级语言）”的程序。编译器的主要工作流程为：1. 读取源代码 (source code)作为输入文件。 2. 使用预处理器 (preprocessor)对源代码作预处理。预处理阶段又分为词法分析、语法分析，最终将生成一颗语法树。有时也可以直接生成三地址码等中间代码。 3. 语义分析将语法树遍历生成中间代码。（语法树也是一种中间码）4. 生成目标代码 (object code) 5. 使用链接器 (Linker)将目标代码处理成可执行程序 (executables)。

**文件说明**

上传的文件中：

Mylexer.l 为lex文件，完成词法分析工作。

Myparser.y 为yacc文件，完成语法分析工作。

AST.h 为mylexer.l与myparser.y共有的头文件，主要包含了语法树结点的定义，在语法分析工作中作为数据结构存储形式的定义。

Myparser.tab.c 与Myparser.tab.h 与lex.yy.h 是编译生成的。

1. Out为可执行文件

For.c为测试文件 out.txt为测试输出文件

运行方式：linux下flex mylexer.l bison myparser.y 生成a.out

然后运行即可

IRgenerator.h、IRgenerator.cpp 生成中间代码

main.cpp 程序入口

LLVMtest.cpp LLVM测试代码

array\_for\_test.c、if\_fun\_op\_test.c测试样例

运行：

make array\_for\_test 生成测试样例array\_for\_test.c的LLVM IR

make array\_for\_testrun 运行 array\_for\_test.c并输出

make if\_fun\_op\_test 生成测试样例if\_fun\_op\_test.c的LLVM IR

make if\_fun\_op\_testrun 运行 if\_fun\_op\_test.c并输出

pgcc运行optimize.cpp,输入想优化的.ll文件,结果即会输出到result.ll中,常值优化在语义分析时实现,未在此步,此步主要是乘法的优化

**分工**

**实验目的**

掌握词法分析、语法分析、语义分析和代码生成方法。

**实验环境**

Windows系统环境或者Linux环境。

1、Linux环境下的编译和运行

（1）Linux 2.6以上版本

（2）GCC3.4以上版本

（3）Bison 2.2以上版本

（4）Flex 2.5.33以上版本

发行版可以采用Ubantu, Gentoo, Fedora Core等。

2、Windows环境下的编译和运行

（1）Visual Studio 6.0

（2）Masm 6.0以上版本

（3）ParseGenerator 4.0 （Lex和Yacc的集成开发包）

**实验内容**

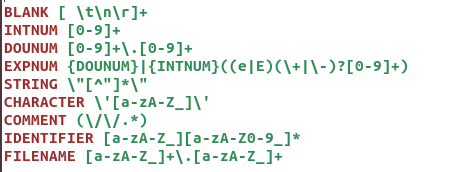
**第一章 词法分析**

在读取源文件后，我们需要对代码进行预处理，第一步通常是词法分析。本次实验中词法由lex完成。

词法分析作为对输入文件的第一道处理，主要的工作是通过正则表达式，将输入文件的词句切分成我们定义好的标记，称为token，以便传送给yacc文件做进一步的语法分析。

**正则表达式**

在lex文件中，首先给出正则表达式的定义来供lex匹配。

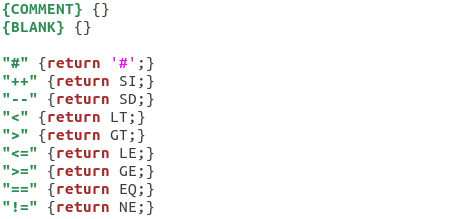


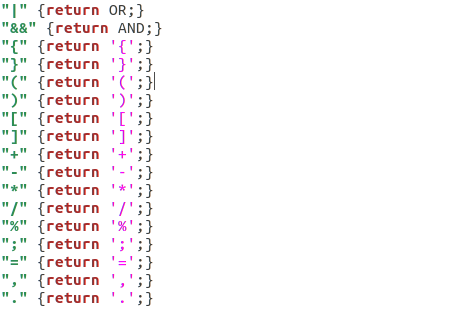
然后再定义匹配与处理，通常处理中需要返回一个标记。匹配的字符串存储在一个yytext的指针类型地址中，如果需要将其保留给yacc文件做语法分析，可以通过yylval绑定给相应的token。Yylval默认为int类型，也可以自己定义数据结构类型来记录匹配的原字符串。Token会被返回给yacc程序完成语法分析。

**匹配**

本工程中给出了一下匹配：

* 注释与空格，以及一些常见符号的匹配。



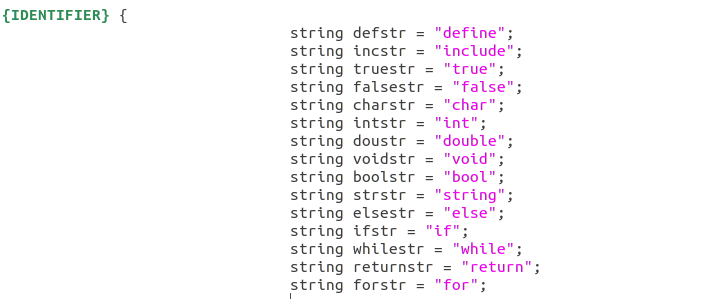


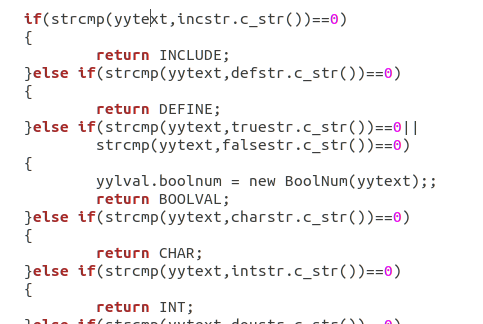
* 对于标识符的匹配。

标识符主要是指一些保留字符串，和自定义的变量名。

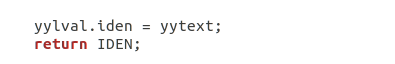
预定义字符串有"define"， "include"， "true"， "false"， "char"， "int"， "double"， "void"， "bool"， "string"， "else"， "if"， "while"， "return"， "for"，将返回相应的token标识。

（部分代码截图如下）



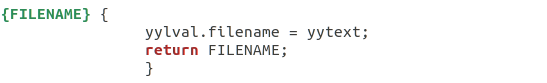


而自定义的变量名则需要返回IDEN标识，并存储该字符串。

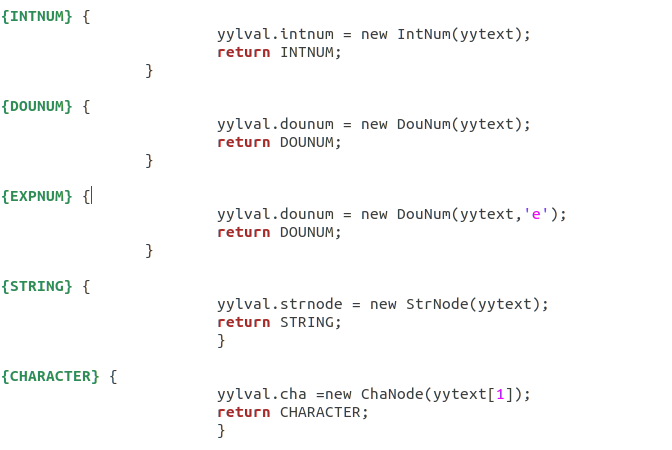


* 头文件名的匹配。

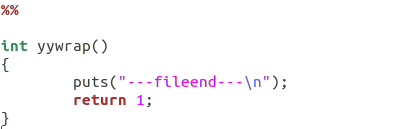
头文件与一般字符串不同（除了含“”外也可能含< >），又与普通iden不同（不含‘.’）。所以单独匹配。



* Int、double、char、string等数值的匹配。（不是标识符，而是具体数据类型的数据存储。）



最后给出yywrap函数的定义，此函数主要负责文件读取完成后的操作。在需要连续读取文件的情况下可以通过在其中对yyin定义来实现连续读取文件，但是本程序只需要处理一个文件，所以直接输出一个字符串。



**第二章 语法分析**

经过lex词法分析后的token作为数据源传给yacc进行进一步分语法分析，可以在这一阶段生成一颗 语法树，也可以直接生成地址码。本程序采取先生成语法树的情形。

**类型定义**

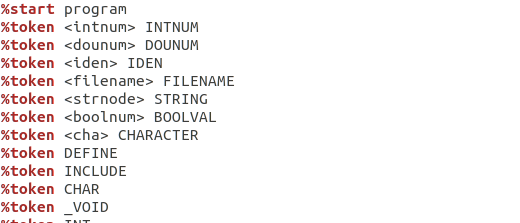
在yacc中，我们首先给出值栈的类型定义。这些定义可以改变值栈类型，（默认是int），方便我们更好地存储数据，完成语法树的构建。

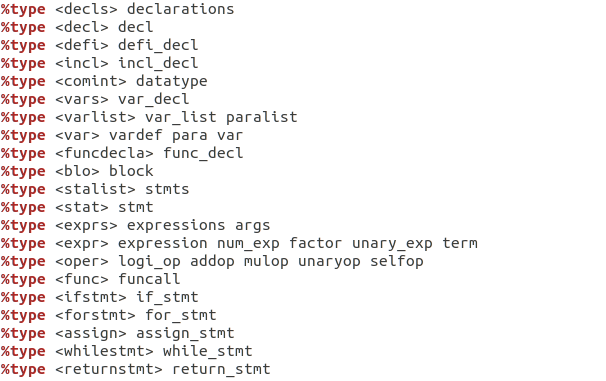
其中一些自定义的变量类型，定义在AST.h文件中，是一些不同类型语法树节点的结构。在后文会贴出部分源码。

（以下是部分源码）



给出值栈定义后，分别定义终结符token和一些将在递归规则中出现的表达式类型。





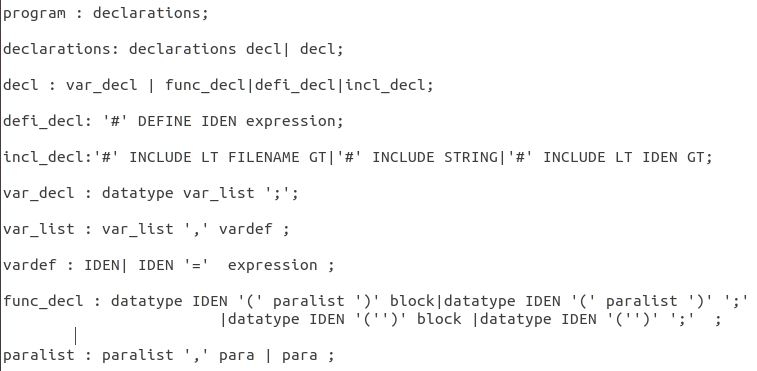
**规则定义与语法树实现**

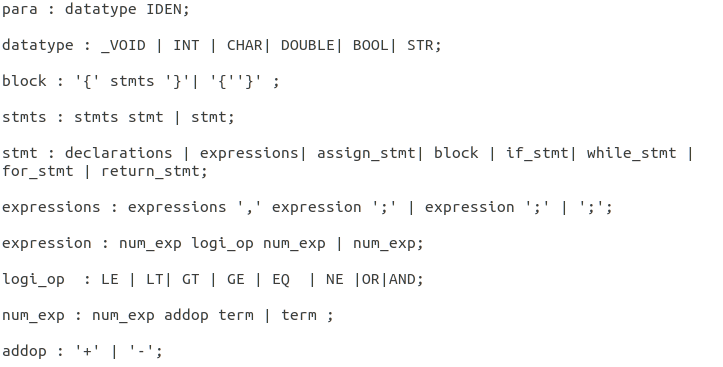
**规则定义**

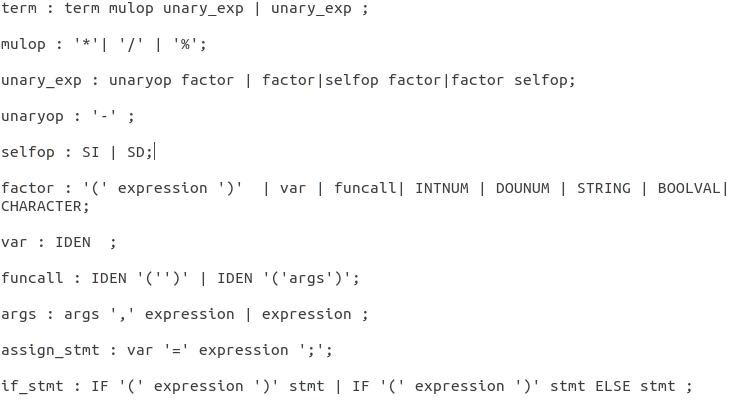
接下来就是比较关键的规则定义与语法树实现了。

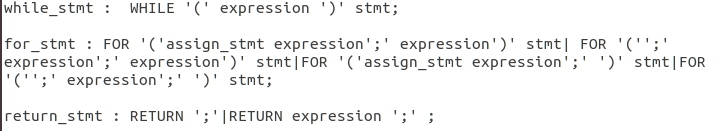
不同语言编译器的规则需要根据语言本身的结构来指定。

根据c语言的特性，我编写了如下的规则：









这些规则包含了c语言基本语法的匹配实现。主要将c语言的语句类型分为了：

1. 声明（含有变量声明、函数声明、define类型声明、include类型声明）。
2. statement语句（含有声明、表达式、赋值语句、代码块、if类型、while类型、for类型、return类型）。
3. 表达式。表达式可以是其他语句的一部分，比如赋值语句。有时候也可以单独作为一个语句（但没有什么实际意义）。表达式下可以处理逻辑运算、数值运算、单目运算、双目运算，根据不同运算符的优先级不同，为其设计了几个递推规则，具体可以查看代码。
4. 另外表达式会处理数值，本程序可以处理如下几种类型的数值：int、double、bool、void、string、char、科学计数法类型数值。同时把自定义变量、函数调用也作为表达可以处理的数值之一。

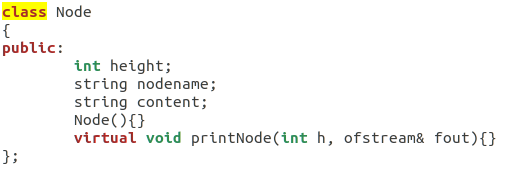
整个语法递推的顶端从声明开始，因为c语言的通常以函数声明为入口，而函数声明中含有一个block，block可以递推成statement。

**语法树定义**

接着就是每个规则具体语法树构建的实现。在这里需要做的就是构建语法树，并将具体数值存储到树中。

首先先介绍AST.h中对语法树节点的定义。

语法树的节点有26种，用以存储不同类型的语句节点。它们有一个共同的父类：

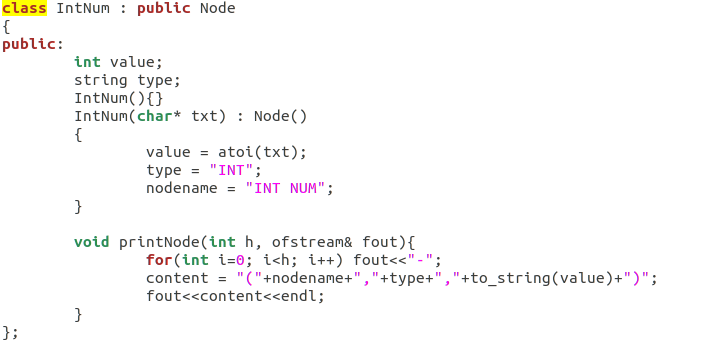


其中height是该节点的高度，nodename是该节点的类型信息。Content是该节点的具体内容。

最重要的是虚函数printNode，需要继承类自己实现，将节点信息打印到指定文件fout里，并且调用该类子节点的printNode函数。这样只需要调用根节点的printNode，就可以完成整颗语法树的打印。

下面给出几个节点的定义代码截图。

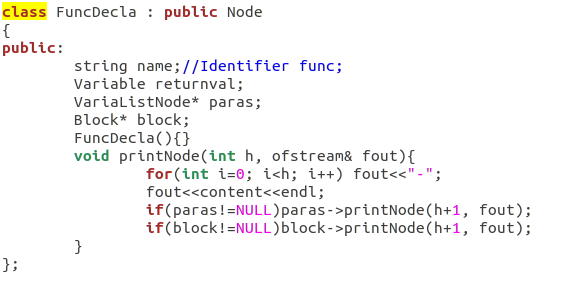
这是一个数值类型节点，类似的还有double、string、char、bool、void等等。



这是一个表达式语句节点的定义。注意这个节点会有自己的子节点。但不同的表达式子节点可能为空，所以printNode中打印非空节点即可。



这是一个函数定义节点。可能需要打印的有参数列表、代码块。



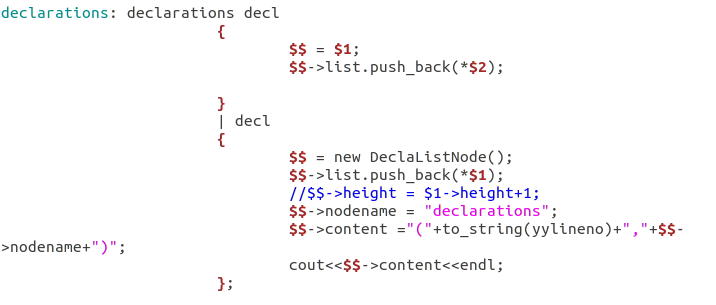
**语法树构建**

接着我们回到yacc继续给出语法树的构建。

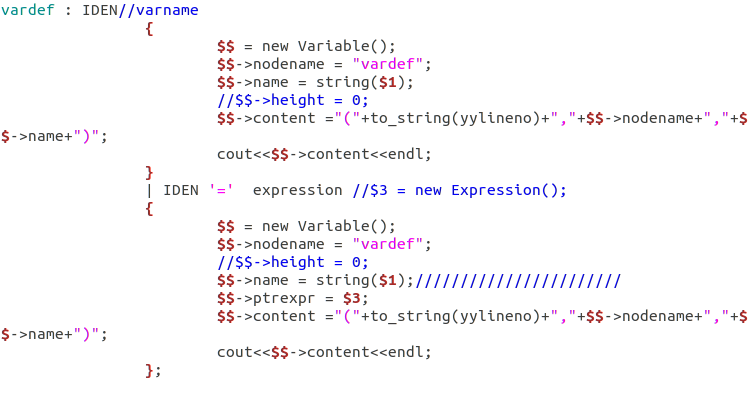
不同的规则下需要的构建动作不同，但主要内容即为值栈的指针分配节点空间、为节点的子节点（如果有）指定对象、给节点的值赋值。

下面给出几个规则的示例：

* Declarations规则



* vardecl变量定义规则



* Expression规则

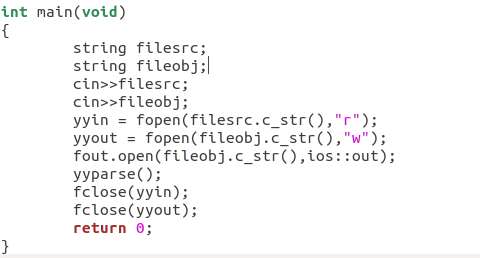


**Main函数**

最后我们需要定义一个main函数，作为yacc程序的入口。在这个程序中，我们需要指定yyin、yyout作为yacc程序读取与输入的io地址。

为了方便数据的保存，我们将yyin与yyout指向两个文件。

通过调用printNode函数，我们将语法树的完整信息打印到了yyout指向的文件中。

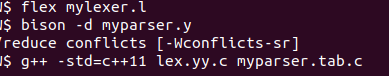


至此，语法分析和文法分析的实现就告一段落了。

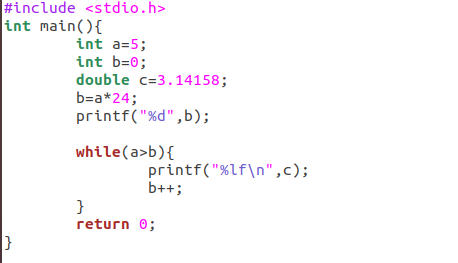
**测试**

我们做几份测试。

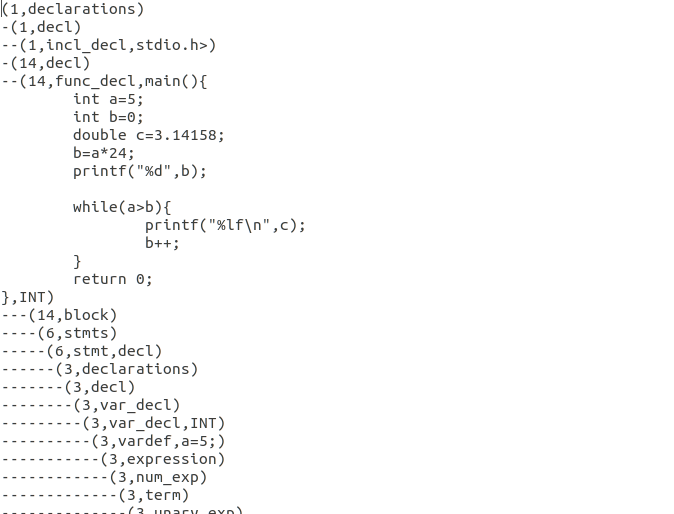
首先编译，生成可执行文件。



测试的文件内容如下：



输出截图：



**第三章 语义分析**

在语义分析阶段，编译器开始对语法树进行一次或多次的遍历，检查程序的语义规则（以类型判断为主）。语义检查的步骤和人对源代码的阅读和理解的步骤差不多，一般都是在遍历语法树的过程中，遇到变量声明和函数声明时，则将变量名——类型、函数名——返回类型——参数数量及类型等信息保存到符号表里，当遇到使用变量和函数的地方，则根据名称在符号表中查找和检查，查找该名称是否被声明过，该名称的类型是否被正确的使用等等。

我们简单以如下几个方面为例展示语义分析的过程：

**语句中的变量、函数是否被声明过**

以变量名为例，在对节点Nidentifier的codeGen时首先判断是否是已经被声明过的变量名，其次用getType判断该变量是否为数组类型。

Value\* NIdentifier::codeGen(CodeGenContext &context) {

Value\* value = context.getSymbolValue(this->name);

if( !value ){

return LogErrorV("Unknown variable name " + this->name);

}

if( value->getType()->isPointerTy() ){

……

if( arrayPtr->getType()->isArrayTy() ){

……

}

}

……

}

**二元运算的两个操作数的类型是否匹配**

指令的L、R必须同属一个类型，结果的类型则必须与操作数的类型相匹配：

if((L->getType()->getTypeID()==Type::DoubleTyID)||(R->getType()->getTypeID() == Type::DoubleTyID)

{

if( (R->getType()->getTypeID() != Type::DoubleTyID) ){

……

}

if( (L->getType()->getTypeID() != Type::DoubleTyID) ){

……

}

}

**赋值运算符两边的操作数的类型是否匹配**

此处我们定义一个trans函数，用于提供某个表达式的值向另一类型的转换指令。若两种类型不存在转换关系时，则输出错误信息并返回源类型的值。而能否转换则通过一个事先定义的typeTable来判断，此处我们允许类型转换如下：int到float，int到double， float到double，float到int，double到int，bool到int：

context.typeCheck.trans(exp, context.typeTable.getVarType(dstTypeStr), context.currentBlock());

**第四章 中间代码LLVM IR生成**

**LLVM IR 基本语法介绍**

LLVM 汇编语言中的注解以分号“；”开始，并持续到行末。

全局标识符要以“@”字符开始。所有的函数名和全局变量都必须以“@”开始。

LLVM 中的局部标识符以百分号“%”开始。

LLVM 拥有一个强大的类型系统，这也是它的一大特性。LLVM将整数类型定义为 iN，其中N是整数占用的字节数。我们可以指定1到223- 1之间的任意位宽度。

对于字符串 "Hello World!"，可以使用类型[13 x i8]，假设每个字符占用1个字节，再加上为NULL字符提供的1个额外字节。

我们可以对“hello-world”字符串的全局字符串常量进行如下声明：@hello = constant [13 x i8] c"Hello World!\00"。使用关键字constant来声明后面紧跟类型和值的常量。以 c 开始，后面紧跟放在双引号中的整个字符串（其中包括 \0 并以0结尾）。

LLVM 允许声明和定义函数。以关键字define开始，后面紧跟返回类型，然后是函数名。返回32字节整数的main的简单定义类似于：define i32 @main() { ; some LLVM assembly code that returns i32 }。

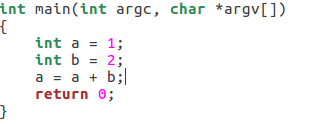
函数声明。以puts方法为例，它是printf: declare i32 puts(i8\*) 的LLVM等同物。该声明以关键字declare开始，后面紧跟着返回类型、函数名，以及该函数的可选参数列表。该声明必须是全局范围的。

每个函数均以返回语句结尾。有两种形式的返回语句：“ret <type> <value> ”或 “ret void”。

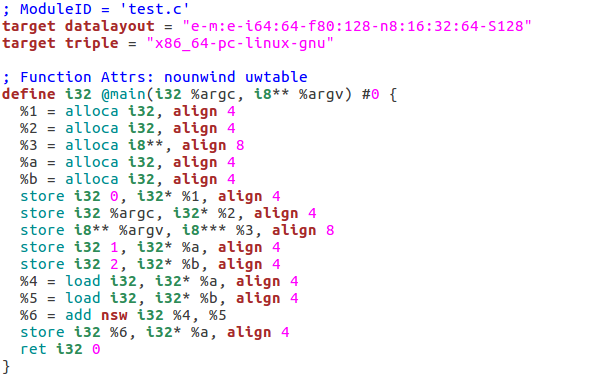
LLVM使用“call <function return type> <function name> <optional function arguments>”来调用函数。每个函数参数都必须放在其类型的前面。返回一个6位的整数并接受一个36位的整数的函数测试的语法为“call i6 @test( i36 %arg1 )”。

**LLVM IR语法说明样例**

源代码：



对应的LLVM IR：



生成方式

从AST到LLVM IR意味着将每一个语义节点转换成等价的LLVM IR指令。LLVM将帮助我们把这步变得非常简单，因为LLVM将真实的指令抽象成类似AST的指令。这意味着我们真正要做的事就是将AST转换成抽象指令。这个过程可以想象成是从我们之前已经定义的抽象语法树的根节点开始遍历每一个树上节点并产生字节码的过程。在开始生成LLVM IR之前，还有一些准备工作要做。首先，给每个AST类添加一个虚函数codeGen，用于实现具体的代码生成：

class NBasic {

// 各个抽象语法树节点基类

public:

virtual ~Node() {}

virtual Value \*codeGen() = 0;

};

class NIdentifier : public NExpression {

public:

std::string name;

NIdentifier(const std::string& name) : name(name) { }

virtual llvm::Value\* codeGen(CodeGenContext& context);

};

每种AST节点的codeGen()方法负责生成该类型AST节点的IR代码及其他必要信息，生成的内容以LLVM Value对象的形式返回。其中LLVM用“Value”类表示“静态一次性赋值（SSA，Static Single Assignment）寄存器”或“SSA值”。SSA值最为突出的特点就在于“固定不变”：SSA值经由对应指令运算得出后便固定下来，直到该指令再次执行之前都不可修改。

其次，我们还需要一个“Error”方法，该方法与语法解析器里用到的报错函数类似，用于报告代码生成过程中发生的错误（例如引用了未经声明的参数）：

Value \*ErrorV(const char \*Str) { Error(Str); return 0; }

下面几个静态变量都是用于完成代码生成的：

static Module \*TheModule;

static IRBuilder<> Builder(getGlobalContext());

static std::map<std::string, Value\*> NamedValues;

其中TheModule是LLVM中用于存放代码段中所有函数和全局变量的结构。从某种意义上讲，可以把它当作LLVM IR代码的顶层容器。

Builder是用于简化LLVM指令生成的辅助对象。IRBuilder类模板的实例可用于跟踪当前插入指令的位置，同时还带有用于生成新指令的方法。

NamedValues映射表用于记录定义于当前作用域内的变量及与之相对应的LLVM表示即代码的符号表。

我们还在CodeGenContext类中使用一个语句块的栈来保存最后进入的block(因为语句都被增加到blocks中)我们同样用个堆栈来保存每组语句块中的符号表。

我们设计的语言知道当前范围内的内容，要支持“全局”上下的做法，必须向上搜索整个堆栈中每一个语句块，直到最后发现所要匹配的符号。在我们进入一个语句块之前，我们需要将语句块压栈，离开语句块时将语句块出栈。

**具体实现**

下面介绍每种AST节点的codeGen()的具体实现方法，这里我们以几个具有代表性的节点对它们的codeGen()展开详细的介绍。

**数值常量**

LLVM IR中的数值常量是由ConstantFP类表示的。在其内部，具体数值由APFloat（Arbitrary Precision Float，可用于存储任意精度的浮点数常量）表示。这段代码简单来说就是新建并返回了一个ConstantFP对象。值得注意的是，在LLVM IR内部，常量都只有一份，并且是共享的。下面以double为例：

Value\* NDouble::codeGen(CodeGenContext& context)

{

return ConstantFP::get(Type::getDoubleTy(context.llvmContext), this->value);

}

**变量**

在LLVM中引用变量也很简单。实际上，位于NamedValues映射表中的变量只可能是函数的调用参数。这段代码首先确认给定的变量名是否存在于符号表中，如果不存在，就说明引用了未定义的变量，然后返回该变量的值。

Value \*NVar::Codegen(CodeGenContext& context) {

Value \*V = NamedValues[Name];

return V ? V : ErrorV("Unknown variable name");

}

**二元运算**

二元运算的中间代码生成的基本思想是递归地生成代码，先处理表达式的左侧，再处理表达式的右侧，最后计算整个二元表达式的值。上述代码就opcode的取值用了一个简单的switch语句，从而为各种二元运算符创建出相应的LLVM指令。

利用LLVM的Builder类我们需想清楚该用哪些操作数（即此处的L和R）生成哪条指令（通过调用CreateFAdd等方法）即可，至于新指令该插入到什么位置，交给IRBuilder就可以了。

在这里，LLVM指令需要遵循严格的约束：例如，add指令的L、R必须同属一个类型，结果的类型则必须与操作数的类型相匹配。

**函数定义**

Value\* NFunDecl::codeGen(CodeGenContext& context)

{

vector<const type\*> argTypes;

VariableList::const\_iterator it;

for (it = arguments.begin(); it != arguments.end(); it++) {

argTypes.push\_back(typeOf((\*\*it).type));

}

FunctionType \*ftype = FunctionType::get(typeOf(type), argTypes, false);

Function \*function = Function::Create(ftype, GlobalValue::InternalLinkage, id.name.c\_str(), context.module);

BasicBlock \*bblock = BasicBlock::Create("entry", function, 0);

context.pushBlock(bblock);

for (it = arguments.begin(); it != arguments.end(); it++) {

(\*\*it).codeGen(context);

}

block.codeGen(context);

ReturnInst::Create(bblock);

context.popBlock();

std::cout << "Creating function: " << id.name << endl;

return function;

}

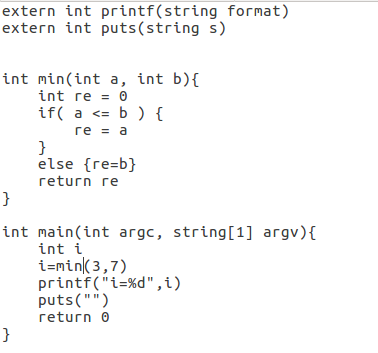
首先需要该函数的返回值类型是“Function\*”而不是“Value\*”。“函数原型”描述的是函数的对外接口（而不是某表达式计算出的值），返回代码生成过程中与之相对应的LLVM Function自然也合情合理。

FunctionType::get调用用于为给定的函数原型创建对应的FunctionType对象。随后，FunctionType::get方法以这“N”个double为参数类型、以单个double为返回值类型，创建出一个参数个数不可变（即最后一个参数false）的函数类型。

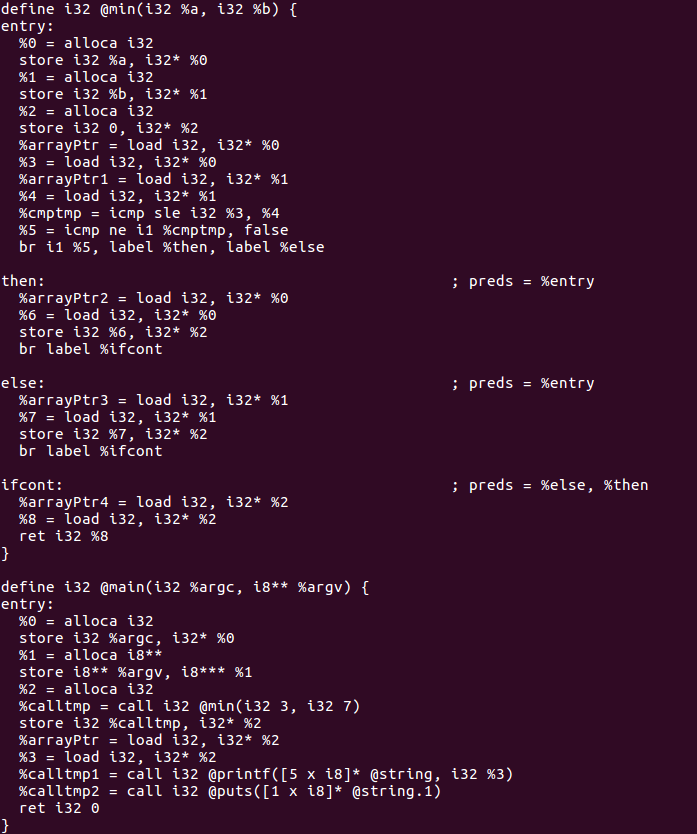
最后实际上创建的是与这个函数原型相对应的函数。其中包含了类型、链接方式和函数名等信息，还指定了该函数待插入的模块。“ExternalLinkage”表示该函数可能定义于当前模块之外，可以被当前模块之外的函数调用。。

**中间代码生成测试**

测试样例：



LLVM IR：



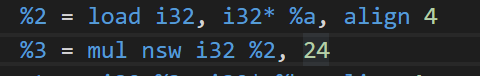
**第五章 中间代码优化**

优化是编译器的一个重要组成部分，由于编译器将源程序翻译成中间代码的工作是机械的、按固定模式进行的，因此，生成的中间代码往往在时间和空间上有很大浪费。当需要生成高效目标代码时，就必须进行优化。

**乘法优化**

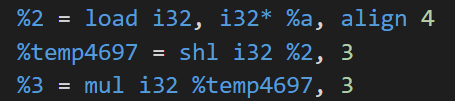
第一个是乘法的优化，在常用的体系架构中，无论是x86还是MIPS,都有单独的乘法单元用于计算乘法操作，此单元效率极低，且相乘的两个数越大效率越低，因此做的第一个优化是针对乘法。

如下图所示,当乘法为reg=reg\*const时。

、

把乘法拆解成乘法和移位两步操作，移位相较于乘法，效率极高，例如上图，原式含义为：

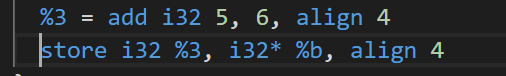
b=a\*24。拆解转化为a<<3，b=a\*3。如下图;



这样一来这条指令就能更有效的执行。

**常量运算优化**

形如下例的指令：

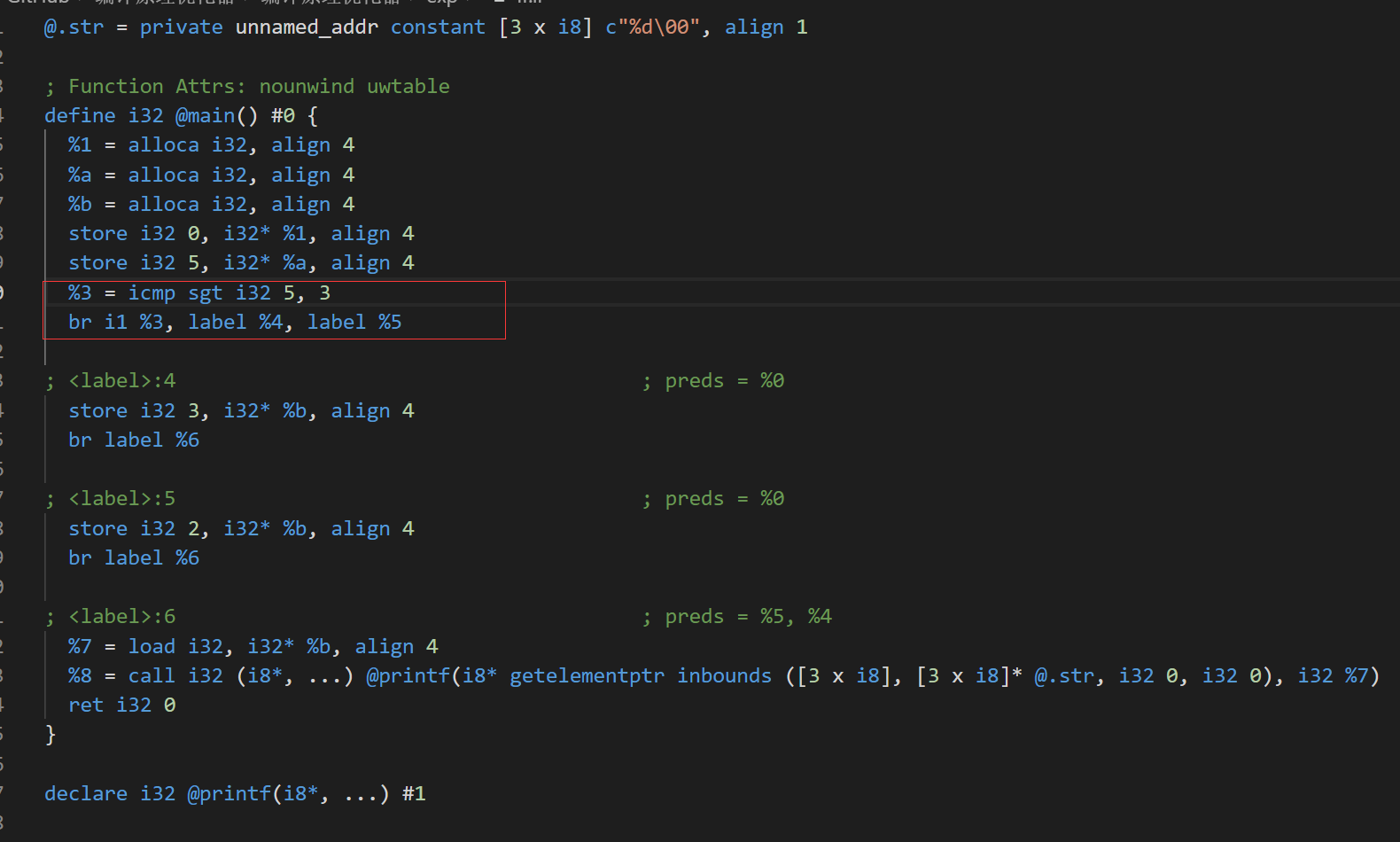


即b=5+6，是可以在中间指令层进行优化的，无论是加法还是乘法，只要运算的两个加数都是const，且紧跟的一条指令是store，则可以把两条指令合为一条：

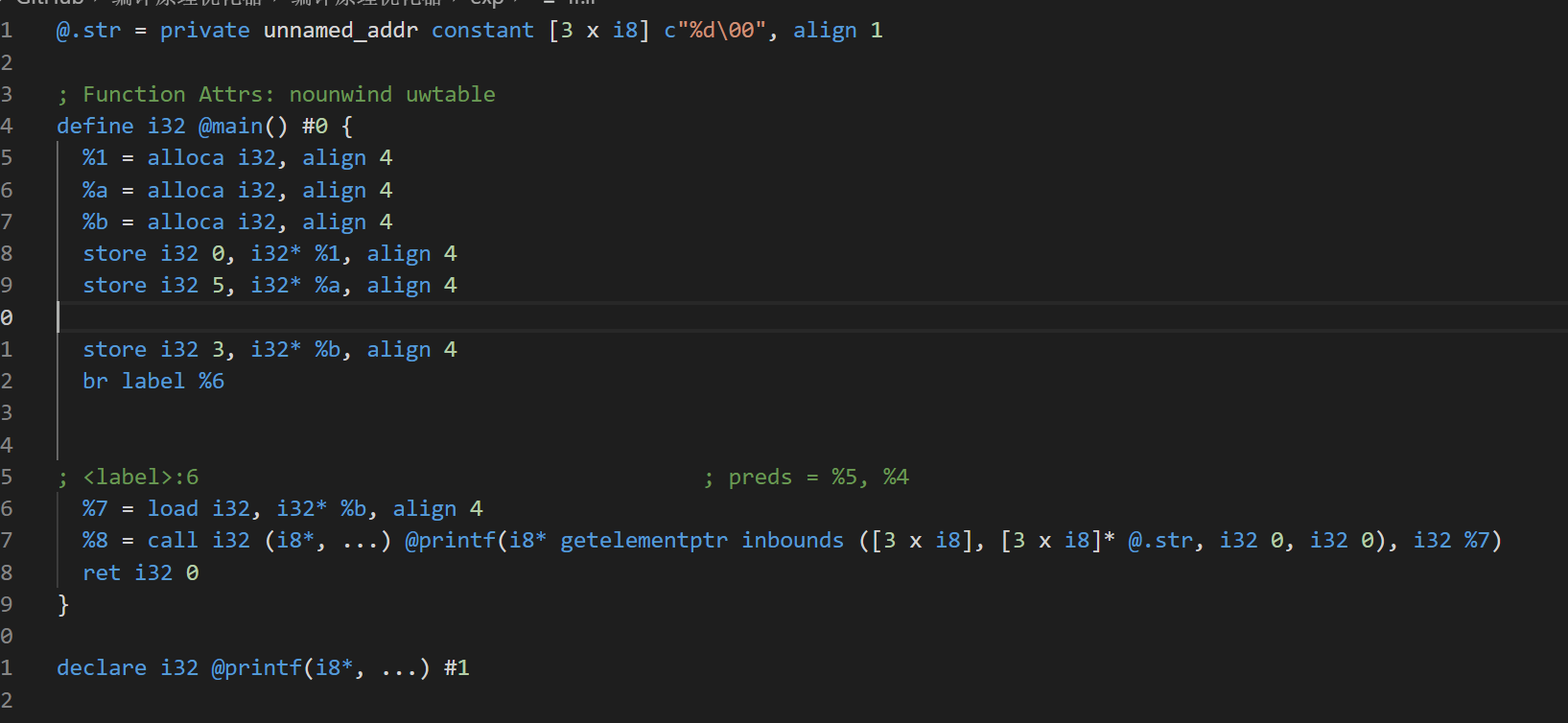


**跳转指令优化**

形如如下的跳转指令：

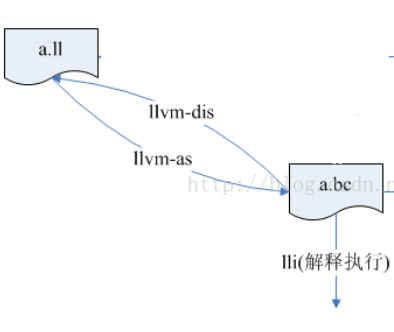


当icmp指令比较的是两个const，且之后紧跟着br指令时，因为指令必定跳转，所以删除部分永远不会访问的指令。优化后的中间代码如下：



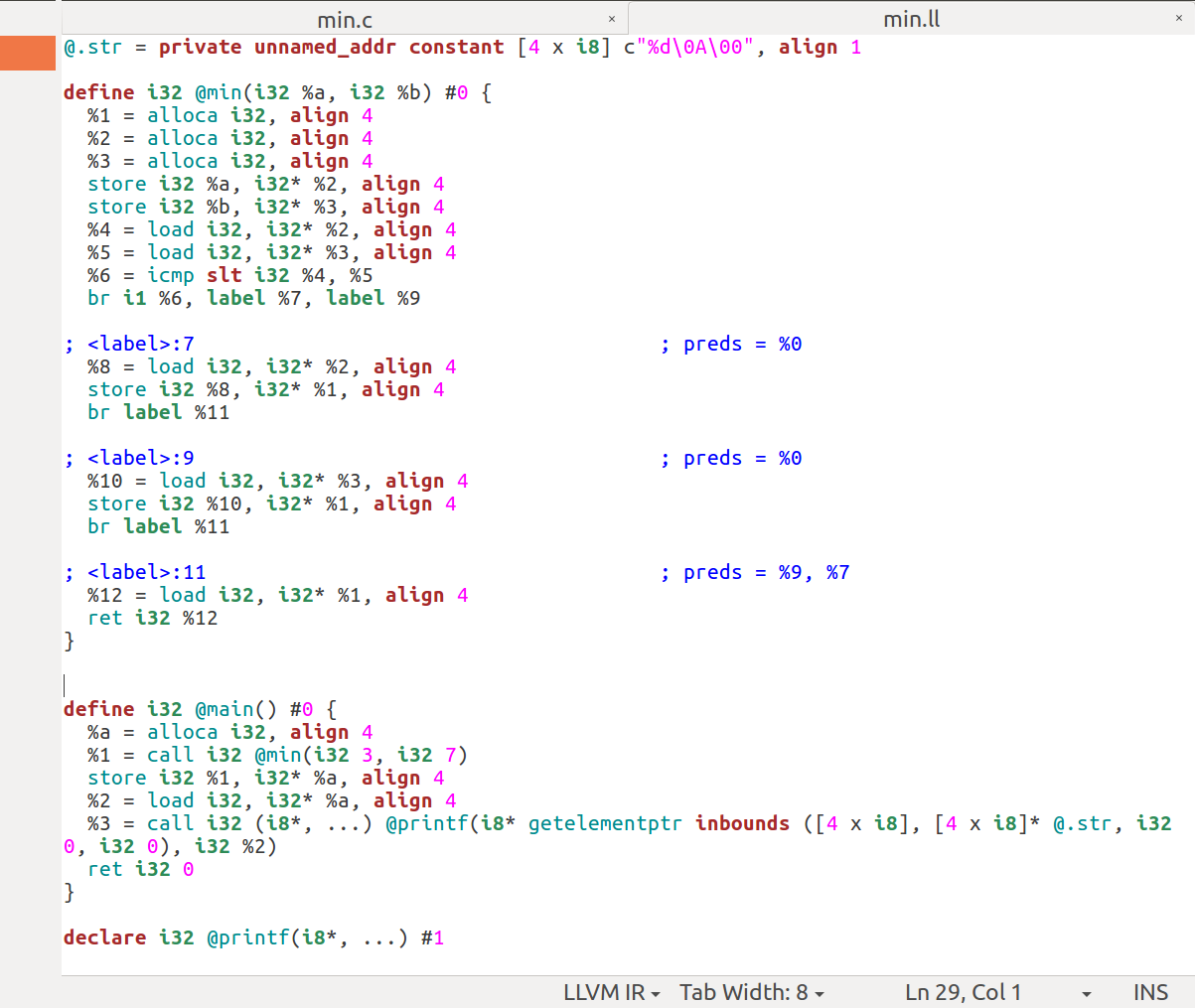
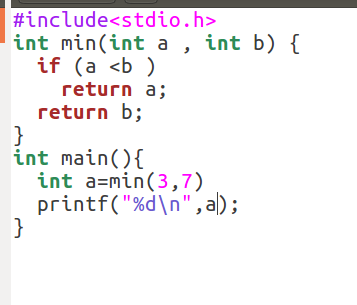
**第六章 目标代码生成**

目标代码生成使用llvm编译工具实现，llvm中各种文件的转化指令如下图：

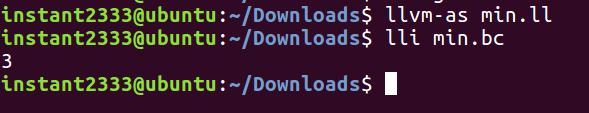


.ll文件是llvm IR格式，.bc是二进制格式的目标代码，可被执行。

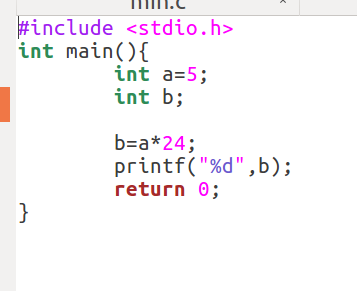
示例源码和编译后的中间代码如下，实现了一个比较整形数大小的函数：

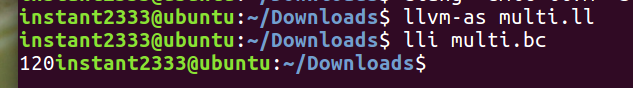


通过命令行调用llvm工具实现到目标代码的转化并运行目标代码：



再看一个实例，源码及运行结果如下：





可以看到命令行正确的输出了程序运行的结果120。