**湖南科技大学 计算机科学与工程学院**

**编译原理**

**实验书**



杨策

2025. 02. 20

# 第一章 实验概览

## 1.1 项目背景

本项目是编译原理课程的配置实验。在这个实验中，我们将基于LLVM IR和工具链，逐步完成一个自己的编译器，支持对SysY（C语言的一个子集）进行编译。通过这个实验，我们将在C++中使用第三方库，实践编译、链接等具体概念，并实现一个编译器的前端。

## 1.2 LLVM介绍

LLVM的官网如下：

https://llvm.org/

本实验中所有用到的演示代码在LLVM 14版本下测试通过。由于 LLVM不保证不同版本间的兼容性，读者如果使用了其他版本，应该查阅对应的文档，以使用正确的接口。

LLVM定义了一套中间代码LLVM IR， 并且有一系列工具，可以完成LLVM IR的优化、链接和目标代码生成。

参考资料：

<https://evian-zhang.github.io/llvm-ir-tutorial/>

这是一个很好的LLVM参考文档。

<https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html>

这是LLVM官方的教程，学有余力的同学可以自学这一教程，完成全部的内容。

SysY语言实际上是大学生编译器比赛的内容，但比赛不允许使用LLVM，因此复杂度会高很多，在完成实验后，学有余力的同学可考虑进一步完善并参加比赛。大赛官网：

<https://compiler.educg.net/#/index?TYPE=COM>

## 1.3 技术选型、软件架构

我们的项目从源代码开始，经过整个编译过程，生成出可执行的目标代码。目标代码生成部分由于与机器相关，过程繁琐、工作量大，作为课程实验内容可能难度过大。因此，我们直接使用LLVM后端，只需要自己完成词法分析、语法分析、语义分析和中间代码生成，其中，中间代码生成部分应该生成符合LLVM规范的中间代码，即LLVM IR。

为了生成LLVM IR，有两种不同的方法，一种是调用LLVM提供的工具库进行生成，一种是在自己理解LLVM IR的语法规则后，完全通过手写代码来进行生成。为尽可能简化开发过程，本教程使用LLVM提供的工具库，而对应的接口为C++接口，因此这部分使用C++来完成。实际的软件开发过程中，也会大量复用已有的第三方库或者是公司内部的工具库。学有余力的同学可尝试根据LLVM IR的规范自己生成IR代码。

在进行软件开发的过程中，使用第三方库是很常见的情况。一个大型软件的开发会涉及到多个不同程序、代码之间的协作。按层次上，可以分为以下几种：进程间通信；进程内模块间；模块内。

进程间通信是耦合非常低的协作方式。比如本文的编译器，实际就是由我们手写的前端与llvm后端，通过进程间通信，实现了完整的编译过程，我们前端与后端之间，使用文件传输数据。这种方式的好处是，不同进程只需要定义好接口（API），即可完全独立开发，互不干涉。比如LLVM后端是由c++开发 ，我们的前端可以用c++，也可以用别的方式，比如rust，python。

同一个进程内的协作开发，我们可以进一步细分为模块间和模块内两种。模块内的协作是最简单，也是所有开发者最先接触到的方式。比如本文的词法分析器和语法分析器，可以实现为两个函数，在主函数中调用这两个函数来分别实现对应功能。这些实现可以放在一个源文件内，也可以放在不同源文件内，不同语言通常有不同的惯例。在编译时，编译器使用多个源文件，编译出一个可执行文件。如果每次有代码修改，都需要重新编译。

在我们的编译器中，我们需要依赖LLVM的工具库，这些工具库以源文件的形式给出，因此我们可以将对应的源文件加入我们的工程，在开发时调用对应的库函数，完成整个程序。但LLVM的源代码非常庞大，一次编译可能需要一个小时，我们自己的前端代码量很小，我们是不希望每次修改一行代码都等一小时编译过程的。因此我们希望能将LLVM工具库编译成某种可以直接调用的方式，在我们的程序中直接使用预编译好的文件，在c语言/c++中，预编译好的文件是二进制文件，直接调用的方式被称为链接。

但使用C++的库进行链接时，仍然会有一个问题。C++语法支持多个不同的函数采用同样的名称，如overload，在链接时为了区分，会生成不同的名字，称为name mangling，而C++标准没有对名字生成的规则做出规定，不同的编译器可以灵活选择，因此不同编译器对同一份代码生成出的二进制文件会不一致，从而无法相互链接。解决方法之一是采用extern C等方法，强制按C语言的方式导出接口，但这样会无法支持C++的一些特性。在我们的程序中，编译LLVM和编译我们的编译项目采用同一个编译器就可以解决这一问题，比如都用VS或者clang++进行编译。

总结一下，程序的整体架构如图：



图1 程序整体架构

中间代码生成部分如前所述，我们将通过调用LLVM工具库来实现，后文大部分的内容是介绍LLVM的这一API如何使用。词法分析和语法分析部分可以手写或者使用自动化工具，如lex、yacc。 Lex和yacc有C++版本，即flex和bison，也有对应Python版本[[1]](#footnote-1)，方便进行开发调试，读者可自行选择适合自己的方式。

## 1.4 实验要求

我们的最终目标是实现一个SysY语言的自定义语言编译器，支持的功能描述如下：

1. 两种基本类型，int和float。
2. 支持变量定义与四则运算。
3. 支持函数定义与函数调用，函数支持无参数与有参数函数。
4. 支持循环语句与分支语句。
5. 支持数组的定义与使用。
6. 库函数可以精简为一个函数print，完成输出功能。（可选）
7. 有良好的工程实现，如日志、错误提示等。

检查需要的材料：

1. 源代码。源代码请在gitee（<https://gitee.com/>）上创建一个私有仓库，并将老师添加为仓库成员。
2. 编译好的可执行文件。
3. 可供验证的测试案例。

# 第二章 环境配置

## 2.1 概论

本章以Windows + Visual Studio为例，介绍如何编译LLVM源代码并在工程中配置对应的LLVM库。如果是使用其他开发工具，请自行查找如何进行配置。在Linux下，虽然一般通过命令行完成开发过程，但进行第三方库的安装和配置会更容易，原因是Linux的发行版本通常已经将很多库预编译好，可以直接进行安装，比如Ubuntu下可以使用apt工具完成。现在Win11已经带有WSL2工具，在Windows下用Linux开发已经很方便了，读者可以自行尝试。

## 2.1 LLVM的安装与使用

LLVM 的使用分成两种情况，一种是使用Clang+LLVM作为编译器，可以直接下载可执行文件并安装。下载地址：

<https://releases.llvm.org/>

Clang不仅是LLVM的前端，同时还整合了后端工具链的功能，进行安装可以方便进行调试（也可在后文的编译中从源代码编译一份Clang）。在安装完成后，打开命令行窗口（Win11下默认为Powershell），运行命令可以输出一些信息：

> clang --version

clang version 17.0.1

Target: x86\_64-pc-windows-msvc

Thread model: posix

InstalledDir: C:\Program Files\LLVM\bin

这里我们发现，clang的版本不是14而是17，因为我在本地安装的可执行文件版本和源代码版本确实不一致，由于LLVM IR相对而言比较稳定，因此只运行clang.exe一般不会遇到兼容性问题。确保正确运行后，可以编译一份Hello World程序，进行测试。假设源代码在hello.c文件中，编译和运行的指令如下：

> clang hello.c

> .\a.exe

Hello World!

clang通过文件扩展名来识别文件类型，使用.c作为扩展名时，源代码将作为c代码被编译。

由于后文需要用到LLVM IR，我们还需要两个功能，一个功能是将c语言源代码编译成LLVM IR的功能。当我们有部分功能觉得难以实现时，可以参照clang输出的IR来实现，另一个功能是将LLVM IR编译成目标文件的功能。

我们通过一个简单程序sum.c来测试这两个功能：

int sum(int i1, int i2) {

return i1+i2;

}

int main() {

sum(1, 2);

return 0;

}

LLVM IR分为二进制和文本两种形式，使用下列命令可以生成文本形式的LLVM IR：

> clang -emit-llvm -S -c sum.c

生成后得到sum.ll文件（和源文件名有关），在其中应该有类似的IR代码：

define dso\_local i32 @main() #0 {

%1 = alloca i32, align 4

store i32 0, ptr %1, align 4

%2 = call i32 @sum(i32 noundef 1, i32 noundef 2)

ret i32 0

}

ll文件可以通过如下命令继续编译成可执行文件：

clang sum.ll

clang在编译时，如果输入文件为ll扩展名，则自动识别为LLVM IR文件，默认输出为a.exe，但由于这个程序没有输出，因此执行后无输出，是正常的，可自行按这一流程将Hello World程序转换为IR，并将IR转换为可执行文件，验证运行是否成功。

## 2.2 LLVM的编译

除了直接运行clang完成编译之外，有时我们需要基于LLVM进行开发，如利用LLVM IRBuilder生成LLVM IR（本实验内容），或者在LLVM IR上运行某些Pass，此时需要使用LLVM的库文件。

在Windows下进行开发时，需要下载源代码并编译之后进行配置，如果开发工具使用Visual Studio，包括IDE以及对应的编译工具，官方教程如下：

https://llvm.org/docs/GettingStartedVS.html

我在本机上使用Visual Studio 2022 Commuity完成了LLVM 14.0.0版本的编译和使用，具体过程如下：

1，安装Python 3

2，安装git

3，安装CMake

之后打开Visual Studio中带有编译环境的Command Prompt，进入llvm目录下，依次执行：

mkdir build

cd build

cmake ../llvm

cmake --build .

编译耗时较长，请耐心等待。

编译得到的比较重要的文件有几部分：

头文件。如llvm-config.h，包含了一些编译选项，会在编译时生成出来，一般在build目录下的include目录下。

可执行文件。如llvm-as.exe，这是llvm的汇编器，负责将LLVM IR转换成llvm bitcode。LLVM由工具链组成，包含多种不同的工具，这一工具是其中之一。一般在类似于Debug/bin的目录下。

静态链接库文件。如LLVMCore.lib。静态链接是预编译好的源代码，在链接时对应二进制代码被加入到目标程序，运行时不需要库文件。一般在类似于Debug/lib的目录下。

上述文件如果找不到，请检查自己的配置，或者在LLVM目录下直接使用文件查找功能进行查找。

## 2.3 Hello, LLVM

当编译完成后，可以尝试使用LLVM的库进行开发。

由于CMake使用VS进行编译，会创建出VS的解决方案和工程，可以直接打开对应解决方案，新建工程并进行设置，也可以新建解决方案，在工程中设置额外的头文件和库文件，在此介绍后者。

VS创建解决方案和工程之后，需要对头文件和库文件进行设置。可自行根据网上教程进行配置，配置完成后可以运行下列源代码进行测试：

#include <llvm/IR/LLVMContext.h>

#include <llvm/IR/IRBuilder.h>

#include <llvm/IR/Module.h>

#include <llvm/IR/Type.h>

#include <llvm/Support/FileSystem.h>

using namespace llvm;

int main()

{

llvm::LLVMContext context;

llvm::Module module("IRbuilderTest", context);

llvm::IRBuilder<> builder(context);

Type\* ret\_type = Type::getInt32Ty(context);

FunctionType\* FT = FunctionType::get(ret\_type, {}, false);

Function\* F = Function::Create(FT, Function::ExternalLinkage, "main", module);

BasicBlock\* bb = BasicBlock::Create(context, "entry", F);

builder.SetInsertPoint(bb);

llvm::FunctionType\* func\_type = llvm::FunctionType::get(  
 llvm::Type::getInt32Ty(context), { llvm::Type::getInt8PtrTy(context) }, true);

Function\* printf = llvm::Function::Create(func\_type, llvm::GlobalValue::ExternalLinkage, "printf", module);

std::vector<Value\*> printf\_args = { builder.CreateGlobalStringPtr("Hello world!\n") };

builder.CreateCall(printf, printf\_args);

Value\* ret\_value = ConstantInt::get(context, APInt(32, 0));

builder.CreateRet(ret\_value);

module.dump();

return 0;

}

这一段略显复杂的代码将输出一段打印Hello world!的LLVM IR。由于使用了大括号初始化std::vector的方式，需要在编译时配置C++11以上的编译支持，推荐使用C++14。

这里LLVMContext、Module、IRBuilder采用了局部变量，在函数结束时自动释放内存，而官方教程中给出的方式是采用unique\_ptr，同样是自动释放。在开发时，我尝试过将Module定义为普通指针，但delete时会报错，不太清楚具体原因，有清楚的读者欢迎交流。

## 2.4 编译项目的配置，LLVM库的链接

如果对编译和链接过程不太熟悉，可参照如下过程进行配置。

首先我们需要配置包含目录，即头文件目录。比如，我们的源文件中使用了LLVM中定义的类llvm::Value，这是LLVM中最基本的一种类，用来表示各种值，是大部分类的基类。为了使用这个类以及对应的成员函数，我们需要知道这个类的定义，一般通过引用这个类所在库的头文件来实现，查阅LLVM的官方文档后，发现其定义在llvm/IR/Value.h中。但直接在源文件中写#include "llvm/IR/Value.h"会出错，原因是在编译时，编译器会在系统目录和当前解决方案设置的源文件目录下进行搜索，但LLVM的源文件在我们设置的目录下，因此编译器找不到，会产生报错信息。

为了解决这个错误，需要把llvm的头文件目录添加进工程设置中。右击工程名称，弹出菜单中选择属性，进入C/C++项，可以设置附加包含目录，可通过下拉菜单的编辑来选择多个目录添加。LLVM需要添加两个目录，一个是LLVM项目源文件中llvm\_src/llvm/include目录，可以在这个目录下找到llvm/IR/Value.h以及其他多个头文件。另一个是LLVM项目编译后生成的源文件，在我的机器上是llvm\_src/build/include目录，在这个目录下可以找到llvm/Config/config.h以及其他多个头文件。将这两个目录加入后，编译器应该可以自动完成语法解析，在编译器中即不再报错。

然后我们需要配置链接时的库文件。上述代码在不报错后，我们右击工程进行生成，链接过程中会有错误，报LLVM相关的一些接口找不到对应的实现。这与C/C++的具体编译方式有关。C++在编译阶段，使用#include指令引入的头文件会进行全文替换。如上面所述的Value类，其定义会加入到我们自己编写的源文件中。C++中定义类时，只需要对所有的成员变量和成员函数进行声明，成员函数的定义可以放在另外的源文件中，如Value类的部分实现在llvm\_src\llvm\lib\IR\Value.cpp中。由于缺乏这些具体的函数定义代码，编译是无法完成的。

幸运的是，我们在编译LLVM的时候，已经将对应的源代码编译成了库文件，因此我们的项目中无须把对应源文件加入工程并重新编译，而是可以直接链接已有的库文件。由于前文所述的name mangling问题，我们开发时，链接的文件与当前工程需要使用同样的编译器生成。

首先，需要在**工程属性->链接器**中，设置**附加库目录**。需要添加编译好的LLVM二进制文件目录，在类似于llvm\_build\Debug\lib\目录下，应该有多个.lib文件，如LLVMCore.lib，从名字可以看出这是LLVM核心库，具体目录和编译时的配置有关。之后需要在**工程属性->链接器->输入**中，在附加依赖项中添加需要的lib文件。经测试，本文使用的LLVM lib包括如下文件：

LLVMCore.lib

LLVMRemarks.lib

LLVMBinaryFormat.lib

LLVMSupport.lib

LLVMBitstreamReader.lib

LLVMTransformUtils.lib

LLVMAnalysis.lib

LLVMProfileData.lib

LLVMDemangle.lib

这些库文件添加后，完成测试代码的编译生成应该是没有问题的，如何仍然报错找不到请耐心检查各配置，比如工程和LLVM lib编译时的目标平台是否一致。如果LLVM编译时选择了x86，当前工程选择的目标平台是x64，两者不匹配，也无法正常完成链接。

## 2.5 日志库的使用

任何大型软件中都存在bug，日志是极为有用的debug工具。写入日志的操作一般需要进行IO调用，而IO调用相比CPU运算是极为缓慢的，因此日志通常是异步写入的。异步IO不是本课程的内容，我们也通过调用第三方库来实现。请自选一个日志库，使用前面学到的知识，加入到本项目中。一些日志库的选择可以参考下面的文档：

https://blog.csdn.net/gatieme/article/details/50603682

C++库中比较特殊的一类是仅有头文件的库，不需要进行链接的配置，使用起来比较简单。在后面的开发过程中，调试信息可以调用日志库输出。请在测试程序中加入日志输出，在程序开始运行和完成运行时分别加上一条日志输出。

## 2.6 总结

完成这一节的内容后，应该能编译出LLVM，并建立自己的项目，成功编译并运行测试代码。

后续的内容，如果是学有余力的同学可以参照LLVM官方教程中的Kaleidoscope来完成，本文的内容也是基于这一教程的。目前IT行业大部分资料都是英文的，掌握英文资料的阅读能力也是开发能力的一部分。英语不太熟练可尝试借助AI来完成翻译过程。对官方教程不能很好掌握但仍然想尝试的同学，也可以按照本教程继续一步一步完成。

# 第三章 最简单的程序

## 3.1 概览

这一章中我们的编译程序支持对最简单的程序进行编译，如下：

int main() {

print(1);

return 0;

}

为了完成这一过程，我们需要完成词法分析、语法分析、中间代码生成的过程。虽然这一章只支持最简单的程序，但为了让程序能运行我们会支持许多不同的功能，因此本章的内容是比较多的。

## 3.2 词法分析与语法分析

（1）词法分析

词法上，需要支持标识符、关键字、整数和一些符号的识别。

标识符我们定义成字母开头，后续任何字母和数字组成的串。

关键字有两个，分别是int 和 return，一个是表示类型，一个是表示return语句。

print则不识别为关键字，而是定义为内部函数，在函数调用部分进行特殊处理，当然也可定义为关键字。

特殊符号包括{}();。

（2）语法分析

本章所需要的完整语法规则如下：

program -> func\_decl\_list;

func\_decl\_list -> func\_decl\_list func\_decl | func\_decl

func\_decl -> type\_spec ID ( ) compound\_stmt

type\_spec -> INT

compound\_stmt -> { stmt\_list }

stmt\_list -> stmt\_list stmt | stmt

stmt -> call\_stmt | return\_stmt

call\_stmt -> ID ( args ) ;

args -> args, expr | expr | empty

arg -> expr

return\_stmt -> RETURN expr ;

expr -> NUM

可以看到，虽然这个程序是最简单的程序，但是需要处理的语法规则仍然是比较多的。一个程序由多个函数的定义组成。函数定义则由返回类型、函数名、函数体三部分组成，我们的函数在定义时暂时不支持参数类型。函数体是复合语句，即由大括号包围的多条语句。语句目前只支持两种，一种是函数调用，一种是返回语句。函数调用的参数以及返回的值可以传入表达式，目前表达式只支持数值。当解析完成之后，应该能生成出语法树，并进行调试输出。

词法分析推荐使用自动化工具进行生成。词法分析的规则比较固定，自动化工具可以处理大部分场景，因此推荐使用自动化工具进行生成。而语法规则相对来说要复杂得多，而且大部分类似于自然语言的语法均不能用较简单的方法直接处理，因此一般有两种选择，一种是使用手写的递归下降语法分析器，一种是使用自动化工具来进行生成，但使用自动化工具也需要较大的工作量，可自行选择自己喜欢的方式。

## 3.3中间代码生成

中间代码生成时，需要根据不同的表达式来进行不同的处理。可以通过遍历语法树来完成中间代码生成。对于每一个结点，在遍历过程中都需要完成对应的操作，具体来说：需要生成对应的中间代码；如果涉及到表达式求值，需要返回对应的值。

对应的伪代码如下：

def IRGen(Node node):

some code

# traverse subtree

child\_values = [ IRGen(child) for child in node.children]

v = cal(child\_values)

some code

return v

在实际的代码中，在遍历子树前后都有可能生成部分IR。我们可以认为返回的值都是LLVM中的值，即Value，因此这一接口可以定义为返回Value \*。对于无返回值的情况，如整个语句或者函数，为方便，可以直接返回nullptr。

对数值字面值NUM，应该返回一个常量，如需要得到123对应的值，在LLVM中的方法为：

Value\* value = ConstantInt::get(context, APInt(32, 123));

LLVM中，int类型有一系列，按长度区分，这里APInt的参数32表示对应的int类型的长度，即返回一个int32类型的值123。最终取得的值赋给value。

返回语句的处理相对简单，在LLVM中可以这样实现：

Builder.CreateRet(value);

调用这一语句之后，LLVM会在当前的Module中自动生成出一条IR的ret指令，操作数就是传入的value变量，是Value \*类型。后文中默认value是Value \*类型。这条IR指令插入的具体位置由IRBuilder内部维护的InsertPoint来决定。

LLVM IR的一个文件称为一个Module，其中有多个Function，而一个Function由多个BasicBlock组成，BasicBlock就是本课程中介绍的基本块（可能会到优化部分才介绍，比较靠后，但这一概念本身很简单，可以自行查阅了解），而InsertPoint设置在某个BasicBlock上，当生成新的IR时，会在对应BasicBlock的最后插入。

通过下列语句可以修改InsertPoint：

Builder.SetInsertPoint(basic\_block);

创建一个函数首先需要定义函数的类型。函数的类型由返回类型以及参数类型组成。LLVM中的IR对象类型为Type，取得int32类型可以使用如下代码：

Type \*int32\_type = llvm::Type::getInt32Ty(context);

参数类型使用std::vector<Type \*>，因此可以手动创建一个vector传入类型，而我们的main函数不需要参数，因此可以初始化一个空vector传入。

FunctionType \*function\_type = FunctionType::get(

llvm::Type::getInt32Ty(context), {}, false);

Function \*function = Function::Create(

function\_type, Function::ExternalLinkage, “main”, module);

BasicBlock \*basic\_block = BasicBlock::Create(

context, “entry”, function);

Builder.SetInsertPoint(basic\_block);

Function::Create的参数中，第3个参数main为函数名，在创建之后，可以通过module->getFunction再取回这个函数。为了完成我们自己的print函数，我们通过调用c语言标准库的printf函数来实现打印到屏幕。经过测试，直接声明一个与标准printf函数相同的函数，在链接时llvm会帮我们链接到c语言标准库函数。printf函数返回值为int，参数为char \*，同时接收可变长参数，代码如下：

const char \*func\_name = "printf";

llvm::FunctionType \*func\_type = llvm::FunctionType::get(  
llvm::Type::getInt32Ty(module->getContext()),  
{llvm::Type::getInt8PtrTy(module->getContext())}, true);

llvm::Function \*printf\_ptr = llvm::Function::Create(  
func\_type, llvm::GlobalValue::ExternalLinkage, func\_name, module);

在调用时，printf的第二个参数是要打印的数值，为llvm::Value，常数的创建在之前已经讨论过，而第一个参数是格式字符串，可以通过Builder.CreateGlobalStringPtr来创建，之后用CreateCall进行函数调用，代码如下：

std::string format\_string = "%d\n";

vector<Value \*> printf\_args;

printf\_args.push\_back(Builder.CreateGlobalStringPtr(format\_string));

printf\_args.push\_back(value);

Builder.CreateCall(printf\_ptr, printf\_args);

最后我们基于上一章的代码，得到下面的完整代码：

llvm::LLVMContext context;

llvm::IRBuilder<> builder(context);

llvm::Module module("IRbuilderTest", context);

Type\* ret\_type = Type::getInt32Ty(context);

FunctionType\* FT = FunctionType::get(ret\_type, {}, false);

Function\* F = Function::Create(FT, Function::ExternalLinkage, "main", module);

BasicBlock\* bb = BasicBlock::Create(context, "entry", F);

builder.SetInsertPoint(bb);

llvm::FunctionType\* func\_type = llvm::FunctionType::get(

llvm::Type::getInt32Ty(context),

{ llvm::Type::getInt8PtrTy(context) },

true);

Function\* printf = llvm::Function::Create(func\_type, llvm::GlobalValue::ExternalLinkage, "printf", module);

Value\* value = ConstantInt::get(context, APInt(32, 1));

std::vector<Value\*> printf\_args = { builder.CreateGlobalStringPtr("%d\n"), value };

builder.CreateCall(printf, printf\_args);

Value\* ret\_value = ConstantInt::get(context, APInt(32, 0));

builder.CreateRet(ret\_value);

module.dump();

这一段代码运行后，得到如下的IR：

; ModuleID = 'IRbuilderTest'

source\_filename = "IRbuilderTest"

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

define i32 @main() {

entry:

%0 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @0, i32 0, i32 0), i32 1)

ret i32 0

}

declare i32 @printf(i8\* %0, ...)

这一段IR是本章开头给出的源代码对应的IR代码，运行结果是输出一个数字1。

## 3.4 测试

通常来说，程序的正确性需要通过测试来验证，围绕测试也发展出了一系列理论和工具。一般将测试分为单元测试和集成测试。单元测试的对象是单个函数或者单个类，对实现的正确性进行测试。而集成测试的对象则是一个完整的功能，对整体的功能性进行测试。

本实验在之前被划分为两个部分，对于两个部分可以单独进行集成测试，这样能更快发现问题。具体的测试案例可以自己编写，在完成代码后，运行测试，验证正确性。推荐编写测试脚本，自动运行测试，并验证测试结果的正确性，这样在每次进行代码修改后，可以方便地进行测试。

# 第四章 类型系统与变量，四则运算

## 4.1 概览

这一章中我们编译程序将支持定义变量，变量有整数int和浮点数float两种类型，同时支持进行四则运算，如下：

int main() {

int a;

float b;

a = 2;

b = a \* 1.5;

print(b);

return 0;

}

为了完成这一过程，我们需要完成词法分析、语法分析、中间代码生成的过程。

## 4.2语法分析

词法分析是编译的基础，但本身难度并不高，因此后文不再赘述这方面的内容，由读者自行完成，后续章节均仅对语法进行讨论。

为简单起见，我们不支持变量在定义时赋初值，因此变量定义会变为单独的语句，并且一次只能定义一个变量。

语句部分，之前我们只支持了函数调用和返回两个语句，此时需要添加表达式计算、赋值、变量声明语句，同时函数调用可以被统一为表达式计算的一部分，即：

stmt -> expr\_stmt | var\_decl | assign\_stmt | return stmt

赋值语句，目前可以简化为，变量=值。

assign\_stmt -> var = expr ;

四则运算的语法实现则较为复杂，涉及运算的优先级（先乘除后加减）与结合率（左结合）。此处有两种方法，一种方法是使用类似于E+E消除二义性的方法，定义出乘除项与加减项，再定义最终结果项。另一种方法是使用py yacc提供的运算符优先级定义规则。

先使用如下规则定义语法：

expr -> expr \* expr | expr / expr | expr + expr | expr - expr

为消除二义性，声明如下优先级规则：

precedence = (

('left', '+', '-'),

('left', '\*', '/')

)

其中precedence为固定，外层列表中项的顺序表示优先级从低到高，内层列表中’left’表示左结合，之后跟上运算符。这一方法仅适用于单字符表示的运算符，因此>=、<=这些多个字符表示的运算符不能用这种方式，必须用前面说的方式一。

经过扩展后，语法规则为：

program -> decl\_list;

decl\_list -> decl\_list decl | decl

decl -> var\_decl | func\_decl

var\_decl -> type\_spec var ;

func\_decl -> type\_spec ID ( ) compound\_stmt

type\_spec -> INT | FLOAT

compound\_stmt -> { stmt\_list }

stmt\_list -> stmt\_list stmt | stmt

stmt -> expr\_stmt | var\_decl | call\_stmt | return\_stmt | assign\_stmt

assign\_stmt -> var = expr ;

expr\_stmt -> expr ;

expr -> expr + expr | expr – expr | expr \* expr | expr / expr

| ( expr ) | var | INT\_LITERAL | FLOAT\_LITERAL

var -> ID

call\_stmt -> ID ( args ) ;

args -> args, expr | expr | empty

arg -> expr

return\_stmt -> RETURN expr ;

## 4.3中间代码生成

中间代码生成部分，主要涉及到两点，分别是变量定义以及值的计算。

在LLVM中，值以llvm::Value类型表示，当一条IR指令会产生一个结果时，这个结果会以llvm::Value进行返回。利用C++的继承机制，有可能是Value的一个子类，如果我们不关心具体的子类类型，可以使用父类指针来统一接收。

比如两个值的加法，调用IRBuilder::CreateAdd可以创建一条加法指令，查阅接口文档[[2]](#footnote-2)可以看到，函数原型为：

Value \* CreateAdd (Value \*LHS, Value \*RHS,   
const Twine &Name="", bool HasNUW=false, bool HasNSW=false)

前两个参数分别为加法的左右两个操作数，第三个参数Name的含义不清楚，后两个参数应该是和处理溢出有关，分别是无符号数与有符号数溢出时的处理操作。任何程序都不可避免要处理一些异常情况，加法的异常情况就是两个数相加后溢出，为了简单考虑，我们的程序可以不进行特殊处理，大家可以自行观察发生溢出后的程序行为。其余四则运算也可以在接口文档中查到，请自行试验并采用正确的代码。

变量的定义则相对复杂，需要理解LLVM IR的规则。LLVM IR定义了一种寄存器机器，值有三种不同的存储方式：寄存器变量，栈变量，全局变量。寄存器变量可以进行运算，每个指令的结果也被存储在一个寄存器中。LLVM IR采用SSA形式，每个寄存器只能赋值一次，可以理解为每个寄存器在定义时可以赋一次初值，后续不能再修改。栈变量和全局变量可以类比C++中的概念。

需要注意的是，这里的寄存器是LLVM IR定义的一个概念，不代表实际的运行情况。比如Intel CPU事实上支持寄存器与内存单元之间的运算，但LLVM IR中不支持。LLVM IR要求所有的内存数据必须先加载到寄存器中，在运算完成后可以再写回内存单元。这一点与Arm架构比较类似。LLVM IR提供了两条指令Load和Store进行处理。另外，内存变量的使用还涉及到内存空间的分配，有一条指令Alloca进行处理。

上述三条IR指令可以通过IRBuilder的三个方法CreateAlloca、CreateLoad、CreateStore来生成。CreateAlloca的一种重载形式如下：

AllocaInst \*CreateAlloca(Type \*Ty, Value \*ArraySize = nullptr, const Twine &Name = "")

第一个参数是需要分配的类型，第二个参数和第三个参数不清楚具体是什么含义，经实验，第二个参数可直接设置为nullptr，第三个参数可直接设置成变量名称。分配一个Int变量的代码段如下：

Type\* type = Type::getInt32Ty(context);

AllocaInst \*alloca = Builder.CreateAlloca(type, 0, var\_name.c\_str());

这两行代码首先获取了Int32类型，之后分配了一个AllocaInst对象alloca，我们需要将其与对应的变量名相关联，以便于后续使用，通常方法是使用HashTable等表结构。据此可定义出不同的作用域规则。

因此，在定义变量时，需要给变量分配空间，可以使用IRBuilder::CreateAlloca方法。在分配空间之后，需要将变量名与对应的存储单元相关联，通常使用HashTable等表结构，方便后续进行查找。变量的存储与访问可以使用IRBuilder::CreateStore和IRBuilder::CreateLoad方法。代码示例如下：

Value\* value = Builder.CreateLoad(type, alloca, “templv”);

Builder.CreateStore(value, alloca);

读者可根据以上代码示范，自行查阅官方文档完成变量的管理。

同样，我们给出代码，展示如何使用程序完成本文开始的那段示例程序的IR翻译。由于各章演示代码中有很多重复的初始化以及输出工作，我们提取出一个抽象数据类型：

class IR\_gen {

private:

llvm::LLVMContext context;

llvm::IRBuilder<> builder;

llvm::Module module;

Function\* printf;

BasicBlock\* bb;

};

通过这个类的构造函数，完成几个成员变量的初始化：

IR\_gen():context(), builder(context), module("IR\_example", context),

printf(nullptr), bb(nullptr) {}

之后通过一个初始化函数init\_module，完成更多初始化工作：

void IR\_gen::init\_module() {

// 初始化 main 函数

Type\* ret\_type = Type::getInt32Ty(context);

FunctionType\* FT = FunctionType::get(ret\_type, {}, false);

Function\* F = Function::Create(FT, Function::ExternalLinkage, "main", module);

bb = BasicBlock::Create(context, "entry", F);

// 声明 printf 函数

llvm::FunctionType\* func\_type = llvm::FunctionType::get(

llvm::Type::getInt32Ty(context),

{ llvm::Type::getInt8PtrTy(context) },

true);

printf = llvm::Function::Create(func\_type, llvm::GlobalValue::ExternalLinkage, "printf", module);

}

之后在这个类的另一个成员函数里，可以完成本章的示例IR生成：

init\_module();

builder.SetInsertPoint(bb);

// 创建变量

Type\* int\_type = Type::getInt32Ty(context);

AllocaInst\* a = builder.CreateAlloca(int\_type, 0, "a");

Type\* float\_type = Type::getFloatTy(context);

AllocaInst\* b = builder.CreateAlloca(float\_type, 0, "b");

// 变量存取和运算

builder.CreateStore(ConstantInt::get(context, APInt(32, 2)), a);

Value\* a\_value = builder.CreateLoad(int\_type, a);

Value\* lhs = builder.CreateSIToFP(a\_value, float\_type);

Value\* rhs = ConstantFP::get(context, APFloat(1.5f));

Value\* temp = builder.CreateFMul(lhs, rhs);

builder.CreateStore(temp, b);

// 输出

Value\* b\_value = builder.CreateLoad(float\_type, b);

Type\* double\_type = Type::getDoubleTy(context);

// printf只接收double类型，需要进行位数扩展

Value\* out = builder.CreateFPExt(b\_value, double\_type);

std::vector<Value\*> printf\_args = { builder.CreateGlobalStringPtr("%f\n"), out };

builder.CreateCall(printf, printf\_args);

Value\* ret\_value = ConstantInt::get(context, APInt(32, 0));

builder.CreateRet(ret\_value);

module.dump();

这段代码整体比较简单，要修改变量的值使用CreateStore，要读取变量的值使用CreateLoad，两个不同类型的值要进行运算，需要先进行类型转换。唯一需要注意的一点是，printf函数输出浮点数需要double类型，如果是float类型，需要先进行类型转换。为避免这一问题，可以考虑在一开始就全部使用double类型。最后得到的IR代码为：

; ModuleID = 'IR\_example'

source\_filename = "IR\_example"

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

define i32 @main() {

entry:

%a = alloca i32, align 4

%b = alloca float, align 4

store i32 2, i32\* %a, align 4

%0 = load i32, i32\* %a, align 4

%1 = sitofp i32 %0 to float

%2 = fmul float %1, 1.500000e+00

store float %2, float\* %b, align 4

%3 = load float, float\* %b, align 4

%4 = fpext float %3 to double

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

ret i32 0

}

declare i32 @printf(i8\* %0, ...)

运行会输出结果3.000000。

# 第五章 函数与复合语句

## 5.1 概览

函数是现代编程语言最重要的功能之一，为实现函数体的定义，我们还需要支持复合语句。

int add(int n1, int n2) {

int s;

s = n1 + n2;

return s;

}

int main() {

int s;

s = add(1, 2);

print(s);

return 0;

}

为了完成这一过程，主要需要支持函数的定义和调用。

5.2 词法分析与语法分析

子函数调用需要支持两点，一是子函数的定义，一是子函数的调用。

基本块的概念；语句的InsertPoint。

## 5.3 中间代码生成

很显然，复合语句主要在语法识别的层面上处理，对于中间代码生成的影响很小，因此本章我们只对如何创建函数进行说明。LLVM支持创建函数，首先需要定义函数的类型，然后根据类型创建函数，示例代码如下：

Type\* int\_type = Type::getInt32Ty(context);

std::vector<Type \*> para\_types;

para\_types.push\_back(int\_type);

para\_types.push\_back(int\_type);

FunctionType \*FT = FunctionType::get(

return\_type, para\_types, false);

Function \*F = Function::Create(FT, Function::ExternalLinkage, func\_name, module);

以上代码首先创建了一个函数类型FT，其返回值是return\_type，参数为两个int\_type，之后使用Function::Create可以创建一个函数。

函数还需要函数体，一个函数体由多个基本块组成，初始可以只添加一个，如下：

BasicBlock \*BB = BasicBlock::Create(g\_context, "entry", F);

Builder.SetInsertPoint(BB);

这里创建基本块时直接绑定在了函数F上。SetInsertPoint函数在前文已经介绍过，后续通过Builder创建的IR语句会顺序插入这一基本块中。

函数的实参可以通过getArg来读取，如下：

AllocaInst \*a = Builder.CreateAlloca(int\_type, 0, "a");

Builder.CreateStore(F->getArg(0), a);

以上代码将函数的第一个参数读取到了变量a中。

返回值通过CreateRet插入，前文中已经介绍过，用法很简单：

Builder.CreateRet(v);

在返回语句之后不应该再有其他语句。

本章以上一章中的IR\_gen为基础，完整演示上述功能。这次我们在IR\_gen中再增加一个函数，实现main函数的收尾，使用printf输出一个int32的变量，并且return 0：

void IR\_gen::finish\_module\_with\_int(Value\* out\_int) {

Type\* int\_type = Type::getInt32Ty(context);

Value\* out\_v = builder.CreateLoad(int\_type, out\_int);

llvm::Constant\* format\_string = builder.CreateGlobalStringPtr("%d\n");

std::vector<Value\*> printf\_args = { format\_string, out\_v };

builder.CreateCall(printf, printf\_args);

Value\* ret\_value = ConstantInt::get(context, APInt(32, 0));

builder.CreateRet(ret\_value);

// 输出 IR 代码

module.dump();

}

以下代码创建一个函数，实现两个操作数相加，并将加法的结果返回，之后在main函数中调用这一函数并输出结果：

init\_module();

// 定义add函数

Type\* int\_type = Type::getInt32Ty(context);

FunctionType\* add\_ft = FunctionType::get(int\_type, { int\_type, int\_type }, false);

Function\* add\_f = Function::Create(add\_ft, Function::ExternalLinkage, "add", module);

BasicBlock\* add\_bb = BasicBlock::Create(context, "entry", add\_f);

builder.SetInsertPoint(add\_bb);

AllocaInst\* n1 = builder.CreateAlloca(int\_type, 0, "n1");

builder.CreateStore(add\_f->getArg(0), n1);

AllocaInst\* n2 = builder.CreateAlloca(int\_type, 0, "n2");

builder.CreateStore(add\_f->getArg(1), n2);

Value\* sum = builder.CreateAdd(builder.CreateLoad(int\_type, n1), builder.CreateLoad(int\_type, n2));

builder.CreateRet(sum);

// 创建变量

builder.SetInsertPoint(bb);

AllocaInst\* s = builder.CreateAlloca(int\_type, 0, "s");

Value\* lv = ConstantInt::get(context, APInt(32, 1));

Value\* rv = ConstantInt::get(context, APInt(32, 2));

Value\* ret\_v = builder.CreateCall(add\_f, { lv, rv });

builder.CreateStore(ret\_v, s);

finish\_module\_with\_int(s);

这段代码运行之后输出如下：

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

define i32 @main() {

entry:

%s = alloca i32, align 4

%0 = call i32 @add(i32 1, i32 2)

store i32 %0, i32\* %s, align 4

%1 = load i32, i32\* %s, align 4

%2 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @0, i32 0, i32 0), i32 %1)

ret i32 0

}

declare i32 @printf(i8\* %0, ...)

define i32 @add(i32 %0, i32 %1) {

entry:

%n1 = alloca i32, align 4

store i32 %0, i32\* %n1, align 4

%n2 = alloca i32, align 4

store i32 %1, i32\* %n2, align 4

%2 = load i32, i32\* %n2, align 4

%3 = load i32, i32\* %n1, align 4

%4 = add i32 %3, %2

ret i32 %4

}

编译并运行，正确输出3。

# 第六章 if语句，逻辑运算

## 6.1 概览

本章将提供选择语句的支持，为实现选择语句的条件判断，我们还需要支持逻辑运算。在完成本章的内容后，应该能支持以下代码：

int abs(int x) {

if (x < 0)

x = 0 - x;

return x;

}

int main() {

int s;

s = abs(-1);

print(s);

return 0;

}

## 6.2 语法分析

if语句的语法从直观上来看是以下写法：

if-stmt -> if (expr) stmt

| if (expr) stmt else stmt

以上写法是经典的二义文法，在if-else中的then分支中有if语句时会带来二义性判断问题，即else分支应该匹配近的if还是远的if。一般的规则是就近匹配，读者可自行在网上查找解决方案。

以下是网上找到的一种可行的改写方法，将if语句分为if-else完全匹配的语句与if比else至少多一个的语句两部分，优先进行完全匹配，再进行不完全匹配。具体语法如下：

stmt -> matched-if-stmt | unmatched-if-stmt

matched-if-stmt -> if (expr) matched-if-stmt

else matched-if-stmt

unmatched-if-stmt -> if (expr) stmt

| if (expr) matched-if-stmt

else unmatched-if-stmt

完整语法是比较复杂的，为了省篇幅，后文我们仍用带有二义的写法。

逻辑运算的支持可以视为对四则运算的扩展，只是优先级比四则运算更低。我们支持逻辑与、或运算，以及比较操作。

经过扩展，语法规则为：

program -> decl\_list;

decl\_list -> decl\_list decl | decl

decl -> var\_decl | func\_decl

var\_decl -> type\_spec var ;

func\_decl -> type\_spec ID ( ) compound\_stmt

type\_spec -> INT | FLOAT

compound\_stmt -> { stmt\_list }

stmt\_list -> stmt\_list stmt | stmt

stmt -> expr\_stmt | var\_decl | call\_stmt | return\_stmt | assign\_stmt | if\_stmt

if-stmt -> if (expr) stmt

| if (expr) stmt else stmt

assign\_stmt -> var = expr ;

expr\_stmt -> expr ;

expr -> expr + expr | expr – expr | expr \* expr | expr / expr

| expr && expr | expr || expr | expr < expr

| expr == expr | expr > expr | ( expr )

| var | INT\_LITERAL | FLOAT\_LITERAL

var -> ID

call\_stmt -> ID ( args ) ;

args -> args, expr | expr | empty

arg -> expr

return\_stmt -> RETURN expr ;

## 6.3 中间代码生成

if语句的生成是相对来说比较复杂的，原因在于我们所采用的中间代码使用SSA。在if和else分支中，可能出现给同一个变量赋不同值的行为，比如下列代码：

if (a > b)

x = a;

else

x = b;

y = x + 1;

在SSA中，上下两处对x的赋值不能使用同一个寄存器，导致在后续代码中对x的使用出现了困难。LLVM IR中针对这种情况有一个专门的函数，被称为phi函数；这个函数的功能就是将if else两个不同分支中的值复合成一个值。以上的代码转换为llvm中间代码可以如此来写：

phi

在实际实现中，手动完成phi函数的处理较为复杂，有兴趣的读者可以自己实现。我们可以通过栈变量来绕过SSA，从而避免手动计算phi函数。具体来说，赋值语句的实现方式不是使用寄存器，而是真的为一个栈变量赋值。

这样实现的代码效率较低，因此我们再增加一个优化pass来生成phi函数。这一步是可选的。

选择语句的实现一般通过无条件跳转与条件跳转来实现。LLVM IR以基本块为单位，而基本块只有一个入口，因此跳转的目标直接是基本块。

无条件跳转指令br的生成方法如下：

BasicBlock\* succ\_bb = BasicBlock::Create(context, "succ", abs\_f);

builder.CreateBr(succ\_bb);

以上代码创建了一个基本块succ\_bb，并且插入了一条无条件跳转，跳转到了succ\_bb。

条件跳转指令同样是br，但带有三个参数，分别是条件，以及条件为真和假时需要跳转到的基本块，如下：

Value\* cond = builder.CreateICmpSLT(lv, rv);

BasicBlock\* then\_bb = BasicBlock::Create(context, "then", abs\_f);

BasicBlock\* succ\_bb = BasicBlock::Create(context, "succ", abs\_f);

builder.CreateCondBr(cond, then\_bb, succ\_bb);

这段代码首先生成了一条比较指令，将lv和rv的值进行比较，CreateICmpSLT中，I表示整数，S表示有符号，LT表示小于，并将比较结果存储在cond中，之后创建了两个基本块，当cond为真时，跳转到then\_bb，否则跳转到succ\_bb。

本章开头的代码对应的IR可使用如下的代码生成：

init\_module();

// 定义abs函数

Type\* int\_type = Type::getInt32Ty(context);

FunctionType\* abs\_ft = FunctionType::get(int\_type, { int\_type }, false);

Function\* abs\_f = Function::Create(abs\_ft, Function::ExternalLinkage, "abs", module);

BasicBlock\* abs\_bb = BasicBlock::Create(context, "entry", abs\_f);

builder.SetInsertPoint(abs\_bb);

AllocaInst\* p\_n = builder.CreateAlloca(int\_type, 0, "n");

builder.CreateStore(abs\_f->getArg(0), p\_n);

Value\* n = builder.CreateLoad(int\_type, p\_n);

Value\* zero = ConstantInt::get(context, APInt(32, 0));

Value\* cond = builder.CreateICmpSLT(n, zero);

// 跳转时需要跳转到基本块，先进行定义

BasicBlock\* then\_bb = BasicBlock::Create(context, "then", abs\_f);

BasicBlock\* succ\_bb = BasicBlock::Create(context, "succ", abs\_f);

builder.CreateCondBr(cond, then\_bb, succ\_bb);

// then分支，最后无条件跳转到succ块

builder.SetInsertPoint(then\_bb);

Value\* v = builder.CreateSub(zero, n);

builder.CreateStore(v, p\_n);

builder.CreateBr(succ\_bb);

// succ块，返回结果

builder.SetInsertPoint(succ\_bb);

Value\* ret\_v = builder.CreateLoad(int\_type, p\_n);

builder.CreateRet(ret\_v);

// main 函数

builder.SetInsertPoint(bb);

AllocaInst\* s = builder.CreateAlloca(int\_type, 0, "s");

Value\* lv = ConstantInt::get(context, APInt(32, -1));

Value\* out\_v = builder.CreateCall(abs\_f, { lv });

builder.CreateStore(out\_v, s);

finish\_module(s);

以下代码将生成出如下IR：

; ModuleID = 'IR\_example'

source\_filename = "IR\_example"

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

define i32 @main() {

entry:

%s = alloca i32, align 4

%0 = call i32 @abs(i32 -1)

store i32 %0, i32\* %s, align 4

%1 = load i32, i32\* %s, align 4

%2 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @0, i32 0, i32 0), i32 %1)

ret i32 0

}

declare i32 @printf(i8\* %0, ...)

define i32 @abs(i32 %0) {

entry:

%n = alloca i32, align 4

store i32 %0, i32\* %n, align 4

%1 = load i32, i32\* %n, align 4

%2 = icmp slt i32 %1, 0

br i1 %2, label %then, label %succ

then: ; preds = %entry

%3 = sub i32 0, %1

store i32 %3, i32\* %n, align 4

br label %succ

succ: ; preds = %then, %entry

%4 = load i32, i32\* %n, align 4

ret i32 %4

}

## 6.4 测试与while语句

读者可自行对if语句的功能进行测试，应该对有else分支及无else两种情况分别设计测试案例进行测试，并对if-else嵌套的二义性情况进行验证。

while语句中用到的IR实际上与if并无区别，只是语法分析和调用的方式有所不同，读者应该能自行根据while循环的功能进行实现。

# 第七章 数组

## 7.1 概览

数组应该支持定义以及使用：

int main() {

int a[3];

a[0] = 1;

a[1] = 2;

a[2] = 4;

print(a[1]);

return 0;

}

## 7.2 中间代码生成：

首先，数组的使用与其他类型的使用比较类似，使用ArrayType::get方法可以取得一个数组类型，第一个参数是元素类型，第二个参数是数组的长度，可以通过嵌套生成多维数组。以下代码产生一个[3 x i32]类型的数组：

Type\* int\_type = Type::getInt32Ty(context);

Type\* array\_type = ArrayType::get(int\_type, 3);

AllocaInst\* array = builder.CreateAlloca(array\_type);

从数组中取得对应的元素是一个相对复杂的操作，需要使用LLVM中的GetElementPtr（GEP）指令。GEP进行指针运算，可以实现取数组/指针中的第i个元素的地址、取结构体中的第i个成员的地址这两种不同的功能。在使用时有一点需要注意，如果对上面的array使用GEP指令，需要进行两次GEP操作，因为CreateAlloca操作得到的是指向[3 x i32]数组的指针，即[3 x i32]\*，用C语言的描述，array是一个指向数组的指针，所以第一次GEP需要在这个指针上进行操作，得到具体的数组，第二次GEP再取得数组中对应的元素（的地址）。示例如下：

Value\* a0 = builder.CreateGEP(array\_type, array,

{ ConstantInt::get(context, APInt(32, 0)), ConstantInt::get(context, APInt(32, 0)) });

builder.CreateStore(ConstantInt::get(context, APInt(32, 1)), a0);

以上例子中，第一个参数是数组类型，第二个参数是数组地址，第三个参数是GEP操作的索引数组，可以一次将多次GEP操作的索引作为数组传入。上面传递的索引{0, 0}，很显然取到的地址是array中第1个元素的首地址，假设为地址p。i32占用4个字节，若传递的参数为{0, 1}，则表示array中第2个元素的地址，为p+4。若传递的参数为{1, 0}，则取到的是[3 x i32]整个数组后的一个数组的首地址，为p+3\*4=p+12。由于使用长度为2的索引数组取GEP是一个常用操作，因此LLVM提供了一个单独的函数CreateConstGEP2\_32，下面的语句与上面等价：

Value\* a0 = builder.CreateConstGEP2\_32(array\_type, array, 0, 0);

builder.CreateStore(ConstantInt::get(context, APInt(32, 1)), a0);

综上，我们使用一段完整代码来完成示例IR的生成：

init\_module();

builder.SetInsertPoint(bb);

// 定义数组

Type\* int\_type = Type::getInt32Ty(context);

Type\* array\_type = ArrayType::get(int\_type, 3);

AllocaInst\* array = builder.CreateAlloca(array\_type);

// 使用GEP取得元素地址，再使用地址取值

Value\* a0 = builder.CreateGEP(array\_type, array,

{ ConstantInt::get(context, APInt(32, 0)), ConstantInt::get(context, APInt(32, 0)) });

builder.CreateStore(ConstantInt::get(context, APInt(32, 1)), a0);

// 对于GEP后面有两个常数参数的情况，可以简写为下列函数

Value\* a1 = builder.CreateConstGEP2\_32(array\_type, array, 0, 1);

builder.CreateStore(ConstantInt::get(context, APInt(32, 2)), a1);

Value\* a2 = builder.CreateConstGEP2\_32(array\_type, array, 0, 2);

builder.CreateStore(ConstantInt::get(context, APInt(32, 4)), a2);

Value\* out = builder.CreateConstGEP2\_32(array\_type, array, 0, 1);

finish\_module\_with\_int(out);

这段代码运行后得到下面的IR：

; ModuleID = 'IR\_example'

source\_filename = "IR\_example"

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

define i32 @main() {

entry:

%0 = alloca [3 x i32], align 4

%1 = getelementptr [3 x i32], [3 x i32]\* %0, i32 0, i32 0

store i32 1, i32\* %1, align 4

%2 = getelementptr [3 x i32], [3 x i32]\* %0, i32 0, i32 1

store i32 2, i32\* %2, align 4

%3 = getelementptr [3 x i32], [3 x i32]\* %0, i32 0, i32 2

store i32 4, i32\* %3, align 4

%4 = getelementptr [3 x i32], [3 x i32]\* %0, i32 0, i32 1

%5 = load i32, i32\* %4, align 4

%6 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @0, i32 0, i32 0), i32 %5)

ret i32 0

}

declare i32 @printf(i8\* %0, ...)

在运行后，程序输出1。

## 7.3 测试

请对一维数组和多维数组的情况进行测试。

1. Python lex和yacc库，可见：<https://ply.readthedocs.io/en/latest/index.html>，网上有旧版本的中文翻译：<https://www.cnblogs.com/P_Chou/p/python-lex-yacc.html> [↑](#footnote-ref-1)
2. https://llvm.org/doxygen/classllvm\_1\_1IRBuilderBase.html [↑](#footnote-ref-2)