Kaleidoscope

代码解释(8/8)

万花筒语言 - LLVM 新手入门教程

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl09.html

https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl10.html

PLCT - SSC

增加调试信息

- 作为总结，我们选择对代码重新进行注释
- 删了JIT和优化，增加debug模式

- `git clone https://github.com/llvm/llvm-project.git`
- `cd llvm-project`
- `mkdir build`
- `cd build`
- `cmake -DLLVM_ENABLE_PROJECTS="clang;llvm;lldb" -G "Unix Makefiles" ../llvm`
- `make -j $(nproc)`
- `make -j $(nproc) Kaleidoscope`

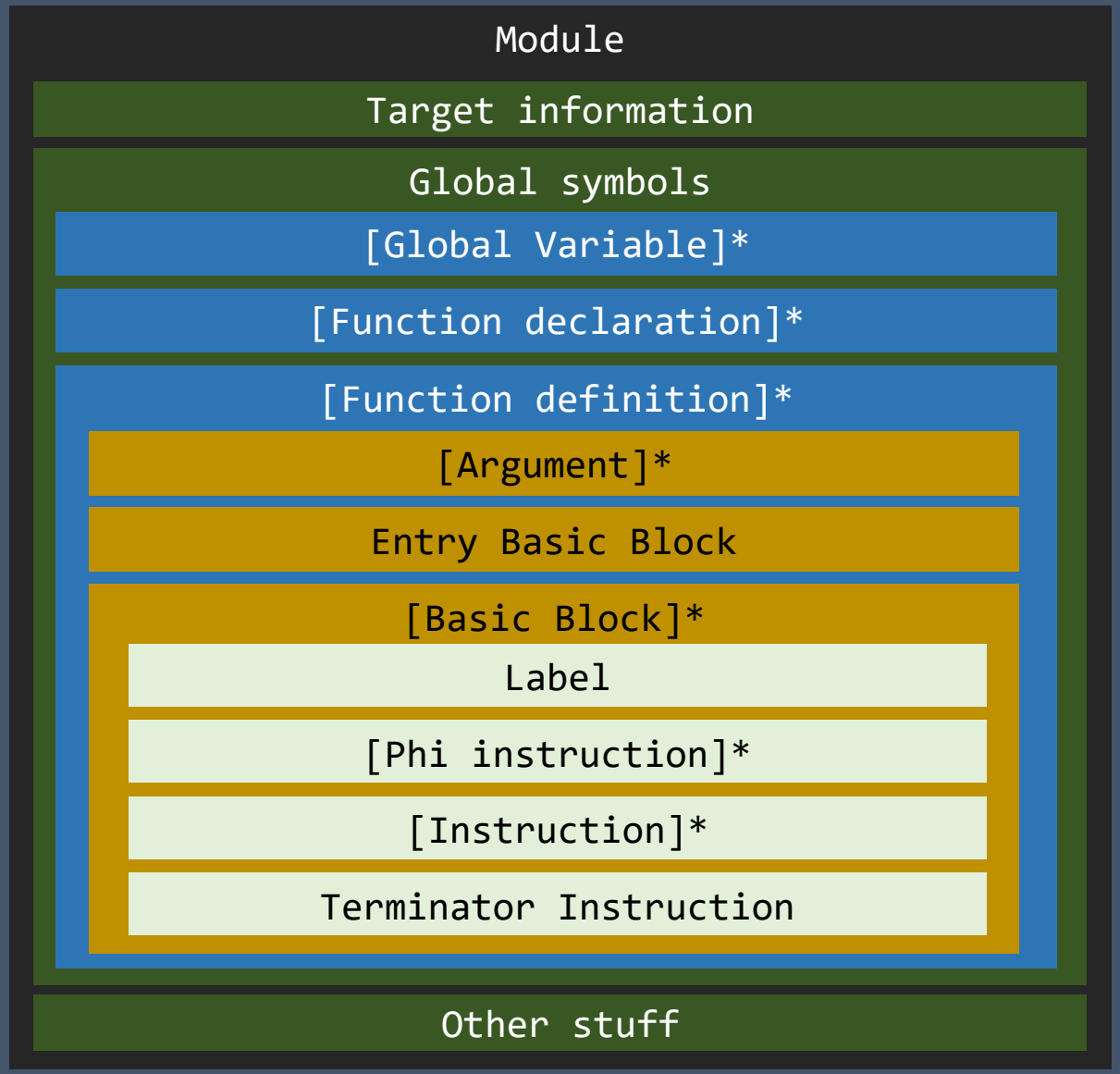
编译、调试

fib.ks

```
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2);

def main() # 为程序提供执行入口
  fib(10);
```

```
Kaleidoscope-Ch9 < fib.ks &> fib.ll
# 读取fib.ks源文件，并输出到fib.ksdb

clang -x ir fib.ll -o fib-debug
# clang编译并输出到fib-debug

lldb fib-debug
# 使用lldb进行debug
```

LLVM IR 代码布局


Module
Target information
Global symbols
[Global Variable]*
[Function declaration]*
[Function definition]*
[Argument]*
Entry Basic Block
[Basic Block]*
Label
[Phi instruction]*
[Instruction]*
Terminator Instruction
Other stuff

汇编

```
; ModuleID = 'my cool jit'
指明源文件，此处被设置为了字符串
source_filename = "my cool jit"
指明编译自什么文件，此处被设置为了字符串
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
数据布局，存储了ELF 数据类型长度等信息

define double @fib(double %x) !dbg !4 {
!4 在后面为fib定义了作用域等信息
entry:
  %x1 = alloca double, align 8
  call void @llvm.dbg.declare(metadata double* %x1, metadata !9, metadata !DIExpression()), !dbg !10
  为%x1定义了调试信息

  store double %x, double* %x1, align 8
  %x2 = load double, double* %x1, align 8, !dbg !11
  %cmptmp = fcmp ult double %x2, 3.000000e+00, !dbg !12
  %booltmp = uitofp i1 %cmptmp to double, !dbg !12
  %ifcond = fcmp one double %booltmp, 0.000000e+00, !dbg !12
  br i1 %ifcond, label %then, label %else, !dbg !12
then:                                             ; preds = %entry
  br label %ifcont, !dbg !13
else:                                             ; preds = %entry
  %x3 = load double, double* %x1, align 8, !dbg !14
  %subtmp = fsub double %x3, 1.000000e+00, !dbg !15
  %calltmp = call double @fib(double %subtmp), !dbg !15
  %x4 = load double, double* %x1, align 8, !dbg !16
  %subtmp5 = fsub double %x4, 2.000000e+00, !dbg !17
  %calltmp6 = call double @fib(double %subtmp5), !dbg !17
  %addtmp = fadd double %calltmp, %calltmp6, !dbg !17
  br label %ifcont, !dbg !17
ifcont:                                           ; preds = %else, %then
  %iftmp = phi double [ 1.000000e+00, %then ], [ %addtmp, %else ], !dbg !17
  ret double %iftmp, !dbg !17
}
; Function Attrs: nounwind readnone speculatable willreturn
declare void @llvm.dbg.declare(metadata, metadata, metadata) #0
```

汇编

```
}
; Function Attrs: nounwind readnone speculatable willreturn
declare void @llvm.dbg.declare(metadata, metadata, metadata) #0
define double @main() !dbg !18 {
entry:
  %calltmp = call double @fib(double 1.000000e+01), !dbg !21
  ret double %calltmp, !dbg !21
}
attributes #0 = { nounwind readnone speculatable willreturn }
!llvm.module.flags = !{!0}
!llvm.dbg.cu = !{!1}
!0 = !{i32 2, !"Debug Info Version", i32 3}
!1 = distinct !DICompileUnit(language: DW_LANG_C, file: !2, producer: "Kaleidoscope Compiler", isOptimized: false, runtimeVersion: 0, emiss
ionKind: FullDebug, enums: !3)
!2 = !DIFile(filename: "fib.ks", directory: ".")
!3 = !{}
!4 = distinct !DISubprogram(name: "fib", scope: !2, file: !2, line: 3, type: !5, scopeLine: 3, flags: DIFlagPrototyped, spFlags: DISPFlagDe
finition, unit: !1, retainedNodes: !8)
!5 = !DISubroutineType(types: !6)
!6 = !{!7, !7}
!7 = !DIBasicType(name: "double", size: 64, encoding: DW_ATE_float)
!8 = !{!9}
!9 = !DILocalVariable(name: "x", arg: 1, scope: !4, file: !2, line: 3, type: !7)
!10 = !DILocation(line: 3, scope: !4)
!11 = !DILocation(line: 4, column: 6, scope: !4)
!12 = !DILocation(line: 4, column: 10, scope: !4)
!13 = !DILocation(line: 5, column: 5, scope: !4)
!14 = !DILocation(line: 7, column: 9, scope: !4)
!15 = !DILocation(line: 7, column: 11, scope: !4)
!16 = !DILocation(line: 7, column: 18, scope: !4)
!17 = !DILocation(line: 7, column: 20, scope: !4)
!18 = distinct !DISubprogram(name: "main", scope: !2, file: !2, line: 9, type: !19, scopeLine: 9, flags: DIFlagPrototyped, spFlags: DISPFla
gDefinition, unit: !1, retainedNodes: !3)
!19 = !DISubroutineType(types: !20)
!20 = !{!7}
!21 = !DILocation(line: 10, column: 7, scope: !18)
```

头文件、命名空间

```
#include "../include/KaleidoscopeJIT.h" // Kaleidoscope的JIT
#include "llvm/ADT/STLExtras.h"         // llvm的STL拓展
#include "llvm/Analysis/BasicAliasAnalysis.h" // LLVM 基础的 无状态和本地别名分析
#include "llvm/Analysis/Passes.h" // 访问器函数的声明，其将pass用于分析库
#include "llvm/IR/DIBuilder.h"    // 益于创建LLVM IR形式的debug信息
#include "llvm/IR/IRBuilder.h"    // 统一且简单的LLVM指令创建接口
#include "llvm/IR/LLVMContext.h" // global类型的容器
#include "llvm/IR/LegacyPassManager.h" // 保存 维护 优化 pass的执行
#include "llvm/IR/Module.h"            // LLVM IR对象的顶层容器
#include "llvm/IR/Verifier.h"          // 函数校验
#include "llvm/Support/Host.h"         // 宿主设备的基础信息
#include "llvm/Support/TargetSelect.h" // 确保 目标的特定类被连接到主可执行文件，并合理初始化
#include "llvm/Transforms/Scalar.h" // Scalar Pass
#include <cctype>                   // 用于查看字符类型
#include <cstdio>                   // 处理IO
#include <map>                      // 用于名称与值的映射
#include <string>                   // 字符串
#include <vector>                   // 变长数组

using namespace llvm;       // 基础类
using namespace llvm::orc; // ORC提供了模块API用于构建JIT编译器
```

词法分析

```
// Token的枚举数组，负数用于区分不同token，正数用于存储 运算符 char类型的ASCII值
enum Token {
  tok_eof = -1,         // ^D 或 ^Z
  tok_def = -2,         // 函数声明
  tok_extern = -3,      // 拓展库函数
  tok_identifier = -4, // 变量名
  tok_number = -5,      // 数字
  tok_if = -6,          // 控制流 if
  tok_then = -7,        // 控制流 then
  tok_else = -8,        // 控制流 else
  tok_for = -9,         // 控制流 for
  tok_in = -10,         // 控制流 in
  tok_binary = -11,     // 二元运算符
  tok_unary = -12,      // 一元运算符
  tok_var = -13         // 可变变量
};

// 返回token的名称
std::string getTokName(int Tok) {
  switch (Tok) {
  case tok_eof:
    return "eof";
  case tok_def:
    return "def";
  case tok_extern:
    return "extern";
  case tok_identifier:
    return "identifier";
  case tok_number:
    return "number";
  case tok_if:
```

词法分析

```
  case tok_if:
    return "if";
  case tok_then:
    return "then";
  case tok_else:
    return "else";
  case tok_for:
    return "for";
  case tok_in:
    return "in";
  case tok_binary:
    return "binary";
  case tok_unary:
    return "unary";
  case tok_var:
    return "var";
  }
  return std::string(1, (char)Tok); // 其他情况返回 Tok值
}

namespace {
class PrototypeAST; // 函数声明AST
class ExprAST;      // 表达式AST
} // namespace
static LLVMContext TheContext;           // 管理和global数据
static IRBuilder<> Builder(TheContext); // IR生成器
struct DebugInfo {
  // 遵循DWARF
  DICompileUnit *TheCU; // Debug Information 调试信息编译单元
  DIType *DblTy;        // 调试信息类型
  std::vector<DIScope *> LexicalBlocks; // 调试词法作用域
```

词法分析

```
  void emitLocation(ExprAST *AST); // 为AST生成调试的代码位置信息
  DIType *getDoubleTy();           // 返回调试信息类型
} KSDbgInfo; // 生成一个调试信息对象, KaleidoscopeDebugInfo

// 存储源代码行列位置
struct SourceLocation {
  int Line; // 行
  int Col;  // 列
};
static SourceLocation CurLoc;           // 当前位置
static SourceLocation LexLoc = {1, 0}; // 词法解析位置

// 对每个读入的字符,先记录其位置,然后返回该字符
static int advance() {
  int LastChar = getchar();

  // linux:\n  windows: \r\n  macos: \r
  if (LastChar == '\n' || LastChar == '\r') {
    LexLoc.Line++;  // 换行时,行数+1
    LexLoc.Col = 0; // 列数清零
  } else
    LexLoc.Col++;  // 未换行时,列数+1
  return LastChar; // 返回当前字符
}

static std::string IdentifierStr; // 如果是 tok_identifier 会保存 变量名
static double NumVal;             // 如果是 tok_number 会保存数值

// 从输入中读取并返回token
static int gettok() {
  static int LastChar = ' ';
```

词法分析

```
// 跳过所有空格
while (isspace(LastChar))
  LastChar = advance();
// 词法解析的位置 传给 当前位置
CurLoc = LexLoc;
// 处理变量名和关键字: [a-zA-Z][a-zA-Z0-9]*
if (isalpha(LastChar)) { // [a-zA-Z]
  IdentifierStr = LastChar;
  while (isalnum((LastChar = advance()))) // [a-zA-Z0-9]*
    IdentifierStr += LastChar;

  if (IdentifierStr == "def")
    return tok_def;
  if (IdentifierStr == "extern")
    return tok_extern;
  if (IdentifierStr == "if")
    return tok_if;
  if (IdentifierStr == "then")
    return tok_then;
  if (IdentifierStr == "else")
    return tok_else;
  if (IdentifierStr == "for")
    return tok_for;
  if (IdentifierStr == "in")
    return tok_in;
  if (IdentifierStr == "binary")
    return tok_binary;
  if (IdentifierStr == "unary")
    return tok_unary;
  if (IdentifierStr == "var")
    return tok_var;
  return tok_identifier;
```

词法分析

```
    return tok_identifier;
  }
  // 数字: [0-9.]+
  if (isdigit(LastChar) || LastChar == '.') { // [0-9.]
    std::string NumStr;
    do {
      NumStr += LastChar;
      LastChar = advance();
    } while (isdigit(LastChar) || LastChar == '.'); // [0-9.]*

    NumVal = strtod(NumStr.c_str(), nullptr);
    return tok_number;
  }
  // 注释作用于整行
  if (LastChar == '#') {
    do
      LastChar = advance();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');

    if (LastChar != EOF) // 不是文件结束,就继续读取
      return gettok();
  }

  // 文件结尾是最后一个字符,直接返回,不再继续读取
  if (LastChar == EOF)
    return tok_eof;

  // 其他情况,直接返回字符的ASCII值,用于我们自定义运算符的识别
  int ThisChar = LastChar;
  LastChar = advance();
  return ThisChar;
}
```

语法分析

```
namespace {
// 输出流
raw_ostream &indent(raw_ostream &O, int size) {
  // n个' '
  return O << std::string(size, ' ');
}

/// 表达式抽象语法树，是所有表达式的基类
class ExprAST {
  SourceLocation Loc; // 代码位置

public:
  // 构造，读取当前文件的代码位置,存入AST的Loc
  ExprAST(SourceLocation Loc = CurLoc) : Loc(Loc) {}
  // 析构
  virtual ~ExprAST() {}
  // 代码生成
  virtual Value *codegen() = 0;
  // 输出行数
  int getLine() const { return Loc.Line; }
  // 输出列数
  int getCol() const { return Loc.Col; }
  // 输出行列数
  virtual raw_ostream &dump(raw_ostream &out, int ind) {
    return out << ':' << getLine() << ':' << getCol() << '\n';
  }
};

/// 数字表达式AST
class NumberExprAST : public ExprAST {
  double Val; // 数值
```

语法分析

```
public:
  NumberExprAST(double Val) : Val(Val) {}
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) override {
    return ExprAST::dump(out << Val, ind);
  }
  Value *codegen() override;
};

// 变量表达式AST
class VariableExprAST : public ExprAST {
  std::string Name; // 变量名

public:
  // 构造，代码位置 变量名
  VariableExprAST(SourceLocation Loc, const std::string &Name)
      : ExprAST(Loc), Name(Name) {}
  // 返回变量名
  const std::string &getName() const { return Name; }
  Value *codegen() override;
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) override {
    return ExprAST::dump(out << Name, ind);
  }
};

/// 一元运算符AST
class UnaryExprAST : public ExprAST {
  char Opcode;                        // 运算符
  std::unique_ptr<ExprAST> Operand; // 操作数

public:
```

语法分析

```
public:
  // 构造，运算符 操作数
  UnaryExprAST(char Opcode, std::unique_ptr<ExprAST> Operand)
      : Opcode(Opcode), Operand(std::move(Operand)) {}
  Value *codegen() override;
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) override {
    ExprAST::dump(out << "unary" << Opcode, ind);
    Operand->dump(out, ind + 1);
    return out;
  }
};

// 二元运算符表达式AST
class BinaryExprAST : public ExprAST {
  char Op;                              // 运算符
  std::unique_ptr<ExprAST> LHS, RHS; // 左部,右部

public:
  // 获取代码位置,运算符,左部,右部
  BinaryExprAST(SourceLocation Loc, char Op, std::unique_ptr<ExprAST> LHS,
                std::unique_ptr<ExprAST> RHS)
      : ExprAST(Loc), Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}
  Value *codegen() override;
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) override {
    ExprAST::dump(out << "binary" << Op, ind);
    LHS->dump(indent(out, ind) << "LHS:", ind + 1);
    RHS->dump(indent(out, ind) << "RHS:", ind + 1);
    return out;
  }
};
```

语法分析

```
/// 函数调用表达式AST
class CallExprAST : public ExprAST {
  std::string Callee;                          // 被调用函数名
  std::vector<std::unique_ptr<ExprAST>> Args; // 参数列表

public:
  // 代码位置,被调用函数名,参数列表
  CallExprAST(SourceLocation Loc, const std::string &Callee,
              std::vector<std::unique_ptr<ExprAST>> Args)
      : ExprAST(Loc), Callee(Callee), Args(std::move(Args)) {}
  Value *codegen() override;
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) override {
    ExprAST::dump(out << "call " << Callee, ind);
    for (const auto &Arg : Args)
      Arg->dump(indent(out, ind + 1), ind + 1);
    return out;
  }
};

/// if表达式AST if then else
class IfExprAST : public ExprAST {
  std::unique_ptr<ExprAST> Cond, Then, Else; // 条件,为true,为false

public:
  // 代码位置,条件,为true,为false
  IfExprAST(SourceLocation Loc, std::unique_ptr<ExprAST> Cond,
            std::unique_ptr<ExprAST> Then, std::unique_ptr<ExprAST> Else)
      : ExprAST(Loc), Cond(std::move(Cond)), Then(std::move(Then)),
        Else(std::move(Else)) {}
  Value *codegen() override;
  // 输出行列数
```

语法分析

```
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) override {
    ExprAST::dump(out << "if", ind);
    Cond->dump(indent(out, ind) << "Cond:", ind + 1);
    Then->dump(indent(out, ind) << "Then:", ind + 1);
    Else->dump(indent(out, ind) << "Else:", ind + 1);
    return out;
  }
};

/// for表达式AST for in
class ForExprAST : public ExprAST {
  std::string VarName; // 循环变量名
  // 初始表达式,结束表达式,步长表达式,主体表达式
  std::unique_ptr<ExprAST> Start, End, Step, Body;

public:
  // 循环变量名,初始表达式,结束表达式,步长表达式,主体表达式
  ForExprAST(const std::string &VarName, std::unique_ptr<ExprAST> Start,
             std::unique_ptr<ExprAST> End, std::unique_ptr<ExprAST> Step,
             std::unique_ptr<ExprAST> Body)
      : VarName(VarName), Start(std::move(Start)), End(std::move(End)),
        Step(std::move(Step)), Body(std::move(Body)) {}
  Value *codegen() override;
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) override {
    ExprAST::dump(out << "for", ind);
    Start->dump(indent(out, ind) << "Cond:", ind + 1);
    End->dump(indent(out, ind) << "End:", ind + 1);
    Step->dump(indent(out, ind) << "Step:", ind + 1);
    Body->dump(indent(out, ind) << "Body:", ind + 1);
    return out;
```

语法分析

```
  }
};

/// 可变变量表达式AST: var in
class VarExprAST : public ExprAST {
  // 存储可变变量的vector
  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;
  // 可变变量主体
  std::unique_ptr<ExprAST> Body;

public:
  VarExprAST(
      // 变量名,AST 组成的pair 的 vector
      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; // 代码生成
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) override {
    ExprAST::dump(out << "var", ind);
    for (const auto &NamedVar : VarNames)
      NamedVar.second->dump(indent(out, ind) << NamedVar.first << ':', ind + 1);
    Body->dump(indent(out, ind) << "Body:", ind + 1);
    return out;
  }
};

// 函数声明AST 或 一元二元运算符表达式AST
class PrototypeAST {
  std::string Name;               // 函数名
  std::vector<std::string> Args; // 参数列表
  bool IsOperator;                // 是否为运算符
```

语法分析

```
  unsigned Precedence;            // 二元运算符优先级
  int Line;                       // 行数

public:
  // 代码位置,函数名,函数参数列表,是否为运算符,优先级
  PrototypeAST(SourceLocation Loc, const std::string &Name,
               std::vector<std::string> Args, bool IsOperator = false,
               unsigned Prec = 0)
      : Name(Name), Args(std::move(Args)), IsOperator(IsOperator),
        Precedence(Prec), Line(Loc.Line) {}
  Function *codegen();
  // 获取函数名
  const std::string &getName() const { return Name; }

  // 是否为一元运算符
  bool isUnaryOp() const { return IsOperator && Args.size() == 1; }
  // 是否为二元运算符
  bool isBinaryOp() const { return IsOperator && Args.size() == 2; }

  // 返回运算符名称,返回最后一个字符格式为:
  // "unary"加上一个char字符,所以返回最后一个字符
  char getOperatorName() const {
    assert(isUnaryOp() || isBinaryOp());
    return Name[Name.size() - 1];
  }
  // 获取二元运算符优先级
  unsigned getBinaryPrecedence() const { return Precedence; }
  // 获取行数
  int getLine() const { return Line; }
};

/// 函数AST
```

语法分析

```
    return Name[Name.size() - 1];
  }
  // 获取二元运算符优先级
  unsigned getBinaryPrecedence() const { return Precedence; }
  // 获取行数
  int getLine() const { return Line; }
};

/// 函数AST
class FunctionAST {
  std::unique_ptr<PrototypeAST> Proto; // 函数声明
  std::unique_ptr<ExprAST> Body;       // 函数体

public:
  // 函数声明,函数体
  FunctionAST(std::unique_ptr<PrototypeAST> Proto,
              std::unique_ptr<ExprAST> Body)
      : Proto(std::move(Proto)), Body(std::move(Body)) {}
  Function *codegen();
  // 输出行列数
  raw_ostream &dump(raw_ostream &out, int ind) {
    indent(out, ind) << "FunctionAST\n";
    ++ind;
    indent(out, ind) << "Body:";
    return Body ? Body->dump(out, ind) : out << "null\n";
  }
};
} // end anonymous namespace
```

语义分析

```
static int CurTok; // 存储当前的Token枚举值(类型)
// 从词法分析中获取下一个Token,并赋值给CurTok
static int getNextToken() { return CurTok = gettok(); }

// 二元运算符的优先级,   运算符到优先级数字的映射
static std::map<char, int> BinopPrecedence;

// 查询二元运算符的优先级
static int GetTokPrecedence() {
  if (!isascii(CurTok))
    return -1;

  // 确保已经映射过
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0)
    return -1;
  return TokPrec;
}

// AST解析的错误提示
std::unique_ptr<ExprAST> LogError(const char *Str) {
  fprintf(stderr, "Error: %s\n", Str);
  return nullptr;
}

// 函数声明解析的错误提示
std::unique_ptr<PrototypeAST> LogErrorP(const char *Str) {
  LogError(Str);
  return nullptr;
}

// 表达式解析的声明
```

语义分析

```
// 表达式解析的声明
static std::unique_ptr<ExprAST> ParseExpression();

// numberexpr ::= number
// 数字解析
static std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto Result = std::make_unique<NumberExprAST>(NumVal);
  getNextToken();
  return std::move(Result);
}

// parenexpr ::= '(' expression ')'
// 括号解析
static std::unique_ptr<ExprAST> ParseParenExpr() {
  getNextToken(); // eat (.
  // V为括号内表达式AST
  auto V = ParseExpression();
  if (!V)
    return nullptr;
  // 判断结束条件
  if (CurTok != ')')
    return LogError("expected ')'");
  getNextToken();
  return V;
}

/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static std::unique_ptr<ExprAST> ParseIdentifierExpr() {
  std::string IdName = IdentifierStr; // 变量名
```

语义分析

```
  SourceLocation LitLoc = CurLoc; // 当前代码位置

  getNextToken();
  // 判断是变量还是函数
  if (CurTok != '(')
    return std::make_unique<VariableExprAST>(LitLoc, IdName);

  // 函数调用
  getNextToken();
  std::vector<std::unique_ptr<ExprAST>> Args;
  if (CurTok != ')') {
    while (1) {
      // 解析参数
      if (auto Arg = ParseExpression())
        Args.push_back(std::move(Arg));
      else
        return nullptr;

      if (CurTok == ')')
        break;

      if (CurTok != ',')
        return LogError("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }

  getNextToken();
  // 构造Call AST
  return std::make_unique<CallExprAST>(LitLoc, IdName, std::move(Args));
}
```

语义分析

```
/// ifexpr ::= 'if' expression 'then' expression 'else' expression
static std::unique_ptr<ExprAST> ParseIfExpr() {
  SourceLocation IfLoc = CurLoc; // 记录当前代码位置

  getNextToken();

  // 判断条件
  auto Cond = ParseExpression();
  if (!Cond)
    return nullptr;
  // then
  if (CurTok != tok_then)
    return LogError("expected then");
  getNextToken();

  auto Then = ParseExpression();
  if (!Then)
    return nullptr;
  // else
  if (CurTok != tok_else)
    return LogError("expected else");

  getNextToken();

  auto Else = ParseExpression();
  if (!Else)
    return nullptr;

  return std::make_unique<IfExprAST>(IfLoc, std::move(Cond), std::move(Then),
                                     std::move(Else));
}
```

语义分析

```
/// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression
static std::unique_ptr<ExprAST> ParseForExpr() {
  getNextToken();

  if (CurTok != tok_identifier)
    return LogError("expected identifier after for");

  std::string IdName = IdentifierStr; // 循环变量名
  getNextToken();
  // 初始赋值
  if (CurTok != '=')
    return LogError("expected '=' after for");
  getNextToken();

  auto Start = ParseExpression();
  if (!Start)
    return nullptr;
  if (CurTok != ',')
    return LogError("expected ',' after for start value");
  getNextToken();
  // 结束条件
  auto End = ParseExpression();
  if (!End)
    return nullptr;

  // 可选 步长
  std::unique_ptr<ExprAST> Step;
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (!Step)
      return nullptr;
```

语义分析

```
  }

  if (CurTok != tok_in)
    return LogError("expected 'in' after for");
  getNextToken(); // eat 'in'.
  // for主体
  auto Body = ParseExpression();
  if (!Body)
    return nullptr;

  return std::make_unique<ForExprAST>(IdName, std::move(Start), std::move(End),
                                      std::move(Step), std::move(Body));
}

/// varexpr ::= 'var' identifier ('=' expression)?
//                   (',' identifier ('=' expression)?)* 'in' expression
static std::unique_ptr<ExprAST> ParseVarExpr() {
  getNextToken();

  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>>
      VarNames; // 可变变量列表

  // 至少需要一个参数
  if (CurTok != tok_identifier)
    return LogError("expected identifier after var");

  while (1) {
    std::string Name = IdentifierStr;
    getNextToken();

    // 初始化是可选的
    std::unique_ptr<ExprAST> Init = nullptr;
```

语义分析

```
    if (CurTok == '=') {
      getNextToken(); // eat the '='.

      Init = ParseExpression();
      if (!Init)
        return nullptr;
    }

    VarNames.push_back(std::make_pair(Name, std::move(Init)));

    if (CurTok != ',')
      break;
    getNextToken(); // eat the ','.

    if (CurTok != tok_identifier)
      return LogError("expected identifier list after var");
  }

  // in
  if (CurTok != tok_in)
    return LogError("expected 'in' keyword after 'var'");
  getNextToken();
  // 解析主体
  auto Body = ParseExpression();
  if (!Body)
    return nullptr;

  return std::make_unique<VarExprAST>(std::move(VarNames), std::move(Body));
}

/// primary
///   ::= identifierexpr
```

```
    return nullptr;

  return std::make_unique<VarExprAST>(std::move(VarNames), std::move(Body));
}

/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
///   ::= ifexpr
///   ::= forexpr
///   ::= varexpr
// 首要解析
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  default:
    return LogError("unknown token when expecting an expression");
  case tok_identifier:
    return ParseIdentifierExpr(); // 变量
  case tok_number:
    return ParseNumberExpr(); // 数字
  case '(':
    return ParseParenExpr(); // 括号
  case tok_if:
    return ParseIfExpr(); // if
  case tok_for:
    return ParseForExpr(); // for
  case tok_var:
    return ParseVarExpr(); // var
  }
}
```

语义分析

语义分析

```
/// unary
///   ::= primary
///   ::= '!' unary
static std::unique_ptr<ExprAST> ParseUnary() {
  // 在此判断是否为一元运算符，否则就解析为Primary的表达式
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // 解析一元运算符
  int Opc = CurTok; // 运算符
  getNextToken();
  if (auto Operand = ParseUnary()) // 解析 主体表达式，进入Primary解析
    return std::make_unique<UnaryExprAST>(
        Opc, std::move(Operand)); // 返回 一元运算符的AST
  return nullptr;
}

/// binoprhs
///   ::= ('+' unary)*
// 解析 运算符+RHS的形式，获取LHS和其左边运算符的优先级
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<ExprAST> LHS) {
  while (1) {
    // 为二元运算符获取优先级
    int TokPrec = GetTokPrecedence();

    // 如果当前运算符优先级小于前一运算符,直接返回LHS
    if (TokPrec < ExprPrec)
      return LHS;

    // 获取二元运算符
    int BinOp = CurTok;
```

语义分析

```
    SourceLocation BinLoc = CurLoc;
    getNextToken();

    // 解析RHS
    auto RHS = ParseUnary();
    if (!RHS)
      return nullptr;

    // 判断此运算符优先级是否小于下一运算符
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {
      RHS = ParseBinOpRHS(TokPrec + 1, std::move(RHS));
      if (!RHS)
        return nullptr;
    }

    // 解析完成,合并为 二元表达式的AST
    LHS = std::make_unique<BinaryExprAST>(BinLoc, BinOp, std::move(LHS),
                                          std::move(RHS));
  }
}

/// expression
///   ::= unary binoprhs
// 解析表达式
static std::unique_ptr<ExprAST> ParseExpression() {
  // 解析一元表达式,如果不是的话,会进行parse primary
  auto LHS = ParseUnary();
  if (!LHS)
    return nullptr;

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

语义分析

```
}

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

  SourceLocation FnLoc = CurLoc; // 代码位置

  unsigned Kind = 0;                // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30; // 默认优先级,在二元未给出优先级时使用
  // 判断是变量,一元运算符,二元远算符
  switch (CurTok) {
  default:
    return LogErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_unary:
    getNextToken();
    if (!isascii(CurTok))
      return LogErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok; // 函数名为 unary+运算符
    Kind = 1;
    getNextToken();
    break;
```

语义分析

```
case tok_binary:
  getNextToken();
  if (!isascii(CurTok))
    return LogErrorP("Expected binary operator");
  FnName = "binary";
  FnName += (char)CurTok; // 函数名为 binary+运算符
  Kind = 2;
  getNextToken();

  // 在给出优先级时使用给二元运算符
  if (CurTok == tok_number) {
    if (NumVal < 1 || NumVal > 100)
      return LogErrorP("Invalid precedence: must be 1..100");
    BinaryPrecedence = (unsigned)NumVal;
    getNextToken();
  }
  break;
}
// 检查函数声明的括号
if (CurTok != '(')
  return LogErrorP("Expected '(' in prototype");
// 参数列表
std::vector<std::string> ArgNames;
while (getNextToken() == tok_identifier)
  ArgNames.push_back(IdentifierStr);
if (CurTok != ')')
  return LogErrorP("Expected ')' in prototype");

getNextToken();

// 验证运算符的类型和参数个数是否相符
if (Kind && ArgNames.size() != Kind)
```

语义分析

```
  // 验证运算符的类型和参数个数是否相符
  if (Kind && ArgNames.size() != Kind)
    return LogErrorP("Invalid number of operands for operator");
  // 构造函数,Kind!=0用以返回布尔值
  return std::make_unique<PrototypeAST>(FnLoc, FnName, ArgNames, Kind != 0,
                                        BinaryPrecedence);
}

/// definition ::= 'def' prototype expression
// 解析函数定义
static std::unique_ptr<FunctionAST> ParseDefinition() {
  getNextToken();
  auto Proto = ParsePrototype(); // 函数声明
  if (!Proto)
    return nullptr;

  if (auto E = ParseExpression()) // 函数体
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  return nullptr;
}

/// toplevelexpr ::= expression
// 将顶层表达式解析为匿名函数
static std::unique_ptr<FunctionAST> ParseTopLevelExpr() {
  SourceLocation FnLoc = CurLoc;
  if (auto E = ParseExpression()) {
    // 构建匿名函数声明，未依照教程修改为main
    auto Proto = std::make_unique<PrototypeAST>(FnLoc, "__anon_expr",
                                                std::vector<std::string>());
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  }
  return nullptr;
}
```

语义分析

```
  if (auto E = ParseExpression()) // 函数体
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  return nullptr;
}

/// toplevelexpr ::= expression
// 将顶层表达式解析为匿名函数
static std::unique_ptr<FunctionAST> ParseTopLevelExpr() {
  SourceLocation FnLoc = CurLoc;
  if (auto E = ParseExpression()) {
    // 构建匿名函数声明，未依照教程修改为main
    auto Proto = std::make_unique<PrototypeAST>(FnLoc, "__anon_expr",
                                                std::vector<std::string>());
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  }
  return nullptr;
}

/// external ::= 'extern' prototype
// 解析extern
static std::unique_ptr<PrototypeAST> ParseExtern() {
  getNextToken(); // eat extern.
  return ParsePrototype();
}
```

调试信息DI

```
static std::unique_ptr<DIBuilder> DBuilder; // Debug Builder
// DI获取double类型
DIType *DebugInfo::getDoubleTy() {
  if (DblTy)
    return DblTy;
  DblTy = DBuilder->createBasicType("double", 64, dwarf::DW_ATE_float);
  return DblTy;
}
// 为AST,生成DI的代码位置，更新Builder的位置记录
void DebugInfo::emitLocation(ExprAST *AST) {
  if (!AST)
    return Builder.SetCurrentDebugLocation(DebugLoc());
  // 记录作用域信息
  DIScope *Scope;
  if (LexicalBlocks.empty())
    Scope = TheCU;
  else
    Scope = LexicalBlocks.back();
  Builder.SetCurrentDebugLocation(
      DebugLoc::get(AST->getLine(), AST->getCol(), Scope));
}
// 创建函数DI,包含函数参数数量,文件位置和名
static DISubroutineType *CreateFunctionType(unsigned NumArgs, DIFile *Unit) {
  SmallVector<Metadata *, 8> EltTys;
  DIType *DblTy = KSDbgInfo.getDoubleTy();
  // 结果的类型Double
  EltTys.push_back(DblTy);
  // 对应参数数量的类型
  for (unsigned i = 0, e = NumArgs; i != e; ++i)
    EltTys.push_back(DblTy);
  return DBuilder->createSubroutineType(DBuilder->getOrCreateTypeArray(EltTys));
}
```

代码生成

```
static std::unique_ptr<Module> TheModule; // 构造module
static std::map<std::string, AllocaInst *>
    NamedValues; // 变量名到Alloca值的映射
static std::unique_ptr<KaleidoscopeJIT> TheJIT; // 构造JIT
static std::map<std::string, std::unique_ptr<PrototypeAST>>
    FunctionProtos; // 函数名,函数声明AST

// 代码生成时的错误处理
Value *LogErrorV(const char *Str) {
  LogError(Str);
  return nullptr;
}
// 查找函数
Function *getFunction(std::string Name) {
  // 查找module中是否已经定义了函数
  if (auto *F = TheModule->getFunction(Name))
    return F;

  // 未找到函数定义,查找函数声明
  auto FI = FunctionProtos.find(Name);
  if (FI != FunctionProtos.end())
    return FI->second->codegen();

  // 其他情况,返回空
  return nullptr;
}

// 在entry的BasicBlock中 创建alloca指令,
// 此处为可变变量所设,本次未经优化,可直接看到alloca指令
static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction,
                                          StringRef VarName) {
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
```

代码生成

```
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                   TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(TheContext), nullptr, VarName);
}
// 生成数字代码
Value *NumberExprAST::codegen() {
  KSDbgInfo.emitLocation(this); // AST生成代码时发出该位置
  return ConstantFP::get(TheContext, APFloat(Val));
}
// 生成变量
Value *VariableExprAST::codegen() {
  // 在函数中查找变量
  Value *V = NamedValues[Name];
  if (!V)
    return LogErrorV("Unknown variable name");

  KSDbgInfo.emitLocation(this); // AST生成代码时发出该位置
  // 加载变量
  return Builder.CreateLoad(V, Name.c_str());
}
// 生成一元运算符AST代码
Value *UnaryExprAST::codegen() {
  Value *OperandV = Operand->codegen(); // 主体生成
  if (!OperandV)
    return nullptr;
  // 查找函数 "unary"+运算符
  Function *F = getFunction(std::string("unary") + Opcode);
  if (!F)
    return LogErrorV("Unknown unary operator");

  KSDbgInfo.emitLocation(this); // AST生成代码时发出该位置
  return Builder.CreateCall(F, OperandV, "unop");
```

```
}
// 生成二元运算符AST代码
Value *BinaryExprAST::codegen() {
  KSDbgInfo.emitLocation(this); // AST生成代码时发出该位置

  // 特别处理=，因为左部不需要解析为表达式
  if (Op == '=') {
    // 此处预设变量为Double类型
    VariableExprAST *LHSE = static_cast<VariableExprAST *>(LHS.get());
    if (!LHSE)
      return LogErrorV("destination of '=' must be a variable");
    // 生成右部
    Value *Val = RHS->codegen();
    if (!Val)
      return nullptr;

    // 查找变量名是否定义
    Value *Variable = NamedValues[LHSE->getName()];
    if (!Variable)
      return LogErrorV("Unknown variable name");
    // 存储函数值,函数名
    Builder.CreateStore(Val, Variable);
    return Val;
  }
  // 表达式左部,右部
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  switch (Op) {
  case '+':
```

代码生成

代码生成

```
  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");
    // 将0/1转换为Double
    return Builder.CreateUIToFP(L, Type::getDoubleTy(TheContext), "booltmp");
  default:
    break;
  }

  // 如果不是内建二元运算符,那就是自定义的, 构造响应函数
  Function *F = getFunction(std::string("binary") + Op);
  assert(F && "binary operator not found!");

  Value *Ops[] = {L, R};
  return Builder.CreateCall(F, Ops, "binop");
}
// 函数调用生成代码
Value *CallExprAST::codegen() {
  KSDbgInfo.emitLocation(this); // AST生成代码时发出该位置

  // 查找被调函数名
  Function *CalleeF = getFunction(Callee);
  if (!CalleeF)
    return LogErrorV("Unknown function referenced");
```

代码生成

```
  // 匹配参数个数
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV("Incorrect # arguments passed");

  // 参数解析
  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->codegen());
    if (!ArgsV.back())
      return nullptr;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
// 生成if
Value *IfExprAST::codegen() {
  KSDbgInfo.emitLocation(this); // AST生成代码时发出该位置
  // 条件生成
  Value *CondV = Cond->codegen();
  if (!CondV)
    return nullptr;

  // 通过和0.0是否相等判断,生成i1布尔值
  CondV = Builder.CreateFCmpONE(
      CondV, ConstantFP::get(TheContext, APFloat(0.0)), "ifcond");
  // 获取函数代码插入点
  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // then块插入到函数后
  BasicBlock *ThenBB = BasicBlock::Create(TheContext, "then", TheFunction);
  // else块
  BasicBlock *ElseBB = BasicBlock::Create(TheContext, "else");
```

代码生成

```
BasicBlock *ElseBB = BasicBlock::Create(TheContext, "else");
// ifcont
BasicBlock *MergeBB = BasicBlock::Create(TheContext, "ifcont");
// br语句
Builder.CreateCondBr(CondV, ThenBB, ElseBB);

// then生成
Builder.SetInsertPoint(ThenBB);

Value *ThenV = Then->codegen();
if (!ThenV)
  return nullptr;
// 创建then->ifcont
Builder.CreateBr(MergeBB);
// 在phi中更新then
ThenBB = Builder.GetInsertBlock();

// 生成else块
TheFunction->getBasicBlockList().push_back(ElseBB);
Builder.SetInsertPoint(ElseBB);

Value *ElseV = Else->codegen();
if (!ElseV)
  return nullptr;
// 创建else->ifcont
Builder.CreateBr(MergeBB);
// 在phi中更新else
ElseBB = Builder.GetInsertBlock();

// 生成合并分支
TheFunction->getBasicBlockList().push_back(MergeBB);
Builder.SetInsertPoint(MergeBB);
```

```
  Builder.SetInsertPoint(MergeBB);
  PHINode *PN = Builder.CreatePHI(Type::getDoubleTy(TheContext), 2, "iftmp");
  // 添加then和else到phi中, [value,label]
  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}

// Output for-loop as:
//   var = alloca double
//   ...
//   start = startexpr
//   store start -> var
//   goto loop
// loop:
//   ...
//   bodyexpr
//   ...
// loopend:
//   step = stepexpr
//   endcond = endexpr
//
//   curvar = load var
//   nextvar = curvar + step
//   store nextvar -> var
//   br endcond, loop, endloop
// outloop:
Value *ForExprAST::codegen() {
  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // 为entry中变量用alloca创建
  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
```

代码生成

代码生成

```
KSDbgInfo.emitLocation(this); // AST生成代码时发出该位置

// 获得循环的初始值
Value *StartVal = Start->codegen();
if (!StartVal)
  return nullptr;

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

// 在当前块后,插入loop
BasicBlock *LoopBB = BasicBlock::Create(TheContext, "loop", TheFunction);
// br进入loop
Builder.CreateBr(LoopBB);
// 记录loop的陷入位置
Builder.SetInsertPoint(LoopBB);

// 保存旧值,获取新值
AllocaInst *OldVal = NamedValues[VarName];
NamedValues[VarName] = Alloca;

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

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

代码生成

```
  return nullptr;
} else {
  // 默认1.0
  StepVal = ConstantFP::get(TheContext, APFloat(1.0));
}

// 计算结束条件
Value *EndCond = End->codegen();
if (!EndCond)
  return nullptr;

// 重新载入,增加,重新储存alloca,以更新可变变量
Value *CurVar = Builder.CreateLoad(Alloca, VarName.c_str());
Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
Builder.CreateStore(NextVar, Alloca);

// 获取布尔值,判断退出条件
EndCond = Builder.CreateFCmpONE(
    EndCond, ConstantFP::get(TheContext, APFloat(0.0)), "loopcond");

// 创建after loop并插入
BasicBlock *AfterBB =
    BasicBlock::Create(TheContext, "afterloop", TheFunction);

// br语句
Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

// 后续代码都被插入after loop
Builder.SetInsertPoint(AfterBB);

// 恢复旧值
if (OldVal)
  NamedValues[VarName] = OldVal;
```

代码生成

```
  else
    NamedValues.erase(VarName);

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

// 可变变量AST的代码生成函数
Value *VarExprAST::codegen() {
  std::vector<AllocaInst *> OldBindings; // 分配stack上内存
  // 获取函数代码插入点
  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // 获取变量并初始化
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first; // first变量名
    ExprAST *Init = VarNames[i].second.get();       // 变量值

    // 初始化
    Value *InitVal;
    if (Init) {
      InitVal = Init->codegen();
      if (!InitVal)
        return nullptr;
    } else { // 初始化默认值为0.0
      InitVal = ConstantFP::get(TheContext, APFloat(0.0));
    }
    // 在entry块中创建alloca指令
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
    Builder.CreateStore(InitVal, Alloca);

    // 还原旧值，为循环变量最后一次生效的值
```

代码生成

```
    // 还原绑定
    NamedValues[VarName] = Alloca;
  }

  KSDbgInfo.emitLocation(this); // AST生成代码时发出该位置

  // 生成主体,所有可变变量都在域内
  Value *BodyVal = Body->codegen();
  if (!BodyVal)
    return nullptr;

  // 弹出可变变量从域内
  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;
}
// 函数声明代码生成
Function *PrototypeAST::codegen() {
  // 构建:  double(double,double)
  std::vector<Type *> Doubles(Args.size(), Type::getDoubleTy(TheContext));
  FunctionType *FT =
      FunctionType::get(Type::getDoubleTy(TheContext), Doubles, false);

  Function *F =
      Function::Create(FT, Function::ExternalLinkage, Name, TheModule.get());

  // 为参数设置名
  unsigned Idx = 0;
  for (auto &Arg : F->args())
    Arg.setName(Args[Idx++]);
```

代码生成

```
    Arg.setName(Args[Idx++]);

  return F;
}

Function *FunctionAST::codegen() {

  // 将函数声明转为函数，保留引用为后续使用
  auto &P = *Proto;
  FunctionProtos[Proto->getName()] = std::move(Proto);
  Function *TheFunction = getFunction(P.getName());
  if (!TheFunction)
    return nullptr;

  // 如果是二元运算符,记录其优先级
  if (P.isBinaryOp())
    BinopPrecedence[P.getOperatorName()] = P.getBinaryPrecedence();

  // 创建entry BasicBlock
  BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);
  Builder.SetInsertPoint(BB);

  // 根据CompileUnit和源代码中代码位置将函数定义添加到调试信息中
  DIFile *Unit = DBuilder->createFile(KSDbgInfo.TheCU->getFilename(),
                                      KSDbgInfo.TheCU->getDirectory());
  DIScope *FContext = Unit;
  unsigned LineNo = P.getLine();
  unsigned ScopeLine = LineNo;
  // DISubprogram，其中包含对该函数所有元数据的引用
  DISubprogram *SP = DBuilder->createFunction(
      FContext, P.getName(), StringRef(), Unit, LineNo,
      CreateFunctionType(TheFunction->arg_size(), Unit), ScopeLine,
```

代码生成

```
    CreateFunctionType(TheFunction->arg_size(), Unit), ScopeLine,
    DINode::FlagPrototyped, DISubprogram::SPFlagDefinition);
TheFunction->setSubprogram(SP);

// 为每个函数生成代码时，将作用域（函数）推到栈顶
KSDbgInfo.LexicalBlocks.push_back(SP);

// 不为前序设置位置 函数中没有位置的引导指令被视为前序，
// 调试器会在中断函数时跳过
KSDbgInfo.emitLocation(nullptr);

// NamedValues清空,并记录此次参数列表
NamedValues.clear();
unsigned ArgIdx = 0;
for (auto &Arg : TheFunction->args()) {
  // Create an alloca for this variable.
  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName());

  // 为变量添加调试描述
  DILocalVariable *D = DBuilder->createParameterVariable(
      SP, Arg.getName(), ++ArgIdx, Unit, LineNo, KSDbgInfo.getDoubleTy(),
      true);

  DBuilder->insertDeclare(Alloca, D, DBuilder->createExpression(),
                          DebugLoc::get(LineNo, 0, SP),
                          Builder.GetInsertBlock());

  // 将初始值存入 alloca
  Builder.CreateStore(&Arg, Alloca);

  // 将参数加入到变量符号表中
  NamedValues[std::string(Arg.getName())] = Alloca;
```

代码生成

```
    // 将参数加入到变量符号表中
    NamedValues[std::string(Arg.getName())] = Alloca;
  }

  KSDbgInfo.emitLocation(Body.get()); // 函数主体生成代码,获得一个新位置

  if (Value *RetVal = Body->codegen()) {
    // 完成函数
    Builder.CreateRet(RetVal);

    // 函数的代码生成结束时将作用域弹出
    KSDbgInfo.LexicalBlocks.pop_back();

    // 验证函数
    verifyFunction(*TheFunction);

    return TheFunction;
  }

  // 函数体读取错误,删除函数
  TheFunction->eraseFromParent();

  if (P.isBinaryOp())
    BinopPrecedence.erase(Proto->getOperatorName());

  // 弹出函数的作用域
  KSDbgInfo.LexicalBlocks.pop_back();
  // 生成无效
  return nullptr;
}
```

顶层解析和JIT

```
static void InitializeModule() {
  // 新建一个module
  TheModule = std::make_unique<Module>("my cool jit", TheContext);
  TheModule->setDataLayout(TheJIT->getTargetMachine().createDataLayout());
}
// 处理函数定义,包括自定义的一元二元运算符
static void HandleDefinition() {
  if (auto FnAST = ParseDefinition()) {
    if (!FnAST->codegen())
      fprintf(stderr, "Error reading function definition:");
  } else {
    getNextToken();
  }
}
// 处理extern的函数声明,一般用于使用拓展的库函数
static void HandleExtern() {
  if (auto ProtoAST = ParseExtern()) {
    if (!ProtoAST->codegen())
      fprintf(stderr, "Error reading extern");
    else
      FunctionProtos[ProtoAST->getName()] = std::move(ProtoAST);
  } else {
    getNextToken();
  }
}
// 处理顶层表达式,用以解析只有表达式的情况
static void HandleTopLevelExpression() {
  if (auto FnAST = ParseTopLevelExpr()) {
    if (!FnAST->codegen()) {
      fprintf(stderr, "Error generating code for top level expr");
    }
  } else {
```

顶层解析和JIT

```
  if (auto FnAST = ParseTopLevelExpr()) {
    if (!FnAST->codegen()) {
      fprintf(stderr, "Error generating code for top level expr");
    }
  } else {
    getNextToken();
  }
}

/// top ::= definition | external | expression | ';'
// 主循环,负责跳转到不同的解析流程
static void MainLoop() {
  while (1) {
    switch (CurTok) {
    case tok_eof: // 检测到终止符，跳出循环
      return;
    case ';': // 忽略顶层表达式的分号
      getNextToken();
      break;
    case tok_def: // def开头的函数
      HandleDefinition();
      break;
    case tok_extern: // extern开头的拓展库函数
      HandleExtern();
      break;
    default: // 默认情况下处理为顶层表达式
      HandleTopLevelExpression();
      break;
    }
  }
}
```

拓展库函数

```
#ifdef _WIN32 // Win32会自行定义,但是Mac不会
#define DLLEXPORT __declspec(dllexport)
#else
#define DLLEXPORT
#endif

// 读取一个double数
extern "C" DLLEXPORT double putchard(double X) {
  fputc((char)X, stderr);
  return 0;
}

// 输出一个double数
extern "C" DLLEXPORT double printd(double X) {
  fprintf(stderr, "%f\n", X);
  return 0;
}
```

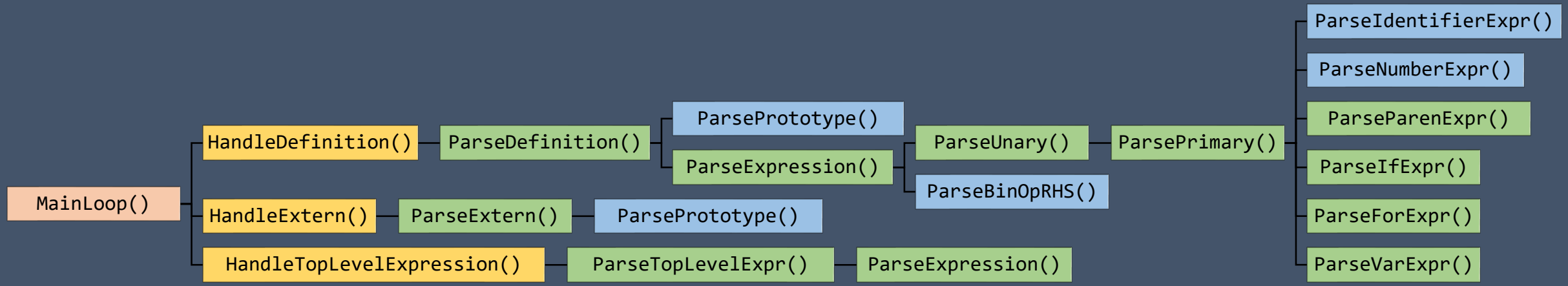
主程序部分

```
int main() {
  // 初始化本机参数
  InitializeNativeTarget();
  InitializeNativeTargetAsmPrinter();
  InitializeNativeTargetAsmParser();

  // 存储二元运算符的优先级,0-100,越大越高
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
  BinopPrecedence['/'] = 40;
  BinopPrecedence['*'] = 40;

  // 读取第一个token
  getNextToken();
  // 构建JIT
  TheJIT = std::make_unique<KaleidoscopeJIT>();
  // 初始化Module
  InitializeModule();

  //增加当前debug版本信息
  TheModule->addModuleFlag(Module::Warning, "Debug Info Version",
                           DEBUG_METADATA_VERSION);

  // Darwin 只支持 dwarf2.
  if (Triple(sys::getProcessTriple()).isOSDarwin())
    TheModule->addModuleFlag(llvm::Module::Warning, "Dwarf Version", 2);

  // 用module构造Debug Info的Builder
  DBuilder = std::make_unique<DIBuilder>(*TheModule);
```

主程序部分

```
  // Darwin 只支持 dwarf2.
  if (Triple(sys::getProcessTriple()).isOSDarwin())
    TheModule->addModuleFlag(llvm::Module::Warning, "Dwarf Version", 2);

  // 用module构造Debug Info的Builder
  DBuilder = std::make_unique<DIBuilder>(*TheModule);

  // 创建module的编译单元
  // 从输入中获取代码是,存为fib.ks
  // 推荐使用文件的实际路径
  KSDbgInfo.TheCU = DBuilder->createCompileUnit(
      dwarf::DW_LANG_C, DBuilder->createFile("fib.ks", "."),
      "Kaleidoscope Compiler", 0, "", 0);

  // 进入程序的主循环
  MainLoop();

  // 完善debug信息
  DBuilder->finalize();

  // 打印出所有生成的代码
  TheModule->print(errs(), nullptr);

  return 0; // 程序运行完成，返回值为0
}
```

解析结构


MainLoop()
HandleDefinition()
ParseDefinition()
ParsePrototype()
ParseExpression()
ParseUnary()
ParsePrimary()
ParseIdentifierExpr()
ParseNumberExpr()
ParseParenExpr()
ParseIfExpr()
ParseForExpr()
ParseVarExpr()
ParseBinOpRHS()
HandleExtern()
ParseExtern()
ParsePrototype()
HandleTopLevelExpression()
ParseTopLevelExpr()
ParseExpression()

结束

- Kaleidoscope JIT:
  - http://llvm.org/docs/tutorial/BuildingAJIT1.html
- 欢迎加入PLCT实验室!
  - https://github.com/isrc-cas/PLCT-Weekly/blob/master/interns.md