

# 武汉大学计算机学院

## 本科生课程设计报告

### 解释器构造总体设计与实现

专 业 名 称：软件工程

课 程 名 称：《解释器构造实践》

团 队 名 称：解释器小组

指 导 教 师：李清安 副教授

团 队 成 员 一：段梦梦（2016302580001）

团 队 成 员 二：李远（2015302580279）

团 队 成 员 三：何万伟（2016302580125）

团 队 成 员 四：王涛（2016302580147）

团 队 成 员 五：段小式（2016302580006）

二〇一八年十一月

# 郑 重 声 明

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名：\_\_\_\_\_ 日期：\_\_\_\_\_

## 摘要

本次实验的实验目的是实现一个基于 LLVM 框架的 VSL 语言的解释器，使用 GCC/Clang 来编译程序，并运行编译生成的可执行程序。主要包括词法分析、语法分析、生成 IR 代码、JIT 优化和扩展 if/then/else、for 循环和变量的词法、语法部分等，同时实现了赋值、PRINT 语句部分，最后成功的生成了二进制代码。

**关键词：**LLVM；IR；二进制；

# 目录

1	项目地址.....	6
2	设计与实现.....	6
2.1	词法分析器.....	6
2.1.1	关键字.....	7
2.1.2	标识符.....	7
2.1.3	数值.....	7
2.1.4	注释.....	7
2.1.5	字符串.....	7
2.2	语法分析器.....	8
2.2.1	解析表达式.....	8
2.2.2	解析函数.....	8
2.3	IR 代码生成.....	9
2.4	JIT 优化.....	10
2.5	流程控制.....	11
2.5.1	IF 条件语句.....	11
2.5.2	WHILE 循环语句.....	13
2.6	赋值语句.....	14
2.7	二进制代码生成与 PRINT 语句实现.....	15
3	测试.....	17

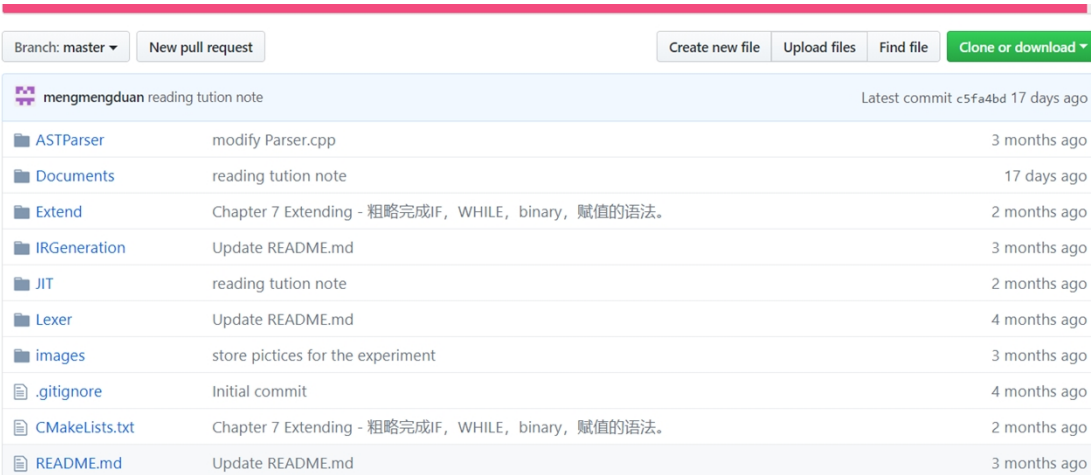
3.1 子任务测试 .....	17
3.1.1 语法分析器测试 .....	17
3.1.2 IR 代码生成 .....	18
3.1.3 IF 语句 .....	19
3.1.4 WHILE 语句 .....	20
3.2 总体测试 .....	20
3.2.1 测试代码 .....	20
3.2.2 测试结果 .....	22
4 项目总结 .....	22
4.1 成员的组成与分工 .....	22
4.1.1 成员组成 .....	22
4.1.2 成员分工 .....	22
4.2 李远 .....	23
4.2.1 个人贡献 .....	23
4.2.2 挑战与解决思路 .....	23
4.3 何万伟 .....	24
4.3.1 个人贡献 .....	24
4.3.2 挑战与解决思路 .....	24
4.4 段梦梦 .....	25
4.4.1 个人贡献 .....	25

4.4.2 挑战与解决思路 .....	25
4.5 王涛 .....	26
4.5.1 个人贡献 .....	26
4.5.2 挑战与解决思路 .....	26
4.6 段小式 .....	26
4.6.1 个人贡献 .....	26
4.6.2 挑战与解决思路 .....	26
5 项目感想与建议 .....	26
5.1 李远 .....	27
5.1.1 感想 .....	27
5.1.2 建议 .....	27
5.2 何万伟 .....	27
5.2.1 感想 .....	27
5.2.2 建议 .....	28
5.3 段梦梦 .....	28
5.3.1 感想 .....	28
5.3.2 建议 .....	28
5.4 王涛 .....	29
5.4.1 感想 .....	29
5.4.2 建议 .....	29

5.5 段小式.....	30
5.5.1 感想.....	30
5.5.2 建议.....	30

## 1 项目地址

本项目将代码放于 Github 代码托管网站之上，便于组内各个成员之间分工合作与代码提交，其网址为 <https://github.com/Interpretion/Interpreter> 其中包括了每个小组成员的 commit 以及编译器的各个子任务与整合，项目结构如下图所示：



Branch: master ▾ New pull request		Create new file Upload files Find file Clone or download ▾
mengmengduan reading tuton note		Latest commit c5fa4bd 17 days ago
ASTParser	modify Parser.cpp	3 months ago
Documents	reading tuton note	17 days ago
Extend	Chapter 7 Extending - 粗略完成if, WHILE, binary, 赋值的语法。	2 months ago
IRGeneration	Update README.md	3 months ago
JIT	reading tuton note	2 months ago
Lexer	Update README.md	4 months ago
images	store pictices for the experiment	3 months ago
.gitignore	Initial commit	4 months ago
CMakeLists.txt	Chapter 7 Extending - 粗略完成if, WHILE, binary, 赋值的语法。	2 months ago
README.md	Update README.md	3 months ago

## 2 设计与实现

本次实验基于 LLVM 框架、GCC，实现 VSL 类 C 语言编译器，使用 GCC/Clang 来编译程序，并运行编译生成的可执行程序。主要包括词法分析、语法分析、生成 IR 代码、JIT 优化和扩展 if/then/else、for 循环和变量的词法、语法部分等。

## 2.1 词法分析器

我们首先将 VSL 语言中代码语句分割成一个个 token，对于每个 token 我们给它附上一个编号，这里采用枚举类型和字符的 ASCII 码值（0-255）来实现这个功能，在枚举类型中设定 VSL 语言中关键字的编号，用 ASCII 码值来标识所遇到的其余字符。项目中不需要手动分割整体代码为 token，而是采取边读取字符边分析的策略，利用 C 语言的 `getchar()` 函数每次只读取一个字符直到识别到一个 token 进行然后进行编号。

### 2.1.1 关键字

将 VSL 语言中的关键字纳入枚举类型中进行编号，如果当前语元是一个关键字就直接返回该关键字在枚举类型中设定的数值。

```
enum Token {
    tok_eof = -1,

    // commands
    tok_FUNC = -2,
    tok_PRINT = -3,
    tok_RETURN = -4,
    tok_CONTINUE = -5,

    // primary
    tok_identifier = -6,
    tok_number = -7,

    // control
    tok_IF = -8,
    tok_THEN = -9,
    tok_ELSE = -10,
    tok_FI = -11,
    tok_WHILE = -12,
    tok_DO = -13,
    tok_DONE = -14,

    // operators
    tok_binary = -15,
    tok_unary = -16,

    // var definition
    tok_VAR = -17,
```

### 2.1.2 标识符

如果当前语元是标识符，我们使用全局变量 `IdentifierStr` 来存储所遇到的标识符。

### 2.1.3 数值

因为在 VSL 语言中只有 INT 整型数据类型，所以我们在设计词法分析器的



时候也只考虑这种类型，不用考虑浮点等其余基本数据类型。

#### 2.1.4 注释

因为 VSL 语言中是双斜杠//引入注释，所以我们需要连续识别两个’ /’ 字符才能够判断其为助手，否则则判断为除法符号。

#### 2.1.5 字符串

我们采用 VSL 中双引号 “ ” 表示字符串的方式进行字符串的识别，先识别左引号，然后遇到右引号完成识别。

### 2.2 语法分析器

我们使抽象语法树 AST 来对代码进行语法分析，使用 AST 使得程序的结构脉络变得十分清晰便于后续编译过程。而且我们设计了 AST 各个层次节点的缩进形式打印，可以在控制台清晰的看见语法书中各个层次节点之间的关系。所有解析语法的 AST 都继承于基类 ExprAST。我们这里还定义了三个用于报错的辅助函数，我们的语法解析器将用它们来处理解析过程中发生的错误。

#### 2.2.1 解析表达式

在解析表达式的时候我们用 AST 的子类 NumberExprAST 来解析数值，用 VariableExprAST 来解析变量，最后再用 BinaryExprAST 来处理表达式中所遇到的二元运算符，依次完成对表达式语法树的建立。在解析各种表达式的代码都已经完成后，我们再添加一个辅助函数，为它们梳理一个统一的入口。

#### 2.2.2 解析函数

在解析函数原型的时候，我们需要分为两部分来考虑，一个是解析函数声明，另外一个就是解析函数定义。其次，因为 VSL 语言中函数体是放在左右两个花括号 {} 中的，所以我们需要先识别左花括号 “{”，然后直到遇到右花括号 “}” 才算将函数体解析完成。

```

// definition ::= 'FUNC' prototype { expression }
static std::unique_ptr<FunctionAST> ParseDefinition() {
    getNextToken(); // eat FUNC.
    auto Proto = ParsePrototype();
    if (!Proto)
        return nullptr;

    if (CurTok != '{')
        return LogErrorF("Expected '{' in function body"); // eat { for VSL
    else
        getNextToken();

    if (auto E = ParseExpression())
        // return llvm::make_unique<FunctionAST>(std::move(Proto),
        // std::move(E));
    if (CurTok != '}')
        return LogErrorF("Expected '}' in function body"); // eat } for VSL
    else {
        getNextToken();
        return llvm::make_unique<FunctionAST>(std::move(Proto), std::move(E));
    }
    return nullptr;
}

```

其次在函数参数列表中的各个参数之间用逗号分隔，这一点也是我们需要实现的。

```

//参数列表中参数之间可用逗号分隔
getNextToken();
while (CurTok == tok_identifier || CurTok == 44) {
    if (CurTok == tok_identifier)
        ArgNames.push_back(IdentifierStr);
    CurTok = getNextToken();
}

```

所以具体步骤设计为先识别关键字 FUNC，然后解析函数名，以及括号中的形参列表，最后再是函数体。

## 2.3 IR 代码生成

IR 代码生成部分依赖于 llvm/IR 模块。对于我们定义的每一类抽象语法树，利用 IR 模块实现其 `codegen` 方法。例如以下三个例子分别为数表达式、变量表达式、函数调用表达式的 `codegen` 实现。

```

Value *NumberExprAST::codegen() {
    return ConstantFP::get(TheContext, APFloat(Val));
}

```

```

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 *CallExprAST::codegen() {
    // Look up the name in the global module table.
    Function *CalleeF = getFunction(Callee);
    if (!CalleeF)
        return LogErrorV("Unknown function referenced");

    // If argument mismatch error.
    if (CalleeF->arg_size() != Args.size())
        return LogErrorV("Incorrect # arguments passed");

    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");
}

```

当我们定义完毕所有 AST 的 `codegen` 方法后，当语法分析器解析到一个抽象语法树，我们可以调用其 `codegen` 方法生成相应的 IR 代码。

## 2.4 JIT 优化

JIT 的添加与优化如下图所示。首先我们设置 `FunctionPassManager`，并绑定到当前的 `llvm` 模块。然后添加几种常用的 `pass`，以优化生成的 IR，减少其冗余度，加快其运行速度。

```

static void InitializeModuleAndPassManager() {
    // Open a new module.
    TheModule = llvm::make_unique<Module>("my cool jit", TheContext);
    TheModule->setDataLayout(TheJIT->getTargetMachine().createDataLayout());

    // Create a new pass manager attached to it.
    TheFPM = llvm::make_unique<legacy::FunctionPassManager>(TheModule.get());

    // Promote allocas to registers.
    TheFPM->add(createPromoteMemoryToRegisterPass());
    // Do simple "peephole" optimizations and bit-twiddling optzns.
    TheFPM->add(createInstructionCombiningPass());
    // Reassociate expressions.
    TheFPM->add(createReassociatePass());
    // Eliminate Common SubExpressions.
    TheFPM->add(createGVNPass());
    // Simplify the control flow graph (deleting unreachable blocks, etc).
    TheFPM->add(createCFGSimplificationPass());

    TheFPM->doInitialization();
}

```

其次，我们在 FunctionAST 的 codegen 方法中在返回前调用 verifyFunction 方法，使 llvm 运行时调用我们指定的 Pass。

```

if (Value *RetVal = Body->codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    // Run the optimizer on the function.
    TheFPM->run(*TheFunction);

    return TheFunction;
}

```

## 2.5 流程控制

### 2.5.1 IF 条件语句

对于 IF 语句，我们的解析器实现如下。

```

/// ifexpr ::= 'IF' expression 'THEN' expression 'ELSE' expression FI
static std::unique_ptr<ExprAST> ParseIfExpr() {
    getNextToken(); // eat the IF.

    // condition.
    auto Cond = ParseExpression();
    if (!Cond)
        return nullptr;

    if (CurTok != tok_THEN)
        return LogError("expected THEN");
    getNextToken(); // eat the THEN

    if (CurTok != tok_RETURN)
        return LogError("expected RETURN");
    getNextToken(); // eat the RETURN

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

    if (CurTok != tok_ELSE)
        return LogError("expected ELSE");
    getNextToken();

    if (CurTok != tok_RETURN)
        return LogError("expected RETURN");
    getNextToken(); // eat the RETURN

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

    if (CurTok != tok_FI)
        return LogError("expected FI");
    getNextToken();

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

```

IF 语句的 AST 构造完成后，在其 codegen 方法构造相应的 llvm 语法块。

```

Value *IfExprAST::codegen() {
    Value *CondV = Cond->codegen();
    if (!CondV)
        return nullptr;

    // Convert condition to a bool by comparing non-equal to 0.0.
    CondV = Builder.CreateFCmpONE(
        CondV, ConstantFP::get(TheContext, APFloat(0.0)), "ifcond");

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

    // Create blocks for the then and else cases. Insert the 'then' block at the
    // end of the function.
    BasicBlock *ThenBB = BasicBlock::Create(TheContext, "then", TheFunction);
    BasicBlock *ElseBB = BasicBlock::Create(TheContext, "else");
    BasicBlock *MergeBB = BasicBlock::Create(TheContext, "ifcont");

    Builder.CreateCondBr(CondV, ThenBB, ElseBB);

    // Emit then value.
    Builder.SetInsertPoint(ThenBB);

    Value *ThenV = Then->codegen();
    if (!ThenV)
        return nullptr;

    Builder.CreateBr(MergeBB);
    // Codegen of 'Then' can change the current block, update ThenBB for the PHI.
    ThenBB = Builder.GetInsertBlock();

    // Emit else block.
    TheFunction->getBasicBlockList().push_back(ElseBB);
    Builder.SetInsertPoint(ElseBB);

    Value *ElseV = Else->codegen();
    if (!ElseV)
        return nullptr;

    Builder.CreateBr(MergeBB);
    // Codegen of 'Else' can change the current block, update ElseBB for the PHI.
    ElseBB = Builder.GetInsertBlock();

    // Emit merge block.
    TheFunction->getBasicBlockList().push_back(MergeBB);
    Builder.SetInsertPoint(MergeBB);
    PHINode *PN = Builder.CreatePHI(Type::getDoubleTy(TheContext), 2, "iftmp");

    PN->addIncoming(ThenV, ThenBB);
    PN->addIncoming(ElseV, ElseBB);
    return PN;
}

```

### 2.5.2 WHILE 循环语句

WHILE 语句的解析器如下。注意到这里的循环体实际上是 Expression，而非语法块。我们利用二元运算符连接表达式使得循环体可以是语法块。



```

static std::unique_ptr<ExprAST> ParseWhileExpr() {
    getNextToken(); // eat the WHILE

    auto End = ParseExpression();
    if (!End)
        return nullptr;
    if (CurTok != tok_D0)
        return LogError("expected 'D0' after WHILE end value");
    getNextToken(); // eat D0
    if (CurTok != '{')
        return LogError("expected '{' after D0");
    getNextToken(); //eat {

    auto Body = ParseExpression();
    if (!Body)
        return nullptr;
    if (CurTok != '}')
        return LogError("expected '}' after WHILE body");
    getNextToken();
    if (CurTok != tok_DONE)
        return LogError("expected DONE after WHILE body");
    getNextToken();

    return llvm::make_unique<WhileExprAST>(std::move(End), std::move(Body));
}

```

## 2.6 赋值语句

VAR 语句的解析器如下。

```

static std::unique_ptr<ExprAST> ParseVarExpr() {
    getNextToken(); // eat the var.

    std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;

    // At least one variable name is required.
    if (CurTok != tok_identifier)
        return LogError("expected identifier after var");

    while (true) {
        std::string Name = IdentifierStr;
        getNextToken(); // eat identifier.

        // Read the optional initializer.
        std::unique_ptr<ExprAST> Init = nullptr;
        if (CurTok == '=') {
            getNextToken(); // eat the '='.

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

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

        // End of var list, exit loop.
        if (CurTok != ',')
            break;
        getNextToken(); // eat the ','.

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

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

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

```

## 2.7 二进制代码生成与 PRINT 语句实现

我们在 mainloop 后加入以下代码，在解释器运行完毕后按 ctrl-d，即生成了针对当前机器的二进制代码。对于 PRINT 方法，我们采用 C++ 代码与二进制代码联合编译的方式实现。



```

// Initialize the target registry etc.
InitializeAllTargetInfos();
InitializeAllTargets();
InitializeAllTargetMCs();
InitializeAllAsmParsers();
InitializeAllAsmPrinters();

auto TargetTriple = sys::getDefaultTargetTriple();
TheModule->setTargetTriple(TargetTriple);

std::string Error;
auto Target = TargetRegistry::lookupTarget(TargetTriple, Error);

// Print an error and exit if we couldn't find the requested target.
// This generally occurs if we've forgotten to initialise the
// TargetRegistry or we have a bogus target triple.
if (!Target) {
    errs() << Error;
    return 1;
}

auto CPU = "generic";
auto Features = "";

TargetOptions opt;
auto RM = Optional<Reloc::Model>();
auto TheTargetMachine =
    Target->createTargetMachine(TargetTriple, CPU, Features, opt, RM);

TheModule->setDataLayout(TheTargetMachine->createDataLayout());

auto Filename = "output.o";
std::error_code EC;
raw_fd_ostream dest(Filename, EC, sys::fs::F_None);

if (EC) {
    errs() << "Could not open file: " << EC.message();
    return 1;
}

legacy::PassManager pass;
auto FileType = TargetMachine::CGFT_ObjectFile;

if (TheTargetMachine->addPassesToEmitFile(pass, dest, nullptr, FileType)) {
    errs() << "TheTargetMachine can't emit a file of this type";
    return 1;
}

pass.run(*TheModule);
dest.flush();

outs() << "Wrote " << Filename << "\n";

```

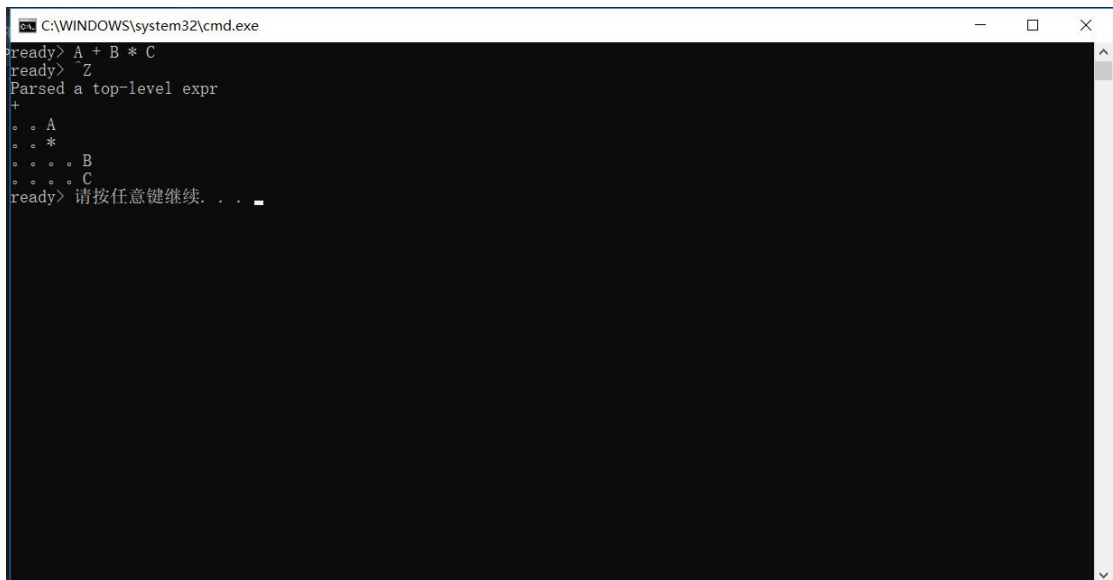
## 3 测试

### 3.1 子任务测试

#### 3.1.1 语法分析器测试

由于语法分析器集成了词法分析器，所以此处只列出语法分析器输出结果，我们将其输出结果利用 AST 的节点层次缩进来表示。

1. 输入语句：A + B \* C，解析输出：Parsed a top-level expr 然后将表达式 AST 的父子，兄弟节点关系用缩进形式表达输出。



```
C:\WINDOWS\system32\cmd.exe
ready> A + B * C
ready> ^Z
Parsed a top-level expr
+
. . . A
. . . *
. . . . B
. . . . . C
ready> 请按任意键继续. . .
```

2. 输入语句：FUNC main(a, b, c) {a + b \* c}解析输出：Parsed a function definition 然后将函数定义 AST 的父子，兄弟节点关系用缩进形式表达输出。

```
C:\WINDOWS\system32\cmd.exe
ready> FUNC main(a, b, c) {a + b * c}
ready> ^Z
Parsed a function definition.
FUNC
. . Prototype
. . . . main
. . . . a
. . . . b
. . . . c
. . Body
. . . . +
. . . . . a
. . . . . *
. . . . . . b
. . . . . . c
ready> 请按任意键继续. . .
```

### 3. 1. 2 IR 代码生成

IR 代码生成测试用例如下

```
Farleys-MacBook-Pro:Extend liyuan$ ./main
ready> FUNC foo(a b) { a*a + 2*a*b + b*b }
ready> Read function definition:
define double @foo(double %a, double %b) {
entry:
    %multmp = fmul double %a, %a
    %multmp6 = fmul double %a, 2.000000e+00
    %multmp8 = fmul double %multmp6, %b
    %addtmp = fadd double %multmp, %multmp8
    %multmp11 = fmul double %b, %b
    %addtmp12 = fadd double %multmp11, %addtmp
    ret double %addtmp12
}
```

### 3.1.3 IF 语句

```
Farleys-MacBook-Pro:Obj liyuan$ ./toy
ready> FUNC f(n)
{
    IF n
    THEN
        RETURN n * f(n-1)
    ELSE
        RETURN 1
    FI
}ready>
Read function definition:
define double @f(double %n) {
entry:
    %n1 = alloca double
    store double %n, double* %n1
    %n2 = load double, double* %n1
    %ifcond = fcmp one double %n2, 0.000000e+00
    br i1 %ifcond, label %then, label %else

then:                                     ; preds = %entry
    %n3 = load double, double* %n1
    %n4 = load double, double* %n1
    %subtmp = fsub double %n4, 1.000000e+00
    %calltmp = call double @f(double %subtmp)
    %multmp = fmul double %n3, %calltmp
    br label %ifcont

else:                                     ; preds = %entry
    br label %ifcont

ifcont:                                   ; preds = %else, %then
    %iftmp = phi double [ %multmp, %then ], [ 1.000000e+00, %else ]
    ret double %iftmp
}
```

### 3.1.4 WHILE 语句

```
ready> FUNC main()
{
    VAR i = 11
    WHILE i
    DO
    {
        printf(f(i)) :
        i = i - 1
    }
    DONE
} #ready>
Read function definition:
define double @main() {
entry:
    %i = alloca double
    store double 1.100000e+01, double* %i
    br label %loop

loop:                                     ; preds = %loop, %entry
    %i1 = load double, double* %i
    %calltmp = call double @f(double %i1)
    %calltmp2 = call double @printf(double %calltmp)
    %i3 = load double, double* %i
    %subtmp = fsub double %i3, 1.000000e+00
    store double %subtmp, double* %i
    %binop = call double @"binary:"(double %calltmp2, double %subtmp)
    %i4 = load double, double* %i
    %loopcond = fcmp one double %i4, 0.000000e+00
    br i1 %loopcond, label %loop, label %afterloop

afterloop:                               ; preds = %loop
    ret double 0.000000e+00
}
```

## 3.2 总体测试

### 3.2.1 测试代码

测试代码如下：

VSL:

FUNC f(n)

```
{
    IF n
```

```

    THEN
        RETURN n * f(n-1)
    ELSE
        RETURN 1
    FI
} #

```

```

C:
#include <iostream>

extern "C" {
double f(double);
}

int main() {
    int i = 0;
    while (11 - i) {
        printf("f( %d ) = %f\n", i, f(i));
        i = i + 1;
    }
}

```

我们为了实现 PRINT 功能，将 C-main 函数的目标文件与 VSL-f 函数的目标文件链接起来。链接编译命令如下：

```
clang++ main.cpp output.o -o main
```

### 3.2.2 测试结果

```
Farleys-MacBook-Pro:Obj liyuan$ ./main
f( 0 ) = 1.000000
f( 1 ) = 1.000000
f( 2 ) = 2.000000
f( 3 ) = 6.000000
f( 4 ) = 24.000000
f( 5 ) = 120.000000
f( 6 ) = 720.000000
f( 7 ) = 5040.000000
f( 8 ) = 40320.000000
f( 9 ) = 362880.000000
f( 10 ) = 3628800.000000
```

## 4 项目总结

### 4.1 成员的组成与分工

#### 4.1.1 成员组成

小组成员如下：

1. 组长：段梦梦，学号：2016302580001
2. 组员：李远，学号：2015302580279
3. 组员：何万伟，学号：2016302580125
4. 组员：王涛，学号：2016302580147
5. 组员：段小式，学号：2016302580006

#### 4.1.2 成员分工

成员分工如下：

1. 段梦梦：
  - ①负责成员的安排及分工
  - ②参与前期词法、语法分析、二进制代码生成以及 PRINT 语句的代码实现的代码撰写

- ③平时阅读笔记的撰写与整合、实验报告的撰写与整合
- ④参与最终代码的整体调试
- 2. 李远：
  - ①IR 代码生成的代码实现
  - ②控制语句、JIT 优化、赋值语句、二进制代码生成以及 PRINT 语句的代码撰写
  - ③参与最终代码的整体调试
  - ④实验报告的撰写
- 3. 何万伟：
  - ①主要负责完成词法分析器，语法分析器和 IR 代码的生成三个模块
  - ②参与最终项目的整体调试与改进
  - ③最终实验报告的撰写
- 4. 段小式：
  - ①阅读笔记第一二章节的撰写
  - ②完成了 if 控制语句以及 while 控制语句的部分
  - ③参与最终项目的整体调试与改进
- 5. 王涛：
  - ①阅读笔记第三四章节的撰写
  - ③参与语法分析、JIT 优化的代码实现
  - ③参与最终项目的整体调试与改进

## 4.2 李远

### 4.2.1 个人贡献

负责了最终实验报告的撰写、以及 LR 代码控制语句、JIT 优化、赋值语句、二进制代码生成以及 PRINT 语句的代码撰写。

### 4.2.2 挑战与解决思路

由于 llvm 依赖环境的复杂性，项目构建与运行十分具有挑战性。我参考了



官方提供的编译指令，并查阅网上资料，修改了编译指令以适应我使用的 Mac OS 系统，并使得项目成功运行起来。另一项挑战是 VSL 语法与 Kaleidoscope 语法实际上差别比较大，我花费很多时间查阅网上资料，参考 Kaleidoscope 语言实现了 VSL 语言的大量功能。

## 4.3 何万伟

### 4.3.1 个人贡献

1. 负责 Github 仓库的建立与代码的管理与提交。
2. 根据 LLVM 万花筒项目说明文档第一章到第三章完成了 VSL 语言的词法分析器、语法分析器和简单的 IR 代码的生成。
3. 负责最后实验报告的排版与整合。
4. 协调团队间的分工合作，定期组织组会的召开。

### 4.3.2 挑战与解决思路

一开始对于 LLVM 框架十分陌生不熟悉，光是安装 LLVM 框架就花了很长一段时间，最后仍然有 bug 编译失败，在查阅资料讲 VS 从社区版换为专业版之后才成功解决了该问题。

其次关于中文 LLVM 网站教程的不全面，所以我选择了阅读英文教程文档，在阅读理解方面花费了大量的精力与时间，对我的英文能力也是一个巨大的挑战。

因为我主要负责实验前一阶段的工作，所以在实现前三章功能的时候并未遇到太多难题。但仍然有许多的细节问题需要注意：

1. 在构造语法树的时候，难点在于解析二元表达式，因为二元表达式具有二义性，解析起来可能会得到不同的结果，所以我最后选择了运算符优先来解析二元表达式。而运算符优先级解析的基本思想就是通过拆解含有二元运算符的表达式来解决可能的二义性问题，这是一种利用二元运算符的优先级来引导递归调用走向的解析技术，于是我便制定一张优先

级表。

2. VSL 语言和 LLVM 官方提出的万花筒语言还是有很多不同的地方需要我们注意，首先是多个关键字的不同，VSL 中标识函数的关键字是“FUNC”，这些我们都需要在枚举类型中改过来。其次是 VSL 的函数体有花括号“{}”，所以我们在解析函数体时应该先识别“{”再识别“}”才能确保这是一个函数。再者就是 Kaleidoscope 可以利用 extern 调用标准函数库，但 VSL 并不可以。最后在 VSL 语言中的注释是用双斜杠“//”表示的，我们需要连续识别两个“/”才能判断这是注释，否则认定为是除号。

## 4.4 段梦梦

### 4.4.1 个人贡献

如成员分工可见，主要负责成员的工作安排、前期词法、语法分析、二进制代码生成以及 PRINT 语句的代码实现、平时阅读笔记五六七八章节的撰写、实验报告的撰写与整合以及最终代码的整体调试。

### 4.4.2 挑战与解决思路

首先，配置环境的时候，由于系统的不同，自己搜索了网上的大量教程，跟着教程一步步的做，终于成功的配置了实验所需的环境。对 llvm 也有了一定的概念。对万花筒项目的代码结构慢慢的清楚。只是在阅读万花筒项目的时候，阅读有一定的难度，虽然英文文献对了解该专业的一些前沿知识有所帮助，但因为英文水平的限制，平时并没有经常接触英文的教程，因此，阅读并且完全了解文中所讲思路有一些困难。由于前期的词法、语法分析较为简单，跟其他成员一同讨论，都成功的解决了。其中后半部分的代码具有一定的难度，因此将这些部分交给代码能力较强的成员李远进行开发，最终在其协助之下基本完成此部分的代码实现。

## 4.5 王涛

### 4.5.1 个人贡献

如成员分工可见，主要负责阅读笔记第三四章节的撰写、参与语法分析、JIT 优化的代码实现以及最终项目的整体调试与改进。

### 4.5.2 挑战与解决思路

撰写阅读笔记时只是简单的将文章中的思路写在阅读笔记上。没有结合 VSL 语言对代码进行深入的设计分析。但之后随着深入理解代码的逻辑，再次回看万花筒项目教程以及自己撰写的阅读笔记，能更好的了解实现一个语言的解释器的逻辑。

## 4.6 段小式

### 4.6.1 个人贡献

如成员分工可见，负责阅读笔记第一二章节的撰写、完成 if 控制语句以及 while 控制语句的部分、参与最终项目的整体调试与改进。

### 4.6.2 挑战与解决思路

在之前完全没有接触过 Github，不了解 Github 到底该怎么操作，所以每次修改与上传代码，都是下载 zip 文件，并且拜托成员上传。并且加上双学位以及后续课程的慢慢增加，之后也没有怎么去弄这一块。在阅读 LLVM 万花筒项目的指导部分，写阅读笔记时对于一些英文单词无法理解，因此都是全部翻译成中文，对照着英文慢慢理解其中的含义，之后就可以尝试着进行写阅读笔记。

## 5 项目感想与建议

本次项目在提交之前已基本实现实验所要求的功能，但当老师在课堂验收演示成果时，由于调试整合调试的原因未能将整个模块的 IR 代码成功生成（只能将子任务的 IR 代码单独生成），故本组成员感到万分歉意与愧疚。但现在该问题已经成功解决，能够将示例代码的 IR 代码全部成功生成，同时还实现了 PRINT 语句和二进制代码的生成，特在提交文件夹中附上最终整体的演示视频，辛苦老师验收了。

### 5.1 李远

#### 5.1.1 感想

通过课程学习，让我增长了对解释器以及语言本身的理解。尽管只有短短一学期的时间，却让我收获甚多，受益匪浅。本学期的这项实验不仅靠的是课堂上的时间，更是课外的细致钻研与问题解决。课堂外的学习是我们掌握知识，提高感悟必不可少的过程。

#### 5.1.2 建议

我认为本实验对于 LLVM 框架的依赖有些过重。我希望未来有机会实现更原生、更底层的解释器，对于语法分析和运行时等基础知识有更深掌握。

### 5.2 何万伟

#### 5.2.1 感想

通过本次实验使我受益匪浅，首先了解到了团队集体开发是一种什么样的模式，大家分工合作，共同完成一个大型项目。其次利用 Github 能够及时的提交和更新代码，同时还能有效的避免代码冲突，大大降低了团队开发的难度。

其次学会了使用 LLVM 框架开发一门语言的解释器，在 LLVM 的基础之上

使得解释器开发的难度也大大的降低了，链接现有的库使得代码量也大大减少，尤其使得 IR 代码生成这一块的实现变得十分简单。

同样在这门课上，使我温习了 C++ 的语法，也在基础代码的算法上有了显著的提高，使得自己的算法和编程能力有了明显的进步。

### 5.2.2 建议

关于对这门课程的建议，我觉得如果老师在每次课上能够根据章节安排给我们多讲解一点关于 LLVM 的理论知识就更好了。因为教程文档上虽然每个步骤都有，但是关于 LLVM 框架具体是如何实现这些步骤的，有些地方还是不是很清楚，只能按照官方文档给的用法使用，会有种一知半解的感觉。

其次使用 LLVM 框架来实现解释器，虽然在方法上比纯手工实现要简便许多，但对于一些用法也变得陌生了许多，因为很多在上学期《编译原理》课程中学到的理论知识都用不上，所以我想以后的实验课是不是可以减少 LLVM 框架的使用，让大家用《编译原理》的知识自己原生去实现一些功能。

## 5.3 段梦梦

### 5.3.1 感想

1、之前虽然接触过 Github, 但不会使用它的一些命令。本次课程的一个收获就是学习了 Github 的简单使用。通过使用 Github 进行代码的托管，下载文件、上传文件等均可以通过终端输入命令来进行，而不需要笨拙的下载 zip 文件或者创建文件夹一个文件一个文件的上传。

2、对一个完整的解释器的构造有了一定的认识。通过本次实验也对上学期所学《编译原理》有了更深一层的了解。在主力组员的帮助之下，也学习到了很多。

3、另外身为组长深知团队是很重要的，自己是小组长，在整个课程过程中，也不断地在思考如何带动每个人参与到其中来，毕竟大家的水平各不相同，需要合理的分工与合作。但是还是因为没什么具体经验，在最后验收的时候，并没

有及时的整合好代码，以至于当场的演示不是很理想。

### 5.3.2 建议

- 1、觉得老师应该分阶段验收，这样可以使同学们更清楚的知道整个流程，也可以敦促大家及时完成相关的任务。避免大家在最后验收的时候，才开始写代码。
- 2、课程刚开始对于 LLVM 不是很了解，甚至安装环境时都不理解为什么要安装这些东西，安装环境时也在网上找了许多教程，才慢慢安装成功。Windows 系统的成员安装时遇到了蛮多问题。希望老师之后可以在课程开始多讲授一些这方面的知识，避免在配置环境时浪费过多时间。

## 5.4 王涛

### 5.4.1 感想

本次解释器构造实验课程主要是利用 llvm 编写 vs1 语言的解释器。通过一个学期的学习与实践，对解释器以及编译的过程有了进一步的认识，同时也初步了解了如何去构造一个简单语法的语言的解释器。其中 llvm 需要用到 c++ 去写，由于对于 c++ 的掌握不是很好，导致写起来有些难度。我们主要参考的就是万花筒教程，按步骤一章一章的跟着讲解走，前面几章是中文，之后是英文，好处是循序渐进，但是随着进度推进，难度也加大了。在编写过程中，由于是要完成一个完整的程序，如果前面没写的很好，后面就会发生好多错误，然后再去反复修改，所以把握每一个阶段的质量很重要。

### 5.4.2 建议

- 1、希望有一个完整的介绍和引导。本课程的目的很明确就是编写一个解释器，可是对于第一次编写的新手来说对于如何去写没有一个大概的思路，甚至不知道该从何下手，所以希望在课程之初可以有一个引导性的介绍于讲解，什么是 LLVM，怎么利用 LLVM 去构造编译器，改造的效果如何，

虽然这些可以自己摸索，但是效率很低。

2、希望对于配置环境有一个教程。每一个实验性的课程，配置环境是必不可少的，但是往往会有很多出乎意料的情况发生，本学期前面一节课就花在了配置环境上面，为了统一实验标准，便利学生，把更多的时间花费在编写代码上会更合理。

3、分段式检查，总结式验收。本课程采取的是在学期末的时候对整个项目进行验收，由于小组管理的不成熟，很有可能不能把握好进度，导致拖拉和赶工，这样对于整体效果是有很大的影响，所以可以采取一个阶段一个阶段的验收，这样可以合理控制实验进度，同时按照平时成绩和期末评测来打分，这样更能够体现水平。

## 5.5 段小式

### 5.5.1 感想

经过了十几周的学习，在课程中我了解了万花筒语言这种过程式语言，以及如何使用它来定义函数、使用条件语句等，对 LLVM 框架有了初步的理解，从词法分析到语法分析、LLVM IR 代码生成、JIT 优化等的一步步实现，在解释器的构造有了更直接深刻的体会。基于上学期学习的编译原理，让我的理解从理论过渡到实践，使得解释器构造的脉络结构更清晰明了。在此之外，文档学习能力及资源利用能力也有了提升，从网络上获取到更多相关的知识并加以结合利用，也因如此在自我学习方面有了一定的改善。

### 5.5.2 建议

希望老师可以在课程授课的时候可以对 LLVM 和解释器的构造流程等理论知识能够稍微多介绍一些，尽管在学习中有 LLVM 教程文档可以用以借鉴学习，但这种形式的学习效率有些低下。此外希望可以对一些难点地方有所介绍，网络资源过于庞杂，使得寻找解决方法往往会花费过多时间且适用性低，如果能对这种难点问题有一些解决思路和方法可以更好的促进构造进展。

