namespace {

class ExprAST {

public:

virtual ~ExprAST() = default;};

class NumberExprAST : public ExprAST {

double Val;

public:

NumberExprAST(double Val) : Val(Val) {}

};

上述代码定义了基类ExprAST和一个用于表示数值常量的子类。其中子类NumberExprAST将数值常量的值存放在成员变量中，以备编译器后续查询。直到目前为止，我们还只搭出了AST的架子，尚未定义任何能够体现AST实用价值的成员方法。例如，只需添加一套虚方法，我们就可以轻松实现代码的格式化打印。以下定义了Kaleidoscope语言最基本的部分所要用到的其他各种表达式的AST节点

class VariableExprAST : public ExprAST {

std::string Name;

public:

VariableExprAST(const std::string &Name) : Name(Name) {}

};

class BinaryExprAST : public ExprAST {

char Op;

std::unique\_ptr<ExprAST> LHS, RHS;

public:

BinaryExprAST(char Op, std::unique\_ptr<ExprAST> LHS,

std::unique\_ptr<ExprAST> RHS)

: Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}

};

class CallExprAST : public ExprAST {

std::string Callee;

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

public:

CallExprAST(const std::string &Callee,

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

: Callee(Callee), Args(std::move(Args)) {}};

VariableExprAST用于保存变量名，BinaryExprAST用于保存运算符（如“+”），CallExprAST用于保存函数名和用作参数的表达式列表。这样设计AST有一个优势，那就是我们无须关注语法便可直接抓住语言本身的特性。注意这里还没有涉及到二元运算符的优先级和词法结构等问题。

定义完毕这几种表达式节点，就足以描述Kaleidoscope语言中的几个最基本的结构了。由于我们还没有实现条件控制流程，它还不算图灵完备；后续还会加以完善。接下来，还有两种结构需要描述，即函数的接口和函数本身：

class PrototypeAST {

std::string Name;

std::vector<std::string> Args;

public:

PrototypeAST(const std::string &Name, std::vector<std::string> Args)

: Name(Name), Args(std::move(Args)) {}

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

class FunctionAST {

std::unique\_ptr<PrototypeAST> Proto;

std::unique\_ptr<ExprAST> Body;

public:

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

std::unique\_ptr<ExprAST> Body)

: Proto(std::move(Proto)), Body(std::move(Body)) {}

};

在Kaleidoscope中，函数的类型是由参数的个数决定的。由于所有的值都是双精度浮点数，没有必要保存参数的类型。在更强大、更实用的语言中，ExprAST类多半还会需要一个类型字段。

有了这些作为基础，我们就可以开始解析Kaleidoscope的表达式和函数体了。

}

static int CurTok;

static int getNextToken() { return CurTok = gettok(); }

这段代码以词法分析器为中心，实现了一个简易的语元缓冲，让我们能够预先读取词法分析器将要返回的下一个语元。在我们的语法解析器中，所有函数都将CurTok视作当前待解析的语元。

static std::map<char, int> BinopPrecedence;

static int GetTokPrecedence() {

if (!isascii(CurTok))

return -1;

int TokPrec = BinopPrecedence[CurTok];

if (TokPrec <= 0)

return -1;

return TokPrec;

}

std::unique\_ptr<ExprAST> LogError(const char \*Str) {

fprintf(stderr, "Error: %s\n", Str);

return nullptr;}

std::unique\_ptr<PrototypeAST> LogErrorP(const char \*Str) {

LogError(Str);

return nullptr;

}

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

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

auto Result = llvm::make\_unique<NumberExprAST>(NumVal);

getNextToken();

return std::move(Result);}

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

getNextToken();

auto V = ParseExpression();

if (!V)

return nullptr;

if (CurTok != ')')

return LogError("expected ')'");

getNextToken();

return V;

}

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

std::string IdName = IdentifierStr;

getNextToken();

if (CurTok != '(')

return llvm::make\_unique<VariableExprAST>(IdName);

getNextToken();

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

if (CurTok != ')') {

while (true) {

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();

}

}

该函数与其它函数的风格别无二致。（调用该函数时当前语元必须是tok\_identifier。）前文提到的有关递归和错误处理的特点它统统具备。这里采用了**预读**（lookahead）的手段来试探当前标识符的类型，判断它究竟是个独立的变量引用还是个函数调用。只要检查紧跟标识符之后的语元是不是“(”，它就能知道到底应该构造VariableExprAST节点还是CallExprAST节点。

getNextToken();

return llvm::make\_unique<CallExprAST>(IdName, std::move(Args));

}

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

switch (CurTok) {

default:

return LogError("unknown token when expecting an expression");

case tok\_identifier:

return ParseIdentifierExpr();

case tok\_number:

return ParseNumberExpr();

case '(':

return ParseParenExpr();

}}

static std::unique\_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,

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

while (true) {

int TokPrec = GetTokPrecedence();

if (TokPrec < ExprPrec)

return LHS;

int BinOp = CurTok;

getNextToken(); // eat binop

auto RHS = ParsePrimary();

if (!RHS)

return nullptr;

int NextPrec = GetTokPrecedence();

if (TokPrec < NextPrec) {

RHS = ParseBinOpRHS(TokPrec + 1, std::move(RHS));

if (!RHS)

return nullptr;

}

LHS = llvm::make\_unique<BinaryExprAST>(BinOp, std::move(LHS),

std::move(RHS));

}}

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

auto LHS = ParsePrimary();

if (!LHS)

return nullptr;

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

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

if (CurTok != tok\_identifier)

return LogErrorP("Expected function name in prototype");

std::string FnName = IdentifierStr;

getNextToken();

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");

getNextToken();

return llvm::make\_unique<PrototypeAST>(FnName, std::move(ArgNames));

}

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

getNextToken();

auto Proto = ParsePrototype();

if (!Proto)

return nullptr;

if (auto E = ParseExpression())

return llvm::make\_unique<FunctionAST>(std::move(Proto), std::move(E));

return nullptr;}

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

if (auto E = ParseExpression()) {

auto Proto = llvm::make\_unique<PrototypeAST>("\_\_anon\_expr",

std::vector<std::string>());

return llvm::make\_unique<FunctionAST>(std::move(Proto), std::move(E));

}

return nullptr;

}

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

getNextToken(); // eat extern.

return ParsePrototype();

}

static void HandleDefinition() {

if (ParseDefinition()) {

fprintf(stderr, "Parsed a function definition.\n");

} else {

getNextToken();

}}

static void HandleExtern() {

if (ParseExtern()) {

fprintf(stderr, "Parsed an extern\n");

} else {

getNextToken();

}}

static void HandleTopLevelExpression() {

if (ParseTopLevelExpr()) {

fprintf(stderr, "Parsed a top-level expr\n");

} else {

getNextToken();

}}

static void MainLoop() {

while (true) {

fprintf(stderr, "ready> ");

switch (CurTok) {

case tok\_eof:

return;

case ';':

getNextToken();

break;

case tok\_def:

HandleDefinition();

break;

case tok\_extern:

HandleExtern();

break;

default:

HandleTopLevelExpression();

break;

}

}}

组员：

阿依多斯 2016302580021

周彪 2016302580062