

武汉大学计算机学院

本科生课程设计报告

SML 语言解释器总体设计与实现

专 业 名 称 : 软件工程

课 程 名 称 : 解释器构造实践

团 队 名 称 : 我解释的都队

指 导 教 师 一: 袁梦霆

团 队 成 员 一: 石亮禾 (2017302580033)

团 队 成 员 二: 宛铠涛 (2017302580072)

团 队 成 员 三: 周浩宇 (2017302580016)

二〇一九年十一月

郑 重 声 明

本团队呈交的设计报告，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。

团队成员签名：石亮禾、宛铠涛、周浩宇 日期：2019-11-29

摘 要

解释器构造实验的实验目的是使用所学习过的编译程序的构造原理、方法和技术实现一个 SML 语言的解释器。

实验设计主要遵循敏捷开发原则、接口隔离原则、里氏代换原则

实验内容主要包括：需求分析、整体架构、词法分析、语法分析、静态语义分析、中间代码生成、JIT 执行。

实验结论为成功实现了一个 SML 语言解释器。使用 C++ 的 LLVM 框架实现中间代码生成和解释执行的功能。实验取得成功

关键词：LLVM；SML；编译原理

目 录

1	实验目的和意义.....	7
1.1	实验目的.....	7
1.2	实验意义.....	7
2	实验设计.....	8
2.1	需求分析.....	8
2.1.1	文法要求.....	8
2.1.2	实验要求.....	8
2.2	开发工具.....	9
2.3	整体架构.....	9
2.3.1	传统编译器架构.....	9
2.3.2	本项目架构.....	9
2.4	词法分析.....	11
2.4.1	设置 Token 表.....	11
2.4.2	gettok()函数.....	12
2.5	语法与静态语义分析.....	15
2.5.1	逻辑视图.....	15
2.5.2	主要功能分析.....	16
2.5.3	AST 树的展示.....	26
2.6	中间代码生成.....	29
2.6.1	工具类和变量.....	29
2.6.2	RealExprAST 类.....	30
2.6.3	ExprsAST 类.....	30
2.6.4	VariableExprAST 类.....	30
2.6.5	BinaryExprAST 类.....	31
2.6.6	CallExprAST 类.....	32
2.6.7	IfExprAST 类.....	33
2.6.8	whileExprAST 类.....	34
2.6.9	LetExprAST 类.....	35
2.6.10	PrototypeAST 类.....	35
2.6.11	FunctionAST 类.....	36
2.6.12	GlobalVariable 类.....	37
2.7	JIT 执行.....	38
2.7.1	JIT 的理解.....	38
2.7.2	具体调用.....	38
3	测试.....	40
3.1	测试概要.....	40
3.1.1	测试目的.....	40
3.1.2	测试概述.....	40
3.2	测试流程.....	40
3.3	测试截图.....	43
3.3.1	全局变量的声明与基本使用.....	43

3.3.2	函数的声明与调用.....	44
3.3.3	if...else...then 条件控制流.....	44
3.3.4	while...do(...)循环控制流.....	44
3.3.5	表达式的计算（包括优先级）.....	45
3.3.6	Let...in...end 局部变量的定义与使用.....	45
3.3.7	函数的嵌套定义.....	46
3.3.8	'实现的函数重载.....	47
3.3.9	Print 函数.....	48
3.3.10	AST 树展示.....	48
4	不足与改进.....	50
5	实验结论与感想.....	51
6	附录.....	52
6.1	成员分工.....	52
6.1.1	前期准备.....	52
6.1.2	基本功能实现.....	52
6.1.3	扩展功能主要贡献.....	52
6.1.4	报告的撰写.....	52
6.1.5	其他.....	52

1 实验目的和意义

1.1 实验目的

本实验的实验目的是使用所学习过的编译程序的构造原理、方法和技术实现一个 SML 语言的解释器，能够允许根据用户输入的 SML 语言代码或者源程序，完成相应的词法分析、语法分析、语义分析，如果程序中出现错误，要求解释器能够展示出其中存在的语法错误和静态语义错误；如果程序没有错误，解释器会给出该程序执行后的结果。

1.2 实验意义

通过团队的分工与合作相结合，采用瀑布式开发，根据已有项目参考和学习 LLVM 框架架构，结合编译原理基础知识，构造一个 SML 语言的解释器，其中包括词法分析、语法分析、静态语义分析、中间代码生成和解释执行等基本过程。在这个过程中提高团队合作编码的能力；拓展了编译原理的理论知识并且延申到实践之中；同时也深入了解到了编译程序的构造原理和 workflows 以及技术实现。

2 实验设计

2.1 需求分析

2.1.1 文法要求

- I. 实现 SML 基本要求（完成包括解释执行在内的所有相关工作）
 - a) While ... do ... 语句。
 - b) If ... then ... else ... 语句。
 - c) 全局变量的定义与使用。
 - d) 基本的表达式（包括一元表达式和二元表达式）。
 - e) let ... in ... end 局部变量定义。
 - f) fun 函数的定义和调用（支持多个参数）。
 - g) 基本运算符的实现如: '+' '-' '*' '/' '%' '|' '<' '>'。
 - h) 运算符的优先级的定义与实现。
 - i) 实现 print 函数
- II. 实现部分扩展功能（完成包括解释执行在内的所有相关工作）
 - a) fun 函数定义中支持多个局部嵌套函数。
 - b) 实现 SML 语言中 '|' 函数重载功能（包括参数为常数）。

2.1.2 实验要求

- I. 词法分析
 - a) 根据输入的源程序，构建标识表。
 - b) 识别标识符与常量。
- II. 语法分析与静态语义分析
 - a) 根据词法构建符号表，判断是否存在语法错误与静态语义错误。
 - b) 构造 AST 语法分析树，并且打印到命令行。
- III. 代码生成
 - a) 扫描输入的 AST，生成等价的 IR 中间代码。
- IV. 解释器执行
 - a) 扫描输入 LLVM IR，调用 LLVM JIT 解释执行。
 - b) 修改 JIT，支持输入输出。

2.2 开发工具

开发环境：Windows10 家庭版

开发语言：C++

开发工具：Visual Studio 2017 Community

框架：LLVM、

团队资源管理器：github

2.3 整体架构

2.3.1 传统编译器架构

解释器的架构大致分为 5 个经典流程：词法分析 -> 语法分析 -> 语义分析 -> 中间代码优化 -> 目标代码生成。

传统的静态编译器（如大多数的 C 语言编译器）通常将编译工作分为三个阶段，分别由三个组件来完成：前端、优化器、后端。（如图 2.1 所示）

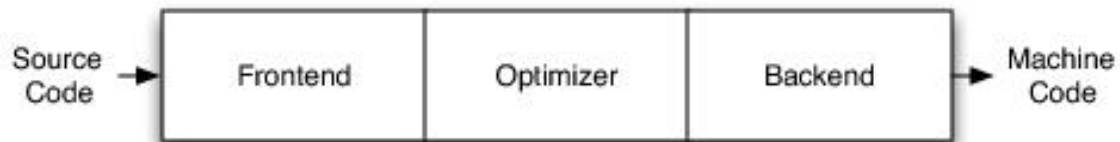


图 2.1 传统编译器架构

传统编译器采用的是管道与过滤风格的架构，采用了三段式的设计：前端负责解析源代码、检查错误，最后建立一个特定语言的抽象语法树（AST）来表示输入的代码，通常 AST 要转换为一种中间代码，然后传递给优化器，优化器针对中间代码进行一系列优化，后端再根据优化后的中间代码生成最后的目标指令。

2.3.2 本项目架构

a) 整体架构

本项目使用 LLVM 作为框架，LLVM 架构与传统架构有一定的相似度，但是它却针对传统架构进行了改进，有着独立的、完善的、严格约束的中间代码表示。这种中间代码就是 LLVM 的字节码，是 LLVM 抽象的精髓，前端生成这种中间代码，

后端自动进行各类优化分析，让用 LLVM 开发的编译器，都能用上最先见的后端优化技术。LLVM 架构模型如图 2.2 所示：

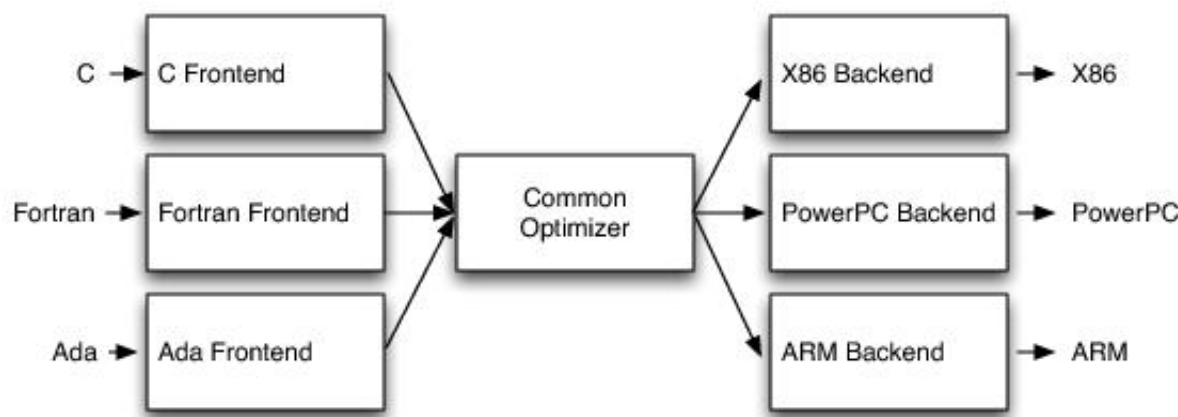


图 2.2 LLVM 架构

这种架构的优点在于当编译器决定支持多种语言或者多种目标设备的时候，如果编译器在优化器这里采用普通的代码表示时，前端可以使用任意的语言来进行编译，后端也可以使用任意的目标设备来汇编。

这种三段式设计的另一优点是编译器提供了一个非常宽泛的语法集，即对于开源编译器项目来说，这意味着会有更多的人参与进来，自然而然地就提升了项目的质量。

LLVM 另外一大特色就是自带 JIT，一个编译器要想实现 JIT，需要进行大量努力，即时翻译代码，还要兼顾效率和编译时间。所以 LLVM 将中间代码优化这个流程做到了极致。

本次解释器核心逻辑的主要工作都集中在前端（Frontend）部分。而本文也采用了如图 2.2 的架构进行前端的设计，接下来将从开发视图来介绍前端的设计。

b) 开发视图

解释器的开发视图如图 2.3 所示：

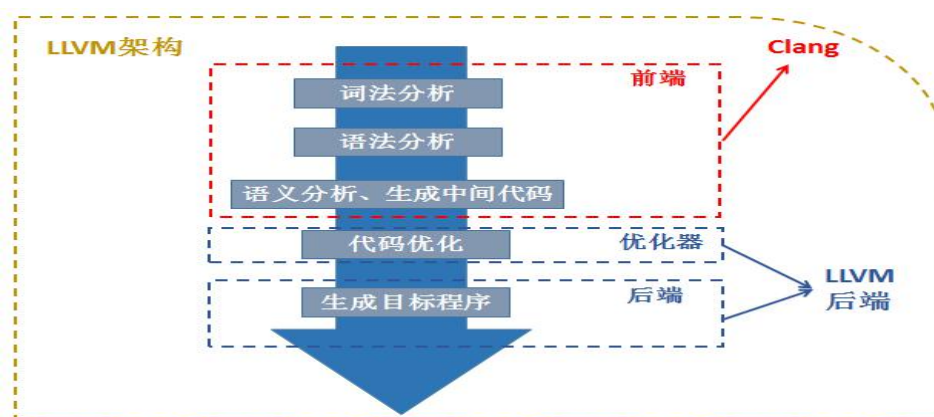


图 2.3 LLVM 开发视图

前端主要分为三层：

第一层为词法分析层，该层根据读取到的源代码内容为输入，对源代码进行词法分析，根据事先构建好的符号表进行判断是否出错，如果词法分析出错则会进行报错处理，如果未出错则输出词法分析结果：一个 Token 的类型、所包含内容。

第二层为语法分析和静态语义分析层，该层以第二层的输出为输入，然后根据 SML 特定的语法规则，对程序代码进行语法分析和静态语义分析，如果出现语法或者语义错误则会进行报错处理，如果未出错则输出语法分析生成的 AST 树，以供语义分析与代码生成层使用。本次项目第二层的缺点是将语法分析和静态语义分析冗杂在了一起，系统耦合度较高，可维护性较差。

第三层则以第二层生成的 AST 树为输入，对其进行语义分析，并生成相应的 IR 代码。

2.4 词法分析

词法分析程序是根据 SML 的词法规则所编写的。

2.4.1 设置 Token 表

首先设置一个 Token 表，在读取的时候用来识别每一个符号，在本次项目中，所需要识别的类型如下：

```
enum Token {
    tok_eof = -1, //结尾
    // commands
    tok_def = -2, //定义
```

```

// let definition
tok_let = -3,

// primary
tok_identifier = -4, //标识符
tok_real = -5,      // double
tok_int = -6,       // int
tok_string = -7,    // string
tok_char = -8,      // char
// control, 控制流
tok_if = -9,
tok_then = -10,
tok_else = -11,
tok_while = -12,
tok_do = -13,
tok_in = -16,
tok_end = -17,
// operators
tok_binary = -14,
tok_unary = -15,
// 变量声明符
tok_val = -18,
//分号
tok_semi = -19,
// print
tok_print = -20

};

```

2.4.2 gettok()函数

gettok()函数是完成词法分析最核心的函数，由于整个程序比较长，接下来对词法分析程序中的部分功能进行简要分析。

词法分析会去掉程序中所有的空白与逗号，而去进行后续的认识：

```

// Skip any whitespace和逗号.
while (isspace>LastChar) || LastChar == ',')
    // while (isspace>LastChar))
    LastChar = getchar();

```

词法分析会识别相应的标识符，在判断当前获取到的是字母时，会循环继续判断下一个字母，并且会把所有获得的字母拼接在一起，每次循环的时候都会判断这些字母是否拼接成为了一个关键字，如果是的话便根据 Token 表返回相应的值，如果直到循环结束都不是的话说明这不是关键字，则返回其它标识符：

```

if (isalpha>LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar()))
        IdentifierStr += LastChar;

    if (IdentifierStr == "fun")
        return tok_def;
    if (IdentifierStr == "let")
        return tok_let;
    if (IdentifierStr == "in")
        return tok_in;
    if (IdentifierStr == "if")
        return tok_if;
    if (IdentifierStr == "then")
        return tok_then;
    if (IdentifierStr == "else")
        return tok_else;
    if (IdentifierStr == "while")
        return tok_while;
    if (IdentifierStr == "do")
        return tok_do;
    if (IdentifierStr == "binary")
        return tok_binary;
    if (IdentifierStr == "end")
        return tok_end;
    if (IdentifierStr == "unary")
        return tok_unary;
    if (IdentifierStr == "val")
        return tok_val;
    if (IdentifierStr == "print")
        return tok_print;
    return tok_identifier; //其他标识符
}

```

当词法分析在判断到当前字符是数字的时候，便会进行循环继续获取后面的字符，如果也是数字，也会进行拼接，最后可以达到识别数字的效果，其中设置了变量 `type` 来判断获取的是否是小数：即当读取到数字并且进入循环时，如果当前字符是 ‘.’ 则会将 `type` 设置为 1，最后返回的时候通过 `type` 的值来判断返回 `tok_int` 还是 `tok_real`。同时，当 `type` 已经设置为 1 的时候再次获取到 ‘.’ 则会中断循环，进行报错处理。最后利用 c++ 内置函数将获取到的数字字符串转换为相应的类型：

```

if (isdigit>LastChar)) { // Number: [0-9]+
    int type = 0;
    std::string NumStr;
    do {
        if (LastChar == '.' && type == 1) {
            break;
        }
        if (LastChar == '.' && type == 0) {
            type = 1;
        }
        NumStr += LastChar;
        LastChar = getchar();
    } while (isdigit>LastChar) || LastChar == '.');

    NumVal = strtod(NumStr.c_str(), nullptr);
    if (type == 1) {
        return tok_real;
    }
    return tok_int;
}

```

同时也会识别中断标识符:

```

// Check for end of file. Don't eat the EOF.
if (LastChar == EOF)
    return tok_eof;

```

当上述条件都不满足时, 会直接返回当前字符:

```

int ThisChar = LastChar;
LastChar = getchar();
return ThisChar;

```

2.5 语法与静态语义分析

2.5.1 逻辑视图

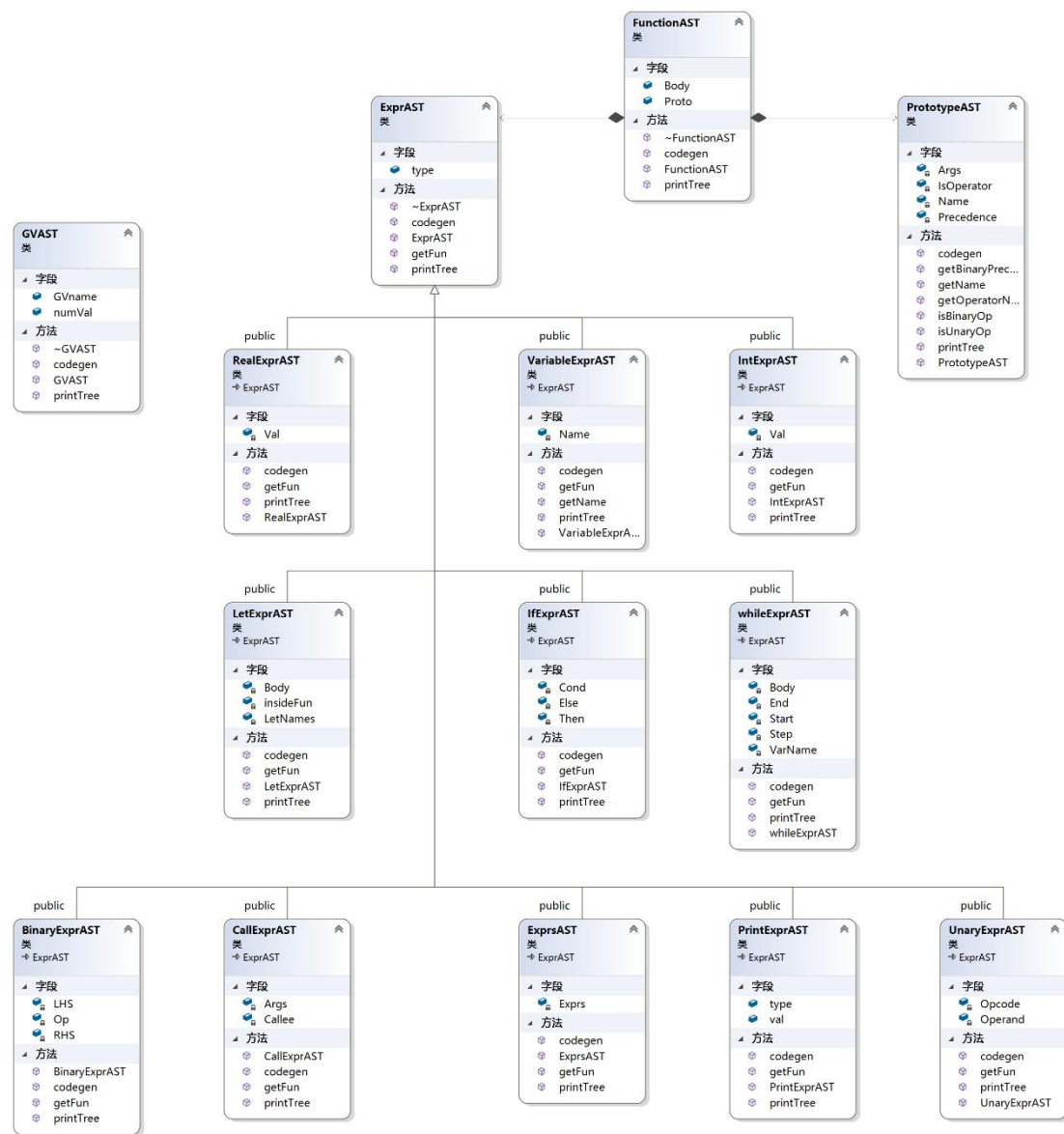


图 2.4 AST 类图

从图中可以看到，大部分 AST 树节点都继承自一个父类：ExprAST，且均有一个函数 `codeGen()`、一个函数 `getFun()` 与 `printTree()`。在每个节点的构造函数中，会根据文法进行相应的语法分析，判断是否出现语法错误。而语义分析与 IR 代码生成程序在做好相应的初始化准备之后，调用 AST 树根节点的 `codeGen()` 函数，此函数会递归调用，遍历整棵树，完成 IR 代码的生成。

2.5.2 主要功能分析

由于程序较长，故选择主要功能展示并且分析说明实现步骤。

a) 核心函数

为实现语法分析功能，设置了 `CurTok` 全局变量以及 `getNextToken()` 函数，其作用是获取在经过词法分析处理后的 `token`，然后可以进行后续的语法与语义分析。

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

当出现语法错误的时候，会调用相应的报错方法进行报错：

```
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;
}
```

b) 基本表达式

i. 设置运算符优先级

存储运算符的数据结构为 `map`，根据名字查询值

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

在设置初值的时候会对运算符的优先级进行设置，便可以实现运算符的优先级功能。

```
BinopPrecedence['='] = 2;
BinopPrecedence['<'] = 10;
BinopPrecedence['>'] = 10;
BinopPrecedence['+'] = 20;
BinopPrecedence['-'] = 20;
BinopPrecedence['%'] = 40; // highest.
BinopPrecedence['*'] = 40; // highest.
BinopPrecedence['/'] = 40; // highest
```

ii. 表达式的解析

在处理表达式的时候，根据表达式的特点，有一元表达式与二元表达式，所

以首先需要进行判断是什么类型的表达式，而两种表达式都可以分为三个部分，即：LHS、OP、RHS，如果是一元表达式的话，OP 与 RHS 将会是 nullptr，即可以由此区分。

首先先通过 ParseUnary()方法获取 LHS 的值，然后传入 ParseBinOpRHS()方法，来进一步获取 Op 与 RHS 的值，同时便可以判断是哪一种表达式

```
static std::unique_ptr<ExprAST> ParseExpression() {
    auto LHS = ParseUnary();
    if (!LHS)
        return nullptr;

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

首先它会判断 Op 的优先级(如果没有设置过优先级的符号则返回的值为-1)，说明这个表达式只是一元表达式，经过这个判断后便可以处理二元表达式，同理一个二元表达式有可能是多个嵌套的二元表达式组成的，只需要进行相应的操作，直到表达式结束便可以处理完这条表达式了，最后将获得的 LHS,Op,RHS 返回，用来后续计算：

```
static std::unique_ptr<ExprAST> ParseBinOpRHS(int ExprPrec,
                                              std::unique_ptr<ExprAST> LHS) {
    // If this is a binop, find its precedence.
    while (true) {
        //判断优先级
        int TokPrec = GetTokPrecedence();
        if (TokPrec < ExprPrec)
            return LHS;
        int BinOp = CurTok;
        getNextToken(); // eat binop

        // Parse the unary expression after the binary operator.
        auto RHS = ParseUnary();
        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));
}
}
```

iii. 表达式的类型

因为本项目采用的自上而下的文法，所以会根据当前获得的标识符来判断表达式的类型，然后执行相应表达式的处理函数：

```
static std::unique_ptr<ExprAST> ParsePrimary() {
    switch (CurTok) {
    default:
        return LogError("unknown token when expecting an expression");
    case tok_identifier:
        return ParseIdentifierExpr();
    case tok_real:
        return ParseRealExpr();
    case tok_int:
        return ParseIntExpr();
    case '(':
        return ParseParenExpr();
    case tok_if:
        return ParseIfExpr();
    case tok_let:
        return ParseLetExpr();
    case tok_while:
        return ParseWhile();
    case tok_print:
        return ParsePrintExpr();
    }
}
```

c) 函数的定义

i. 文法定义

definition ::= 'fun' prototype expression

ii. 代码说明

在确定模块为函数的定义时，首先会先吃入'fun'，然后分别进行 **prototype** 与 **expression** 的处理，代码如下，仅仅简单的分别调用相应的函数：

```
static std::unique_ptr<FunctionAST> ParseDefinition() {
    getNextToken(); // eat fun.
    auto Proto = ParsePrototype();
```

```

if (LineFun == 1) {
    return llvm::make_unique<FunctionAST>(std::move(P), std::move(B));
}
if (!Proto)
    return nullptr;

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

```

d) 函数的调用

i. 文法定义

Identifierexpr ::= identifier 变量

 ::= identifier '(' expression* ')' 函数调用

ii. 代码说明

该模块将函数和变量的调用同时实现了，二者的区别主要在“()”上，所以先判断是否存在’(‘，如果存在则证明是变量，可以直接返回该变量名：

```
return llvm::make_unique<VariableExprAST>(IdName);
```

如果不存在说明是调用函数，可以进行后续操作，函数定义的时候可以定义多个参数，所以调用的时候也相同，故在吃掉’(‘后需要一个 **vector** 来存储所有获得的值：

```
std::vector<std::unique_ptr<ExprAST>> Args;
```

相应的处理方法则是存入后判断是否还存在下一个参数，直到遇到’)’才表面调用结束：

```

if (CurTok != ')') {
    while (true) {
        if (auto Arg = ParseExpression())
            Args.push_back(std::move(Arg));
        else
            return nullptr;
        if (CurTok == ')')
            break;
    }
}

```

调用结束后吃掉 ‘)’ ‘返回函数名和它的参数：

```
return llvm::make_unique<CallExprAST>(IdName, std::move(Args));
```

e) If…else…then…语句

i. 文法定义

ifexpr ::= 'if' expression 'then' expression 'else' expression

ii. 代码说明

首先会吃掉‘if’，然后完成第一个表达式的分析

```
static std::unique_ptr<ExprAST> ParseIfExpr() {  
    getNextToken(); // eat the if.
```

调用 ParseExpression()函数即可以完成

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

完成判断后会根据文法进行语法检测，判断下一个字符是否是‘then’，如果不是的话便会调用报错方法进行报错，反之进行后续操作：

```
    if (CurTok != tok_then)  
        return LogError("expected then");  
    getNextToken(); // eat the then
```

吃掉‘then’之后会同上处理第二个表达式，如果‘then’之后的表达式为空则返回空：

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

对else的处理也同上，此处不再赘述：

```
    if (CurTok != tok_else)  
        return LogError("expected else");  
    getNextToken();  
    auto Else = ParseExpression();  
    if (!Else)  
        return nullptr;
```

最后确定语法未出错后实现构造函数，构造IfExprAST节点，完成了对if…

else…then语句的语法分析：

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

f) ‘|’实现函数重载（包括为常量的参数）

i. 增加的参数

为了实现'|'造成的函数重载功能，设置了以下参数，其中condNum表示参数是否为常数并且会进行保存；LineFun表示声明的函数是否带有'|'；P与B为全局的PrototypeAST与ExprAST，用来对'|'的多个函数进行保存。

```
double condNum;
bool LineFun = 0;
std::unique_ptr<PrototypeAST> P;
std::unique_ptr<ExprAST> B;
```

ii. 代码实现

实现该功能的主要函数如下：

```
static std::unique_ptr<PrototypeAST> ParsePrototype()
```

首先该函数会获取函数名，然后会对参数进行处理，如果在扫描的过程中得知这是一个带有|的函数，那么就会进行特殊的处理：跳入ParesFun()函数之中：

```
std::vector<std::string> Args;
while (1) {
    std::string Arg = ParsePattern();
    if (47 < (char)Arg[0] && 58 > (char)Arg[0]) {
        ParseFun();
        LineFun = 1;
        return llvm::make_unique<PrototypeAST>(FnName, Args, Kind != 0,
                                                BinaryPrecedence);
    }
    Args.push_back(Arg);
```

在ParseFun()函数中重新定义了个处理函数的文法，即不需要fun就可以定义函数的文法，在这个函数中会存入所有的参数，达到定义函数的效果：

```
static void ParseFun() {
    if (CurTok == ')') {
        getNextToken(); // eat ')'
    }
    if (CurTok == '=') {
        getNextToken();
    } else {
        printf("出错");
    }
    std::unique_ptr<ExprAST> thenBB = ParseExpression(); //得到then节点
    // getNextToken(); //eat ';'
    getNextToken(); // eat '|'
    getNextToken(); // eat FnName
```

```

if (CurTok == '(') {
    getNextToken(); // eat '('
}
std::string ArgName = IdentifierStr;
std::vector<std::string> Args;
Args.push_back(ArgName);
getNextToken(); // eat 参数名
if (CurTok == ')') {
    getNextToken(); // eat ')'
}

getNextToken();

```

最后将在ParseFun()中特殊定义的函数保存到全局变量P中即可：

```
P = llvm::make_unique<PrototypeAST>(FnName, Args, 0 != 0, 30);
```

g) While...do 语句

i.文法定义

whileExprAST ::= 'while' expression 'do' (expression ; expression ;)

ii.代码说明

语法检测步骤与 if-else-do 语句类似，在判断是 while 后会处理语句，并且再判断是否是 do，do 之后会有 '('，在获取到 '(' 之后会识别 Expr，因为在这里会出现多条表达式，所以在最开始定义了一个 vector 类型的 Exprs，用来存放所有读取到的表达式，这里实现的步骤是：在当前字符不是 ')' 的时候可以判断接下来还会有新的 Expr，所以会进行循环，直到遇到 ')' 表示 'do' 后面的表达式结束，具体代码如下：

```

static std::unique_ptr<ExprAST> ParseWhile() {

    std::vector<std::unique_ptr<ExprAST>> Exprs;
    getNextToken(); // eat the while.
    auto end = ParseExpression();
    if (!end)
        return nullptr;
    if (CurTok != tok_do) {
        return LogError("expected 'do' after while condition value");
    }
    getNextToken(); // eat do
    if (CurTok != '(') {
        return LogError("expected '(' after 'do'");
    }
    getNextToken(); // eat '('

```

```

auto Expr = ParseExpression();
if (!Expr)
    return nullptr;
if (CurTok == tok_semi) {
    getNextToken();
}
Exprs.push_back(std::move(Expr));
while (CurTok != ')') {
    Expr = ParseExpression();
    if (!Expr)
        return nullptr;
    if (CurTok == tok_semi) {
        getNextToken(); // eat ';'
    }
    Exprs.push_back(std::move(Expr));
}
getNextToken(); // eat ')'

//制造初始条件 (start)
auto start = llvm::make_unique<RealExprAST>(0);
//制造步长 (step)
auto step = llvm::make_unique<RealExprAST>(1);

auto body = llvm::make_unique<ExprsAST>(std::move(Exprs));

return std::make_unique<whileExprAST>("slh", std::move(start), std::move(end),
                                     std::move(step), std::move(body));
}

```

Strat、step、body 等变量将用来实现循环功能，此功能将在后续进行解说

h) let-in-end 局部变量定义以及嵌套函数定义

i. 文法定义

$$\text{letexpr} ::= \text{'let' identifier ('=' expression)? (' identifier ('=' expression)?)* 'in' expression 'end'}$$

ii. Let-in-end 局部变量定义代码说明

由于语法分析部分代码于上诉思路类似，在此不再赘述，以下为部分关键代码：

实现该功能的函数名：

```
static std::unique_ptr<ExprAST> ParseLetExpr() {...}
```

在 `let` 之后是定义局部变量的语句,所以会有着多条局部变量;在 `in` 之后会有着关于定义好后的局部变量的相关计算,也会存在多条表达式,故与上述处理方式相同,我们提前设置 `vector` 来存放变量和表达式,在后面的处理中也会根据读取到的符号来判断是否进行循环:

```
std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> LetNames;
std::vector<std::unique_ptr<ExprAST>> Exprs;
```

其中 `LetNames` 第一个参数用来存放变量名,第二个参数用来存放表达式(这里的表达式是给局部变量赋值的表达式),所以在处理 `let` 里面的符号时,会吃掉变量类型 (`val`) 后获取变量名:

```
std::string Name = IdentifierStr;
```

然后再获取等号后面的表达式,这里用中间变量 `Init` 存放:

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

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

最后将二者一同存入 `LetNames`:

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

在判断 `let` 结束的方法是遇到了 `'in'`, 便结束循环:

```
if (CurTok == tok_in)
    break;
```

`in` 后面的语句与上述处理多条表达式的方法相同,此处不再解释。

最后在成功检测后,构建相应的 `AST` 节点,以便后续进行中间代码生成:

```
auto Body = llvm::make_unique<ExprsAST>(std::move(Exprs));
return llvm::make_unique<LetExprAST>(std::move(LetNames), std::move(Body),
                                     std::move(insideFuntion), 11);
```

iii. 局部嵌套函数定义代码声明

● 设计思路

为了实现局部嵌套函数的定义功能,在上面功能的基础上,原本 `let` 之后只能定义 `val` 类型的变量,所以我们增加了条件的判断,使之可以识别 `fun`, 在得

到 `fun` 之后进行 `fun` 的语法分析即可。

- 实现代码

整体代码如下:

```
if (CurTok == tok_val) {...}

else if (CurTok == tok_def) {
    //如果读到的是 fun, 则转换函数定义

    auto insideFuntion = ParseDefinition();

    .....

}
```

第一个判断语句如果为真说明该表达式是局部变量定义, 则进入局部变量处理代码; 如果假则判断是否为 `fun`, 说明进入了嵌套函数的定义中, 而嵌套函数中的代码与局部变量代码唯一不同就是多了处理函数定义的方法 `ParseDefinition()`, 其余部分都完全相同, 最后返回的时候将 `insideFuntion` 添加进构造函数即可实现:

```
return llvm::make_unique<LetExprAST>(std::move(LetNames), std::move(Body),
                                     std::move(insideFuntion), 12);
```

i) `print` 函数的实现

i. 主要形式

`Print (...) | print “ ... ”`

ii. 代码部分

这里只是做语法分析, 在语法识别正确后, 会用 `name` 来保存所需要输出的内容, 最后构造一个 `PrintExprAST` 节点后续利用 IR 来完成 `print` 函数的主要功能:

```
static std::unique_ptr<ExprAST> ParsePrintExpr() {
    getNextToken(); // eat 'print'
    std::string name = "";
    if (CurTok == '(') {
        getNextToken(); // eat '('
        name = IdentifierStr;
        getNextToken(); // eat 变量
        if (CurTok != ')') {
            return LogError("expected ')'");
        }
    }
}
```

```

    }
    getNextToken(); // eat '('
    return llvm::make_unique<PrintExprAST>(name, 1);
} else if (CurTok == '"') {
    getNextToken(); // eat '"'
    name = IdentifierStr;
    getNextToken(); // eat 字符串
    if (CurTok != '"') {
        return LogError("expected \"");
    }
    getNextToken(); // eat ""
    return llvm::make_unique<PrintExprAST>(name, 0);
}
}

```

2.5.3 AST 树的展示

a) 原理方法

在语法分析之后，我们已经正确的获得了程序代码等价转换成的 AST 树以及树的根节点。与中间代码生成的思路类似，我们为每一个 AST 节点类设置 `printTree()` 函数，基类 `ExprAST` 中的 `printTree()` 函数为虚函数，所有继承自它的子类针对各自的组成特点重写该函数，负责打印自身的信息，并调用子节点的 `printTree()` 函数，这样一来，我们只需要调用根节点的 `printTree()` 函数，就可以递归遍历整棵树并打印出所有的信息，且以树的形式展示。

`printTree()` 函数的参数为一个 `int` 型的 `level`，即该节点所在层数，具体在命令行打印出来的树中的体现就是前缀空格长度。

b) 部分 AST 节点 `printTree()` 方法的详细介绍

● 以 `RealExprAST`（浮点型数）为代表的 不含子节点类节点

```

void RealExprAST::printTree(int level) {
    std::string preStr = ""; // 打印前缀

    for (int i = 0; i < level; i++) {
        preStr += "    ";
    }

    std::cout << preStr << "RealExprAST: NumVal: " << Val << "\n";
}

```

```
}
```

对于此类节点，只需要输出本身的信息即可。为了形成层次感，我们需要将同一层的节点在命令台显示时放在同一列，这样就需要我们给同一层的节点加上相同长度的前缀。实现的方式就是，利用 for 循环，根据 level 参数加上指定长度的空格。

在输出了指定长度的前缀后，再输出本身的信息，对于 RealExprAST 类节点来说就是类型名称、数值大小。（可以加上所在行数，但由于我们采用命令行输入，未设置此功能。）

● 以 IfExprAST、CallExprAST 为代表的含有子节点的节点

① IfExprAST: （条件语句）

```
void IfExprAST::printTree(int level) {  
    std::string preStr = ""; // 打印前缀  
    for (int i = 0; i < level; i++) {  
        preStr += "    ";  
    }  
    std::cout << preStr << "IfElseThenAST:\n";  
    //依次输出参数的树  
    std::cout << preStr << "    "  
        << "判断条件节点: \n";  
    Cond->printTree(level + 2);  
    std::cout << preStr << "    "  
        << "条件为真节点: \n";  
    Then->printTree(level + 2);  
    std::cout << preStr << "    "  
        << "条件为假节点: \n";  
    Else->printTree(level + 2);  
}
```

可以看到，同之前介绍的 RealExprAST 类相同的是，我们根据所在层次输出

了指定长度的前缀，并输出了本身的信息：IfElseThenAST。

随后，依次调用条件节点、条件为真节点、条件为假节点的 `printTree()` 函数，输出子节点的信息。值得注意的是，为了更好地展示节点的组成含义，我为每个节点添加了说明信息，即在调用判断节点的 `printTree()` 函数之前，先输出“判断条件节点：\n”，所以后续条件节点的参数为 `level + 2` 而不是 `level + 1`。

② CallExprAST: （函数调用）

```
void CallExprAST::printTree(int level) {  
    std::string preStr = ""; // 打印前缀  
    for (int i = 0; i < level; i++) {  
        preStr += "    ";  
    }  
    std::cout << preStr << "CallExprAST: CallName: " << Callee << "\n";  
    //依次输出参数的树  
    for (int i = 0; i < Args.size(); i++) {  
        std::cout << preStr << "    "  
            << "单条语句节点: ";  
        Args[i]->printTree(0);  
    }  
}
```

同样的，先打印前缀，然后输出本节点信息：“CallExprAST: CallName: ”+调用的函数名称。

随后，输出“参数节点：\n”，节点并依次调用输入参数的 `printTree()` 函数，从而输出参数节点的信息。

最终我们可以得到一颗层次分明的 AST 树（横置的）。有关树的形状将会在测试部分展示。

2.6 中间代码生成

以上的程序代码都在完成词法、语法以及静态语义检查的步骤，这个时候我们已经获得了由词法语法分析器构造的 AST 语法树。下一步的目的是将 AST 树转换成 IR 中间代码。

中间代码的生成主要依靠 `codegen()` 函数实现，在基类 `ExprAST` 中，定义了抽象方法 `codegen()`，在其他所有 AST 节点类中，都重载此函数，实现相对应的代码生成功能。于是，当我们拿到一棵树的根节点时，就可以调用该根节点的 `codegen()` 函数，并递归调用其子节点的 `codegen()` 函数，最终返回整棵树的 IR 代码，并提供给 JIT 程序，为程序执行做准备。下面将详细介绍 IR 生成过程中用到的有关变量以及每一个 AST 节点的 `codegen()` 函数实现细节。

2.6.1 工具类和变量

`static LLVMContext TheContext`：一个 LLVM 给我们的工具类，在拥有一个实例后就可以使用它得到 LLVM 传递给我们的众多 API

`static IRBuilder<> Builder(TheContext)`：一个用于生成 IR 指令的工具类，里面封装了很多创建 IR 代码的方法，我们只需要使用该类的方法便可以简捷地生成对应的 IR 代码。

`static std::unique_ptr<Module> TheModule`：在 LLVM 中，是一个装载代码和变量、函数的特殊容器，我们往往是将很多基本块装进这个 module，最终将 module 传给 JIT 执行。

`static std::map<std::string, AllocaInst*> NamedValues`：用来保存局部变量的变量名和存储地址的 map 对象，每次新建一个变量就会将变量地址存进来，每次读取变量也是从中读取变量的地址。

`static std::map<std::string, GlobalVariable*> GlobalValues`：用来存储全局变量的变量名和存储地址的 map 对象，作用效果和原理与 `NamedValues` 类似。

`static std::map<std::string, std::unique_ptr<PrototypeAST>> FunctionProtos`：用来存储函数名和函数声明的 map 对象，作用效果和原理与 `NamedValues` 类似。

`static std::unique_ptr<KaleidoscopeJIT> TheJIT`: 个人理解为一个相对封闭的可以自己配置运行环境的运行驱动, 在 JIT 配置过程中我们可以自适应的加入特定的优化 Pass, 可以运行 module 容器并返回结果。(有关这部分的详细细节会在下一章讲述)

2.6.2 RealExprAST 类

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

该节点表示的是代码中的数值常量, 在 IR 中, 数字常量由 ConstantFP 类表示, 将数字值保存在 APFloat 内部, 所以我们可以直接调用 ConstantFP 获取指定数字(任意精度)并返回。

2.6.3 ExprsAST 类

```
Value *ExprsAST::codegen() {  
    Value *val = nullptr;  
    for (unsigned i = 0, e = Exprs.size(); i != e; ++i) {  
        val = Exprs[i]->codegen();  
    }  
    return val;  
}
```

该类的做法是, 将 vector<ExprAST> 向量中的所有节点遍历, 并调用其 codegen() 函数。

2.6.4 VariableExprAST 类

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

```

    if (!V) {

        return LogErrorV("Unknown variable name");

    }

}

// Load the value.

return Builder.CreateLoad(V, Name.c_str());

}

```

这是存储变量的节点，所有我们要做的就是将变量对应的值读取出来。我们的原则是首先查看该变量是否是局部变量，如果在局部变量表中找到，则将其地址调出；若局部变量表中没有，则去全局变量表中查找；若两者皆没有找到，则说明该变量并没有被声明过，返回报错值。

最后一句 `CreateLoad` 函数是关键，它将创建 IR 的 `load` 指令，作用是将变量值从地址中取出来。

2.6.5 BinaryExprAST 类

对于该类有多种情况讨论：

a) 运算符为 ‘=’

```

VariableExprAST *LHSE = static_cast<VariableExprAST *>(LHS.get());

if (!LHSE)

    return LogErrorV("destination of '=' must be a variable");

```

首先是将左表达式转换成变量节点（因为赋值运算的目标值只能是个已声明的变量），如果发现无法转换，则说明左侧不是一个变量，系统报错。

```

Value *Val = RHS->codegen();

if (!Val)

    return nullptr;

```

调用右表达式节点的 `codegen()` 函数，得到其 `Value` 值，如果发现其值为空，则说明右表达式在转换时出现问题，报错。

```

Value *Variable = NamedValues[LHSE->getName()];

if (!Variable) {

    Variable = GlobalValues[LHSE->getName()];

```

```

    if (!Variable)

        return LogErrorV("Unknown variable name");
}

Builder.CreateStore(Val, Variable);

return Val;

```

接下来就是从变量表中读取变量地址并 load 变量值，最终返回右表达式的值。

运算符为 ‘+’ 、 ‘-’ 、 ‘*’ 、 ‘/’ 、 ‘%’ 、 ‘<’ 、 ‘>’ 、 ‘~’ ；

```

Value *L = LHS->codegen();
Value *R = RHS->codegen();

```

先依次调用左右表达式的 codegen()函数，得到其 IR 值。

```

return Builder.CreateFAdd(L, R, "addtmp");

```

再对运算符的内容进行判断，以 ‘+’ 为例，将会返回相加指令。这里值得注意的是，我们使用了 Builder 类的 CreateFAdd 方法，Builder 类中还有很多类似于此的方法，可以用来很方便的创建运算指令。其他运算符皆调用 Builder 类的类似方式创建，这里不再赘述。

b) 运算符为其他

如果运算符不是语言自带的，那么可能是用户自定义的（不过暂时没发现 SML 语言有自定义运算符的语法）。如果是这种情况，则会查找函数表，并将此符号作为一个函数来调用。

2.6.6 CallExprAST 类

```

Function *CalleeF = getFunction(Callee);

if (!CalleeF)

    return LogErrorV("Unknown function referenced");

```

首先使用 getFunction()函数获得该函数的 Function 节点对象。getFunction()函数的具体实现逻辑为：首先使用 TheModule->getFunction(name)从 module 中找寻该函数的定义，如果该函数没有具体实现（只是声明了），则需要去 FunctionProtos 中查找该函数的 prototypeAST 节点（因为没有实现只声明的函数

不会加入到 **module** 中，但是会在 **FunctionProtos** 中有记录。但是这部分的功能对于 **SML** 来说是多余的，因为 **SML** 语法好像没有用来声明函数的 **extern** 语句)

```
if (CalleeF->arg_size() != Args.size())  
  
    return LogErrorV("Incorrect # arguments passed");
```

当搜索到对应的函数节点后，接下来进行参数的匹配验证，如果参数的个数和函数原型的参数个数不匹配，则报错。

```
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");
```

如果参数匹配，则将用户输入的参数压入向量 **ArgsV** 中，并调用 **Builder** 的 **CreatCall** 方法创建函数调用的 **IR** 指令并返回。这里又一次用到了 **Builder** 类的方法。

2.6.7 IfExprAST 类

该类是记载条件判断逻辑的 **AST** 节点。这个类的 **codegen()** 方法的诀窍在于逻辑判断和不同块的跳转、语句的插入点。

```
CondV = Builder.CreateFCmpONE(CondV, ConstantFP::get(TheContext, APFloat(0.0)),  
"ifcond");
```

首先调用该节点的条件子节点的 **codegen** 函数，得到该条件值，并将其与数字 **0** 相比较得到条件是否为真。此时的 **CondV** 为 **IR** 的比较指令。

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

接下来创建三个基本块。在 **LLVM** 中，**module** 往往由很多个块组成，我们可以自由的在块之间跳转。注意，在这里我们将 **ThenBB** 的位置置于 **TheFunction**

后，这是符合事实逻辑的。

```
Builder.SetInsertPoint(ThenBB);  
  
Value *ThenV = Then->codegen();
```

接下来将 IR 代码插入点设置为 ThenBB 块（初始为空）的后面并写入 Then 节点的代码（本来理应是顺序写入，但是 CondV 的 codegen 函数很可能将插入点改变到了其他位置，所以我们得重置一下）

```
Builder.CreateBr(MergeBB);
```

这一步很关键，由于处理完 Then 部分的代码后应该跳过 Else 部分，所以我们创建了一个无条件跳转指令。Else 块的 IR 生成与 Then 类似，所以不再赘述。

```
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;
```

最后的 MergeBB 块和前述两个块一样设置插入点。同时在块内创建了一个 PHI 指令，这个指令的作用是将 ThenBB 和 ElseBB 的程序支路汇合。并根据数据来源决定返回值。最后三行是建立块与对应 Value 间的联系。

2.6.8 whileExprAST 类

该类的 codegen 函数思想与 if 跳转有相通之处，都是对条件进行判断并跳转至其他块。

```
BasicBlock *LoopBB = BasicBlock::Create(TheContext, "loop", TheFunction);  
  
BasicBlock *AfterBB =  
  
    BasicBlock::Create(TheContext, "afterloop", TheFunction);
```

一共有两个块结构，其中 loopBB 是中间主体块，AfterBB 是 while 之后的语句块。由于该类的 codegen 与 if 类似，所以相关的块不再赘述。

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

与之前不同的是，这里有一个结束条件 `EndCond`，我们需要调用 `Builder` 的比较函数来判断条件是否为真，并返回结果。

```
Builder.CreateCondBr(EndCond, LoopBB, AfterBB);
```

创建跳 IR 跳转指令，转根据条件的真假来决定是否跳出循环。

2.6.9 LetExprAST 类

该类的 `codegen` 函数与之前不同的地方在于需要创建局部变量，也是该类的特色所在。我将重点介绍。

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

```
AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, LetName);
```

```
Builder.CreateStore(InitVal, Alloca);
```

创建一块内存用来存储局部变量，并将该局部变量的初值存进变量的地址中。

```
std::vector<AllocaInst *> OldBindings;
```

```
OldBindings.push_back(NamedValues[LetName]);
```

```
NamedValues[LetName] = Alloca;
```

这里有一个很关键的操作，是将 `NameValues` 中的变量值暂存进 `OldBindings` 中。作用是：当 `let` 中出现了该函数原有的变量，则将变量表中的地址暂时保存至一个数据结构中，用 `let` 新定义的变量覆盖 `NameValues` 表中的地址值。待 `let` 语句结束后，重新将地址值从 `OldBindings` 中压入 `NameValues`。从中我们可以看出设定的语法规则：`let` 中新定义的变量和函数原有的变量重名时，会使用 `let` 中定义的变量。另外，最终函数返回的是 `in` 中的主体部分的 `Value` 值。

2.6.10 PrototypeAST 类

该类的作用是存储函数的声明（包括函数名和参数列表），所以 `codegen` 函数的作用也是围绕这两部分展开。

```
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());

```

首先创建一个 `double` 类型的向量作为参数向量（暂时所有的参数类型都为 `double` 型，后续还想添加更多功能，比如可以指定参数类型）。使用 `Function` 类创建一个函数声明，参数包括函数名、函数类型定义、当前模块指针、链接性。

```

unsigned Idx = 0;

for (auto &Arg : F->args())

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

```

接下来为函数的参数依次设置参数名，最后将函数指针 `F` 作为返回值返回。

2.6.11 FunctionAST 类

该类是函数的定义类，包含了两个成员（声明和主体定义）。`Codegen` 函数也同样是围绕这两部分展开。

```

auto &P = *Proto;

FunctionProtos[Proto->getName()] = std::move(Proto);

Function *TheFunction = getFunction(P.getName());

```

首先将函数的声明加入到 `FunctionProtos` 表中，这样可以使得以后调用函数时能够通过查表得方式查找到该函数。

```

BasicBlock *BB = BasicBlock::Create(TheContext, "entry", TheFunction);

Builder.SetInsertPoint(BB);

```

由于一开始只有声明，是没有基本块的，所以我们加入基本块。并将 `IR` 代码生成位置置于该块后。

```

NamedValues.clear();

for (auto &Arg : TheFunction->args()) {

    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName());

    Builder.CreateStore(&Arg, Alloca);

    NamedValues[Arg.getName()] = Alloca;

}

```

这几步对参数初始化是十分重要的，先是将 `NamedValues` 表清空，目的是将之前的函数遗留的局部变量清除，避免对当前函数的变量产生同名干扰。然后对参数依次遍历，并为每一个参数分配地址空间、加入 `NamedValues` 结构中。

```
if (Value *RetVal = Body->codegen()) {  
    Builder.CreateRet(RetVal);  
    verifyFunction(*TheFunction);  
    return TheFunction;  
}
```

接下来是对函数主体进行生成，调用函数主体节点的 `codegen` 函数，得到其返回值，如果其不为空（主体节点生成 IR 代码成功），则将其作为本函数的返回值。程序的最后，是让该函数得到正确性验证，调用 LLVM 提供的库函数 `verifyFunction()` 即可。如果这几步都没有出错，最终返回该函数 `TheFunction`。

2.6.12 GlobalVariable 类

该类用来声明全局变量，关于全局变量的定义和初始化是该类的 `codegen()` 函数的重点和难点。

```
GlobalVariable *global_val = new GlobalVariable(  
    *TheModule, Type::getDoubleTy(TheContext), false,  
    GlobalValue::PrivateLinkage,  
    ConstantFP::get(Builder.getDoubleTy(), numVal), GVname);
```

首先创建一个新的 `GlobalVariable` 对象，参数的意义如下：

- a. `*TheModule`: 当前模块的指针
- b. `Type::getDoubleTy(TheContext)`: 变量的类型，需要使用 `getDoubleTy` 方法获得（这里暂时只支持 `double` 型）
- c. `False`: 是否为不变量，由于这里是声明一个全局变量，所以自然为 `false`。
- d. `GlobalValue::PrivateLinkage`: 外部链接性
- e. `ConstantFP::get(Builder.getDoubleTy(), numVal)`: 变量的初始链接（定义初值）
- f. `GVname`: 变量名字

```
global_val->print(errs());
```

```
GlobalValues[GVname] = global_val;
```

在变量创建完毕后，需要将其名字地址对加入 `GlobalValues` 中，以便后续使用时可以通过查 `GlobalValues` 获取相应地址。

2.7 JIT 执行

2.7.1 JIT 的理解

JIT 编译器，英文写作 `Just-In-Time Compiler`，中文意思是即时编译器。

JIT 是一种提高程序运行效率的方法。通常，程序有两种运行方式：静态编译与动态解释。静态编译的程序在执行前全部被翻译为机器码，而动态解释执行的则是一句一句边运行边翻译。

LLVM 框架提供了对 IR 代码的 JIT 支持，我们项目中使用了 LLVM 官方教程中所用的 `KaleidoscopeJIT` 类。该类的使用十分简单，只有传入适当的函数名，`KaleidoscopeJIT` 就可以为我们执行相应的函数并且返回执行结果。

2.7.2 具体调用

首先我们会添加一下函数进行初始化，我们准备环境以为当前本机目标创建代码，然后声明并初始化 JIT。这是通过调用一些 `InitializeNativeTarget` 函数，并且添加一个全局变量 `TheJIT` 并在 `main` 以下位置将其初始化来完成的：

```
InitializeNativeTarget();
InitializeNativeTargetAsmPrinter();
InitializeNativeTargetAsmParser();

TheJIT = llvm::make_unique<KaleidoscopeJIT>();
InitializeModule();
```

然后为 JIT 设置布局：

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

以通过 JIT 更改的解析顶级表达式的代码如下：

```
static void HandleTopLevelExpression() {
```

```

if (auto FnAST = ParseTopLevelExpr()) {
    if (FnAST->codegen()) {
        auto H = TheJIT->addModule(std::move(TheModule));
        InitializeModule();
        auto ExprSymbol = TheJIT->findSymbol("__anon_expr");
        assert(ExprSymbol && "Function not found");
        double (*FP)() =
            (double (*)(intptr_t))cantFail(ExprSymbol.getAddress());
        fprintf(stderr, "Evaluated to %f\n", FP());
        TheJIT->removeModule(H);
    }
} else {
    getNextToken();
}
}

```

具体的调用过程是，先通过 `AddModule()` 将需要执行的函数所在的 `Module` 添加到 JIT 中，然后调用 `findSymbol` 函数传入需要执行的 `Function` 的名字，这个函数会给我们一个表达式符号对象。最后我们调用 `cantFail()` 函数并将该表达式符号对象的地址作为参数传入，JIT 就会为我们执行该函数，最后我们不支持对顶级表达式进行重新求值，因此在完成释放关联内存的操作后，我们将从 JIT 中删除该模块。

3 测试

3.1 测试概要

3.1.1 测试目的

这是为解释器实验成果所写的测试文档，目的在于测试我们是否已经完成了所有需求，以及找出我们实现的功能的缺陷和可能存在的问题，并解决其中存在的问题，为我们的这一次实验工作做一个小结，以此来检测我们项目的完成情况。

3.1.2 测试概述

对于我们解释器，我们小组打算采用黑盒测试方法。即通过输入正确的以及错误的语句，对其进行语义分析，并生成 IR 代码，通过对 IR 代码添加 JIT 支持，以得到变量和函数的值。

我们根据 JIT 的运算结果来判断我们的 IR 代码是否正确，如果所有的测试都和预期一致，在多次实验后则可以判断该项目是不存在问题的。我们根据在断点处返回的变量的值是否正确来判断后端是否能传送正确的值到前端。

测试方法主要是写测试代码，进行测试。由于我们采用的万花筒的 JIT 类，因此对于其中的错误会直接输出到命令行。

我们采用穷举法，对测试的每一个细节都会有充分的展示。

3.2 测试流程

测试原理：通过程序生成测试代码，将测试代码写到控制台，由程序进行执行，如果没有出现 error 并且输出的值是正确的，说明 IR 生成是成功的。

测试：	测试操作（出错后修改并重复此流程）	结果
声明（初始化）全局变量	<pre>val a = 4; val b = 4;</pre>	没有出错，符合预期

全局变量的基本使用	<pre> a = a + 7; b = b - 7; a = a + b; b = a * b; a = b / a; </pre>	没有出错，符合预期
函数的声明与调用	<pre> fun add(x) = x + 1; add(3); add(20); </pre>	没有出错，符合预期
if ... else...then 条件控制流	<pre> val q=9; if q>4 then q=4 else q=0; q; </pre>	没有出错，符合预期
while...do(...) 循环控制流	<pre> val a=0; val b=0; while a < 10 do (b = b + a ; a = a + 1); </pre>	没有出错，符合预期

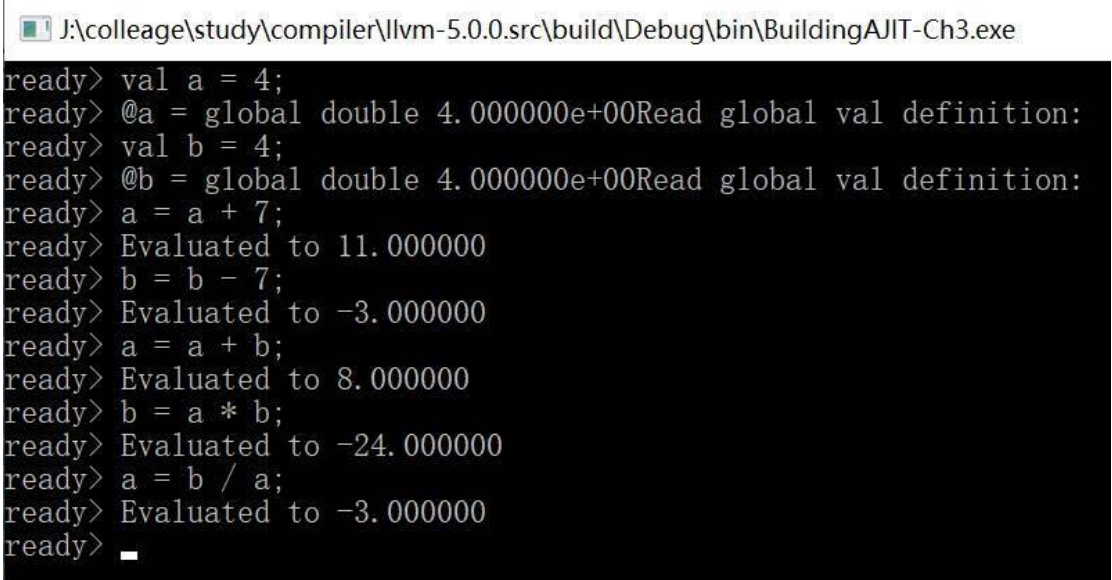
表达式的计算 (包括优先级)	$3 + 4 * 5;$ $3 * 4 + 5;$ $(3 + 4) * 5;$ $3 + (4 * 5);$ $3 * 2 + (8 / 2 - 9 \% 5) * 4;$	没有出错，符合预期
Let...in...end 局部变量的定义与使用	$\text{let val } r = 3 \text{ val } t = 5 \text{ in } r + t \text{ end};$ $\text{let val } r = 3 \text{ in while } r < 5 \text{ do } (r = r + 1;) r \text{ end};$ $\text{fun a}(e) = \text{let val } v = 9 \text{ in } v \text{ end};$ $v;$	没有出错，符合预期
函数的嵌套定义	$\text{fun t}(n) = \text{let fun s}(d) = d * 3 \text{ in } n + s(2 * n) \text{ end};$ $t(3);$ $t(1);$	没有出错，符合预期
‘ ’实现的函数重载	$\text{fun a } 1 = 1 a \text{ } n = n + a(n - 1);$ $a(1);$ $a(4);$; 没有出错，符合预期

print函数	print "HelloWord";	没有出错，符合预期
AST树展示	fun p(e) = let val c = 0 in while e > 0 do (c = e + c; e = e - 1) c; end;	没有出错，符合预期

3.3 测试截图

3.3.1 全局变量的声明与基本使用

声明了两个全局变量 a,b 并且可以进行运算：



```
J:\colleague\study\compiler\llvm-5.0.0.src\build\Debug\bin\BuildingAJIT-Ch3.exe
ready> val a = 4;
ready> @a = global double 4.000000e+00Read global val definition:
ready> val b = 4;
ready> @b = global double 4.000000e+00Read global val definition:
ready> a = a + 7;
ready> Evaluated to 11.000000
ready> b = b - 7;
ready> Evaluated to -3.000000
ready> a = a + b;
ready> Evaluated to 8.000000
ready> b = a * b;
ready> Evaluated to -24.000000
ready> a = b / a;
ready> Evaluated to -3.000000
ready> _
```

图 3.1 全局变量的声明与基本使用

3.3.2 函数的声明与调用

此处声明了一个名为 add 的函数，IR 代码证明声明成功，如果在下面调用则可以输出结果， $\text{add}(3)=3+1=4$; $\text{add}(20)=20+1=21$ ，结果正确，功能完成。

```
ready> fun add(s) = s + 1;
ready> Read function definition:
define double @add(double %s) {
entry:
    %s1 = alloca double
    store double %s, double* %s1
    %s2 = load double, double* %s1
    %addtmp = fadd double %s2, 1.000000e+00
    ret double %addtmp
}

ready> add(3);
ready> Evaluated to 4.000000
ready> add(20);
ready> Evaluated to 21.000000
```

图 3.2 函数的声明与调用

3.3.3 if ...else ...then 条件控制流

此处声明了一个全局变量 q 并且为 9，接下来对 q 进行条件控制判断，如果大于 4 便赋值为 4，否则赋值为 0；结果为 4，结果正确，功能完成。

```
ready> val q = 9;
ready> @q = global double 9.000000e+00Read global val definition:
ready> if q > 4 then q = 4 else q = 0;
ready> Evaluated to 4.000000
ready> q;
ready> Evaluated to 4.000000
ready>
```

图 3.3 if...else...then 条件控制流

3.3.4 while...do(...)循环控制流

此处为验证循环控制流，设置了 a,b 两个全局变量，在 $a < 0$ 时执行 $b=b+a$; $a=a+1$ ，循环结束后 $b=45$, $a=10$ ，结果正确，功能完成。

```
J:\colleague\study\compiler\llvm-5.0.0.src\build\Debug\bin\BuildingAJIT-Ch3.exe
ready> val a = 0;
ready> @a = global double 0.000000e+00Read global val definition:
ready> val b = 0;
ready> @b = global double 0.000000e+00Read global val definition:
ready> while a < 10 do (b = b+a;a = a+1;);
ready> Evaluated to 0.000000
ready> b;
ready> Evaluated to 45.000000
ready> a;
ready> Evaluated to 10.000000
ready>
```

图 3.4 while...do(...)循环控制流

3.3.5 表达式的计算（包括优先级）

该解释器中()的优先级大于* / %的优先级大于+ -的优先级，结果正确，功能完成。

```
ready> 3+4*5;
ready> Evaluated to 23.000000
ready> 3*4+5;
ready> Evaluated to 17.000000
ready> (3+4)*5;
ready> Evaluated to 35.000000
ready> 3+(4*5);
ready> Evaluated to 23.000000
ready> 3*2+(8/2-9%5)*4;
ready> Evaluated to 6.000000
ready> _
```

图 3.5 表达式的计算

3.3.6 Let...in...end 局部变量的定义与使用

定义了两个局部变量 r 和 t，in 返回使用局部变量的表达式：

```
ready> let val r = 3 val t = 5 in r + t end;
ready> Evaluated to 8.000000
ready> let val r = 3 in while r < 5 do (r = r+1;) r end;
ready> Evaluated to 5.000000
ready>
```

图 3.6 Let...in...end 局部变量的定义与使用 1

如果是在函数中定义的局部变量，在函数外部使用便会报错未知变量，结果正确，功能完成。

```
ready> fun a(e) = let val v = 9 in v end;
ready> Read function definition:
define double @a(double %e) {
entry:
    %v = alloca double
    %e1 = alloca double
    store double %e, double* %e1
    store double 9.000000e+00, double* %v
    %v2 = load double, double* %v
    ret double %v2
}

ready> v;
ready> Error: Unknown variable name
ready> _
```

图 3.7 Let...in...end 局部变量的定义与使用 2

3.3.7 函数的嵌套定义

在函数 t 中定义了局部嵌套函数 s ，其中 $t(3)=3+s(2*3)=3+2*3*3=21$ ，同理 $t(1)=7$ ，结果正确，功能完成。

```

ready> fun t (n) = let fun s (d) = d*3 in n + s(2*n) end;
ready> Read InsideFunction definition:
define double @s(double %d) {
entry:
    %d1 = alloca double
    store double %d, double* %d1
    %d2 = load double, double* %d1
    %multmp = fmul double %d2, 3.000000e+00
    ret double %multmp
}

Read function definition:
define double @t(double %n) {
entry:
    %n1 = alloca double
    store double %n, double* %n1
    %n2 = load double, double* %n1
    %n3 = load double, double* %n1
    %multmp = fmul double 2.000000e+00, %n3
    %calltmp = call double @s(double %multmp)
    %addtmp = fadd double %n2, %calltmp
    ret double %addtmp
}

ready> t(3);
ready> Evaluated to 21.000000
ready> t(1);
ready> Evaluated to 7.000000
ready>

```

图 3.8 函数的嵌套定义

3.3.8 |'实现的函数重载

此处用'|'使得声明的函数可以根据不同参数来进入不同的表达式，同时可以支持参数为常数的声明，由下可知结果正确，功能完成。


```

ready> fun a 1 = 1 | a n = n+a(n-1);
ready> Read function definition:
define double @a(double %n) {
entry:
  %n1 = alloca double
  store double %n, double* %n1
  %n2 = load double, double* %n1
  %eqtmp = fcmp ueq double %n2, 1.000000e+00
  %booltmp = uitofp il %eqtmp to double
  %ifcond = fcmp one double %booltmp, 0.000000e+00
  br il %ifcond, label %then, label %else

then:                                     ; preds = %entry
  br label %ifcont

else:                                     ; preds = %entry
  %n3 = load double, double* %n1
  %n4 = load double, double* %n1
  %subtmp = fsub double %n4, 1.000000e+00
  %calltmp = call double @a(double %subtmp)
  %addtmp = fadd double %n3, %calltmp
  br label %ifcont

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

ready> a(1);
ready> Evaluated to 1.000000
ready> a(4);
ready> Evaluated to 10.000000
ready> _

```

图 3.9 ‘|’ 实现的函数重载

3.3.9 Print 函数

实现了 print 函数，可以输出字符，结果正确，功能完成。

```

ready> print "HelloWord";
ready>
declare double @putchard.1(double)
HelloWord
ready> _

```

图 3.10 print 函数

3.3.10 AST 树展示

为便于展示，我们输出了一个横置的树，原本同一层的节点展示为了同一列，比如：“局部变量定义”和”in 部分节点”属于 LetExprAST 下的同一层节点,结果正

确，功能完成。

```
ready> fun p(e) = let val c = 0 in while e > 0 do (c = e+c;e=e-1) c; end;
ready> FunctionAST:
  PrototypeAST:函数名: p  参数名: e
  函数主体:
    LetExprAST:
      局部变量定义:
        c:
          IntExprAST: NumVal: 0
      in部分节点:
        ExprsAST:
          whileExprAST:
            循环条件节点:
              BinaryExprAST: OperatorName: >
                左操作数节点:
                  VariableExprAST: VarName: e
                右操作数节点:
                  IntExprAST: NumVal: 0
            循环主体节点:
              ExprsAST:
                BinaryExprAST: OperatorName: =
                  左操作数节点:
                    VariableExprAST: VarName: c
                  右操作数节点:
                    BinaryExprAST: OperatorName: +
                      左操作数节点:
                        VariableExprAST: VarName: e
                      右操作数节点:
                        VariableExprAST: VarName: c
                BinaryExprAST: OperatorName: =
                  左操作数节点:
                    VariableExprAST: VarName: e
                  右操作数节点:
                    BinaryExprAST: OperatorName: -
                      左操作数节点:
                        VariableExprAST: VarName: e
                      右操作数节点:
                        IntExprAST: NumVal: 1
              VariableExprAST: VarName: c
ready> _
```

图 3.11 AST 树展示

4 不足与改进

- 语法、语义以及中间代码生成之间存在高耦合，可以将其分离，使得程序更容易拓展。
- 暂时支持的语法规则不够完善，后续将继续加入包括数组、结构体、 λ 表达式等语法功能。
- 考虑将词法分析、语法分析、语义分析、中间代码生成相互剥离后，将其拓展至其他语言，比如 C 语言。
- 目前输入方式为命令行输入，考虑后续加入支持文本输入的功能。
- 错误的处理不够系统，不够完善，只输出了一些错误信息，而未系统地为错误归类，编号。

5 实验结论与感想

此次解释器构造实验中，我们成功的构造了一个识别 SML 语言的解释器，使用 C++ 语言实现对 SML 语言的词法分析、语法分析、静态语义分析，利用 LLVM 框架实现中间代码生成和解释执行。

在本次团队开发中我们收获颇多，首先我们将编译原理课上所学到的知识从理论过渡到了实践中，也更为深入的了解了解释器的工作机理以及编译程序的结构，同时也学习了解到了 LLVM 架构与普通架构的不同和优点，也体验到了使用这个架构的优越性。最开始几周零基础的学习 LLVM 的特性以及函数的使用，在网络上搜寻各种与之有关的资料（虽然这之类的资料非常少，而且有关 SML 语言的语法的资料也很少），效率也比较低。到后面慢慢的摸清万花筒代码每个部分的原理以及整个流程，然后努力的用技术去实现，逐步完成了最简单的词法分析、语法分析、中间代码生成和解释执行。然后我们才慢慢的在这个基础上修改和添加自己所需要的功能。我们都是各自完成自己部分最后再汇总，因为采用了瀑布式开发，所以整体上开发过程明确而且紧凑，在学习中不断进步，最终实现了一个比较完整的解释器。

最后在完成后我们也进行了技术上的总结，因为时间有些紧迫，所以我们并没有做到十分完美，但是我们项目的架构、关键算法等都可以进一步优化，对自己严格要求才能做出好的成果。学无止境，技术上的学习我们还要做很多很多。

6 附录

6.1 成员分工

6.1.1 前期准备

石亮禾：LLVM 框架调研、项目计划撰写

宛铠涛：SML 语言特性学习、文法定义

周浩宇：万花筒案例学习、学习 LLVM 官方文档

6.1.2 基本功能实现

为了方便对接，语法、IR 生成有两个人同时学习并编写，三人共同学习并完成代码。

石亮禾：语法分析、静态语义分析、中间代码生成。

宛铠涛：词法分析、语法分析。

周浩宇：中间代码生成、JIT 配置

6.1.3 扩展功能主要贡献

石亮禾：全局变量的实现、函数的重载即‘`|`’的使用、`let` 中支持局部嵌套函数、实现 `while...do...` 循环、`if...else...then`、带有符号优先级的计算、AST 树的命令行展示。

宛铠涛：进行测试并修复部分 bug、`print` 函数、改进 `prototype` 的语法分析

周浩宇：局部变量的实现、新增 `%>` 等运算符、`let...in...end` 且支持多条语句

6.1.4 报告的撰写

石亮禾：中间代码生成、AST 树展示部分

宛铠涛：词法分析

周浩宇：语法规义分析、测试文档、JIT 执行、排版

6.1.5 其他

石亮禾：最终项目演示

宛铠涛：PPT 制作

周浩宇：中期项目展示

教师评语评分

评语：

评分：

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分制评分。）