# 使用 LLVM 实现一个简单编译器

**1 目标**

本节参考了 LLVM 的Kaleidoscope 教程，增加了对代码的注释以及一些理解，修改了部分代码。现在开始要使用 LLVM 实现一个编译器，完成对如下代码的编译运行。

# 斐波那契数列函数定义

def fib(x)

if x < 3 then

1

else

fib(x - 1) + fib(x - 2)

fib(40)

# 函数声明

extern sin(arg)

extern cos(arg)

extern atan2(arg1 arg2)

# 声明后的函数可调用

atan2(sin(.4), cos(42))

这个语言称为 Kaleidoscope, 从代码可以看出，Kaleidoscope 支持函数、条件分支、数值计算等语言特性。为了方便，Kaleidoscope 唯一支持的数据类型为 float64, 所以示例中的所有数值都是 float64。

**2 词法分析**

编译的第一个步骤称为 Lex（英文全名：lexical analysis）, 词法分析，其功能是将文本输入转为多个 tokens, 比如对于如下代码：

atan2(sin(.4), cos(42))

就应该转为：

tokens = ["atan2", "(", "sin", "(", .4, ")", ",", "cos", "(", 42, ")", ")"]

接下来使用 C++来写这个 Lexer, 由于这是教程代码，所以并没有使用工程项目应有的设计：

// 如果不是以下5种情况，Lexer返回[0-255]的ASCII值，否则返回以下枚举值

enum Token {

TOKEN\_EOF = -1, // 文件结束标识符

TOKEN\_DEF = -2, // 关键字def

TOKEN\_EXTERN = -3, // 关键字extern

TOKEN\_IDENTIFIER = -4, // 名字

TOKEN\_NUMBER = -5 // 数值

};

std::string g\_identifier\_str; // Filled in if TOKEN\_IDENTIFIER

double g\_number\_val; // Filled in if TOKEN\_NUMBER

// 从标准输入解析一个Token并返回

int GetToken() {

static int last\_char = ' ';

// 忽略空白字符

while (isspace(last\_char)) {

last\_char = getchar();

}

// 识别字符串

if (isalpha(last\_char)) {

g\_identifier\_str = last\_char;

while (isalnum((last\_char = getchar()))) {

g\_identifier\_str += last\_char;

}

if (g\_identifier\_str == "def") {

return TOKEN\_DEF;

} else if (g\_identifier\_str == "extern") {

return TOKEN\_EXTERN;

} else {

return TOKEN\_IDENTIFIER;

}

}

// 识别数值

if (isdigit(last\_char) || last\_char == '.') {

std::string num\_str;

do {

num\_str += last\_char;

last\_char = getchar();

} while (isdigit(last\_char) || last\_char == '.');

g\_number\_val = strtod(num\_str.c\_str(), nullptr);

return TOKEN\_NUMBER;

}

// 忽略注释

if (last\_char == '#') {

do {

last\_char = getchar();

} while (last\_char != EOF &amp;&amp; last\_char != '\n' &amp;&amp; last\_char != '\r');

if (last\_char != EOF) {

return GetToken();

}

}

// 识别文件结束

if (last\_char == EOF) {

return TOKEN\_EOF;

}

// 直接返回ASCII

int this\_char = last\_char;

last\_char = getchar();

return this\_char;

}

使用 Lexer 对之前的代码处理结果为（使用空格分隔 tokens）：

def fib ( x ) if x < 3 then 1 else fib ( x - 1 ) + fib ( x - 2 ) fib ( 40 ) extern sin ( arg )

extern cos ( arg ) extern atan2 ( arg1 arg2 ) atan2 ( sin ( 0.4 ) , cos ( 42 ) )

Lexer 的输入是代码文本，输出是有序的一个个 Token。

**3 语法分析**

编译的第二个步骤称为 Parse, 语法分析，其功能是将 Lexer 输出的 tokens 转为 AST (Abstract Syntax Tree)。首先定义表达式的 AST 节点：

// 所有表达式节点的基类

class ExprAST {

public:

virtual ~ExprAST() {}

};

// 字面值表达式

class NumberExprAST : public ExprAST {

public:

NumberExprAST(double val) : val\_(val) {}

private:

double val\_;

};

// 变量表达式

class VariableExprAST : public ExprAST {

public:

VariableExprAST(const std::string&amp; name) : name\_(name) {}

private:

std::string name\_;

};

// 二元操作表达式

class BinaryExprAST : public ExprAST {

public:

BinaryExprAST(char op, std::unique\_ptr<ExprAST> lhs,

std::unique\_ptr<ExprAST> rhs)

: op\_(op), lhs\_(std::move(lhs)), rhs\_(std::move(rhs)) {}

private:

char op\_;

std::unique\_ptr<ExprAST> lhs\_;

std::unique\_ptr<ExprAST> rhs\_;

};

// 函数调用表达式

class CallExprAST : public ExprAST {

public:

CallExprAST(const std::string&amp; callee,

std::vector<std::unique\_ptr<ExprAST>> args)

: callee\_(callee), args\_(std::move(args)) {}

private:

std::string callee\_;

std::vector<std::unique\_ptr<ExprAST>> args\_;

};

为了便于理解，关于条件表达式的内容放在后面，这里暂不考虑。接着定义函数声明和函数的AST节点：

// 函数接口

class PrototypeAST {

public:

PrototypeAST(const std::string&amp; name, std::vector<std::string> args)

: name\_(name), args\_(std::move(args)) {}

const std::string&amp; name() const { return name\_; }

private:

std::string name\_;

std::vector<std::string> args\_;

};

// 函数

class FunctionAST {

public:

FunctionAST(std::unique\_ptr<PrototypeAST> proto,

std::unique\_ptr<ExprAST> body)

: proto\_(std::move(proto)), body\_(std::move(body)) {}

private:

std::unique\_ptr<PrototypeAST> proto\_;

std::unique\_ptr<ExprAST> body\_;

};

接下来要进行 Parse, 在正式 Parse 前，定义如下函数方便后续处理：

int g\_current\_token; // 当前待处理的Token

int GetNextToken() {

return g\_current\_token = GetToken();

}

首先处理最简单的字面值：

// numberexpr ::= number

std::unique\_ptr<ExprAST> ParseNumberExpr() {

auto result = std::make\_unique<NumberExprAST>(g\_number\_val);

GetNextToken();

return std::move(result);

}

这段程序非常简单，当前符号为 TOKEN\_NUMBER 时被调用，使用 g\_number\_val，创建一个 NumberExprAST, 因为当前符号处理完毕，让 Lexer 前进一个 Token, 最后返回。接着处理圆括号操作符、变量、函数调用：

// parenexpr ::= ( expression )

std::unique\_ptr<ExprAST> ParseParenExpr() {

GetNextToken(); // eat (

auto expr = ParseExpression();

GetNextToken(); // eat )

return expr;

}

/// identifierexpr

/// ::= identifier

/// ::= identifier ( expression, expression, ..., expression )

std::unique\_ptr<ExprAST> ParseIdentifierExpr() {

std::string id = g\_identifier\_str;

GetNextToken();

if (g\_current\_token != '(') {

return std::make\_unique<VariableExprAST>(id);

} else {

GetNextToken(); // eat (

std::vector<std::unique\_ptr<ExprAST>> args;

while (g\_current\_token != ')') {

args.push\_back(ParseExpression());

if (g\_current\_token == ')') {

break;

} else {

GetNextToken(); // eat ,

}

}

GetNextToken(); // eat )

return std::make\_unique<CallExprAST>(id, std::move(args));

}

}

上面代码中的 ParseExpression 与 ParseParenExpr 等存在循环依赖，这里按照其名字理解意思即可，具体实现在后面。将 NumberExpr、ParenExpr、IdentifierExpr 视为 PrimaryExpr, 封装 ParsePrimary 方便后续调用：

/// primary

/// ::= identifierexpr

/// ::= numberexpr

/// ::= parenexpr

std::unique\_ptr<ExprAST> ParsePrimary() {

switch (g\_current\_token) {

case TOKEN\_IDENTIFIER: return ParseIdentifierExpr();

case TOKEN\_NUMBER: return ParseNumberExpr();

case '(': return ParseParenExpr();

default: return nullptr;

}

}

接下来考虑如何处理二元操作符，为了方便，Kaleidoscope 只支持 4 种二元操作符，优先级为：

'<' < '+' = '-' < '\*'

即'<'的优先级最低，而'\*'的优先级最高，在代码中实现为：

// 定义优先级

const std::map<char, int> g\_binop\_precedence = {

{'<', 10}, {'+', 20}, {'-', 20}, {'\*', 40}};

// 获得当前Token的优先级

int GetTokenPrecedence() {

auto it = g\_binop\_precedence.find(g\_current\_token);

if (it != g\_binop\_precedence.end()) {

return it->second;

} else {

return -1;

}

}

对于带优先级的二元操作符的解析，会将其分成多个片段。比如一个表达式：

a + b + (c + d) \* e \* f + g

首先解析 a, 然后处理多个二元组：

[+, b], [+, (c+d)], [\*, e], [\*, f], [+, g]

也就是说，复杂表达式可以抽象为一个 PrimaryExpr 跟着多个[binop, PrimaryExpr]二元组，注意由于圆括号属于 PrimaryExpr, 所以这里不需要考虑怎么特殊处理(c+d)，因为会被 ParsePrimary 自动处理。

// parse

// lhs [binop primary] [binop primary] ...

// 如遇到优先级小于min\_precedence的操作符，则停止

std::unique\_ptr<ExprAST> ParseBinOpRhs(int min\_precedence,

std::unique\_ptr<ExprAST> lhs) {

while (true) {

int current\_precedence = GetTokenPrecedence();

if (current\_precedence < min\_precedence) {

// 如果当前token不是二元操作符，current\_precedence为-1, 结束任务

// 如果遇到优先级更低的操作符，也结束任务

return lhs;

}

int binop = g\_current\_token;

GetNextToken(); // eat binop

auto rhs = ParsePrimary();

// 现在有两种可能的解析方式

// \* (lhs binop rhs) binop unparsed

// \* lhs binop (rhs binop unparsed)

int next\_precedence = GetTokenPrecedence();

if (current\_precedence < next\_precedence) {

// 将高于current\_precedence的右边的操作符处理掉返回

rhs = ParseBinOpRhs(current\_precedence + 1, std::move(rhs));

}

lhs =

std::make\_unique<BinaryExprAST>(binop, std::move(lhs), std::move(rhs));

// 继续循环

}

}

// expression

// ::= primary [binop primary] [binop primary] ...

std::unique\_ptr<ExprAST> ParseExpression() {

auto lhs = ParsePrimary();

return ParseBinOpRhs(0, std::move(lhs));

}

最复杂的部分完成后，按部就班把函数写完：

// prototype

// ::= id ( id id ... id)

std::unique\_ptr<PrototypeAST> ParsePrototype() {

std::string function\_name = g\_identifier\_str;

GetNextToken();

std::vector<std::string> arg\_names;

while (GetNextToken() == TOKEN\_IDENTIFIER) {

arg\_names.push\_back(g\_identifier\_str);

}

GetNextToken(); // eat )

return std::make\_unique<PrototypeAST>(function\_name, std::move(arg\_names));

}

// definition ::= def prototype expression

std::unique\_ptr<FunctionAST> ParseDefinition() {

GetNextToken(); // eat def

auto proto = ParsePrototype();

auto expr = ParseExpression();

return std::make\_unique<FunctionAST>(std::move(proto), std::move(expr));

}

// external ::= extern prototype

std::unique\_ptr<PrototypeAST> ParseExtern() {

GetNextToken(); // eat extern

return ParsePrototype();

}

最后，为顶层的代码实现匿名函数：

// toplevelexpr ::= expression

std::unique\_ptr<FunctionAST> ParseTopLevelExpr() {

auto expr = ParseExpression();

auto proto = std::make\_unique<PrototypeAST>("", std::vector<std::string>());

return std::make\_unique<FunctionAST>(std::move(proto), std::move(expr));

}

顶层代码的意思是放在全局，而不放在函数内定义的一些执行语句。比如变量赋值，函数调用等。编写一个 main 函数：

int main() {

GetNextToken();

while (true) {

switch (g\_current\_token) {

case TOKEN\_EOF: return 0;

case TOKEN\_DEF: {

ParseDefinition();

std::cout << "parsed a function definition" << std::endl;

break;

}

case TOKEN\_EXTERN: {

ParseExtern();

std::cout << "parsed a extern" << std::endl;

break;

}

default: {

ParseTopLevelExpr();

std::cout << "parsed a top level expr" << std::endl;

break;

}

}

}

return 0;

}

编译：

clang++ main.cpp `llvm-config --cxxflags --ldflags --libs`

输入如下代码进行测试：

def foo(x y)

x + foo(y, 4)

def foo(x y)

x + y

y

extern sin(a)

得到输出：

parsed a function definition

parsed a function definition

parsed a top level expr

parsed a extern

至此成功将 Lexer 输出的 tokens 转为 AST。

**4 LLVM IR的代码生成**

终于开始 codegen 了，首先 include 一些 LLVM 头文件，定义一些全局变量：

#include "llvm/ADT/APFloat.h"

#include "llvm/ADT/STLExtras.h"

#include "llvm/IR/BasicBlock.h"

#include "llvm/IR/Constants.h"

#include "llvm/IR/DerivedTypes.h"

#include "llvm/IR/Function.h"

#include "llvm/IR/IRBuilder.h"

#include "llvm/IR/LLVMContext.h"

#include "llvm/IR/LegacyPassManager.h"

#include "llvm/IR/Module.h"

#include "llvm/IR/Type.h"

#include "llvm/IR/Verifier.h"

#include "llvm/Support/TargetSelect.h"

#include "llvm/Target/TargetMachine.h"

#include "llvm/Transforms/InstCombine/InstCombine.h"

#include "llvm/Transforms/Scalar.h"

#include "llvm/Transforms/Scalar/GVN.h"

// 记录了LLVM的核心数据结构，比如类型和常量表，不过不太需要关心它的内部

llvm::LLVMContext g\_llvm\_context;

// 用于创建LLVM指令

llvm::IRBuilder<> g\_ir\_builder(g\_llvm\_context);

// 用于管理函数和全局变量，可以粗浅地理解为类c++的编译单元(单个cpp文件)

llvm::Module g\_module("my cool jit", g\_llvm\_context);

// 用于记录函数的变量参数

std::map<std::string, llvm::Value\*> g\_named\_values;

然后给每个 AST Class 增加一个 CodeGen 接口：

// 所有 `表达式` 节点的基类

class ExprAST {

public:

virtual ~ExprAST() {}

virtual llvm::Value\* CodeGen() = 0;

};

// 字面值表达式

class NumberExprAST : public ExprAST {

public:

NumberExprAST(double val) : val\_(val) {}

llvm::Value\* CodeGen() override;

private:

double val\_;

};

首先实现 NumberExprAST 的 CodeGen：

llvm::Value\* NumberExprAST::CodeGen() {

return llvm::ConstantFP::get(g\_llvm\_context, llvm::APFloat(val\_));

}

由于 Kaleidoscope 只有一种数据类型 FP64, 所以直接调用 ConstantFP 传入即可。APFloat 是 llvm 内部的数据结构，用于存储任意精度浮点。在 LLVM IR 中，所有常量是唯一且共享的，所以这里使用的get，而不是 new/create。

然后实现 VariableExprAST 的 CodeGen：

llvm::Value\* VariableExprAST::CodeGen() {

return g\_named\_values.at(name\_);

}

由于 Kaleidoscope 的 VariableExpr 只存在于函数内对函数参数的引用，假定函数参数已经被注册到 g\_name\_values 中，所以 VariableExpr 直接查表返回即可。

接着实现 BinaryExprAST, 分别 codegen lhs, rhs 然后创建指令处理 lhs, rhs 即可：

llvm::Value\* BinaryExprAST::CodeGen() {

llvm::Value\* lhs = lhs\_->CodeGen();

llvm::Value\* rhs = rhs\_->CodeGen();

switch (op\_) {

case '<': {

llvm::Value\* tmp = g\_ir\_builder.CreateFCmpULT(lhs, rhs, "cmptmp");

// 把 0/1 转为 0.0/1.0

return g\_ir\_builder.CreateUIToFP(

tmp, llvm::Type::getDoubleTy(g\_llvm\_context), "booltmp");

}

case '+': return g\_ir\_builder.CreateFAdd(lhs, rhs, "addtmp");

case '-': return g\_ir\_builder.CreateFSub(lhs, rhs, "subtmp");

case '\*': return g\_ir\_builder.CreateFMul(lhs, rhs, "multmp");

default: return nullptr;

}

}

// 实现 CallExprAST：

llvm::Value\* CallExprAST::CodeGen() {

// g\_module中存储了全局变量/函数等

llvm::Function\* callee = g\_module.getFunction(callee\_);

std::vector<llvm::Value\*> args;

for (std::unique\_ptr<ExprAST>&amp; arg\_expr : args\_) {

args.push\_back(arg\_expr->CodeGen());

}

return g\_ir\_builder.CreateCall(callee, args, "calltmp");

}

实现 ProtoTypeAST：

llvm::Value\* PrototypeAST::CodeGen() {

// 创建kaleidoscope的函数类型 double (doube, double, ..., double)

std::vector<llvm::Type\*> doubles(args\_.size(),

llvm::Type::getDoubleTy(g\_llvm\_context));

// 函数类型是唯一的，所以使用get而不是new/create

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

llvm::Type::getDoubleTy(g\_llvm\_context), doubles, false);

// 创建函数, ExternalLinkage意味着函数可能不在当前module中定义，在当前module

// 即g\_module中注册名字为name\_, 后面可以使用这个名字在g\_module中查询

llvm::Function\* func = llvm::Function::Create(

function\_type, llvm::Function::ExternalLinkage, name\_, &amp;g\_module);

// 增加IR可读性，设置function的argument name

int index = 0;

for (auto&amp; arg : func->args()) {

arg.setName(args\_[index++]);

}

return func;

}

实现 FunctionAST：

llvm::Value\* FunctionAST::CodeGen() {

// 检查函数声明是否已完成codegen(比如之前的extern声明), 如果没有则执行codegen

llvm::Function\* func = g\_module.getFunction(proto\_->name());

if (func == nullptr) {

func = proto\_->CodeGen();

}

// 创建一个Block并且设置为指令插入位置。

// llvm block用于定义control flow graph, 由于暂不实现control flow, 创建

// 一个单独的block即可

llvm::BasicBlock\* block =

llvm::BasicBlock::Create(g\_llvm\_context, "entry", func);

g\_ir\_builder.SetInsertPoint(block);

// 将函数参数注册到g\_named\_values中，让VariableExprAST可以codegen

g\_named\_values.clear();

for (llvm::Value&amp; arg : func->args()) {

g\_named\_values[arg.getName()] = &amp;arg;

}

// codegen body然后return

llvm::Value\* ret\_val = body\_->CodeGen();

g\_ir\_builder.CreateRet(ret\_val);

llvm::verifyFunction(\*func);

return func;

}

至此，所有 codegen 都已完成，修改 main：

int main() {

GetNextToken();

while (true) {

switch (g\_current\_token) {

case TOKEN\_EOF: return 0;

case TOKEN\_DEF: {

auto ast = ParseDefinition();

std::cout << "parsed a function definition" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cerr << std::endl;

break;

}

case TOKEN\_EXTERN: {

auto ast = ParseExtern();

std::cout << "parsed a extern" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cerr << std::endl;

break;

}

default: {

auto ast = ParseTopLevelExpr();

std::cout << "parsed a top level expr" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cerr << std::endl;

break;

}

}

}

return 0;

}

输入测试：

// 4 + 5

def foo(a b)

a\*a + 2\*a\*b + b\*b

foo(2, 3)

def bar(a)

foo(a, 4) + bar(31337)

extern cos(x)

cos(1.234)

得到输出：

parsed a top level expr

define double @0() {

entry:

ret double 9.000000e+00

}

parsed a function definition

define double @foo(double %a, double %b) {

entry:

%multmp = fmul double %a, %a

%multmp1 = fmul double 2.000000e+00, %a

%multmp2 = fmul double %multmp1, %b

%addtmp = fadd double %multmp, %multmp2

%multmp3 = fmul double %b, %b

%addtmp4 = fadd double %addtmp, %multmp3

ret double %addtmp4

}

parsed a top level expr

define double @1() {

entry:

%calltmp = call double @foo(double 2.000000e+00, double 3.000000e+00)

ret double %calltmp

}

parsed a function definition

define double @bar(double %a) {

entry:

%calltmp = call double @foo(double %a, double 4.000000e+00)

%calltmp1 = call double @bar(double 3.133700e+04)

%addtmp = fadd double %calltmp, %calltmp1

ret double %addtmp

}

parsed a extern

declare double @cos(double)

parsed a top level expr

define double @2() {

entry:

%calltmp = call double @cos(double 1.234000e+00)

ret double %calltmp

}

至此，已成功将 Parser 输出的 AST 转为 LLVM IR。

**5 优化器**

使用上一节的程序处理如下代码：

def test(x)

1 + 2 + x

可以得到：

parsed a function definition

define double @test(double %x) {

entry:

%addtmp = fadd double 3.000000e+00, %x

ret double %addtmp

}

可以看到，生成的指令直接是 1+2 的结果，而没有 1 + 2 的指令，这种自动把常量计算完毕而不是生成加法指令的优化称为常量折叠。

在大部分时候仅有这个优化仍然不够，比如如下代码：

def test(x)

(1 + 2 + x) \* (x + (1 + 2))

可以得到编译结果：

parsed a function definition

define double @test(double %x) {

entry:

%addtmp = fadd double 3.000000e+00, %x

%addtmp1 = fadd double %x, 3.000000e+00

%multmp = fmul double %addtmp, %addtmp1

ret double %multmp

}

生成了两个加法指令，但最优做法只需要一个加法即可，因为乘法的两边 lhs 和 rhs

是相等的。

这需要其他的优化技术，llvm 以passes的形式提供，llvm 中的 passes 可以选择是否启用，可以设置 passes 的顺序。

这里对每个函数单独做优化，定义 g\_fpm, 增加几个 passes：

llvm::legacy::FunctionPassManager g\_fpm(&amp;g\_module);

int main() {

g\_fpm.add(llvm::createInstructionCombiningPass());

g\_fpm.add(llvm::createReassociatePass());

g\_fpm.add(llvm::createGVNPass());

g\_fpm.add(llvm::createCFGSimplificationPass());

g\_fpm.doInitialization();

...

}

在 FunctionAST 的 CodeGen 中增加一句：

llvm::Value\* ret\_val = body\_->CodeGen();

g\_ir\_builder.CreateRet(ret\_val);

llvm::verifyFunction(\*func);

g\_fpm.run(\*func); // 增加这句

return func;

即启动了对每个函数的优化，接下来测试之前的代码：

parsed a function definition

define double @test(double %x) {

entry:

%addtmp = fadd double %x, 3.000000e+00

%multmp = fmul double %addtmp, %addtmp

ret double %multmp

}

可以看到，和期望的一样，加法指令减少到一个。

**6 添加JIT编译器**

由于 JIT 模式中需要反复创建新的模块, 所以将全局变量 g\_module 改为 unique\_ptr。

// 用于管理函数和全局变量，可以粗浅地理解为类c++的编译单元(单个cpp文件)

std::unique\_ptr<llvm::Module> g\_module =

std::make\_unique<llvm::Module>("my cool jit", g\_llvm\_context);

为了专注于 JIT，可以把优化的 passes 删掉。

修改 ParseTopLevelExpr，给 PrototypeAST 命名为\_\_anon\_expr, 让后面可以通过这个名字找到它。

// toplevelexpr ::= expression

std::unique\_ptr<FunctionAST> ParseTopLevelExpr() {

auto expr = ParseExpression();

auto proto =

std::make\_unique<PrototypeAST>("\_\_anon\_expr", std::vector<std::string>());

return std::make\_unique<FunctionAST>(std::move(proto), std::move(expr));

}

然后从 llvm-project 中拷贝一份代码 llvm/examples/Kaleidoscope/include/KaleidoscopeJIT.h 到本地再 include, 其定义了

KaleidoscopeJIT 类，关于这个类，在后面会做解读，这里先不管。

定义全局变量 g\_jit, 并使用 InitializeNativeTarget\*函数初始化环境。

#include "KaleidoscopeJIT.h"

std::unique\_ptr<llvm::orc::KaleidoscopeJIT> g\_jit;

int main() {

llvm::InitializeNativeTarget();

llvm::InitializeNativeTargetAsmPrinter();

llvm::InitializeNativeTargetAsmParser();

g\_jit.reset(new llvm::orc::KaleidoscopeJIT);

g\_module->setDataLayout(g\_jit->getTargetMachine().createDataLayout());

...

}

修改 main 处理顶级表达式的代码为：

auto ast = ParseTopLevelExpr();

std::cout << "parsed a top level expr" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cout << std::endl;

auto h = g\_jit->addModule(std::move(g\_module));

// 重新创建g\_module在下次使用

g\_module =

std::make\_unique<llvm::Module>("my cool jit", g\_llvm\_context);

g\_module->setDataLayout(g\_jit->getTargetMachine().createDataLayout());

// 通过名字找到编译的函数符号

auto symbol = g\_jit->findSymbol("\_\_anon\_expr");

// 强转为C函数指针

double (\*fp)() = (double (\*)())(symbol.getAddress().get());

// 执行输出

std::cout << fp() << std::endl;

g\_jit->removeModule(h);

break;

输入：

// 4 + 5

def foo(a b)

a\*a + 2\*a\*b + b\*b

foo(2, 3)

得到输出：

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

ret double 9.000000e+00

}

9

parsed a function definition

define double @foo(double %a, double %b) {

entry:

%multmp = fmul double %a, %a

%multmp1 = fmul double 2.000000e+00, %a

%multmp2 = fmul double %multmp1, %b

%addtmp = fadd double %multmp, %multmp2

%multmp3 = fmul double %b, %b

%addtmp4 = fadd double %addtmp, %multmp3

ret double %addtmp4

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 2.000000e+00, double 3.000000e+00)

ret double %calltmp

}

25

可以看到代码已经顺利执行，但现在的实现仍然是有问题的，比如上面的输入，foo 函数的定义和调用是被归在同一个 module 中，当第一次调用完成后，由于 removeModule, 第二次调用 foo 会失败。

在解决这个问题之前，先把 main 函数内对不同符号的处理拆成多个函数，如下：

void ReCreateModule() {

g\_module = std::make\_unique<llvm::Module>("my cool jit", g\_llvm\_context);

g\_module->setDataLayout(g\_jit->getTargetMachine().createDataLayout());

}

void ParseDefinitionToken() {

auto ast = ParseDefinition();

std::cout << "parsed a function definition" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cerr << std::endl;

}

void ParseExternToken() {

auto ast = ParseExtern();

std::cout << "parsed a extern" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cerr << std::endl;

}

void ParseTopLevel() {

auto ast = ParseTopLevelExpr();

std::cout << "parsed a top level expr" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cout << std::endl;

auto h = g\_jit->addModule(std::move(g\_module));

// 重新创建g\_module在下次使用

ReCreateModule();

// 通过名字找到编译的函数符号

auto symbol = g\_jit->findSymbol("\_\_anon\_expr");

// 强转为C函数指针

double (\*fp)() = (double (\*)())(symbol.getAddress().get());

// 执行输出

std::cout << fp() << std::endl;

g\_jit->removeModule(h);

}

int main() {

llvm::InitializeNativeTarget();

llvm::InitializeNativeTargetAsmPrinter();

llvm::InitializeNativeTargetAsmParser();

g\_jit.reset(new llvm::orc::KaleidoscopeJIT);

g\_module->setDataLayout(g\_jit->getTargetMachine().createDataLayout());

GetNextToken();

while (true) {

switch (g\_current\_token) {

case TOKEN\_EOF: return 0;

case TOKEN\_DEF: ParseDefinitionToken(); break;

case TOKEN\_EXTERN: ParseExternToken(); break;

default: ParseTopLevel(); break;

}

}

return 0;

}

为了解决第二次调用 foo 失败的问题，需要让 function 和 top level expr 处于不同的 Module, 而处于不同 Module 的话，CallExprAST 的 CodeGen 在当前 module 会找不到 function, 所以需要自动在 CallExprAST 做 CodeGen 时在当前 Module 声明这个函数，即自动地增加 extern, 也就是在当前 Module 自动做对应 PrototypeAST 的 CodeGen。

首先，增加一个全局变量存储从函数名到函数接口的映射，并增加一个查询函数。

std::map<std::string, std::unique\_ptr<PrototypeAST>> name2proto\_ast;

llvm::Function\* GetFunction(const std::string&amp; name) {

llvm::Function\* callee = g\_module->getFunction(name);

if (callee != nullptr) { // 当前module存在函数定义

return callee;

} else {

// 声明函数

return name2proto\_ast.at(name)->CodeGen();

}

}

更改 CallExprAST 的 CodeGen, 让其使用上面定义的 GetFuntion：

llvm::Value\* CallExprAST::CodeGen() {

llvm::Function\* callee = GetFunction(callee\_);

std::vector<llvm::Value\*> args;

for (std::unique\_ptr<ExprAST>&amp; arg\_expr : args\_) {

args.push\_back(arg\_expr->CodeGen());

}

return g\_ir\_builder.CreateCall(callee, args, "calltmp");

}

更改 FunctionAST 的 CodeGen, 让其将结果写入 name2proto\_ast：

llvm::Value\* FunctionAST::CodeGen() {

PrototypeAST&amp; proto = \*proto\_;

name2proto\_ast[proto.name()] = std::move(proto\_); // transfer ownership

llvm::Function\* func = GetFunction(proto.name());

// 创建一个Block并且设置为指令插入位置。

// llvm block用于定义control flow graph, 由于暂不实现control flow, 创建

// 一个单独的block即可

llvm::BasicBlock\* block =

llvm::BasicBlock::Create(g\_llvm\_context, "entry", func);

g\_ir\_builder.SetInsertPoint(block);

// 将函数参数注册到g\_named\_values中，让VariableExprAST可以codegen

g\_named\_values.clear();

for (llvm::Value&amp; arg : func->args()) {

g\_named\_values[arg.getName()] = &amp;arg;

}

// codegen body然后return

llvm::Value\* ret\_val = body\_->CodeGen();

g\_ir\_builder.CreateRet(ret\_val);

llvm::verifyFunction(\*func);

return func;

}

修改 ParseExternToken 将结果写入 name2proto\_ast：

void ParseExternToken() {

auto ast = ParseExtern();

std::cout << "parsed a extern" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cerr << std::endl;

name2proto\_ast[ast->name()] = std::move(ast);

}

修改 ParseDefinitionToken 让其使用独立 Module：

void ParseDefinitionToken() {

auto ast = ParseDefinition();

std::cout << "parsed a function definition" << std::endl;

ast->CodeGen()->print(llvm::errs());

std::cerr << std::endl;

g\_jit->addModule(std::move(g\_module));

ReCreateModule();

}

修改完毕，输入测试：

def foo(x)

x + 1

foo(2)

def foo(x)

x + 2

foo(2)

extern sin(x)

extern cos(x)

sin(1.0)

def foo(x)

sin(x) \* sin(x) + cos(x) \* cos(x)

foo(4)

foo(3)

得到输出：

parsed a function definition

define double @foo(double %x) {

entry:

%addtmp = fadd double %x, 1.000000e+00

ret double %addtmp

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 2.000000e+00)

ret double %calltmp

}

3

parsed a function definition

define double @foo(double %x) {

entry:

%addtmp = fadd double %x, 2.000000e+00

ret double %addtmp

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 2.000000e+00)

ret double %calltmp

}

4

parsed a extern

declare double @sin(double)

parsed a extern

declare double @cos(double)

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @sin(double 1.000000e+00)

ret double %calltmp

}

0.841471

parsed a function definition

define double @foo(double %x) {

entry:

%calltmp = call double @sin(double %x)

%calltmp1 = call double @sin(double %x)

%multmp = fmul double %calltmp, %calltmp1

%calltmp2 = call double @cos(double %x)

%calltmp3 = call double @cos(double %x)

%multmp4 = fmul double %calltmp2, %calltmp3

%addtmp = fadd double %multmp, %multmp4

ret double %addtmp

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 4.000000e+00)

ret double %calltmp

}

1

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 3.000000e+00)

ret double %calltmp

}

1

成功运行，执行正确！代码可以正确解析 sin, cos 的原因在 KaleidoscopeJIT.h 中，截取其寻找符号的代码。

JITSymbol findMangledSymbol(const std::string &amp;Name) {

#ifdef \_WIN32

// The symbol lookup of ObjectLinkingLayer uses the SymbolRef::SF\_Exported

// flag to decide whether a symbol will be visible or not, when we call

// IRCompileLayer::findSymbolIn with ExportedSymbolsOnly set to true.

//

// But for Windows COFF objects, this flag is currently never set.

// For a potential solution see: https://reviews.llvm.org/rL258665

// For now, we allow non-exported symbols on Windows as a workaround.

const bool ExportedSymbolsOnly = false;

#else

const bool ExportedSymbolsOnly = true;

#endif

// Search modules in reverse order: from last added to first added.

// This is the opposite of the usual search order for dlsym, but makes more

// sense in a REPL where we want to bind to the newest available definition.

for (auto H : make\_range(ModuleKeys.rbegin(), ModuleKeys.rend()))

if (auto Sym = CompileLayer.findSymbolIn(H, Name, ExportedSymbolsOnly))

return Sym;

// If we can't find the symbol in the JIT, try looking in the host process.

if (auto SymAddr = RTDyldMemoryManager::getSymbolAddressInProcess(Name))

return JITSymbol(SymAddr, JITSymbolFlags::Exported);

#ifdef \_WIN32

// For Windows retry without "\_" at beginning, as RTDyldMemoryManager uses

// GetProcAddress and standard libraries like msvcrt.dll use names

// with and without "\_" (for example "\_itoa" but "sin").

if (Name.length() > 2 &amp;&amp; Name[0] == '\_')

if (auto SymAddr =

RTDyldMemoryManager::getSymbolAddressInProcess(Name.substr(1)))

return JITSymbol(SymAddr, JITSymbolFlags::Exported);

#endif

return null

可以看到，在之前定义的 Module 找不到后会在 host process 中寻找这个符号。

**7 静态单一赋值**

继续给 Kaleidoscope 添加功能之前，需要先介绍 SSA（英文全名：Static Single Assignment），考虑下面代码：

y := 1

y := 2

x := y

可以发现第一个赋值是不必须的，而且第三行使用的 y 来自第二行的赋值，改成 SSA 格式为

y\_1 = 1

y\_2 = 2

x\_1 = y\_2

改完可以方便编译器进行优化，比如把第一个赋值删去，于是可以给出 SSA 的定义：

每个变量仅且必须被赋值一次，原本代码中的多次变量赋值会被赋予版本号然后视为不同变量；

每个变量在被使用之前必须被定义。

考虑如下图11.3所示的控制流图：

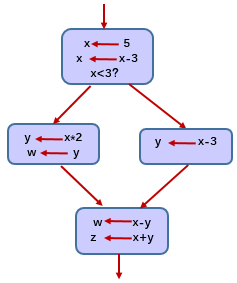


图11.3 控制流图

将控制流图加上版本号，得到如图11.4所示：

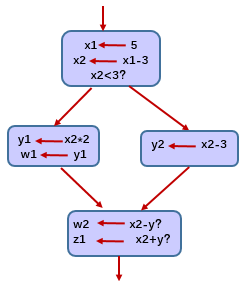


图11.4 加上版本号的控制流图

可以看到，这里遇到一个问题，最下面的 block 里面的 y 应该使用 y1 还是 y2, 为了解决这个问题，插入一个特殊语句称为 phi函数, 其会根据控制流从 y1 和 y2 中选择一个值作为 y3, 如图11.5所示：

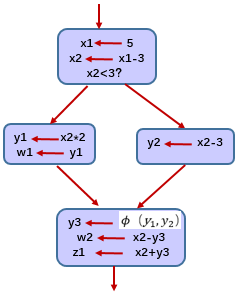


图11.5 确定y值的加上版本号的控制流图

可以看到，对于 x 不需要 phi function, 因为两个分支到最后的都是 x2。

**8 控制流**

现在实现的 Kaleidoscope 还不够完善，缺少 if else 控制流，比如不支持如下代码：

def fib(x)

if x < 3 then

1

else

fib(x - 1) + fib(x - 2)

首先让 Lexer 能识别 if then else 三个关键字，增加 TOKEN 类型：

TOKEN\_IF = -6, // if

TOKEN\_THEN = -7, // then

TOKEN\_ELSE = -8, // else

增加识别规则：

// 识别字符串

if (isalpha(last\_char)) {

g\_identifier\_str = last\_char;

while (isalnum((last\_char = getchar()))) {

g\_identifier\_str += last\_char;

}

if (g\_identifier\_str == "def") {

return TOKEN\_DEF;

} else if (g\_identifier\_str == "extern") {

return TOKEN\_EXTERN;

} else if (g\_identifier\_str == "if") {

return TOKEN\_IF;

} else if (g\_identifier\_str == "then") {

return TOKEN\_THEN;

} else if (g\_identifier\_str == "else") {

return TOKEN\_ELSE;

} else {

return TOKEN\_IDENTIFIER;

}

}

增加 IfExprAST：

// if then else

class IfExprAST : public ExprAST {

public:

IfExprAST(std::unique\_ptr<ExprAST> cond, std::unique\_ptr<ExprAST> then\_expr,

std::unique\_ptr<ExprAST> else\_expr)

: cond\_(std::move(cond)),

then\_expr\_(std::move(then\_expr)),

else\_expr\_(std::move(else\_expr)) {}

llvm::Value\* CodeGen() override;

private:

std::unique\_ptr<ExprAST> cond\_;

std::unique\_ptr<ExprAST> then\_expr\_;

std::unique\_ptr<ExprAST> else\_expr\_;

};

增加对 IfExprAST 的解析：

std::unique\_ptr<ExprAST> ParseIfExpr() {

GetNextToken(); // eat if

std::unique\_ptr<ExprAST> cond = ParseExpression();

GetNextToken(); // eat then

std::unique\_ptr<ExprAST> then\_expr = ParseExpression();

GetNextToken(); // eat else

std::unique\_ptr<ExprAST> else\_expr = ParseExpression();

return std::make\_unique<IfExprAST>(std::move(cond), std::move(then\_expr),

std::move(else\_expr));

}

增加到 ParsePrimary 中：

// primary

// ::= identifierexpr

// ::= numberexpr

// ::= parenexpr

std::unique\_ptr<ExprAST> ParsePrimary() {

switch (g\_current\_token) {

case TOKEN\_IDENTIFIER: return ParseIdentifierExpr();

case TOKEN\_NUMBER: return ParseNumberExpr();

case '(': return ParseParenExpr();

case TOKEN\_IF: return ParseIfExpr();

default: return nullptr;

}

}

完成了 lex 和 parse，接下来是最有意思的 codegen：

llvm::Value\* IfExprAST::CodeGen() {

llvm::Value\* cond\_value = cond\_->CodeGen();

// 创建fcmp one指令, cond\_value = (cond\_value != 0.0)

// 转为1bit (bool)类型

cond\_value = g\_ir\_builder.CreateFCmpONE(

cond\_value, llvm::ConstantFP::get(g\_llvm\_context, llvm::APFloat(0.0)),

"ifcond");

// 在每个function内会创建一个block, 这里一定在这个block内，根据block得到

// 对应的上层function

llvm::Function\* func = g\_ir\_builder.GetInsertBlock()->getParent();

// 为then else以及最后的final创建block

llvm::BasicBlock\* then\_block =

llvm::BasicBlock::Create(g\_llvm\_context, "then", func);

llvm::BasicBlock\* else\_block =

llvm::BasicBlock::Create(g\_llvm\_context, "else");

llvm::BasicBlock\* final\_block =

llvm::BasicBlock::Create(g\_llvm\_context, "ifcont");

// 创建跳转指令，根据cond\_value选择then\_block/else\_block

g\_ir\_builder.CreateCondBr(cond\_value, then\_block, else\_block);

// codegen then\_block, 增加跳转final\_block指令

g\_ir\_builder.SetInsertPoint(then\_block);

llvm::Value\* then\_value = then\_expr\_->CodeGen();

g\_ir\_builder.CreateBr(final\_block);

// then语句内可能会有嵌套的if/then/else, 在嵌套的codegen时，会改变当前的

// InsertBlock, 需要有最终结果的那个block作为这里的then\_block

then\_block = g\_ir\_builder.GetInsertBlock();

// 在这里才加入是为了让这个block位于上面的then里嵌套block的后面

func->getBasicBlockList().push\_back(else\_block);

// 与then类似

g\_ir\_builder.SetInsertPoint(else\_block);

llvm::Value\* else\_value = else\_expr\_->CodeGen();

g\_ir\_builder.CreateBr(final\_block);

else\_block = g\_ir\_builder.GetInsertBlock();

// codegen final

func->getBasicBlockList().push\_back(final\_block);

g\_ir\_builder.SetInsertPoint(final\_block);

llvm::PHINode\* pn = g\_ir\_builder.CreatePHI(

llvm::Type::getDoubleTy(g\_llvm\_context), 2, "iftmp");

pn->addIncoming(then\_value, then\_block);

pn->addIncoming(else\_value, else\_block);

return pn;

}

这里使用了上一节 SSA 中提到的 phi函数，输入：

def foo(x)

if x < 3 then

1

else

foo(x - 1) + foo(x - 2)

foo(1)

foo(2)

foo(3)

foo(4)

得到输出：

parsed a function definition

define double @foo(double %x) {

entry:

%cmptmp = fcmp ult double %x, 3.000000e+00

%booltmp = uitofp i1 %cmptmp to double

%ifcond = fcmp one double %booltmp, 0.000000e+00

br i1 %ifcond, label %then, label %else

then: ; preds = %entry

br label %ifcont

else: ; preds = %entry

%subtmp = fsub double %x, 1.000000e+00

%calltmp = call double @foo(double %subtmp)

%subtmp1 = fsub double %x, 2.000000e+00

%calltmp2 = call double @foo(double %subtmp1)

%addtmp = fadd double %calltmp, %calltmp2

br label %ifcont

ifcont: ; preds = %else, %then

%iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]

ret double %iftmp

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 1.000000e+00)

ret double %calltmp

}

1

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 2.000000e+00)

ret double %calltmp

}

1

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 3.000000e+00)

ret double %calltmp

}

2

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @foo(double 4.000000e+00)

ret double %calltmp

}

3

成功完成了斐波那契数列的计算，接下来需要增加循环的支持，在此之前实现一个 printd 函数：

extern "C" double printd(double x) {

printf("%lf\n", x);

return 0.0;

}

编译：

clang++ -g main.cpp \`llvm-config --cxxflags --ldflags --libs\` -Wl,-no-as-needed -rdynamic

输入：

extern printd(x)

printd(12)

得到输出：

parsed a extern

declare double @printd(double)

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%calltmp = call double @printd(double 1.200000e+01)

ret double %calltmp

}

12.000000

0

可以看到，成功给 Kaleiscope 添加了 printd 函数，接下来看需要实现的循环语法, 使用 C++代码作为注释：

def printstar(n):

for i = 1, i < n, 1.0 in # for (double i = 1.0; i < n; i += 1.0)

printd(n)

同样，增加 for 和 in 的 TOKEN：

enum Token {

TOKEN\_EOF = -1, // 文件结束标识符

TOKEN\_DEF = -2, // 关键字def

TOKEN\_EXTERN = -3, // 关键字extern

TOKEN\_IDENTIFIER = -4, // 名字

TOKEN\_NUMBER = -5, // 数值

TOKEN\_IF = -6, // if

TOKEN\_THEN = -7, // then

TOKEN\_ELSE = -8, // else

TOKEN\_FOR = -9, // for

TOKEN\_IN = -10 // in

};

增加 TOKEN 的识别：

// 识别字符串

if (isalpha(last\_char)) {

g\_identifier\_str = last\_char;

while (isalnum((last\_char = getchar()))) {

g\_identifier\_str += last\_char;

}

if (g\_identifier\_str == "def") {

return TOKEN\_DEF;

} else if (g\_identifier\_str == "extern") {

return TOKEN\_EXTERN;

} else if (g\_identifier\_str == "if") {

return TOKEN\_IF;

} else if (g\_identifier\_str == "then") {

return TOKEN\_THEN;

} else if (g\_identifier\_str == "else") {

return TOKEN\_ELSE;

} else if (g\_identifier\_str == "for") {

return TOKEN\_FOR;

} else if (g\_identifier\_str == "in") {

return TOKEN\_IN;

} else {

return TOKEN\_IDENTIFIER;

}

}

增加 ForExprAST：

// for in

class ForExprAST : public ExprAST {

public:

ForExprAST(const std::string&amp; var\_name, std::unique\_ptr<ExprAST> start\_expr,

std::unique\_ptr<ExprAST> end\_expr,

std::unique\_ptr<ExprAST> step\_expr,

std::unique\_ptr<ExprAST> body\_expr)

: var\_name\_(var\_name),

start\_expr\_(std::move(start\_expr)),

end\_expr\_(std::move(end\_expr)),

step\_expr\_(std::move(step\_expr)),

body\_expr\_(std::move(body\_expr)) {}

llvm::Value\* CodeGen() override;

private:

std::string var\_name\_;

std::unique\_ptr<ExprAST> start\_expr\_;

std::unique\_ptr<ExprAST> end\_expr\_;

std::unique\_ptr<ExprAST> step\_expr\_;

std::unique\_ptr<ExprAST> body\_expr\_;

};

添加到 Primary 的解析中：

// forexpr ::= for var\_name = start\_expr, end\_expr, step\_expr in body\_expr

std::unique\_ptr<ExprAST> ParseForExpr() {

GetNextToken(); // eat for

std::string var\_name = g\_identifier\_str;

GetNextToken(); // eat var\_name

GetNextToken(); // eat =

std::unique\_ptr<ExprAST> start\_expr = ParseExpression();

GetNextToken(); // eat ,

std::unique\_ptr<ExprAST> end\_expr = ParseExpression();

GetNextToken(); // eat ,

std::unique\_ptr<ExprAST> step\_expr = ParseExpression();

GetNextToken(); // eat in

std::unique\_ptr<ExprAST> body\_expr = ParseExpression();

return std::make\_unique<ForExprAST>(var\_name, std::move(start\_expr),

std::move(end\_expr), std::move(step\_expr),

std::move(body\_expr));

}

// primary

// ::= identifierexpr

// ::= numberexpr

// ::= parenexpr

std::unique\_ptr<ExprAST> ParsePrimary() {

switch (g\_current\_token) {

case TOKEN\_IDENTIFIER: return ParseIdentifierExpr();

case TOKEN\_NUMBER: return ParseNumberExpr();

case '(': return ParseParenExpr();

case TOKEN\_IF: return ParseIfExpr();

case TOKEN\_FOR: return ParseForExpr();

default: return nullptr;

}

}

开始 codegen：

llvm::Value\* ForExprAST::CodeGen() {

// codegen start

llvm::Value\* start\_val = start\_expr\_->CodeGen();

// 获取当前function

llvm::Function\* func = g\_ir\_builder.GetInsertBlock()->getParent();

// 保存当前的block

llvm::BasicBlock\* pre\_block = g\_ir\_builder.GetInsertBlock();

// 新增一个loop block到当前function

llvm::BasicBlock\* loop\_block =

llvm::BasicBlock::Create(g\_llvm\_context, "loop", func);

// 为当前block增加到loop\_block的跳转指令

g\_ir\_builder.CreateBr(loop\_block);

// 开始在loop\_block内增加指令

g\_ir\_builder.SetInsertPoint(loop\_block);

llvm::PHINode\* var = g\_ir\_builder.CreatePHI(

llvm::Type::getDoubleTy(g\_llvm\_context), 2, var\_name\_.c\_str());

// 如果来自pre\_block的跳转，则取start\_val的值

var->addIncoming(start\_val, pre\_block);

// 现在新增了一个变量var，因为可能会被后面的代码引用，所以要注册到

// g\_named\_values中，其可能会和函数参数重名，但这里为了方便不管

// 这个特殊情况，直接注册到g\_named\_values中，

g\_named\_values[var\_name\_] = var;

// 在loop\_block中增加body的指令

body\_expr\_->CodeGen();

// codegen step\_expr

llvm::Value\* step\_value = step\_expr\_->CodeGen();

// next\_var = var + step\_value

llvm::Value\* next\_value = g\_ir\_builder.CreateFAdd(var, step\_value, "nextvar");

// codegen end\_expr

llvm::Value\* end\_value = end\_expr\_->CodeGen();

// end\_value = (end\_value != 0.0)

end\_value = g\_ir\_builder.CreateFCmpONE(

end\_value, llvm::ConstantFP::get(g\_llvm\_context, llvm::APFloat(0.0)),

"loopcond");

// 和if/then/else一样，这里的block可能会发生变化，保存当前的block

llvm::BasicBlock\* loop\_end\_block = g\_ir\_builder.GetInsertBlock();

// 创建循环结束后的block

llvm::BasicBlock\* after\_block =

llvm::BasicBlock::Create(g\_llvm\_context, "afterloop", func);

// 根据end\_value选择是再来一次loop\_block还是进入after\_block

g\_ir\_builder.CreateCondBr(end\_value, loop\_block, after\_block);

// 给after\_block增加指令

g\_ir\_builder.SetInsertPoint(after\_block);

// 如果是再次循环，取新的值

var->addIncoming(next\_value, loop\_end\_block);

// 循环结束，避免被再次引用

g\_named\_values.erase(var\_name\_);

// return 0

return llvm::Constant::getNullValue(llvm::Type::getDoubleTy(g\_llvm\_context));

}

输入：

extern printd(x)

def foo(x)

if x < 3 then

1

else

foo(x - 1) + foo(x - 2)

for i = 1, i < 10, 1.0 in

printd(foo(i))

输出：

parsed a extern

declare double @printd(double)

parsed a function definition

define double @foo(double %x) {

entry:

%cmptmp = fcmp ult double %x, 3.000000e+00

%booltmp = uitofp i1 %cmptmp to double

%ifcond = fcmp one double %booltmp, 0.000000e+00

br i1 %ifcond, label %then, label %else

then: ; preds = %entry

br label %ifcont

else: ; preds = %entry

%subtmp = fsub double %x, 1.000000e+00

%calltmp = call double @foo(double %subtmp)

%subtmp1 = fsub double %x, 2.000000e+00

%calltmp2 = call double @foo(double %subtmp1)

%addtmp = fadd double %calltmp, %calltmp2

br label %ifcont

ifcont: ; preds = %else, %then

%iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ]

ret double %iftmp

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

br label %loop

loop: ; preds = %loop, %entry

%i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]

%calltmp = call double @foo(double %i)

%calltmp1 = call double @printd(double %calltmp)

%nextvar = fadd double %i, 1.000000e+00

%cmptmp = fcmp ult double %i, 1.000000e+01

%booltmp = uitofp i1 %cmptmp to double

%loopcond = fcmp one double %booltmp, 0.000000e+00

br i1 %loopcond, label %loop, label %afterloop

afterloop: ; preds = %loop

ret double 0.000000e+00

}

1.000000

1.000000

2.000000

3.000000

5.000000

8.000000

13.000000

21.000000

34.000000

55.000000

0

成功遍历了斐波那契数列。

**9 用户定义运算符**

在 C++中，用户可以重载操作符而不能增加操作符。在这里，将给 Kaleidoscope 增加一个功能，让用户可以增加二元操作符。

# 新增二元操作符 `>`, 优先级等于内置的 `<`

def binary> 10 (LHS RHS)

RHS < LHS

# 新增二元操作符 `|`, 优先级为5

def binary| 5 (LHS RHS)

if LHS then

1

else if RHS then

1

else

0

# 新增二元操作符 `=`，优先级为9，这个操作符类似C++的 `==`

def binary= 9 (LHS RHS)

!(LHS < RHS | LHS > RHS)

增加 TOKEN 的类型：

enum Token {

...

TOKEN\_BINARY = -11, // binary

};

增加 TOKEN 的识别：

// 从标准输入解析一个Token并返回

int GetToken() {

...

// 识别字符串

if (isalpha(last\_char)) {

...

if (g\_identifier\_str == "def") {

return TOKEN\_DEF;

} else if (g\_identifier\_str == "extern") {

return TOKEN\_EXTERN;

} else if (g\_identifier\_str == "if") {

return TOKEN\_IF;

} else if (g\_identifier\_str == "then") {

return TOKEN\_THEN;

} else if (g\_identifier\_str == "else") {

return TOKEN\_ELSE;

} else if (g\_identifier\_str == "for") {

return TOKEN\_FOR;

} else if (g\_identifier\_str == "in") {

return TOKEN\_IN;

} else if (g\_identifier\_str == "binary") {

return TOKEN\_BINARY;

} else {

return TOKEN\_IDENTIFIER;

}

}

...

}

把新增的二元操作符视为一个函数，所以不需要新增 AST，但是需要修改 PrototypeAST。

// 函数接口

class PrototypeAST {

public:

PrototypeAST(const std::string&amp; name, std::vector<std::string> args,

bool is\_operator = false, int op\_precedence = 0)

: name\_(name),

args\_(std::move(args)),

is\_operator\_(is\_operator),

op\_precedence\_(op\_precedence) {}

llvm::Function\* CodeGen();

const std::string&amp; name() const { return name\_; }

int op\_precedence() const { return op\_precedence\_; }

bool IsUnaryOp() const { return is\_operator\_ &amp;&amp; args\_.size() == 1; }

bool IsBinaryOp() const { return is\_operator\_ &amp;&amp; args\_.size() == 2; }

// like `|` in `binary|`

char GetOpName() { return name\_[name\_.size() - 1]; }

private:

std::string name\_;

std::vector<std::string> args\_;

bool is\_operator\_;

int op\_precedence\_;

};

修改语法分析部分：

// prototype

// ::= id ( id id ... id)

// ::= binary binop precedence (id id)

std::unique\_ptr<PrototypeAST> ParsePrototype() {

std::string function\_name;

bool is\_operator = false;

int precedence = 0;

switch (g\_current\_token) {

case TOKEN\_IDENTIFIER: {

function\_name = g\_identifier\_str;

is\_operator = false;

GetNextToken(); // eat id

break;

}

case TOKEN\_BINARY: {

GetNextToken(); // eat binary

function\_name = "binary";

function\_name += (char)(g\_current\_token);

is\_operator = true;

GetNextToken(); // eat binop

precedence = g\_number\_val;

GetNextToken(); // eat precedence

break;

}

}

std::vector<std::string> arg\_names;

while (GetNextToken() == TOKEN\_IDENTIFIER) {

arg\_names.push\_back(g\_identifier\_str);

}

GetNextToken(); // eat )

return std::make\_unique<PrototypeAST>(function\_name, arg\_names, is\_operator,

precedence);

}

修改 BinaryExprAST 的 CodeGen 处理自定义运算符, 增加函数调用指令：

llvm::Value\* BinaryExprAST::CodeGen() {

llvm::Value\* lhs = lhs\_->CodeGen();

llvm::Value\* rhs = rhs\_->CodeGen();

switch (op\_) {

case '<': {

llvm::Value\* tmp = g\_ir\_builder.CreateFCmpULT(lhs, rhs, "cmptmp");

// 把 0/1 转为 0.0/1.0

return g\_ir\_builder.CreateUIToFP(

tmp, llvm::Type::getDoubleTy(g\_llvm\_context), "booltmp");

}

case '+': return g\_ir\_builder.CreateFAdd(lhs, rhs, "addtmp");

case '-': return g\_ir\_builder.CreateFSub(lhs, rhs, "subtmp");

case '\*': return g\_ir\_builder.CreateFMul(lhs, rhs, "multmp");

default: {

// user defined operator

llvm::Function\* func = GetFunction(std::string("binary") + op\_);

llvm::Value\* operands[2] = {lhs, rhs};

return g\_ir\_builder.CreateCall(func, operands, "binop");

}

}

}

在 FunctionAST 的 CodeGen 时，注册操作符优先级，从而让自定义操作符被识别为操作符。

llvm::Value\* FunctionAST::CodeGen() {

PrototypeAST&amp; proto = \*proto\_;

name2proto\_ast[proto.name()] = std::move(proto\_); // transfer ownership

llvm::Function\* func = GetFunction(proto.name());

if (proto.IsBinaryOp()) {

g\_binop\_precedence[proto.GetOpName()] = proto.op\_precedence();

}

// 创建一个Block并且设置为指令插入位置。

// llvm block用于定义control flow graph, 由于暂不实现control flow, 创建

// 一个单独的block即可

llvm::BasicBlock\* block =

llvm::BasicBlock::Create(g\_llvm\_context, "entry", func);

g\_ir\_builder.SetInsertPoint(block);

// 将函数参数注册到g\_named\_values中，让VariableExprAST可以codegen

g\_named\_values.clear();

for (llvm::Value&amp; arg : func->args()) {

g\_named\_values[arg.getName()] = &amp;arg;

}

// codegen body然后return

llvm::Value\* ret\_val = body\_->CodeGen();

g\_ir\_builder.CreateRet(ret\_val);

llvm::verifyFunction(\*func);

return func;

}

输入：

# 新增二元操作符 `>`, 优先级等于内置的 `<`

def binary> 10 (LHS RHS)

RHS < LHS

1 > 2

2 > 1

# 新增二元操作符 `|`, 优先级为5

def binary| 5 (LHS RHS)

if LHS then

1

else if RHS then

1

else

0

1 | 0

0 | 1

0 | 0

1 | 1

得到输出：

parsed a function definition

define double @"binary>"(double %LHS, double %RHS) {

entry:

%cmptmp = fcmp ult double %RHS, %LHS

%booltmp = uitofp i1 %cmptmp to double

ret double %booltmp

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%binop = call double @"binary>"(double 1.000000e+00, double 2.000000e+00)

ret double %binop

}

0

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%binop = call double @"binary>"(double 2.000000e+00, double 1.000000e+00)

ret double %binop

}

1

parsed a function definition

define double @"binary|"(double %LHS, double %RHS) {

entry:

%ifcond = fcmp one double %LHS, 0.000000e+00

br i1 %ifcond, label %then, label %else

then: ; preds = %entry

br label %ifcont4

else: ; preds = %entry

%ifcond1 = fcmp one double %RHS, 0.000000e+00

br i1 %ifcond1, label %then2, label %else3

then2: ; preds = %else

br label %ifcont

else3: ; preds = %else

br label %ifcont

ifcont: ; preds = %else3, %then2

%iftmp = phi double [ 1.000000e+00, %then2 ], [ 0.000000e+00, %else3 ]

br label %ifcont4

ifcont4: ; preds = %ifcont, %then

%iftmp5 = phi double [ 1.000000e+00, %then ], [ %iftmp, %ifcont ]

ret double %iftmp5

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%binop = call double @"binary|"(double 1.000000e+00, double 0.000000e+00)

ret double %binop

}

1

解析顶级表达式

define double @\_\_anon\_expr() {

entry:

%binop = call double @"binary|"(double 0.000000e+00, double 1.000000e+00)

ret double %binop

}

1

解析顶级表达式

define double @\_\_anon\_expr() {

entry:

%binop = call double @"binary|"(double 0.000000e+00, double 0.000000e+00)

ret double %binop

}

0

解析顶级表达式

define double @\_\_anon\_expr() {

entry:

%binop = call double @"binary|"(double 1.000000e+00, double 1.000000e+00)

ret double %binop

}

1

**10 可变变量**

将让 Kaleidoscope 支持可变变量，首先看如下 C 代码：

int G, H;

int test(\_Bool Condition) {

int X;

if (Condition)

X = G;

else

X = H;

return X;

}

由于变量 X 的值依赖于程序的执行路径，会加入一个 phi node 来选取分支结果。上面代码的 LLVM IR 如下：

@G = weak global i32 0 ; type of @G is i32\*

@H = weak global i32 0 ; type of @H is i32\*

define i32 @test(i1 %Condition) {

entry:

br i1 %Condition, label %cond\_true, label %cond\_false

cond\_true:

%X.0 = load i32\* @G

br label %cond\_next

cond\_false:

%X.1 = load i32\* @H

br label %cond\_next

cond\_next:

%X.2 = phi i32 [ %X.1, %cond\_false ], [ %X.0, %cond\_true ]

ret i32 %X.2

}

上面的 X 是符合 SSA 格式的，但是这里真正的难题是给可变变量赋值时怎么自动添加 phi node。先了解一些信息，LLVM 要求寄存器变量是 SSA 格式，但却不允许内存对象是 SSA 格式。比如上面的例子中，G 和 H 就没有版本号。在 LLVM 中，所有内存访问都是显示的 load/store 指令，并且不存在取内存地址的操作。注意上面的例子中，即使@G/@H 全局变量定义时用的 i32, 但其类型仍然是 i32\*, 表示在全局数据区存放 i32 的空间地址。

现在假设想创建一个类似@G 但是在栈上的内存变量，基本指令如下：

define i32 @example() {entry:

%X = alloca i32 ; type of %X is i32\*.

...

%tmp = load i32\* %X ; load the stack value %X from the stack.

%tmp2 = add i32 %tmp, 1 ; increment it

store i32 %tmp2, i32\* %X ; store it back

...

于是可以把上面使用 phi 节点的 LLVM IR 改写为使用栈上变量：

@G = weak global i32 0 ; type of @G is i32\*

@H = weak global i32 0 ; type of @H is i32\*

define i32 @test(i1 %Condition) {

entry:

%X = alloca i32 ; type of %X is i32\*.

br i1 %Condition, label %cond\_true, label %cond\_false

cond\_true:

%X.0 = load i32\* @G

store i32 %X.0, i32\* %X ; Update X

br label %cond\_next

cond\_false:

%X.1 = load i32\* @H

store i32 %X.1, i32\* %X ; Update X

br label %cond\_next

cond\_next:

%X.2 = load i32\* %X ; Read X

ret i32 %X.2

}

于是找到了一个处理任意可变变量而且不需要创建 phi node 的办法：

每个可变变量在栈上创建

变量读取变为从堆栈加载

变量更新变为存储到堆栈

使用栈上地址作为变量地址

但是这会带来一个新的问题，因为内存速度不如寄存器，大量使用栈会有性能问题。不过，LLVM 优化器有一个 pass 称为"mem2reg", 专门将 stack 的使用自动地尽可能转为使用 phi node, 下面为自动优化的结果：

@G = weak global i32 0

@H = weak global i32 0

define i32 @test(i1 %Condition) {

entry:

br i1 %Condition, label %cond\_true, label %cond\_false

cond\_true:

%X.0 = load i32\* @G

br label %cond\_next

cond\_false:

%X.1 = load i32\* @H

br label %cond\_next

cond\_next:

%X.01 = phi i32 [ %X.1, %cond\_false ], [ %X.0, %cond\_true ]

ret i32 %X.01}

mem2reg 实现了一个称为迭代优势边界的标准算法来自动创建 SSA 格式。对 mem2reg 的使用需要注意：

mem2reg 只能优化栈上变量，不会优化全局变量和堆上变量；

mem2reg 只优化输入块中的栈上变量创建, 因为在输入块中就意味着只创建一次；

如果对栈上变量有 load 和 store 之外的操作, mem2reg 也不会优化；

mem2reg 只能优化基本类型的栈上变量，比如指针，数值和数组。其中数组的大小必须为 1. 对于结构体和数组等的优化需要另一个称为"sroa"的 pass。

因为后面需要启用 mem2reg，先把优化器加回来，修改全局定义：

std::unique\_ptr<llvm::Module> g\_module;

std::unique\_ptr<llvm::legacy::FunctionPassManager> g\_fpm;

修改 ReCreateModule：

void ReCreateModule() {

g\_module = std::make\_unique<llvm::Module>("my cool jit", g\_llvm\_context);

g\_module->setDataLayout(g\_jit->getTargetMachine().createDataLayout());

g\_fpm = std::make\_unique<llvm::legacy::FunctionPassManager>(g\_module.get());

g\_fpm->add(llvm::createInstructionCombiningPass());

g\_fpm->add(llvm::createReassociatePass());

g\_fpm->add(llvm::createGVNPass());

g\_fpm->add(llvm::createCFGSimplificationPass());

g\_fpm->doInitialization();

}

在 FunctionAST::CodeGen 中执行优化器：

g\_ir\_builder.CreateRet(ret\_val);

llvm::verifyFunction(\*func);

g\_fpm->run(\*func);

修改 main：

int main() {

llvm::InitializeNativeTarget();

llvm::InitializeNativeTargetAsmPrinter();

llvm::InitializeNativeTargetAsmParser();

g\_jit.reset(new llvm::orc::KaleidoscopeJIT);

ReCreateModule();

...

}

有两种类型的变量，分别是函数参数以及 for 循环的变量，这里将这两种变量也修改为使用内存，再让 mem2reg 进行优化。因为所有的变量都会使用内存，修改 g\_named\_value 存储的类型为 AllocaInst\*：

std::map<std::string, llvm::AllocaInst\*> g\_named\_values;

编写一个函数 CreateEntryBlockAlloca，简化后续工作，其功能是往函数的 EntryBlock 的最开始的地方添加分配内存指令：

llvm::AllocaInst\* CreateEntryBlockAlloca(llvm::Function\* func,

const std::string&amp; var\_name) {

llvm::IRBuilder<> ir\_builder(&amp;(func->getEntryBlock()),

func->getEntryBlock().begin());

return ir\_builder.CreateAlloca(llvm::Type::getDoubleTy(g\_llvm\_context), 0,

var\_name.c\_str());

}

修改 VariableExprAST::CodeGen, 由于所有变量都放在内存上，所以增加 load 指令：

llvm::Value\* VariableExprAST::CodeGen() {

llvm::AllocaInst\* val = g\_named\_values.at(name\_);

return g\_ir\_builder.CreateLoad(val, name\_.c\_str());

}

接下来修改 for 循环里变量的 CodeGen：

llvm::Value\* ForExprAST::CodeGen() {

// 获取当前function

llvm::Function\* func = g\_ir\_builder.GetInsertBlock()->getParent();

// 将变量创建为栈上变量，不再是phi node

llvm::AllocaInst\* var = CreateEntryBlockAlloca(func, var\_name\_);

// codegen start

llvm::Value\* start\_val = start\_expr\_->CodeGen();

// 将初始值赋给var

g\_ir\_builder.CreateStore(start\_val, var);

// 新增一个loop block到当前function

llvm::BasicBlock\* loop\_block =

llvm::BasicBlock::Create(g\_llvm\_context, "loop", func);

// 为当前block增加到loop\_block的跳转指令

g\_ir\_builder.CreateBr(loop\_block);

// 开始在loop\_block内增加指令

g\_ir\_builder.SetInsertPoint(loop\_block);

// 现在新增了一个变量var，因为可能会被后面的代码引用，所以要注册到

// g\_named\_values中，其可能会和函数参数重名，但这里为了方便不管

// 这个特殊情况，直接注册到g\_named\_values中，

g\_named\_values[var\_name\_] = var;

// 在loop\_block中增加body的指令

body\_expr\_->CodeGen();

// codegen step\_expr

llvm::Value\* step\_value = step\_expr\_->CodeGen();

// var = var + step\_value

llvm::Value\* cur\_value = g\_ir\_builder.CreateLoad(var);

llvm::Value\* next\_value =

g\_ir\_builder.CreateFAdd(cur\_value, step\_value, "nextvar");

g\_ir\_builder.CreateStore(next\_value, var);

// codegen end\_expr

llvm::Value\* end\_value = end\_expr\_->CodeGen();

// end\_value = (end\_value != 0.0)

end\_value = g\_ir\_builder.CreateFCmpONE(

end\_value, llvm::ConstantFP::get(g\_llvm\_context, llvm::APFloat(0.0)),

"loopcond");

// 和if/then/else一样，这里的block可能会发生变化，保存当前的block

llvm::BasicBlock\* loop\_end\_block = g\_ir\_builder.GetInsertBlock();

// 创建循环结束后的block

llvm::BasicBlock\* after\_block =

llvm::BasicBlock::Create(g\_llvm\_context, "afterloop", func);

// 根据end\_value选择是再来一次loop\_block还是进入after\_block

g\_ir\_builder.CreateCondBr(end\_value, loop\_block, after\_block);

// 给after\_block增加指令

g\_ir\_builder.SetInsertPoint(after\_block);

// 循环结束，避免被再次引用

g\_named\_values.erase(var\_name\_);

// return 0

return llvm::Constant::getNullValue(llvm::Type::getDoubleTy(g\_llvm\_context));

}

修改 FunctionAST::codegen()使得参数可变：

llvm::Value\* FunctionAST::CodeGen() {

PrototypeAST&amp; proto = \*proto\_;

name2proto\_ast[proto.name()] = std::move(proto\_); // transfer ownership

llvm::Function\* func = GetFunction(proto.name());

if (proto.IsBinaryOp()) {

g\_binop\_precedence[proto.GetOpName()] = proto.op\_precedence();

}

// 创建一个Block并且设置为指令插入位置。

// llvm block用于定义control flow graph, 由于暂不实现control flow, 创建

// 一个单独的block即可

llvm::BasicBlock\* block =

llvm::BasicBlock::Create(g\_llvm\_context, "entry", func);

g\_ir\_builder.SetInsertPoint(block);

// 将函数参数注册到g\_named\_values中，让VariableExprAST可以codegen

g\_named\_values.clear();

for (llvm::Value&amp; arg : func->args()) {

// 为每个参数创建一个栈上变量，并赋初值，修改g\_named\_values使得后面的引用

// 会引用这个栈上变量

llvm::AllocaInst\* var = CreateEntryBlockAlloca(func, arg.getName());

g\_ir\_builder.CreateStore(&amp;arg, var);

g\_named\_values[arg.getName()] = var;

}

// codegen body然后return

llvm::Value\* ret\_val = body\_->CodeGen();

g\_ir\_builder.CreateRet(ret\_val);

llvm::verifyFunction(\*func);

g\_fpm->run(\*func);

return func;

}

输入：

extern printd(x)

def foo(x)

if x < 3 then

1

else

foo(x - 1) + foo(x - 2)

for i = 1, i < 10, 1.0 in

printd(foo(i))

输出：

parsed a extern [13/48988]

declare double @printd(double)

parsed a function definition

define double @foo(double %x) {

entry:

%x1 = alloca double, align 8

store double %x, double\* %x1, align 8

%cmptmp = fcmp ult double %x, 3.000000e+00

br i1 %cmptmp, label %ifcont, label %else

else: ; preds = %entry

%subtmp = fadd double %x, -1.000000e+00

%calltmp = call double @foo(double %subtmp)

%subtmp5 = fadd double %x, -2.000000e+00

%calltmp6 = call double @foo(double %subtmp5)

%addtmp = fadd double %calltmp, %calltmp6

br label %ifcont

ifcont: ; preds = %entry, %else

%iftmp = phi double [ %addtmp, %else ], [ 1.000000e+00, %entry ]

ret double %iftmp

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

%i = alloca double, align 8

store double 1.000000e+00, double\* %i, align 8

br label %loop

loop: ; preds = %loop, %entry

%i1 = phi double [ %nextvar, %loop ], [ 1.000000e+00, %entry ]

%calltmp = call double @foo(double %i1)

%calltmp2 = call double @printd(double %calltmp)

%nextvar = fadd double %i1, 1.000000e+00

store double %nextvar, double\* %i, align 8

%cmptmp = fcmp ult double %nextvar, 1.000000e+01

br i1 %cmptmp, label %loop, label %afterloop

afterloop: ; preds = %loop

ret double 0.000000e+00

}

1.000000

1.000000

2.000000

3.000000

5.000000

8.000000

13.000000

21.000000

34.000000

0

可以看到，新版本的 IR 中已经没有了 phi node, 接下来加入优化器：

g\_fpm->add(llvm::createPromoteMemoryToRegisterPass());

g\_fpm->add(llvm::createInstructionCombiningPass());

g\_fpm->add(llvm::createReassociatePass());

再次得到输出：

parsed a extern

declare double @printd(double)

parsed a function definition

define double @foo(double %x) {

entry:

%cmptmp = fcmp ult double %x, 3.000000e+00

br i1 %cmptmp, label %ifcont, label %else

else: ; preds = %entry

%subtmp = fadd double %x, -1.000000e+00

%calltmp = call double @foo(double %subtmp)

%subtmp5 = fadd double %x, -2.000000e+00

%calltmp6 = call double @foo(double %subtmp5)

%addtmp = fadd double %calltmp, %calltmp6

br label %ifcont

ifcont: ; preds = %entry, %else

%iftmp = phi double [ %addtmp, %else ], [ 1.000000e+00, %entry ]

ret double %iftmp

}

parsed a top level expr

define double @\_\_anon\_expr() {

entry:

br label %loop

loop: ; preds = %loop, %entry

%i1 = phi double [ %nextvar, %loop ], [ 1.000000e+00, %entry ]

%calltmp = call double @foo(double %i1)

%calltmp2 = call double @printd(double %calltmp)

%nextvar = fadd double %i1, 1.000000e+00

%cmptmp = fcmp ult double %nextvar, 1.000000e+01

br i1 %cmptmp, label %loop, label %afterloop

afterloop: ; preds = %loop

ret double 0.000000e+00

}

1.000000

1.000000

2.000000

3.000000

5.000000

8.000000

13.000000

21.000000

34.000000

0

可以看到，栈上变量自动地变为寄存器变量，且 phi node 自动地被添加。