Kaleidoscope

代码解释(6/8)

万花筒语言 - LLVM 新手入门教程

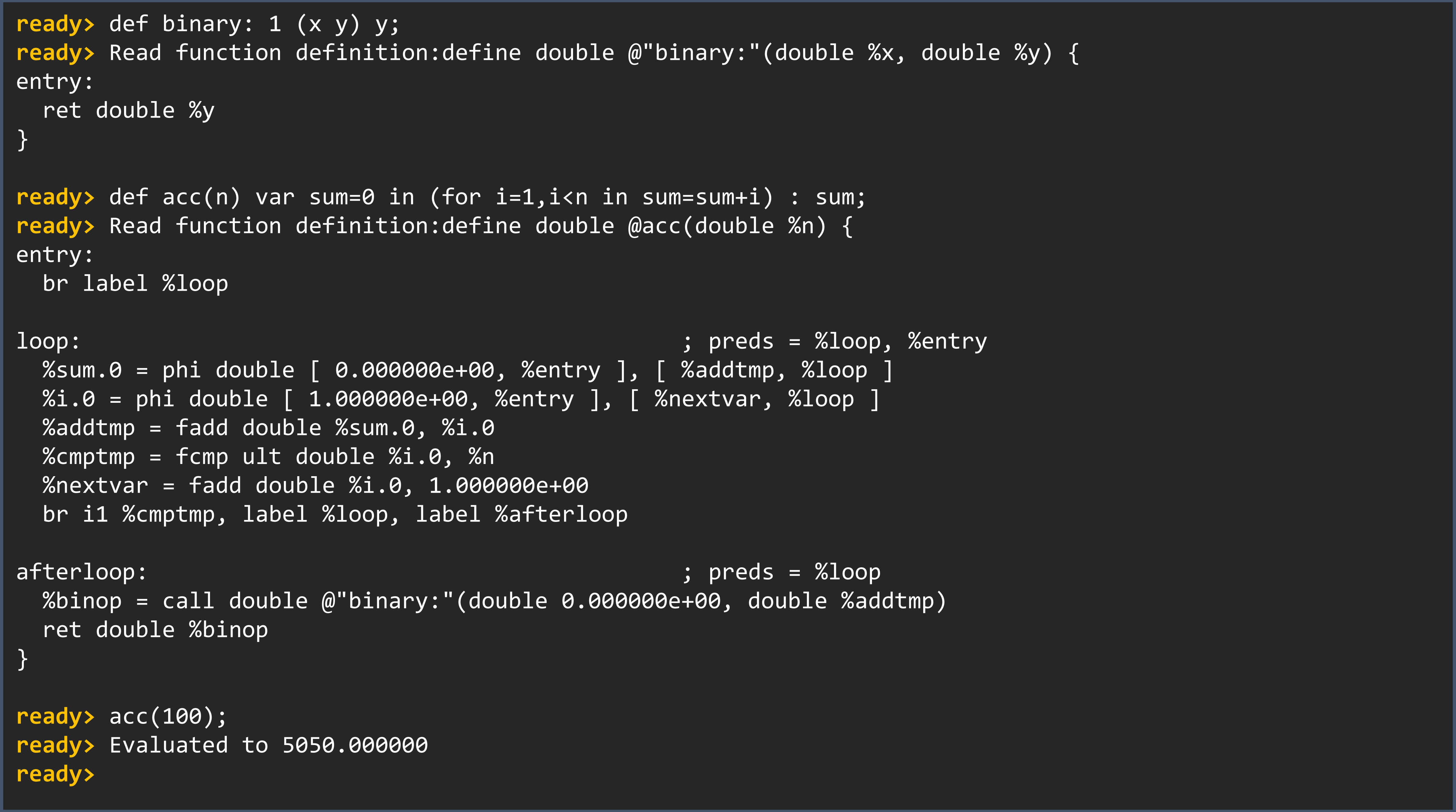
https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl07.html

PLCT - SSC

进度说明

- Kaleidoscope前端系列共 8 期
- Building a JIT共 5 期
  - This tutorial is currently being updated to account for ORC API changes. Only Chapters 1 and 2 are up-to-date.
  - Example code from Chapters 3 to 5 will compile and run, but has not been updated

语言拓展：可变变量



```
ready> def binary: 1 (x y) y;
ready> Read function definition:define double @"binary:"(double %x, double %y) {
entry:
  ret double %y
}

ready> def acc(n) var sum=0 in (for i=1,i<n in sum=sum+i) : sum;
ready> Read function definition:define double @acc(double %n) {
entry:
  br label %loop

loop:                                             ; preds = %loop, %entry
  %sum.0 = phi double [ 0.000000e+00, %entry ], [ %addtmp, %loop ]
  %i.0 = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]
  %addtmp = fadd double %sum.0, %i.0
  %cmptmp = fcmp ult double %i.0, %n
  %nextvar = fadd double %i.0, 1.000000e+00
  br i1 %cmptmp, label %loop, label %afterloop

afterloop:                                        ; preds = %loop
  %binop = call double @"binary:"(double 0.000000e+00, double %addtmp)
  ret double %binop
}

ready> acc(100);
ready> Evaluated to 5050.000000
ready>
```

```
<result> = alloca [inalloca] <type> [, <ty> <NumElements>] [, align <alignment>] [, addrspace(<num>)]
; yields type addrspace(num)*:result
```

alloca指令为当前的函数分配stack上的内存空间,函数结束时自动释放

头文件、词法分析、AST

```
#include “llvm/Transforms/Utils.h” // Utils transformations库
```

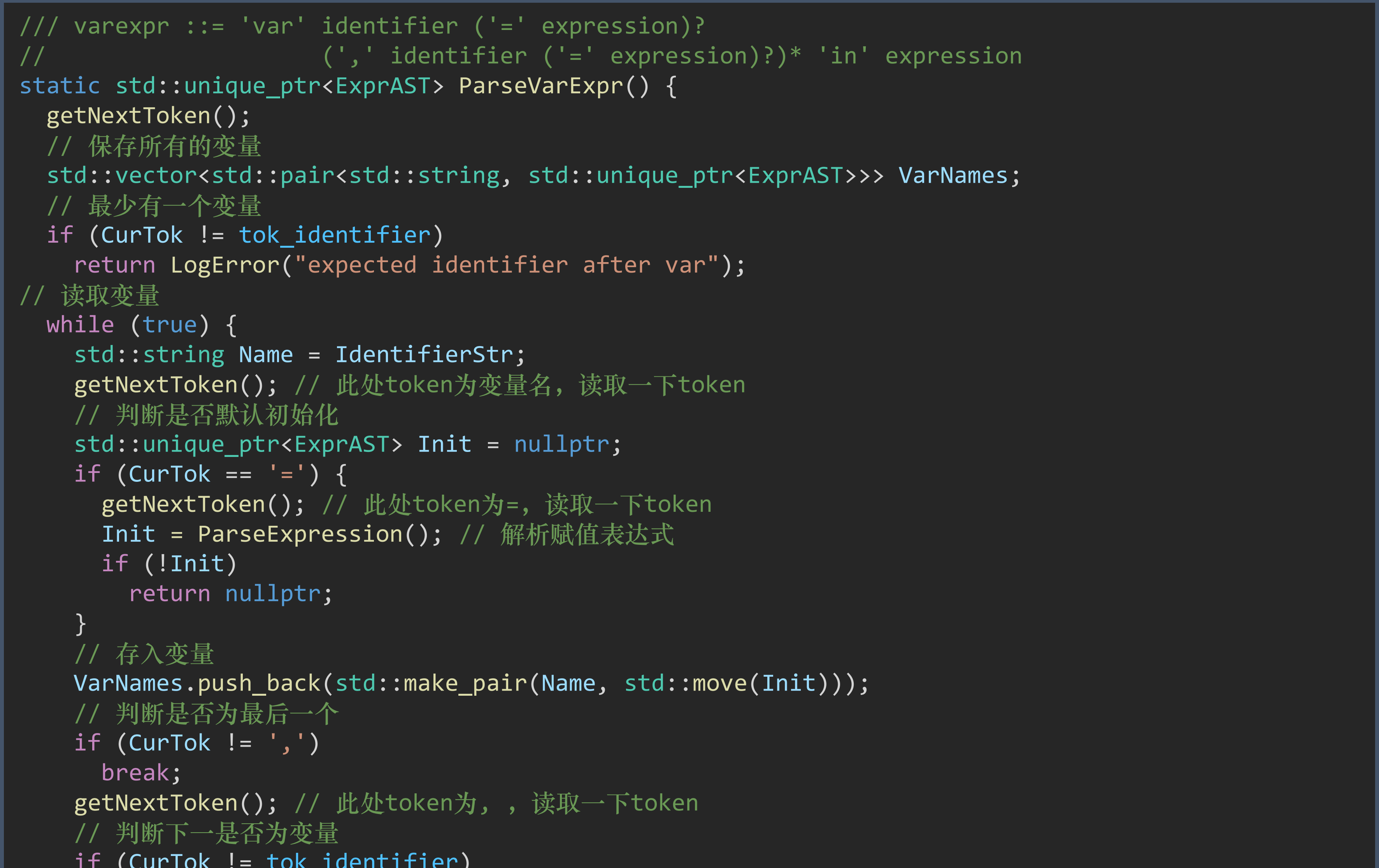
```
enum Token {

  // var definition
  tok_var = -13
};
```

```
static int gettok() {

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    if (IdentifierStr == "var")
      return tok_var;
    return tok_identifier;
  }
  LastChar = getchar();
  return ThisChar;
}
```

```
class NumberExprAST : public ExprAST {
  const std::string &getName() const { return Name; }
};
```

```
class VarExprAST : public ExprAST {
  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;
  std::unique_ptr<ExprAST> Body;

public:
  VarExprAST(
      std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames,
      std::unique_ptr<ExprAST> Body)
      : VarNames(std::move(VarNames)), Body(std::move(Body)) {}

  Value *codegen() override;
};
```

语法解析



```
/// varexpr ::= 'var' identifier ('=' expression)?
//                    (',' identifier ('=' expression)?)* 'in' expression
static std::unique_ptr<ExprAST> ParseVarExpr() {
  getNextToken();
  // 保存所有的变量
  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;
  // 最少有一个变量
  if (CurTok != tok_identifier)
    return LogError("expected identifier after var");
// 读取变量
  while (true) {
    std::string Name = IdentifierStr;
    getNextToken(); // 此处token为变量名，读取一下token
    // 判断是否默认初始化
    std::unique_ptr<ExprAST> Init = nullptr;
    if (CurTok == '=') {
      getNextToken(); // 此处token为=，读取一下token
      Init = ParseExpression(); // 解析赋值表达式
      if (!Init)
        return nullptr;
    }
    // 存入变量
    VarNames.push_back(std::make_pair(Name, std::move(Init)));
    // 判断是否为最后一个
    if (CurTok != ',')
      break;
    getNextToken(); // 此处token为，，读取一下token
    // 判断下一是否为变量
    if (CurTok != tok_identifier)
    }
    // 存入变量
    VarNames.push_back(std::make_pair(Name, std::move(Init)));
    // 判断是否为最后一个
    if (CurTok != ',')
      break;
    getNextToken(); // 此处token为，，读取一下token
    // 判断下一是否为变量
    if (CurTok != tok_identifier)
      return LogError("expected identifier list after var");
  }
  // 变量读取结束，检测in
  if (CurTok != tok_in)
    return LogError("expected 'in' keyword after 'var'");
  getNextToken(); // // 此处token为in，读取一下token
  // 解析主体部分
  auto Body = ParseExpression();
  if (!Body)
    return nullptr;
  // 返回生成的AST
  return std::make_unique<VarExprAST>(std::move(VarNames), std::move(Body));
}
```

```
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  case tok_var:
    return ParseVarExpr();
  }
}
```

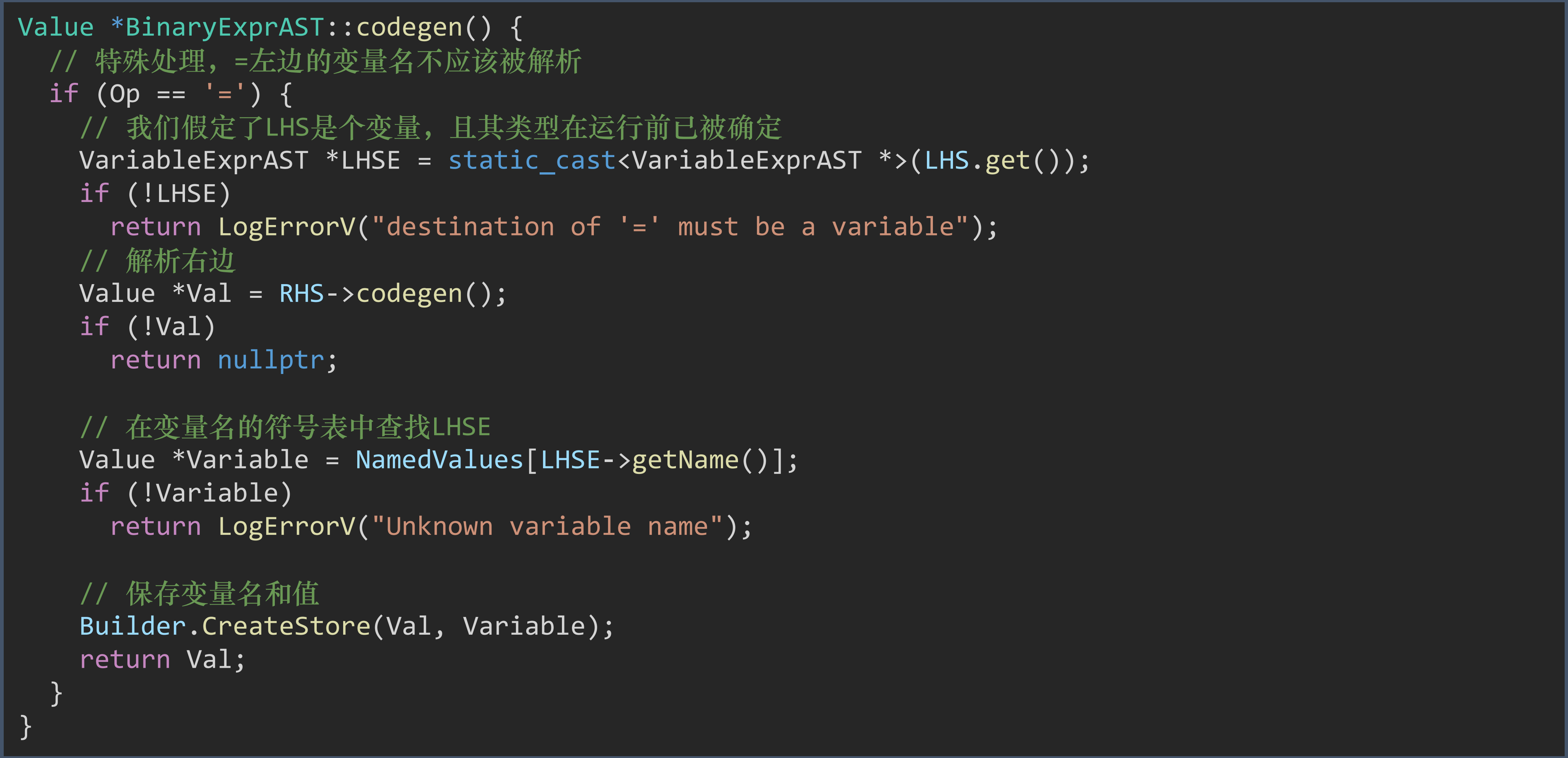
代码生成

```
static std::map<std::string, Value *> NamedValues;
static std::map<std::string, AllocaInst *> NamedValues;
// Alloca是在内存上的stack分配空间
```

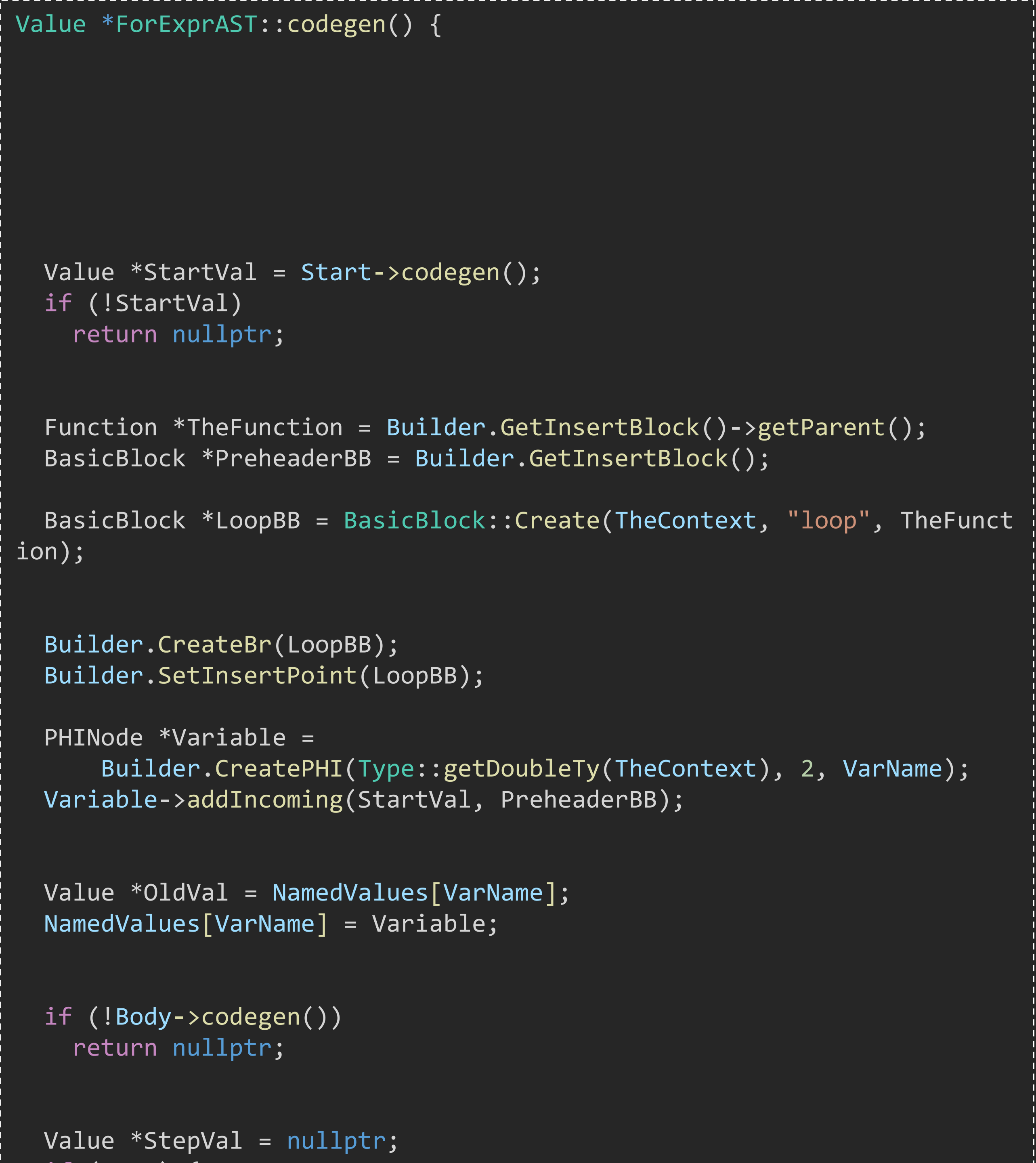
```
// 为可变变量的Alloca指令在entry块中生成代码
static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction,
                                          StringRef VarName) {
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                   TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(TheContext), nullptr, VarName);
}
```

```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (!V)
    return LogErrorV("Unknown variable name");
  return V;
}
```

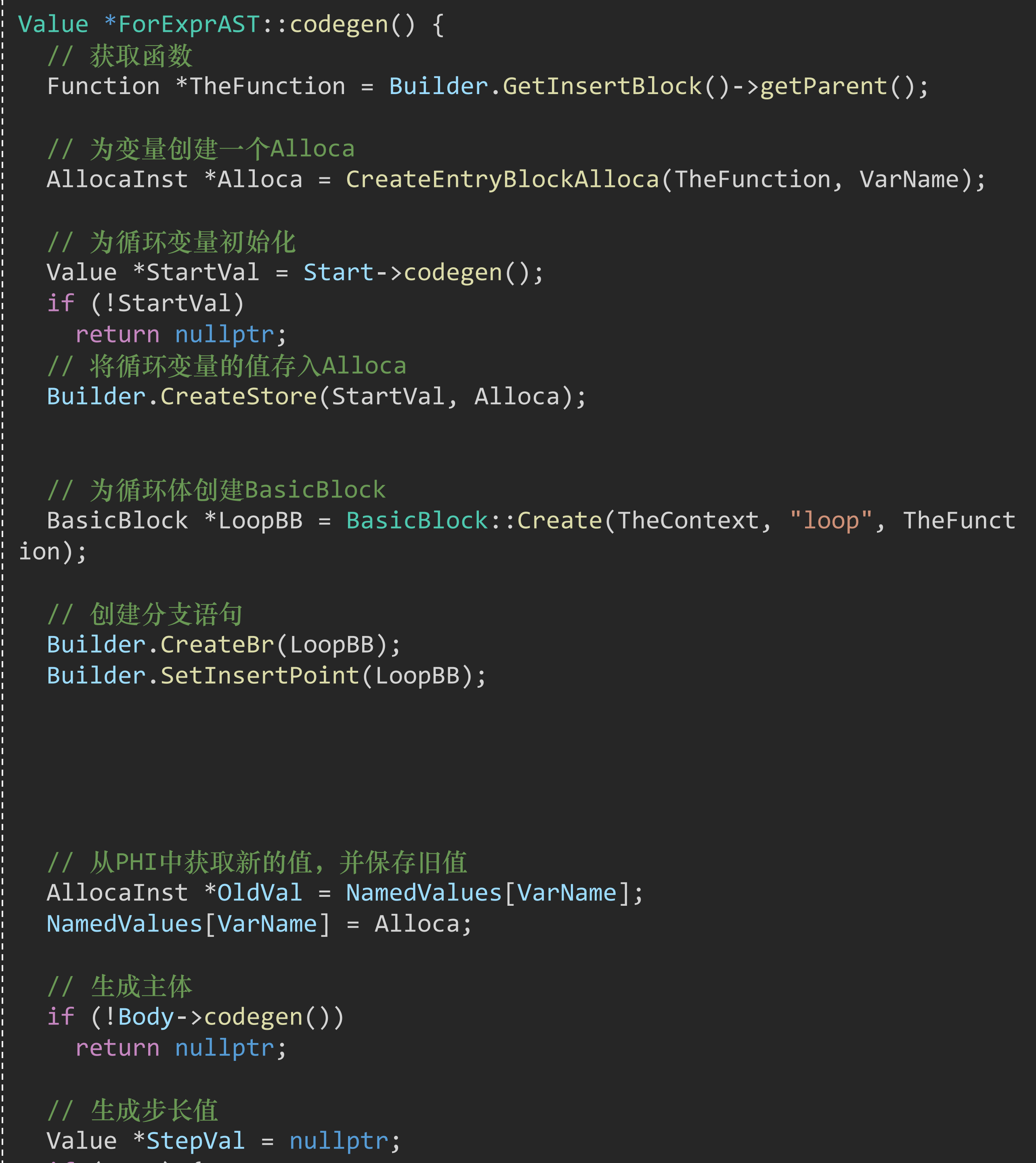
```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (!V)
    return LogErrorV("Unknown variable name");

  // Load the value.
  return Builder.CreateLoad(V, Name.c_str());
}
```



```
Value *BinaryExprAST::codegen() {
  // 特殊处理，=左边的变量名不应该被解析
  if (Op == '=') {
    // 我们假定了LHS是个变量，且其类型在运行前已被确定
    VariableExprAST *LHSE = static_cast<VariableExprAST *>(LHS.get());
    if (!LHSE)
      return LogErrorV("destination of '=' must be a variable");
    // 解析右边
    Value *Val = RHS->codegen();
    if (!Val)
      return nullptr;

    // 在变量名的符号表中查找LHSE
    Value *Variable = NamedValues[LHSE->getName()];
    if (!Variable)
      return LogErrorV("Unknown variable name");

    // 保存变量名和值
    Builder.CreateStore(Val, Variable);
    return Val;
  }
}
```



```
Value *ForExprAST::codegen() {







  Value *StartVal = Start->codegen();
  if (!StartVal)
    return nullptr;


  Function *TheFunction = Builder.GetInsertBlock()->getParent();
  BasicBlock *PreheaderBB = Builder.GetInsertBlock();

  BasicBlock *LoopBB = BasicBlock::Create(TheContext, "loop", TheFunct
ion);


  Builder.CreateBr(LoopBB);
  Builder.SetInsertPoint(LoopBB);

  PHINode *Variable =
      Builder.CreatePHI(Type::getDoubleTy(TheContext), 2, VarName);
  Variable->addIncoming(StartVal, PreheaderBB);


  Value *OldVal = NamedValues[VarName];
  NamedValues[VarName] = Variable;


  if (!Body->codegen())
    return nullptr;


  Value *StepVal = nullptr;
```

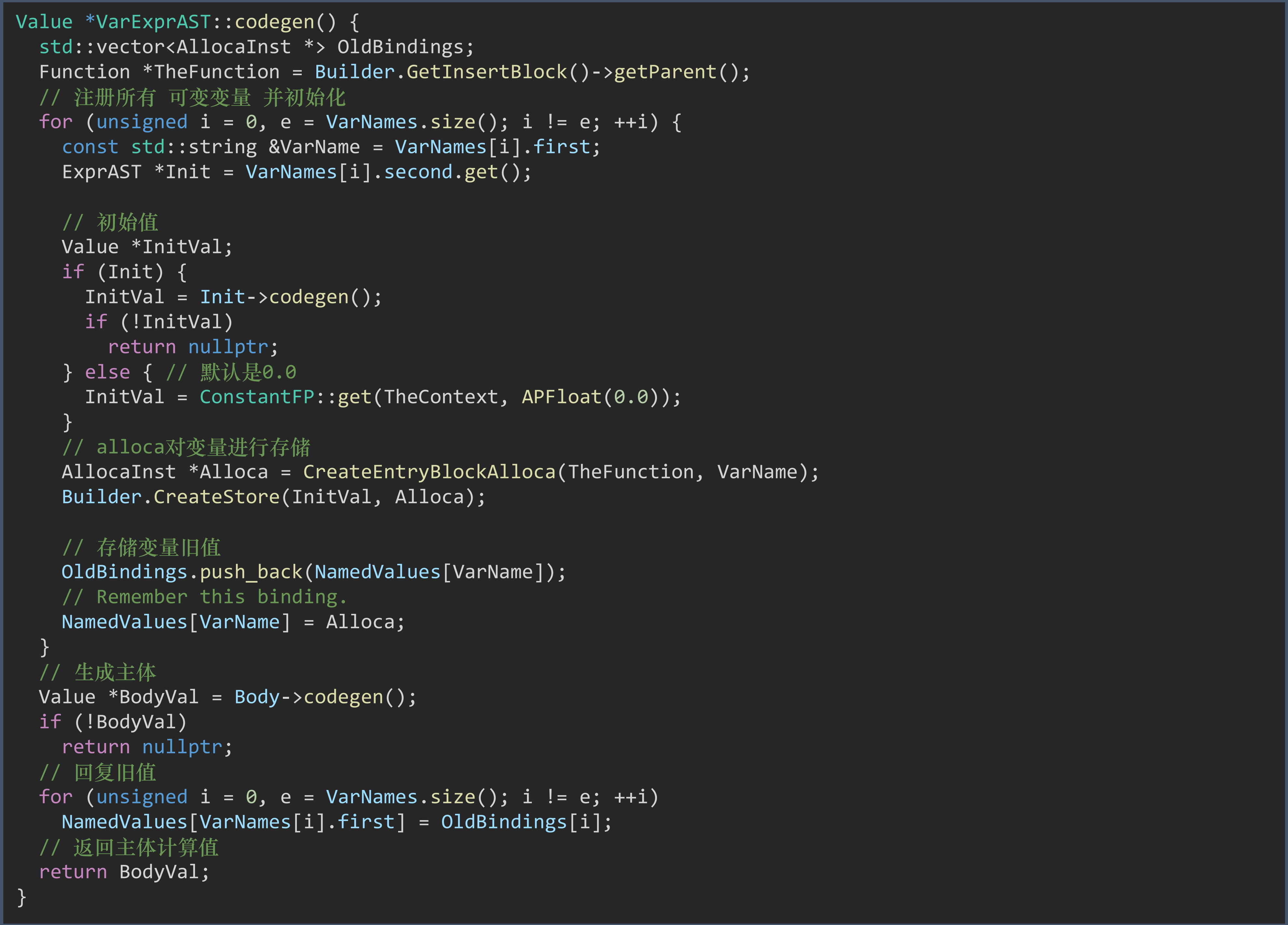


```
Value *ForExprAST::codegen() {
  // 获取函数
  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // 为变量创建一个Alloca
  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);

  // 为循环变量初始化
  Value *StartVal = Start->codegen();
  if (!StartVal)
    return nullptr;
  // 将循环变量的值存入Alloca
  Builder.CreateStore(StartVal, Alloca);


  // 为循环体创建BasicBlock
  BasicBlock *LoopBB = BasicBlock::Create(TheContext, "loop", TheFunct
ion);

  // 创建分支语句
  Builder.CreateBr(LoopBB);
  Builder.SetInsertPoint(LoopBB);





  // 从PHI中获取新的值，并保存旧值
  AllocaInst *OldVal = NamedValues[VarName];
  NamedValues[VarName] = Alloca;

  // 生成主体
  if (!Body->codegen())
    return nullptr;

  // 生成步长值
  Value *StepVal = nullptr;
Value *StepVal = nullptr;
if (Step) {
  StepVal = Step->codegen();
  if (!StepVal)
    return nullptr;
} else {

  StepVal = ConstantFP::get(TheContext, APFloat(1.0));
}

Value *NextVar = Builder.CreateFAdd(Variable, StepVal, "nextvar");


Value *EndCond = End->codegen();
if (!EndCond)
  return nullptr;







EndCond = Builder.CreateFCmpONE(
    EndCond, ConstantFP::get(TheContext, APFloat(0.0)), "loopcond");

  BasicBlock *LoopEndBB = Builder.GetInsertBlock();
BasicBlock *AfterBB =
    BasicBlock::Create(TheContext, "afterloop", TheFunction);

Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

Builder.SetInsertPoint(AfterBB);

Variable->addIncoming(NextVar, LoopEndBB);

if (OldVal)
```

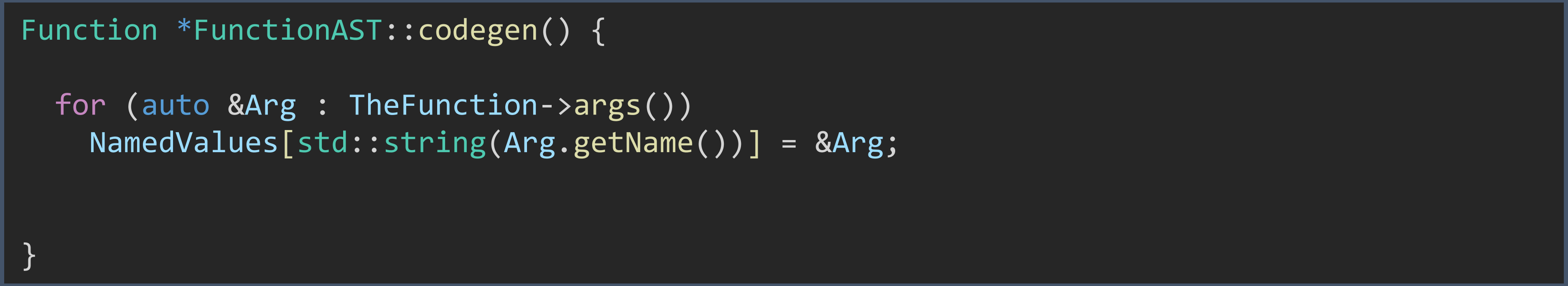
```
// 生成步长值
Value *StepVal = nullptr;
if (Step) {
  StepVal = Step->codegen();
  if (!StepVal)
    return nullptr;
} else {
  // 默认为1.0
  StepVal = ConstantFP::get(TheContext, APFloat(1.0));
}



// 计算终止条件
Value *EndCond = End->codegen();
if (!EndCond)
  return nullptr;

// 计算新的循环变量值
Value *CurVar = Builder.CreateLoad(Alloca, VarName.c_str());
Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
Builder.CreateStore(NextVar, Alloca);

// 条件判断
EndCond = Builder.CreateFCmpONE(
    EndCond, ConstantFP::get(TheContext, APFloat(0.0)), "loopcond");

// 创建after loop的BB
BasicBlock *AfterBB =
    BasicBlock::Create(TheContext, "afterloop", TheFunction);
// 插入br语句
Builder.CreateCondBr(EndCond, LoopBB, AfterBB);
// 循环后语句插入位置
Builder.SetInsertPoint(AfterBB);


// 将最后的循环值保存
if (OldVal)
```

```
    StepVal = ConstantFP::get(TheContext, APFloat(1.0));
  }

  Value *NextVar = Builder.CreateFAdd(Variable, StepVal, "nextvar");


  Value *EndCond = End->codegen();
  if (!EndCond)
    return nullptr;







  EndCond = Builder.CreateFCmpONE(
      EndCond, ConstantFP::get(TheContext, APFloat(0.0)), "loopcond");

    BasicBlock *LoopEndBB = Builder.GetInsertBlock();
  BasicBlock *AfterBB =
      BasicBlock::Create(TheContext, "afterloop", TheFunction);

  Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

  Builder.SetInsertPoint(AfterBB);

  Variable->addIncoming(NextVar, LoopEndBB);

  if (OldVal)
    NamedValues[VarName] = OldVal;
  else
    NamedValues.erase(VarName);

  // for expr always returns 0.0.
  return Constant::getNullValue(Type::getDoubleTy(TheContext));
}
```

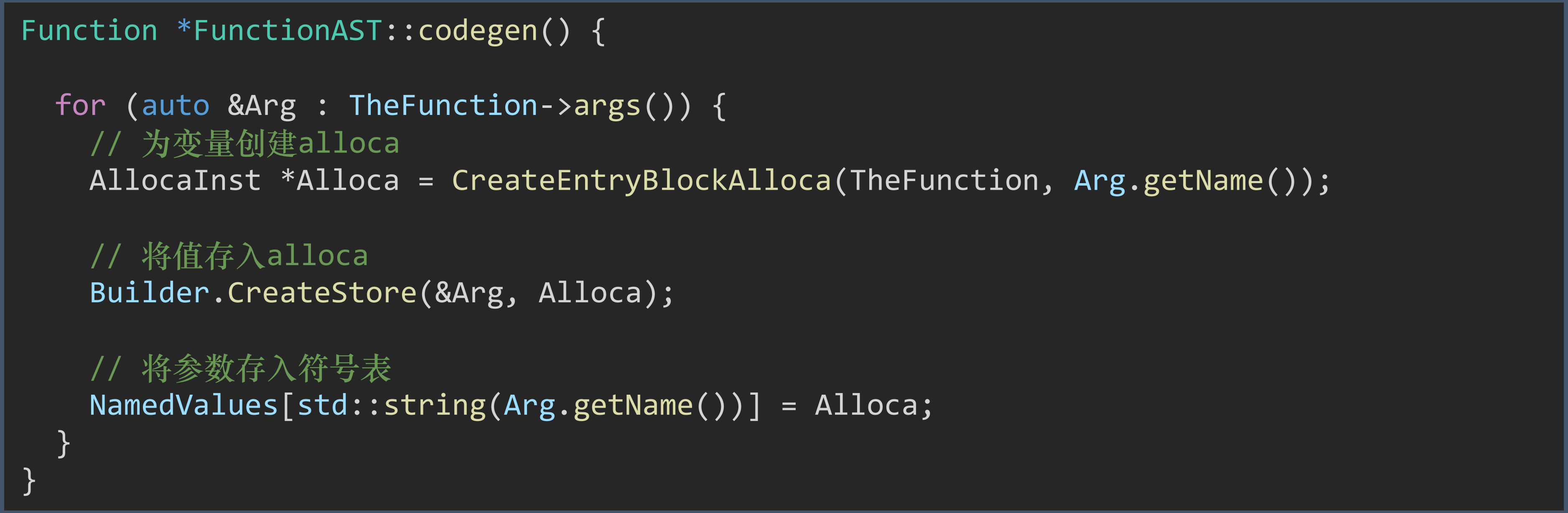
```
    StepVal = ConstantFP::get(TheContext, APFloat(1.0));
  }


  // 计算终止条件
  Value *EndCond = End->codegen();
  if (!EndCond)
    return nullptr;

  // 计算新的循环变量值
  Value *CurVar = Builder.CreateLoad(Alloca, VarName.c_str());
  Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
  Builder.CreateStore(NextVar, Alloca);

  // 条件判断
  EndCond = Builder.CreateFCmpONE(
      EndCond, ConstantFP::get(TheContext, APFloat(0.0)), "loopcond");

  // 创建after loop的BB
  BasicBlock *AfterBB =
      BasicBlock::Create(TheContext, "afterloop", TheFunction);
  // 插入br语句
  Builder.CreateCondBr(EndCond, LoopBB, AfterBB);
  // 循环后语句插入位置
  Builder.SetInsertPoint(AfterBB);


  // 将最后的循环值保存
  if (OldVal)
    NamedValues[VarName] = OldVal;
  else
    NamedValues.erase(VarName);

  // 返回0.0
  return Constant::getNullValue(Type::getDoubleTy(TheContext));
}
```



```cpp
Value *VarExprAST::codegen() {
  std::vector<AllocaInst *> OldBindings;
  Function *TheFunction = Builder.GetInsertBlock()->getParent();
  // 注册所有 可变变量 并初始化
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second.get();

    // 初始值
    Value *InitVal;
    if (Init) {
      InitVal = Init->codegen();
      if (!InitVal)
        return nullptr;
    } else { // 默认是0.0
      InitVal = ConstantFP::get(TheContext, APFloat(0.0));
    }
    // alloca对变量进行存储
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
    Builder.CreateStore(InitVal, Alloca);

    // 存储变量旧值
    OldBindings.push_back(NamedValues[VarName]);
    // Remember this binding.
    NamedValues[VarName] = Alloca;
  }
  // 生成主体
  Value *BodyVal = Body->codegen();
  if (!BodyVal)
    return nullptr;
  // 回复旧值
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
    NamedValues[VarNames[i].first] = OldBindings[i];
  // 返回主体计算值
  return BodyVal;
}
```



```
Function *FunctionAST::codegen() {

  for (auto &Arg : TheFunction->args())
    NamedValues[std::string(Arg.getName())] = &Arg;


}
```



```
Function *FunctionAST::codegen() {

  for (auto &Arg : TheFunction->args()) {
    // 为变量创建alloca
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName());

    // 将值存入alloca
    Builder.CreateStore(&Arg, Alloca);

    // 将参数存入符号表
    NamedValues[std::string(Arg.getName())] = Alloca;
  }
}
```

pass、main函数

```
static void InitializeModuleAndPassManager() {

  // 将alloca转换为寄存器
  TheFPM->add(createPromoteMemoryToRegisterPass());

  TheFPM->doInitialization();
}
```

```
int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['='] = 2;

  return 0;
}
```

def acc(n) var sum=0 in (for i=1,i<n in sum=sum+i) : sum;

```
•  从执行了def binary: 1 (x y) y;之后开始
•  HandleDefinition();                                                                   // for
     •  FnAST = ParseDefinition()                                                         // acc
          •  Proto = ParsePrototype()
               •  FnName = IdentifierStr                                                  // (
               •  vector<string> ArgNames;
               •  while (getNextToken() == tok_identifier)                                // n )
          •  E = ParseExpression()                                                        // var
               •  LHS = ParseUnary()
                    •  ParsePrimary()
                         •  ParseVarExpr();
                              •  vector<pair<string, unique_ptr<ExprAST>>> VarNames;      // sum
                              •  unique_ptr<ExprAST>                                      Init // =
                              •  Init = ParseExpression();                                // 0 in
                              •  VarNames.push_back(make_pair(Name, move(Init)))
                              •  Body = ParseExpression()            // ( for i = 1 , i < n in sum = sum + i ) : sum ;
     •  FnIR = FnAST->codegen()
     •  Function *TheFunction = getFunction(P.getName());                                 生成函数
          •  if (P.isBinaryOp())                                                          判断是否为二元运算符
          •  BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
          •  Builder.SetInsertPoint(BB);
          •  for (auto &Arg : TheFunction->args())
               •  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName());   为变量在stack的内存上分配
               •  Builder.CreateStore(&Arg, Alloca);                                      将参数保存在Alloca
               •  NamedValues[std::string(Arg.getName())] = Alloca;                       将变量添加到符号表
          •  RetVal = Body->codegen()
               •  const std::string &VarName = VarNames[i].first;                         处理 可变变量
               •  ExprAST *Init = VarNames[i].second.get();
               •  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
               •  Builder.CreateStore(InitVal, Alloca);
               •  Value *BodyVal = Body->codegen();                                       生成var主体
                    •  BinaryExprAST::codegen()                                           生成:
                         •  ForExprAST::codegen()                                         生成for
     •  forFnIR->print(errs());
```