

浙江大学

本科实验报告

课程名称：编译原理

学 院：计算机科学与技术学院

系：计算机科学与技术

专 业：计算机科学与技术

组 号：6

指导教师：冯雁

姓名	学号

2022年 5月 22日

Maggot : A C-minus Compiler

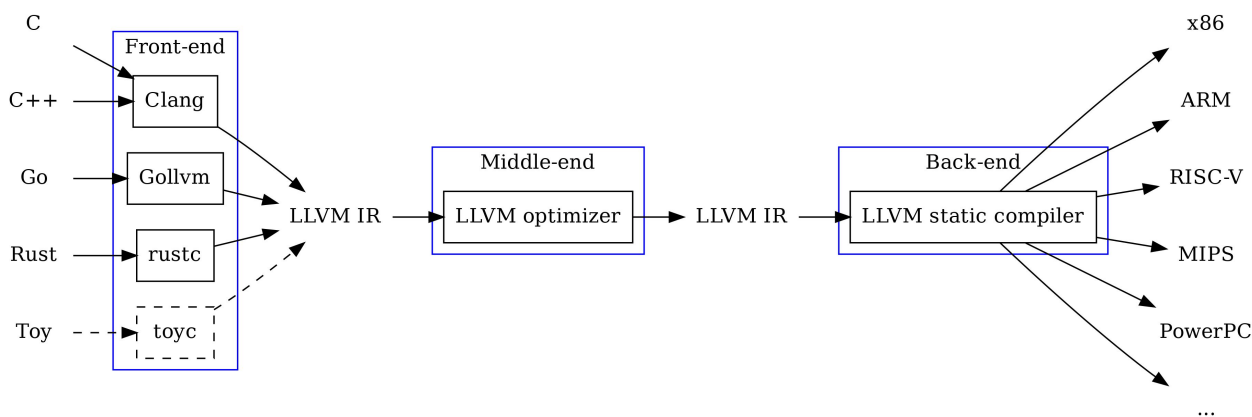
Zhejiang University; Computer Science and Technology Colledge

1 项目简介

本项目是基于flex, bison以及LLVM, 使用c++11实现的类C语言语法编译器前端, 可以将输入的源代码文件, 使用flex结合yacc对源代码进行词法语法分析, 根据parsing tree生成整个源代码文件对应的抽象语法树AST(Abstract Syntax Tree), 再根据生成的AST, 调用LLVM相应的接口, 最终生成符合LLVM中间语言语法, 机器无关的中间代码, LLVM IR(intermediate representation)

最后, 通过封装的接口, 本项目依靠llvm-as, llc, clang++工具, 可以选择输出源代码解析结果的机器无关的IR代码, 以及BitCode代码, 以及机器有关的汇编代码文件或者可执行文件

同时本项目提供了链接C标准库的方式, 可以通过编写库文件的方式 调用C的标准库



本项目解析的语法是C语言的一个子集, 同时扩展了一部分语法, 目前已支持的数据类型包括

- void
- bool
- char
- int
- float
- string
- 一维数组

支持的主要语法包括:

- 变量的声明以及初始化 (暂不支持数组的初始化)
- 字符数组初始化
 - 函数原型声明, 函数定义和函数调用, 数组函数传参
- 控制流语句if-else和while以及任意层级的嵌套使用
- 行注释和块注释
- 二元运算符、赋值以及表达式的隐式类型转换
- 全局变量的使用

2 运行环境

实验测试环境均在以下环境下测试

```
Linux Maka 5.4.0-90-generic #101-Ubuntu SMP Fri Oct 15 20:00:55 UTC 2021 x86_64
x86_64 x86_64 GNU/Linux
```

2.1 编译安装

想要运行程序,最简单的方法是编译和安装所有依赖

实验环境主要依赖于bison, flex, llvm-12

```
sudo apt-get install -y build-essential automake cmake git flex bison
sudo apt-get install -y lld-12 llvm-12 llvm-12-dev clang-12 || sudo apt-get
install -y lld llvm llvm-dev clang
sudo apt-get install -y gcc-$(gcc --version|head -n1|sed 's/\..*//'|sed 's/.*/
//')-plugin-dev libstdc++-$(gcc --version|head -n1|sed 's/\..*//'|sed 's/.*/
//')-dev
git clone https://github.com/zpqqql0/perpetual-motion-machine.git
make
```

2.2 工程结构

maggot/	
├─ include	头文件
│ └─ codeGen.h	生成中间代码
│ └─ declaration.h	声明类结点
│ └─ expression.h	表达式结点
│ └─ main.h	
│ └─ statement.h	语句式结点
│ └─ utils.h	基本工具
├─ lib	
│ └─ libstd.cpp	输入输出库
├─ maggot	可执行文件
├─ makefile	
├─ src	源文件
│ └─ codeGen.cpp	生成中间代码
│ └─ maggot.l	词法分析
│ └─ maggot.ypp	语法分析与AST
│ └─ main.cpp	
│ └─ utils.cpp	基本工具
└─ test	
└─ ...	

2.3 运行

- debug: 输出调试信息
- show-ast: 输出生成的抽象语法树
- show-ir: 输出生成的LLVM IR中间代码
- output: 指定输出文件的路径
- ir: 指定输出文件为生成的IR代码
- bitcode: 指定输出文件为生成的bitcode代码
- assembly: 指定输出文件为生成的assembly代码
- binary: 指定输出

Usage: ./maggot [OPTIONS] file

OPTIONS:

```
--debug[=false]
    Enable [disable] debug log messages.
    Default: false (disabled)

--help, -h
    Print this help message.

--show-ast[=false]
    Display the AST.
    Default: false (disabled)

--show-ir[=false]
    Display the LLVM::IR generated by CodeGen
    Default: false (disabled)

--output, -o <file>
    Place the output into <file>.
    Default: a.out

--ir[=true]
    Output the IR Code to file
    Default: true (enabled)

--bitcode[=false]
    Output the Bitcode to file
    Default: false (disabled)

--assembly[=false]
    Output the Assembly to file
    Default: false (disabled)

--binary[=false]
    Output the Binary to file
    Default: false (disabled)
```

2.4 Examples

```
$ make
clang++ -Wall -fPIC -I include/ `llvm-config --cxxflags` -fno-rtti -frtti -c
src/main.cpp -o src/main.o
clang++ -Wall -fPIC -I include/ `llvm-config --cxxflags` -fno-rtti -frtti -c
src/utils.cpp -o src/utils.o
yacc -d src/maggot.ypp -v -o src/y.tab.cpp
src/maggot.ypp: warning: 1 shift/reduce conflict [-Wconflicts-sr]
clang++ -Wall -fPIC -I include/ `llvm-config --cxxflags` -fno-rtti -frtti -c
src/y.tab.cpp -o src/y.tab.o
lex --outfile=src/lex.yy.cpp src/maggot.l
clang++ -Wall -fPIC -I include/ `llvm-config --cxxflags` -fno-rtti -frtti -c
src/lex.yy.cpp -o src/lex.yy.o
clang++ -Wall -fPIC -I include/ `llvm-config --cxxflags` -fno-rtti -frtti -c
src/codeGen.cpp -o src/codeGen.o
clang++ -Wall -fPIC -I include/ `llvm-config --cxxflags` -fno-rtti -frtti `llvm-
config --system-libs --ldflags --libs core` src/main.o src/utils.o src/y.tab.o
src/lex.yy.o src/codeGen.o -o maggot
$ ./maggot --show-ast --show-ir --debug -o ./test.ll ./test/uminus.cpp
error: Cannot display and output at the same time!
$ ./maggot --show-ast --show-ir --debug ./test/uminus.cpp
AST:
....
IR:
.....
$ ./maggot --show-ast --debug -o ./uminus.ll
```

3 词法分析

本编译器实现的语法是类似C++的语法。我们使用Lex将输入转化为一系列的标记token。本编译器定义的token包括：C语言关键字if、else、return、while、int、float、bool等保留关键字token，加减乘除、位运算、括号、逗号等符号token，表示用户自定义的标志符token、数字常量token、字符串常量token、布尔值常量token等，以及注释token。其中，浮点数常量支持小数表示法和指数表示法，整型常量支持十进制表示、八进制表示以及十六进制表示。

我们需要在yacc源文件maggot.ypp中声明这些token，并在lex源文件maggot.l中定义这些token对应的操作，操作主要是将token的值传给yacc进行处理或者通知yacc匹配的token。

4 语法分析

4.1 语法树的构建

语法分析是本项目的关键环节之一。由于源代码是以字符串即字节流的形式存在的，难以进行后续的编译解释工作，因此我们需要将源代码转化成能够反映语法规则的数据结构。将源代码转化成语法树也方便我们检查编译器运行的正确性。

语法树的构建过程其实也就是yacc的BNF语法规则的编写过程。我们的语法在教材提供的C-语法规则的基础上进行改进，与C-语法规则的区别有：

1. 支持使用布尔类型和浮点类型

2. 支持在一条声明语句中声明多个变量或数组
3. 支持声明函数原型
4. 支持编写行注释和块注释
5. 支持在复合语句中任意进行变量声明和编写语句
6. 支持使用负数

语法可以生成的代码举例如下：

```
int sort(int low, int high);
// sort a list
int sort(int low, int high)
{
    // sort a list
    int i;
    int k;
    // loop
    while (i < high - 1)
    {
        int t;
        k = minloc(i, high);
        t = x[k];
        x[k] = x[i];
        x[i] = t;
        i = i + 1;
    }
    return 0;
}
```

4.2 语法树的输出

编写语法规则其实已经构建出了语法树，还需要我们将这个语法树保存下来并输出。考虑到树的特点以及OOP的编程思想，我们需要为语法树的每一种节点定义一个对象，用指针来维护树的结构。在选择定义哪一些结点对象以及如何输出语法树时，我们借鉴了clang输出的语法树，并尝试在维护语法树及输出语法树的时候模仿其风格，通过以下命令可以查看clang对于某个源文件构建的语法树：

```
clang -fsyntax-only -Xclang -ast-dump test/sample.cpp
```

我们定义了一个基类BaseAST作为所有节点的基类，避免了复杂的继承关系。所有出现在语法树中的类均是BaseAST的子类。基类成员包括一个向量children，其保存着指向子节点的指针，使得我们可以通过根节点访问到整个树的结构。每个对象需要做的就是保存其本身的信息以及子节点的指针。如以下例子所示：

```
class BaseAST
{
public:
    std::vector<BaseAST*> children;
    ...
    virtual void print(int level) {
        padding(level);
    }
}
```

```

        std::cout << "BASE" << std::endl;
    }

    virtual llvm::Value *CodeGen() {
        return nullptr;
    }
};

class Program: public BaseAST
{
private:
    /* data */
public:
    ...
    virtual void print(int level = 0) {
        std::cout << "AST: " << std::endl;
        for (size_t i = 0; i < children.size(); i++)
        {
            children[i]->print(0);
        }
    }

    virtual llvm::Value *CodeGen();
};

class IntegerAST : public BaseAST
{
    int value;

public:
    ...
    virtual void print(int level)
    {
        padding(level);
        cout << _PURPLE << "IntegerLiteral "
            << " " << _CYAN << value << White << endl;
        return;
    }

    virtual llvm::Value *CodeGen();
};

```

在yacc源文件中，当发生归约时，我们提取结点的信息并新建对象，保存结点的信息到对象中并维护树的结构，如在关系表达式中，为操作符新建一个对象并保存其两个操作符：

```

relation_expr: relation_expr LE addition { $$ = new BinaryOpAST(OPLE); $$-
>children.push_back($1); $$->children.push_back($3);}
        | relation_expr LT addition { $$ = new BinaryOpAST(OPLT); $$-
>children.push_back($1); $$->children.push_back($3);}
        | relation_expr GT addition { $$ = new BinaryOpAST(OPGT); $$-
>children.push_back($1); $$->children.push_back($3);}
        | relation_expr GE addition { $$ = new BinaryOpAST(OPGE); $$-
>children.push_back($1); $$->children.push_back($3);}
        | relation_expr EQ addition { $$ = new BinaryOpAST(OPEQ); $$-
>children.push_back($1); $$->children.push_back($3);}
        | relation_expr NEQ addition { $$ = new BinaryOpAST(OPNEQ); $$-
>children.push_back($1); $$->children.push_back($3);}
        | addition { $$ = $1;}
;

```

最后，在主函数中解析源文件完毕后，调用根节点的输出函数。每个结点都会先输出结点本身的信息再打印子结点的信息。解析sample.cpp得到的部分语法树输出如下：

```

[n7utb@Maka:~/perpetual-motion-machine/maggot]$ ./maggot --show-ast --show-ir test/sample.cpp
AST:
`-DeclStmt
  `~VarDecl x global int *
  `~VarDecl test global int
    `~IntegerLiteral 0
  `~VarDecl num global int
    `~BinaryOperator -
      `~IntegerLiteral 222
      `~UnaryOperator -
        `~IntegerLiteral 233
`-DeclStmt
  `~VarDecl flag global bool
    `~BoolLiteral false
`-FunctionDecl minloc int
  `~ParmVarDecl low int
  `~ParmVarDecl high int
`-FunctionDecl sort void
  `~ParmVarDecl low int
  `~ParmVarDecl high int
`-FunctionDecl minloc int
  `~ParmVarDecl low int
  `~ParmVarDecl high int
  `~CompoundStmt
    `~DeclStmt
      `~VarDecl i local int
    `~DeclStmt
      `~VarDecl y local int
    `~DeclStmt
      `~VarDecl k local int

```

5 语义分析

5.1 实现概述

编程语言的语义指的是一段代码的实际含义，例如 `int a = b + 3;` 即定义一个整型变量a，并且将b与3相加后的结果储存在a中。这里3的明确定义为：32位有符号整型字面量。而在其与变量b相加以及将结果储存在a的两个步骤中，均有可能因为类型不一致而发生相应的类型转换。

在语义分析中，主要的关键步骤就是类型检查。具体而言，类型检查需要做的事情有：

- 定义隐式类型转化规则
- 明确每一个表达式的声明类型
- 判定每一个字面量的类型
- 判断赋值与传参的时候，是否需要隐式类型转换
- 判断表达式中是否存在类型引起的非法

对于表达式或字面量的类型，我们在相应的AST节点中使用了type类，配合宏定义记录其类型。并在计算表达式时，进行类型检查与转换。

Maggot按照是否是指针，一共使用了llvm的两个类用于储存类型信息，分别是type与ArrayType。

两者均具有生成某一特定类型的类、获得该实例的类型信息的一些内置函数，可供调用。

5.2 类型系统

Maggot的类型均采用宏定义的形式进行判别，代码中以重载实现的两个同名函数进行具体类型的生成：

```
Type *TypeGen(int type)
{
    switch (type)
    {
        case TYPEINT:
            return Type::getInt32Ty(TheContext);
        case TYPEINTARRAY:
            return Type::getInt32Ty(TheContext);
        case TYPEFLOAT:
            return Type::getFloatTy(TheContext);
        case TYPEFLOATARRAY:
            return Type::getFloatTy(TheContext);
        case TYPEBOOL:
            return Type::getInt1Ty(TheContext);
        case TYPEVOID:
            return Type::getVoidTy(TheContext);
        case TYPECHAR:
            return Type::getInt8Ty(TheContext);
        default:
            return Type::getInt32Ty(TheContext);
    }
}

ArrayType *TypeGen(int type, int size)
{
    switch (type)
    {
        case TYPEINTARRAY:
            return ArrayType::get(TypeGen(type), size);
        case TYPEFLOATARRAY:
            return ArrayType::get(TypeGen(type), size);
        default:
            return ArrayType::get(TypeGen(TYPEINT), size);
    }
}
```

```
}  
}
```

其中case后的即为实现的类型宏定义。注意到default，这是Maggot设定的默认类型，即当发生错误从而导致发送的type值并无对应宏定义时所选择的类型。

这样的实现方式，便捷了后续利用llvm对类型相关操作（包括生成、转换等）的进一步处理。

5.3 类型转换

类型转换主要出现在表达式（包括赋值与计算）中。下面举一例子说明这一转换的实现机制

具体的一段核心代码（二元表达式）如下：

```
// if return float, make some necessary conversion  
if (fflag && type != OPASSIGN)  
{  
    if (!L->getType()->isFloatTy())  
    {  
        conversion = true;  
        L = Builder.CreateSIToFP(L, TypeGen(TYPEFLOAT));  
    }  
    if (!R->getType()->isFloatTy())  
    {  
        conversion = true;  
        R = Builder.CreateSIToFP(R, TypeGen(TYPEFLOAT));  
    }  
}
```

当发现需要返回类型为浮点数时，检查二元表达式运算符两侧是否均为浮点数类型，如果至少有一方不是的时候，将conversion置位，从而在后续计算过程中先转换类型再运算。

```
case OPADD:  
    return conversion  
        ? fflag ? Builder.CreateFAdd(L, R, "addftmp") : Builder.CreateAdd(L, R,  
"addtmp")  
        : Builder.CreateNSWAdd(L, R, "addtmp");
```

如进行加法时，预先判别是否需要进行类型转换，如果需要则进行浮点数转换的加法，否则进行普通的加法运算。

6 优化考虑

6.1 符号修饰

20世纪70年代以前，编译器编译代码时产生的目标文件中，符号名与相应的变量和函数的名字是一样的，随着编程语言的发展，例如C语言，如果一个C语言程序要使用这些库的话，其自身就不能使用这些库中已经声明了的函数和变量的名字作为符号名，否则将会跟现有的目标文件发生名称冲突。

为了防止这类符号名冲突，各平台下的编程语言规定了各自的符号生成语法。如C在UNIX下在函数名和变量前加下划线作为符号名。这种给函数名增加特定符号来使其符号名唯一的方式就是符号修饰。

这种简单的符号修饰没有从根本上解决符号冲突的问题，比如同一种编程语言编写的目标文件之间还有可能产生符号冲突，当程序很大时，不同的部门之间也有可能会产生符号冲突，于是C++这类语言开始加上了命名空间(namespace)来解决多模块符号冲突问题。

为了支持C++拥有类、继承、虚机制、重载、命名空间等这些特性，人们发明了符号修饰修饰（或者称为符号改编），被修饰之前的函数名称以及其返回值和参数等信息，被称为函数签名。不同的编译器采用不同的名字修饰方法，因为导致由不同的编译器编译产生目标文件无法正常相互链接。

Maggot选择采用直接使用变量/函数名作为符号名的方式来生成符号，因此Maggot编译出来的汇编文件无法与使用clang编译出来的目标文件链接在一起

比如

```
int add(int a, int b)
{
    return a + b;
}
```

clang在编译这个函数的时候会将add的函数名修饰成 _Z3addii，而如果在源文件中使用该函数，并且使用maggot编译后，add的符号名还为add，此时如果使用链接器将两者链接在一起，会导致undefined reference symbol的错误。

```
[n7utb@Maka:~/perpetual-motion-machine/maggot/test/output]$ readelf -s ./symbol.o

Symbol table '.symtab' contains 4 entries:
   Num:    Value              Size Type    Bind   Vis      Ndx Name
   ---:    ---:              ---: ---:    ---:   ---:   ---:
    0: 0000000000000000         0 NOTYPE  LOCAL  DEFAULT UND
    1: 0000000000000000         0 FILE    LOCAL  DEFAULT ABS symbol.cpp
    2: 0000000000000000         0 SECTION LOCAL  DEFAULT 2
    3: 0000000000000000        18 FUNC    GLOBAL DEFAULT 2 _Z3addii
```

6.2 链接标准库

c++为了与C兼容，在符号管理上，C++可以使用extern “C” 的关键字来声明或者定义一个C的符号，继续上面的例子，使用extern “C” 之后使用clang++编译得到

```
[n7utb@Maka:~/perpetual-motion-machine/maggot/test/output]$ readelf -s ./symbol.o

Symbol table '.symtab' contains 4 entries:
   Num:    Value              Size Type    Bind   Vis      Ndx Name
   ---:    ---:              ---: ---:    ---:   ---:   ---:
    0: 0000000000000000         0 NOTYPE  LOCAL  DEFAULT UND
    1: 0000000000000000         0 FILE    LOCAL  DEFAULT ABS symbol.cpp
    2: 0000000000000000         0 SECTION LOCAL  DEFAULT 2
    3: 0000000000000000        18 FUNC    GLOBAL DEFAULT 2 add
```

在此基础上，Maggot允许通过编写库，通过自行编写的库使用标准库，再由源代码文件自动链接库，从而调用C标准库的相关函数，

比如在当前工程结构中，lib目录下提供了示例的库文件，该库文件通过extern “C” 关键字修饰，封装了几个C标准库的函数

```
#include <stdio.h>
extern "C"
{
    extern int printf() {
```

```

    fprintf(stdout, "MAGGOT\n");
    return 0;
}
extern int print_i(int i)
{
    fprintf(stdout, "the integer is %d\n", i);
    return 0;
}
extern int print_f(float f)
{
    fprintf(stdout, "the float is %f\n", f);
    return 0;
}
}

```

源代码文件可以在函数声明中, 声明需要使用到的库函数, 并在之后调用, maggot会在链接时自动将stdlib链接进

```

int print_i(int i);
int main() {
    int a = 2;
    print_i(a);
    return 0;
}

```

最终编译, 运行效果如下

```

al -I /lib64/libstdc++.so.6 -I /lib64/libstdc++.so.6
[n7utb@Maka:~/perpetual-motion-machine/maggot]$ ./maggot --binary -o ./test/output/output ./test/output/output.cpp
[n7utb@Maka:~/perpetual-motion-machine/maggot]$ ./test/output/output
the integer is 2

```

7 代码生成

7.1 中间代码生成

7.1.1 实现概述

Maggot在实现AST的基础上即可开始生成中间代码 (LLVM IR), 具体的方式是基于AST使用llvm的函数为每一个节点编写函数codeGen。基于Maggot的AST实现方式, 并且考虑到语法中涉及的语法属性均为综合属性, 因此可以用递归的方式由program根节点调用codeGen (实际是先序遍历), 最后完成整体的中间代码生成。

核心函数codeGen的返回值是LLVM Value类型的指针, 这个类型表示 SSA (Static Single Assignment静态一次赋值), 即类似于寄存器的机制, 一经赋值就不会再变化, 直至为其赋值的语句再次执行。

总之, Maggot自底向上地构建AST, 其后自上向下调用codeGen, 并递归返回从而生成中间代码。

7.1.2 实现细节

基于LLVM的规则, 首先定义了如下四个全局变量。

```

static LLVMContext TheContext;
static IRBuilder<> Builder(TheContext);
static std::unique_ptr<Module> TheModule = std::make_unique<Module>("maggot",
TheContext);
static std::map<std::string, Value *> NamedValues;

```

其中TheContext是用于记录LLVM核心结构的对象，例如类型与常量表。可以理解为程序环境。

Builder拥有许多便于生成中间代码的函数，例如运算、定义、赋值、流程控制等。

TheModule是用于储存函数与全局变量的LLVM结构，可以说它是LLVM用于储存代码的顶层结构，它会记录我们生成的全部代码。

NamedValues则是跟踪当前生存域的变量值的map结构。

下面叙述一些核心部分的codeGen的原理。

7.1.2.1 变量声明

```

Value *VarDeclList::CodeGen()
{
    if (isGlobal)
    {
        for (int i = 0; i < vars.size(); i++)
        {
            VarDeclAST *var = vars[i];
            Value *InitVal;
            // InitVal = ConstantFP::get(TheContext, APFloat(0.0));
            if (var->children.size())
            {
                InitVal = var->children[0]->CodeGen();
            }
            else
            {
                InitVal = getInitialValue(var->type);
            }
            // maintain global variable
            TheModule->getOrInsertGlobal(var->name, var->length == -1 ?
TypeGen(var->type) : TypeGen(var->type, var->length));
            GlobalVariable *gVar = TheModule->getNamedGlobal(var->name);
            // gVar->setAlignment(MaybeAlign(4));
            gVar->setInitializer(static_cast<Constant *>(InitVal));
        }
    }
    else
    {
        CompoundStmtAST *current = compoundstack.back();
        Function *TheFunction = Builder.GetInsertBlock()->getParent();
    }
}

```

```

    for (int i = 0; i < vars.size(); i++)
    {
        VarDeclAST *var = vars[i];
        Value *InitVal;
        debug("%s, %d", var->name.c_str(), var->type);
        if (var->children.size())
        {
            InitVal = var->children[0]->CodeGen();
        }
        else
        {
            InitVal = getInitialValue(var->type);
        }
        AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, var->name,
var->length == -1 ? TypeGen(var->type) : TypeGen(var->type, var->length));
        if (var->length == -1)
            Builder.CreateStore(InitVal, Alloca);
        current->OldBindings.push_back(make_pair(var->name, NamedValues[var-
>name]));
        NamedValues[var->name] = Alloca;
    }
}
return nullptr;
}

```

变量声明的语句，如 `int a, b = 1, c = b + 3;`，会首先判断是否是全局变量的声明。

如果是全局变量，则调用TheModule中对应的全局变量的声明方法，并根据是否有初始值来选择是否要以默认的初始值进行初始化。

如果是局部变量，则需要创造一个新的生存域（Block），并在这里生成变量。

同时，变量作用域的实现上，本工具并没有采用常见的symbol table的实现，设计上，采用一个compound 栈来规范变量的作用域，当每个compound开始时，将该compound的指针压入一个该compound栈中，每当变量声明时，

```

/***** statements *****/
Value *CompoundStmtAST::CodeGen()
{
    // Value *ret = nullptr;
    compoundstack.push_back(this);
    for (size_t i = 0; i < children.size(); i++)
    {
        children[i]->CodeGen();
    }
    for (auto &p : OldBindings)
    {
        NamedValues[p.first] = p.second;
    }
}

```

```

    compoundstack.pop_back();
    return nullptr;
}

```

取出compound栈中的栈顶对象,即该变量的作用域位于该compound的范围内,变量声明时,会将NamedValues中相同变量名的对象存放在oldBindings中暂存,在compound中所有子节点的codegen执行结束之后,compound内声明的变量的作用域结束,使用oldBinds中存放的变量重新放回NamedValues进入作用

另外变量声明中同样需要对数组类型和指针类型进行处理。

从上述的代码也可以发现, Maggot对于没有给予初始值的变量的初始化并不会因为是否是全局变量做不同的赋值。

7.1.2.2 函数定义

```

Function *FuncAST::CodeGen()
{
    // First, check for an existing function from a previous 'extern'
    declaration.
    Function *TheFunction = TheModule->getFunction(funcname);

    if (!TheFunction)
    {
        ParmVarDeclList *argslist = dynamic_cast<ParmVarDeclList *>(children[0]);
        size_t argssize = argslist->parms.size();

        std::vector<Type *> args;
        for (int i = 0; i < argssize; i++)
        {
            args.push_back(TypeGen(argslist->parms[i]->type));
        }
        FunctionType *FT =
            FunctionType::get(TypeGen(rettype), args, false);

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

        // Set names for all arguments.
        unsigned Idx = 0;
        for (auto &Arg : F->args())
        {
            Arg.setName(argslist->parms[Idx]->name);
            Idx++;
        }

        TheFunction = TheModule->getFunction(funcname);
    }

    if (!TheFunction->empty())

```

```

        return (Function *)LogErrorV("Function cannot be redefined.");

    // Create a new basic block to start insertion into.
    BasicBlock *BB = BasicBlock::Create(TheContext, funcname, TheFunction);
    Builder.SetInsertPoint(BB);

    // Record the function arguments in the NamedValues map.
    NamedValues.clear();
    ParmVarDeclList *argslist = dynamic_cast<ParmVarDeclList *>(children[0]);
    unsigned Idx = 0;
    for (auto &Arg : TheFunction->args())
    {
        AllocaInst *alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName(),
        Arg.getType());
        NamedValues[argslist->parms[Idx]->name] = alloca;
        Builder.CreateStore(&Arg, alloca);
        Idx++;
    }

    children[1]->CodeGen();

    // Finish off the function.
    verifyFunction(*TheFunction);

    return TheFunction;
}

```

首先会检查函数是否已被定义，如果重复定义（重名即认为已定义）则报错。

其次将检测是否已有函数声明，若无则补充函数声明的写入。

其后检测函数定义是否无函数体，若无则报错。

最后将新建一个基础块来插入函数体，并在NamedValues里记录函数的参数格式。然后递归调用函数体的codeGen从而完成全部的函数代码生成。

7.1.2.3 流程控制-条件语句

```

Value *SelectionStmtAST::CodeGen()
{
    // calculate condition value
    Value *cond = children[0]->CodeGen();

    if (!cond) // no condition codes, error
    {
        return LogErrorV("Need condition expression in if statement.");
    }

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

```



```

BasicBlock *thenBB = BasicBlock::Create(TheContext, "THEN", atFunc);
BasicBlock *elseBB = BasicBlock::Create(TheContext, "ELSE", atFunc);
BasicBlock *contBB = BasicBlock::Create(TheContext, "IFCONT", atFunc);
// conditionally branch
Builder.CreateCondBr(cond, thenBB, elseBB);

// then statements
Builder.SetInsertPoint(thenBB);
children[1]->CodeGen();
// branch to continue part
Builder.CreateBr(contBB);
Builder.SetInsertPoint(contBB);

// else generation
Builder.SetInsertPoint(elseBB);

// else statements
if (children.size() == 3)
{
    children[2]->CodeGen();
}
// branch to continue part
Builder.CreateBr(contBB);
// continue generation
Builder.SetInsertPoint(contBB);

return nullptr;
}

```

条件语句的生成的整体结构十分类似编写汇编代码的思路，先将核心的代码分割为三个部分，即条件、Then部分与Else部分。

在每段代码生成之前，需要创建label，在生成后需要创建branch实现条件跳转。

需要注意的是要判别是否有Else部分的代码。

循环语句的实现机制与此类似。

7.1.3 生成格式

Maggot中间代码生成的格式采用了LLVM IR格式，这一选择一方面是为了方便代码的实现，更方便与Clang等当下流行的主流编译器靠近。

这一格式为采用了Tripple-Address的汇编代码，可读性较强。

具体格式可在测试案例部分进行查看。

7.2 目标代码生成

LLVM中的平台无关代码生成器（Code Generator），同时也是一个编译器开发框架（Framework）。它提供了一些可复用的组件，帮助用户将LLVM IR编译到特定的平台上。

LLVM平台描述相关的类（Target Description Classes）为不同的平台提供了相同的抽象接口。这些类在设计上仅用来表达目标平台的属性，例如平台所支持的指令和寄存器，但不会保存任何和具体算法相关的描述。

简便起见，Maggot直接采用了系统调用`llvm-as`命令的方式生成最终代码，并指定架构为x86 64位架构。

是否要生成目标代码根据用户是否输入`bitcode`选项确定。

在此基础上，Maggot可以生成汇编代码以及可执行文件。

具体实现方法为,使用`system`函数调用`llvm-as`, `llc`, `clang++`等工具链进行编译

```
if (option_output_ir || option_output_bc || option_output_as ||
    option_output_bin)
{
    raw_fd_ostream file(option_output_ir ? file_output : "__temp.ll",
        errorcode);
    TheModule->print(file, nullptr);
}
if (option_output_bc || option_output_as || option_output_bin)
{
    std::system((std::string("llvm-as __temp.ll -o ") + (option_output_bc ?
        file_output : std::string("__temp.bc"))).c_str());
    std::system("rm __temp.ll");
}
if (option_output_as || option_output_bin)
{
    std::system((std::string("llc --march=x86-64 __temp.bc -o ") +
        (option_output_bc ? file_output : std::string("__temp.s"))).c_str());
    std::system("rm __temp.bc");
}
if (option_output_bin)
{
    std::system((std::string("clang++ __temp.s ./lib/libstd.o -o ") +
        file_output).c_str());
    std::system("rm __temp.s");
}
```

8 测试案例

8.1 综合测试

整数排序算法

```
int x[10], oct = 0x22, num = 222 - -233;
bool flag = false;
```

```
int print_i(int i);
int minloc(int low, int high);
int sort(int low, int high);
```

```
int minloc(int low, int high)
{
    int i, y, k;
    k = low;
    y = x[low];
    i = low + 1;
    while (i < high)
    {
        if (x[i] < y)
        {
            y = x[i];
            k = i;
        }
        i = i + 1;
    }
    return k;
}
```

```
// sort a list
int sort(int low, int high)
{
    // sort a list
    int i, k;
    i = low;
    while (i < high - 1)
    {
        int t;
        k = minloc(i, high);
        t = x[k];
        x[k] = x[i];
        x[i] = t;
        i = i + 1;
    }
    return 0;
}

int input()
{
    return 1;
}
```

```
int main()
{
    int i, m, n, a[2];
```

```

float f;
bool b;
f = 1e6;
b = true;
i = 0;
x[0] = 23;
x[1] = 44;
x[2] = 5;
x[3] = 9;
x[4] = 50;
x[5] = 14;
sort(0, 6);
while(i<6){
    print_i(x[i]);
    i = i + 1;
}
return 0;
}

```

生成的AST

```

`-DeclStmt
  `-VarDecl x global int *
  `-VarDecl oct global int
    `-IntegerLiteral 0
  `-VarDecl num global int
    `-BinaryOperator -
      `-IntegerLiteral 222
    `-UnaryOperator -
      `-IntegerLiteral 233
`-DeclStmt
  `-VarDecl flag global bool
    `-BoolLiteral false
`-FunctionDecl print_i int
  `-ParmVarDecl i int
`-FunctionDecl minloc int
  `-ParmVarDecl low int
  `-ParmVarDecl high int
`-FunctionDecl sort int
  `-ParmVarDecl low int
  `-ParmVarDecl high int
`-FunctionDecl minloc int
  `-ParmVarDecl low int
  `-ParmVarDecl high int
`-CompoundStmt
  `-DeclStmt
    `-VarDecl i local int
    `-VarDecl y local int

```

```

    ` -VarDecl k local  int
  ` -BinaryOperator  =
    ` -DeclRefExpr Var k
    ` -DeclRefExpr Var low
  ` -BinaryOperator  =
    ` -DeclRefExpr Var y
    ` -ArraySubscriptExpr
      ` -DeclRefExpr Var x
      ` -DeclRefExpr Var low
  ` -BinaryOperator  =
    ` -DeclRefExpr Var i
    ` -BinaryOperator  +
      ` -DeclRefExpr Var low
      ` -IntegerLiteral 1
  ` -WhileStmt
    ` -BinaryOperator  <
      ` -DeclRefExpr Var i
      ` -DeclRefExpr Var high
    ` -CompoundStmt
      ` -IfStmt
        ` -BinaryOperator  <
          ` -ArraySubscriptExpr
            ` -DeclRefExpr Var x
            ` -DeclRefExpr Var i
          ` -DeclRefExpr Var y
        ` -CompoundStmt
          ` -BinaryOperator  =
            ` -DeclRefExpr Var y
            ` -ArraySubscriptExpr
              ` -DeclRefExpr Var x
              ` -DeclRefExpr Var i
          ` -BinaryOperator  =
            ` -DeclRefExpr Var k
            ` -DeclRefExpr Var i
        ` -BinaryOperator  =
          ` -DeclRefExpr Var i
          ` -BinaryOperator  +
            ` -DeclRefExpr Var i
            ` -IntegerLiteral 1
      ` -ReturnStmt
        ` -DeclRefExpr Var k
  ` -FunctionDecl sort  int
  ` -ParmVarDecl low  int
  ` -ParmVarDecl high int
  ` -CompoundStmt
    ` -DeclStmt
      ` -VarDecl i local  int
      ` -VarDecl k local  int

```

```

    `-BinaryOperator  =
        `-DeclRefExpr Var i
        `-DeclRefExpr Var low
    `-WhileStmt
        `-BinaryOperator  <
            `-DeclRefExpr Var i
            `-BinaryOperator  -
                `-DeclRefExpr Var high
                `-IntegerLiteral  1
    `-CompoundStmt
        `-DeclStmt
            `-VarDecl t local  int
        `-BinaryOperator  =
            `-DeclRefExpr Var k
            `-CallExpr
                `-DeclRefExpr Func minloc
                `-DeclRefExpr Var i
                `-DeclRefExpr Var high
        `-BinaryOperator  =
            `-DeclRefExpr Var t
            `-ArraySubscriptExpr
                `-DeclRefExpr Var x
                `-DeclRefExpr Var k
        `-BinaryOperator  =
            `-ArraySubscriptExpr
                `-DeclRefExpr Var x
                `-DeclRefExpr Var k
            `-ArraySubscriptExpr
                `-DeclRefExpr Var x
                `-DeclRefExpr Var i
        `-BinaryOperator  =
            `-ArraySubscriptExpr
                `-DeclRefExpr Var x
                `-DeclRefExpr Var i
            `-DeclRefExpr Var t
        `-BinaryOperator  =
            `-DeclRefExpr Var i
            `-BinaryOperator  +
                `-DeclRefExpr Var i
                `-IntegerLiteral  1
    `-ReturnStmt
        `-IntegerLiteral  0
`-FunctionDecl input  int
    `-CompoundStmt
        `-ReturnStmt
            `-IntegerLiteral  1
`-FunctionDecl main  int
    `-CompoundStmt

```

```
`-DeclStmt
  `-VarDecl i local  int
  `-VarDecl m local  int
  `-VarDecl n local  int
  `-VarDecl a local  int *
`-DeclStmt
  `-VarDecl f local  float
`-DeclStmt
  `-VarDecl b local  bool
`-BinaryOperator =
  `-DeclRefExpr Var f
  `-FloatLiteral 1e+06
`-BinaryOperator =
  `-DeclRefExpr Var b
  `-BoolLiteral  true
`-BinaryOperator =
  `-DeclRefExpr Var i
  `-IntegerLiteral 0
`-BinaryOperator =
  `-ArraySubscriptExpr
    `-DeclRefExpr Var x
    `-IntegerLiteral 0
  `-IntegerLiteral 23
`-BinaryOperator =
  `-ArraySubscriptExpr
    `-DeclRefExpr Var x
    `-IntegerLiteral 1
  `-IntegerLiteral 44
`-BinaryOperator =
  `-ArraySubscriptExpr
    `-DeclRefExpr Var x
    `-IntegerLiteral 2
  `-IntegerLiteral 5
`-BinaryOperator =
  `-ArraySubscriptExpr
    `-DeclRefExpr Var x
    `-IntegerLiteral 3
  `-IntegerLiteral 9
`-BinaryOperator =
  `-ArraySubscriptExpr
    `-DeclRefExpr Var x
    `-IntegerLiteral 4
  `-IntegerLiteral 50
`-BinaryOperator =
  `-ArraySubscriptExpr
    `-DeclRefExpr Var x
    `-IntegerLiteral 5
  `-IntegerLiteral 14
```

```

`-CallExpr
  `-DeclRefExpr Func sort
  `-IntegerLiteral 0
  `-IntegerLiteral 6
`-WhileStmt
  `-BinaryOperator <
    `-DeclRefExpr Var i
    `-IntegerLiteral 6
  `-CompoundStmt
    `-CallExpr
      `-DeclRefExpr Func print_i
      `-ArraySubscriptExpr
        `-DeclRefExpr Var x
        `-DeclRefExpr Var i
      `-BinaryOperator =
        `-DeclRefExpr Var i
        `-BinaryOperator +
          `-DeclRefExpr Var i
          `-IntegerLiteral 1
    `-ReturnStmt
      `-IntegerLiteral 0

```

生成的IR中间代码

```

source_filename = "./test/sample.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@x = global [10 x i32] zeroinitializer
@oct = global i32 0
@num = global i32 455
@flag = global i1 false

declare i32 @print_i(i32)

define i32 @minloc(i32 %low, i32 %high) {
minloc:
  %k = alloca i32
  %y = alloca i32
  %i = alloca i32
  %high2 = alloca i32
  %low1 = alloca i32
  store i32 %low, i32* %low1
  store i32 %high, i32* %high2
  store i32 0, i32* %i
  store i32 0, i32* %y
  store i32 0, i32* %k

```



```

%low3 = load i32, i32* %low1
store i32 %low3, i32* %k
%low4 = load i32, i32* %low1
%0 = getelementptr inbounds [10 x i32], [10 x i32]* @x, i32 0, i32 %low4
%1 = load i32, i32* %0
store i32 %1, i32* %y
%low5 = load i32, i32* %low1
%addtmp = add nsw i32 %low5, 1
store i32 %addtmp, i32* %i
br label %COND

COND:
; preds = %IFCONT, %minloc
%i6 = load i32, i32* %i
%high7 = load i32, i32* %high2
%cmplt = icmp slt i32 %i6, %high7
br i1 %cmplt, label %LOOP, label %ENDLOOP

LOOP:
; preds = %COND
%i8 = load i32, i32* %i
%2 = getelementptr inbounds [10 x i32], [10 x i32]* @x, i32 0, i32 %i8
%3 = load i32, i32* %2
%y9 = load i32, i32* %y
%cmplt10 = icmp slt i32 %3, %y9
br i1 %cmplt10, label %THEN, label %ELSE

ENDLOOP:
; preds = %COND
%k15 = load i32, i32* %k
ret i32 %k15

THEN:
; preds = %LOOP
%i11 = load i32, i32* %i
%4 = getelementptr inbounds [10 x i32], [10 x i32]* @x, i32 0, i32 %i11
%5 = load i32, i32* %4
store i32 %5, i32* %y
%i12 = load i32, i32* %i
store i32 %i12, i32* %k
br label %IFCONT

ELSE:
; preds = %LOOP
br label %IFCONT

IFCONT:
; preds = %ELSE, %THEN
%i13 = load i32, i32* %i
%addtmp14 = add nsw i32 %i13, 1
store i32 %addtmp14, i32* %i
br label %COND
}

```

```

define i32 @sort(i32 %low, i32 %high) {
sort:
    %t = alloca i32
    %k = alloca i32
    %i = alloca i32
    %high2 = alloca i32
    %low1 = alloca i32
    store i32 %low, i32* %low1
    store i32 %high, i32* %high2
    store i32 0, i32* %i
    store i32 0, i32* %k
    %low3 = load i32, i32* %low1
    store i32 %low3, i32* %i
    br label %COND

COND:                                     ; preds = %LOOP, %sort
    %i4 = load i32, i32* %i
    %high5 = load i32, i32* %high2
    %subtmp = sub nsw i32 %high5, 1
    %cmplt = icmp slt i32 %i4, %subtmp
    br i1 %cmplt, label %LOOP, label %ENDLOOP

LOOP:                                     ; preds = %COND
    store i32 0, i32* %t
    %i6 = load i32, i32* %i
    %high7 = load i32, i32* %high2
    %calltmp = call i32 @minloc(i32 %i6, i32 %high7)
    store i32 %calltmp, i32* %k
    %k8 = load i32, i32* %k
    %0 = getelementptr inbounds [10 x i32], [10 x i32]* @x, i32 0, i32 %k8
    %1 = load i32, i32* %0
    store i32 %1, i32* %t
    %k9 = load i32, i32* %k
    %2 = getelementptr inbounds [10 x i32], [10 x i32]* @x, i32 0, i32 %k9
    %i10 = load i32, i32* %i
    %3 = getelementptr inbounds [10 x i32], [10 x i32]* @x, i32 0, i32 %i10
    %4 = load i32, i32* %3
    store i32 %4, i32* %2
    %i11 = load i32, i32* %i
    %5 = getelementptr inbounds [10 x i32], [10 x i32]* @x, i32 0, i32 %i11
    %t12 = load i32, i32* %t
    store i32 %t12, i32* %5
    %i13 = load i32, i32* %i
    %addtmp = add nsw i32 %i13, 1
    store i32 %addtmp, i32* %i
    br label %COND

ENDLOOP:                                 ; preds = %COND

```

```

    ret i32 0
}

define i32 @input() {
input:
    ret i32 1
}

define i32 @main() {
main:
    %b = alloca i1
    %f = alloca float
    %a = alloca [2 x i32]
    %n = alloca i32
    %m = alloca i32
    %i = alloca i32
    store i32 0, i32* %i
    store i32 0, i32* %m
    store i32 0, i32* %n
    store float 0.000000e+00, float* %f
    store i1 false, i1* %b
    store float 1.000000e+06, float* %f
    store i1 true, i1* %b
    store i32 0, i32* %i
    store i32 23, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @x, i32 0,
i32 0)
    store i32 44, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @x, i32 0,
i32 1)
    store i32 5, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @x, i32 0,
i32 2)
    store i32 9, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @x, i32 0,
i32 3)
    store i32 50, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @x, i32 0,
i32 4)
    store i32 14, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @x, i32 0,
i32 5)
    %calltmp = call i32 @sort(i32 0, i32 6)
    br label %COND

COND:
; preds = %LOOP, %main
    %i1 = load i32, i32* %i
    %cmp1t = icmp slt i32 %i1, 6
    br i1 %cmp1t, label %LOOP, label %ENDLOOP

LOOP:
; preds = %COND
    %i2 = load i32, i32* %i
    %0 = getelementptr inbounds [10 x i32], [10 x i32]* @x, i32 0, i32 %i2
    %1 = load i32, i32* %0

```

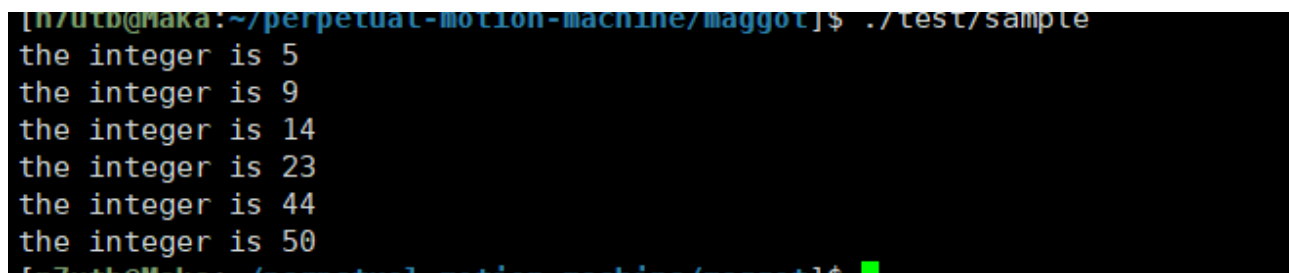
```

    %calltmp3 = call i32 @print_i(i32 %1)
    %i4 = load i32, i32* %i
    %addtmp = add nsw i32 %i4, 1
    store i32 %addtmp, i32* %i
    br label %COND

ENDLOOP:                                ; preds = %COND
    ret i32 0
}

```

运行结果



```

[11/01/2020@Maka:~/perpetual-motion-machine/maggot]$ ./test/sample
the integer is 5
the integer is 9
the integer is 14
the integer is 23
the integer is 44
the integer is 50
[11/01/2020@Maka:~/perpetual-motion-machine/maggot]$

```

排序结果正确

8.2 数组测试

```

int k[10];
int print_f(float f);
int main(void)
{
    float j[10];
    j[0] = 2.0;
    j[1] = 6.0 / 10.0;
    print_f(j[0]);
    print_f(j[1]);
    return 0;
}

```

生成的AST

```

`-DeclStmt
  `~VarDecl k global int *
`-FunctionDecl print_f int
  `~ParmVarDecl f float
`-FunctionDecl main int
  `~CompoundStmt
    `~DeclStmt
      `~VarDecl j local float *
    `~BinaryOperator =
      `~ArraySubscriptExpr

```

```

    `-DeclRefExpr Var j
    `-IntegerLiteral 0
    `-FloatLiteral 2
    `-BinaryOperator =
    `-ArraySubscriptExpr
    `-DeclRefExpr Var j
    `-IntegerLiteral 1
    `-BinaryOperator /
    `-FloatLiteral 6
    `-FloatLiteral 10
    `-CallExpr
    `-DeclRefExpr Func print_f
    `-ArraySubscriptExpr
    `-DeclRefExpr Var j
    `-IntegerLiteral 0
    `-CallExpr
    `-DeclRefExpr Func print_f
    `-ArraySubscriptExpr
    `-DeclRefExpr Var j
    `-IntegerLiteral 1
    `-ReturnStmt
    `-IntegerLiteral 0

```

生成的IR

```

; ModuleID = 'maggot'
source_filename = "./test/array.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@k = global [10 x i32] zeroinitializer

declare i32 @print_f(float)

define i32 @main() {
main:
    %j = alloca [10 x float]
    %0 = getelementptr inbounds [10 x float], [10 x float]* %j, i32 0, i32 0
    store float 2.000000e+00, float* %0
    %1 = getelementptr inbounds [10 x float], [10 x float]* %j, i32 0, i32 1
    store float 0x3FE3333340000000, float* %1
    %2 = getelementptr inbounds [10 x float], [10 x float]* %j, i32 0, i32 0
    %3 = load float, float* %2
    %calltmp = call i32 @print_f(float %3)
    %4 = getelementptr inbounds [10 x float], [10 x float]* %j, i32 0, i32 1
    %5 = load float, float* %4
    %calltmp1 = call i32 @print_f(float %5)

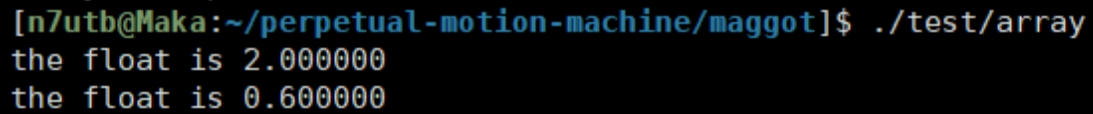
```

```

    ret i32 0
    ret i32 0
}

```

代码运行结果



```

[n7utb@Maka:~/perpetual-motion-machine/maggot]$ ./test/array
the float is 2.000000
the float is 0.600000

```

8.3 链接库文件

```

int print_i(int i);
int main(){
    int a = 2;
    print_i(a);
    return 0;
}

```

生成的AST

```

`-FunctionDecl print_i int
  `-ParmVarDecl i int
`-FunctionDecl main int
  `-CompoundStmt
    `-DeclStmt
      `-VarDecl a local int
        `-IntegerLiteral 2
    `-CallExpr
      `-DeclRefExpr Func print_i
      `-DeclRefExpr Var a
    `-ReturnStmt
      `-IntegerLiteral 0

```

生成的IR

```

; ModuleID = 'maggot'
source_filename = "./test/output/output.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

declare i32 @print_i(i32)

define i32 @main() {
main:
    %a = alloca i32
    store i32 2, i32* %a

```

```

    %a1 = load i32, i32* %a
    %calltmp = call i32 @print_i(i32 %a1)
    ret i32 0
    ret i32 0
}

```

运行效果

```

[n7utb@Maka:~/perpetual-motion-machine/maggot]$ ./test/output/output
the integer is 2

```

8.4 if-while嵌套

源代码:

```

int print_i(int i);
int main()
{
    int a = 1;
    if (a <= 1) {
        while(a <= 5) {
            print_i(a);
            a = a + 1;
        }
    }
    return 0;
}

```

AST:

```

`-FunctionDecl print_i int
  `-ParmVarDecl i int
`-FunctionDecl main int
  `-CompoundStmt
    `-DeclStmt
      `-VarDecl a local int
        `-IntegerLiteral 1
    `-IfStmt
      `-BinaryOperator <=
        `-DeclRefExpr Var a
        `-IntegerLiteral 1
    `-CompoundStmt
      `-WhileStmt
        `-BinaryOperator <=
          `-DeclRefExpr Var a
          `-IntegerLiteral 5
        `-CompoundStmt
          `-CallExpr

```

```

    `-DeclRefExpr Func print_i
    `-DeclRefExpr Var a
    `-BinaryOperator =
    `-DeclRefExpr Var a
    `-BinaryOperator +
    `-DeclRefExpr Var a
    `-IntegerLiteral 1
    `~ReturnStmt
    `~IntegerLiteral 0

```

LLVM IR:

```

; ModuleID = 'maggot'
source_filename = "./test/uminus.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

declare i32 @print_i(i32)

define i32 @main() {
main:
    %a = alloca i32
    store i32 1, i32* %a
    %a1 = load i32, i32* %a
    %cmp1tmp = icmp sle i32 %a1, 1
    br i1 %cmp1tmp, label %THEN, label %ELSE

THEN:                                     ; preds = %main
    br label %COND

ELSE:                                     ; preds = %main
    br label %IFCONT

IFCONT:                                  ; preds = %ELSE, %ENDLOOP
    ret i32 0

COND:                                     ; preds = %LOOP, %THEN
    %a2 = load i32, i32* %a
    %cmp2tmp3 = icmp sle i32 %a2, 5
    br i1 %cmp2tmp3, label %LOOP, label %ENDLOOP

LOOP:                                    ; preds = %COND
    %a4 = load i32, i32* %a
    %call1tmp = call i32 @print_i(i32 %a4)
    %a5 = load i32, i32* %a
    %addtmp = add nsw i32 %a5, 1
    store i32 %addtmp, i32* %a

```



```

    br label %COND

ENDLOOP:                                ; preds = %COND
    br label %IFCONT
}

```

输出效果:

```

[n7utb@Maka:~/perpetual-motion-machine/maggot]$ ./test/uminus
the integer is 1
the integer is 2
the integer is 3
the integer is 4
the integer is 5

```

8.5 数组传参

实现可以将数组传给函数调用

```

int print(char s[]);
char y[10]= "aaa";
int main(void)
{
    char x[10]="bbb";
    print(y);
    print(x);
    return 0;
}

```

输出的AST如下

```

AST:
`-FunctionDecl print int
  `-ParmVarDecl s char *
`-DeclStmt
  `-VarDecl y global char *
    `-StringLiteral aaa
`-FunctionDecl main int
  `-CompoundStmt
    `-DeclStmt
      `-VarDecl x local char *
        `-StringLiteral bbb
    `-CallExpr
      `-DeclRefExpr Func print
      `-DeclRefExpr Var y
    `-CallExpr
      `-DeclRefExpr Func print
      `-DeclRefExpr Var x

```

```
`-ReturnStmt
  `-IntegerLiteral 0
```

输出的IR

```
; ModuleID = 'maggot'
source_filename = "./test/array.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@y = global [10 x i8] c"aaa\00\00\00\00\00\00\00"
@0 = private unnamed_addr constant [4 x i8] c"bbb\00", align 1

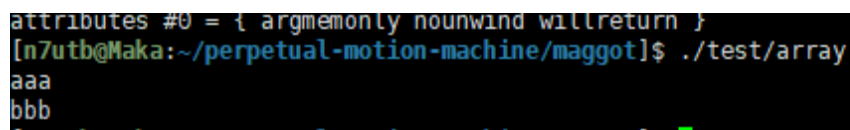
declare i32 @print(i8*)

define i32 @main() {
main:
    %x = alloca [10 x i8]
    %0 = bitcast [10 x i8]* %x to i8*
    call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 1 %0, i8* align 1 getelementptr
inbounds ([4 x i8], [4 x i8]* @0, i64 0, i64 0), i64 10, i1 false)
    %calltmp = call i32 @print(i8* getelementptr inbounds ([10 x i8], [10 x i8]*
@y, i64 0, i64 0))
    %1 = getelementptr inbounds [10 x i8], [10 x i8]* %x, i64 0, i64 0
    %calltmp1 = call i32 @print(i8* %1)
    ret i32 0
}

; Function Attrs: argmemonly nounwind willreturn
declare void @llvm.memcpy.p0i8.p0i8.i64(i8* noalias nocapture writeonly, i8*
noalias nocapture readonly, i64, i1 immarg) #0

attributes #0 = { argmemonly nounwind willreturn }
```

最后函数执行结果



```
attributes #0 = { argmemonly nounwind willreturn }
[n7utb@Maka:~/perpetual-motion-machine/maggot]$ ./test/array
aaa
bbb
```

9 分工情况

姓 名	主 要 工 作
徐梓凯	语法，AST，中间代码，目标代码
张沛全	词法，语法，AST，中间代码
包德政	AST，中间代码，最终代码

10 参考资料

1. LLVM Tutorial
2. LLVM Language Reference Manual