****

**编译原理实验报告**

|  |  |
| --- | --- |
| **课程名称：** | **编译原理** |
| **专业班级：** | **ACM1501** |
| **学　　号：** | **U201514613** |
| **姓　　名：** | **苟桂霖** |
| **指导教师：** | **刘铭** |
| **报告日期：** | **2018年6月30日** |

**计算机科学与技术学院**

目录

[1 选题背景 1](#_Toc518417068)

[1.1 任务 1](#_Toc518417069)

[1.2 目标 1](#_Toc518417070)

[1.3 源语言定义 2](#_Toc518417071)

[1.4 测试用例 2](#_Toc518417072)

[2 词法分析和语法分析 4](#_Toc518417073)

[2.1 单词文法描述 4](#_Toc518417074)

[2.2 词法分析器的设计 5](#_Toc518417075)

[2.3 语言文法描述 6](#_Toc518417076)

[2.4 语法分析器设计 7](#_Toc518417077)

[2.5 语法分析器实现结果展示 10](#_Toc518417078)

[3 语义分析 13](#_Toc518417079)

[3.1 语义表示方法描述 13](#_Toc518417080)

[3.2 符号表结构定义 13](#_Toc518417081)

[3.3 错误类型码定义 13](#_Toc518417082)

[3.4 语义分析实现技术 14](#_Toc518417083)

[3.4.1 加载符号表 14](#_Toc518417084)

[3.4.2 类型静态分析 15](#_Toc518417085)

[3.4.3 函数体静态分析 15](#_Toc518417086)

[3.5 语义分析结果展示 16](#_Toc518417087)

[4 中间代码生成 22](#_Toc518417088)

[4.1 中间代码格式定义 22](#_Toc518417089)

[4.2 中间代码生成规则定义 22](#_Toc518417090)

[4.3 中间代码生成过程 22](#_Toc518417091)

[4.4 中间代码生成结果展示 25](#_Toc518417092)

[5 目标代码生成 29](#_Toc518417093)

[5.1 设计思路 29](#_Toc518417094)

[5.2 目标代码运行结果展示 29](#_Toc518417095)

[6 结束语 30](#_Toc518417096)

[6.1 实验小结 30](#_Toc518417097)

[6.2 自己的心得体会 30](#_Toc518417098)

[参考文献 31](#_Toc518417099)

# 选题背景

## 任务

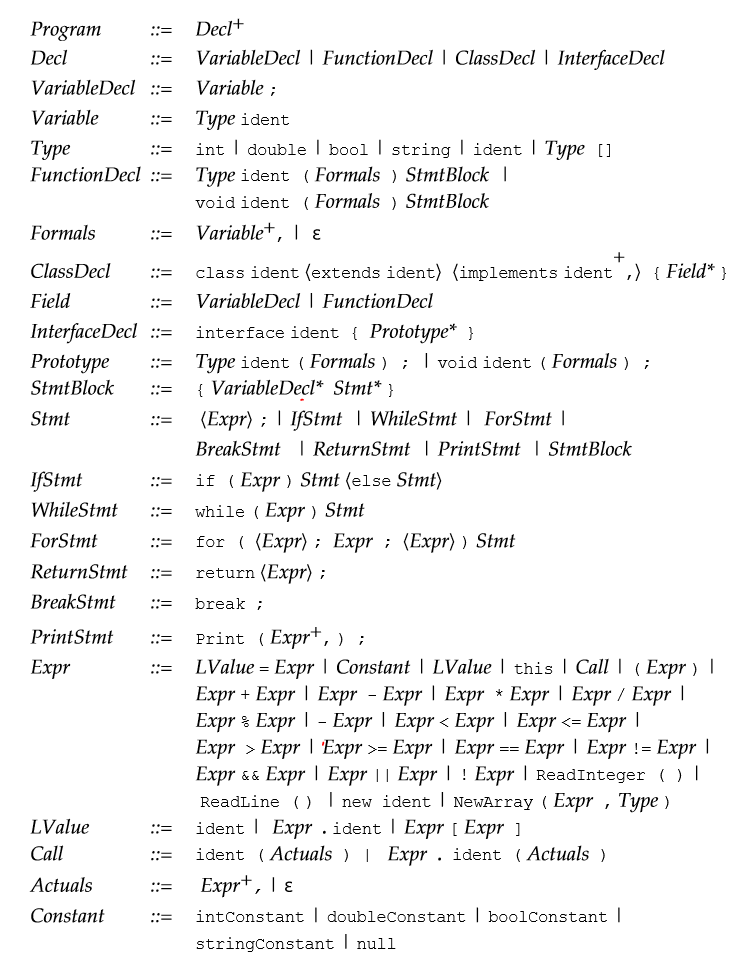
主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生对系统软件编写的能力。

## 目标

本次课程实验目标是构造一个高级语言的子集的编译器，目标代码是汇编语言或中间代码。按照任务书，实现的方案可以有很多种选择。

## 源语言定义

我选择采用decaf语言，语义上完全模拟Java的对应行为（除GC外）



在Expr中，使用算符优先文法决定运算优先级。

## 测试用例

以下代码被编译的decaf代码，实现了一个元素交错的链表。涵盖了选择分支，循环，函数多态特性。

class NodeExt extends Node {

int y;

void f() {

Print(y);

}

}

class Node {

int x;

Node next;

void f() {

Print(x);

}

}

class Main {

int main() {

Node list;

Node node;

Node iter;

int i;

NodeExt ext;

list = null;

for(i = 0; i < 100; i = i + 1){

if(i % 2 == 0){

ext = New NodeExt;

ext.y = -i;

node = ext;

node.next = list;

} else {

node = New Node;

node.x = i;

node.next = list;

}

list = node;

}

iter = list;

while(iter != null){

iter.f();

iter = iter.next;

}

return 0;

}

}

# 词法分析和语法分析

## 单词文法描述

Decaf中的词包含分为关键字，符号，标识符，数字，注释，以及最后的EOF文件终结符。

关键词如下

void string bool int double

null true false class extends interface implements

this while if else for return break

Print ReadInteger ReadLine NewArray New

符号与其对应名字如下

|  |  |
| --- | --- |
| name | symbol |
| plus | + |
| minus | - |
| mul | \* |
| div | / |
| mod | % |
| less | < |
| less\_eq | <= |
| greater | > |
| greater\_eq | >= |
| assign | = |
| eq | == |
| not\_eq | != |
| and | && |
| or | || |
| not | ! |
| semicolon | ; |
| comma | , |
| dot | . |
| square\_bra | [ |
| square\_ket | ] |
| round\_bra | ( |
| round\_ket | ) |
| curly\_bra | { |
| curly\_ket | } |

数字分为整型数字和浮点数字，满足Java中的规范

标识符分为类型标识符和普通标识符，其中类型标志符要求首字母大写，普通标识符要求首字母小写，以减轻语法分析的压力。首字母以后，都紧接零个或多个大小写字母，数字，下划线的任意排列组合。

注释分为块注释/\* ... \*/与行注释//... 行注释。

## 词法分析器的设计

词法分析我选择使用Flex实现。

Flex文件主要分为4个部分。第一部分是前置声明，需要引用合适的变量，引用正确的头文件；第二部分需要合理配置Flex选项；第三部分是规则集合，是词法分析文件的重点所在，可见后文详解；最后一部分需要定义yylex和yyeror函数，给bison提供接口。

词法分析需要提供tokens的定义，在刚开始实现时需要手工提供，但是后期只需引用bison生成的hxx文件即可。

Flex作为词法分析器，可以匹配简单的正则表达式。对于关键字和符号，其本身就是正则表达式的定义，可以直接书写。剩余的，主要是标识符，数字，注释，以及最后的EOF文件终结符。

对于数字，有两种情形，可能是整型数字，那么可以使用{DIGIT}+进行匹配。其中，DIGIT是被预先设定为[0-9]的宏。匹配到这样一个整型数字后，我们返回NUM\_int这一个token. 同理，对于浮点数字，使用[(-|+)]{DIGIT}+"."{DIGIT}\*[(e|E)(-|+){DIGIT}+]进行匹配，最后将返回NUM\_double。

对于标识符，语法分析时会区分为普通标志符（如函数，变量名）和类型标识符（如类名，接口名）。为了减轻语法分析器的压力，我们规定普通标志符是小写字母开头，紧接0到多个字母，数字或下划线的字符串，可以用[a-z\_][A-Za-z0-9\_]\*进行匹配，返回的token是ID。而对于类型标准符，我们规定其必须是大写字母开头，因而需要使用 [A-Z][A-Za-z0-9\_]\* 进行匹配，返回的token是TYPE\_ID。这样做，也符合了Java规范，使得程序可读性更好。

对于注释，需要实现的有块注释/\* ... \*/与行注释//... 行注释，直接使用正则表达式匹配到行尾的换行即可，也就是使用"//".\*\n 进行匹配。对于块注释，需要利用Flex的状态机特性，使用如下语法进行状态转移

"/\*" { BEGIN(C\_COMMENT); }

<C\_COMMENT>"\*/" { BEGIN(INITIAL); }

<C\_COMMENT>. { }

所有的注释都会被舍弃，不会返回任何token。

对于文件尾部的EOF，flex规定其必须返回值为0的token，以结束词法分析，照做即可。

最后，虽然关键字和符号的处理比较简单，但是数量较多，因而可以使用python脚本，通过读取keyword.txt 与 symbol.txt文件，自动按照Flex规定的格式生成相应的项，方便后期修改。目前本实验使用的策略是，关键字加入T\_前缀作为token名，符号如果是单字符则原样照抄，如果是多字符就使用科学命名法加上T\_前缀，以避免混淆。

最后的代码可参见lexer文件夹。

## 语言文法描述

手工将1.3源语言定义修改为如下BNF表达式，以体现terminal的处理。

Program ::= Decl+

ident ::= ID

type\_ident ::= TYPE\_ID

Decl ::= VariableDecl | FunctionDecl | ClassDecl | InterfaceDecl

VariableDecl ::= Variable ;

Variable ::= Type ident

TypeBase ::= T\_int | T\_double | T\_bool | T\_string

Type ::= TypeBase | type\_ident | Type [ ]

FunctionDecl ::= Type ident ( Formals ) StmtBlock | \

T\_void ident ( Formals ) StmtBlock

Formals ::= Variable,\*

ClassDecl ::= T\_class type\_ident Extender? Implementor? { Field\* }

Extender ::= T\_extends type\_ident

Implementor ::= T\_implements type\_ident+

Field ::= VariableDecl | FunctionDecl

InterfaceDecl ::= T\_interface type\_ident { Prototype\* }

Prototype ::= Type ident ( Formals ) ; | T\_void ident ( Formals ) ;

StmtBlock ::= { Stmt\* }

Stmt ::= Expr? ; | IfStmt | WhileStmt | ForStmt | \

BreakStmt | ReturnStmt | PrintStmt | StmtBlock | VariableDecl

IfStmt ::= T\_if ( Expr ) Stmt ElseTail?

ElseTail ::= T\_else Stmt

WhileStmt ::= T\_while ( Expr ) Stmt

ForStmt ::= T\_for ( Expr? ; Expr ; Expr? ) Stmt

ReturnStmt ::= T\_return Expr? ;

BreakStmt ::= T\_break ;

PrintStmt ::= T\_Print ( Expr,+ ) ;

Expr ::= LValue = Expr | Constant | LValue | T\_this | Call | ( Expr ) | \

Expr + Expr | Expr - Expr | Expr \* Expr | Expr / Expr | \

Expr % Expr | - Expr | Expr < Expr | Expr T\_less\_eq Expr | \

Expr > Expr | Expr T\_greater\_eq Expr | Expr T\_eq Expr | Expr T\_not\_eq Expr | \

Expr T\_and Expr | Expr T\_or Expr | ! Expr | \

T\_ReadInteger ( ) | T\_ReadLine ( ) | T\_New ident\_type | \

T\_NewArray ( Expr , Type )

LValue ::= ident | Expr . ident | Expr [ Expr ]

Call ::= ident ( Actuals ) | Expr . ident ( Actuals )

Actuals ::= Expr,\*

Constant ::= NUM\_int | NUM\_double | T\_null

## 语法分析器设计

语法分析混合使用了LALR与算符优先文法。前者只需要提前定义token以供解析，而后者还需要制定token的优先级与左右结合性。

在token.hand.yxx中，我们提供了算符优先文法相关的token的优先级与左右结合性设定。

%token ';'

%left ','

%right '='

%left T\_or

%left T\_and

%left '<' T\_less\_eq '>' T\_greater\_eq T\_eq T\_not\_eq

%left '+' '-'

%left '\*' '/' '%'

%left '.'

%precedence NEG

%right '!'

%left '[' ']'

%token '(' ')' '{' '}'

而LALR部分，通过bnf.hs规则描述文件体现。这个描述文件使用了扩展的语法，摘抄至书上给出的decaf语法规则。bnf.hs中包含? \* +三种正则表达式扩展，还包含,\*与,+这两种“逗号列表分隔记号”（语义上，?代表零或一项，+代表一项到多项，\*代表零项到多项，逗号的出现表示这些项需要使用逗号分隔）但是，bnf.hs并不能直接被bison识别，还需要进行一些变换。由于这些变换都很有规律，可以编写Python脚本parse\_gen.py，将bnf.hs的数学符号变换至bison可以识别的符号，如将":="修改为":"，同时使用CommaList, CommaListOptional等后缀，将bnf.hs中的正则扩展文法转化为普通的递归文法，生成rules.gen.yxx文件，得到正确的bison 解析规则。其中，语义动作暂时被设为空值。

本实验采用了多遍的方式生成最后的代码，因而语法分析只需要生成带标记的语法树。为了实现这一点，我们需在语义动作中构造语法树。考虑到自动化构造的难度较高且不太灵活，选择使用手工构造的方式。

首先，我们需要定义语法树的类型。为了方便未来访问者模式的实现，我们首先定义了ASTNodeBase作为基类，定义了token\_type这唯一一种共享的属性。

class ASTNodeBase {

public:

virtual void accept(class Visitor &v) = 0;

ASTNodeBase() { token\_type = "void"; }

public:

std::string token\_type;

};

using node\_ptr\_t = ASTNodeBase \*;

using optional\_node\_ptr\_t = std::optional<node\_ptr\_t>;

然后扩展出以下实体节点(注意：为了节省篇幅, 只列出了部分节点，详见node\_defs.h)：

class Integer : public ASTNodeBase {

public:

virtual void accept(Visitor& v) { v.visit(this); }

int num;

};

class Call : public ASTNodeBase {

public:

virtual void accept(Visitor& v) { v.visit(this); }

// can be nullptr

optional\_node\_ptr\_t domain\_expr;

node\_ptr\_t ident;

string /\*\*/ ident\_name;

List\* actuals;

};

//......

class New : public ASTNodeBase {

public:

virtual void accept(Visitor& v) { v.visit(this); }

// can be nullptr

node\_ptr\_t type;

};

class BinaryExpr : public ASTNodeBase {

public:

virtual void accept(Visitor& v) { v.visit(this); }

node\_ptr\_t left;

int op;

node\_ptr\_t right;

};

class Program : public ASTNodeBase {

public:

virtual void accept(Visitor& v) { v.visit(this); }

List\* decls;

};

可以看出，每一个节点都重载了accept方法，作为访问者模式的入口。节点内部定义了特有的属性，而其中很多都需要在构造时被初始化。为了简化构造时的代码，我们使用visitor\_gen.py 生成了mknode.h，给需要初始化的属性提供了形如mkDouble(double)的初始化函数。

使用语义制导翻译生成语法书，采用了S-属性制导翻译，因此信息自下而上翻译。大多数规则的翻译都比较直接，充分利用mknode.h中的构造函数即可。有以下几点需要额外注意

1. 对于自身没有额外语义，且只有单一语法子句的语义token，一般直接将下级子树根节点传递给上一级，比如 "(" expr ")": {return $2;}，如此可以减少节点种类和数量。

2. 对于函数块Block，由于涉及到变量作用域问题，必须单独给一个节点，不能直接将下级指针传递给上一级。

3. 对于空语义规则，有多种处理不同的处理方法。对于List为空的语句，需要应当创建空表。对于可选选项（如extends），应该使用std::optional给出显式的空值。对于空语句，则应该返回NoAction节点。这样做可以极大简化后期处理工作，不再需要处理空指针问题。

4. List在创建时，为了降低树的深度，采用如下方法

FieldList:

Field {$$ = mkList(); ($$)->list.push\_back($1); }

| FieldList Field {$$ = $1; ($$)->list.push\_back($2); }

;

FieldListOptional:

%empty {$$ = mkList();}

| FieldList {$$ = $1;}

;

多种不同类型的表共用一种节点，以共享处理代码，最后在访问者模式中使用特殊方法处理。

最后的翻译文件可参见rules.hand.yxx文件，它的左半部分来源于自动生成的rule.gen.yxx

最后，我们将树的根节点地址在input（对应扩展语法规则起点S'）中写进root，结束语义分析工作。

为了验证语义分析结果的正确性，也为了后期访问者模式的使用作准备，我们尝试打印整棵语法树。打印语法树的核心代码是PrintVisitor.cpp, 利用python脚本visitor\_gen.py，我们为其生成了配套的头文件与基类，通过访问者模式的accept（被重载为<<符号）方法进行递归解析，得到正确结果。在打印过程中，使用了Indent 类生成可以自动处理缩进的树结构，同时记录打印时的调用栈， 给复杂节点的处理提供更多信息。

## 语法分析器实现结果展示

打印1.4中的测试用例如下：

<Program>

. <ListItems, decl>

. | <ClassDecl>

. | \* <type\_id, NodeExt>

. | \* <Extender>

. | \* . <type\_id, Node>

. | \* <ListItems, Implementor>

. | \* <ListItems, Fields>

. | \* . <TypedVariable>

. | \* . | <TypeBase, int>

. | \* . | <-1>

. | \* . | <Identifier, y>

. | \* . <FunctionDecl, return>

. | \* . | <TypeBase, void>

. | \* . | <-1>

. | \* . | <Identifier, f>

. | \* . | <ListItems, Parameters>

. | \* . | <Block>

. | \* . | \* <ListItems, block>

. | \* . | \* . <Print>

. | \* . | \* . | <ListItems, PrintArgs>

. | \* . | \* . | \* <2>

. | \* . | \* . | \* <Identifier, y>

. | <ClassDecl>

. | \* <type\_id, Node>

. | \* <ListItems, Implementor>

. | \* <ListItems, Fields>

. | \* . <TypedVariable>

. | \* . | <TypeBase, int>

. | \* . | <-1>

. | \* . | <Identifier, x>

. | \* . <TypedVariable>

. | \* . | <type\_id, Node>

. | \* . | <-1>

. | \* . | <Identifier, next>

. | \* . <FunctionDecl, return>

. | \* . | <TypeBase, void>

. | \* . | <-1>

. | \* . | <Identifier, f>

. | \* . | <ListItems, Parameters>

. | \* . | <Block>

. | \* . | \* <ListItems, block>

. | \* . | \* . <Print>

. | \* . | \* . | <ListItems, PrintArgs>

. | \* . | \* . | \* <0>

. | \* . | \* . | \* <Identifier, x>

. | <ClassDecl>

. | \* <type\_id, Main>

. | \* <ListItems, Implementor>

. | \* <ListItems, Fields>

. | \* . <FunctionDecl, return>

. | \* . | <TypeBase, int>

. | \* . | <-1>

. | \* . | <Identifier, main>

. | \* . | <ListItems, Parameters>

. | \* . | <Block>

. | \* . | \* <ListItems, block>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <type\_id, Node>

. | \* . | \* . | <-1>

. | \* . | \* . | <Identifier, list>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <type\_id, Node>

. | \* . | \* . | <-1>

. | \* . | \* . | <Identifier, node>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <type\_id, Node>

. | \* . | \* . | <-1>

. | \* . | \* . | <Identifier, iter>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <TypeBase, int>

. | \* . | \* . | <-1>

. | \* . | \* . | <Identifier, i>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <type\_id, NodeExt>

. | \* . | \* . | <-1>

. | \* . | \* . | <Identifier, ext>

. | \* . | \* . <Assign>

. | \* . | \* . | <0>

. | \* . | \* . | <Identifier, list>

. | \* . | \* . | <NullPointer>

. | \* . | \* . <Block>

. | \* . | \* . | <For>

. | \* . | \* . | \* <Assign>

. | \* . | \* . | \* . <3>

. | \* . | \* . | \* . <Identifier, i>

. | \* . | \* . | \* . <Integer, 0>

. | \* . | \* . | \* <BinaryExpr, "<">

. | \* . | \* . | \* . <3>

. | \* . | \* . | \* . <Identifier, i>

. | \* . | \* . | \* . <Integer, 100>

. | \* . | \* . | \* <Assign>

. | \* . | \* . | \* . <3>

. | \* . | \* . | \* . <Identifier, i>

. | \* . | \* . | \* . <BinaryExpr, "+">

. | \* . | \* . | \* . | <3>

. | \* . | \* . | \* . | <Identifier, i>

. | \* . | \* . | \* . | <Integer, 1>

. | \* . | \* . | \* <Block>

. | \* . | \* . | \* . <ListItems, block>

. | \* . | \* . | \* . | \* <BinaryExpr, "==">

. | \* . | \* . | \* . | \* . <BinaryExpr, "%">

. | \* . | \* . | \* . | \* . | <3>

. | \* . | \* . | \* . | \* . | <Identifier, i>

. | \* . | \* . | \* . | \* . | <Integer, 2>

. | \* . | \* . | \* . | \* . <Integer, 0>

. | \* . | \* . | \* . | <Assign>

. | \* . | \* . | \* . | \* <0>

. | \* . | \* . | \* . | \* <Identifier, list>

. | \* . | \* . | \* . | \* <1>

. | \* . | \* . | \* . | \* <Identifier, node>

. | \* . | \* . <Assign>

. | \* . | \* . | <2>

. | \* . | \* . | <Identifier, iter>

. | \* . | \* . | <0>

. | \* . | \* . | <Identifier, list>

. | \* . | \* . <While>

. | \* . | \* . | <BinaryExpr, "!=">

. | \* . | \* . | \* <2>

. | \* . | \* . | \* <Identifier, iter>

. | \* . | \* . | \* <NullPointer>

. | \* . | \* . | <Block>

. | \* . | \* . | \* <Block>

. | \* . | \* . | \* . <ListItems, block>

. | \* . | \* . | \* . | <Call>

. | \* . | \* . | \* . | \* <2>

. | \* . | \* . | \* . | \* <Identifier, iter>

. | \* . | \* . | \* . | \* <-1>

. | \* . | \* . | \* . | \* <Identifier, f>

. | \* . | \* . | \* . | \* <ListItems, actuals>

. | \* . | \* . | \* . | <Assign>

. | \* . | \* . | \* . | \* <2>

. | \* . | \* . | \* . | \* <Identifier, iter>

. | \* . | \* . | \* . | \* <MemberDot>

. | \* . | \* . | \* . | \* . <2>

. | \* . | \* . | \* . | \* . <Identifier, iter>

. | \* . | \* . | \* . | \* . <-1>

. | \* . | \* . | \* . | \* . <Identifier, next>

. | \* . | \* . <Return>

. | \* . | \* . | <Integer, 0>

# 语义分析

## 语义表示方法描述

语义分析的主要功能由符号表加载，静态错误检查。我们使用了多遍的方法进行编译，以提供“先使用后声明”的类型支持

## 符号表结构定义

我们所定义的decaf语言，允许后向引用。这意味着对于class来说，我们可以先使用，后声明。所以，静态语义分析的第一步是加载符号表。

在decaf语言中，类class可以拥有单个继承类和多个接口，同时内部可以定义一系列成员变量与成员函数；而接口interface只能声明成员函数以供调用。

符号表被设计为了多重字典结构。

1. 顶层ClassEntries是类/接口名到内部数据的字典，而内部数据被定义为ClassBody或者InterfaceBody，使用std::variant特性进行区分，如下

using DeclEntry = std::variant<ClassBody, InterfaceBody>;

class ClassEntries : public SeqMap<string, DeclEntry>

2. InterfaceBody内部是函数名到函数定义体FuncEntry的字典，ClassBody除函数字典外还包括extender，implementors继承特性，以及variables变量字典。

3. 函数定义体FuncEntry包含返回值return\_type，形参paramenters，以及可选的指向语法树对应定义的指针body。最后者只对class中的函数定义有效，对interface的声明则设置为空值。

为了方便使用，在符号表的成员变量定义外，我们还定义了大量辅助查询的成员变量函数。

## 错误类型码定义

本编译器在静态分析阶段可以检查出的错误类型如下（通过assert）

1. extends语句继承了interface或者未定义类型
2. implements语句使用了class或者未定义类型
3. 成员变量类型是未定义类型
4. 函数返回值或者形参类型是未定义类型
5. 类型不满足接口对函数定义的要求
6. 同一类中，接口/父类/自身的相同函数名的签名不一致
7. 表达式类型不匹配（具体参见前文）
8. 在函数体中使用了未定义的变量
9. 函数调用签名和实参类型不匹配
10. 函数调用时，interface中没有函数的prototype
11. 函数调用时，对象（this指针或者expr生成的对象）没有相应的函数definition
12. 在同一个块中遇到了同名的变量定义
13. 访问了未定义的成员变量
14. 访问了未定义的成员函数

## 语义分析实现技术

### 加载符号表

根据符号表特征，我们可以着手设计LoadSymbolTableVisitor类. 由于递归调用损失了父节点的信息，同时访问者模式在设计时没有预留传递参数与返回值的接口，我们需要在LoadSymbolTableVisitor类中，使用类成员变量进行临时数据转发，同时采用StateHolder传递调用栈信息。

1. 访问者模式的数据转发以get\_id函数为例。get\_id可以读取node所对应的id信息，方法是首先情况临时成员变量current\_id，然后使用\*this << node调用accept方法解析node节点将current\_id覆盖，最后返回current\_id即可。这样做等价于将void accept 方法扩展为了 string accept方法，且无需额外加入新的访问者基类函数。

string LoadSymbolTableVisitor::get\_id(node\_ptr\_t node) {

current\_id = "@undefined";

\*this << node;

return current\_id;

}

2. 访问者模式的调用栈传递基于StateHolder类。StateHolder类利用了C++语言的RAII特性。在函数块/作用域块的开始，进行HOLD操作，StateHolder构造函数中把信息压入Visitor类调用栈call\_stack中，在函数调用完成后将信息弹出。在需要函数栈信息时，直接使用top等方法查阅父节点类型即可。

有了以上工具，我们可以按部就班地加载全局符号表。其核心代码可以参见LoadSymbolTableVisitor。

符号分析完成后，我们实现了print\_sym\_table，可以将符号表完整地打印下来。

### 类型静态分析

由于上一个pass已经将数据加载进符号表中，这一个pass不需要分析语法树，可以直接对符号表进行操作。这一步由staticAnalyse的BuildAuxInfo类完成。

由于decaf允许乱序定义类型，但是继承类的处理依赖基类，所以我们必须先分析继承关系图中的基类，再分析继承类。本实验处理方法是遇到extender时，先递归分析父类，分析完成（或者已经分析过）后再转入子类继续分析。这个过程可能走入死循环，因为可能出现继承图的有向环，此时我们通过状态染色的方法，记录基类的访问状态，保证不出现环状继承。

分析过程中，首先加载父类的available函数与变量，随后读取自身的变量与函数加入available中。随后，对比interface检查是否满足要求。

在加载变量与函数的同时，需要检查变量类型、函数返回值与形参类型是否在符号表中。对于数组，需要按照其基本类型进行处理。

根据以上步骤与方法，遍历符号表sym\_table解析每一个interface和

class的类型即可。

### 函数体静态分析

上一个pass完成后，我们拥有了每一个class的变量表和函数信息（不包含函数体，只有指向函数体语法树根节点的指针），同时也加载了interface的信息。接下来，我们所需要做的是分析函数体语法树。

这个阶段，我们的目标有：

1. 给所有的变量，包括局部变量、成员变量制定函数体唯一的uid. 特别地，我们规定成员变量按照顺序标记为0, 1, 2, ..., (k-1) 其中k是成员变量的数量。this标记为-1，其余的局部变量从k依次标记。相同的变量标记为同样的uid

2. 解析标准类型，将类型信息直接写入ident\_name中，方便后续使用

3. 分析表达式类型，给需要加注类型的表达式加上类型信息。对于单目或多目运算符，分析表达式两端是否合法，如a+b表达式中a和b必须为string/int/double类型。

4. 对于Block，内部可能有局部变量声明，需要 将其加载进Block内部的BlockExt块中，以方便查找。

5. 解析函数的调用过程，分析函数签名与函数实参是否对应。

具体实现上，解析标准类型时，我们扩展了visitor模式accept方法，使用了与get\_id相同的技巧，实现了get\_type，相当于给accept扩展出了返回值特性。

对于类型的传递，我们遵守了以下规则：

int (+,-,\*,/,%) int -> int

double (+,-,\*,/) double -> double

string + string -> string

bool (&&,||) bool -> bool

int (==, !=, <, > , >=, <=) int -> bool

double (==, !=, <, > , >=, <=) double -> bool

! bool -> bool

-int -> int

-double -> double

New TypeName -> Type

NewArray TypeName -> Type[]

"Type[]" [int] -> Type

Print( (int, double, string, bool) ) -> void

ReadLine()-> void

ReadInt() -> int

ReadDouble()->void

Type=Type -> Type

Object.ident -> member type of ident in Object

Object (==, !=) Object -> bool [要求两者有一方是nullptr，或者类型相等，Object包含string和自定义类型/数组]

本实验不允许type-upscaling，违法上述规则的语句都会报错。

除了基本类型int/double/bool外，所有的对象都是按照引用复制的，因此本次pass在填写类型的时候不区分左右值。但是在未来代码生成的时候需要明确区分。

再实现类型转化时，最麻烦的在于MemberDot语义，也就是成员变量引用。我们通过查询符号表实现，而符号表在前面的pass中已经加载待用。

而生成uid时，在进入函数体前需要将this对应的成员变量名加载进入函数体对应的block中，记作superblock. 随后，在block中解析时，进入新block时复制旧block的信息，遇到新的局部变量定义时，加入当前的block变量定义中。如果在相同函数块中有同名变量，则报错，在上层中有同名变量，可以选择覆盖掉。退出block后，将当前的block指针切换到上层block中即可。

解析函数签名与实参对应关系的方法比较直接，通过符号表找到签名后，和实参类型一一比较即可。

## 语义分析结果展示

语义分析实际上效果式加注了类型

<291, void>

<Program>

. <131, void>

. <ListItems, decl>

. | <200, void>

. | <ClassDecl>

. | \* <265, void>

. | \* <type\_id, NodeExt>

. | \* <204, void>

. | \* <Extender>

. | \* . <265, void>

. | \* . <type\_id, Node>

. | \* <131, void>

. | \* <ListItems, Implementor>

. | \* <131, void>

. | \* <ListItems, Fields>

. | \* . <282, void>

. | \* . <TypedVariable>

. | \* . | <240, void>

. | \* . | <TypeBase, int>

. | \* . | <270, void, -1>

. | \* . | <Identifier, y>

. | \* . <218, void>

. | \* . <FunctionDecl, return>

. | \* . | <240, void>

. | \* . | <TypeBase, void>

. | \* . | <270, void, -1>

. | \* . | <Identifier, f>

. | \* . | <131, void>

. | \* . | <ListItems, Parameters>

. | \* . | <167, void>

. | \* . | <Block>

. | \* . | \* <131, void>

. | \* . | \* <ListItems, block>

. | \* . | \* . <124, void>

. | \* . | \* . <Print>

. | \* . | \* . | <131, void>

. | \* . | \* . | <ListItems, PrintArgs>

. | \* . | \* . | \* <270, int, 2>

. | \* . | \* . | \* <Identifier, y>

. | <200, void>

. | <ClassDecl>

. | \* <265, void>

. | \* <type\_id, Node>

. | \* <131, void>

. | \* <ListItems, Implementor>

. | \* <131, void>

. | \* <ListItems, Fields>

. | \* . <282, void>

. | \* . <TypedVariable>

. | \* . | <240, void>

. | \* . | <TypeBase, int>

. | \* . | <270, void, -1>

. | \* . | <Identifier, x>

. | \* . <282, void>

. | \* . <TypedVariable>

. | \* . | <265, void>

. | \* . | <type\_id, Node>

. | \* . | <270, void, -1>

. | \* . | <Identifier, next>

. | \* . <218, void>

. | \* . <FunctionDecl, return>

. | \* . | <240, void>

. | \* . | <TypeBase, void>

. | \* . | <270, void, -1>

. | \* . | <Identifier, f>

. | \* . | <131, void>

. | \* . | <ListItems, Parameters>

. | \* . | <167, void>

. | \* . | <Block>

. | \* . | \* <131, void>

. | \* . | \* <ListItems, block>

. | \* . | \* . <124, void>

. | \* . | \* . <Print>

. | \* . | \* . | <131, void>

. | \* . | \* . | <ListItems, PrintArgs>

. | \* . | \* . | \* <270, int, 0>

. | \* . | \* . | \* <Identifier, x>

. | <200, void>

. | <ClassDecl>

. | \* <265, void>

. | \* <type\_id, Main>

. | \* <131, void>

. | \* <ListItems, Implementor>

. | \* <131, void>

. | \* <ListItems, Fields>

. | \* . <218, void>

. | \* . <FunctionDecl, return>

. | \* . | <240, void>

. | \* . | <TypeBase, int>

. | \* . | <270, void, -1>

. | \* . | <Identifier, main>

. | \* . | <131, void>

. | \* . | <ListItems, Parameters>

. | \* . | <167, void>

. | \* . | <Block>

. | \* . | \* <131, void>

. | \* . | \* <ListItems, block>

. | \* . | \* . <282, void>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <265, Node>

. | \* . | \* . | <type\_id, Node>

. | \* . | \* . | <270, void, -1>

. | \* . | \* . | <Identifier, list>

. | \* . | \* . <282, void>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <265, Node>

. | \* . | \* . | <type\_id, Node>

. | \* . | \* . | <270, void, -1>

. | \* . | \* . | <Identifier, node>

. | \* . | \* . <282, void>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <265, Node>

. | \* . | \* . | <type\_id, Node>

. | \* . | \* . | <270, void, -1>

. | \* . | \* . | <Identifier, iter>

. | \* . | \* . <282, void>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <240, int>

. | \* . | \* . | <TypeBase, int>

. | \* . | \* . | <270, void, -1>

. | \* . | \* . | <Identifier, i>

. | \* . | \* . <282, void>

. | \* . | \* . <TypedVariable>

. | \* . | \* . | <265, NodeExt>

. | \* . | \* . | <type\_id, NodeExt>

. | \* . | \* . | <270, void, -1>

. | \* . | \* . | <Identifier, ext>

. | \* . | \* . <275, Node>

. | \* . | \* . <Assign>

. | \* . | \* . | <270, Node, 0>

. | \* . | \* . | <Identifier, list>

. | \* . | \* . | <24, nullptr>

. | \* . | \* . | <NullPointer>

. | \* . | \* . <167, void>

. | \* . | \* . <Block>

. | \* . | \* . | <151, void>

. | \* . | \* . | <For>

. | \* . | \* . | \* <275, int>

. | \* . | \* . | \* <Assign>

. | \* . | \* . | \* . <270, int, 3>

. | \* . | \* . | \* . <Identifier, i>

. | \* . | \* . | \* . <14, int>

. | \* . | \* . | \* . <Integer, 0>

. | \* . | \* . | \* <86, bool>

. | \* . | \* . | \* <BinaryExpr, "<">

. | \* . | \* . | \* . <270, int, 3>

. | \* . | \* . | \* . <Identifier, i>

. | \* . | \* . | \* . <14, int>

. | \* . | \* . | \* . <Integer, 100>

. | \* . | \* . | \* <275, int>

. | \* . | \* . | \* <Assign>

. | \* . | \* . | \* . <270, int, 3>

. | \* . | \* . | \* . <Identifier, i>

. | \* . | \* . | \* . <86, int>

. | \* . | \* . | \* . <BinaryExpr, "+">

. | \* . | \* . | \* . | <270, int, 3>

. | \* . | \* . | \* . | <Identifier, i>

. | \* . | \* . | \* . | <14, int>

. | \* . | \* . | \* . | <Integer, 1>

. | \* . | \* . | \* <167, void>

. | \* . | \* . | \* <Block>

. | \* . | \* . | \* . <131, void>

. | \* . | \* . | \* . <ListItems, block>

. | \* . | \* . | \* . | <174, void>

. | \* . | \* . | \* . | \* <86, bool>

. | \* . | \* . | \* . | \* <BinaryExpr, "==">

. | \* . | \* . | \* . | \* . <86, int>

. | \* . | \* . | \* . | \* . <BinaryExpr, "%">

. | \* . | \* . | \* . | \* . | <270, int, 3>

. | \* . | \* . | \* . | \* . | <Identifier, i>

. | \* . | \* . | \* . | \* . | <14, int>

. | \* . | \* . | \* . | \* . | <Integer, 2>

. | \* . | \* . | \* . | \* . <14, int>

. | \* . | \* . | \* . | \* . <Integer, 0>

. | \* . | \* . | \* . | <275, Node>

. | \* . | \* . | \* . | <Assign>

. | \* . | \* . | \* . | \* <270, Node, 0>

. | \* . | \* . | \* . | \* <Identifier, list>

. | \* . | \* . | \* . | \* <270, Node, 1>

. | \* . | \* . | \* . | \* <Identifier, node>

. | \* . | \* . <275, Node>

. | \* . | \* . <Assign>

. | \* . | \* . | <270, Node, 2>

. | \* . | \* . | <Identifier, iter>

. | \* . | \* . | <270, Node, 0>

. | \* . | \* . | <Identifier, list>

. | \* . | \* . <160, void>

. | \* . | \* . <While>

. | \* . | \* . | <86, bool>

. | \* . | \* . | <BinaryExpr, "!=">

. | \* . | \* . | \* <270, Node, 2>

. | \* . | \* . | \* <Identifier, iter>

. | \* . | \* . | \* <24, nullptr>

. | \* . | \* . | \* <NullPointer>

. | \* . | \* . | <167, void>

. | \* . | \* . | <Block>

. | \* . | \* . | \* <167, void>

. | \* . | \* . | \* <Block>

. | \* . | \* . | \* . <131, void>

. | \* . | \* . | \* . <ListItems, block>

. | \* . | \* . | \* . | <29, void>

. | \* . | \* . | \* . | <Call>

. | \* . | \* . | \* . | \* <270, Node, 2>

. | \* . | \* . | \* . | \* <Identifier, iter>

. | \* . | \* . | \* . | \* <270, void, -1>

. | \* . | \* . | \* . | \* <Identifier, f>

. | \* . | \* . | \* . | \* <131, void>

. | \* . | \* . | \* . | \* <ListItems, actuals>

. | \* . | \* . | \* . | <275, Node>

. | \* . | \* . | \* . | <Assign>

. | \* . | \* . | \* . | \* <270, Node, 2>

. | \* . | \* . | \* . | \* <Identifier, iter>

. | \* . | \* . | \* . | \* <46, Node>

. | \* . | \* . | \* . | \* <MemberDot>

. | \* . | \* . | \* . | \* . <270, Node, 2>

. | \* . | \* . | \* . | \* . <Identifier, iter>

. | \* . | \* . | \* . | \* . <270, Node, -1>

. | \* . | \* . | \* . | \* . <Identifier, next>

. | \* . | \* . <145, void>

. | \* . | \* . <Return>

. | \* . | \* . | <14, int>

. | \* . | \* . | <Integer, 0>

# 中间代码生成

## 中间代码格式定义

使用LLVM的中间码

## 中间代码生成规则定义

使用LLVM Builder工具，只需要处理LLVM的API调用即可

## 中间代码生成过程

我们使用了LLVM Builder进行中间代码生成。

其核心代码为llvm\_driver文件夹下的llvm.cpp, llvm.h, llvm\_driver.cpp以及CodegenVisitor.cpp

首先，llvm.h定义了LLVMEngine类，包含了绝大多数需要记录的中间代码生成所需要的数据. 在CodegenVisitor中，也定义了部分变量，但是都是和语法树访问过程紧密相关的临时变量。

在main.cpp函数中，完成静态分析后启动llvm\_driver的codegen函数。

1. 首先在LLVMEngine的构造函数中，初始化LLVM所需要的环境变量，定义基本类型，同时加载runtime.c中的函数和类型。

2. 在codegen中，为所有的class添加声明

3. 对于符号表中的元素，如果是interface，使用eng.decare\_func生成其对应函数签名；如果是类型，在类型声明的基础上补充其定义（成员变量），生成函数签名

4. 调用eng.create\_call\_table生成面向对象关键数据虚表.

5. 遍历所有类的函数声明，为其生成相应的代码

6. 输出生成好的中间代码。

在第一步中，我们加载了runtime.c的大量辅助函数，主要是读入数据，打印数据，内存分配，字符串拼接，与虚表查询。我们还加载了和虚表紧密相关的\_\_entry/\_\_vtable

两种类型。

在第三步生成函数签名的过程中，我们需要考虑面向对象的设计，为所有的函数签名添加this指针。LLVM是强类型语言，不同类型的指针相互转化是需要cast，但是同一个函数会被其子类甚至interface调用。为了方便起见，我们在签名中给this的位置设置为void\*，使用前后都需要进行统一的cast，简化了虚函数调用的设计。

在第四步中需要设计合理的vtable结构。我们的最终设计如下

typedef struct \_\_entry {

unsigned long long fid;

void\* fptr;

} \_\_entry;

typedef struct \_\_vtable {

unsigned long long len;

\_\_entry table[1];

} \_\_vtable;

\_\_entry是虚表的一条记录，包含全局唯一的函数签名uid，以及对应函数指针。

\_\_vtable中包含虚表的长度len，以及函数记录表table[]。 表中的1仅仅起到占位的作用，其实际长度由len决定

第5步是代码生成的关键步骤，使用LLVMEngine和CodegenVisitor共同完成。

生成函数定义时，我们规定Main类下必须有一个main(void)函数作为程序的entry\_point，同时Main类下不允许有成员变量或者其他函数。因此，我们只需要声明main函数类型，设置基本块插入点，清空变量引用表，即可开始使用CodegenVisitor遍历语法子树生成main函数的具体代码

而对于普通函数，则还需要处理形参，this与类成员变量的访问。在静态检查的过程中，我们已经生成了uid，所以需要将this和类成员变量加载进局部表的相应位置。函数的形参也需要CreateAlloca得到可以修改的左值副本后，加载进变量引用表。接下来，同样开始使用CodegenVisitor遍历语法子树，生成函数的具体代码。

在CodegenVisitor中，进行了实际的代码生成动作。

每当语句块Block后，需要定义语句块中出现的局部变量，随后逐句翻译。

翻译到表达式语句时，需要递归分析。叶子节点一般是常量或者变量，前者可以直接生成，后者查找变量引用表得到地址，使用load语句读入即可。而内部节点，可能是各类简单表达式，如加减乘除，可以直接翻译为对应的语句；还有可能是赋值语句，此时我们需要区分左值和右值：等号左边类型是store语句的地址，等号右边是store语句的值，通过right\_value状态在解析过程中处理。

此外，我们需要处理控制流语句，主要有if, for, while. 以最麻烦的for语句为例，我们首先处理init表达式，随后定义condBB，loopBB, nextBB三种基本块。首先，从init结束点跳转进入condBB；在condBB中，插入比较分支语句，分别跳转至loopBB与nextBB；在loopBB中，解析循环体本身，再插入step语句，跳转至condBB；最后nextBB设置插入点，继续以后的代码生成。

对于return，由于其本身对应一条汇编指令，直接作为普通语句处理即可。

而对于break语句，需要利用for与while中记录好的地址，跳转到相应的nextBB处。

对于输入ReadXXX和打印Print，我们直接根据类型调用runtime中预定义的函数即可。

对于New语句，我们计算类型所需要占用的内存空间，通过runtime中预定义的dog\_malloc获取需要的空间，然后初始化其虚表地址。

最后，我们还需要处理成员函数调用语句。

首先，我们已经在eng.create\_call\_table中，生成了每一个类对应的虚表。

举例说明。假设我们有以下代码

class Node{

void f(){}

void g(){}

}

class NodeExt extends Node{

void g(){}

void h(){}

}

class Main{

int main(){

Node obj = New NodeExt;

obj.g();

}

}

不妨设我们有一张函数唯一编码表

|  |  |
| --- | --- |
| 0 | f |
| 1 | fun\_a |
| 2 | g |
| ... | ... |
| 7 | h |

则NodeExt的虚表结构如下：

|  |  |
| --- | --- |
| 0 | Node.f |
| 2 | NodeExt.g |
| 7 | NodeExt.h |

我们来分析main函数语句的调用过程。在New中，我们将NodeExt的虚表首地址写入了第一个成员变量。函数调用obj.g()中，我们首先找到当前函数g所对应的唯一编号fid=2，随后加载被调用对象obj的第一个成员变量将虚表首地址加载出来（这里为NodeExt的虚表地址）。使用runtime中的load\_ptr函数，我们通过虚表和fid，二分查找找到成员函数入口地址NodeExt.g，即为我们需要调用的函数。找到函数后，将this指针和形参传入并调用即可。

## 中间代码生成结果展示

; ModuleID = 'dog-decaf'

source\_filename = "dog-decaf"

%struct.\_\_entry = type { i64, i8\* }

%struct.string = type { i32, i8\* }

%struct.\_\_vtable = type { i64, [1 x %struct.\_\_entry] }

%NodeExt = type { %struct.\_\_vtable\*, i32, %Node\*, i32 }

%Node = type { %struct.\_\_vtable\*, i32, %Node\* }

@\_\_vtable\_Node = private constant { i64, [1 x %struct.\_\_entry] } { i64 1, [1 x %struct.\_\_entry] [%struct.\_\_entry { i64 0, i8\* bitcast (void (i8\*)\* @Node\_\_f to i8\*) }] }

@\_\_vtable\_NodeExt = private constant { i64, [1 x %struct.\_\_entry] } { i64 1, [1 x %struct.\_\_entry] [%struct.\_\_entry { i64 0, i8\* bitcast (void (i8\*)\* @NodeExt\_\_f to i8\*) }] }

declare i32 @readint()

declare %struct.string\* @readline()

declare void @printint(i32)

declare void @printdouble(double)

declare void @printbool(i1)

declare void @printstring(%struct.string\*)

declare %struct.string\* @string\_cat(%struct.string\*, %struct.string\*)

declare i8\* @dog\_malloc(i32)

declare i8\* @load\_ptr(%struct.\_\_vtable\*, i32)

define void @NodeExt\_\_f(i8\*) {

entry:

%1 = bitcast i8\* %0 to %NodeExt\*

%2 = getelementptr %NodeExt, %NodeExt\* %1, i32 0, i32 1

%3 = getelementptr %NodeExt, %NodeExt\* %1, i32 0, i32 2

%4 = getelementptr %NodeExt, %NodeExt\* %1, i32 0, i32 3

%ld = load i32, i32\* %4

call void @printint(i32 %ld)

ret void

}

define void @Node\_\_f(i8\*) {

entry:

%1 = bitcast i8\* %0 to %Node\*

%2 = getelementptr %Node, %Node\* %1, i32 0, i32 1

%3 = getelementptr %Node, %Node\* %1, i32 0, i32 2

%ld = load i32, i32\* %2

call void @printint(i32 %ld)

ret void

}

define i32 @main() {

entry:

%local\_decl = alloca %NodeExt\*

%local\_decl1 = alloca i32

%local\_decl2 = alloca %Node\*

%local\_decl3 = alloca %Node\*

%local\_decl4 = alloca %Node\*

store %Node\* null, %Node\*\* %local\_decl3

store i32 0, i32\* %local\_decl1

br label %cond

cond: ; preds = %cont, %entry

%ld = load i32, i32\* %local\_decl1

%0 = icmp slt i32 %ld, 100

br i1 %0, label %whileloop, label %next

whileloop: ; preds = %cond

%ld5 = load i32, i32\* %local\_decl1

%1 = srem i32 %ld5, 2

%2 = icmp eq i32 %1, 0

br i1 %2, label %then, label %else

then: ; preds = %whileloop

%dog\_malloc = call i8\* @dog\_malloc(i32 32)

%3 = bitcast i8\* %dog\_malloc to %NodeExt\*

%lenAddr = getelementptr %NodeExt, %NodeExt\* %3, i32 0, i32 0

store %struct.\_\_vtable\* bitcast ({ i64, [1 x %struct.\_\_entry] }\* @\_\_vtable\_NodeExt to %struct.\_\_vtable\*), %struct.\_\_vtable\*\* %lenAddr

store %NodeExt\* %3, %NodeExt\*\* %local\_decl

%ld6 = load i32, i32\* %local\_decl1

%4 = sub i32 0, %ld6

%ld7 = load %NodeExt\*, %NodeExt\*\* %local\_decl

%5 = getelementptr %NodeExt, %NodeExt\* %ld7, i32 0, i32 3

store i32 %4, i32\* %5

%ld8 = load %NodeExt\*, %NodeExt\*\* %local\_decl

%6 = bitcast %NodeExt\* %ld8 to %Node\*

store %Node\* %6, %Node\*\* %local\_decl4

%ld9 = load %Node\*, %Node\*\* %local\_decl3

%ld10 = load %Node\*, %Node\*\* %local\_decl4

%7 = getelementptr %Node, %Node\* %ld10, i32 0, i32 2

store %Node\* %ld9, %Node\*\* %7

br label %cont

else: ; preds = %whileloop

%dog\_malloc11 = call i8\* @dog\_malloc(i32 24)

%8 = bitcast i8\* %dog\_malloc11 to %Node\*

%lenAddr12 = getelementptr %Node, %Node\* %8, i32 0, i32 0

store %struct.\_\_vtable\* bitcast ({ i64, [1 x %struct.\_\_entry] }\* @\_\_vtable\_Node to %struct.\_\_vtable\*), %struct.\_\_vtable\*\* %lenAddr12

store %Node\* %8, %Node\*\* %local\_decl4

%ld13 = load i32, i32\* %local\_decl1

%ld14 = load %Node\*, %Node\*\* %local\_decl4

%9 = getelementptr %Node, %Node\* %ld14, i32 0, i32 1

store i32 %ld13, i32\* %9

%ld15 = load %Node\*, %Node\*\* %local\_decl3

%ld16 = load %Node\*, %Node\*\* %local\_decl4

%10 = getelementptr %Node, %Node\* %ld16, i32 0, i32 2

store %Node\* %ld15, %Node\*\* %10

br label %cont

cont: ; preds = %else, %then

%ld17 = load %Node\*, %Node\*\* %local\_decl4

store %Node\* %ld17, %Node\*\* %local\_decl3

%ld18 = load i32, i32\* %local\_decl1

%11 = add i32 %ld18, 1

store i32 %11, i32\* %local\_decl1

br label %cond

next: ; preds = %cond

%ld19 = load %Node\*, %Node\*\* %local\_decl3

store %Node\* %ld19, %Node\*\* %local\_decl2

br label %cond20

cond20: ; preds = %whileloop22, %next

%ld21 = load %Node\*, %Node\*\* %local\_decl2

%12 = ptrtoint %Node\* %ld21 to i64

%13 = icmp ne i64 %12, 0

br i1 %13, label %whileloop22, label %next26

whileloop22: ; preds = %cond20

%ld23 = load %Node\*, %Node\*\* %local\_decl2

%14 = getelementptr %Node, %Node\* %ld23, i32 0, i32 0

%loadSymPtr = load %struct.\_\_vtable\*, %struct.\_\_vtable\*\* %14

%15 = call i8\* @load\_ptr(%struct.\_\_vtable\* %loadSymPtr, i32 0)

%16 = bitcast i8\* %15 to void (i8\*)\*

%17 = bitcast %Node\* %ld23 to i8\*

call void %16(i8\* %17)

%ld24 = load %Node\*, %Node\*\* %local\_decl2

%18 = getelementptr %Node, %Node\* %ld24, i32 0, i32 2

%ld25 = load %Node\*, %Node\*\* %18

store %Node\* %ld25, %Node\*\* %local\_decl2

br label %cond20

next26: ; preds = %cond20

ret i32 0

}

!llvm.module.flags = !{!0}

!0 = !{i32 7, !"PIC Level", i32 2}

# 目标代码生成

## 设计思路

由于直接生成了LLVM的中间码，可以使用llc工具直接编译到x86-64/MIPS等一系列目标代码。需要注意的是必须使用fPIC进行汇编。

汇编之后，需要编写运行库并进行编译链接，详情可见runtime.c

最后的脚本如下：

make -C build

cd final\_build

clang -fPIC -c runtime.c -emit-llvm

cd ..

./build/main ./data/naive.decaf

cd final\_build

rm final -f

llc main.ll -relocation-model=pic

gcc -fPIC main.s runtime.c -o final

./final

echo 'exitcode = ' $?

## 目标代码运行结果展示

99 -98 97 -96 95 -94 93 -92 91 -90 89 -88 87 -86 85 -84 83 -82 81 -80 79 -78 77 -76 75 -74 73 -72 71 -70 69 -68 67 -66 65 -64 63 -62 61 -60 59 -58 57 -56 55 -54 53 -52 51 -50 49 -48 47 -46 45 -44 43 -42 41 -40 39 -38 37 -36 35 -34 33 -32 31 -30 29 -28 27 -26 25 -24 23 -22 21 -20 19 -18 17 -16 15 -14 13 -12 11 -10 9 -8 7 -6 5 -4 3 -2 1 0 exitcode = 0

# 结束语

## 实验小结

本次实验我掌握了Flex，Bison，LLVM工具链的初步使用技巧，对编译前端和中端的知识体系有了更加深刻的认识。

## 自己的心得体会

本次实验工程量巨大，我为了减少工作量写了大量自动化生成脚本，即使这样最终也收获了4000行的代码量。如果算上自动生成的cpp/h文件，已经突破了7000行。除了Flex/Bison/LLVM外，我没有使用任何框架，所有的代码部分都是自己设计的，这极大提高了我的编码能力。

参考文献

[1] 吕映芝等. 编译原理(第二版). 北京：清华大学出版社，2005

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

[3] 王元珍等. 80X86汇编语言程序设计. 武汉：华中科技大学出版社,2005

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008