浙江大学

本科实验报告

课程名称：	编译原理
成员：	庞懿非（3190104534） 游浩然（3190104795） 张羽翔（3180106335）
学　　院：	计算机科学与技术学院
系：	计算机科学与技术专业
专　　业：	计算机科学与技术专业
指导教师：	冯雁

2022 年 5 月 21 日

0 序言

概述：

本实验基于c++语言在LLVM架构上实现了一个类c语言的编译系统，本系统以符合本编译器定义语言规范(详见词法分析与语法分析章节)的文本文件为输入，输出LLVM的中间代码，可通过clang的命令将中间代码生成为目标代码。该编译器的设计实现涵盖词法分析、语法分析、语义分析、中间代码生成、中间代码优化、代码生成、测试等阶段和环节，所使用的具体技术包括但不限于：

1. c++实现词法分析、语法分析、语义分析；
2. 在LLVM架构下实现中间代码生成、代码优化、目标代码生成。

开发环境：

1. 操作系统：Linux

   发行版：Ubuntu 18.04

   虚拟机：VMware WorkStations 16 Player Pro

2. 编译环境：

   clang 13.0.0

   LLVM 13.0.0

3. 编辑器：VS Code

文件说明：

工程目录解析:

- 文法.txt：实现的CFG语法以及简单描述。具体描述可见于实验报告。
- hpp文件：
  - **util.hpp**：词法分析、语法分析部分的头文件，定义了词法语法分析中的类、枚举常量，声明分析方法。
  - **ParExp1.hpp**：声明了语法分析类到中间代码生成的接口以及用于语义分析的全局变量。
  - **codegen1.hpp**：代码生成部分头文件，声明了和语言结构相关的众多类以及用于**LLVM**生成中间代码的全局变量和方法。
- cpp文件：
  - **token.cpp**：使用DFA进行词法分析。
  - **praser.cpp**：递归下降的进行CFG解析、语法分析，使用LL(1)算法。
  - **ParExp1.cpp**：定义了语法分析生成的AST到后续设计的类之间的接口，并且进行语义分析（符号表+类型检查）。
  - **codegen1.cpp**：基于**LLVM**实现了众多类的代码生成函数。main函数在这个文件中，进行DFA构造、token读取、语法树生成、语义分析、转换为新定义类的语法树、中间代码生成、（中间代码优化）所有过程的函数调用。
  - **codegen_opt.cpp**：在codegen1.cpp的基础上设计了优化模块代码以及嵌入其他函数的方法。
- **Makefile**：可以使用make来生成可执行文件，可执行文件有两种类型，Name类型为不带任何优化的版本，Name1类型为带有嵌入C函数，带有5个可选优化的版本（根据LLVM每个优化pass各自独立的设计，优化类型可以任意组合）；你可以通过编辑Name的值指定生成文件的名字，请勿同时在多个线程执行make命令。（相关内容已经下载Makefile中）
- **input**：input文件为默认的自定义语法的输入文件（即一种CFG文法的推导结果），程序将读取文件并且检查错误，无错误则生成中间代码。
- inputrecord：一些input可选的内容以及历史记录。
- **可执行文件**：

    - final_ori11：未带优化的可执行文件，以input文件为输入，输出中间代码或者输出错误信息到终端。
    - final_optwithC11：带优化的可执行文件（勾选了实验中4.1 4.2 4.3的优化，其他优化可以通过打开注释重新make），嵌入`printi(__int16_t)`函数声明打印整数，其他同上，具体请参看实验报告第5章。
- test文件夹
    - testn：实验报告中第n个测试样例的代码。
    - testn_ori：testn中的代码放入input文件，执行final_ori11输出的中间代码。
    - testn_opt：testn中的代码放入input文件，执行final_optwithC11输出的优化过的中间代码。
- 其他文件
    - printi.c：定义了printi函数，使用编译命令

      ```
      clang -S -emit-llvm printi.c -o printi.ll
      ```

      生成可读LLVM字节码文件printi.ll，后续可与本程序执行生成的中间代码共同编译生成可执行文件。（该文件完全是为了嵌入打印功能）

      ```
      clang printi.ll output.ll -o output
      ```

      生成的output可以执行并且进行打印。

组员分工：

组里有一些特殊情况，张羽翔同学在完成了自己的一部分任务后，与5月13日开始没有任何贡献，5月16日开始不读消息，所以组员分工可能会比较细致展开。具体情况可见于总结。

任务	时间	完成人员
文法讨论与确定	5月10日	庞懿非 张羽翔 游浩然
分工讨论	5月10日	庞懿非 张羽翔 游浩然
LLVM内容学习（环境配置、基础概念、中间代码、接口调用、优化方式、运行时环境等）	5月10-16日	庞懿非 游浩然
基于lex的词法分析（后废弃）	5月11日	张羽翔
词法分析和语法分析	5月13日凌晨	张羽翔（后失联）
构建中间代码生成类	5月16-17日	庞懿非
完成中间代码生成的基本内容（5月18日获得了中间代码输出）	5月17-18日	庞懿非
调整原先的语法分析树，并且进行一些改造和适配	5月16日	庞懿非 游浩然
接口设计（原先的语法分析树到中间代码类）	5月16-18	游浩然
样例测试和结果运行	5月18-19日	游浩然 庞懿非
工程debug和整体调试	5月18-22日	庞懿非 游浩然
嵌入优化模块	5月19-20日	庞懿非
优化模块调试和优化结果分析	5月20-22日	庞懿非 游浩然
规范样例设计与测试	5月20日	游浩然
加入简化的打印函数声明（使得生成的文件可以自己打印输出整数）	5月20日	庞懿非
补充语义分析	5月21日	庞懿非
实验报告撰写	5月19-22日	游浩然（主要） 庞懿非（补充）
实验报告格式调整与补充	5月22日	庞懿非 游浩然
工程文档的整理与调整	5月22日	庞懿非 游浩然

1 词法分析

词法分析是计算机科学中将字符序列转换为标记（token）序列的过程。在词法分析阶段，编译器读入源程序字符串流，将字符流转换为标记序列，同时将所需要的信息存储，然后将结果交给语法分析器。

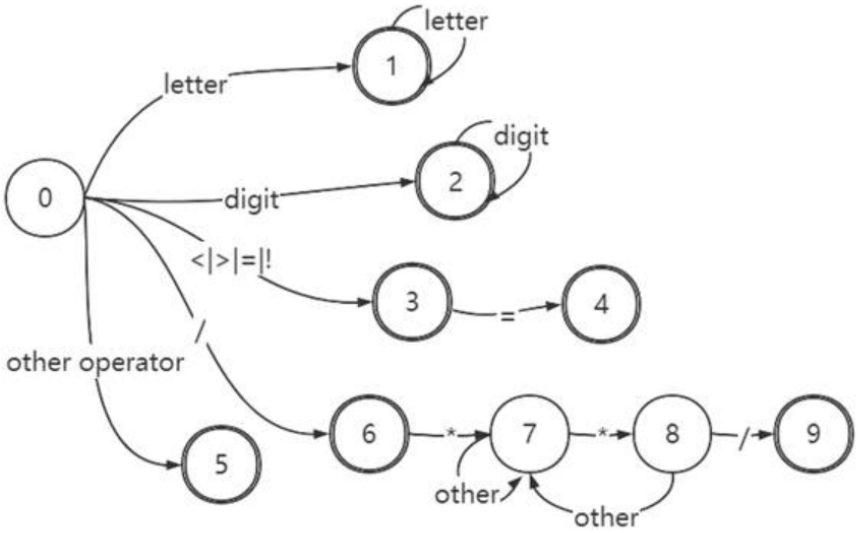
词法分析采用c++代码来实现，构造DFA并根据DFA返回token的类型和值：

1. **字符类别判断函数**

```
bool IsLetter(char ch)
{
    return 'A' <= ch && ch <= 'Z' || 'a' <= ch && ch <= 'z';
}

bool IsDigit(char ch)
{
    return '0' <= ch && ch <= '9';
}

bool IsOtherOp(char ch)
{
    switch (ch)
    {
    case '+': case '-': case '*':
    case ':':
    case ',':
    case '(': case ')':
    case '[': case ']':
    case '{':case '}':
    case ';':
        return true;
    default:
        return false;
    }
}

bool IsBlank(char ch)
{
    return ch==' '||ch=='\t';
}

bool IsOther(char ch)
{
    return true;
}
```

2. **识别token的DFA**

   其中状态1表示的是识别到英文字符串。状态2表示的是识别到数字。状态3表示的是运算符(<,>,=,!)，状态4表示的是运算符（<=,>=,==,!=）。状态6到状态9的路径表示识别注释。状态5表示其他操作。



3. **DFA中的状态类型**

`SUNACCEPTED`：表示不接受。

`SID`：表示identifier。

`SRESERVED`：表示保留字。

`SNUMBER`：表示数字。

`SOPERATOR`：表示操作符。

`SCOMMENT`：表示注释。

`SENDFILE`：表示输入文件的结束。

```
enum StateType        //dfa中的状态类型
{
    SUNACCEPTED = 0,
    SID,
    SRESERVED,
    SNUMBER,
    SOPERATOR,
    SCOMMENT,
    SENDFILE
};
```

4. **DFA中的节点和边**

`Edge`：表示节点的一条边，`action`表示是否符合状态转移的条件（通过函数判断），`transition`表示在该条件下的下一个状态。

`Node`：表示一个节点，`type`表示节点的状态，`edges`表示这个节点的所有边。

```
typedef bool(*ACTION)(char);
struct Edge            //构建dfa的edge
{
    int transition;
    ACTION action;

    Edge(int transition, ACTION action)
```

```
    {
        this->transition = transition;
        this->action = action;
    }
};

struct Node            //节点
{
    StateType type;
    std::list<Edge> edges;
};
```

5. **Tokentype**

```
enum TokenType        //token类型定义
{
    ENDFILE=0,
    IF,ELSE,INT,RETURN,VOID,WHILE,
    ID,NUMBER,
    ASSIGN,LT,GT,EQ,NEQ,GEQ,LEQ,

PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,LBRACKET,RBRACKET,LBRACE,RBRACE,COMMA,SE
MI,
};
```

其中`LPAREN`表示'('，`LBRACKET`表示'['，`LBRACE`表示'{'。以R开头的表示对应的右括号。`COMMA`表示','，`SEMI`表示';'。完整如下：

```
    Token2String[IF]="if";
    Token2String[ELSE]="else";
    Token2String[INT]="int";
    Token2String[VOID]="void";
    Token2String[WHILE]="while";
    Token2String[RETURN]="return";
    Token2String[ASSIGN]="=";
    Token2String[LT]="<";
    Token2String[GT]=">";
    Token2String[EQ]="==";
    Token2String[NEQ]="!=";
    Token2String[GEQ]=">=";
    Token2String[LEQ]="<=";
    Token2String[PLUS]="+";
    Token2String[MINUS]="-";
    Token2String[TIMES]="*";
    Token2String[OVER]="/";
    Token2String[LPAREN]="(";
    Token2String[RPAREN]=")";
    Token2String[LBRACKET]="[";
    Token2String[RBRACKET]="]";
    Token2String[LBRACE]="{";
    Token2String[RBRACE]="}";
    Token2String[SEMI]=";";
    Token2String[COMMA]=",";
```

6. **Token类**

`line`：用于记录token的位置。

`value`：用于保存token的值。

`type`：用于记录token的类型。

```
struct Token       //token
{
    int line;
    std::string value;
    TokenType type;
    Token(){
        line=1;
        type=INT;
        value.reserve(100);
    }
    Token(const int line,const std::string& value, TokenType type)
    {
        this->line = line;
        this->value = value;
        this->type = type;
    }
};
```

7. **构建DFA**

```
vector<Node> DFA;
void BuildDFA()       //构建dfa
{
    DFA.clear();

    Node node0;
    node0.type = StateType::SUNACCEPTED;
    node0.edges.emplace_back(1, IsLetter);
    node0.edges.emplace_back(2, IsDigit);
    node0.edges.emplace_back(3, [](char ch) {return ch == '<' || ch == '>'
|| ch == '=' || ch == '!'; });
    node0.edges.emplace_back(5, IsOtherOp);
    node0.edges.emplace_back(6, [](char ch) {return ch == '/'; });
    DFA.push_back(node0);

    Node node1;
    node1.type = StateType::SID;
    node1.edges.emplace_back(1, IsLetter);
    DFA.push_back(node1);

    Node node2;
    node2.type = StateType::SNUMBER;
    node2.edges.emplace_back(2, IsDigit);
    DFA.push_back(node2);

    Node node3;
    node3.type = StateType::SOPERATOR;
```

```
    node3.edges.emplace_back(4, [](char ch) {return ch == '='; });
    DFA.push_back(node3);

    Node node4;
    node4.type = StateType::SOPERATOR;
    DFA.push_back(node4);

    Node node5;
    node5.type = StateType::SOPERATOR;
    DFA.push_back(node5);

    Node node6;
    node6.type = StateType::SOPERATOR;
    node6.edges.emplace_back(7, [](char ch) {return ch == '*'; });
    DFA.push_back(node6);

    Node node7;
    node7.type = StateType::SUNACCEPTED;
    node7.edges.emplace_back(8, [](char ch) {return ch == '*'; });
    node7.edges.emplace_back(7, IsOther);

    DFA.push_back(node7);

    Node node8;
    node8.type = StateType::SUNACCEPTED;
    node8.edges.emplace_back(9, [](char ch) {return ch == '/'; });
    node8.edges.emplace_back(7, IsOther);
    DFA.push_back(node8);

    Node node9;
    node9.type = StateType::SCOMMENT;
    DFA.push_back(node9);

}
```

构造的过程中用到了`emplace_back()`。`emplace_back()`是c++11的新特性。和`push_back()`的区别在于`push_back()`方法要调用构造函数和复制构造函数，这也就代表着要先构造一个临时对象，然后把临时的copy构造函数拷贝或者移动到容器最后面。而`emplace_back()`在实现时，则是直接在容器的尾部创建这个元素，省去了拷贝或移动元素的过程。

- **构造节点0**：如果读到字母转移到1，如果读到数字转移到2，读到四个特殊的操作符转移到3，读到'/'转移到6，读到其他字符则转移到5。

```
 Node node0;
node0.type = StateType::SUNACCEPTED;
node0.edges.emplace_back(1, IsLetter);
node0.edges.emplace_back(2, IsDigit);
node0.edges.emplace_back(3, [](char ch) {return ch == '<' || ch == '>'
|| ch == '=' || ch == '!'; });
node0.edges.emplace_back(5, IsOtherOp);
node0.edges.emplace_back(6, [](char ch) {return ch == '/'; });
DFA.push_back(node0);
```

- **构造节点1**：节点的类型设置为SID。如果读到字母转移到1。

```
 Node node1;
node1.type = StateType::SID;
node1.edges.emplace_back(1, IsLetter);
DFA.push_back(node1);
```

- **构造节点2**：节点的类型设置为SNUMBER。如果读到数字转移到2。

```
 Node node2;
node2.type = StateType::SNUMBER;
node2.edges.emplace_back(2, IsDigit);
DFA.push_back(node2);
```

- **构造节点3**：节点的类型设置为SOPERATOR。如果读到等于号转移到4。

```
 Node node3;
node3.type = StateType::SOPERATOR;
node3.edges.emplace_back(4, [](char ch) {return ch == '='; });
DFA.push_back(node3);
```

- **构造节点4**：节点的类型设置为SOPERATOR。

```
 Node node4;
node4.type = StateType::SOPERATOR;
DFA.push_back(node4);
```

- **构造节点5**：节点的类型设置为SOPERATOR。

```
 Node node5;
node5.type = StateType::SOPERATOR;
DFA.push_back(node5);
```

- **构造节点6**：节点的类型设置为SOPERATOR。如果读到'*'转移到节点7。

```
 Node node6;
node6.type = StateType::SOPERATOR;
 node6.edges.emplace_back(7, [](char ch) {return ch == '*'; });
DFA.push_back(node6);
```

- **构造节点7**：节点的类型设置为SUNACCEPTED。如果读到'*'转移到节点8，否则仍然停留在7。

```
 Node node7;
node7.type = StateType::SUNACCEPTED;
 node7.edges.emplace_back(8, [](char ch) {return ch == '*'; });
node7.edges.emplace_back(7, IsOther);
DFA.push_back(node7);
```

- **构造节点8**：节点的类型设置为SUNACCEPTED。如果读到'/'转移到节点9，否则回到节点7。

```
Node node8;
 node8.type = StateType::SUNACCEPTED;
node8.edges.emplace_back(9, [](char ch) {return ch == '/'; });
 node8.edges.emplace_back(7, IsOther);
DFA.push_back(node8);
```

- **构造节点9**：节点的类型设置为SCOMMENT。

```
Node node9;
node9.type = StateType::SCOMMENT;
DFA.push_back(node9);
```

8. **状态类转Token类（根据状态的类型返回token类型）**

```
Token StateTpye2TokenType(int line,string token,StateType state){       //状态类型转token
    switch (state)
    {
    case StateType::SID:
        if(String2Token[token]>0){
            return Token(lineNumber,token,String2Token[token]);
        }else{
            return Token(lineNumber,token,TokenType::ID);
        }
    case StateType::SNUMBER:
        return Token(lineNumber,token,TokenType::NUMBER);
    case StateType::SENDFILE:
        return Token(lineNumber,token,TokenType::ENDFILE);
    default:
        return Token(lineNumber,token,String2Token[token]);
    }
}
```

9. **获取token**

设置初始状态为0.如果读到的字符符合状态转移的条件，将当前状态转移到对应的状态。具体实现方法遍历节点的每一条边，如果`action`=true，说明符合条件，转移到这条边对应的`transition`。

如果不符合条件，先判断当前是否在状态0。若不在状态0：若字符不是空格，将其用ungetc回退到输入流。若状态不是注释的结束，调用`StateTpye2TokenType`返回对应的`Token`。否则说明是注释完成，将token清空，状态置0。

如果是在状态0，且字符不是空格、回车、换行，说明出现了词法错误。

```
Token GetToken(){           //获取token
    int state = 0;
    string token="";
    while (true){
        char ch = fgetc(stdin);
        colNumber++;
        bool flag = false;
        for (auto it : DFA[state].edges){
```

```
            if (it.action(ch)){
                flag = true;
                state = it.transition;
                token.push_back(ch);
                break;
            }
        }
        if (!flag){
            if (DFA[state].type != StateType::SUNACCEPTED){
                if (!IsBlank(ch)){
                    ungetc(ch,stdin);
                }
                if (DFA[state].type != StateType::SCOMMENT){
                    return
StateTpye2TokenType(lineNumber,token,DFA[state].type);
                }else{
                    token.clear();
                    state=0;
                }
            }else if ((ch!='\n'&&ch!='\t'&&ch!=' '&&ch != EOF)){
                ShowLexError(token);
            }
        }
        if (ch == '\n'){
            lineNumber++;
            colNumber = 0;
        }else if (ch == EOF){
            return
StateTpye2TokenType(lineNumber,token,StateType::SENDFILE);
        }
    }
}
```

2 语法分析

在计算机科学和语言学中，语法分析是根据某种给定的形式文法对由单词序列（如英语单词序列）构成的输入文本进行分析并确定其语法结构的一种过程。在语法分析阶段，编译器接收词法分析器发送的标记序列，最终输出抽象语法树数据结构。

1. **本实验基于C-设计了LL(1)的文法**

```
program → declaration-list

declaration-list →  declaration declaration-list | declaration

declaration → var-declaration|fun-declaration

var-declaration → type-specifier ID | type-specifier ID[NUM];

type-specifier → int|void

fun-declaration → type-specifier ID(params) compound-stmt
```

```
params → params-list|void

param-list → parm param-list | param

param → type-specifier ID

compound-stmt → {local-declarations statements}

local-declarations → var-declaration local-declarations | empty

statement-list → statement | {statements}

statement → expression-stmt|compound-stmt|selection-stmt|iteration-
stmt|return-stmt

statements → statement+

expression-stmt → expression; | ;

selection-stmt → if(expression) statement-list | if(expression) statement-
list else statement-list

iteration-stmt → while(expression) statement-list

return-stmt → return ;| return expression;

expression → var=expression|simple-expression

var → ID|ID[NUM]

simple-expression → additive-expression | additive-expression relop
additive-expression

relop → <=|<|>|>=|==|!=

additive-expression → term | term addop term

addop → +|-

term → factor | factor mulop factor

mulop → *|/

factor → (expression)|var|call|NUM

call → ID(args)

args → arg-list|empty

arg-list → expression,arg-list | expression
```

对以上每条文法规则，给出了相关语义的简短解释。

程序由声明的列表（或序列）组成，声明可以是函数或变量声明，顺序是任意的。至少必须有一个声明。接下来是语义限制。所有的变量和函数在使用前必须声明（这避免了向后引用，即 backpatching）。

```
program → declaration-list

declaration-list →  declaration declaration-list | declaration

declaration → var-declaration | fun-declaration
```

变量声明或者声明了简单的整数类型变量，或者是基类型为整数的数组变量，索引范围从0到NUM-1。语法中仅有的基本类型是整型和空类型。在一个变量声明中，只能使用类型指示符int。void用于函数声明。每个声明只能声明一个变量。

```
var-declaration → type-specifier ID | type-specifier ID[NUM];

type-specifier → int|void
```

函数声明由返回类型指示符、标识符以及在圆括号内的用逗号分开的参数列表组成，后面跟着一个复合语句，是函数的代码。如果函数的返回类型是 void，那么函数不返回任何值（即是一个过程）。函数的参数可以是void（即没有参数），或者一列整型参数。简单的整型参数由值传递。一个函数参数的作用域等于函数声明的复合语句，函数的每次请求都有一个独立的参数集。函数可以是递归的（对于使用声明允许的范围）。

```
fun-declaration → type-specifier ID(params) compound-stmt

params → params-list|void

param-list → parm param-list | param

param → type-specifier ID
```

复合语句由用花括号围起来的一组声明和语句组成。复合语句通过用给定的顺序执行语句序列来执行。局部声明的作用域等于复合语句的语句列表，局部变量在使用时优先于全局变量。

```
compound-stmt → {local-declarations statements}
```

局部变量声明可以为空。

```
local-declarations → var-declaration local-declarations | empty

statement-list → statement | {statements}
```

statement有多种类型，statements是大于等于1个statement的组合。

```
statement → expression-stmt|compound-stmt|selection-stmt|iteration-
stmt|return-stmt
statements → statement+
```

表达式语句以用于赋值、计算和函数调用等操作。

```
expression-stmt → expression; | ;
```

if语句有通常的语义：表达式进行计算；非0值引起第一条语句的执行； 0值引起第二条语句的执行，如果它存在的话。这个规则导致了典型的悬挂else二义性，可以用一种标准的方法解决：else部分通常作为当前if的一个子结构立即分析(“最近嵌套”非二义性规则)。

```
selection-stmt → if(expression) statement-list | if(expression) statement-
list else statement-list
```

while语句重复执行表达式，并且如果表达式的求值为非0，则执行语句，当表达式的值为0时结束。

```
iteration-stmt → while(expression) statement-list
```

返回语句可以返回一个值也可无值返回。函数类型不为void就必须返回一个值。

```
return-stmt → return ;| return expression;
```

表达式是一个变量引用，后面跟着赋值符号（等号）和一个表达式，或者就是一个简单的表达式。赋值有通常的存储语义：找到由var表示的变量的地址，然后由赋值符右边的子表达式进行求值，子表达式的值存储到给定的地址。这个值也作为整个表达式的值返回。var是简单的（整型）变量或下标为常数的数组变量。

```
expression → var=expression|simple-expression

var → ID|ID[NUM]
```

简单表达式由无结合的关系操作符组成 （即无括号的表达式仅有一个关系操作符）。简单表达式在它不包含关系操作符时，其值是加法表达式的值，或者如果关系算式求值为true，其值为1，求值为false时值为0。

```
simple-expression → additive-expression | additive-expression relop
additive-expression

relop → <=|<|>|>=|==|!=
```

加法表达式和项表示了算术操作符的结合性和优先级。除号表示整数除；即任何余数都被截去。

```
additive-expression → term | term addop term

addop → +|-

term → factor | factor mulop factor

mulop → *|
```

factor是围在括号内的表达式；或一个变量，求出其变量的值；或者一个函数调用，求出函数的返回值；或者一个NUM，其值由扫描器计算。

```
factor → (expression)|var|call|NUM
```

函数调用的组成是一个ID(函数名)，后面是用括号围起来的参数。参数或者为空，或者由逗号分割的表达式列表组成，表示在一次调用期间分配的参数的值。函数在调用之前必须声明，声明中参数的数目必须等于调用中参数的数目。

```
call → ID(args)

args → arg-list|empty

arg-list → expression, arg-list | expression
```

2. **语法分析由c++实现**

   节点的类型:

```
enum TreeNodeType    //语法树中的节点类型

{
  Func,Int,Id,Params,Param,Comp,Const,Call,Args,Void,Var_Decl,Arry_Decl,Arry
_Elem,Assign,Op,
  Selection_Stmt,Iteration_Stmt,Return_Stmt,Statements
};
```

   数据结构:

```
struct TreeTokenNode{
    char val[30];
    TokenType type;
    TreeTokenNode(){
        memset(val,0,sizeof(val));
    }
};
struct TreeNode           //树节点
{
    struct TreeNode *child[4];
    struct TreeNode *brother;
    TreeNodeType TreenodeType;
    TreeTokenNode token;
};
```

其中`TreeNode`为一个节点，共有四个儿子节点（可以为空）以及一个兄弟节点（也可以为空）。`TreeNodeType`表示节点的类型。`TreeTokenNode`用于保存节点的类型信息，`val`用于保存字符信息，`Tokentype`表示Token的类型。

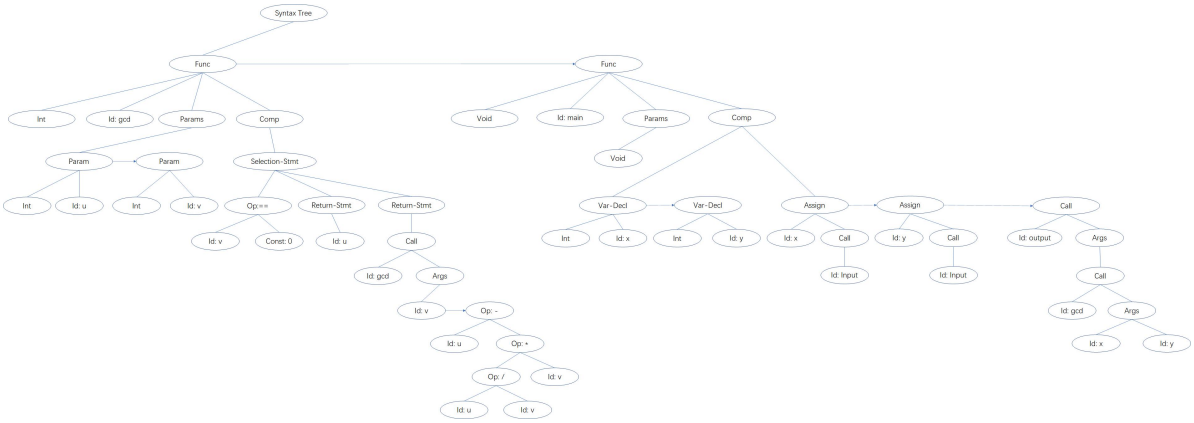
文法分析的过程中，根据设计的文法以及获取到的token来构造语法分析树的节点，整体思想是采用topdown的递归构造的方式来实现。下面是启动构建语法分析树部分的代码，其余部分的具体实现可在**praser.cpp**中查看。

```
//declaration_list->declaration_list{declaration}
TreeNode * declaration_list()
{
    TreeNode * root = declaration();
    TreeNode * now =root;
    while(GlobalToken.type==INT||GlobalToken.type==VOID){        //只有两种类型
声明，int 和 void
        TreeNode *nxt = declaration();
        if (nxt!=NULL){
            now->brother=nxt;
            now=nxt;
```

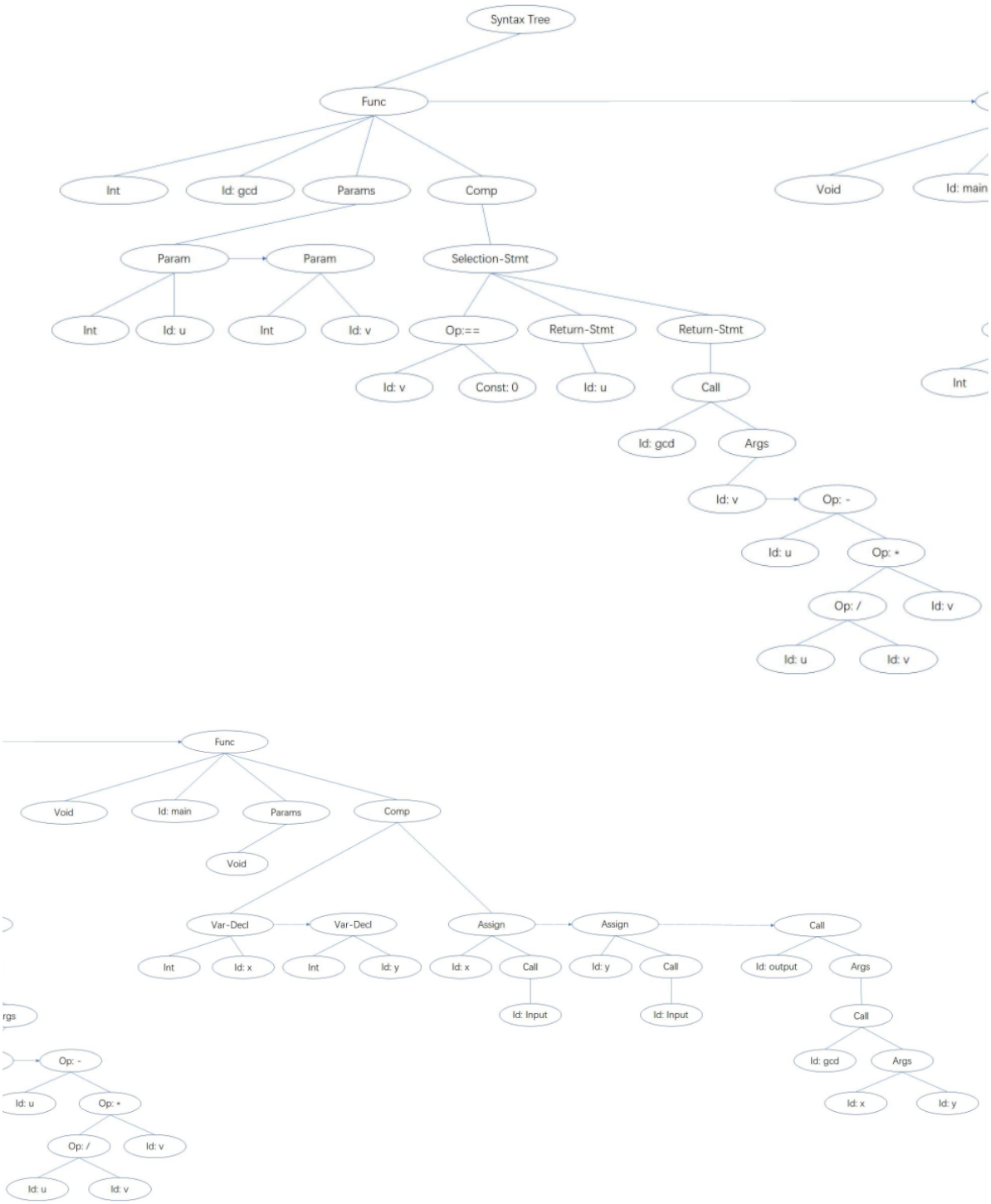
```
        }
    }
    match(TokenType::ENDFILE);
    return root;
}
void ParseSyntax(/*TreeNode *rootS*/)      //语法树分析
{
    cout<<"Syntax Tree:"<<endl;
    GlobalToken = GetToken();
    root1 = declaration_list();
    PreOrder(root1);
    //return root;
}
```

语法分析树样例:

```
int gcd (int u,int v)
{
    if (v == 0)
        return u ;
    else
        return gcd(v,u-u/v*v);
        /* u-u/v*v == u mod v */
}

void main(void)
{
    int x;
    int y;
    x = input();
    y = input();
    output(gcd(x,y));
}
```



放大图:



3 中间代码生成

3.1 LLVM概述

LLVM是架构编译器的框架系统，以C++编写而成，包含一系列模块化的编译器组件和工具链，用来开发编译器前端和后端。LLVM用于优化任意程序语言编写的程序的编译时间（compile-time）、链接时间（link-time）、运行时间（run-time）以及空闲时间（idle-time）。

LLVM的一大特色就是，有着独立的、完善的、严格约束的中间代码表示。这种中间代码，就是LLVM的字节码，是LLVM抽象的精髓，前端生成这种中间代码，后端进行各类优化分析，用LLVM开发的编译器只需要处理前端，也能用上后端的结构和最先进的后端优化技术。

前端Frontend

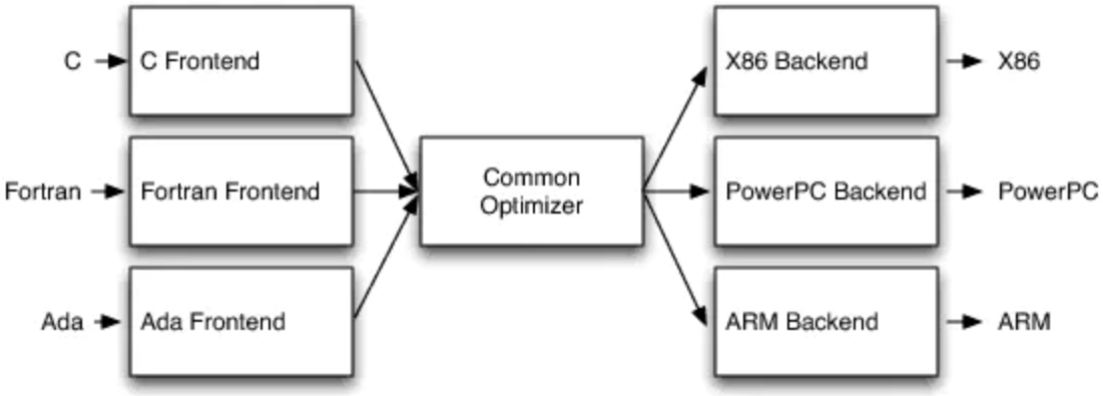
编译器前端的任务是解析源代码（编译阶段），它会进行词法分析、语法分析、语义分析、检查源代码是否存在错误，然后构建抽象语法树（Abstract Syntax Tree AST），LLVM的前端还会生成中间代码（intermediate representation，简称IR），可以理解为LLVM架构是编译器 + 优化器 + 后端实现，优化器接收的是IR中间代码，输出的是优化后的IR，经过后端翻译成目标指令集。

优化器 Optimizer

优化器负责进行各种优化，改善代码的运行时间，减小生成代码数量，例如消除冗余计算、CFG简化、冗余块消除、内存映射优化等。

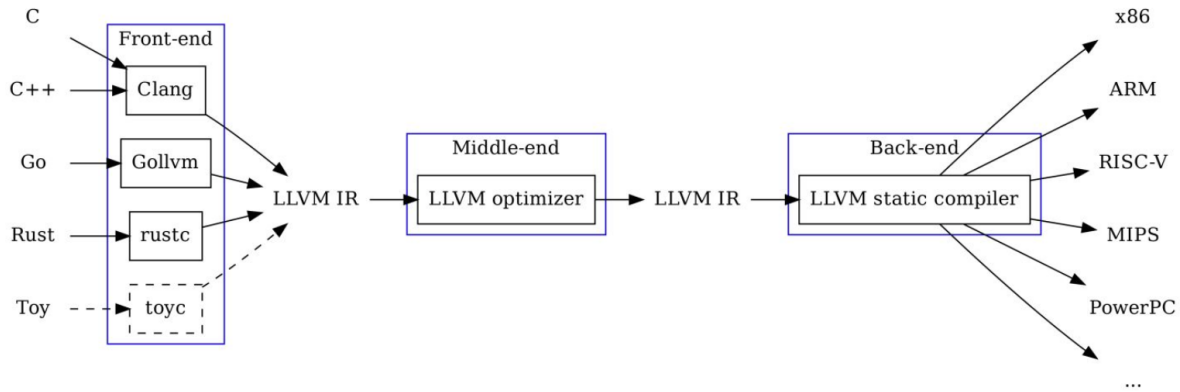
后端 Backend（代码生成器 Code Generator）

将代码映射到目标指令集，生成机器代码，并且进行机器代码相关的代码优化。LLVM将中间代码优化这个流程做到了极致，LLVM工具链，不但可以生成所支持的各个后端平台的代码，更可以方便的将各语言的前端编译后的模块链接到一起，你可以方便的在你的语言中调用C函数。



3.2 LLVM IR概述

LLVM IR是LLVM的核心所在，通过将不同高级语言的前端变换成LLVM IR进行优化、链接后再传给不同目标的后端转换成为二进制代码，前端、优化、后端三个阶段互相解耦，这种模块化的设计使得LLVM优化不依赖于任何源码和目标机器。

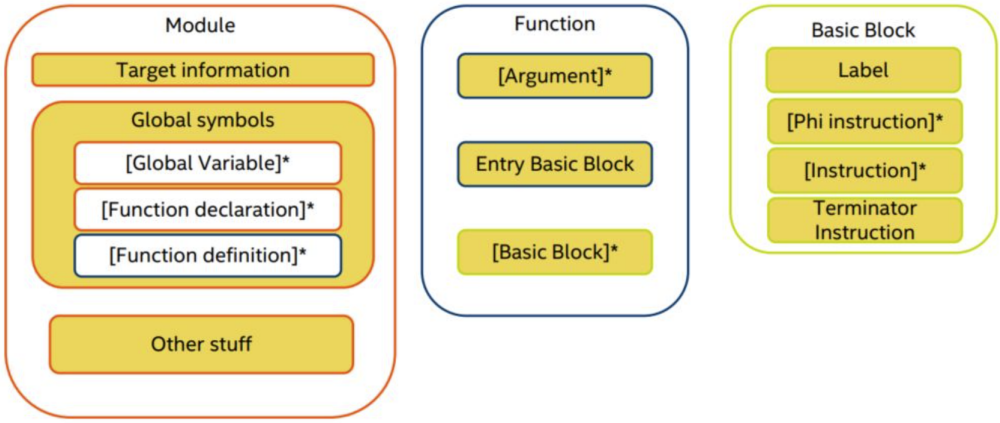


IR布局：

每个IR文件称为一个Module，它是其他所有IR对象的顶级容器，包含了目标信息、全局符号和所依赖的其他模块和符号表等对象的列表，其中全局符号又包括了全局变量、函数声明和函数定义。

函数由参数和多个基本块组成，其中第一个基本块称为entry基本块，这是函数开始执行的起点，另外LLVM的函数拥有独立的符号表，可以对标识符进行查询和搜索。

每一个基本块包含了标签和各种指令的集合，标签作为指令的索引用于实现指令间的跳转，指令包含Phi指令、一般指令以及终止指令等。



IR上下文环境：

LLVM::Context：提供用户创建变量等对象的上下文环境，尤其在多线程环境下至关重要。它是一个不透明的对象，拥有许多核心 LLVM 数据结构，例如类型和常量值表。

LLVM::IRBuilder：提供创建LLVM指令并将其插入基础块的API。Builder对象是一个帮助对象，可以轻松生成 LLVM 指令。IRBuilder类模板的实例跟踪插入指令的当前位置，并具有创建新指令的方法。

3.3 LLVM IR生成

运行环境设计

LLVM IR的生成依赖上下文环境，我们使用了全局变量来保存环境，在递归遍历AST节点的时候以函数为单位调用`codegen`函数进行每个节点的IR生成。

环境配置：

全局的上下文变量和构造器变量以及模块实例

```
std::unique_ptr<LLVMContext> TheContext;
std::unique_ptr<Module> TheModule;
std::unique_ptr<IRBuilder<>> Builder;
```

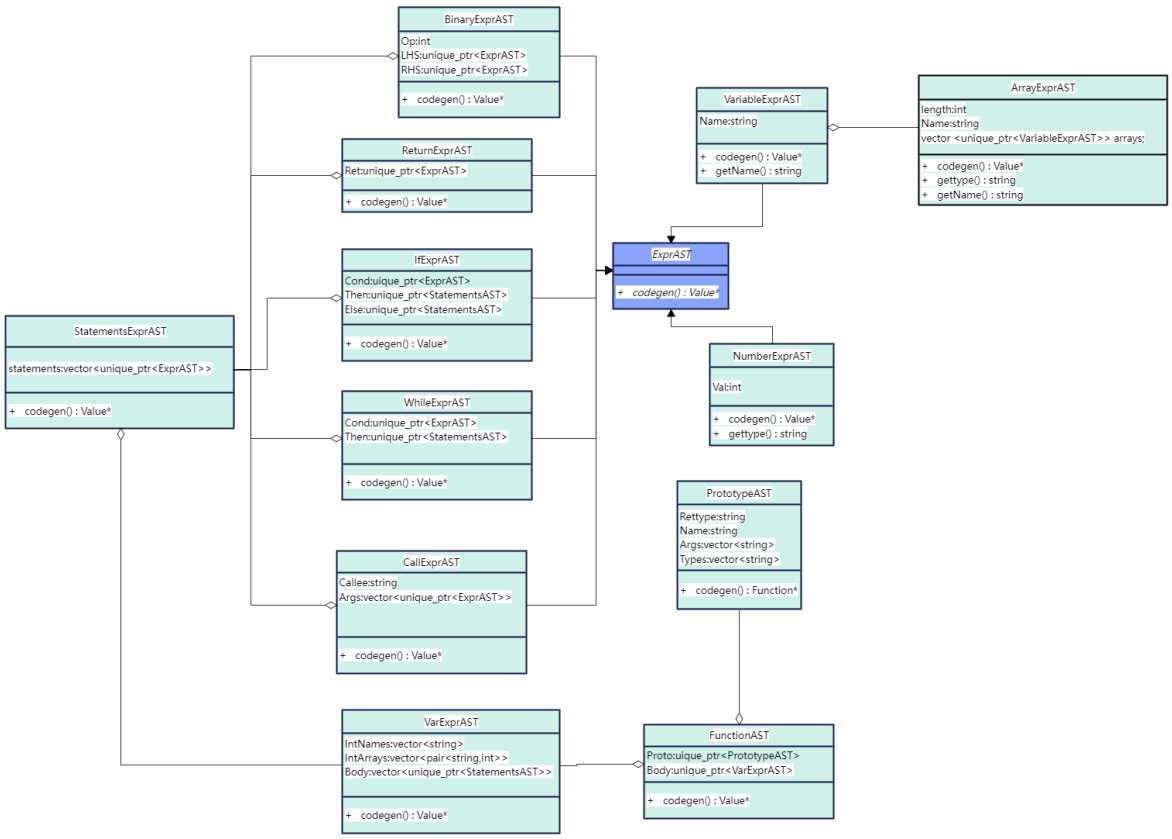
符号表使用`TheModule`中存储的符号表

通过下述语句获取全局变量地址和函数指针。

```
TheModule->getGlobalVariable(name);
TheModule->getFunction(name);
```

类的设计

类图如下所示：



0. `ExprAST`：定义所有语句类的父类。定义了虚函数`codegen`用于生成目标代码，返回值类型为llvm::Value*，存储语句执行的结果（返回值）。

```
/* 0 */
/* Expression AST - Base class for all expression nodes. */
class ExprAST {
    public:
    virtual ~ExprAST() = default;
    /* virtual function for code generation */
    virtual Value *codegen() = 0;
};
```

1. `NumberExprAST`：数字类，由于文法只允许整型数字，使用整型变量`Val`存储数值，在程序中整数使用`signed int16`。

```
/* 1 */
/* NumberExprAST - Expression class for numeric literals like "1". */
class NumberExprAST : public ExprAST {
    int Val;
    public:
    NumberExprAST(int Val) : Val(Val) {}
    string getType(){ return "int";}
    Value *codegen() override;
};
```

2. `VariableExprAST`：表示变量的类。含有变量`Name`存储变量名，后续通过变量名获取存储地址。

```
/* 2 */
/* VariableExprAST - Expression class for referencing a variable, like "a".
*/
class VariableExprAST : public ExprAST {
    std::string Name;
    public:
    VariableExprAST(const std::string &Name) : Name(Name) {}
    Value *codegen() override;
    const std::string &getName() const { return Name; }
};
```

3. `ArrayExprAST`：表示数组的类。`length`表示数组大小，`Name`表示数组名称。在程序中，Array是通过一连串结构类似变量组成的，并不是传统意义上的指针和偏移量的数组，这样设计主要是为了简化逻辑。

```
/* 3 */
/* ArrayExprAST
   this can be initialized with all 0, by making a variableExprAST vector
   with value 0 NUM times, element of the array is stored as NUM variables,
   so it is not the true array and the pointer, just an array of variables
   with almost the same outlook "ArrayName[index]",  like "a[0]". */
class ArrayExprAST : public ExprAST {
    int length;
    std::string Name;
    vector <std::unique_ptr<VariableExprAST>> arrays;

    public:
    ArrayExprAST(int NUM, string Name);
    string getType(){ return "int_array";}
    Value *codegen() override;
    const std::string &getName() const { return Name; }
};
```

4. `BinaryExprAST`：表示二元表达式的类。含有整型变量`Op`存储操作符类型，具体数值对应于“util.h"中的token枚举值，变量`LHS`和`RHS`指向操作符两边的表达式。

```
/* 4 */
/* BinaryExprAST - Expression class for a binary operator. */
class BinaryExprAST : public ExprAST {
    int Op;
    std::unique_ptr<ExprAST> LHS, RHS;

    public:
    BinaryExprAST(int op, std::unique_ptr<ExprAST> LHS,
                  std::unique_ptr<ExprAST> RHS)
        : Op(op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}
    Value *codegen() override;
};
```

5. `StatementsExprAST`：表示语句组的类。含有变量`statements`存储一系列语句，对应非终结符的statements，具体可由CFG中的条件判断语句、循环语句、返回语句、函数调用语句和简单语句组成。

```
/* 5 */
/* StatementsAST - corresponding to statement-lists */
class StatementsAST : public ExprAST {
    vector <std::unique_ptr<ExprAST>> statements;

    public:
    StatementsAST(vector <std::unique_ptr<ExprAST>>
statements1):statements(std::move(statements1)){}

    Value *codegen() override;
};
```

6. `IfExprAST`：表示if语句的类。含有变量`Cond` `Then` `Else`指向条件语句与两个分支语句，判断条件是普通表达式，两个分支都可以是语句组statements。

```
/* 6 */
/* IfExprAST - Expression class for if/then/else. */
class IfExprAST : public ExprAST {
    std::unique_ptr<ExprAST> Cond;
    std::unique_ptr<StatementsAST>  Then, Else;

    public:
    IfExprAST(std::unique_ptr<ExprAST> Cond, std::unique_ptr<StatementsAST>
Then,
              std::unique_ptr<StatementsAST> Else)
        : Cond(std::move(Cond)), Then(std::move(Then)),
Else(std::move(Else)) {}

    Value *codegen() override;
};
```

7. `WhileExprAST`：表示循环语句的类。含有变量`Cond` `Then`指向条件语句与循环语句。

```
/* 7 */
/* WhileExprAST--while is somehow similar to if else if condition
   then compound-statement else exit may be reused */
class WhileExprAST : public ExprAST {
    std::unique_ptr<ExprAST> Cond;
    std::unique_ptr<StatementsAST> Then;

    public:
    WhileExprAST(std::unique_ptr<ExprAST> Cond,
std::unique_ptr<StatementsAST> Then)
        : Cond(std::move(Cond)), Then(std::move(Then)) {}

    Value *codegen() override;
};
```

8. `ReturnExprAST`：表示return语句的类。`Ret`表示return的语句。

```
/* 8 */
/* ReturnExprAST */
class ReturnExprAST : public ExprAST {
    std::unique_ptr<ExprAST> Ret;

    public:
    ReturnExprAST(std::unique_ptr<ExprAST> Ret_expression)
        : Ret(std::move(Ret_expression)) {}
    Value *codegen() override;
};
```

9. `CallExprAST`：表示调用函数的类。变量`Callee`是调用的函数名，`Args`代表函数参数组。

```
/* 9 */
/* CallExprAST - Expression class for function calls. */
class CallExprAST : public ExprAST {
    std::string Callee;
    std::vector<std::unique_ptr<ExprAST>> Args;

    public:
    CallExprAST(const std::string &Callee,
                std::vector<std::unique_ptr<ExprAST>> Args)
        : Callee(Callee), Args(std::move(Args)) {}
    Value *codegen() override;
};
```

10. `PrototypeAST`：表示函数声明的类。`RetType`表示函数的返回类型，为void或者int型，`Name`表示函数名，`Args`表示参数名，`Types`表示参数类型，现阶段只支持`int`类型的参数，与`Args`一一对应。

```
/* 10 */
/* PrototypeAST - This class represents the "prototype" for a function,
   which captures its name, and its argument types and names (thus
implicitly
   the number of arguments the function takes).*/
class PrototypeAST {
    std::string RetType;
    std::string Name;
    std::vector<std::string> Args;
    std::vector<std::string> Types;

    public:
    PrototypeAST(const std::string rettype, const std::string &name,
std::vector<std::string> Args,
                 std::vector<std::string> Types)
        : RetType(rettype),Name(name), Args(std::move(Args)),
Types(std::move(Types)) {}

    const std::string &getName() const { return Name; }
    Function *codegen();
};
```

11. `VarExprAST`：表示函数体的类，由局部变量声明和执行语句组成。成员变量`IntNames`表示局部整型变量，`IntArrays`表示局部整型数组，`Body`表示函数体中的语句组。

```
/* 11 */
/*  functionbody
    local declaration and statements */
class VarExprAST : public ExprAST {
    std::vector<std::string> IntNames;
    std::vector<std::pair<std::string,int>> IntArrays;
    std::unique_ptr<StatementsAST> Body;

    public:
    VarExprAST(
        std::vector<std::string> Names,
std::vector<std::pair<std::string,int>> Arrays,
        std::unique_ptr<StatementsAST> Body)
        : IntNames(std::move(Names)), IntArrays(std::move(Arrays)),
Body(std::move(Body)) {}

    Value *codegen() override;
};
```

12. `FunctionAST`：表示函数的类，由函数声明、局部变量定义以及执行语句组组成。`Proto`表示函数声明，`Body`表示函数体（包括局部变量定义以及执行语句组）。

```
/* 12 */
/* FunctionAST - This class represents a function definition itself. */
class FunctionAST {
    std::unique_ptr<PrototypeAST> Proto;
    std::unique_ptr<VarExprAST> Body;

    public:
    FunctionAST(std::unique_ptr<PrototypeAST> Proto,
                std::unique_ptr<VarExprAST> Body)
        : Proto(std::move(Proto)), Body(std::move(Body)) {}
    Function *codegen();
};
```

代码生成描述

代码生成其实也是一个递归调用的过程，通过一系列基本类的代码生成的调用从而得到整个中间代码。这里所说的基本类就是最简单的表达式，对应于设计的类中的`NumberExprAST VariableExprAST BinaryExprAST IfExprAST WhileExprAST ReturnExprAST CallExprAST`等类，其他类通过调用这些类构建自己的模块。

1. `NumberExprAST`：在 LLVM中，数字常量用`ConstantInt`类表示，该类在内部保存数值在`LLVMContext`中，并且通过`APInt`获取存储的整数。这段代码在IR中直接体现为整数。

```
Value *NumberExprAST::codegen() {
  return ConstantInt::get(*TheContext, APInt(16,Val));
}
```

2. `VariableExprAST`：变量对应的代码生成是去获取变量存储的值，通过在局部变量和全局变量寻找对应的变量的存储位置，然后通过load操作获取值，对应的IR代码则为load。

```
Value *VariableExprAST::codegen(){
```

```
    /* get local variabls' address */
    AllocaInst *A = NameIntValues[Name];
    if (!A){
        /* no local variable can be get, get global variable */
        GlobalVariable *B = GetValue(Name);
        if (!B)
            /* no variable */
            return LogErrorV1("Unknown variable name");
        else
            /* load the value from the address and return */
            /* this creates a statement in IR */
            return Builder->CreateLoad(B->getValueType(),B, Name);
    }
    return Builder->CreateLoad(A->getAllocatedType(), A, Name.c_str());
}
```

3. `BinaryExprAST`（包括`Assign`）也被归入这个类型当中。
    - 如果是赋值操作，且左边是一个变量，则对等号的右边进行`codegen`，将获取的值用store操作存储在变量名对应的地址中。

```
if (Op == ASSIGN) {
    /* Assignment requires the LHS to be an identifier.
       This assume we're building without RTTI because LLVM builds that
way by
       default.  If you build LLVM with RTTI this can be changed to a
       dynamic_cast for automatic error checking. */
    VariableExprAST *LHSE = static_cast<VariableExprAST *>(LHS.get());
    if (!LHSE)
      return LogErrorV1("destination of '=' must be a variable");
    /* Codegen the RHS. */
    Value *Val = RHS->codegen();
    if (!Val)
      return nullptr;

    /* Look up the name. */
    Value *Variable = NameIntValues[LHSE->getName()];
    if (!Variable){
      Variable = GetValue(LHSE->getName());
      if (!Variable)
      return LogErrorV1("Unknown variable name");
    }
    /* type casting, the RHS's result may be i1, casting it to i16,
       mention that signed i1 only has 0 and -1, so we need set false as
       unsigned i1 can be 0 or 1 */
    Val = Builder->CreateIntCast(Val, Type::getInt16Ty(*TheContext),
false);

    /* this creates a statement in IR, storing the value to the address
*/
    Builder->CreateStore(Val, Variable);
    return Val;
}
```

- 如果是二元运算符，LLVM的`IRBuilder`集成了丰富的二元操作接口，包括`+ - * / > >= < <= == !=`等，根据Op的类型调用对应的接口创建了语句并返回计算结果，注意：比较操作的结果是1位int（即bool）在对变量赋值的时候要做类型转换，因为LLVM IR只支持同类型数据的操作。

```
Value *BinaryExprAST::codegen() {
  if (Op == ASSIGN) {
   /* above */
  }

  /* if not ASSIGN, both sides are expression */
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;
  /* create statement according to Op, its value it enum in "util.hpp"
*/
  switch (Op) {
  case PLUS:
    return Builder->CreateAdd(L, R, "addtmp");
  case MINUS:
    return Builder->CreateSub(L, R, "subtmp");
  case TIMES:
    return Builder->CreateMul(L, R, "multmp");
  case OVER:
    return Builder->CreateSDiv(L, R, "multmp");
  case LT:
    L = Builder->CreateICmpSLT(L, R, "lttmp");break;
  case GT:
    L = Builder->CreateICmpSGT(L, R, "gttmp");break;
  case EQ:
    L = Builder->CreateICmpEQ(L, R, "eqtmp");break;
  case NEQ:
    L = Builder->CreateICmpNE(L, R, "neqtmp");break;
  case GEQ:
    L = Builder->CreateICmpSGE(L, R, "geqtmp");break;
  case LEQ:
    L = Builder->CreateICmpSLE(L, R, "leqtmp");break;

  default:
    return LogErrorV1("invalid binary operator");
  }
  return L;
}
```

4. `ArrayExprAST`：数组的实现做了很大的简化，主要原因是我对指针的操作并不熟悉所以在设计之初就打算用多个变量的方式代替数组来进行计算。在后续的实现中证明，这种方法是非常不好的。在我的实现中，比如定义了`int a[2]`，我使用两个变量`a[0] a[1]`来存储数组元素，在访问的时候也只允许在数组的index处使用整数。

```
/* array generation */
Value *ArrayExprAST::codegen() {
    for( int i = 0 ; i < length; i++){
      Value * eachVariable = arrays[i]->codegen();
      if(!eachVariable){
        string error = "this element of array:" + Name + "does not exist.";
        return LogErrorV1(error.c_str());
        break;
      }
    }
    /* return 0 */
    return Constant::getNullValue(Type::getInt16Ty(*TheContext));
}
```

5. `StatementsExprAST`：对`vector`中的每一条语句调用`codegen`进行生成。

```
Value *StatementsAST::codegen() {
    for( int i = 0 ; i < statements.size(); i++){
      Value * eachVariable = statements[i]->codegen();
      if(!eachVariable){
        return LogErrorV1("this statement is wrong.");
        break;
      }
    }
    //return 0
    return Constant::getNullValue(Type::getInt16Ty(*TheCotext));
}
```

6. `IfExprAST` if语句被分为四个基本块：

   `ifcond`为条件判断表达式，根据返回值和0是否相同决定跳转方向。

   `then`是条件为真执行的基础块，插入到函数代码中并且生成其所含语句组的中间代码，最后无条件跳转到`merge`块。

   `else`是条件为假执行的基础块，在`then`代码生成结束之后插入到函数中并且生成其所含语句组的中间代码，最后无条件跳转到`merge`块。

   `merge`是语句结束之后的基础块，作为后面代码的插入点，可以插入函数的其他语句。

```
Value *IfExprAST::codegen() {
  Value *CondV = Cond->codegen();
  if (!CondV)
    return nullptr;

  /* Convert condition to a bool by comparing non-equal to 0 */
  CondV = Builder->CreateICmpNE(
      CondV, ConstantInt::get(*TheContext, APInt(1,0)), "ifcond");//?

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

  /* Create blocks for the then and else cases.  Insert the 'then' block at
the
     end of the function now to generate code in this block. */
  BasicBlock *ThenBB = BasicBlock::Create(*TheContext, "then", TheFunction);
  BasicBlock *ElseBB = BasicBlock::Create(*TheContext, "else");
  BasicBlock *MergeBB = BasicBlock::Create(*TheContext, "ifcont");
```

```
  /* conditional jump */
  Builder->CreateCondBr(CondV, ThenBB, ElseBB);

  /* then generate code in Then block */
  Builder->SetInsertPoint(ThenBB);

  Value *ThenV = Then->codegen();
  if (!ThenV)
    return nullptr;

  /* unconditional jump to merge block */
  Builder->CreateBr(MergeBB);

  /* Codegen of 'Then' can change the current block, update ThenBB for the
PHI.
     needed if there is a PHI */
  ThenBB = Builder->GetInsertBlock();

  /* push Else block in the blocklist of the funciton */
  TheFunction->getBasicBlockList().push_back(ElseBB);
  Builder->SetInsertPoint(ElseBB);

  Value *ElseV = Else->codegen();
  if (!ElseV)
    return nullptr;

  Builder->CreateBr(MergeBB);

  /* Codegen of 'Else' can change the current block, update ElseBB for the
PHI. */
  ElseBB = Builder->GetInsertBlock();

  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder->SetInsertPoint(MergeBB);
  /* the block should not be empty so we will insert sth in another
statement codegen */
  return Constant::getNullValue(Type::getInt16Ty(*TheContext));
}
```

7. `WhileExprAST`：while语句与if语句的逻辑类似，但没有else块，如果条件表达式为假直接跳转到merge块。`then`块末尾重新进行判断和跳转。

```
Value *WhileExprAST::codegen(){
  Value *CondV = Cond->codegen();
  if (!CondV)
    return nullptr;

  /* Convert condition to a bool by comparing non-equal to 0 */
  CondV = Builder->CreateICmpNE(
      CondV, ConstantInt::get(*TheContext, APInt(1,0)), "ifcond");

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

  /* Create blocks for the then case.  Insert the 'then' block at the
     end of the function. */
  BasicBlock *ThenBB = BasicBlock::Create(*TheContext, "then", TheFunction);
```

```
  BasicBlock *MergeBB = BasicBlock::Create(*TheContext, "ifcont");

  Builder->CreateCondBr(CondV, ThenBB, MergeBB);

  Builder->SetInsertPoint(ThenBB);

  Value *ThenV = Then->codegen();
  if (!ThenV)
    return nullptr;
  /* check the condition again in Then block */
  CondV = Cond->codegen();
  if (!CondV)
    return nullptr;

  CondV = Builder->CreateICmpNE(
      CondV, ConstantInt::get(*TheContext, APInt(1,0)), "ifcond");

  Builder->CreateCondBr(CondV, ThenBB, MergeBB);

  ThenBB = Builder->GetInsertBlock();

  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder->SetInsertPoint(MergeBB);

  return Constant::getNullValue(Type::getInt16Ty(*TheContext));
}
```

8. `ReturnExprAST`：return语句对Ret的表达式进行`codegen`代码生成，并创建返回语句。

```
Value *ReturnExprAST::codegen(){
Value * RetVal = Ret->codegen();
    /* no return, return 0 to prevent error */
    if(!RetVal){
      RetVal = Constant::getNullValue(Type::getInt16Ty(*TheContext));
    }
    return Builder->CreateRet(RetVal);
}
```

9. `CallExprAST`：函数调用语句先从`TheModule`中查找被调用的函数，然后判断参数数量是否相等。符合调用条件则将参数值分别进行代码生成并压入，最后调用IRbuilder的`CreateCall`创建函数调用语句。

```
Value *CallExprAST::codegen() {
  /* Look up the name in the global module table. */
  Function *CalleeF = TheModule->getFunction(Callee);
  if (!CalleeF)
    return LogErrorV1("Unknown function referenced");

  /* If argument mismatch error. */
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV1("Incorrect # arguments passed");

  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    /* push the argument value in */
    /* do not allow arrays */
```

```
    ArgsV.push_back(Args[i]->codegen());
    if (!ArgsV.back())
      return nullptr;
  }

  return Builder->CreateCall(CalleeF, ArgsV, "calltmp");
}
```

10. `PrototpreAST`：该类作为函数声明，遍历`Types`将其转化成`LLVM::Type`并保存到参数类型列表中。根据返回值类型、参数列表以及参数是否可变长创建函数类型`FunctionType`，然后根据函数类型、函数名在模块中创建`Function`，最后设置各个参数的名字。生成的函数原型会存储在LLVM模块`TheModule`中。

```
Function *PrototypeAST::codegen() {
  /* the vector for Type * of the function */
  std::vector<Type *> Arguments;
  for(vector<std::string>::iterator it = Types.begin(); it != Types.end();
it++){
    if(*it == "int"){
      Arguments.push_back(Type::getInt16Ty(*TheContext));
    }
    else if(*it == "int_array"){
      Arguments.push_back(Type::getInt16PtrTy(*TheContext));
      /* not set the size, do not know if there will be errors */
      /* only use pointer of i16 iin this place, do not allow array as
arguments of functions */
    }
  }

  /* generate functiontype with return value set i16 for simpilication, the
senmatics checks
     is down before */
  /* return value, arguments type, the number of arguments can not be
changed ---false */
  FunctionType *FT =
      FunctionType::get(Type::getInt16Ty(*TheContext), Arguments, false);

  /*generate function in the Module with the name and type and
externallinkage */
  Function *F =
      Function::Create(FT, Function::ExternalLinkage, Name,
TheModule.get());

  /* Set names for all arguments. */
  unsigned Idx = 0;
  for (auto &Arg : F->args())
    Arg.setName(Args[Idx++]);

  return F;
}
```

11. `VarExprAST`：对应函数主体部分。首先对声明的局部变量分配地址空间并且初始化为0，并且将原先已经存在的同名局部变量地址通过`oldBindings`存储起来。分配完成后对函数内的语句组进行代码生成（调用`Statements::codegen()`），生成结束后再将`OldBindings`中保存的旧值恢复到全局变量`NameIntValues`中。

```
Value *VarExprAST::codegen() {
  /* record the old values before this codegen, just like storing value in
the stack */
  std::vector<AllocaInst *> OldBindings;

  /* get the function* */
  Function *TheFunction = Builder->GetInsertBlock()->getParent();

  /* Register all variables and emit their initializer. */
  for (unsigned i = 0, e = IntNames.size(); i != e; ++i) {

    const std::string &VarName = IntNames[i];
    /*initialize to be 0 */
    Value *InitVal;
    InitVal = ConstantInt::get(*TheContext, APInt(16,0));

    /* allocate the space for the new local variable */
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
    Builder->CreateStore(InitVal, Alloca);

    /* Remember the old variable binding so that we can restore the binding
when
       we call anothe function. */
    OldBindings.push_back(NameIntValues[VarName]);

    /* Remember the new binding. */
    NameIntValues[VarName] = Alloca;
  }

  /* array there is the same to variables */
  for (unsigned i = 0, e = IntArrays.size(); i != e; ++i) {
    /* similar operation to above */
  }
  /* above is initialization */

  /* Codegen the body, now that all vars are in scope after senmatics */
  Value *BodyVal = Body->codegen();
  if (!BodyVal)
    return nullptr;

  /* Pop all our variables from scope and restore the allocate space */
  for (unsigned i = 0, e = IntNames.size(); i != e; ++i)
    NameIntValues[IntNames[i]] = OldBindings[i];

  for (unsigned i = 0, e = IntArrays.size(); i != e; ++i) {
    /* similar to above */
  }
  /* Return the body computation value, which is useless in fact */
  return BodyVal;
}
```

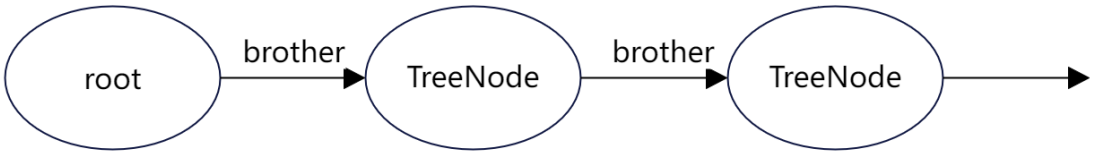
12. `FunctionAST`：先在`TheModule`中查找函数名，若不存在则生成函数声明。创建一个新的`entry`基本块作为代码的插入点。对每个参数分配空间，接着同样需要将先前与参数同名的变量地址存到`oldBindings`中。随后进行函数体代码的生成（即调用`VarExprAST::codegen()`）。最后后再将`oldBindings`中保存的值恢复到`NameIntValues`中。

```
Function *FunctionAST::codegen() {

  /* get the function* from the Module according to the name */
  Function *TheFunction = TheModule->getFunction(Proto->getName());

  if (!TheFunction)
    TheFunction = Proto->codegen();

  if (!TheFunction)
    return nullptr;

  /* Create a new basic block to start the functions' insertion into. */
  BasicBlock *BB = BasicBlock::Create(*TheContext, "entry", TheFunction);
  Builder->SetInsertPoint(BB);

  std::vector<AllocaInst *> OldBindings;
  std::vector<StringRef> VarNames;
  int number = 0;
  /* allocate the space for the argument and put the value to them */
  for (auto &Arg : TheFunction->args()) {
    StringRef VarName = Arg.getName();
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
    Builder->CreateStore(&Arg, Alloca);

    OldBindings.push_back(NameIntValues[std::string(VarName)]);
    /* Remember this new binding. */
    NameIntValues[std::string(VarName)] = Alloca;
    VarNames.push_back(VarName);
    number ++;
  }

  if (Value *RetVal = Body->codegen()) {

    /* Finish off the function. */
    /* this create return is used for ensuring the blocks' safety */
    Builder->CreateRet(RetVal);

    /* Validate the generated code, checking for consistency. */
    verifyFunction(*TheFunction);

    return TheFunction;
  }

  /* restoring the value */
  for (unsigned i = 0; i < number; ++i)
    NameIntValues[std::string(VarNames[i])] = OldBindings[i];

  /* Error reading body, remove function from the Module. */
  TheFunction->eraseFromParent();
  return nullptr;
}
```

3.4 接口设计

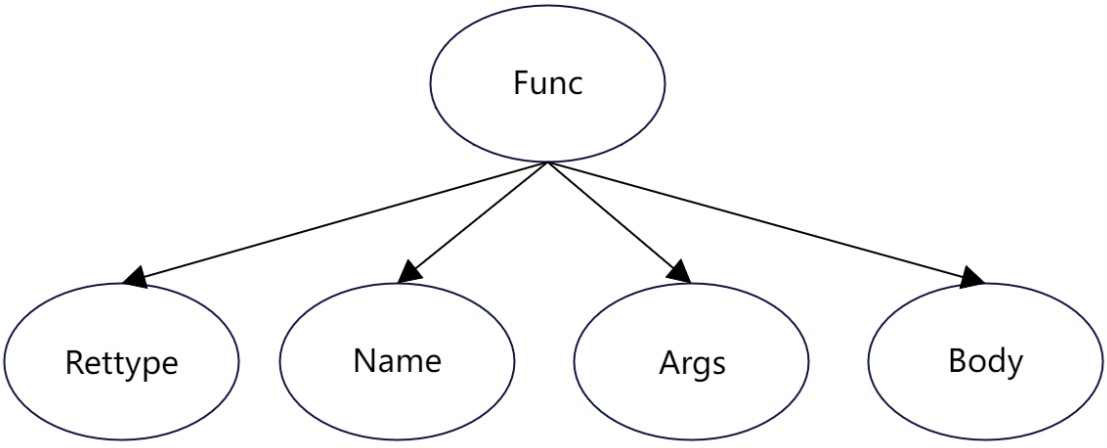
从语法分析树的根节点递归向下遍历，生成具体的类。根据节点类型的不同调用不同函数进行生成。

1. root为整个程序的第一个声明，节点类型可能是`Func`或`Var_Decl`。root节点的brother为下一个声明。




```
void ParseTree(TreeNode *root){             //parse a tree
    if(!root) {
        fprintf(stderr, "root is null\n");
        return;
    }
    TreeNode * tmp = root;
    while(tmp!=nullptr){
        if(tmp->TreenodeType == Func){
            HandleDefinition(tmp);
        }
        else if(tmp->TreenodeType == Var_Decl){
            HandleTopLevelExpression(tmp);
        }
        tmp = tmp->brother;
    }
}
```

2. 如果结点类型为`Func`，为函数的定义。调用函数`ParseFuncExpr`生成`FunctionAST`类结构`FnAST`,再调用其中的`codegen`函数进行代码生成。`Func`节点的结构如下，下面的节点依次为`Func`的四个儿子节点`child[0]-child[3]`。

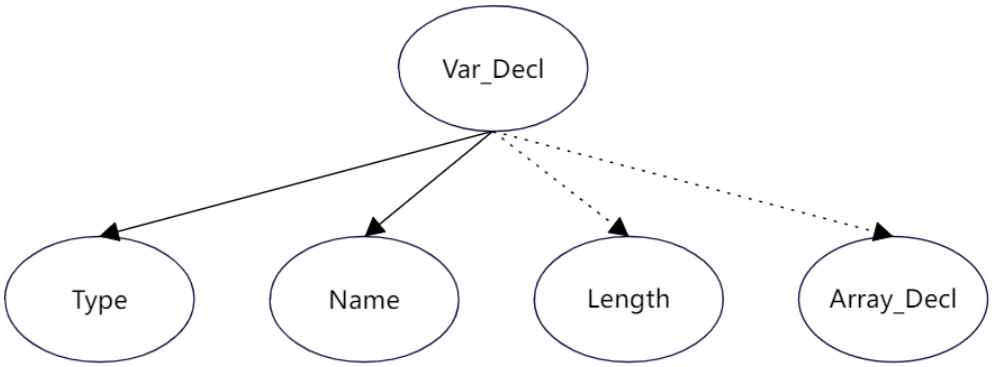


```
 void HandleDefinition(TreeNode *t) {
  if (auto FnAST = ParseFuncExpr(t)) {
    if (auto *FnIR = FnAST->codegen()) {
      fprintf(stderr, "Finish codegen\n");
      fprintf(stderr, "Read function definition:");
      FnIR->print(errs());
      fprintf(stderr, "\n");
    }
  }
}
```

3. 在生成抽象类结构`FnAST`的过程中会一直递归调用`ParseExpression`对这个函数结点的各个后代进行具体的类生成。对各种类型结点生成类的具体代码可在`ParExp.cpp`中查看。

```
 std::unique_ptr<ExprAST> ParseExpression(TreeNode *t);
 std::unique_ptr<ExprAST> ParseNumberExpr(TreeNode *t);
 std::unique_ptr<ExprAST> ParseIdentifierExpr(TreeNode *t);
 std::unique_ptr<StatementsAST> ParseStatementsExpr(TreeNode *t);
 std::unique_ptr<ExprAST> ParseIfExpr(TreeNode *t);
 std::unique_ptr<ExprAST> ParseArrayExpr(TreeNode *t);
 std::unique_ptr<ExprAST> ParseWhileExpr(TreeNode *t);
 std::unique_ptr<ExprAST> ParseCallExpr(TreeNode *t);
 std::unique_ptr<ExprAST> ParseArrayElementExpr(TreeNode *t);
 std::unique_ptr<ExprAST> ParseBinaryExpr(TreeNode *t);
 std::unique_ptr<FunctionAST> ParseFuncExpr(TreeNode *t);
 std::unique_ptr<ExprAST> ParseVarDeclExpr(TreeNode *t);

std::unique_ptr<ExprAST> ParseExpression(TreeNode *t) {
  switch (t->TreenodeType) {
  default:
    //return LogError("unknown token when expecting an expression");
  case Id:
    return ParseIdentifierExpr(t);
  case Const:
    return ParseNumberExpr(t);
  case Selection_Stmt:
    return ParseIfExpr(t);
  case Iteration_Stmt:
    return ParseWhileExpr(t);
  case Return_Stmt:
    return ParseReturnExpr(t);
  case Call:
    return ParseCallExpr(t);
  case Arry_Elem:
    return ParseArrayElementExpr(t);
  case Assign:
    return ParseBinaryExpr(t);
  case Op:
    return ParseBinaryExpr(t);
  case Var_Decl:
    return ParseVarDeclExpr(t);
  }
}
```

4. 如果结点类型为`Var_Decl`,说明为全局变量的声明，调用函数`HandleTopLevelExpression`进行生成。其中这个函数中会调用`ParseTopLevelExpr`生成类。`Var_Decl`节点的结构如下，下面的节点依次为`Var_Decl`节点的四个儿子节点`child[0]-child[3]`。其中`child[2]`和`child[3]`只有当类型为数组的时候才存在，因此用虚线表示。




```
 void HandleTopLevelExpression(TreeNode *t) {
  // Evaluate a top-level expression into an anonymous function.
      auto FnAST = ParseTopLevelExpr(t);
      return ;
}
```

5. `ParseTopLevelExpr`函数会进一步调用`ParseExpression`，解析全局变量并且使用**codegen1.cpp**中的全局变量定义接口进行构造。

```
 std::unique_ptr<FunctionAST> ParseTopLevelExpr(TreeNode *t) {
  auto E = ParseExpression(t)
  return nullptr;
}
```

3.5 语义分析设计

语义分析在程序中主要的内容是符号表的设计、符号表的检查以及类型检查，具体规则按照C语言的语义规则进行设计。语义分析是在接口上实现的，在由抽像语法树转变为基于LLVM设计的类的过程中进行语法分析，不需要进行值的存储，具体值都将在中间代码生成的时候进行计算和访问。

1. **全局变量符号表**

```
std::map<std::string,int> GlobalVariables;
```

全局变量可以有整型变量或者整型数组变量，通过`map`进行存储，整型变量默认的`int`大小为-1，数组默认大小即为数组长度，统一存在`GlobalVariables`中，由于程序在这里只进行语义分析，所以不需要存储具体值。在访问全局变量的时候调用函数，查找是否有重名变量或者插入新的全局变量。

```
bool CheckGlobalName(string checkname,int index = -1);
bool InsertGlobalName(string newname, int size = -1);
```

2. **全局函数记录**

函数具有函数名、返回值、参数（类型+名字）、局部变量（类型+名字）等信息，程序中设计了一个类专门存储这些信息。

```
class FuncInfo{
public:
  string ret_type;
  std::map<std::string,int> arguments;
  std::map<std::string,int> localvar;
  FuncInfo(string type):ret_type(type){}
};
```

参数和局部变量的存储方法仍然是按照全局变量存储的形式进行存储，通过`FuncInfo`类，我们在全局建立一个函数符号表，并且记录正在执行解析的函数名。

```
std::map<std::string, FuncInfo> GlobalFunctions;
std::string FuncName = "";
```

3. **变量冲突规则**
   - 全局变量不重名：包括全局变量和函数名不重复，通过`CheckNameExist`进行检查。

     ```
     bool CheckNameExist(string checkallname);
     ```

   - 一个函数内局部变量和参数不重名，局部变量和参数本身不重名：通过对`FuncInfo`类里的信息进行类似的查找，注意区分数组和整型，这两类变量在`int`上的值不同。例如：

     ```
     if(iter_function->second.arguments.size() != 0 & iter_function-
     >second.arguments.find(stmp) != iter_function->second.arguments.end() ){
         /* already exist */
         cout << "the argument " + stmp + " already exists!\n";
         errors ++;
     }
     ```

   - 函数执行语句不会访问未知的变量：即变量在局部变量、参数、全局变量都不存在，或者存在但是类型不一致，都被认为是非法变量。例如：

     ```
     map<std::string, FuncInfo>::iterator iter_function =
     GlobalFunctions.find(FuncName); if(iter_function-
     >second.arguments.size() != 0 & iter_function-
     >second.arguments.find(IdName) != iter_function->second.arguments.end())
     {
         if(iter_function->second.arguments.find(IdName)->second > -1){
           cout << IdName + " is an array, not a variable!\n";
           errors++;
         }
     }
     else if(iter_function->second.localvar.size() != 0 & iter_function-
     >second.localvar.find(IdName) != iter_function->second.localvar.end())
     {
         if(iter_function->second.localvar.find(IdName)->second > -1){
           cout << IdName + " is an array, not a variable!\n";
           errors++;
         }
     }
     ```

- `void`类型函数不能作为运算的一部分，只能单独执行：通过全局变量`ispoorCall`标识是否在识别计算的表达式，包括返回语句、二元计算式、判断和循环的条件判断式等，在调用函数解析时判断函数是否为`void`类型，从而发现可能的错误。

```
bool ispoorCall = true;
```

4. **错误输出**

所有的错误都将在发现的时候输出，并且记录错误的个数；当存在错误之后，中间代码将停止生成，优化操作也随之结束。但是程序将分析完所有的输入以及语义分析错误才会停止。

```
int errors = 0;
//.........
if(errors > 0){
      return ;
}
```

5. **例子演示**

下面所示输入代码有3处错误，执行结束后提示如下图所示：

```
int a[10];
int h;
int h(int x){
    int x;
    return a[11];
}
```

```
This function name: h is used in other declaration. Error!the program exits in advance due to 1 errors!
```

由于在读到函数名h的时候发现重复定义错误，函数无法定义会提前终止读取此函数信息进入下一个函数（这里为结束），所以只捕捉到一个错误。

修改上述函数名后继续操作：

```
int a[10];
int h;
int s(int x){
    int x;
    return a[11];
}
```

```
the variable x already exists in local variables or arguments!
size=2
a
a11out of bound
the program exits in advance due to 2 errors!
```

执行结果如上，发现x的重复定义以及返回数组a越界一共2个错误。

4 优化设计

LLVM的优化是基于 Pass 的，可以进行多轮多阶段不同类别的优化，每个优化的 Pass 各自独立设计。这些优化 Pass 以IR为输入，以IR为输出，根据一些规则精简IR中的指令数量或者缩短预期执行时间。在clang中可以指定不同的优化编译代码，指定优化级别是 `-O<N>` 这个参数，`-O0` 是默认值，表示不进行任何优化，通常在调试程序的时候会用。`-O2` 是中度优化水平，会开启大多数优化器，`-O1` 水平更低更保守。`-O3` 接近 `-O2`，区别是包含了一些相对更加消耗编译时间的优化器和为了提升运行效率而产生更大代码的优化器（例如把5次的循环拆成重复的5组指令而免除自增运算）。其他的，`-O4` 会开启链接时优化，`-Os` 会尽量降低代码大小，`-Oz` 相比 `-Os` 进一步降低代码大小。

优化分为不同的级别，不同的优化级别所使用的 Pass 不同，高级别的优化往往比较激进，有导致程序 bug 的风险。所以往往在实际应用中使用相对保守的策略。我在实验中使用了下述六种优化（第一种默认），在都使用的时候一些中间代码优化后会消除不可达到的块从而导致最终链接出现错误，我们在后续展示的优化使用了一种比较稳定的优化组合（4.1+4.2+4.3）。

优化模块嵌入

```
TheFPM = std::make_unique<legacy::FunctionPassManager>(TheModule.get());
TheFPM->add(somePass());
```

优化模块通过`llvm::legacy::FunctionPassManager`进行初始化，并且添加相应的优化 Pass 。

当然除了嵌入优化模块，我们可以直接通过类似下面的命令来优化。

```
$ llvm-as < example.ll | opt -mem2reg | llvm-dis
```

`opt`为优化命令，后续的参数表示进行优化的 Pass 。

4.1 常量折叠

常量折叠是一种非常常见和重要的优化，在LLVM生成IR的时候支持通过IRBuilder进行常量折叠优化，而不需要通过AST中的任何额外操作来提供支持。在调用IRBuilder进行代码生成时它会自动检查是否存在常量折叠的机会，如果有则直接返回常量而不是创建计算指令。以下是一个常量折叠的例子：

```
def test(x) 1+2+x;
```

优化前：

```
define double @test(double %x) {
entry:
        %addtmp = fadd double 2.000000e+00, 1.000000e+00
        %addtmp1 = fadd double %addtmp, %x
        ret double %addtmp1
}
```

优化后：

```
define double @test(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        ret double %addtmp
}
```

4.2 使用createInstructionCombiningPass（）

该优化的主要方法是在不改变语义的基础上通过合并一些指令，减少指令的数量，因为这个方法并不修改CFG，所以是一个相对安全的优化 Pass 。

```
// InstructionCombining - Combine instructions to form fewer, simple
// instructions. This pass does not modify the CFG, and has a tendency to make
// instructions dead, so a subsequent DCE pass is useful.
//
// This pass combines things like:
    %Y = add int 1, %X
    %Z = add int 1, %Y
    into:
    %Z = add int 2, %X
```

4.3 使用createReassociatePass();

该优化的功能主要是使式子之间建立联系，从而识别一些相同的表达式避免重复计算

```
//
// Reassociate - This pass reassociates commutative expressions in an order that
// is designed to promote better constant propagation, GCSE, LICM, PRE...
//
//For example:
4 + (x + 5)  ->  x + (4 + 5)
//
```

实例：

```
define double @test(double %x) {
entry:
        %addtmp = fadd double 3.000000e+00, %x
        %addtmp1 = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp1
        ret double %multmp
}
```

使用该优化之后，前两条语句被关联起来，优化后的结果如下：

```
define double @test(double %x) {
entry:
        %addtmp = fadd double %x, 3.000000e+00
        %multmp = fmul double %addtmp, %addtmp
        ret double %multmp
}
```

4.4 使用createPromoteMemoryToRegisterPass();

该优化主要作用是将IR代码中对内存的依赖转变为对寄存器的使用，减少对内存的操作。

```
//
// PromoteMemoryToRegister - This pass is used to promote memory references to
// be register references. A simple example of the transformation performed by
// this pass is:
//
//        FROM CODE                          TO CODE
    %X = alloca i32, i32 1                 ret i32 42
    store i32 42, i32 *%X
    %Y = load i32* %X
    ret i32 %Y
//
```

实例：

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32           ; type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  store i32 %X.0, i32* %X   ; Update X
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  store i32 %X.1, i32* %X   ; Update X
  br label %cond_next

cond_next:
  %X.2 = load i32* %X       ; Read X
  ret i32 %X.2
}
```

经过命令`llvm-as < example.ll | opt -mem2reg | llvm-dis`对上述文件进行`MemoryToRegister`的优化，结果如下：

```
@G = weak global i32 0
@H = weak global i32 0

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
```

```
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.01 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.01
}
```

这类优化一般会将内存操作转变为`phi`节点，这个也是LLVM IR中非常重要的概念，这里不进行展开。

4.5 使用createGVNPass();

GVN全称是Global Variable Numbering，主要是通过划分函数的同余类，相同的类中执行结束的值在每次程序执行后都相同。这一个 Pass 比较深奥，主要是运用了数学方法归纳得出，这里不进行讲解。

```
/// Create a legacy GVN pass. This also allows parameterizing whether or not
MemDep is enabled.
```

4.6 使用createCFGSimplificationPass();

该优化主要通过简化CFG语法，例如合并基础块、消除不可到达块等，这类优化很有可能会导致IR在生成目标代码时出现bug，需要谨慎使用。

```
// CFGSimplification - Merge basic blocks, eliminate unreachable blocks,
// simplify terminator instructions, convert switches to lookup tables, etc.
```

小结：

优化可以发生在编译阶段，中间代码阶段以及链接阶段进行，使用LLVM架构在中间对IR进行优化以及链接时优化都是非常方便的。但是正如我们所知：优化程度越高，也就越有可能出现错误。近年来有很多关于优化以及优化带来的后续问题的论文，这一问题也不断地被重视和讨论。我们在实现的过程中发现有时候优化的代码非常简洁，却有一些block莫名的被优化掉，除了我们本身IR实现过程中存在一些问题之外，优化的使用还有很多额外的内容需要考虑，优化的组合和优化的合理性、互斥性、等效性等问题同样也非常重要和有趣。

5 目标代码生成

为了验证程序的正确性，我们使用输出函数生成的可执行文件的执行结果的方式进行验证。

目标代码生成主要有两类做法：

一类是没有嵌入C语言函数的由**codegen1.cpp**生成的可执行文件，将符合定义的CFG的文法写入**input**文件中执行，得到中间代码并且保存到.ll文件中，之后使用**main.c**调用该函数，编译之后可以在终端打印结果。具体操作如下所示。

另一类是通过嵌入的C语言函数，执行后得到.ll文件，再于**printi.ll**共同生成可执行文件，即可通过`printi(__int16_t x)`打印数值。

5.1 简单的代码生成方式：

在**codegen1.cpp**中编写main函数用于生成中间代码：

其中`ParseSyntax()`构建了语法分析树，`ParseTree(root1)`将语法分析树转化成类并调用`codegen`函数生成中间代码，`TheModule->print(errs(), nullptr)`将中间代码打印到终端。

```
int main(){
  InitializeModule();
  char inputfile[200] = "input";
    freopen(inputfile,"r",stdin);
    if (stdin==NULL)
    {
        fprintf(stderr,"File %s Not Found\n",inputfile);
        exit(1);
    }
    BuildDFA();
    LexInit();
    PraseInit();
    ParseSyntax();
  fprintf(stderr, "Begin ParseTree\n");
    ParseTree(root1);
  TheModule->print(errs(), nullptr);
  return 0;
}
```

以如下代码为例：

文件名：input

```
int gcd (int u,int v)
{
if (v == 0){
  return u ;
}
else
 return gcd(v,u-u/v*v);
}
```

进入到代码所在目录，输入命令：

```
make
```

会根据makefile文件生成a.out。输入命令：

```
./a.out
```

会在终端输出中间代码IR，如下图所示。

```
define i16 @gcd(i16 %u, i16 %v) {
entry:
  %v2 = alloca i16, align 2
  %u1 = alloca i16, align 2
  store i16 %u, i16* %u1, align 2
  store i16 %v, i16* %v2, align 2
  %v3 = load i16, i16* %v2, align 2
  %cmptmp = icmp eq i16 %v3, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  %u4 = load i16, i16* %u1, align 2
  ret i16 %u4
  br label %ifcont

else:                                             ; preds = %entry
  %v5 = load i16, i16* %v2, align 2
  %u6 = load i16, i16* %u1, align 2
  %u7 = load i16, i16* %u1, align 2
  %v8 = load i16, i16* %v2, align 2
  %multmp = sdiv i16 %u7, %v8
  %v9 = load i16, i16* %v2, align 2
```

新建一个`input.ll`文件，将上述打印出的完整的中间代码复制到其中，如下。

```
define i16 @gcd(i16 %u, i16 %v) {
entry:
  %v2 = alloca i16, align 2
  %u1 = alloca i16, align 2
  store i16 %u, i16* %u1, align 2
  store i16 %v, i16* %v2, align 2
  %v3 = load i16, i16* %v2, align 2
  %cmptmp = icmp eq i16 %v3, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  %u4 = load i16, i16* %u1, align 2
  ret i16 %u4
  br label %ifcont

else:                                             ; preds = %entry
  %v5 = load i16, i16* %v2, align 2
  %u6 = load i16, i16* %u1, align 2
  %u7 = load i16, i16* %u1, align 2
  %v8 = load i16, i16* %v2, align 2
  %multmp = sdiv i16 %u7, %v8
  %v9 = load i16, i16* %v2, align 2
  %multmp10 = mul i16 %multmp, %v9
  %subtmp = sub i16 %u6, %multmp10
  %calltmp = call i16 @gcd(i16 %v5, i16 %subtmp)
  ret i16 %calltmp
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}
```

新建main.cpp文件，如下：

```
#include <iostream>

extern "C" {
   short int gcd(short int,short int);
}

int main() {
    std::cout << "gcd of 88 and 99: " << gcd(88, 99) << std::endl;
}
```

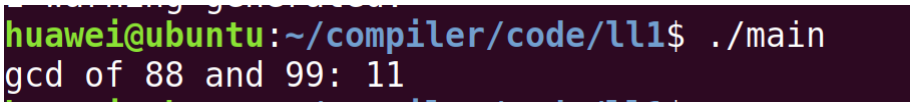
在终端输入命令：

```
clang++ main.cpp input.ll -o main
```

生成了main文件。输入命令：

```
./main
```

得到如下结果：



5.2 嵌入C语言函数的代码生成

先通过**printi.c**生成`printi(__int16_t X)`函数中间代码

```
#include<stdio.h>

__int16_t printi(__int16_t X){
    fprintf(stderr, "%d\n", (__int16_t)X);
    return 0;
}
```

执行命令

```
clang -S -emit-llvm printi.c -o printi.ll
```

生成的**printi.ll**如下：

```
; ModuleID = 'printi.c'
source_filename = "printi.c"
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-unknown-linux-gnu"

%struct._IO_FILE = type { i32, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*,
i8*, %struct._IO_marker*, %struct._IO_FILE*, i32, i32, i64, i16, i8, [1 x i8],
i8*, i64, i8*, i8*, i8*, i8*, i64, i32, [20 x i8] }
%struct._IO_marker = type { %struct._IO_marker*, %struct._IO_FILE*, i32 }

@stderr = external dso_local global %struct._IO_FILE*, align 8
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
```

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local signext i16 @printi(i16 signext %0) #0 {
  %2 = alloca i16, align 2
  store i16 %0, i16* %2, align 2
  %3 = load %struct._IO_FILE*, %struct._IO_FILE** @stderr, align 8
  %4 = load i16, i16* %2, align 2
  %5 = sext i16 %4 to i32
  %6 = call i32 (%struct._IO_FILE*, i8*, ...) @fprintf(%struct._IO_FILE* %3, i8*
getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i64 0, i64 0), i32 %5)
  ret i16 0
}

declare dso_local i32 @fprintf(%struct._IO_FILE*, i8*, ...) #1

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-
legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-
size"="8" "target-cpu"="x86-64" "target-
features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-
protector-buffer-size"="8" "target-cpu"="x86-64" "target-
features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"uwtable", i32 1}
!2 = !{i32 7, !"frame-pointer", i32 2}
!3 = !{!"clang version 13.0.0"}
```

之后使用可调用printi的文件到文件**input**中，如下：

```
int fib(int x){
    if(x < 3){
        return 1;
    }
    else{
        return fib(x-1) + fib(x-2);
    }
}
int main(void){
  printi(fib(6));
  return 0;
}
```

执行命令`./final_optwithC`得到中间代码，存入文件**test.ll**中。

```
; ModuleID = 'my cool jit'
source_filename = "my cool jit"

declare i16 @printi(i16)

define i16 @fib(i16 %x) {
entry:
  %x1 = alloca i16, align 2
  store i16 %x, i16* %x1, align 2
```

```
  %lttmp = icmp slt i16 %x, 3
  br i1 %lttmp, label %then, label %else

then:                                             ; preds = %entry
  ret i16 1
  br label %ifcont

else:                                             ; preds = %entry
  %x3 = load i16, i16* %x1, align 2
  %subtmp = add i16 %x3, -1
  %calltmp = call i16 @fib(i16 %subtmp)
  %subtmp5 = add i16 %x3, -2
  %calltmp6 = call i16 @fib(i16 %subtmp5)
  %addtmp = add i16 %calltmp6, %calltmp
  ret i16 %addtmp
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}

define i16 @main() {
entry:
  %calltmp = call i16 @fib(i16 6)
  %calltmp1 = call i16 @printi(i16 %calltmp)
  ret i16 0
  ret i16 0
}
```

执行命令`clang test.ll printi.ll -o main`，得到可执行文件，执行得到结果如下。

```
pyfccc@ubuntu:~/code/ll$ clang test.ll printi.ll -o main
warning: overriding the module target triple with x86_64-unknown-linux-gnu [-Wov
erride-module]
1 warning generated.
pyfccc@ubuntu:~/code/ll$ ./main
8
```

6 测试样例设计

6.1 全局变量使用

测试代码：求n+1.

```
int a;
int f(int n){
a = n;
return a+1;
}
```

AST结果：

```
  Var_Decl
    Int
    Id: a
```

```
  Func
    Int
    Id: f
    Params
      Param
        Int
        Id: n
    Comp
      Assign
        Id: a
        Id: n
      Return_Stmt
        Op: +
          Id: a
          Const: 1
```

IR生成结果：

```
@a = global i16 0, align 2

define i16 @f(i16 %n) {
entry:
  %n1 = alloca i16, align 2
  store i16 %n, i16* %n1, align 2
  %n2 = load i16, i16* %n1, align 2
  store i16 %n2, i16* @a, align 2
  %a = load i16, i16* @a, align 2
  %addtmp = add i16 %a, 1
  ret i16 %addtmp
  ret i16 0
}
```

经过编译优化后生成的IR：

```
@a = global i16 0, align 2

define i16 @f(i16 %n) {
entry:
  store i16 %n, i16* @a, align 2
  %addtmp = add i16 %n, 1
  ret i16 %addtmp
  ret i16 0
}
```

测试用main函数：

```
#include <iostream>

extern "C" {
    short int f(short int);
}

int main() {
std::cout <<"output = " <<(int16_t)f(10) <<std::endl;
    return 0;
}
```

运行结果：

结果正确。

6.2 局部变量定义以及赋值

测试代码：求n+1。

```
int f(int n){
int a;
a = n;
return a+1;
}
```

AST结果：

```
  Func
    Int
    Id: f
    Params
      Param
        Int
        Id: n
    Comp
      Var_Decl
        Int
        Id: a
      Assign
        Id: a
        Id: n
      Return_Stmt
        Op: +
          Id: a
          Const: 1
```

IR生成结果：

```
define i16 @f(i16 %n) {
entry:
  %a = alloca i16, align 2
  %n1 = alloca i16, align 2
```

```
  store i16 %n, i16* %n1, align 2
  store i16 0, i16* %a, align 2
  %n2 = load i16, i16* %n1, align 2
  store i16 %n2, i16* %a, align 2
  %a3 = load i16, i16* %a, align 2
  %addtmp = add i16 %a3, 1
  ret i16 %addtmp
  ret i16 0
}
```

经过编译优化后生成的IR:

```
define i16 @f(i16 %n) {
entry:
  %addtmp = add i16 %n, 1
  ret i16 %addtmp
  ret i16 0
}
```

测试用main函数:

```
#include <iostream>

extern "C" {
    short int f(short int);
}

int main() {
std::cout <<"output = " <<(int16_t)f(10) <<std::endl;
    return 0;
}
```

运行结果:

```
output = 11
```

结果正确。

6.3 数组的使用和设计

测试代码：返回10的2次方.

```
int a[3];
int f(int n){
a[0] = 1;
a[1] = 10;
a[2] = 100;
return a[2];
}
```

AST结果:

```
  Var_Decl
```

```
    Int
    Id: a
    Const: 3
    Arry_Decl
  Func
    Int
    Id: f
    Params
      Param
        Int
        Id: n
    Comp
      Assign
        Arry_Elem
          Id: a
          Const: 0
        Const: 1
      Assign
        Arry_Elem
          Id: a
          Const: 1
        Const: 10
      Assign
        Arry_Elem
          Id: a
          Const: 2
        Const: 100
      Return_Stmt
        Arry_Elem
          Id: a
          Const: 2
```

IR生成结果：

```
@"a[0]" = global i16 0, align 2
@"a[1]" = global i16 0, align 2
@"a[2]" = global i16 0, align 2

define i16 @f(i16 %n) {
entry:
  %n1 = alloca i16, align 2
  store i16 %n, i16* %n1, align 2
  store i16 1, i16* @"a[0]", align 2
  store i16 10, i16* @"a[1]", align 2
  store i16 100, i16* @"a[2]", align 2
  %"a[2]" = load i16, i16* @"a[2]", align 2
  ret i16 %"a[2]"
  ret i16 0
}
```

经过编译优化后生成的IR:

```
@"a[0]" = global i16 0, align 2
@"a[1]" = global i16 0, align 2
@"a[2]" = global i16 0, align 2

define i16 @f(i16 %n) {
entry:
  store i16 1, i16* @"a[0]", align 2
  store i16 10, i16* @"a[1]", align 2
  store i16 100, i16* @"a[2]", align 2
  ret i16 100
  ret i16 0
}
```

测试用main函数：

```
#include <iostream>

extern "C" {
    short int f(short int);
}

int main() {
std::cout <<"output = " <<(int16_t)f(10) <<std::endl;
    return 0;
}
```

运行结果：

```
output = 100
```

结果正确。

6.4 分支

测试代码：判断n是否大于0。

```
int f(int n){
if(n>0)return 1;
else return 0;
}
```

AST结果：

```
  Func
    Int
    Id: f
    Params
      Param
        Int
        Id: n
    Comp
      Selection_Stmt
        Op: >
```

```
          Id: n
          Const: 0
        Return_Stmt
          Const: 1
        Return_Stmt
          Const: 0
```

IR生成结果：

```
define i16 @f(i16 %n) {
entry:
  %n1 = alloca i16, align 2
  store i16 %n, i16* %n1, align 2
  %n2 = load i16, i16* %n1, align 2
  %cmptmp = icmp sgt i16 %n2, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  ret i16 1
  br label %ifcont

else:                                             ; preds = %entry
  ret i16 0
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}
```

经过编译优化后生成的IR:

```
define i16 @f(i16 %n) {
entry:
  %cmptmp = icmp sgt i16 %n, 0
  br i1 %cmptmp, label %then, label %else

then:                                             ; preds = %entry
  ret i16 1
  br label %ifcont

else:                                             ; preds = %entry
  ret i16 0
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}
```

测试用main函数：

```
#include <iostream>

extern "C" {
    short int f(short int);
}

int main() {
std::cout <<"output = " <<(int16_t)f(10) <<std::endl;
std::cout <<"output = " <<(int16_t)f(-10) <<std::endl;
    return 0;
}
```

运行结果：

```
output = 1
output = 0
```

结果正确。

6.5 循环

测试代码：返回小于n的正整数的和。

```
int f(int n){
int sum;
int i;
sum = 0;
i=n;
  while(i>0){
   sum = sum + i;
   i = i - 1;
}
return sum;
}
```

AST结果：

```
    Func
    Int
    Id: f
    Params
      Param
        Int
        Id: n
    Comp
      Var_Decl
        Int
        Id: sum
      Var_Decl
        Int
        Id: i
      Assign
        Id: sum
        Const: 0
      Assign
```

```
        Id: i
        Id: n
      Iteration_Stmt
        Op: >
          Id: i
          Const: 0
        Assign
          Id: sum
          Op: +
            Id: sum
            Id: i
        Assign
          Id: i
          Op: -
            Id: i
            Const: 1
      Return_Stmt
        Id: sum
```

IR生成结果：

```
define i16 @f(i16 %n) {
entry:
  %i = alloca i16, align 2
  %sum = alloca i16, align 2
  %n1 = alloca i16, align 2
  store i16 %n, i16* %n1, align 2
  store i16 0, i16* %sum, align 2
  store i16 0, i16* %i, align 2
  store i16 0, i16* %sum, align 2
  %n2 = load i16, i16* %n1, align 2
  store i16 %n2, i16* %i, align 2
  %i3 = load i16, i16* %i, align 2
  %cmptmp = icmp sgt i16 %i3, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %ifcont

then:                                             ; preds = %then, %entry
  %sum4 = load i16, i16* %sum, align 2
  %i5 = load i16, i16* %i, align 2
  %addtmp = add i16 %sum4, %i5
  store i16 %addtmp, i16* %sum, align 2
  %i6 = load i16, i16* %i, align 2
  %subtmp = sub i16 %i6, 1
  store i16 %subtmp, i16* %i, align 2
  %i7 = load i16, i16* %i, align 2
  %cmptmp8 = icmp sgt i16 %i7, 0
  %ifcond9 = icmp ne i1 %cmptmp8, false
  br i1 %ifcond9, label %then, label %ifcont
  br label %ifcont

ifcont:                                           ; preds = %then, %then, %entry
  %sum10 = load i16, i16* %sum, align 2
  ret i16 %sum10
  ret i16 0
}
```

经过编译优化后生成的IR:

```
define i16 @f(i16 %n) {
entry:
  %cmptmp = icmp sgt i16 %n, 0
  br i1 %cmptmp, label %then, label %ifcont

then:                                             ; preds = %then, %entry
  %subtmp = add i16 %n, -1
  %cmptmp8 = icmp sgt i16 %subtmp, 0
  br i1 %cmptmp8, label %then, label %ifcont
  br label %ifcont

ifcont:                                           ; preds = %then, %then, %entry
  %sum.0 = phi i16 [ %n, %then ], [ 0, %entry ], [ undef, %then ]
  ret i16 %sum.0
  ret i16 0
}
```

测试用main函数:

```
#include <iostream>

extern "C" {
    short int f(short int);
}

int main() {
std::cout <<"output = " <<(int16_t)f(3) <<std::endl;
std::cout <<"output = " <<(int16_t)f(5) <<std::endl;
    return 0;
}
```

运行结果:

```
output = 6
output = 15
```

结果正确。

6.6 复合语句嵌套

测试代码：返回小于n的正偶数的和。

```
int f(int n){
int sum;
int i;
sum = 0;
i=n;
  while(i>0){
   if(i-i/2*2==0){
   sum = sum + i;
   i = i - 1;
}
```

```
else i = i-1;

}
return sum;
}
```

AST结果：

```
Func
  Int
  Id: f
  Params
    Param
      Int
      Id: n
  Comp
    Var_Decl
      Int
      Id: sum
    Var_Decl
      Int
      Id: i
    Assign
      Id: sum
      Const: 0
    Assign
      Id: i
      Id: n
    Iteration_Stmt
      Op: >
        Id: i
        Const: 0
      Selection_Stmt
        Op: ==
          Op: -
            Id: i
            Op: *
              Op: /
                Id: i
                Const: 2
              Const: 2
          Const: 0
        Assign
          Id: sum
          Op: +
            Id: sum
            Id: i
        Assign
          Id: i
          Op: -
            Id: i
            Const: 1
        Assign
          Id: i
          Op: -
            Id: i
            Const: 1
```

```
      Return_Stmt
        Id: sum
```

IR生成结果：

```
define i16 @f(i16 %n) {
entry:
  %i = alloca i16, align 2
  %sum = alloca i16, align 2
  %n1 = alloca i16, align 2
  store i16 %n, i16* %n1, align 2
  store i16 0, i16* %sum, align 2
  store i16 0, i16* %i, align 2
  store i16 0, i16* %sum, align 2
  %n2 = load i16, i16* %n1, align 2
  store i16 %n2, i16* %i, align 2
  %i3 = load i16, i16* %i, align 2
  %cmptmp = icmp sgt i16 %i3, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %ifcont19

then:                                             ; preds = %ifcont, %entry
  %i4 = load i16, i16* %i, align 2
  %i5 = load i16, i16* %i, align 2
  %multmp = sdiv i16 %i5, 2
  %multmp6 = mul i16 %multmp, 2
  %subtmp = sub i16 %i4, %multmp6
  %cmptmp7 = icmp eq i16 %subtmp, 0
  %ifcond8 = icmp ne i1 %cmptmp7, false
  br i1 %ifcond8, label %then9, label %else

then9:                                            ; preds = %then
  %sum10 = load i16, i16* %sum, align 2
  %i11 = load i16, i16* %i, align 2
  %addtmp = add i16 %sum10, %i11
  store i16 %addtmp, i16* %sum, align 2
  %i12 = load i16, i16* %i, align 2
  %subtmp13 = sub i16 %i12, 1
  store i16 %subtmp13, i16* %i, align 2
  br label %ifcont

else:                                             ; preds = %then
  %i14 = load i16, i16* %i, align 2
  %subtmp15 = sub i16 %i14, 1
  store i16 %subtmp15, i16* %i, align 2
  br label %ifcont

ifcont:                                           ; preds = %else, %then9
  %i16 = load i16, i16* %i, align 2
  %cmptmp17 = icmp sgt i16 %i16, 0
  %ifcond18 = icmp ne i1 %cmptmp17, false
  br i1 %ifcond18, label %then, label %ifcont19
  br label %ifcont19

ifcont19:                                         ; preds = %ifcont, %ifcont,
%entry
  %sum20 = load i16, i16* %sum, align 2
```

```
  ret i16 %sum20
  ret i16 0
}
```

经过编译优化后生成的IR:

```
define i16 @f(i16 %n) {
entry:
  %cmptmp = icmp sgt i16 %n, 0
  br i1 %cmptmp, label %then, label %ifcont19

then:                                             ; preds = %ifcont, %entry
  %0 = and i16 %n, 1
  %cmptmp7 = icmp eq i16 %0, 0
  br i1 %cmptmp7, label %then9, label %else

then9:                                            ; preds = %then
  br label %ifcont

else:                                             ; preds = %then
  br label %ifcont

ifcont:                                           ; preds = %else, %then9
  %sum.0 = phi i16 [ %n, %then9 ], [ 0, %else ]
  %i.0 = add i16 %n, -1
  %cmptmp17 = icmp sgt i16 %i.0, 0
  br i1 %cmptmp17, label %then, label %ifcont19
  br label %ifcont19

ifcont19:                                         ; preds = %ifcont, %ifcont,
%entry
  %sum.1 = phi i16 [ %sum.0, %ifcont ], [ 0, %entry ], [ undef, %ifcont ]
  ret i16 %sum.1
  ret i16 0
}
```

测试用main函数:

```
#include <iostream>

extern "C" {
    short int f(short int);
}

int main() {
std::cout <<"output = " <<(int16_t)f(3) <<std::endl;
std::cout <<"output = " <<(int16_t)f(5) <<std::endl;
    return 0;
}
```

运行结果:

```
output = 2
output = 6
```

结果正确。

6.7 函数调用

测试代码：返回小于n的正整数的和的两倍。

```
int g(int n){
int sum;
int i;
sum = 0;
i=n;
  while(i>0){
   sum = sum + i;
   i = i - 1;
}
return sum;
}

int f(int n){
return 2*g(n);
}
```

AST结果：

```
  Func
    Int
    Id: g
    Params
      Param
        Int
        Id: n
    Comp
      Var_Decl
        Int
        Id: sum
      Var_Decl
        Int
        Id: i
      Assign
        Id: sum
        Const: 0
      Assign
        Id: i
        Id: n
      Iteration_Stmt
        Op: >
          Id: i
          Const: 0
        Assign
          Id: sum
          Op: +
            Id: sum
            Id: i
        Assign
          Id: i
          Op: -
            Id: i
```

```
            Const: 1
      Return_Stmt
        Id: sum
  Func
    Int
    Id: f
    Params
      Param
        Int
        Id: n
    Comp
      Return_Stmt
        Op: *
          Const: 2
          Call
            Id: g
            Args
              Id: n  Var_Decl
    Int
    Id: a
    Const: 3
    Arry_Decl
  Func
    Int
    Id: f
    Params
      Param
        Int
        Id: n
    Comp
      Assign
        Arry_Elem
          Id: a
          Const: 0
        Const: 1
      Assign
        Arry_Elem
          Id: a
          Const: 1
        Const: 10
      Assign
        Arry_Elem
          Id: a
          Const: 2
        Const: 100
      Return_Stmt
        Arry_Elem
          Id: a
          Const: 2
```

IR生成结果：

```
define i16 @g(i16 %n) {
entry:
  %i = alloca i16, align 2
  %sum = alloca i16, align 2
  %n1 = alloca i16, align 2
```

```
  store i16 %n, i16* %n1, align 2
  store i16 0, i16* %sum, align 2
  store i16 0, i16* %i, align 2
  store i16 0, i16* %sum, align 2
  %n2 = load i16, i16* %n1, align 2
  store i16 %n2, i16* %i, align 2
  %i3 = load i16, i16* %i, align 2
  %cmptmp = icmp sgt i16 %i3, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %ifcont

then:                                             ; preds = %then, %entry
  %sum4 = load i16, i16* %sum, align 2
  %i5 = load i16, i16* %i, align 2
  %addtmp = add i16 %sum4, %i5
  store i16 %addtmp, i16* %sum, align 2
  %i6 = load i16, i16* %i, align 2
  %subtmp = sub i16 %i6, 1
  store i16 %subtmp, i16* %i, align 2
  %i7 = load i16, i16* %i, align 2
  %cmptmp8 = icmp sgt i16 %i7, 0
  %ifcond9 = icmp ne i1 %cmptmp8, false
  br i1 %ifcond9, label %then, label %ifcont
  br label %ifcont

ifcont:                                           ; preds = %then, %then, %entry
  %sum10 = load i16, i16* %sum, align 2
  ret i16 %sum10
  ret i16 0
}

define i16 @f(i16 %n) {
entry:
  %n1 = alloca i16, align 2
  store i16 %n, i16* %n1, align 2
  %n2 = load i16, i16* %n1, align 2
  %calltmp = call i16 @g(i16 %n2)
  %multmp = mul i16 2, %calltmp
  ret i16 %multmp
  ret i16 0
}
```

经过编译优化后生成的IR:

```
define i16 @g(i16 %n) {
entry:
  %cmptmp = icmp sgt i16 %n, 0
  br i1 %cmptmp, label %then, label %ifcont

then:                                             ; preds = %then, %entry
  %subtmp = add i16 %n, -1
  %cmptmp8 = icmp sgt i16 %subtmp, 0
  br i1 %cmptmp8, label %then, label %ifcont
  br label %ifcont

ifcont:                                           ; preds = %then, %then, %entry
```

```
  %sum.0 = phi i16 [ %n, %then ], [ 0, %entry ], [ undef, %then ]
  ret i16 %sum.0
  ret i16 0
}

define i16 @f(i16 %n) {
entry:
  %calltmp = call i16 @g(i16 %n)
  %multmp = shl i16 %calltmp, 1
  ret i16 %multmp
  ret i16 0
}
```

测试用main函数：

```
#include <iostream>

extern "C" {
    short int f(short int);
 short int g(short int);
}

int main() {
std::cout <<"output = " <<(int16_t)f(3) <<std::endl;
std::cout <<"output = " <<(int16_t)f(5) <<std::endl;
    return 0;
}
```

运行结果：

```
output = 12
output = 30
```

结构正确。

6.8 函数递归调用

测试代码：求两个数的最大公因数。

```
int gcd (int u,int v)
{
if (v == 0){
  return u ;
}

else
 return gcd(v,u-u/v*v);

}
```

AST结果：

```
   Func
    Int
```

```
    Id: gcd
    Params
      Param
        Int
        Id: u
      Param
        Int
        Id: v
    Comp
      Selection_Stmt
        Op: ==
          Id: v
          Const: 0
        Return_Stmt
          Id: u
        Return_Stmt
          Call
            Id: gcd
            Args
              Id: v
              Op: -
                Id: u
                Op: *
                  Op: /
                    Id: u
                    Id: v
                  Id: v
```

IR生成结果：

```
define i16 @gcd(i16 %u, i16 %v) {
entry:
  %v2 = alloca i16, align 2
  %u1 = alloca i16, align 2
  store i16 %u, i16* %u1, align 2
  store i16 %v, i16* %v2, align 2
  %v3 = load i16, i16* %v2, align 2
  %cmptmp = icmp eq i16 %v3, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  %u4 = load i16, i16* %u1, align 2
  ret i16 %u4
  br label %ifcont

else:                                             ; preds = %entry
  %v5 = load i16, i16* %v2, align 2
  %u6 = load i16, i16* %u1, align 2
  %u7 = load i16, i16* %u1, align 2
  %v8 = load i16, i16* %v2, align 2
  %multmp = sdiv i16 %u7, %v8
  %v9 = load i16, i16* %v2, align 2
  %multmp10 = mul i16 %multmp, %v9
  %subtmp = sub i16 %u6, %multmp10
  %calltmp = call i16 @gcd(i16 %v5, i16 %subtmp)
```

```
  ret i16 %calltmp
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}
```

经过编译优化后生成的IR:

```
define i16 @gcd(i16 %u, i16 %v) {
entry:
  %cmptmp = icmp eq i16 %v, 0
  br i1 %cmptmp, label %then, label %else

then:                                             ; preds = %entry
  ret i16 %u
  br label %ifcont

else:                                             ; preds = %entry
  %0 = srem i16 %u, %v
  %calltmp = call i16 @gcd(i16 %v, i16 %0)
  ret i16 %calltmp
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}
```

测试用main函数:

```
#include <iostream>

extern "C" {
    short int gcd(short int,short int);
}

int main() {
std::cout <<"output = " <<(int16_t)gcd(5,10) <<std::endl;
std::cout <<"output = " <<(int16_t)gcd(30,40) <<std::endl;
    return 0;
}
```

运行结果:

```
output = 5
output = 10
```

结果正确。

6.9 综合测试

测试代码：

```
int fib(int x){
    if(x < 3){
        return 1;
    }
    else{
        return fib(x-1) + fib(x-2);
    }
}

int f(int n){
int sum;
int i;
sum = 0;
i=n;
  while(i>0){
   sum = sum + i;
   i = i - 1;
}
return sum;
}

int g(int x,int flag){
if(flag==0)return fib(x);
else return f(x);
}
```

AST结果：

```
  Func
    Int
    Id: fib
    Params
      Param
        Int
        Id: x
    Comp
      Selection_Stmt
        Op: <
          Id: x
          Const: 3
        Return_Stmt
          Const: 1
        Return_Stmt
          Op: +
            Call
              Id: fib
              Args
                Op: -
                  Id: x
                  Const: 1
            Call
              Id: fib
              Args
```

```
              Op: -
                Id: x
                Const: 2
Func
  Int
  Id: f
  Params
    Param
      Int
      Id: n
  Comp
    Var_Decl
      Int
      Id: sum
    Var_Decl
      Int
      Id: i
    Assign
      Id: sum
      Const: 0
    Assign
      Id: i
      Id: n
    Iteration_Stmt
      Op: >
        Id: i
        Const: 0
      Assign
        Id: sum
        Op: +
          Id: sum
          Id: i
      Assign
        Id: i
        Op: -
          Id: i
          Const: 1
    Return_Stmt
      Id: sum
Func
  Int
  Id: g
  Params
    Param
      Int
      Id: x
    Param
      Int
      Id: flag
  Comp
    Selection_Stmt
      Op: ==
        Id: flag
        Const: 0
      Return_Stmt
        Call
          Id: fib
          Args
```

```
            Id: x
        Return_Stmt
          Call
            Id: f
            Args
              Id: x
```

IR生成结果：

```
define i16 @fib(i16 %x) {
entry:
  %x1 = alloca i16, align 2
  store i16 %x, i16* %x1, align 2
  %x2 = load i16, i16* %x1, align 2
  %cmptmp = icmp slt i16 %x2, 3
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  ret i16 1
  br label %ifcont

else:                                             ; preds = %entry
  %x3 = load i16, i16* %x1, align 2
  %subtmp = sub i16 %x3, 1
  %calltmp = call i16 @fib(i16 %subtmp)
  %x4 = load i16, i16* %x1, align 2
  %subtmp5 = sub i16 %x4, 2
  %calltmp6 = call i16 @fib(i16 %subtmp5)
  %addtmp = add i16 %calltmp, %calltmp6
  ret i16 %addtmp
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}

define i16 @f(i16 %n) {
entry:
  %i = alloca i16, align 2
  %sum = alloca i16, align 2
  %n1 = alloca i16, align 2
  store i16 %n, i16* %n1, align 2
  store i16 0, i16* %sum, align 2
  store i16 0, i16* %i, align 2
  store i16 0, i16* %sum, align 2
  %n2 = load i16, i16* %n1, align 2
  store i16 %n2, i16* %i, align 2
  %i3 = load i16, i16* %i, align 2
  %cmptmp = icmp sgt i16 %i3, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %ifcont

then:                                             ; preds = %then, %entry
  %sum4 = load i16, i16* %sum, align 2
  %i5 = load i16, i16* %i, align 2
  %addtmp = add i16 %sum4, %i5
```

```
  store i16 %addtmp, i16* %sum, align 2
  %i6 = load i16, i16* %i, align 2
  %subtmp = sub i16 %i6, 1
  store i16 %subtmp, i16* %i, align 2
  %i7 = load i16, i16* %i, align 2
  %cmptmp8 = icmp sgt i16 %i7, 0
  %ifcond9 = icmp ne i1 %cmptmp8, false
  br i1 %ifcond9, label %then, label %ifcont
  br label %ifcont

ifcont:                                           ; preds = %then, %then, %entry
  %sum10 = load i16, i16* %sum, align 2
  ret i16 %sum10
  ret i16 0
}

define i16 @g(i16 %x, i16 %flag) {
entry:
  %flag2 = alloca i16, align 2
  %x1 = alloca i16, align 2
  store i16 %x, i16* %x1, align 2
  store i16 %flag, i16* %flag2, align 2
  %flag3 = load i16, i16* %flag2, align 2
  %cmptmp = icmp eq i16 %flag3, 0
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  %x4 = load i16, i16* %x1, align 2
  %calltmp = call i16 @fib(i16 %x4)
  ret i16 %calltmp
  br label %ifcont

else:                                             ; preds = %entry
  %x5 = load i16, i16* %x1, align 2
  %calltmp6 = call i16 @f(i16 %x5)
  ret i16 %calltmp6
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}
```

经过编译优化后生成的IR:

```
define i16 @fib(i16 %x) {
entry:
  %cmptmp = icmp slt i16 %x, 3
  br i1 %cmptmp, label %then, label %else

then:                                             ; preds = %entry
  ret i16 1
  br label %ifcont

else:                                             ; preds = %entry
  %subtmp = add i16 %x, -1
```

```
  %calltmp = call i16 @fib(i16 %subtmp)
  %subtmp5 = add i16 %x, -2
  %calltmp6 = call i16 @fib(i16 %subtmp5)
  %addtmp = add i16 %calltmp6, %calltmp
  ret i16 %addtmp
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}

define i16 @f(i16 %n) {
entry:
  %cmptmp = icmp sgt i16 %n, 0
  br i1 %cmptmp, label %then, label %ifcont

then:                                             ; preds = %then, %entry
  %subtmp = add i16 %n, -1
  %cmptmp8 = icmp sgt i16 %subtmp, 0
  br i1 %cmptmp8, label %then, label %ifcont
  br label %ifcont

ifcont:                                           ; preds = %then, %then, %entry
  %sum.0 = phi i16 [ %n, %then ], [ 0, %entry ], [ undef, %then ]
  ret i16 %sum.0
  ret i16 0
}

define i16 @g(i16 %x, i16 %flag) {
entry:
  %cmptmp = icmp eq i16 %flag, 0
  br i1 %cmptmp, label %then, label %else

then:                                             ; preds = %entry
  %calltmp = call i16 @fib(i16 %x)
  ret i16 %calltmp
  br label %ifcont

else:                                             ; preds = %entry
  %calltmp6 = call i16 @f(i16 %x)
  ret i16 %calltmp6
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  ret i16 0
}
```

测试用main函数：

```
#include <iostream>

extern "C" {
short int fib(short int);
    short int f(short int);
 short int g(short int,short int);
}

int main() {
std::cout <<"output = " <<(int16_t)g(5,0) <<std::endl;
std::cout <<"output = " <<(int16_t)g(5,1) <<std::endl;
    return 0;
}
```

运行结果：

```
output = 5
output = 15
```

第一行输出的是fib(5)，等于5，正确。第二行输出的是f(5)，等于15，正确。

7 总结与回顾

我们基于LLVM架构下完成了一个简单的编译器，支持一个类似C-语言的CFG，但是其实做了一点简化。在刚入手项目的时候还没有讲到IR，这一切对于我们来说非常陌生，我四处的查询阅览LLVM的介绍资料，总算在对LLVM架构有了一定的基础上开始布置小组任务，刚开始进度非常缓慢，我希望组员们和我一起看一些文章或者视频讲解，在了解我们在做什么的基础上进行设计和分工，但是效果并不太好。一方面是因为我自己对此了解有限，而且看这些新的概念非常耗费精力；另一方面是我的队友们不是特别积极，因为当时才是5月8日，现在是5月22日，从那天开始我连续做了15天。15天之前一切看起来还不是那么紧迫。

在探索了两三天后，我决定从LLVM Tutorial的Kaleidoscope入手开始从零到一的学习，刚开始进展很慢，很艰难，后来看着一位youtuber的视频才慢慢重燃希望，一点点看下去。途中讨论过几次但是队友们大多没什么想法，变成我的一言堂。我自己花了五六天的时间一点点啃，最终才有了一点收获。在此期间，一些同学已经写完。因为担心队友的能力，我分给了大四队友最简单的词法分析和语法分析内容，这些内容仅仅依靠前半学期的知识就可以解决，并且之前也时实践过。在我们讨论了父类子类的设计之后我决定把这个简单的任务给我的队友，之后等他完成再进行后续的工作。在此之前，我们已经确定了CFG，以及简单讨论了类的设计。我让他也看看从而可以很好的和我们对接，后续再安排其他的任务给他。这一天是5月10日，从进度上来说还是很快的。这时候，我一直在找可以帮助我们入门的文档，官方文档有些过于复杂了对于我来说，我的另一个队友则负责和我一起研究LLVM，不过他的进度稍慢。

5月11日，大四队友给了我一份有bug的词法分析lex（进度设置到5.11发出来），由于我们设计的语言非常简单，我原计划这份文件很快就能完成。bug改好了之后文件发给了他，是5月11日中午。

之后，在我的催促下；5月13日，大四队友发出了语法分析和词法分析的内容，我打开看了发现完全没有使用lex和yacc（这并不会影响什么，因为我们的语言基本是LL(1)的，如果有例外也很容易处理，使用LALR(1)或者LL(1)效果上是一致的），甚至语法都和我们设计的不同。在我催促下他发了一个讲解视频，大概10分钟，没有太多实质性的内容。后续我细看了他的代码并且整理了逻辑。代码中没有用到类的概念，不同类别使用枚举类型值区分，这样的设计对于我来说是极其不友好的，几乎没有可拓展性；并且没有设计接口以及没有注释。

后续我希望他能进行AST的可视化以及测试样例的设计以及自己模块的报告的完成，在5月16日他回复一次“好的”后失联、未读、并且没有完成任何后续工作。如果只从代码数量来说他确实也贡献了1000行左右，但是从难度上来说，语法分析本身逻辑上很简单，不需要使用其他库或者API，相对来说是很简单的，任务量并不大。最开始我分配给他最简单的部分，在开会之初都有介绍，希望他能配合我们后续的代码生成（这一点是相当难的我也不敢交给他写），可是后来根本无法联系上这位同学。看到他提供的代码，我觉得自己可能更像是一个数据结构助教布置了一次写B+树的作业，完全没有考虑后续的实现的问题，并且后续拒绝沟通和交流，这里无需多言读者可以自行体会。

大概在5月15-16日，我和另一位队友看完了LLVM Tutorial，我开始模仿者构建类，另一个队友在我的基础上实现接口以及语义分析，我原本计划后续代码生成共同设计，但是队友那边出现了一些状况，所以中间代码生成的部分也是我自己完成的。到5月17日，已经是我第9天做LLVM的内容，这一天代码写了很多，但是跑不出结果，当时每天都很焦虑很痛苦。

在5月18日，队友语义分析失败了，但是接口写好了，我在他的接口上运行，用lldb调试了大概一天左右，后来能生成中间代码了，于是一切变的明朗起来。我和我的这个队友互相配合，不断的调试错误，后续实验延期两天，我又加了优化、嵌入函数以及语义分析的代码，我的队友测试并不断告我出现的bug，最终我们写好了这个编译器。在实验过程中，由于压力过大并且感觉最终无法实现，我对一些内容进行了简化，所以最终的CFG文法也略微进行了简化，不过和日常的使用差别不是很大。

今天，5月22日，写了四天的报告（一边写一边加功能一边调试）终于要尘埃落定，想起和LLVM那些痛苦又奇妙的回忆，真是令人感慨。同时见证了LLVM的强大功能，令我非常震撼！此外，还学会使用clang，这个编译器也要比gcc好用很多。

——庞懿非