Kaleidoscope

代码解释(5)

万花筒语言 - LLVM 新手入门教程

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl06.html

PLCT - SSC

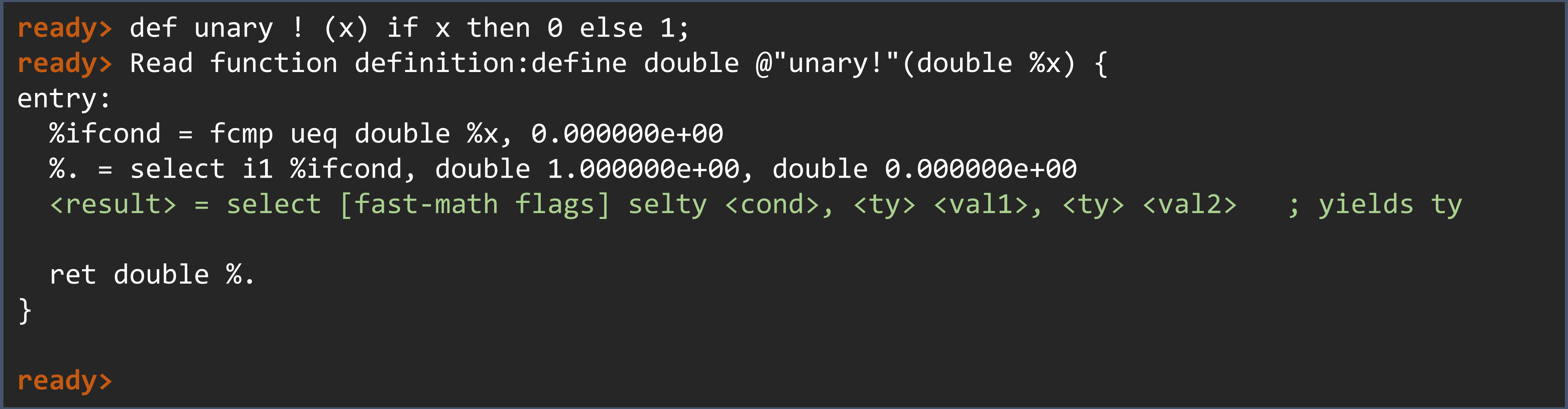
语言拓展：自定义运算符

- 本次教程实现了可以利用 已定义的运算符 来构造新的运算符
- 我们本次自行拓展了 除法‘/’运算符

```
def unary! (x)
    if x
    then 0
    else 1;
```

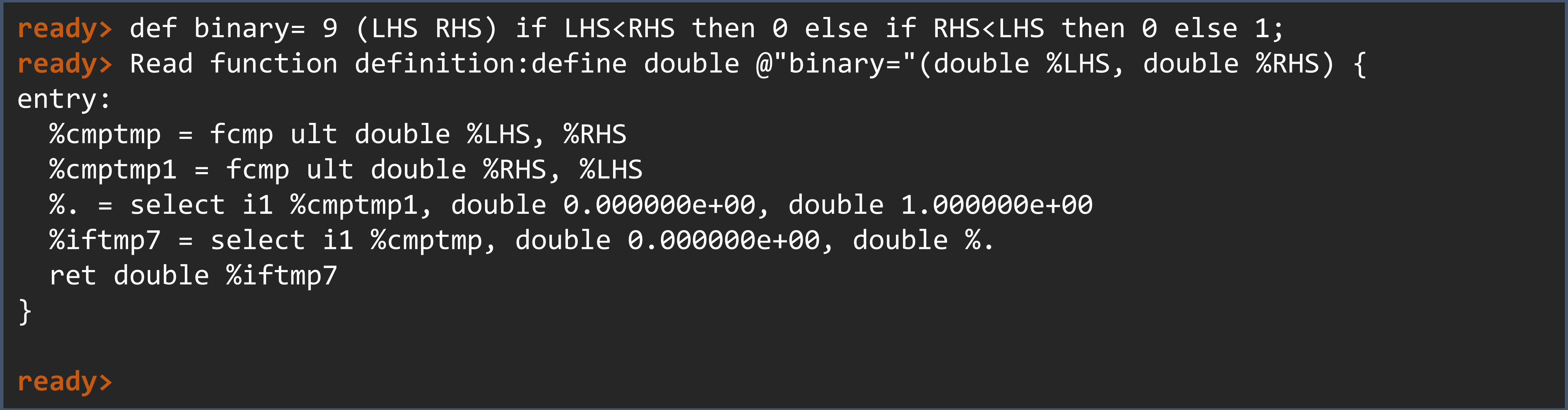
```
def binary= 9 (LHS RHS)
    if LHS<RHS
    then 0
    else
        if RHS<LHS
        then 0
        else 1;
```

def unary ! (x) if x then 0 else 1;



```
ready> def unary ! (x) if x then 0 else 1;
ready> Read function definition:define double @"unary!"(double %x) {
entry:
  %ifcond = fcmp ueq double %x, 0.000000e+00
  %. = select i1 %ifcond, double 1.000000e+00, double 0.000000e+00
  <result> = select [fast-math flags] selty <cond>, <ty> <val1>, <ty> <val2>   ; yields ty

  ret double %.
}

ready>
```

```
def binary= 9 (LHS RHS)
    if LHS<RHS then 0 else
        if RHS<LHS then 0 else 1;
```



```
ready> def binary= 9 (LHS RHS) if LHS<RHS then 0 else if RHS<LHS then 0 else 1;
ready> Read function definition:define double @"binary="(double %LHS, double %RHS) {
entry:
  %cmptmp = fcmp ult double %LHS, %RHS
  %cmptmp1 = fcmp ult double %RHS, %LHS
  %. = select i1 %cmptmp1, double 0.000000e+00, double 1.000000e+00
  %iftmp7 = select i1 %cmptmp, double 0.000000e+00, double %.
  ret double %iftmp7
}

ready>
```

词法分析

```
enum Token {

  tok_if = -6,
  tok_then = -7,
  tok_else = -8,
  tok_for = -9,
  tok_in = -10,

  tok_binary = -11,
  tok_unary = -12
};
```

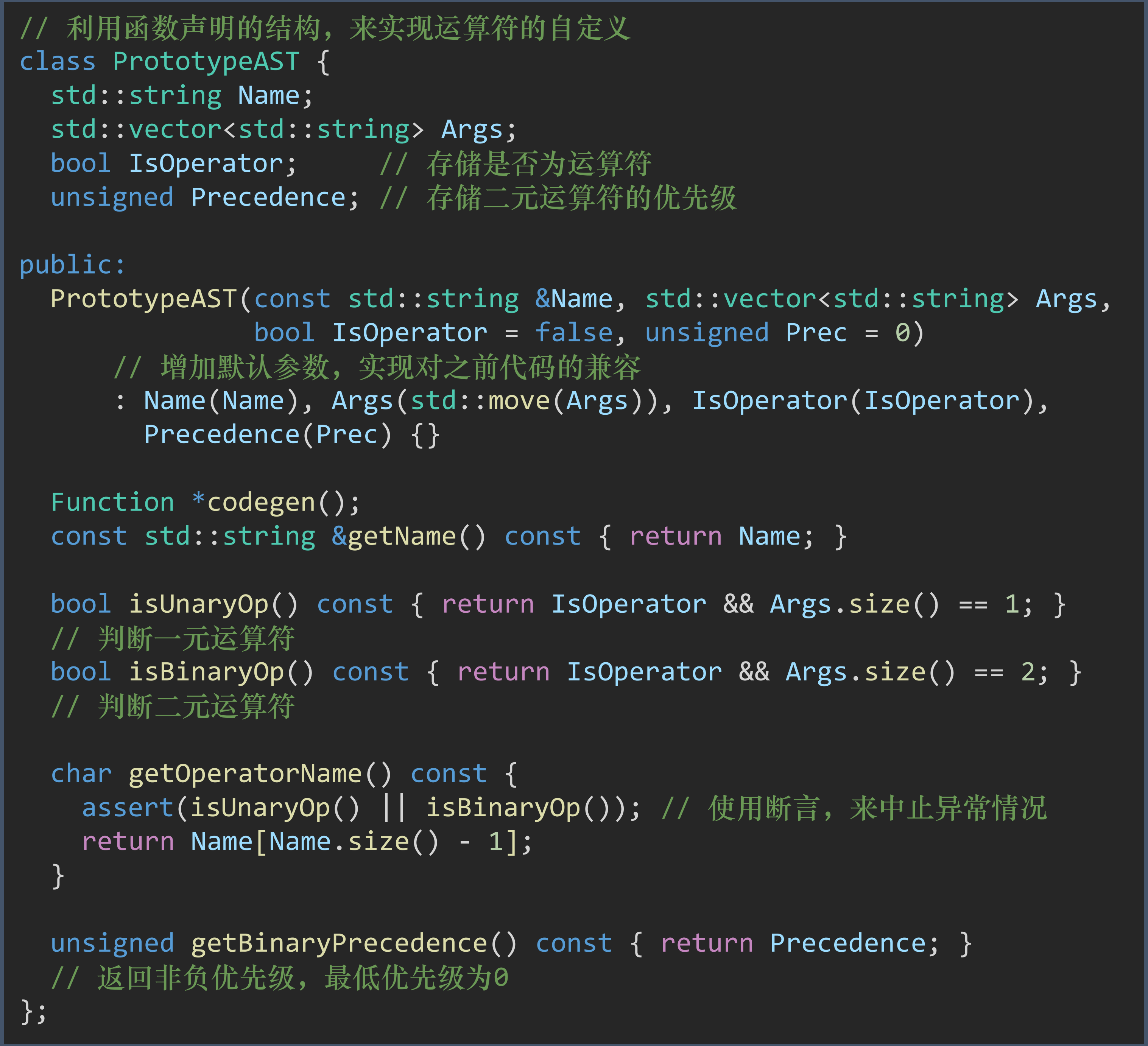
```
static int gettok() {
  static int LastChar = ' ';

  while (isspace(LastChar))
    LastChar = getchar();

  if (isalpha(LastChar)) {
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar())))
      IdentifierStr += LastChar;

    if (IdentifierStr == "in")
      return tok_in;
    if (IdentifierStr == "binary") // 二元操作符
      return tok_binary;
    if (IdentifierStr == "unary") // 一元操作符
      return tok_unary;
    return tok_identifier;
  }
```

抽象语法树

```
class UnaryExprAST : public ExprAST {
  char Opcode;                           // 一元操作符
  std::unique_ptr<ExprAST> Operand; // 一元操作数

public:
  UnaryExprAST(char Opcode, std::unique_ptr<ExprAST> Operand)
      : Opcode(Opcode), Operand(std::move(Operand)) {}

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

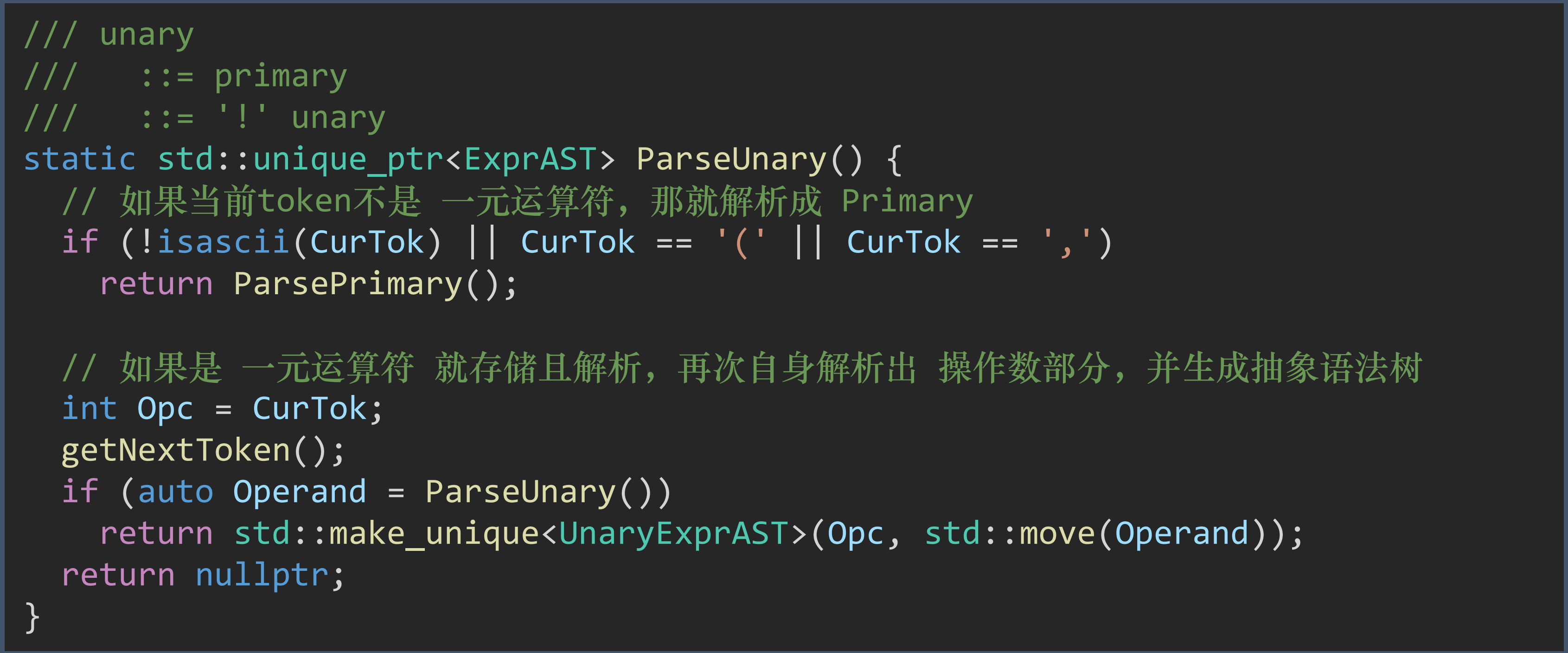
```
class BinaryExprAST : public ExprAST {
  char Op;                               // 二元操作符
  std::unique_ptr<ExprAST> LHS, RHS; // 左部右部

public:
  BinaryExprAST(char Op, std::unique_ptr<ExprAST> LHS,
                std::unique_ptr<ExprAST> RHS)
      : Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}

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

函数声明语法树



```
// 利用函数声明的结构，来实现运算符的自定义
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
  bool IsOperator;     // 存储是否为运算符
  unsigned Precedence; // 存储二元运算符的优先级

public:
  PrototypeAST(const std::string &Name, std::vector<std::string> Args,
               bool IsOperator = false, unsigned Prec = 0)
      // 增加默认参数，实现对之前代码的兼容
      : Name(Name), Args(std::move(Args)), IsOperator(IsOperator),
        Precedence(Prec) {}

  Function *codegen();
  const std::string &getName() const { return Name; }

  bool isUnaryOp() const { return IsOperator && Args.size() == 1; }
  // 判断一元运算符
  bool isBinaryOp() const { return IsOperator && Args.size() == 2; }
  // 判断二元运算符

  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp()); // 使用断言，来中止异常情况
    return Name[Name.size() - 1];
  }

  unsigned getBinaryPrecedence() const { return Precedence; }
  // 返回非负优先级，最低优先级为0
};
```

解析 一元运算符



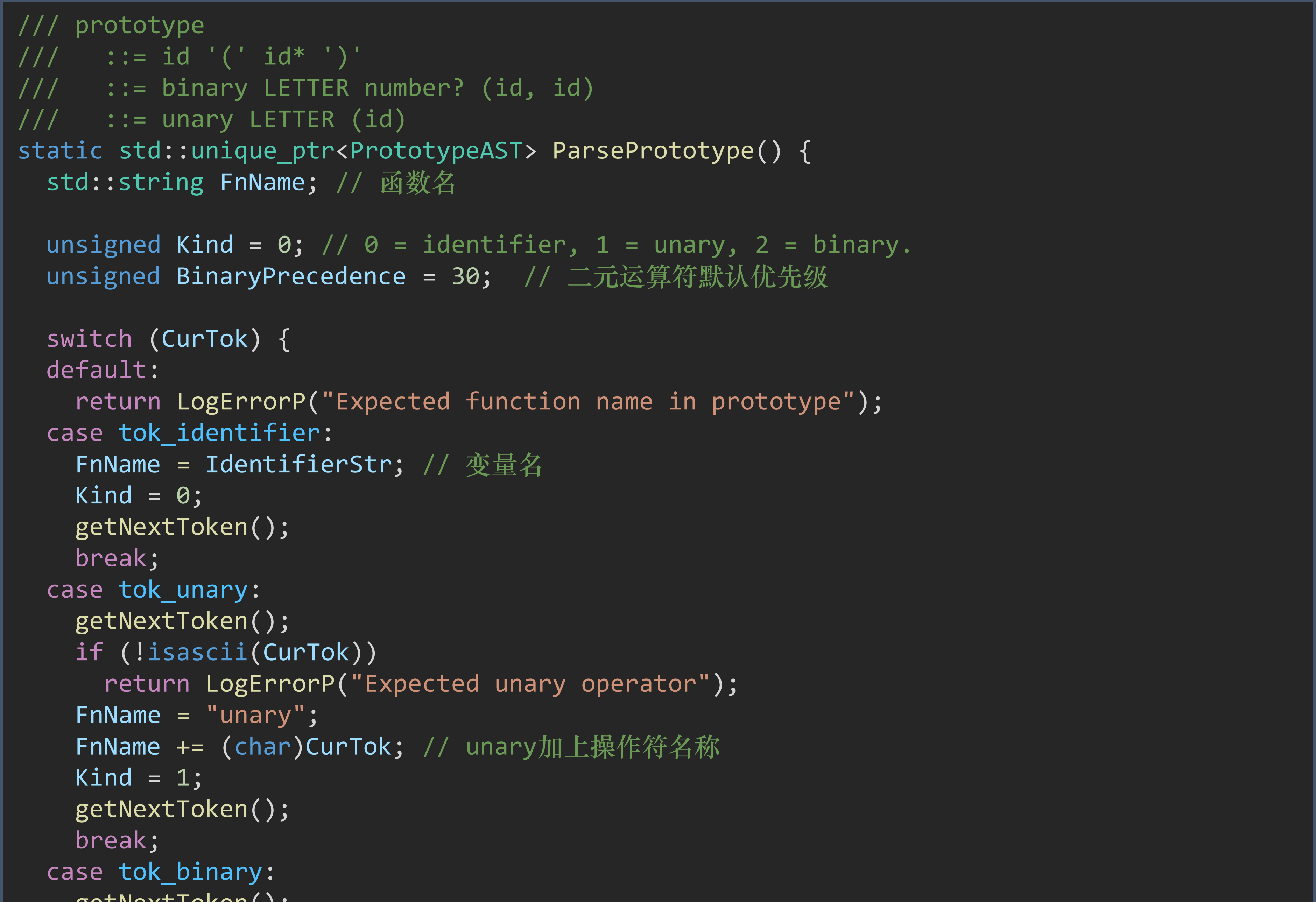
```cpp
/// unary
///   ::= primary
///   ::= '!' unary
static std::unique_ptr<ExprAST> ParseUnary() {
  // 如果当前token不是 一元运算符，那就解析成 Primary
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // 如果是 一元运算符 就存储且解析，再次自身解析出 操作数部分，并生成抽象语法树
  int Opc = CurTok;
  getNextToken();
  if (auto Operand = ParseUnary())
    return std::make_unique<UnaryExprAST>(Opc, std::move(Operand));
  return nullptr;
}
```

解析 运算符与右部

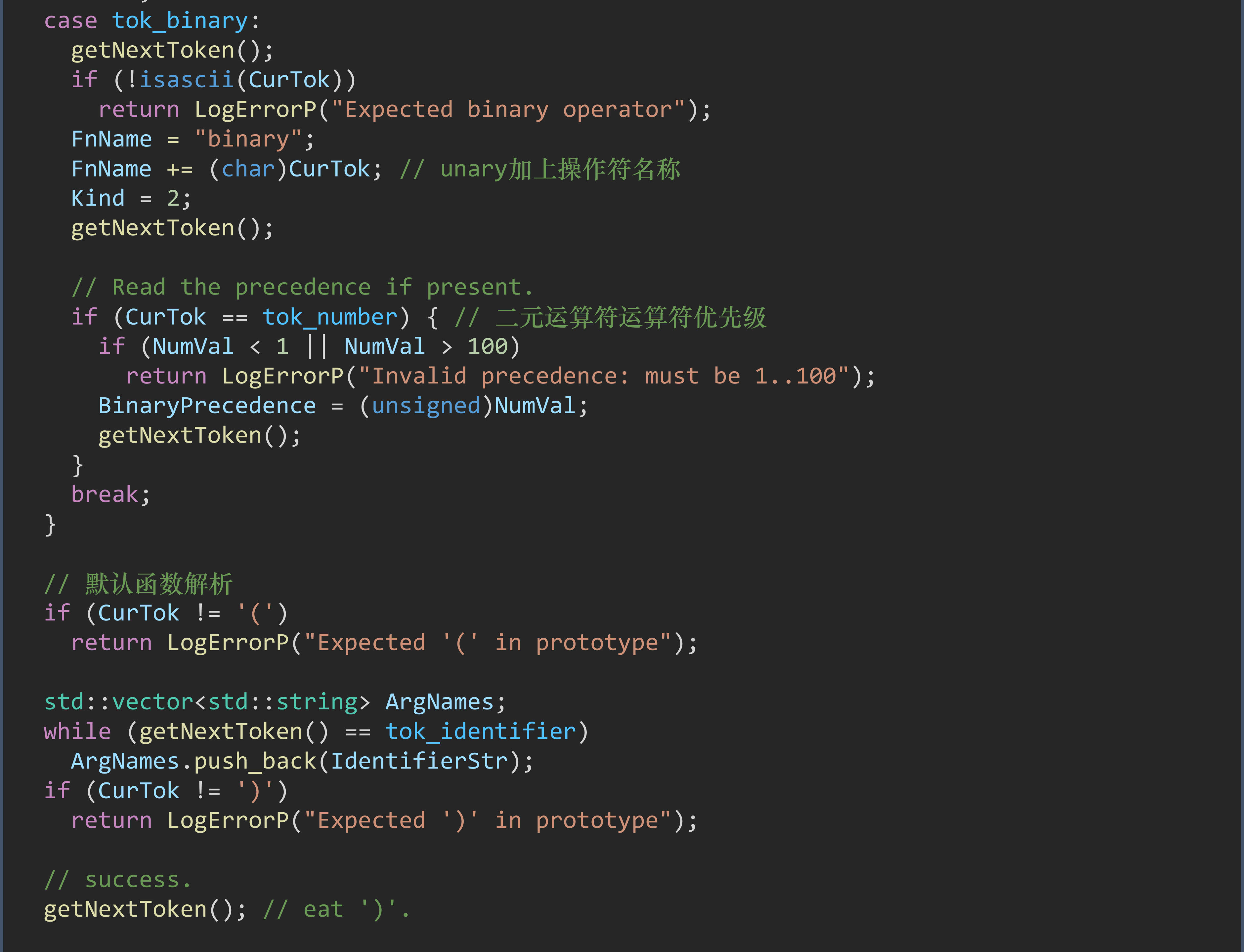
```
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<ExprAST> LHS) {
  while (true) {
    // 在解析为Primary之前，先进行判断是否为 一元运算符
    auto RHS = ParsePrimary();
    auto RHS = ParseUnary();
  }
}
```

```
static std::unique_ptr<ExprAST> ParseExpression() {
  auto LHS = ParsePrimary();
  auto LHS = ParseUnary();

  return ParseBinOpRHS(0, std::move(LHS));
}
```

解析 函数声明



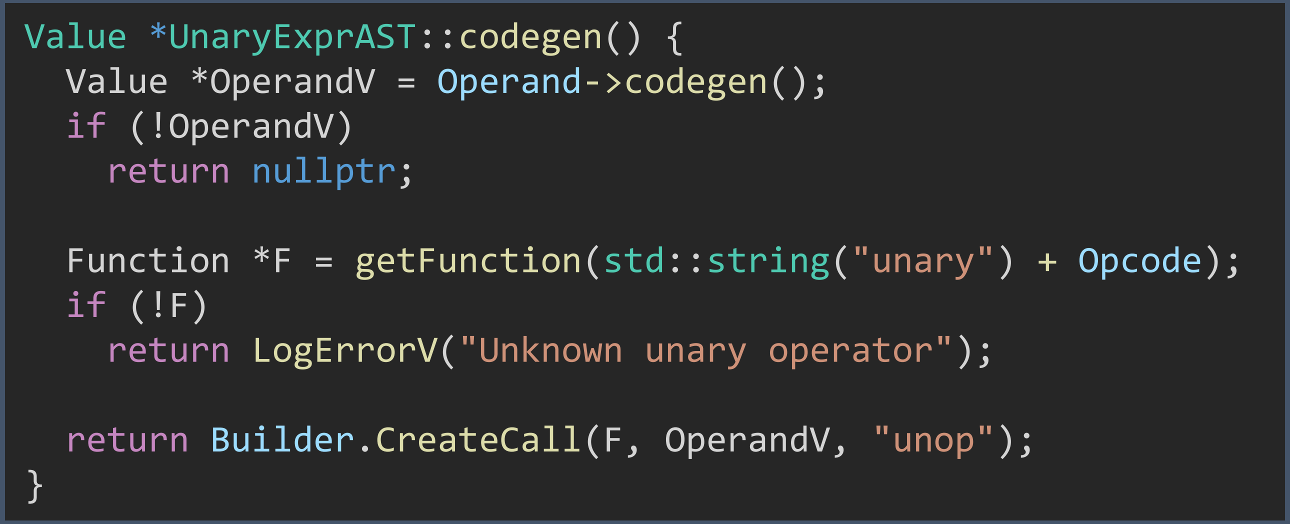
```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
///   ::= unary LETTER (id)
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  std::string FnName; // 函数名

  unsigned Kind = 0; // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;  // 二元运算符默认优先级

  switch (CurTok) {
  default:
    return LogErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr; // 变量名
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
    getNextToken();
    if (!isascii(CurTok))
      return LogErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok; // unary加上操作符名称
    Kind = 1;
    getNextToken();
    break;
  case tok_binary:
    getNextToken();
```



```
case tok_binary:
  getNextToken();
  if (!isascii(CurTok))
    return LogErrorP("Expected binary operator");
  FnName = "binary";
  FnName += (char)CurTok; // unary加上操作符名称
  Kind = 2;
  getNextToken();

  // Read the precedence if present.
  if (CurTok == tok_number) { // 二元运算符运算符优先级
    if (NumVal < 1 || NumVal > 100)
      return LogErrorP("Invalid precedence: must be 1..100");
    BinaryPrecedence = (unsigned)NumVal;
    getNextToken();
  }
  break;
}

// 默认函数解析
if (CurTok != '(')
  return LogErrorP("Expected '(' in prototype");

std::vector<std::string> ArgNames;
while (getNextToken() == tok_identifier)
  ArgNames.push_back(IdentifierStr);
if (CurTok != ')')
  return LogErrorP("Expected ')' in prototype");

// success.
getNextToken(); // eat ')'.
```

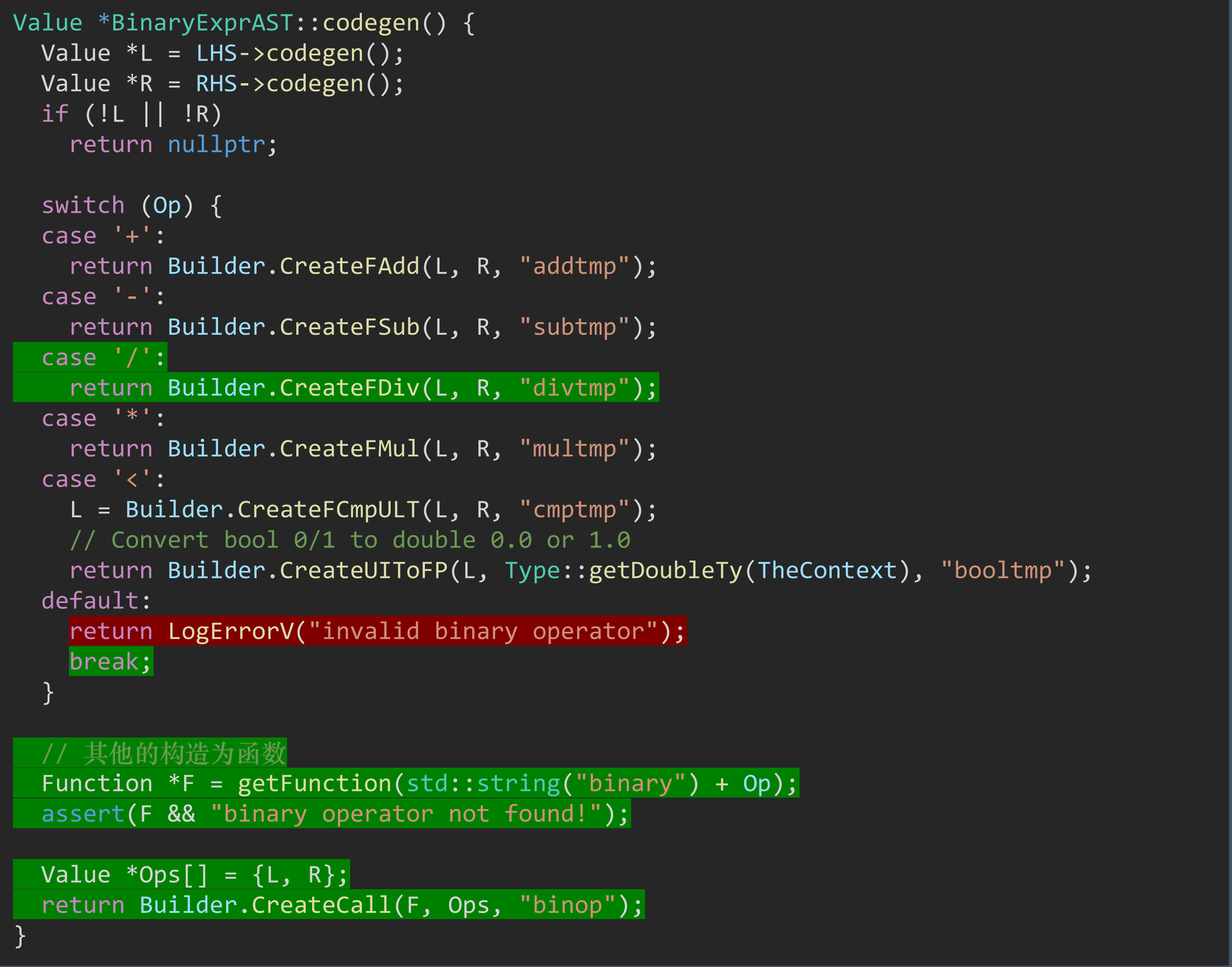
```
  // 默认函数解析
  if (CurTok != '(')
    return LogErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return LogErrorP("Expected ')' in prototype");

  // success.
  getNextToken(); // eat ')'.

  // 验证
  if (Kind && ArgNames.size() != Kind)
    return LogErrorP("Invalid number of operands for operator");

  // Kind用来控制是否是操作符
  return std::make_unique<PrototypeAST>(FnName, ArgNames, Kind != 0,
                                        BinaryPrecedence);
}
```

一元运算符 生成



```
Value *UnaryExprAST::codegen() {
  Value *OperandV = Operand->codegen();
  if (!OperandV)
    return nullptr;

  Function *F = getFunction(std::string("unary") + Opcode);
  if (!F)
    return LogErrorV("Unknown unary operator");

  return Builder.CreateCall(F, OperandV, "unop");
}
```

二元运算符生成



```cpp
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  switch (Op) {
  case '+':
    return Builder.CreateFAdd(L, R, "addtmp");
  case '-':
    return Builder.CreateFSub(L, R, "subtmp");
  case '/':
    return Builder.CreateFDiv(L, R, "divtmp");
  case '*':
    return Builder.CreateFMul(L, R, "multmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L, Type::getDoubleTy(TheContext), "booltmp");
  default:
    return LogErrorV("invalid binary operator");
    break;
  }

  // 其他的构造为函数
  Function *F = getFunction(std::string("binary") + Op);
  assert(F && "binary operator not found!");

  Value *Ops[] = {L, R};
  return Builder.CreateCall(F, Ops, "binop");
}
```

函数 生成

```
Function *FunctionAST::codegen() {
  // Transfer ownership of the prototype to the FunctionProtos map, but keep a
  // reference to it for use below.
  auto &P = *Proto;
  FunctionProtos[Proto->getName()] = std::move(Proto);
  Function *TheFunction = getFunction(P.getName());
  if (!TheFunction)
    return nullptr;

  // If this is an operator, install it.
  if (P.isBinaryOp())
    BinopPrecedence[P.getOperatorName()] = P.getBinaryPrecedence();

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
  Builder.SetInsertPoint(BB);


  // Error reading body, remove function.
  TheFunction->eraseFromParent();

  if (P.isBinaryOp())
    BinopPrecedence.erase(P.getOperatorName());
  return nullptr;
}
```

def unary! (x) if x then 0 else 1;

- main() 获取了def的token
    - MainLoop()
        - HandleDefinition()
            - FnAST = ParseDefinition() 获取了unary的token
                - ParsePrototype() 获取了!的token
                    - FnName = "unary";
                    - FnName += (char)CurTok;
                    - getNextToken(); 获取了(的token
                    - std::vector<std::string> ArgNames;
                    - while (getNextToken() == tok_identifier) ArgNames.push_back(IdentifierStr)
                    - 获取了x的token, 获取了)的token
                    - getNextToken(); 获取了if的token
                - ParseExpression() 获取了x then 0 else 1 ; 的token
            - FnIR = FnAST->codegen()
                - auto &P = *Proto;
                - FunctionProtos[Proto->getName()] = std::move(Proto);
                - Function *TheFunction = getFunction(P.getName());
                - BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
                - Builder.SetInsertPoint(BB);
                - RetVal = Body->codegen()
                - Builder.CreateRet(RetVal);
                - TheFPM->run(*TheFunction);