浙江大学

本科实验报告

课程名称	编译原理
姓 名	曹田雨，高珂，许恩宁
学 院	计算机学院
系	计算机系
专 业	计算机科学与技术
学 号	3200105453， 3200105111，3200102872
指导教师	王强

序言

0.1 概述

我们所实现的编译器是三个大组件组成，数据以pipeline的方式从一个到另一个。我们使用不同的工具来帮助我们建立这些组件中的每一个。下面是每个步骤和我们将使用的工具的图示：

我们**使用Flex进行词法分析**：将输入数据分割成一组tokens（标识符、关键词、数字、大括号、小括号等）。**用Bison进行语义分析**：在解析标记的同时生成一个AST。Bison将在这里做大部分的工作，我们只需要定义我们的AST。**用LLVM汇编**： 我们遍历AST的每个结点，并为每个结点生成机器码。

0.2 文件说明

```
.
├── CMakeLists.txt              # 通过词法分析和语法分析，构建生成Compiler
├── README.md
├── documents                   # report
│   ├── fig
│   │   ├── ...
│   └── report.md
├── parser_for_visualization    # 可视化语法树
├── Compiler                    # 生成中间代码
├── src
│   ├── CodeGen.cpp
│   ├── CodeGen.h
│   ├── Makefile                # 进行词法分析和语法分析
│   ├── main.cpp
│   ├── node.cpp
│   ├── node.h
│   ├── parser.y
│   └── token.l
├── test                        # 测试代码集
│   ├── fib
│   │   ├── fib                 # 可执行文件
│   │   ├── fib.bc
│   │   ├── fib.c
│   │   ├── fib.ll              # 中间代码
│   │   ├── fib.o
│   │   ├── fib.s               # 机器码
│   │   └── run.sh              # 由中间代码生成可执行文件
│   ├── quicksort
│   │   ├── ...
│   ├── float
│   │   ├── ...
│   └── while
│       ├── ...
└── visualization               # 可视化语法树
    ├── AST.json
    └── json2tree
        ├── css
        │   ├── ...
        ├── img
        │   ├── ...
        ├── js
        │   ├── ...
        ├── json2tree.html      # 可视化
```

```
        └── webfonts
            ├── ...
```

0.3 分工

成员	分工
曹田雨	词法分析，语法分析，整体调试与案例测试
高珂	语义分析，代码生成(LLVM)，整体调试与案例测试
许恩宁	可视化语法树

1 词法分析 Lex

1.1 正规表达式

```
alpha       [a-zA-Z_]
digits      [0-9]
alnum       [a-zA-Z0-9_]

%%
[ \t\n]                                                 ;
{alpha}({alnum}*)                                       { SAVE_TOKEN; return
IDENTIFIER;}
({digits}+"."{digits}*)|({digits}*"."{digits}+)         { SAVE_TOKEN; return TFLOAT;}
[1-9]{digits}*                                          { SAVE_TOKEN; return TINTEGER;}
    // matches either a single character enclosed in single quotes or an escape
sequence enclosed in single quotes
\'.\'|\'\\.\'                                           { SAVE_TOKEN; return TCHAR;}
\"(\\.|[^"\\])*\"                                       { SAVE_TOKEN; return TSTRING;}

"&&"            { return TOKEN(AND); }
"||"            { return TOKEN(OR); }
"<"             { return TOKEN(LE); }
">"             { return TOKEN(GT); }
"<="            { return TOKEN(LEQ); }
">="            { return TOKEN(GEQ); }
"=="            { return TOKEN(EQU); }
"!="            { return TOKEN(NEQ); }
";"             { return TOKEN(';'); }
"{"             { return TOKEN('{'); }
"}"             { return TOKEN('}'); }
","             { return TOKEN(','); }
":"             { return TOKEN(':'); }
"="             { return TOKEN('='); }
```

```
"("             { return TOKEN('('); }
")"             { return TOKEN(')'); }
"["             { return TOKEN('['); }
"]"             { return TOKEN(']'); }
"."             { return TOKEN('.'); }
"&"             { return TOKEN(GAD); }
"!"             { return TOKEN(NOT); }
"~"             { return TOKEN('~'); }
"-"             { return TOKEN(MINUS); }
"+"             { return TOKEN(PLUS); }
"*"             { return TOKEN(MUL); }
"/"             { return TOKEN(DIV); }
"%"             { return TOKEN('%'); }
"^"             { return TOKEN('^'); }
"|"             { return TOKEN('|'); }
"?"             { return TOKEN('?'); }

"if"            { return TOKEN(IF); }
"else"          { return TOKEN(ELSE); }
"while"         { return TOKEN(WHILE); }
"break"         { return TOKEN(BREAK); }
"return"        { return TOKEN(RETURN); }

.               { printf("Unknown token: %s in line: %d\n", yytext, yylineno); }

%%
```

1.2 实现原理和方法

1.2.1 Flex

- Flex（Flexible Scanner）是一种快速词法分析器生成器，用于生成用于解析文本的词法分析器。词法分析器是编译器中的一个组件，负责将输入文本分解为标记（tokens），以便后续的语法分析。

  Flex使用基于正则表达式的模式匹配技术，将输入文本分解成标记序列。它生成的词法分析器是C或C++语言的源代码，可以与编译器的其他组件集成。

- Lex输入文件 (a.l) 格式

  ```
  {definitions}
  %%
  {rules}
  %%
  {auxiliary routines}
  ```

1.2.2 语言：类C

我们实现一种类C语言，实现了C语言的功能。我们支持的tokens有：

```
int, float, char, string
+, -, *, /, ==, >=, <=, <, >, !=, !, &(取地址), =
||, &&
array, func
if, else, while, break, return
```

2 语法分析 Yacc (Bison)

2.1 上下文无关文法 Context-free Language

```
program:
    statements
    ;

statements:
    stmt
    | statements stmt
    ;

block:
    '{' statements '}'
    | '{' '}'
    ;

stmt:
    var_decl ';'
    | func_decl
    | expr ';'
    | IF '(' expr ')' block
    | IF '(' expr ')' block ELSE block
    | WHILE '(' expr ')' block
    | BREAK ';'
    | RETURN ';'
    | RETURN expr ';'
    ;

var_decl:
    ident ident  // int a;
    | ident ident '=' expr  // int a = b + 2;
    | ident ident '[' TINTEGER ']'  // int a[10]
    ;
```

```
func_decl:
    ident ident '(' func_decl_args ')' block  // int f(args){}
    ;

func_decl_args:
    /*no args*/  // no args
    | var_decl  // int a
    | func_decl_args ',' var_decl  // int a, int b
    ;

ident:
    IDENTIFIER
    ;

const_value:
    TINTEGER
    | TFLOAT
    | TCHAR
    | TSTRING ;
    ;

expr:
    const_value // 12
    | expr PLUS expr
    | expr MINUS expr
    | expr MUL expr
    | expr DIV expr
    | expr EQU expr
    | expr LEQ expr
    | expr GEQ expr
    | expr NEQ expr
    | expr LE expr
    | expr GT expr
    | expr AND expr
    | expr OR expr
    | '(' expr ')'  // (a+b)
    | ident '=' expr  // a = a + 2
    | ident '(' ')'
    | ident '(' call_args ')'  // f(a, b)
    | ident  // a
    | ident '[' expr ']'
    | ident '[' expr ']' '=' expr
    | GAD ident
    | GAD ident '[' expr ']'
    ;

call_args:
    /*no args*/
    | expr
```

```
    | call_args ',' expr
    ;
```

2.2 实现原理和方法

2.2.1 Yacc

Yacc（Yet Another Compiler Compiler）是一个用于生成语法分析器的工具，用于解析和处理上下文无关文法（Context-Free Grammar）。Yacc接受一个文法规范作为输入，并生成用于解析该文法的语法分析器代码。语法分析器用于将输入的符号序列（通常是由词法分析器生成的标记序列）转换为语法结构，例如抽象语法树（Abstract Syntax Tree）。

2.2.2 Bison

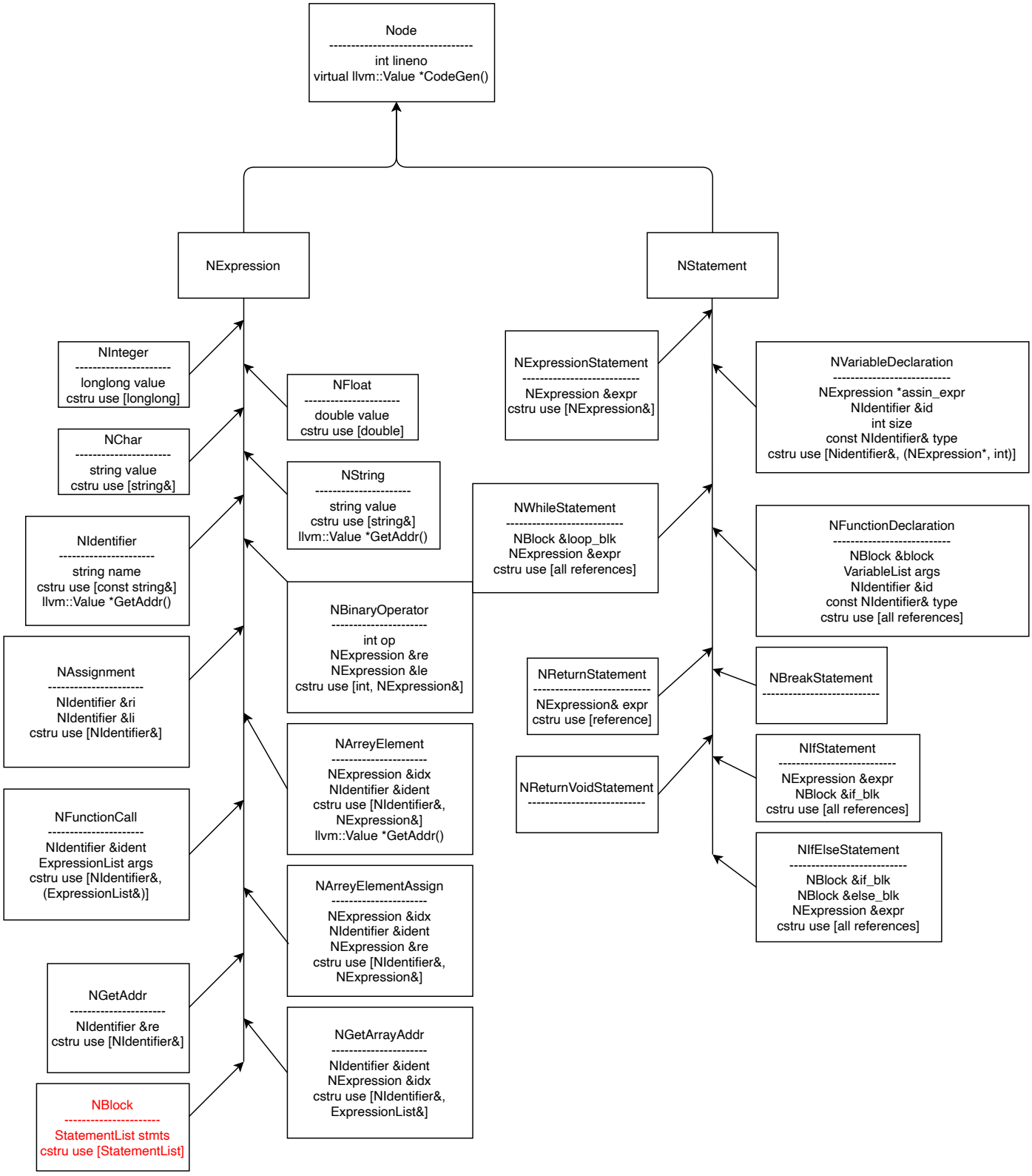
Bison是GNU计划中的一个工具，用于生成语法分析器（parser）代码。它是在Yacc（Yet Another Compiler Compiler）的基础上开发的，并提供了一些额外的功能和改进。

Bison接受一个文法规范作为输入，并生成用于解析该文法的语法分析器代码。类似于Yacc，Bison也使用LALR（Look-Ahead Left-to-Right Rightmost derivation）算法进行语法分析。与Flex类似，Bison的输入文件也由3段组成，用%%分隔：

```
{definitions}
%%
{rules}
%%
{user's code}
```

2.2.3 AST

Flex 和 Bison 将输入的程序文本解析为抽象语法树 (AST, Abstract Syntax Tree)的形式。以下是我们定义的语法树的结点类型：

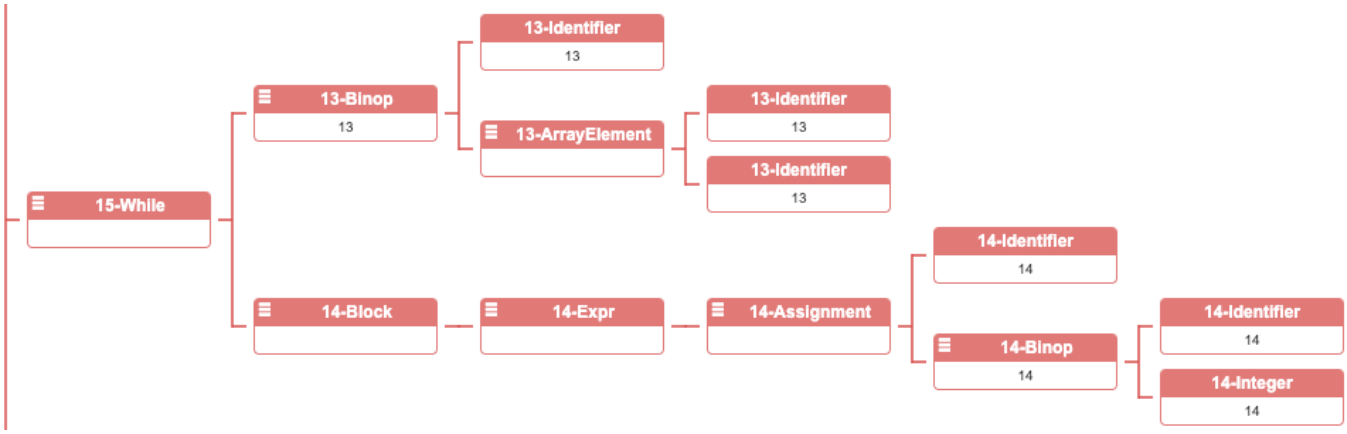


2.2.4 AST抽象语法树可视化

AST可视化以Node为单位，分别将AST的每个节点输出为JSON格式，最后将JSON文件`AST.json`传入可视化文档`json2tree.html`，实现如下图所示的效果。

```
var datascource = {
      "name": "Node's name",
```

```
    "value": "Node's value",
    "children": [
      { "name": "Child-1's name", "value": "Child-1's value",
        "children": [
          ...
        ]
      },
      { "name": "Child-2's name", "value": "Child-2's value" },
      { "name": "Child-3's name", "value": "Child-3's value",
        "children": [
          ...
        ]
      }
    ]
  };
```



3 语义分析

LLVM是构架编译器(compiler)的框架系统，以C++编写而成，提供将抽象语法树节点转化成中间代码的功能；同时，llvm提供系统方法，将中间代码转化成目标机器代码。

语义分析主要完成利用llvm函数，将AST节点转化成llvm中间代码的工作。

框架分析：

与语义相关的部分主要有文件node.cpp, CodeGen.h, CodeGen.cpp。node.cpp文件主要实现每个Node节点中对应的**CodeGen(CodeGenContext &CodeGenContext)**方法，为每种不同种类的节点实现**生成LLVMIR中间代码**；

首先在CodeGen.cpp中定义全局文本CodeGenContext，存储全局状态如module，function，return标记等，用于记录语义分析状态，决定接下来进行的操作。

```
class CodeGenContext{
public:
    vector<symbolTable *> symbolTable_stack; //符号栈

public:
```

```
    llvm::Module *myModule;
    llvm::Function *printf,*scanf, *gets;
    llvm::Function* currentFunc;
    llvm::BasicBlock* returnBB;
    llvm::Value* returnVal;
    bool isArgs;
    bool hasReturn;

    CodeGenContext();
```

接下来定义全局变量llvm::myContext及llvm::IRbuilder，用于中间代码的转换。

```
extern llvm::LLVMContext myContext; //定义全局context
extern llvm::IRBuilder<> myBuilder; //定义全局IRbuilder
```

同时利用一个**symbolTable类**作为符号表，存储块内的变量表 local_var 及对应的变量-llvm 类型表 local_var_type；

```
class symbolTable{
public:
    map<string, llvm::Value*> local_var; //局部变量 map
    map<string, llvm::Type*> local_var_type;//局部变量 string-llvm::type 的 map
};
```

接着我们在node.cpp中实现node.h中抽象语法单元要求的CodeGen模块。我们的主要操作有类型声明和赋值、取地址、生成和调用函数、控制流，以其中典型为例剖析。

1. getAddr：

```
llvm::Value* NArrayElement::GetAddr(CodeGenContext &CodeGenContext){
    cout<<"get arrayElement Addr:"<<ident.name<<"[]"<<endl;
    llvm::Value* arrayValue = CodeGenContext.findVariable(ident.name);
    if(arrayValue == nullptr){
        cerr << "undeclared array " << ident.name << endl;
        return nullptr;
    }
    //利用虚函数获得返回值
    llvm::Value* indexValue = idx.CodeGen(CodeGenContext);//执行NExpression获得id
    vector<llvm::Value*> indexList;
    // 如果是一个指针
    if(arrayValue->getType()->getPointerElementType()->isPointerTy()) {//指针也具有[]方法：取得指向的地址+idx
        arrayValue = myBuilder.CreateLoad(arrayValue->getType()->getPointerElementType(), arrayValue);
        indexList.push_back(indexValue);
    }
    // 如果是一个数组
    else {
```

```
        indexList.push_back(myBuilder.getInt32(0));//数组基地址
        indexList.push_back(indexValue);
    }
    llvm::Value* elePtr =  myBuilder.CreateInBoundsGEP(arrayValue,
llvm::ArrayRef<llvm::Value*>(indexList), "elePtr");
    return elePtr;
}
```

2. get variable: 从符号表中获得我们定义的变量（指针类型）

```
llvm::Value* NIdentifier::GetAddr(CodeGenContext &CodeGenContext){
    cout << "IdentifierNode : " << name << endl;

    llvm::Value* variable = CodeGenContext.findVariable(name);//从符号表中找到存储的变量（同
时能够得知变量的类型）
    if(variable == nullptr){
        std::cerr << "undeclared variable " << name << endl;
        return nullptr;
    }
    return variable;
}
```

3. function declaration

```
llvm::Value* NFunctionDeclaration::CodeGen(CodeGenContext &CodeGenContext){
    vector<llvm::Type*> argTypes;
    for(auto it : args){
        if(it->size == 0)//一般类型
            argTypes.push_back(getLLvmType(it->type.name));
        else {//数组
            argTypes.push_back(getPtrLLvmType(it->type.name));
        }
    }
    llvm::FunctionType *ftype = llvm::FunctionType::get(getLLvmType(type.name),
makeArrayRef(argTypes), false);
    llvm::Function *function = llvm::Function::Create(ftype,
llvm::GlobalValue::ExternalLinkage, id.name.c_str(), CodeGenContext.myModule);
    llvm::BasicBlock *bblock = llvm::BasicBlock::Create(myContext, "entry", function,
0);

    myBuilder.SetInsertPoint(bblock);
    CodeGenContext.currentFunc = function;
    CodeGenContext.returnBB = llvm::BasicBlock::Create(myContext, "return", function,
0);

    // 定义一个变量用来存储函数的返回值
    if(type.name.compare("void") != 0) {
```

```
        CodeGenContext.returnVal = new llvm::AllocaInst(getLLvmType(type.name), bblock->getParent()->getParent()->getDataLayout().getAllocaAddrSpace(), "", bblock);
    }

    CodeGenContext.pushBlock();//add new symble table


    llvm::Function::arg_iterator argsValues = function->arg_begin();
    llvm::Value* argumentValue;

    CodeGenContext.isArgs = true;
    for(auto it : args){
        (*it).CodeGen(CodeGenContext);
        argumentValue = &*argsValues++;
        argumentValue->setName((it)->id.name.c_str());
        llvm::StoreInst *inst = new llvm::StoreInst(argumentValue, CodeGenContext.getTop()[(it)->id.name], false, bblock);
    }
    CodeGenContext.isArgs = false;

    block.CodeGen(CodeGenContext);
    CodeGenContext.hasReturn = false;

    myBuilder.SetInsertPoint(CodeGenContext.returnBB);
    if(type.name.compare("void") == 0) {
        myBuilder.CreateRetVoid();
    } else {
        llvm::Value* ret = myBuilder.CreateLoad(getLLvmType(type.name), CodeGenContext.returnVal, "");
        myBuilder.CreateRet(ret);
    }

    CodeGenContext.popBlock();
    CodeGenContext.currentFunc = nullptr;
    std::cout << "Creating function: " << id.name << endl;
    return function;
}
```

4. function call

```
llvm::Value* NFunctionCall::CodeGen(CodeGenContext &CodeGenContext){
    if(ident.name == "printf"){ //printf、scanf、gets由llvm内部调用C语言函数处理
        return emitPrintf(CodeGenContext, args);
    } else if(ident.name == "scanf"){ //若调用 scanf 函数
        return emitScanf(CodeGenContext, args);
    } else if(ident.name == "gets") { // 若调用 gets 函数
        return emitGets(CodeGenContext, args);
    }
```

```
    //在module中查找以ident命名的函数体
    llvm::Function *func = CodeGenContext.myModule->getFunction(ident.name.c_str());
    if (func == NULL) {
        std::cerr << "no such function " << ident.name << endl;
    }

    vector<llvm::Value*> tmp;
    vector<NExpression*>::iterator i;
    for(auto i : args){  //对每个Expression进行IR代码生成并将结果存入tmp中
        tmp.push_back((*i).CodeGen(CodeGenContext));
    }
    //调用
    try{

 llvm::CallInst::Create(func,llvm::makeArrayRef(tmp),"",myBuilder.GetInsertBlock());
    }
    catch(...){
        cout<<"error in call:"<<lineno<<endl;
        throw;
    }
    cout << "Creating method call: " << ident.name << endl;
    return
llvm::CallInst::Create(func,llvm::makeArrayRef(tmp),"",myBuilder.GetInsertBlock());
}
```

1. control flow: 涉及到模块代码生成并压栈

```
llvm::Value* NIfStatement::CodeGen(CodeGenContext &CodeGenContext){
    cout << "Generating code for if-only"<<endl;
    llvm::Function *TheFunction = CodeGenContext.currentFunc;

    llvm::BasicBlock *IfBB = llvm::BasicBlock::Create(myContext, "if", TheFunction);
    llvm::BasicBlock *ThenBB = llvm::BasicBlock::Create(myContext,
"afterifonly",TheFunction);

    // 跳转判断语句
    llvm::Value *condValue = expr.CodeGen(CodeGenContext), *thenValue = nullptr,
*elseValue = nullptr;
    condValue = myBuilder.CreateICmpNE(condValue,
llvm::ConstantInt::get(llvm::Type::getInt1Ty(myContext), 0, true), "ifCond");
    auto branch = myBuilder.CreateCondBr(condValue, IfBB, ThenBB);

    myBuilder.SetInsertPoint(IfBB);
    // 将 if 的域放入栈顶
    CodeGenContext.pushBlock();
    if_blk.CodeGen(CodeGenContext);
    CodeGenContext.popBlock();
```

```
    if(CodeGenContext.hasReturn)
        CodeGenContext.hasReturn = false;
    else
        myBuilder.CreateBr(ThenBB);

    myBuilder.SetInsertPoint(ThenBB);
    return branch;
}
```

4 运行环境

包依赖：

`Flex`

`Bison`

`LLVM-10`

基于C/C++，我们使用了`Cmake`配置工程，理论上可以在多个平台上构建编译运行。

5 代码生成

递归调用CodeGen，将生成的IR代码存储在module中，并打印至标准输出。

```
void CodeGenContext::Run(NBlock* Root){
    Root->CodeGen(*this);
    llvm::verifyModule(*this->myModule, &llvm::outs());
    this->myModule->print(llvm::outs(), nullptr);//print module details
}
```

6 符号表设计

符号表采用map，将变量名和llvm::Value*存储在map中，经由llvm::Value指针可以获得存储的数据的类型以及存储地址。

利用一个**symbolTable类**作为符号表，存储块内的变量表 local_var 及对应的变量-llvm 类型表 local_var_type；

```
class symbolTable{
public:
    map<string, llvm::Value*> local_var; //局部变量 map
    map<string, llvm::Type*> local_var_type;//局部变量 string-llvm::type 的 map
};
```

7 测试案例和结果

- 运行程序，生成LLVM IR中间代码在.ll 文件中
- 执行以下命令生成可执行文件

```
bash run.sh test_name
```

`run.sh`

```
#!/bin/bash

llvm-as $1".ll"
llc $1".bc"
clang-11 -c $1".s"
clang-11 -o $1 $1".o"
```

- 测试结果
  - 基础功能测试

```
// func
int main(){
    // int, float, char, string
    int a = 1;
    float b = 2.0;
    char c = 'h';
    printf("%d %f %s\n", a, b, c);

    // +, -, *, /, ==, >=, <=, <, >, !=, !, =
    // ||, &&
    // if, else
    int m = 3;
    int n = 1;
    int p = 2;
    if(n > p || n <= m) {
        printf("True\n");
    }
    else {
        printf("False\n");
    }

    // array
    // while, break, return
    int A[3];
    int i=0;
    while(i<3){
```

```
        A[i] = i;
        i = i + 1;
    }
    i = 0;
    while(i<3){
        printf("%d ", A[i]);
        break;
    }

    return 0;
}
```

测试结果：

```
cloudstorm@cloudstorm-virtual-machine:~/桌面/6-5MiniCompiler/Mini-Compiler/build$ cd ../test/test6_8_1/
cloudstorm@cloudstorm-virtual-machine:~/桌面/6-5MiniCompiler/Mini-Compiler/test/test6_8_1$ bash run.bash test
cloudstorm@cloudstorm-virtual-machine:~/桌面/6-5MiniCompiler/Mini-Compiler/test/test6_8_1$ test
cloudstorm@cloudstorm-virtual-machine:~/桌面/6-5MiniCompiler/Mini-Compiler/test/test6_8_1$ ./test
1 2.000000 h
True
```

- 复杂样例1：递归求解Fibonacci数列

```
int fib(int a){
    if(a == 1 || a == 0){
        return 1;
    }
    return fib(a-1)+fib(a-2);

}
int main(){
    printf("%d", fib(5));
    return 0;
}
```

测试结果：

```
cloudstorm@cloudstorm-virtual-machine:~/桌面/6-5MiniCompiler/Mini-Compiler/test/fib$ ./fib
8cloudstorm@cloudstorm-virtual-machine:~/桌面/6-5MiniCompiler/Mini-Compiler/test/fib$
```

- 复杂样例2：Quicksort

```
void quicksort(int A[10], int left, int right) {
    int i;
    int j;
    int x;
    int y;
    i = left;
    j = right;
    x = A[(left + right) / 2];
    while(i <= j) {
        while (A[i] < x) {
```

```
            i = i + 1;
        }
        while (x < A[j]) {
            j = j - 1;
        }
        if (i <= j) {
            y = A[i];
            A[i] = A[j];
            A[j] = y;
            i = i + 1;
            j = j - 1;
        }
    }
    if (left < j) {
        quicksort(A, left, j);
    }
    if (i < right) {
        quicksort(A, i, right);
    }
    return;
}

int main()
{
    int B[1000000];
    int N;
    scanf("%d", &N);
    int i = 0;
    while(i < N) {
        scanf("%d", &B[i]);
        i = i + 1;
    }
    int left = 0;
    int right = N - 1;
    quicksort(B, left, right);
    i = 0;
    while(i < N) {
        printf("%d\n", B[i]);
        i = i + 1;
    }

    return 0;
}
```

测试结果：

```
cloudstorm@cloudstorm-virtual-machine:~/桌面/6-5MiniCompiler/Mini-Compiler/test/quicksort$ ./quick
6 55 4 7 56 3 7 22 1
3
4
7
7
55
56
cloudstorm@cloudstorm-virtual-machine:~/桌面/6-5MiniCompiler/Mini-Compiler/test/quicksort$
```