浙江大学

本科实验报告

课程名称：	编译原理
姓　　名：	沈韵泓
学　　院：	计算机科学与技术学院
系：	计算机系
专　　业：	软件工程
学　　号：	3200104392
指导教师：	王强

2023 年 5 月 20 日

0 序言

本次实验完成了一个类C语言编译器，支持将其编译至 LLVM IR，随后通过 LLVM 编译至 x86_64 汇编语言，并使用 GCC 生成最终的可执行文件。

本项目通过 CMake 搭建工程，同时提供一系列 make 命令用于快速生成目标语言文件与可执行文件。

本项目使用 Echarts 提供的树状图实现抽象语法树的可视化。

0.1 实验环境与依赖

- 实验环境
  - Ubuntu 20.04
  - GCC 9.0.4
  - Bison 3.5.1
  - Flex 2.6.4
- 依赖
  - Cmake 3.16.3
  - LLVM 10.0.0
  - VSCode 中的 Live Server 插件

0.2 项目结构

```
.
├─── Report.pdf         // 实验报告
├─── CMakeLists.txt
├─── Makefile
├─── README.md          // 构建及运行说明
├─── build
|    └─── Compiler      // 可执行文件
├─── src                // 源代码文件
|    ├─── Makefile
|    ├─── context.cpp   // 上下文
|    ├─── context.h     // 上下文头文件
|    ├─── genCode.cpp   // 中间代码生成
|    ├─── genJSON.cpp   // 抽象语法树生成
|    ├─── lex.l         // 词法分析
|    ├─── main.cpp      // 入口文件
|    ├─── node.h        // 节点定义
|    ├─── parse.y       // 语法&语义分析
|    ├─── util.cpp      // 工具类（类型转换等）
|    └─── util.h        // 工具类头文件
```

```
├── test                // 测试代码文件
|   ├── test.c          // 测试输入
|   └── Makefile
└── visual
    ├── Makefile
    ├── data.js         // JSON结构的抽象语法树
    └── index.html      // 抽象语法树可视化
```

0.3 支持特性

本编译器支持的语言特性如下：

- 类型支持：(正或负的) `int (Dec/Hex/Oct)` / `float`，`char`，`void`，支持数组
- 语句支持：`IF`，`IF-ELSE`，`WHILE`，`FOR`，`break`，`continue`，空语句
- 表达式支持：`[]` 数组下标访问，`&` 取地址，`++` 前后增，`+-*/`，`+= -= *= /=`，`> >= < <= == !=`，`&&`，`||`
- 隐式类型转换：`int + float = float`，`int i = 1.0 => i = 1`
- C标准函数：`scanf`，`printf`，`gets`
- 符号表作用域：支持在 `IF`，`IF-ELSE`，`WHILE`，`FOR` 内定义变量，并覆盖外层作用域中的同名变量。
- 支持函数先声明后定义

0.4 组员分工

本实验各部分均由沈韵泓完成。

1 词法分析

1.1 正规表达式

正规表达式的基本语法如下：

- 基本表达式
  - `a` - 匹配字符串 `"a"`
  - `ε` - 匹配空字符串
- 基本运算
  - `|` - 或
  - `*` - 闭包（重复≥ 0次）
- 扩展运算
  - `+` - 重复≥ 1次
  - `.` - 除换行符外的任意**单个**字符
  - `[]` - 匹配给定自负中的任意一个
  - `[^abc]` - 非（声明范围的补集）
  - `?` - 前面的一个表达式出现 **0/1** 次

1.2 实现原理及方法

1.2.1 本编辑器涉及的 Token

基于上述 LEX 基本语法与本编译器支持的语法特性，我设计了以下的正规表达式与 Token 的对应关系：

> 合法 `identifier` 应当以 下划线/字母 开头，切仅包含 数字/字母/下划线。

正规表达式	TOKEN	说明
`"//"[^\n]*`		匹配单行注释
`"while"`	WHILE	
`"break"`	BREAK	
`"continue"`	CONTINUE	
`"if"`	IF	
`"else"`	ELSE	
`"for"`	FOR	
`"return"`	RETURN	
`"int"|"float"|"char"|"void"`	TYPE	支持的数据类型
`[a-zA-Z_]([a-zA-Z_]|[0-9])*`	IDENTIFIER	合法标识符
`[-+]?0[xX][a-fA-F0-9]+`	CONST_HEX	（正或负的）十六进制整数
`[-+]?0[0-7]*`	CONST_OCT	（正或负的）八进制整数
`[-+]?[1-9][0-9]*`	CONST_DEC	（正或负的）十进制整数
`[0-9]+([Ee][+-]?[0-9]+)`	CONST_FLOAT	
`[0-9]*"."[0-9]+([Ee][+-]?[0-9]+)?`	CONST_FLOAT	
`[0-9]+"."[0-9]*([Ee][+-]?[0-9]+)?`	CONST_FLOAT	
`\'.\'|\'\\.\'`	CONST_CHAR	单引号包裹 - 字符
`\"(\\.|[^"\\])*\"`	CONST_STRING	双引号包裹 - 字符串
`"&&"`, `"||"`	AND, OR	
`"<="`, `">="`	LEQ, GEQ	
`"=="`, `"!="`	EQU, NEQ	
`">"`, `"<"`	LESST, GREATERT	
`"++"`	INC	自增

`"+="`, `"-="`, `"*="`, `"/="`	PLUS/MINUS/MUL/DIV_EQ	
`"+"`, `"-"`, `"*"`, `"/"`	PLUS, MINUS, MUL, DIV	
`","`, `";"`	COMMA, SEMI	
`"="`	ASSIGN	赋值
`"{"`, `"}"`	L/R_BRACE	
`"("`, `")"`	L/R_ROUND_BRAC	用于函数参数
`"["`, `"]"`	L/R_SQUARE_BRAC	用于数组
`"&"`	GAD	取地址符
`[ \t\v\n\f]`		涵盖 `.` 无法识别的字符

1.2.2 LEX 文件实现

LEX 文件通常由：定义（definition），规则（rule）及辅助程序（auxiliary）三部分组成。

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

Definitions

此处定义了必须的头文件、宏与函数：

- 宏 `SAVE_TOKEN` 用于将待进行后续处理的的字符串保存至 `(string*) yyval.string` 中
- 宏 `TOKEN()` 用于将 token 保存至 `(int) yyval.token` 中，用于后续识别运算符类型

> - `yyval` 的相关结构在 `yacc` 中有详细定义，相关片段如下：
>
>   ```
>   %union {
>       int token;
>       std::string *string;
>   }
>   ```
>
> - `yytext` 与 `yyleng` 分别用于返回本次匹配的字串内容与长度

```
#include <stdio.h>
#include <iostream>
#include <string>

#include "node.h"
#include "parse.hpp"

#define SAVE_TOKEN yylval.string = new std::string(yytext, yyleng)
#define TOKEN(t) (yylval.token = t)
```

Rules

本编译器使用如下规则匹配关键字、运算符与其他有意义符号：

```
    /* comment */
"//"[^\n]*                              { }

    /* keywords */
"while"                                 { return TOKEN(WHILE);    }
"break"                                 { return TOKEN(BREAK);    }
"continue"                              { return TOKEN(CONTINUE); }
"if"                                    { return TOKEN(IF);       }
"else"                                  { return TOKEN(ELSE);     }
"for"                                   { return TOKEN(FOR);      }
"return"                                { return TOKEN(RETURN);   }


    /* support types */
"int"|"float"|"char"|"void"             { SAVE_TOKEN; return TYPE; }

    /* identifier */
[a-zA-Z_]([a-zA-Z_]|[0-9])*             { SAVE_TOKEN; return IDENTIFIER; }

    /* constant */
[-+]?0[xX][a-fA-F0-9]+                  { SAVE_TOKEN; return CONST_HEX; }
[-+]?0[0-7]*                            { SAVE_TOKEN; return CONST_OCT; }
[-+]?[1-9][0-9]*                        { SAVE_TOKEN; return CONST_DEC; }
[0-9]+([Ee][+-]?[0-9]+)                 { SAVE_TOKEN; return CONST_FLOAT; }
[0-9]*"."[0-9]+([Ee][+-]?[0-9]+)?       { SAVE_TOKEN; return CONST_FLOAT; }
[0-9]+"."[0-9]*([Ee][+-]?[0-9]+)?       { SAVE_TOKEN; return CONST_FLOAT; }
\'.\'|\'\\.\'                           { SAVE_TOKEN; return CONST_CHAR; }
\"(\\.|[^"\\])*\"                       { SAVE_TOKEN; return CONST_STRING; }

    /* operators */
"&&"                                    { return TOKEN(AND); }
"||"                                    { return TOKEN(OR); }

"<="                                    { return TOKEN(LEQ); }
">="                                    { return TOKEN(GEQ); }
"=="                                    { return TOKEN(EQU); }
"!="                                    { return TOKEN(NEQ); }
"<"                                     { return TOKEN(LESST); }
">"                                     { return TOKEN(GREATERT); }

"++"                                    { return TOKEN(INC); }
"+="                                    { return TOKEN(PLUS_EQ); }
"-="                                    { return TOKEN(MINUS_EQ); }
"*="                                    { return TOKEN(MUL_EQ); }
"/="                                    { return TOKEN(DIV_EQ); }
"-"                                     { return TOKEN(MINUS); }
"+"                                     { return TOKEN(PLUS); }
"*"                                     { return TOKEN(MUL); }
```

```
"/"                                     { return TOKEN(DIV); }

","                                     { return TOKEN(COMMA); }
";"                                     { return TOKEN(SEMI); }
"="                                     { return TOKEN(ASSIGN); }
"{"                                     { return TOKEN(L_BRACE); }
"}"                                     { return TOKEN(R_BRACE); }
"("                                     { return TOKEN(L_ROUND_BRAC); }
")"                                     { return TOKEN(R_ROUND_BRAC); }
"["                                     { return TOKEN(L_SQUARE_BRAC); }
"]"                                     { return TOKEN(R_SQUARE_BRAC); }
"&"                                     { return TOKEN(GAD); }

[ \t\v\n\f]                             { }
.                                       { printf("unknown token : %s in line: %d\n", yytext,
yylineno); }
```

Auxiliary Routines

本编译器的语法分析部分仅涉及一个辅助程序 `yywrap()` ：

> 返回 1 以约束程序在遇到文件结尾时结束本次扫描。

```
int yywrap(void) {
    return 1;
}
```

2 语法分析

2.1 支持语法

由于本编译器仅支持 C语言 的部分特性，在此对本编译器支持的语法进行详述：

- 变量
  - 本编译器仅支持一次定义一个变量：

    ```
    int i;       // 合法
    int i, j, k; // 非法
    ```

  - 本编译器支持在定义单个变量的同时对其进行初始化：

    ```
    int i = 3;
    ```

    - 局部变量未进行初始化时，其值为随机数
    - 全局变量未进行初始化时，将统一初始化为 0
  - 本编译器不支持在定义数组时进行初始化：

```
int arr[3] = {1, 2, 3}; // 非法

// 您应当逐个对数组变量进行赋值
int arr[3];
arr[0] = 0;
arr[1] = 1;
arr[2] = 2;
```

- 函数
  - 本编译器默认支持 `printf`，`scanf`，`gets` 函数，无需声明即可使用
  - 本编译器支持函数先声明后使用，下面的例子是合法的：

    ```
    void HelloWorld();
    void HelloWorld() {
      printf("Hello World!");
      return;
    }
    ```

    需要注意的是，函数声明与定义的返回值类型应该一致，下面的例子是非法的：

    ```
    void HelloWorld();
    int HelloWorld() {
      printf("Hello World!");
      return 1;
    }
    ```

  - 所有函数都应该包含 `return` 语句（包括返回值为 `void` 的函数），下面的例子是非法的：

    ```
    void HelloWorld() {
      printf("Hello World!");
    }
    ```

- 空语句
  本编译器支持空语句，单独使用 `;` 是合法的
- IF 语句

  本编译器支持 IF 和 IF-ELSE 结构，但所有代码块应当用 `{}` 包裹，条件非空且不允许声明新的变量：

  ```
  // 以下例子合法
  if(i < 0) {
    printf("i < 0\n");
  } else {
    printf("i >= 0\n");
  }

  // 非法 - 代码块应当以 {} 包裹
  if(i < 0)
    printf("i < 0\n");

  // 非法 - 条件不得为空
  if() {
  ```

```
  printf("true\n");
}

// 非法 - 不得在条件中声明新的变量
if(int j = 5) {
  printf("j = 5\n");
}
```

- WHILE 语句

  本编译器支持 WHILE 结构，代码块应当用 `{}` 包裹，条件非空且不允许声明新的变量：

```
while(i < 10) {
  printf(" i = %d\n", i);
  i += 1;
}
```

- FOR 语句

  本编译器支持结构，代码块应当用 `{}` 包裹，所有条件非空且不允许声明新的变量：

```
for(i=0 ; i<=50 ; i++) {
  sum += i;
}
```

- break & continue

  本编译器支持 break & continue 语句，且仅适用于 FOR / WHILE 循环内部使用

- 四则运算

  由于本编译器支持负数，因此您必须在 `+ -` 号后添加空格，避免识别错误：

```
// 以下操作合法
i = 3 - 2;

// 以下操作非法（被认为是两个连续出现的常量 Token：‘3’ & ‘-2’）
i = 3-2;
```

2.2 YACC

本编译器使用 YACC 作为分析程序生成器，输入为说明文件 `parse.y`，输出由 C 源代码组成的文件。`.y` 文件的基本格式如下：

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

2.2.1 Definitions

1. 该部分定义了一些必要的头文件、全局变量与工具函数：
    - `prog` 为整个抽象语法树的根节点
    - 函数 `HtoD()` 与 `OtoD()` 用于将 `yyval.string` 中传入的十六/八进制字符串转为十进制整数

```
%{
    #include <cstdio>
    #include <cstdlib>
    #include <sstream>
    #include "node.h"
    #define YYERROR_VERBOSE 1
    BlockNode *prog;
    extern int yylex();
    extern int yylineno;
    // 处理报错信息
    void yyerror(const char *errMsg) {
      std::printf("Error: %s @ line %d\n", errMsg, yylineno);
    }
    // 进制转换
    int HtoD(string *s) {
        int d_int;
        stringstream ss;
        ss << hex << *s;
        ss >> d_int;
        return d_int;
    }
    int OtoD(string *s) {
        int d_int;
        stringstream ss;
        ss << oct << *s;
        ss >> d_int;
        return d_int;
    }
%}
```

2. 该部分定义了语法分析过程中可能涉及的类型

```
%define parse.error verbose
%union {
    int token;
    int constInt;
    std::string *string;
    TypeNode *type;
    IdentifierNode *identifier;
    TreeNode *node;
    BlockNode *block;
    StatementNode *statement;
    ExpressionNode *expression;
    std::vector<ExpressionNode*> *expList;
    VariableDeclarationNode *varDeclaration;
```

```
    std::vector<VariableDeclarationNode*> *varDeclarationList;
}
```

3. 该部分定义了各 token 对应的类型、运算符结合性与 CFG 文法的起始符 `program`

```
%token <string> IDENTIFIER TYPE
%token <string> CONST_HEX CONST_OCT CONST_DEC CONST_FLOAT CONST_CHAR CONST_STRING
%token <token> PLUS MINUS MUL DIV INC
%token <token> PLUS_EQ MINUS_EQ MUL_EQ DIV_EQ
%token <token> AND OR                           // &&  ||
%token <token> GAD                              // &
%token <token> EQU NEQ LESST GREATERT LEQ GEQ   // == !=  <  >  <=  >=
%token <token> RETURN
%token <token> SEMI COMMA ASSIGN                // ; , =
%token <token> L_BRACE R_BRACE                  // {}
%token <token> L_ROUND_BRAC R_ROUND_BRAC        // ()
%token <token>  L_SQUARE_BRAC R_SQUARE_BRAC     // []
%token <token> IF ELSE WHILE BREAK CONTINUE FOR

%left AND OR
%left EQU NEQ LESST GREATERT LEQ GEQ
%left PLUS MINUS MUL DIV
%left L_ROUND_BRAC R_ROUND_BRAC L_SQUARE_BRAC R_SQUARE_BRAC

%type <identifier> identifier
%type <type> type
%type <expression> const expression
%type <constInt> const_int
%type <statement> statement return_statement while_statement if_statement for_statement
empty_statement
%type <statement> var_decl func_decl func_def
%type <varDeclarationList> params
%type <expList> args
%type <block> program statements block

%start program
```

2.2.2 Rules

每一条规约的左侧为语法规则，`{}` 内为对应的 C语言 操作。`$$` 为规约后压入栈中的值，`$1 ~ $N` 为规约前栈中的值。

此处仅对 CFG 文法进行展示：

```
program:
    statements
    ;

statements:
    statement

    | statements statement
```

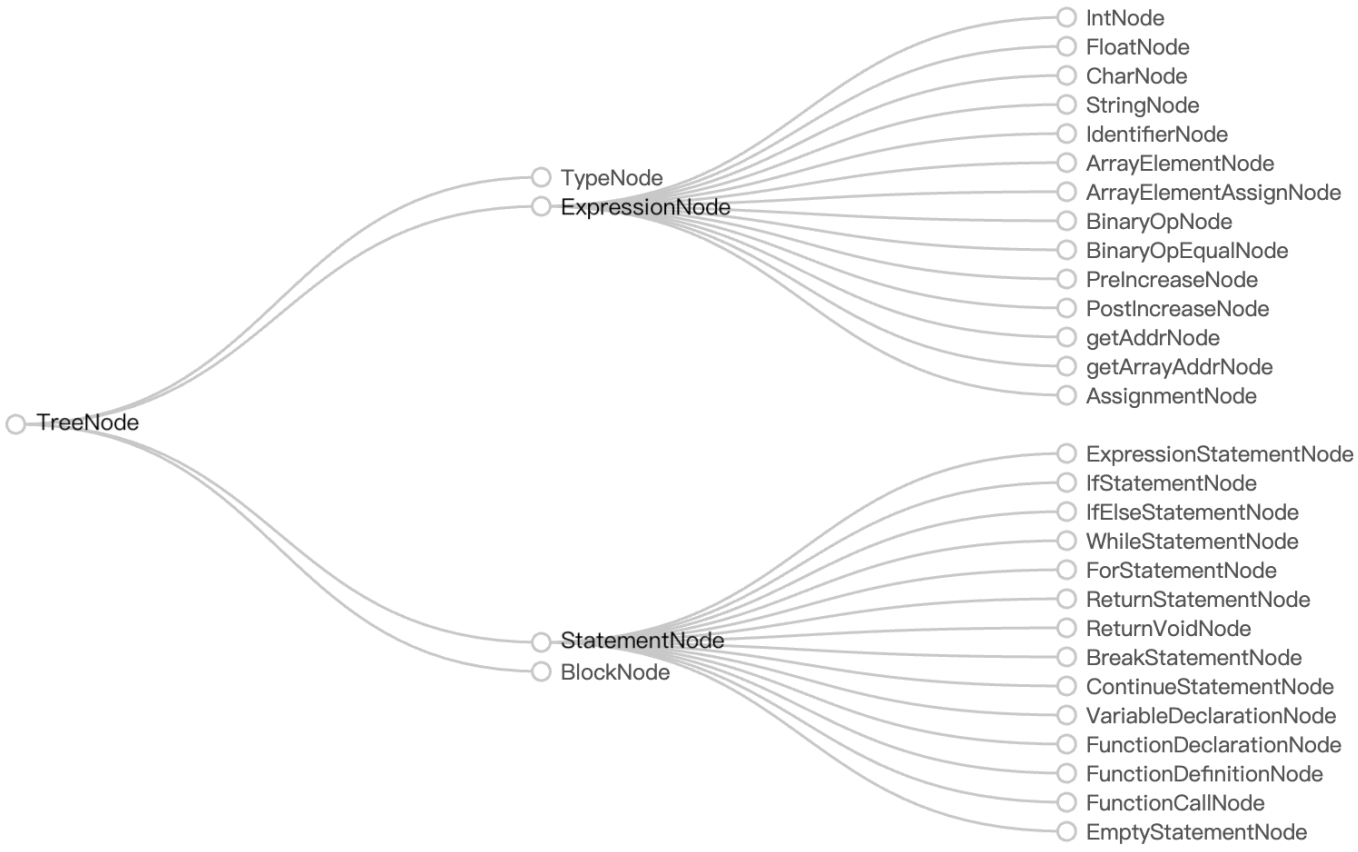
```
    ;

statement:
    var_decl SEMI
    | func_decl SEMI
    | func_def
    | while_statement
    | if_statement
    | for_statement
    | return_statement
    | empty_statement
    | expression SEMI
    ;

return_statement:
    RETURN SEMI
    | RETURN expression SEMI
    ;

while_statement:
    WHILE L_ROUND_BRAC expression R_ROUND_BRAC block
    | CONTINUE SEMI
    | BREAK SEMI
    ;

if_statement:
    IF L_ROUND_BRAC expression R_ROUND_BRAC block
    | IF L_ROUND_BRAC expression R_ROUND_BRAC block ELSE block
    ;

for_statement:
    FOR L_ROUND_BRAC expression SEMI expression SEMI expression R_ROUND_BRAC block
    ;

empty_statement:
    SEMI
    ;

block:
    L_BRACE statements R_BRACE
    | L_BRACE R_BRACE
    ;

var_decl:
    type identifier
    | type identifier ASSIGN expression
    | type identifier L_SQUARE_BRAC const_int R_SQUARE_BRAC
    ;

func_def:
    type identifier L_ROUND_BRAC params R_ROUND_BRAC block
    ;
```

```
func_decl:
    type identifier L_ROUND_BRAC params R_ROUND_BRAC
    ;

params:
    { }
    | var_decl
    | params COMMA var_decl
    ;

identifier:
    IDENTIFIER
    ;

type:
    TYPE
    ;

const:
    const_int
    | CONST_FLOAT
    | CONST_CHAR
    | CONST_STRING
    ;

const_int:
    CONST_DEC
    | CONST_HEX
    | CONST_OCT
    ;

args:
    { }
    | expression
    | args COMMA expression
    ;

expression:
    identifier ASSIGN expression
    | identifier L_ROUND_BRAC args R_ROUND_BRAC
    | identifier
    | GAD identifier
    | GAD identifier L_SQUARE_BRAC expression R_SQUARE_BRAC
    | identifier INC
    | INC identifier
    | identifier PLUS_EQ expression
    | identifier MINUS_EQ expression
    | identifier MUL_EQ expression
    | identifier DIV_EQ expression
    | expression MUL expression
    | expression DIV expression
```

```
    | expression PLUS expression
    | expression MINUS expression
    | expression AND expression
    | expression OR expression
    | expression LESST expression
    | expression GREATERT expression
    | expression EQU expression
    | expression NEQ expression
    | expression LEQ expression
    | expression GEQ expression
    | L_ROUND_BRAC expression R_ROUND_BRAC
    | identifier L_SQUARE_BRAC expression R_SQUARE_BRAC
    | identifier L_SQUARE_BRAC expression R_SQUARE_BRAC ASSIGN expression
    | const
    ;
```

2.3 抽象语法树

语法分析程序的输出是抽象语法树，各节点类型之间的继承关系如下：



2.3.1 TreeNode 类

TreeNode 类规定了抽象语法树所有节点需要包含的基本方法，包括：

- 成员变量 `lineNo` 及包含参数 `lineNo` 的构造函数（用于在报错时输出所在行号）
- 虚函数 `llvm:Value* genCode(Context &curContext)` ，用于基于当前上下文情况生成节点对应的中间代码
- 纯虚函数 `void genJSON(string &s)` ，用于在给定字符串中拼接 AST 可视化所需的 JSON 字符串

```
class TreeNode {
public:
  TreeNode(int lineNo) : lineNo(lineNo) {}
  virtual llvm::Value *genCode(Context &curContext) { return nullptr; }
  virtual void genJSON(string &s) = 0;

  int lineNo;
};
```

2.3.2 TypeNode 类

继承自 TreeNode 类，用于保存 LEX 中识别的合法变量类型（`int`，`float`，`char`，`void`）

```
class TypeNode : public TreeNode {
public:
  TypeNode(string &name, int lineNo) : TreeNode(lineNo), name(name) {}
  llvm::Value *genCode(Context &curContext) { return nullptr; }
  void genJSON(string &s);

  string name;
};
```

2.3.3 ExpressionNode 类

继承自 TreeNode 类型，包含所有表达式（具有返回值的结构），包括常量表达式，变量表达式，等等。

具有虚函数 `llvm::Value *getAddr(Context &curContext)`，用于对部分子类进行取地址操作。

```
class ExpressionNode : public TreeNode {
public:
  ExpressionNode(int lineNo) : TreeNode(lineNo) {}
  virtual llvm::Value *getAddr(Context &curContext) { return nullptr; }
};
```

2.3.4 StatementNode 类

继承自 TreeNode 类，包括所有的语句类型，如 FOR 循环，WHILE 循环，函数声明，函数定义，等等。

```
class StatementNode : public TreeNode {
public:
  StatementNode(int lineNo) : TreeNode(lineNo) {}
};
```

2.3.5 BlockNode 类

继承自 TreeNode 类，由一连串的语句序列构成。

```
class BlockNode : public TreeNode {
public:
  BlockNode(int lineNo) : TreeNode(lineNo) {}
  BlockNode(vector<StatementNode*> statementList, int lineNo) : TreeNode(lineNo),
 statementList(statementList) {}
  llvm::Value *genCode(Context &curContext);
  void genJSON(string &s);

  vector<StatementNode*> statementList;
};
```

2.4 抽象语法树可视化

本项目中的抽象语法树可视化籍由 `Echarts` 提供的树状图组件实现，其要求的 JSON 数据源的基本格式如下：

```
{
  "name" : "Level 0",
  "children" : [
    {
      "name" : "Level 1-1",
      "children" : [...]
    },
    { "name" : "Level 1-2"},
    ...
  ]
}
```

具体实现过程如下：

1. 输入经 Flex & Bison 处理后，形成了以 `BlockNode* prog` 为根节点的树状结构
2. 初始化空字符串 `string s`，并从根节点 `prog` 出发递归调用 `genJSON(string &s)`（相关实现在 `genJSON.cpp` 中），在字符串中拼接对应的 JSON 内容
3. 将以下内容输出至 `test/data.js` 中：

   ```
   // 为确保能够正确以 ES6 语法引入，必须以该形式书写
   const data = string s;
   export default {
     data
   };
   ```

4. 通过 Live Server 插件运行 `test/index.html` 以确保 `data.js` 能被正确引入
5. 在打开的网页中查看抽象语法树可视化图

2.4.1 genJSON 函数实现（部分）

```
// 整型常量节点
void IntNode::genJSON(string &s) {
    s.append("\n{\n");
    s.append("\"name\" : \"IntValue\",\n");
    s.append("\"children\" : [{ \"name\" : \"" + to_string(this->value) + "\" }]\n");
```

```
    s.append("}");
}

// 二元运算节点
void BinaryOpNode::genJSON(string &s) {
    s.append("\n{\n");
    s.append("\"name\" : \"BinaryOperation\",\n");
    s.append("\"children\" : [");

    // left oprand
    this->lhs.genJSON(s);
    s.append(",");

    // operator
    s.append("\n{\n");
    s.append("\"name\" : \"Operator\",\n");
    s.append("\"children\" : [{ \"name\" : \"");
    switch (this->op) {
        case PLUS :     { s.append("+");    break; }
        case MINUS :    { s.append("-");    break; }
        case MUL :      { s.append("*");    break; }
        case DIV :      { s.append("/");    break; }
        case OR :       { s.append("||");   break; }
        case AND :      { s.append("&&");   break; }
        case EQU :      { s.append("==");   break; }
        case NEQ :      { s.append("!=");   break; }
        case LESST :    { s.append("<");    break; }
        case GREATERT : { s.append(">");    break; }
        case LEQ :      { s.append("<=");   break; }
        case GEQ :      { s.append(">=");   break; }
        default:        { s.append(" ");    break; } //"ERROR"
    }
    s.append("\" }]\n");
    s.append("},");

    // right oprand
    this->rhs.genJSON(s);

    s.append("\n]\n");
    s.append("}");
}

// For 循环语句节点
void ForStatementNode::genJSON(string &s) {
    s.append("\n{\n");
    s.append("\"name\" : \"ForStatementNode\",\n");
    s.append("\"children\" : [");

    this->initiation.genJSON(s);
    s.append(",");

    this->condition.genJSON(s);
```

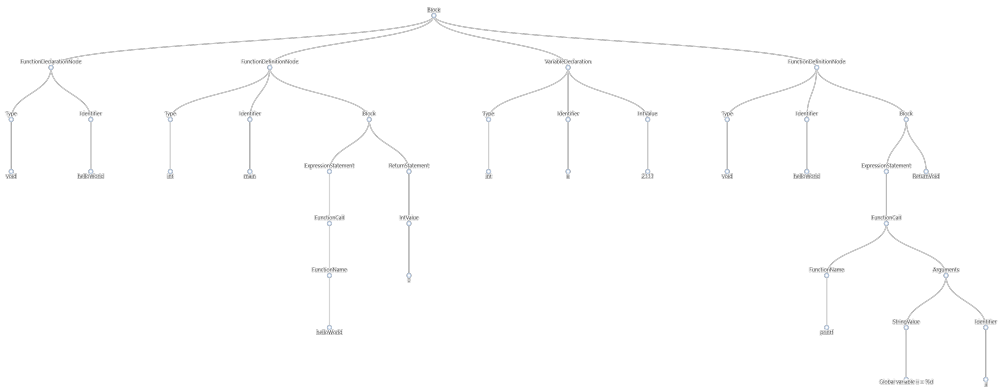
```
    s.append(",");

    this->increment.genJSON(s);
    s.append(",");

    this->body.genJSON(s);

    s.append("]");
    s.append("}");
}
```

2.4.2 AST 可视化结果实例

对如下的简单程序进行可视化：

```
void helloWorld();

int main() {
    helloWorld();
    return 0;
}

int ii = 2333;

void helloWorld() {
    printf("Global variable ii = %d\n", ii);
    return;
}
```

可视化后的抽象语法树如下图所示：



3 语义分析与中间代码生成

在这一步中，皆由 LLVM 提供的 C++ API 实现语义分析及到中间代码 LLVM IR 的转化

3.1 运行环境设计

```
llvm::Module *module; // 程序所在模块

vector<symbolTable *> symbolTable_stack;    //  符号表栈

stack<llvm::BasicBlock *> AfterBBStack;   // afterloop 标签构成的栈    -> 记录 break 返回位置
stack<llvm::BasicBlock *> ContBBStack;    // cond/loopinc 标签构成的栈 -> 记录 continue 返回位置

// 当前正在处理的函数信息
llvm::Function* currentFunc;                // 正在处理的函数       (为空表示在函数外)
bool isArgs;                                // 标志当前是否处理形参 (为真需要返回指针)
bool hasReturn;                             // 是否跳出基本块       (为真则截断后续代码)
llvm::BasicBlock* returnBB;                 // 执行返回操作的基本块 (非void需要处理返回值)
llvm::Value* returnVal;                     // 返回值
```

3.2 符号表设计

本编译器为每一个作用域建立如下所示的符号表，在进入时压入符号表栈 `vector<symbolTable *> symbolTable_stack`，并在离开时出栈，使得临时变量仅在有限的作用域内生效，且内层临时变量可覆盖外层同名变量。

> 全局变量与常量、函数一同被记录在全局作用域 `module` 中，也可以被内层同名变量覆盖：
>
> 这是由工具函数 `llvm::*Value** getVariable(*string* *variableName*)` 首先在 `symbolTable_stack` 中自顶向下查找，最后再查找全局变量的逻辑决定的。

```
class symbolTable{
public:
    // 也可以认为 llvm::Value* 就是变量的地址
    map<string, llvm::Value*> local_var;        //  局部符号表 identifier -> Value
    map<string, llvm::Type*>  local_var_type;   //  局部符号表 identifier -> Type
};
```

每一层的符号表中均存在两个 Map，分别记录 `identifier` 字符串与其值 (`llvm:Value*`) & 类型 (`llvm:Type*`) 的映射关系。

当程序试图访问名为 `str` 的类型时，可以快速通过对应 entry 是否存在来判断当前作用域中是否定义了指定变量，并以 `str` 作为键快速获取对应信息。

3.3 中间代码生成实现

3.3.1 类型转换处理

由于 TypeNode 本身只负责存储变量类型，并不会创建任何 IR 代码，此处介绍 C语言类型 与 LLVM 类型之间的转换，以及不同类型之间的映射关系。

- C 语言类型 -> LLVM 类型
  由于存在普通变量、数组变量及数组元素变量种情况，下面提供了三个工具函数实现从 C语言类型 (`string type`) 到 `llvm::Type` 的转换：

```
llvm::Type* getLLvmType(string type);
llvm::Type* getLLvmPtrType(string type);
llvm::Type* getArrayLLvmType(string type, int size);
```

本编译器支持的基本类型与 LLVM 类型的对应关系如下表所示：

C Type	LLVM Type
`int`	`i32`
`float`	`float`
`char`	`i8`
`void`	`void`

> 虽然本编译器不支持 `bool` 类型的变量，但在判断条件是否成立时仍涉及对应的 LLVM 类型 `i1`

- 隐式类型转换
  在支持的类型中，我们认为 `float > int > char` 。
  - 当左右两式类型不等时，我们均将较低的类型向较高的方向转换。下列函数实现了该逻辑：
    所有的 IF-ELSE 分支旨在寻找两者中最高等级的类型，并转化较低等级的值

    ```
    void balanceType(llvm::Value* &left, llvm::Value* &right) {
      if (left->getType() == llvm::Type::getFloatTy(GlobalContext)) {
          right = typeCast(right, llvm::Type::getFloatTy(GlobalContext));
      } else {
          if (right->getType() == llvm::Type::getFloatTy(GlobalContext)) {
              left = typeCast(left, llvm::Type::getFloatTy(GlobalContext));
          } else {
              if (left->getType() == llvm::Type::getInt32Ty(GlobalContext)) {
                  right = typeCast(right, llvm::Type::getInt32Ty(GlobalContext));
              } else if(right->getType() == llvm::Type::getInt32Ty(GlobalContext)) {
                  left = typeCast(left, llvm::Type::getInt32Ty(GlobalContext));
              } else {
                  throw logic_error("cann't use bool in +-*/");
              }
          }
      }
      return;
    }
    ```

  - 该函数中涉及的 `typeCast` 用于返回一条映射语句，用于临时将 `src` 映射为 `dst` 的类型：

    ```
    llvm::Value* typeCast(llvm::Value* src, llvm::Type* dst) {
        llvm::Instruction::CastOps op = getCastedInt(src->getType(), dst);
        return Builder.CreateCast(op, src, dst, "tmptypecast");
    }
    ```

  - `getCastedInt()` 同样是一个自定义函数，用于实现 `float-int-char` 之间的两两类型映射：

> 进行变量赋值时可能导致由高类型至低类型的逆向映射

```
llvm::Instruction::CastOps getCastedInt(llvm::Type* src, llvm::Type* dst) {
    if (src == llvm::Type::getFloatTy(GlobalContext)
        &&
        dst == llvm::Type::getInt32Ty(GlobalContext)) {
        return llvm::Instruction::FPToSI;  // float -> int
    } else if (src == llvm::Type::getInt32Ty(GlobalContext)
               &&
               dst == llvm::Type::getFloatTy(GlobalContext)) {
        return llvm::Instruction::SIToFP;  // int -> float
    } else if (src == llvm::Type::getInt8Ty(GlobalContext)
               &&
               dst == llvm::Type::getFloatTy(GlobalContext)) {
        return llvm::Instruction::UIToFP;  // char -> float
    } else if (src == llvm::Type::getInt8Ty(GlobalContext)
               &&
               dst == llvm::Type::getInt32Ty(GlobalContext)) {
        return llvm::Instruction::ZExt;    // char -> int
    } else if (src == llvm::Type::getInt32Ty(GlobalContext)
               &&
               dst == llvm::Type::getInt8Ty(GlobalContext)) {
        return llvm::Instruction::Trunc;   // int -> char
    } else if (src == llvm::Type::getFloatTy(GlobalContext)
               &&
               dst == llvm::Type::getInt8Ty(GlobalContext)) {
        return llvm::Instruction::FPTOUI; // float -> char
    } else {
        throw logic_error("[ERROR] Wrong typecast");
    }
}
```

3.3.2 ExpressionNode

1. 常量处理
   常量节点
   - Int 类型 & Float 类型
     简单返回其对应的 LLVM 类型常量值即可

     ```
     // int
     llvm::ConstantInt::get(llvm::Type::getInt32Ty(GlobalContext), this->value, true);

     // float
     return llvm::ConstantFP::get(llvm::Type::getFloatTy(GlobalContext), this->value);
     ```

   - Char 类型
     由于在词法分析中我们将单引号连同字符本身一同传入了 CharNode，此处仅需返回 idx=1 处字符的转化结果；

此外，转义字符将比普通字符多一个 `\`，因此我们应当根据 idx=2 处的字符判断转义字符的具体类型，并返回对应的转化值：

```
llvm::Value* CharNode::genCode(Context &curContext) {
    if (this->value.size() == 3) // '?'
        return Builder.getInt8(this->value.at(1));
    else {                       // '\?' input escape
        switch (this->value[2]) {
            case 'n' : return Builder.getInt8('\n'); break;
            case '\\': return Builder.getInt8('\\'); break;
            case 'a' : return Builder.getInt8('\a'); break;
            case 'b' : return Builder.getInt8('\b'); break;
            case 'f' : return Builder.getInt8('\f'); break;
            case 't' : return Builder.getInt8('\t'); break;
            case 'v' : return Builder.getInt8('\v'); break;
            case '\'': return Builder.getInt8('\''); break;
            case '\"': return Builder.getInt8('\"'); break;
            case '0' : return Builder.getInt8('\0'); break;
            default  : throw logic_error("[ERROR] Illegal char [" + this->value +
"]");
        }
    }
}
```

- String 类型

同样的，我们在处理字符串时传入了两侧的双引号，此处需要额外处理；
常量字符串仅用于 `printf` 函数的输出，我们为其创建对应的全局常量字符串（`i8(char)`数组），并返回对于其首地址的首地址的指针：

```
llvm::Value* StringNode::genCode(Context &curContext) {
    string str = value.substr(1, value.length() - 2); // "???" 去掉两边的引号
    escapeLineBreak(str);

    // 为需要引用的字符串创建对应的全局变量
    llvm::Constant *strConst = llvm::ConstantDataArray::getString(GlobalContext,
str);
    llvm::Value *globalVar = new llvm::GlobalVariable(*(curContext.module),
strConst->getType(), true, llvm::GlobalValue::PrivateLinkage, strConst,
"_Const_String_");

    // 用于将类型识别为 i8* 并指向字符数组的起始位置
    vector<llvm::Value*> indexList;
    indexList.push_back(Builder.getInt32(0));
    indexList.push_back(Builder.getInt32(0));

    return Builder.CreateInBoundsGEP(globalVar, llvm::ArrayRef<llvm::Value*>
(indexList), "tmpstring");
}
```

2. 变量处理
    1. 获取普通变量地址

通过查找逐层查找符号表栈（最后查找全局变量）即可实现，具体逻辑如下：

```
llvm::Value* Context::getVariable(string variableName) {
    // 首先按照作用域由近到远查找局部变量
    vector<symbolTable*>::reverse_iterator it = symbolTable_stack.rbegin();
    for (; it != symbolTable_stack.rend(); ++it) {
        symbolTable* curTable = *it;
        if(curTable->local_var.find(variableName) != curTable->local_var.end())
            return curTable->local_var[variableName];
    }
    // 若不存在局部变量，则查找全局变量（还是不存在会返回 nullptr）
    return this->module->getGlobalVariable(variableName, true);
}
```

2. 获取数组变量地址

   由于符号表栈中存储了所有数组的首地址，所以数组首地址的获取与普通变量一致（查找符号表栈即可）；若需要查找指定下标的数组元素，则需要使用 `CreateInBoundsGEP` 返回基于首地址偏移后的地址：

```
llvm::Value* ArrayElementNode::getAddr(Context &curContext) {
    // 查找符号表得到数组首地址
    llvm::Value* var = curContext.getVariable(identifier.name);
    if (!var) {
        throw logic_error("undeclared array [" + identifier.name + "]");
    }

    llvm::Value* indexValue = index.genCode(curContext);
    vector<llvm::Value*> indexList;
    indexList.push_back(Builder.getInt32(0));
    indexList.push_back(indexValue);          // 偏移量

    // 返回偏移后的地址
    return Builder.CreateInBoundsGEP(var, llvm::ArrayRef<llvm::Value*>(indexList),
"tmparray");
}
```

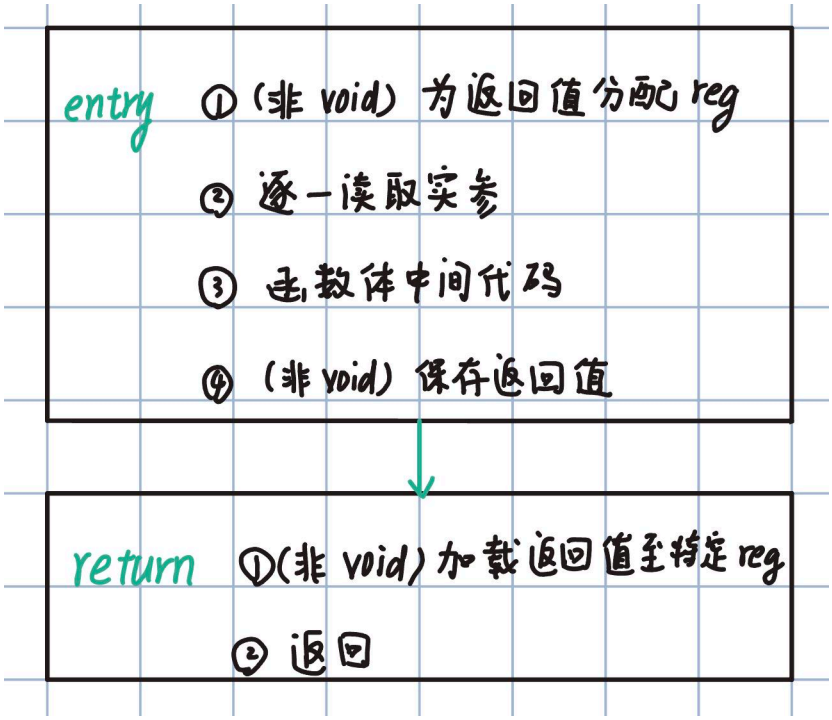
3. 二元运算处理 `+ - * /`
   - 首先读取两侧表达式的值与类型，若不等则进行映射（低 -> 高）；
   - 数值运算：
     根据运算符及左值是否为 `float` 类型选择对应操作（运算向左赋值）
   - 逻辑运算：
     判断左右值的类型是否为 `Int1Ty (bool)`，否则报错
4. 赋值处理
   - 在符号栈中查找该变量是否已经定义，获取其地址
   - 计算右值
   - 获取当前基本块，并插入赋值语句

```
llvm::Value* AssignmentNode::genCode(Context &curContext) {
    // 在符号表和全局变量中查找
    llvm::Value* var = curContext.getVariable(identifier.name);
```

```
    if (!var) {
        throw logic_error("undeclared variable [" + identifier.name + "]");
    }

    llvm::Value* val = value.genCode(curContext);

    // 函数外单独赋值的全局变量
    if (!curContext.currentFunc) {
        throw logic_error("Variable [" + identifier.name + "] can't be assigned out of
function");
    }

    auto CurrentBlock = Builder.GetInsertBlock();
    if (val->getType() != getVarPtrType(var))
        val = typeCast(val, getVarPtrType(var));

    return new llvm::StoreInst(val, var, false, CurrentBlock);
}
```

3.3.3 StatementNode

1. 函数声明处理
    - 查找是否已经存在同名函数（由于C语言不允许嵌套定义函数，直接在全局域中查找即可）
    - 初始化参数列表（数组需要返回指针类型）
    - 注册函数
2. 函数定义处理



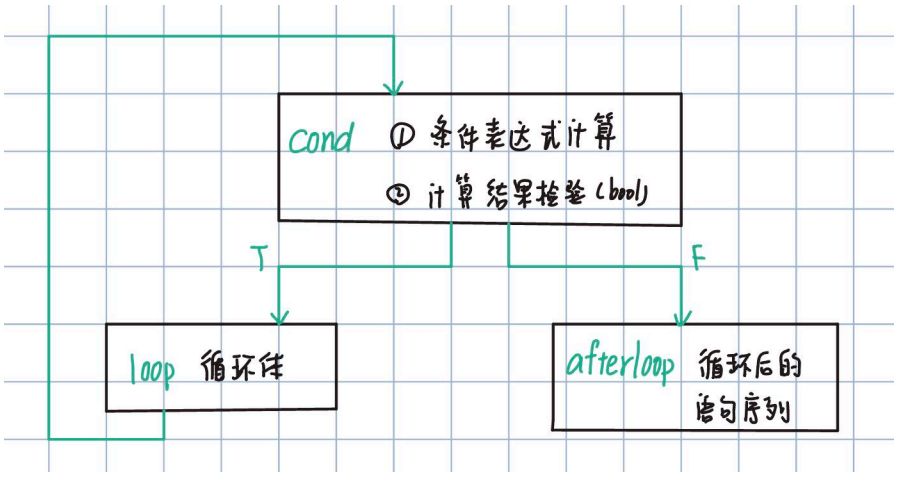

```
llvm::Value* FunctionDefinitionNode::genCode(Context &curContext) {
    // 初始化参数类型列表
    vector<llvm::Type*> argTypes;
    for(auto it : args){
        if (it->size == 0) {
            argTypes.push_back(getLLvmType(it->type.name));
```

```
        } else {
            argTypes.push_back(getLLvmPtrType(it->type.name));
        }
    }

    // 查看先前是否声明，避免重复声明（不允许同名函数）
    llvm::Function *function = curContext.module->getFunction(identifier.name.c_str());
    llvm::FunctionType *fType;
    if (!function) {
        fType = llvm::FunctionType::get(getLLvmType(type.name), makeArrayRef(argTypes),
false);
        function = llvm::Function::Create(fType, llvm::GlobalValue::ExternalLinkage,
identifier.name.c_str(), curContext.module);
    } else {
        fType = function->getFunctionType();
        if(fType != llvm::FunctionType::get(getLLvmType(type.name),
makeArrayRef(argTypes), false)) {
            throw logic_error("Unmatch return/param type for FUNC[" + identifier.name +
"]");
        }
    }

    // initiation
    curContext.currentFunc = function;
    llvm::BasicBlock *bBlock = llvm::BasicBlock::Create(GlobalContext, "entry", function,
0);
    curContext.returnBB = llvm::BasicBlock::Create(GlobalContext, "return", function, 0);
    if (type.name.compare("void") != 0) { // 用于处理返回值
        curContext.returnVal = new llvm::AllocaInst(getLLvmType(type.name), bBlock-
>getParent()->getParent()->getDataLayout().getAllocaAddrSpace(), "", bBlock);
    }

    curContext.pushSymbolTable();
    Builder.SetInsertPoint(bBlock);

    // 处理形参列表
    llvm::Function::arg_iterator argsValues = function->arg_begin();
    llvm::Value* argumentValue;
    curContext.isArgs = true;
    for (auto it : args) {
        (*it).genCode(curContext);
        argumentValue = &*argsValues++;
        argumentValue->setName((it)->identifier.name.c_str());
        llvm::StoreInst *inst = new llvm::StoreInst(argumentValue,
curContext.getTopSymbolTable()->getVarValue((it)->identifier.name) , false, bBlock);
  }
    curContext.isArgs = false;

    // 处理函数体
    block.genCode(curContext);
    curContext.hasReturn = false;
```

```
    // 处理预返回阶段
    Builder.SetInsertPoint(curContext.returnBB);
    if (type.name.compare("void") == 0) {
        Builder.CreateRetVoid();
    } else {
        llvm::Value* ret = Builder.CreateLoad(getLLvmType(type.name),
curContext.returnVal, "");
        Builder.CreateRet(ret);
    }

    // 复位
    curContext.popSymbolTable();
    curContext.currentFunc = nullptr;

  return function;
}
```

3. 函数调用处理
   主意读取实参，并使用 `llvm::CallInst::Create()` 创建函数调用即可
4. Return 语句处理
   - 非空返回
     1. 计算返回值
     2. 映射至函数类型
     3. 将映射后的返回值存储至预先分配的寄存器
     4. 标记函数结束，跳转至 `return` 标签
   - 空返回
     标记函数结束，跳转至 `return` 标签即可
5. 变量声明处理

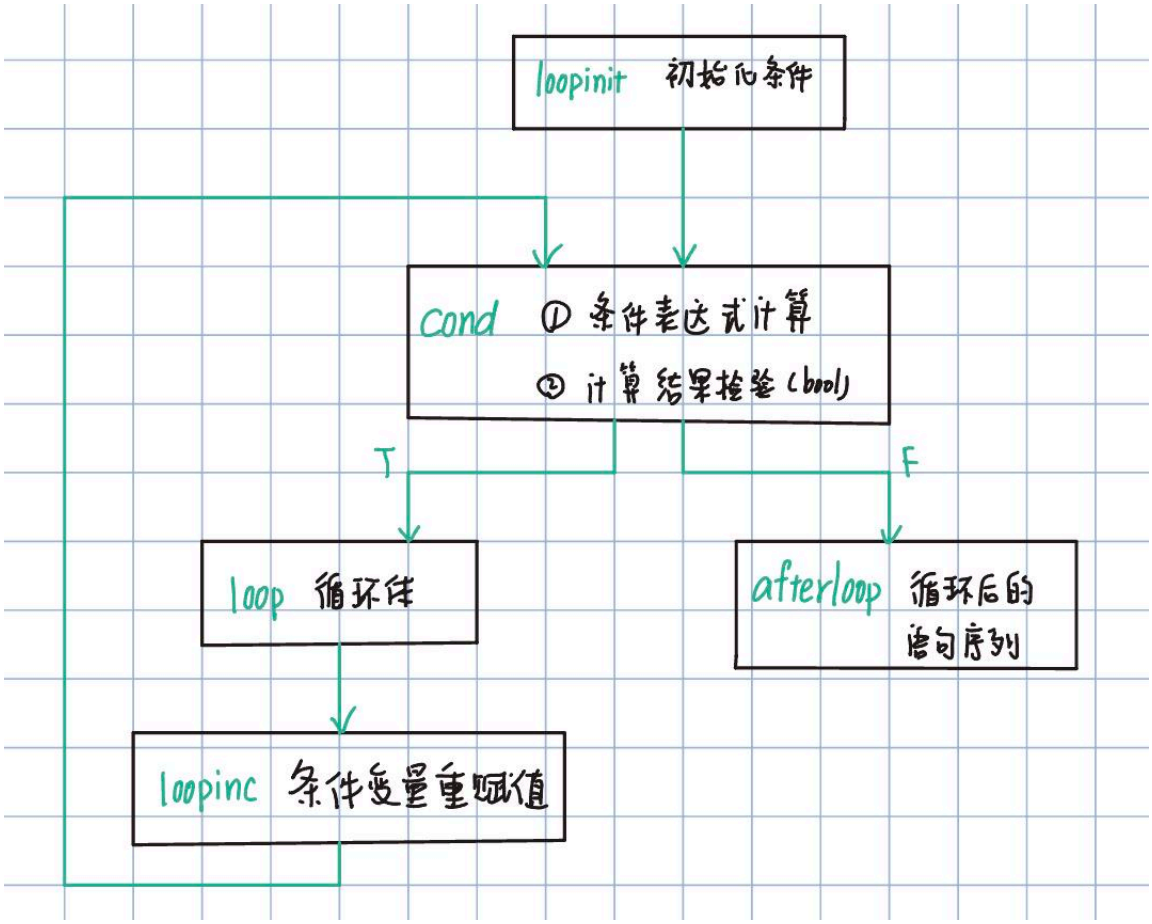
   检测是否重复定义，将类型与地址写入当前作用域中注册（栈顶符号表）即可
   - 局部变量：在当前作用域中注册（栈顶符号表）
   - 全局变量：在全局域中注册，未立即初始化则统一初始化为 0
6. While 循环处理




```
llvm::Value* WhileStatementNode::genCode(Context &curContext){
    // initiation
```

```
    llvm::Function *curFunc = curContext.currentFunc;
    llvm::BasicBlock *condBB = llvm::BasicBlock::Create(GlobalContext, "cond", curFunc);
    llvm::BasicBlock *loopBB = llvm::BasicBlock::Create(GlobalContext, "loop", curFunc);
    llvm::BasicBlock *afterBB = llvm::BasicBlock::Create(GlobalContext, "afterLoop", curFunc);
    AfterBBStack.push(afterBB);
    ContBBStack.push(condBB);

    // 跳转到 cond 判断是否进入循环
    Builder.CreateBr(condBB);
    Builder.SetInsertPoint(condBB);

    llvm::Value *condValue = expression.genCode(curContext);
    condValue = Builder.CreateICmpNE(condValue,
llvm::ConstantInt::get(llvm::Type::getInt1Ty(GlobalContext), 0, true), "whileCond");
    // true - 进入循环体, false - 进入后续操作
    auto branch = Builder.CreateCondBr(condValue, loopBB, afterBB);
    condBB = Builder.GetInsertBlock();

    // 处理循环体
    Builder.SetInsertPoint(loopBB);
    curContext.pushSymbolTable();
    block.genCode(curContext);
    if (curContext.hasReturn) {
        curContext.hasReturn = false;
    } else {
        Builder.CreateBr(condBB);
    }
    curContext.popSymbolTable();

    // 后续操作
    Builder.SetInsertPoint(afterBB);

    // 复位
    ContBBStack.pop();
    AfterBBStack.pop();

    return branch;
}
```

7. For 循环处理




```
llvm::Value* ForStatementNode::genCode(Context &curContext) {
    // initiation
    llvm::Function *curFunc = curContext.currentFunc;
    llvm::BasicBlock *initBB = llvm::BasicBlock::Create(GlobalContext, "loopinit", curFunc);
    llvm::BasicBlock *condBB = llvm::BasicBlock::Create(GlobalContext, "cond", curFunc);
    llvm::BasicBlock *loopBB = llvm::BasicBlock::Create(GlobalContext, "loop", curFunc);
    llvm::BasicBlock *incBB = llvm::BasicBlock::Create(GlobalContext, "loopinc", curFunc);
    llvm::BasicBlock *afterBB = llvm::BasicBlock::Create(GlobalContext, "afterloop", curFunc);
    AfterBBStack.push(afterBB);
    ContBBStack.push(incBB);

    // 跳到循环初始化操作
    Builder.CreateBr(initBB);
    Builder.SetInsertPoint(initBB);
    this->initiation.genCode(curContext);

    // 跳到循环条件判断
    Builder.CreateBr(condBB);
    Builder.SetInsertPoint(condBB);

    llvm::Value *condValue = this->condition.genCode(curContext);
    condValue = Builder.CreateICmpNE(condValue,
llvm::ConstantInt::get(llvm::Type::getInt1Ty(GlobalContext), 0 , true), "forCond");
    // ture - 跳进循环体
    auto branch = Builder.CreateCondBr(condValue, loopBB, afterBB);
    condBB = Builder.GetInsertBlock();
```

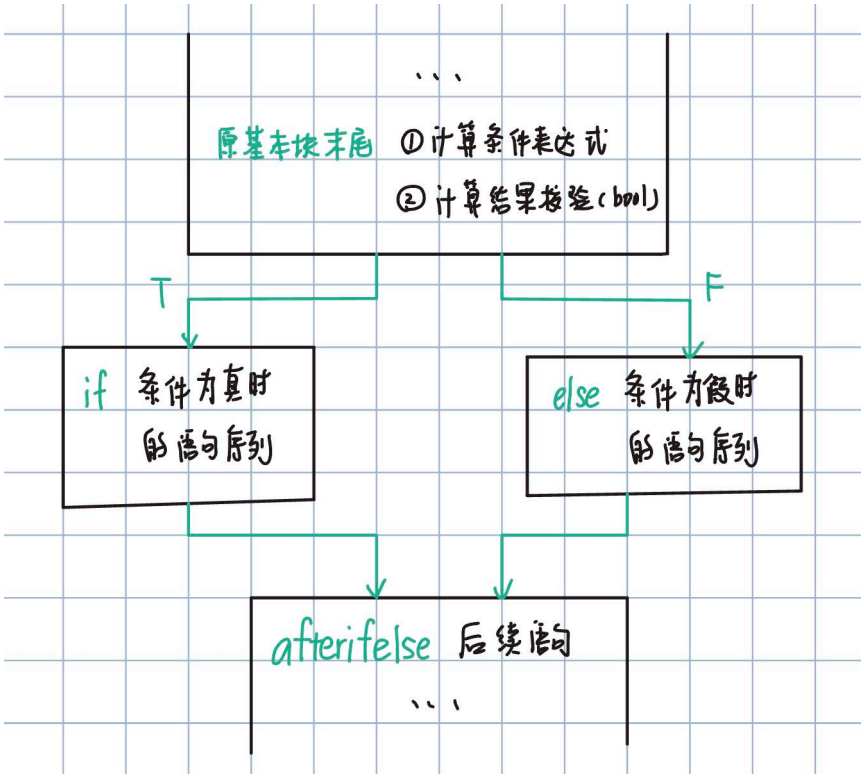
```
    // 处理 inc 操作（完成后回到 cond 进行判断）
    Builder.SetInsertPoint(incBB);
    this->increment.genCode(curContext);
    Builder.CreateBr(condBB);

    // 处理 循环体
    Builder.SetInsertPoint(loopBB);
    curContext.pushSymbolTable();
    this->body.genCode(curContext);
    if (curContext.hasReturn) {
        curContext.hasReturn = false;
    } else { // 先做 inc 再做 cond 判断
        Builder.CreateBr(incBB);
    }
    curContext.popSymbolTable();

    // 后续操作
    Builder.SetInsertPoint(afterBB);

    // 复位
    AfterBBStack.pop();
    ContBBStack.pop();

    return nullptr;
}
```

8. Break & Continue 语句处理
   两者均将 `hasReturn` 标记为 `ture` 以截断后续代码，不同的是：

- break 无条件跳转至 `AfterBBStack` 的栈顶基本块（最近的 `afterloop` 标签）
- continue 无条件跳转至 `CondBBStack` 的栈顶基本块：
  - 对于 For 循环来说，这是最近的 `loopinc` 标签（执行完毕后跳转至 `cond` 标签）
  - 对于 While 循环来说，这是最近的 `cond` 标签

9. IF-ELSE 语句处理

> IF 语句逻辑与之类似（仅缺少 `else` 标签，在此不做赘述）




```
llvm::Value* IfElseStatementNode::genCode(Context &curContext) {
    // initiation
    llvm::Function *curFunc = curContext.currentFunc;
    llvm::BasicBlock *IfBB = llvm::BasicBlock::Create(GlobalContext, "if", curFunc);
    llvm::BasicBlock *ElseBB = llvm::BasicBlock::Create(GlobalContext, "else", curFunc);
    llvm::BasicBlock *ThenBB = llvm::BasicBlock::Create(GlobalContext, "afterifelse", curFunc);

    // 跳转判断 (false 跳到 else)
    llvm::Value *condValue = expression.genCode(curContext);
    condValue = Builder.CreateICmpNE(condValue,
llvm::ConstantInt::get(llvm::Type::getInt1Ty(GlobalContext), 0, true), "ifCond");
    auto branch = Builder.CreateCondBr(condValue, IfBB, ElseBB);

    // cond = true
    Builder.SetInsertPoint(IfBB);
    curContext.pushSymbolTable();
    ifBlock.genCode(curContext);
    curContext.popSymbolTable();

    if (curContext.hasReturn) {
        curContext.hasReturn = false;
    } else {
        Builder.CreateBr(ThenBB);
    }

    // cond = false
    Builder.SetInsertPoint(ElseBB);
    curContext.pushSymbolTable();
    elseBlock.genCode(curContext);
    curContext.popSymbolTable();
```

```
    if (curContext.hasReturn) {
        curContext.hasReturn = false;
    } else {
        Builder.CreateBr(ThenBB);
    }

    // 返回后续操作
    Builder.SetInsertPoint(ThenBB);

    return branch;
}
```

3.3.4 BlockNode

- 顺序为块内的每一条语句生成对应代码即可；
- 但由于基本块内不得出现无条件跳转语句，因此忽略其后剩余的语句。

```
llvm::Value* BlockNode::genCode(Context &curContext) {
    // 处理 block 内的每一条 statement
    for (auto stmt : statementList) {
        (*stmt).genCode(curContext);
        if (curContext.hasReturn == true) {
            break;
        }
    }
    return nullptr;
}
```

4 目标代码生成

在这一步中，我们先使用 `llvm-as` 将之前生成的 LLVM IR 文件（`.ll`）转为 LLVM bitcode（`.bc`），随后使用 llc 将其编译为指定架构的汇编语言文件（`.s`）。

此处指定输出的汇编语言为 x86_64，相关指令如下：

```
llvm-as path/to/xxx.ll              # 之前生成的 IR 文件
llc --march=x86-64 path/to/xxx.bc   # 上一步生成的字节码文件
```

生成的 `.s` 文件即为目标语言汇编代码。

> 您可以直接在根路径下使用 `make test` 命令：这将在 `/test` 路径下生成对应的 `test.s` (x86_64 汇编) 文件，同时使用 GCC 编译生成可执行文件并运行。

5 测试案例和结果

5.1 数据类型测试

> 测试支持的 `int` `float` `char` 及数组类型

- 测试输入

```c
int main() {
    int i;

    float f = 3.7;
    printf("f = %f\n", f);

    char str[6];
    str[0] = 'h';
    str[1] = 'e';
    str[2] = 'l';
    str[3] = 'l';
    str[4] = 'o';
    str[5] = '\0';
    printf("Char Array str[] = %s\n", str);

    return 0;
}
```

- 测试输出

```
desktop/C-Minus-Compiler [ make test                                    rebase * ] 11:02 AM
make[1]: Entering directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
Preparing for test ...
Generating X86_64 ...
f = 3.700000
Char Array str[] = hello
make[1]: Leaving directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
```

5.2 表达式测试

> 同时测试了普通运算、对负数的运算、运算后赋值、自增操作

- 测试输入

```c
int main() {
    int i = 0;
    // +
    i = i + -1;
    printf("i = %d\n", i);
    // *
    i = i * -2;
    printf("i = %d\n", i);
    // -=
    i -= -6;
    printf("i = %d\n", i);
    // /=
```

```
    i /= 4;
    printf("i = %d\n", i);
    // ++
    i++;
    printf("i = %d\n", i);
    return 0;
}
```

- 测试输出

```
● desktop/C-Minus-Compiler [ make test                                    rebase * ] 11:28 AM
make[1]: Entering directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
Preparing for test ...
Generating X86_64 ...
i = -1
i = 2
i = 8
i = 2
i = 3
make[1]: Leaving directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
```

5.3 语句测试

5.3.1 IF 语句

- 测试输入

```
int main() {
    int i = 1;
    if (i == 1) {
        i *= 2;
    }
    printf("i = %d\n", i);
    return 0;
}
```

- 测试输出

```
● desktop/C-Minus-Compiler [ make test                                    rebase * ] 11:20 AM
make[1]: Entering directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
Preparing for test ...
Generating X86_64 ...
i = 2
make[1]: Leaving directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
```

5.3.2 IF-ELSE 语句

- 测试输入

```c
int main() {
    int i = 1;
    int j = -2;
    if (i == 2) {
        i /= 2;
        printf("i = %d\n", i);
    } else {
        printf("j = %d\n", j);
    }
    printf("After if-else ... \n");
    return 0;
}
```

- 测试输出

```
● desktop/C-Minus-Compiler [ make test                                    rebase * ] 11:23 AM
make[1]: Entering directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
Preparing for test ...
Generating X86_64 ...
j = -2
After if-else ...
make[1]: Leaving directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
```

5.3.3 WHILE 语句

> 同时测试负十六进制数识别

- 测试输入

```c
int main() {
    int i = -0xa;
    while (i <= 0) {
        printf("i = %d\n", i);
        i += 2;
    }
    return 0;
}
```

- 测试输出

```
● desktop/C-Minus-Compiler [ make test                                    rebase * ] 11:18 AM
make[1]: Entering directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
Preparing for test ...
Generating X86_64 ...
i = -10
i = -8
i = -6
i = -4
i = -2
i = 0
make[1]: Leaving directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
```

5.3.4 FOR 语句

> 同时测试 BREAK & CONTINUE 语句

- 测试输入

```
int main() {
    int i;
    for (i = 0 ; i < 7 ; i++) {
        if ( i == 3 ) {
            continue;
        }
        if ( i == 5) {
            break;
        }
        printf("i = %d\n", i);
    }
    return 0;
}
```

- 测试输出

```
desktop/C-Minus-Compiler [ make test                                    rebase * ] 11:11 AM
make[1]: Entering directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
Preparing for test ...
Generating X86_64 ...
i = 0
i = 1
i = 2
i = 4
make[1]: Leaving directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
```

5.4 函数测试

5.4.1 声明并定义

- 测试输入

```
void HelloWorld(int i) {
    printf("Hello, recv 'i' = %d\n", i);
    return;
}

int main() {
    int i = 233;
    HelloWorld(i);
    return 0;
}
```

- 测试输出

```
● desktop/C-Minus-Compiler [ make test                                    rebase * ] 11:02 AM
make[1]: Entering directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
Preparing for test ...
Generating X86_64 ...
Hello, recv 'i' = 233
make[1]: Leaving directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
```

5.4.2 先声明后定义

> 同时测试了函数的先声明后定义 & 局部变量同名覆盖

- 测试输入

```c
void HelloWorld();
int ii = 111;

int main() {
    printf("'ii' = %d\n", ii);
    HelloWorld();
    return 0;
}

void HelloWorld() {
    int ii = 222;
    printf("Hello, 'ii' = %d\n", ii);
    return;
}
```

- 测试输出

```
● desktop/C-Minus-Compiler [ make test                                    rebase * ] 11:09 AM
make[1]: Entering directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
Preparing for test ...
Generating X86_64 ...
'ii' = 111
Hello, 'ii' = 222
make[1]: Leaving directory '/mnt/c/users/SeaBee/desktop/C-Minus-Compiler/test'
```

6 组内成绩分配表

组员姓名	百分比
沈韵泓	100%