

浙江大学



课程名称:	编译原理
学年学期:	2020-2021 学年春夏学期
学 院:	计算机科学与技术学院
指导教师:	李 莹
组 别:	第十组
小组成员:	艾 琪 3180105950
	王 睿 3180103650
	宋炜铁 3180103425

2021 年 6 月 21 日

概述

1 项目简介

本项目是基于flex, bison以及LLVM, 使用c++11实现的C-语法编译器, 使用flex结合yacc对源代码进行词法、语法分析; 在语法分析阶段生成整个源代码相应的抽象语法树后, 根据LLVM IR (Intermediate Representation) 模块中定义的中间代码语法输出符合LLVM中间语言语法、机器无关的中间代码; 最后, 本项目通过调用LLVM后端接口, 将中间代码编译生成可执行文件。同时我们也利用D3.js实现了AST可视化。

本项目解析的语法与是C语言的一个子集, 目前已支持的数据类型及操作包括:

- void
- int
- float
- char
- bool
- 数组 (包括int, float, char, bool类型)
支持的语法包括:
- 变量的声明、初始化
- 函数声明和调用 (参数类型可以是任意已支持类型)
- 控制流语句if-else, while及任意层级嵌套使用 (支持break)
- 单行注释 (// 或 /* */)
- 二元运算符、赋值、隐式类型转换
- 全局变量的使用
- ...

2 提交文件说明

- src: 编译器源代码文件夹
 - lexical.l: Flex源代码, 主要实现词法分析, 生成Token
 - grammar.y: Yacc源代码, 主要实现语法分析, 生成抽象语法树
 - lexical.cpp: Flex根据lexical生成的词法分析器 (可Make生成)
 - grammar.hpp: Yacc根据grammar.y生成的语法分析器头文件 (可Make生成)
 - grammar.cpp: Yacc根据grammar.y生成的语法分析器C++文件 (可Make生成)
 - ast.h: 抽象语法树头文件, 定义AST节点类
 - ast.cpp: 抽象语法树实现文件, 主要包含irbuild和jsonGen方法的实现
 - code_gen.h: 中间代码生成器头文件, 定义生成器环境
 - code_gen.cpp: 中间代码生成器实现文件
 - main.cpp: 主函数所在文件, 主要负责调用词法分析器、语法分析器、代码生成器
 - type.h: 定义实现的数据类型
 - Makefile: 定义编译链接规则
- vis: 可视化文件夹
 - ast_tree.json: 基于AST生成的JSON文件
 - disp.html: 可视化AST的网页文件
- test: 测试文件夹
 - parse: 编译器可执行文件

- quicksort
 - quicksort.cmm: C--源代码
 - quicksort.out: 可执行文件
- matrix-multiplication
 - matrix-multiplication.cmm
 - matrix-multiplication.out
- auto-advisor
 - auto-advisor.cmm
 - auto-advisor.out
- docs: 报告文档文件夹
 - report.pdf: 报告文档
 - slides.pdf: 展示文档

3 开发环境

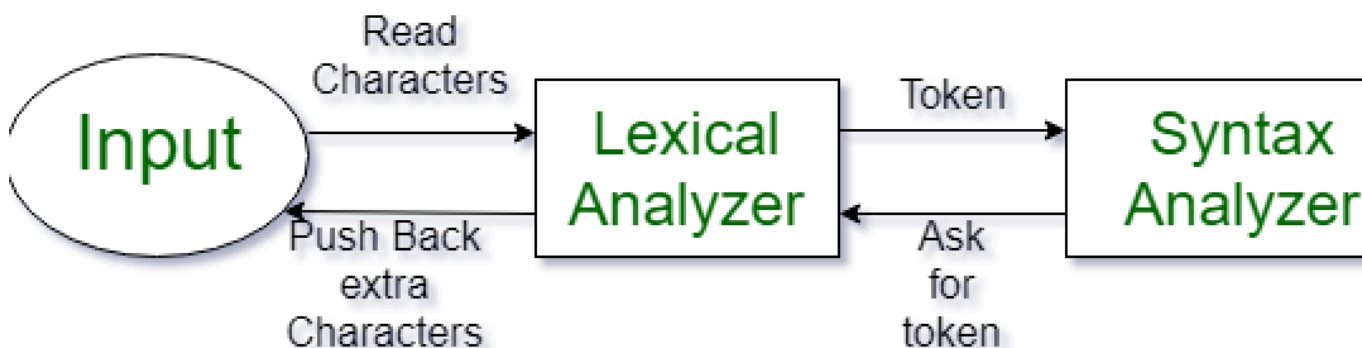
- 操作系统: Linux
- 编译环境:
 - Flex
 - Bison
 - LLVM 13.0

第一章 词法分析

1.1 flex介绍

我们使用了flex来完成词法分析过程。flex（快速词法分析产生器，fast lexical analyzer generator）是一种词法分析程序。它是lex的开放源代码版本，以BSD许可证发布。通常与GNU bison一同运作，但是它本身不是GNU计划的一部分。

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。



标准lex文件由三部分组成，分别是定义区、规则区和用户子过程区。在定义区，用户可以编写C语言中的声明语句，导入需要的头文件或声明变量。在规则区，用户需要编写以正则表达式和对应的动作的形式的代码。在用户子过程区，用户可以定义函数。

```
{definitions}
%%
{rules}
%%
{subroutines}
```

1.2 具体实现

词法分析是编译器实现中相对简单的一步。想好所需要实现的语法后，需要将输入转化为一系列的内部标记token。本项目定义的token包括：C语言关键字if、else、return、while、int、float等每个关键字对应的token，加减乘除、位运算、括号、逗号、取地址等符号token，以及表示用户自定义的标志符（identifier）的token、数字常量token（整型和浮点数）、字符串常量token等。

我们需要先在yacc源文件grammar.y中声明这些token，并在lex源文件lexical.l中定义这些token对应的操作。

token大致分为如下几种情况：

- C Minus语言关键字

```
RETURN return
IF if
ELSE else
WHILE while
BREAK break
BOOL "true"|"false"
TYPE "int"|"float"|"boolean"|"char"
...
%%
...
{IF}    {yylval.label_tree = new Node(yytext, "IF", 0); return IF;}
{ELSE}   {yylval.label_tree = new Node(yytext, "ELSE", 0); return ELSE;}
{WHILE}  {yylval.label_tree = new Node(yytext, "WHILE", 0); return WHILE;}
{BREAK}  {yylval.label_tree = new Node(yytext, "BREAK", 0); return BREAK;}
{RETURN} {yylval.label_tree = new Node(yytext, "RETURN", 0); return RETURN;}
{BOOL}   {yylval.label_tree = new Node(yytext, "BOOL", 0); return BOOL;}
{TYPE}   {yylval.label_tree = new Node(yytext, "TYPE", 0); return TYPE;}
```

- C Minus语言操作符

```
LP \(
RP \)
LB \[
RB \]
LC \{
RC \}
ASSIGNOP =
RELOP  ">"|"<"| ">="|"<="|"=="|"!="
PLUS \+
MINUS -
```

```

STAR \*
DIV \/
AND &&
OR "||"
NOT !
...
%%
...
{LP}    {yyval.label_tree = new Node(yytext, "LP", 0); return LP;}
{RP}    {yyval.label_tree = new Node(yytext, "RP", 0); return RP;}
{LB}    {yyval.label_tree = new Node(yytext, "LB", 0); return LB;}
{RB}    {yyval.label_tree = new Node(yytext, "RB", 0); return RB;}
{LC}    {yyval.label_tree = new Node(yytext, "LC", 0); return LC;}
{RC}    {yyval.label_tree = new Node(yytext, "RC", 0); return RC;}
{ASSIGNOP} {yyval.label_tree = new Node(yytext, "ASSIGNOP", 0); return ASSIGNOP;}
{RELOP}  {yyval.label_tree = new Node(yytext, "RELOP", 0); return RELOP;}
{PLUS}   {yyval.label_tree = new Node(yytext, "PLUS", 0); return PLUS;}
{MINUS}  {yyval.label_tree = new Node(yytext, "MINUS", 0); return MINUS;}
{STAR}   {yyval.label_tree = new Node(yytext, "STAR", 0); return STAR;}
{DIV}    {yyval.label_tree = new Node(yytext, "DIV", 0); return DIV;}
{AND}    {yyval.label_tree = new Node(yytext, "AND", 0); return AND;}
{OR}     {yyval.label_tree = new Node(yytext, "OR", 0); return OR;}
{NOT}    {yyval.label_tree = new Node(yytext, "NOT", 0); return NOT;}
{AND}    {yyval.label_tree = new Node(yytext, "AND", 0); return AND;}
{OR}     {yyval.label_tree = new Node(yytext, "OR", 0); return OR;}
{NOT}    {yyval.label_tree = new Node(yytext, "NOT", 0); return NOT;}

```

- C Minus语言标识符和其他标记

```

digit  [0-9]
digits [0-9]+
CHAR  \'\.\'|\'\\\'
STRING \"(\\.|[^\"])*\"
INT  0|[1-9]{digit}*
FLOAT {digits}\.{digits}
ID  [_a-zA-Z][_0-9a-zA-Z]*
CR  \r
LF  \n
TAB \t|" "
...
%%
...
{CHAR}  {yyval.label_tree = new Node(yytext, "CHAR", 0); return CHAR;}
{INT}   {yyval.label_tree = new Node(yytext, "INT", 0); return INT;}
{FLOAT} {yyval.label_tree = new Node(yytext, "FLOAT", 0); return FLOAT;}
{STRING} {yyval.label_tree = new Node(yytext, "STRING", 0); return STRING;}
{ID}    {yyval.label_tree = new Node(yytext, "ID", 0); return ID;}

{CR}   {};

```

```
{LF} {yycolumn=1;}
{TAB}  {}
```

- C Minus语言注释

```
COMMENTMULTILINE  \/\*([^\*]|(\*)*[/])*(\*)*\*/
COMMENTLINE  \/\[/[^\n]*

...
%%

...
{COMMENTMULTILINE} {}
{COMMENTLINE} {}
```

第二章 语法分析

在计算机科学和语言学中，语法分析（英语：syntactic analysis，也叫parsing）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。

语法分析器（parser）通常是作为编译器或解释器的组件出现的，它的作用是进行语法检查、并构建由输入的单词组成的数据结构（一般是语法分析树、抽象语法树等层次化的数据结构）。语法分析器通常使用一个独立的词法分析器从输入字符流中分离出一个个的“单词”，并将单词流作为其输入。实际开发中，语法分析器可以手工编写，也可以使用工具（半）自动生成。

2.1 Yacc介绍

Yacc是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器），它使用巴克斯范式(BNF)定义语法，能处理上下文无关文法(context-free)。Yacc生成的编译器主要是用C语言写成的语法解析器(Parser)，需要与词法解析器Lex一起使用，再把两部分产生出来的C程序一并编译。

与Lex相似，Yacc的输入文件由以%%分割的三部分组成，分别是声明区、规则区和程序区。三部分的功能与Lex相似，不同的是规则区的正则表达式替换为CFG，在声明区要提前声明好使用到的终结符以及非终结符的类型。

```
/* definitions */
....

%%

/* rules */
....
%%

/* auxiliary routines */
....
```

2.2 抽象语法树

在计算机科学中，抽象语法树（Abstract Syntax Tree, AST），或简称语法树（Syntax tree），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。

之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于if-condition-then这样的条件跳转语句，可以使用带有两个分支的节点来表示。

2.2.1 Node类

我们将抽象语法树的所有节点抽象为一个Node类，Node有节点名、结点类型、儿子数、行数等属性，也实现了初始化、添加结点、设置类型、获取类型等相关函数。

```
class Node {
public:
    // Value or name of node, if type of node is int, the value of nodeName is the
    // value of the integer, float, bool, char are similar
    // if type is var, the value is the name of this variable
    string *nodeName;
    // The type of the node
    string *nodeType;
    // The type of exp, var or const
    int valueType;
    // The number of child of the node
    int childNum;
    // Child nodes of this node
    Node **childNode;
    // The number of rows of the node in the file
    int lineNo;

    // Function
    int getValueType();
    int getValueType(Node *node);
    void setValueType(int type);
    Node(char * nodeName, string nodeType, int lineNo);
    Node(string nodeName, string nodeType, int childNum, ...);
    ...
}
```

2.3 语法分析的具体实现

首先在声明区声明好终结符和非终结符类型。

```
/*declared types*/
%union
{
    struct Node* label_tree;
}
```

```

%token <label_tree> INT
%token <label_tree> FLOAT
%token <label_tree> CHAR
%token <label_tree> BOOL
%token <label_tree> STRING
%token <label_tree> ID
%token <label_tree> SEMI
%token <label_tree> COMMA
%token <label_tree> ASSIGNOP
%token <label_tree> RELOP
%token <label_tree> PLUS MINUS STAR DIV
%token <label_tree> AND OR
%token <label_tree> NOT
%token <label_tree> TYPE
%token <label_tree> LP RP LB RB LC RC
%token <label_tree> RETURN
%token <label_tree> IF
%token <label_tree> ELSE
%token <label_tree> WHILE
%token <label_tree> BREAK

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%right ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right NOT
%left LP RP LB RB

%type <label_tree> Program ExtDefList ExtDef ExtDecList Specifier
%type <label_tree> VarDec FunDec VarList ParamDec CompSt StmtList Stmt DefList Def
DecList Exp Args

```

接着按照自顶向下的顺序构造语法书，部分文法如下：

```

Program:
    ExtDefList {
        $$ = new Node("", "Program", 1, $1);
        ROOT = $$;
    };

ExtDefList:
    ExtDef ExtDefList {
        $$ = new Node("", "ExtDefList", 2, $1, $2);
    };

```



```

    }
    | %empty {
        $$ = nullptr;
    }
    ;

ExtDef:
    Specifier ExtDecList SEMI {
        $$ = new Node("", "ExtDef", 3, $1, $2, $3);
    }
    | Specifier FunDec CompSt {
        $$ = new Node("", "ExtDef", 3, $1, $2, $3);
    }
    ;

ExtDecList:
    VarDec {
        $$ = new Node("", "ExtDecList", 1, $1);
    }
    | VarDec COMMA ExtDecList {
        $$ = new Node("", "ExtDecList", 3, $1, $2, $3);
    }
    ;

Specifier:
    TYPE {
        $$ = new Node("", "Specifier", 1, $1);
    }
    ;

VarDec:
    ...

FunDec:
    ...

VarList:
    ParamDec COMMA VarList {
        $$ = new Node("", "VarList", 3, $1, $2, $3);
    }
    | ParamDec {
        $$ = new Node("", "VarList", 1, $1);
    }
    ;

ParamDec:
    Specifier VarDec {
        $$ = new Node("", "ParamDec", 2, $1, $2);
    }

```

```

;

CompSt:
.

StmtList:
...

Stmt:
...

DefList:
...

Def:
...

DecList:
...

Exp:
...

Args:
    Exp COMMA Args {
        $$ = new Node("", "Args", 3, $1, $2, $3);
    }
    | Exp {
        $$ = new Node("", "Args", 1, $1);
    }
;

%%

```

2.4 抽象语法树可视化

语法树可视化使用d3.js完成，d3.js可以通过json数据绘制出树图html，AST可以通过节点的jsonGen方法获得json数据。

jsonGen函数:

```

Json::Value Node::jsonGen(){
    Json::Value root;
    string padding = "";
    if(this->nodeType->compare("Specifier") == 0 || this->nodeType->compare("Exp") ==
0){
        int valueType = this->getValueType();
        switch(valueType){
            case VOID:

```

```

        padding = "void";
        break;
    case VAR:
        padding = "var";
        break;
    case ARRAY:
        padding = "array";
        break;
    case FUN:
        padding = "func";
        break;
    case TYPE_INT:
        padding = "int";
        break;
    case TYPE_INT_ARRAY:
        padding = "int array";
        break;
    case TYPE_FLOAT:
        padding = "float";
        break;
    case TYPE_FLOAT_ARRAY:
        padding = "float array";
        break;
    case TYPE_CHAR:
        padding = "char";
        break;
    case TYPE_CHAR_ARRAY:
        padding = "char array";
        break;
    case TYPE_BOOL:
        padding = "bool";
        break;
    case TYPE_BOOL_ARRAY:
        padding = "bool array";
        break;
    default:
        break;
    }

}

root["name"] = *this->nodeType + (padding == "" ? "" : ":" + padding);

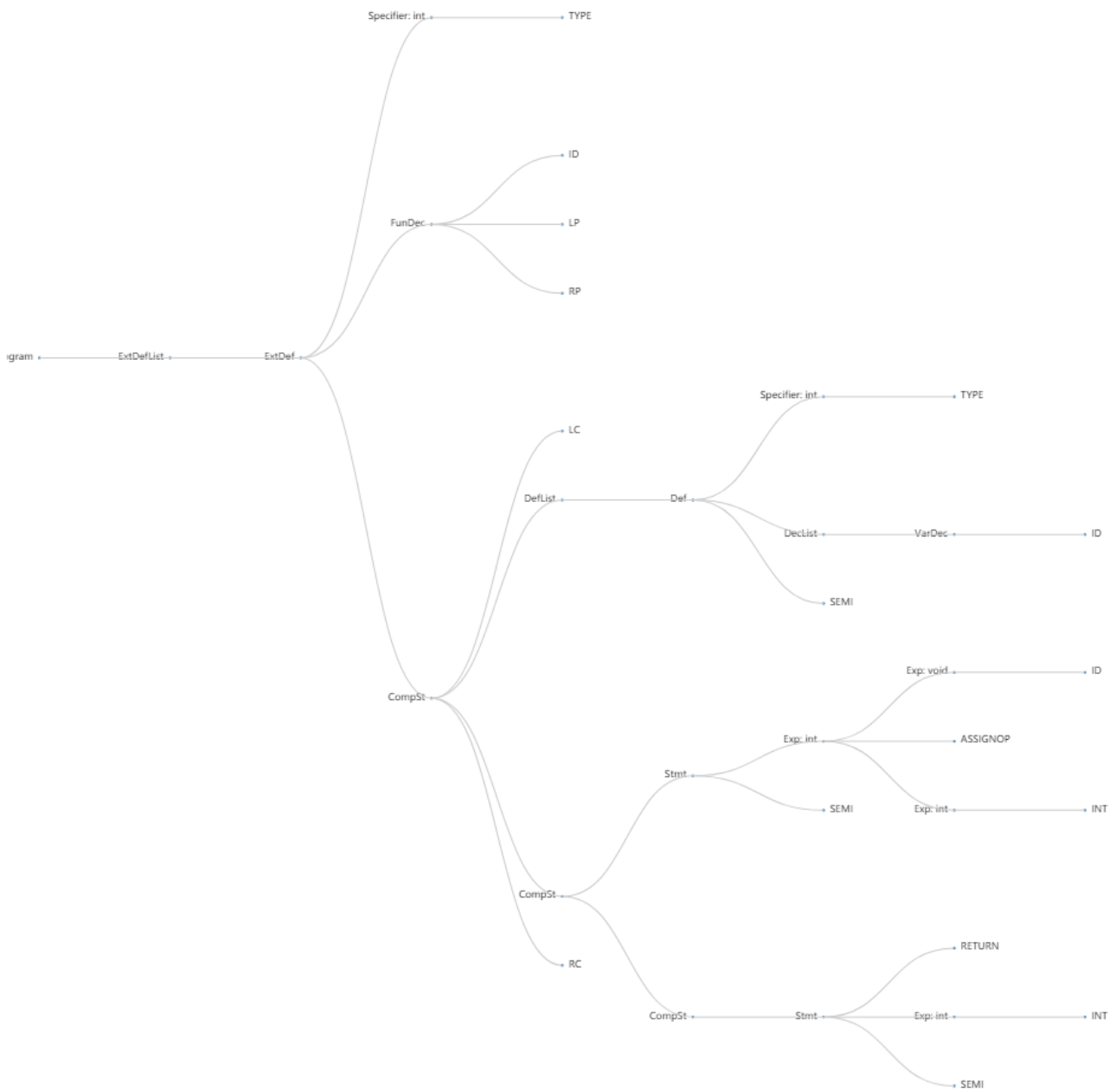
for(int i = 0; i < this->childNum; i++){
    if(this->childNodes[i]){
        root["children"].append(this->childNodes[i]->jsonGen());
    }
}

return root;

```

}

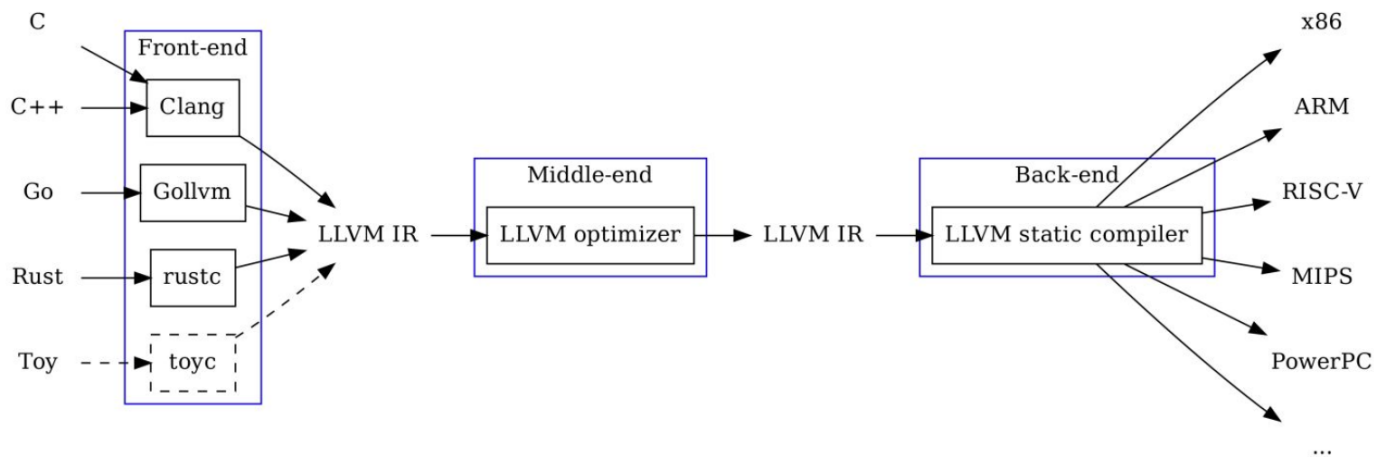
具体效果如下：



第三章 语义分析

3.1 llvm概述

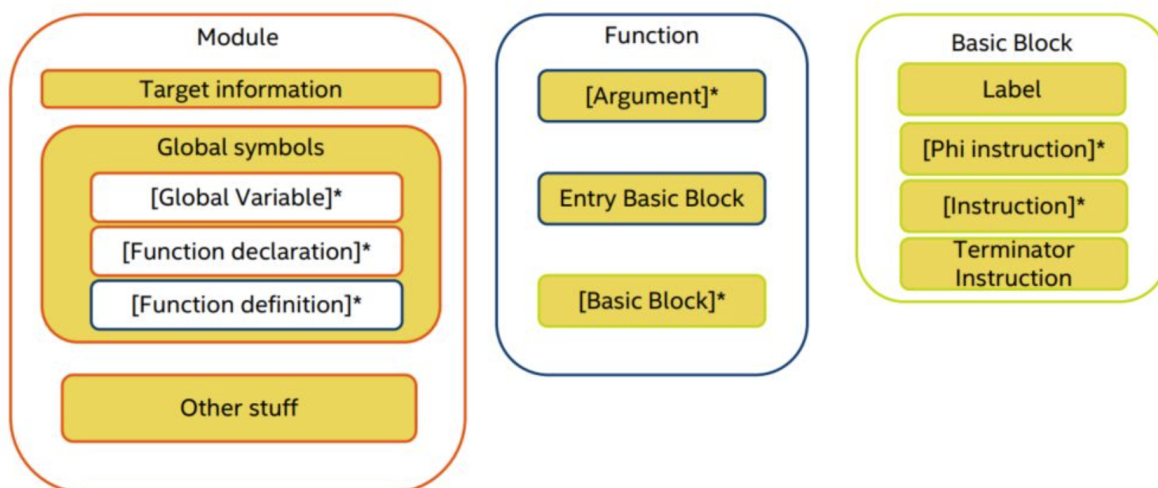
llvm是一组编译器和工具链技术，可用于开发前端任何编程语言和一个后端为任何指令集架构。llvm是围绕独立于语言的中间表示(IR) 设计的，它作为一种可移植的高级汇编语言，可以通过多次通过各种转换进行优化。



3.2 llvm IR

llvm IR是一种中间代码，它可以方便的转化为汇编代码或者可执行文件，我们可以利用llvm来实现llvm IR转化为可执行文件的部分，我们这里只需要完成c--代码转化为IR的过程，llvm提供了相关的api，当我们分析出语法的对应格式之后，我们可以调用相关api来生成对应的中间代码。

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。



3.3 IR 生成

我们通过自顶向下遍历抽象语法树的过程当中生成IR，我们通过判断节点对应的类型来调用不同的IR生成函数，当然我们不需要遍历整个语法树，如我们在遇到函数定义时，`irBuildFun()` 函数会调用相关的其他IR生成函数，此时我们不需要继续向下遍历语法树了，`irBuildFun()` 会自己调用子节点的IR生成函数来处理。

主要IR生成函数如下：

```

class Node {
    ...
public:
    llvm::Value *irBuild();
    llvm::Value *irBuildExp();
    llvm::Value *irBuildFun();
    llvm::Value *irBuildVar();
    llvm::Value *irBuildStmt();
    llvm::Value *irBuildWhile();
    llvm::Value *irBuildIf();
    llvm::Value *irBuildReturn();
    llvm::Value *irBuildCompSt();
    llvm::Value *irBuildRELOP();
    llvm::Value *irBuildPrint();
    llvm::Value *irBuildPrintf();
    llvm::Value *irBuildScan();
    llvm::Value *irBuildAddr();
    ...
}

```

3.3.1 运行时环境设计

llvm IR的生成要依赖下文环境，所以我们需要保存上下文环境。同时我们设计了一个CodeGen类来保存module、函数栈等数据。

静态全局变量：

```

static llvm::LLVMContext context;
static llvm::IRBuilder<> builder(context);

```

3.3.2 符号表设计

我们利用llvm内部的符号表和函数栈来实现。

- 全局变量声明时我们会调用 `llvm::GlobalVariable::GlobalVariable(module, type, isConstant, linkage, initializer, name)`，它会将该全局变量插入全局变量表，之后我们可以使用 `llvm::getGlobalVariable(name)` 来查找。
- 局部变量声明时，我们会将局部变量插入对应函数的符号表



- 查找变量时我们会使用 `codegen::findValue(name)` 查询符号表，它首先会查询当前函数的符号表，若未找到，则查询全局变量符号表。

3.3.3 类型设计

我们这里支持三类类型，第一种是基础类型 `int`、`float`、`bool`、`char`，分别对应llvm中的 `i32`、`float`、`i1`、`i8`；其次我们支持数组。为了方便之后的计算与表示我们定义如下：

```
#define VOID -1
#define VAR 0
#define ARRAY 1
#define FUN 2
#define TYPE_INT 4
#define TYPE_INT_ARRAY 5
#define TYPE_FLOAT 6
#define TYPE_FLOAT_ARRAY 7
#define TYPE_CHAR 8
#define TYPE_CHAR_ARRAY 9
#define TYPE_BOOL 10
#define TYPE_BOOL_ARRAY 11
```

这样当我们声明变量是，首先获取基础的类型，此时如果变量声明如 `a[10]` 之类，它的类型会被声明为 `ARRAY`，否则为 `VAR`，之后我们将变量类型和基础类型相加即可得到这个变量的真实类型。如果要使用指针则声明变量如 `a[]` 即可。

这里我们设计了两个接口，一个是 `int Node::getValueType(Node *node)`，另一个是 `llvm::Value *getLlvmType(int type)`，第一个函数将获取对应节点的类型，第二个函数将类型转化为 `llvm::Value`。

3.3.4 变量声明处理

变量声明主要分为两部分，第一部分是获取变量的基本类型，这里我们使用 `int Node::getValueType()` 获取对应的类型；第二部分是获取变量名并判断变量是数组还是指针抑或是普通变量，这里我们调用 `vector<pair<string, int>> *Node::getNameList(int type)` 获取变量名和其是否是数组或者指针。这里每对 `pair` 中的第二个参数都是 `ARRAY + array_size` 或 `VAR`，当 `array_size` 为0时，我们就认为它是指针。这样便可以确定每个变量的类型及变量名，之后按照变量是否是全局变量插入对应的符号表即可。

除此之外，所有的全局变量都会被初始化为0。

3.3.5 函数定义处理

函数定义过程与变量声明类似，也需要确认返回类型和函数名，函数定义过程中还需要处理形参，这里我们使用 `vector<pair<string, llvm::Type*>> *Node::getParam()` 获取形参的名字和类型。之后我们使用 `static llvm::FunctionType *llvm::FunctionType::get(...)` 创建函数类型，然后使用 `static llvm::Function *llvm::Function::Create(...)` 创建函数。这样创建之后函数的形参都被命名为 `%1` 之类，这不便于我们之后的ir生成，所以这里我们要通过 `void llvm::Value::setName(const llvm::Twine &Name)` 修改形参的名字。（由于部分函数参数过长，所以就使用 `...` 代替了，之后的函数除 `printf` 外都是此含义）

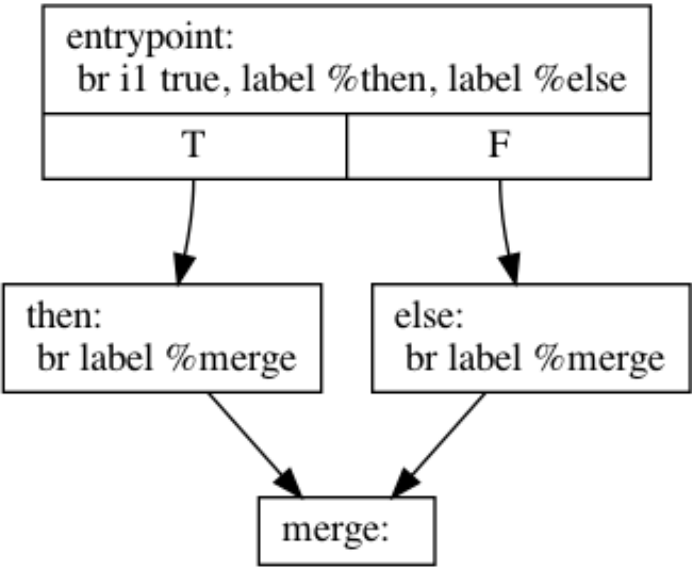
在处理完之后的函数这些之后，会调用对应子节点的 `llvm::Value *Node::irBuildCompSt()` 函数将之后的函数体转化为ir。

3.3.6 字符串常量声明处理

与其他常量不同，普通的 `int`、`float`、`char`、`bool` 常量返回其值即可，而字符串常量则需返回其首地址，我们可以将该地址赋值给一个 `char` 指针，也就是对应 `llvm::Type` 中的 `CharPtr`，显然我们是将该地址存在对应指针值的地址，这样我们就医使用变量来指向该字符串，输出函数不支持字符串数组的输出，所以我们也要隐性转换其类型为 `CharPtr`。

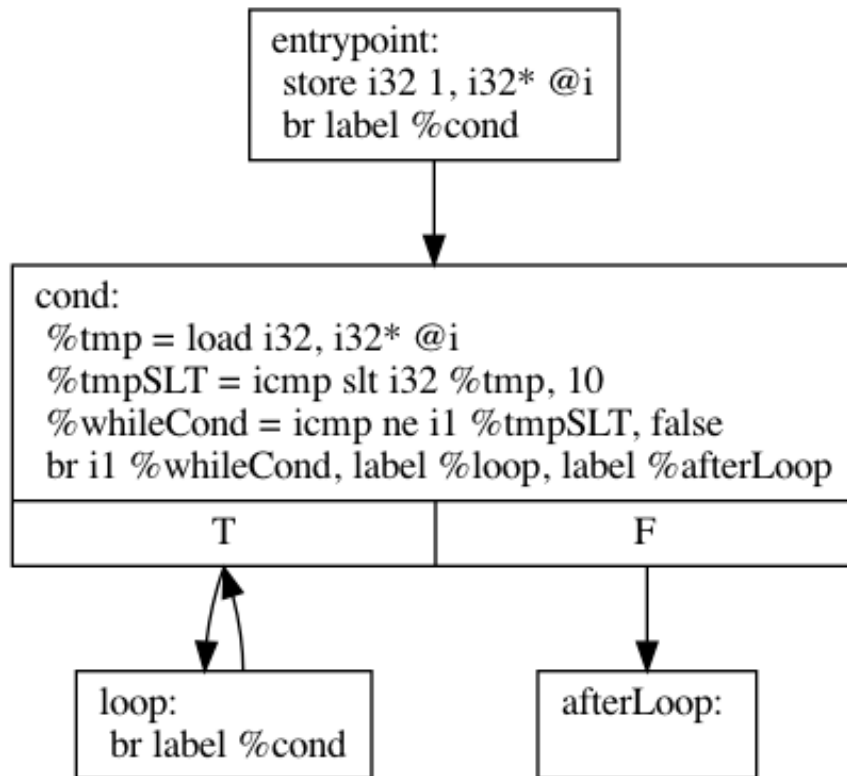
3.3.7 分支语句处理

这里我们实现了 `if` 分支语句，这里 `if` 语句可以分为四个基本块，分别是 `condition` 块、`then` 块、`else` 块，以及最后的 `merge` 块。无论 `if` 语句是否包含 `else`，我们都会实现一个 `else` 块，当 `if` 语句不包含 `else` 块时，我们会创建一个空的 `else` 块。



3.3.8 循环语句处理

这里我们实现的循环语句是 `while` 语句，`while` 语句与 `if` 语句类似，`while` 语句分为三个基本块，`condition` 块、`loop` 块、`afterLoop` 块。我们在这里也实现了 `break` 的功能，实现 `break` 的功能需要我们找到最近的一个 `afterLoop` 块，由于我们将 `break` 当作一个语句处理，所以它的处理和 `while` 语句的处理是分离的，我们无法直接获取当前的 `afterLoop` 块，所以我们维持着一个栈，来保存当前的 `afterLoop` 块，由此我们便可以找到合适的 `afterLoop` 块



3.3.9 类型自动转换处理

在处理表达式语句之前，我们首先需要考虑类型的问题，一方面是不同类型的值的表达式调用的ir指令不同，另一方面，当表达式两边类型不同时，我们如何处理这个问题。这里我们处理的是相关的类型转换问题。

llvm IR提供了许多有关类型转换的指令，如 `trunc`、`zext`、`sext`、`sitofp` 等，这里llvm的相关api也很多，我们使用 `llvm::Value *llvm::IRBuilderBase::CreateCast(llvm::Instruction::CastOps Op, llvm::Value *V, llvm::Type *DestTy, const llvm::Twine &Name = "")` 来实现这个功能。这里参数中的 `Op` 是指调用对应指令的值，这个值的定义在 `Instruction.def` 中，如下所示：

```

FIRST_CAST_INST(38)
HANDLE_CAST_INST(38, Trunc    , TruncInst  ) // Truncate integers
HANDLE_CAST_INST(39, ZExt    , ZExtInst  ) // Zero extend integers
HANDLE_CAST_INST(40, SExt    , SExtInst  ) // Sign extend integers
HANDLE_CAST_INST(41, FPToUI  , FPToUIInst) // floating point -> UInt
HANDLE_CAST_INST(42, FPToSI  , FPToSIInst) // floating point -> SInt
HANDLE_CAST_INST(43, UIToFP  , UIToFPInst) // UInt -> floating point
HANDLE_CAST_INST(44, SIToFP  , SIToFPInst) // SInt -> floating point
HANDLE_CAST_INST(45, FPTrunc , FPTruncInst) // Truncate floating point
HANDLE_CAST_INST(46, FPExt   , FPExtInst ) // Extend floating point
HANDLE_CAST_INST(47, PtrToInt, PtrToIntInst) // Pointer -> Integer
HANDLE_CAST_INST(48, IntToPtr, IntToPtrInst) // Integer -> Pointer
HANDLE_CAST_INST(49, BitCast , BitCastInst) // Type cast
HANDLE_CAST_INST(50, AddrSpaceCast, AddrSpaceCastInst) // addrspace cast
LAST_CAST_INST(50)

```

这里我们主要实现了5中类型转换，分别为 `float --> int`、`int --> float`、`int --> char`、`char --> int`、`char --> float`。考虑到 `float` 一般不会转为 `char`，这里也没有实现有关 `bool` 的转化，主要是在于这些类型转为 `bool`，一般都是采用了 `trunc` 这个指令，这样偶数转为 `bool` 也就是 `i1` 后值为0，那要实现我们期望的转换，就需要实现一个 `if` 语句，这其实和在之后的代码中自己判断并没有任何区别，可能只是简化了代码，但最终结果相差不大，甚至多了几条指令，所以这里并没有实现这一部分。之后也许会简单实现这个功能。

3.3.10 表达式处理

表达式处理，首先是判断表达式的类型，赋值表达式与其他表达式不同。

3.3.10.1 普通表达式处理

普通表达式都只需要处理操作数的值，这里不涉及赋值操作，最后只需要返回一个值即可。我们只需要取到操作数的值，并依据表达式的类型选择对应的方法即可。这里还涉及到类型的隐性转换，我们只需要判断类型，并调用 3.3.9节提供的类型转换函数即可。这里我们调用 `llvm::Value *Node::irBuildExp()`。

3.3.10.2 赋值表达式处理

赋值表达式与前者不同之处在于，前两者二元表达式左侧都是变量的值，但赋值表达式左侧不应当是表达式的值，而是表达式的地址，这样我们才可以将右侧的值存入左侧的地址当中。所以这里我们特意为左侧表达式创建了方法 `llvm::Value *Node::irBuildAddr()` 来获取其地址。

第四章 代码生成

通过我们编译出的 `parse` 可执行文件可以将源代码转化为 `.ll` 文件，我们使用 `llvm-as` 命令用 `.ll` 文件生成 `.bc` 文件，之后使用 `llc -filetype=obj *.bc` 便可生成目标文件，之后使用 `clang` 便可生成可执行文件。

第五章 测试案例

5.1 课程要求测试

我们编写 `.cmm` 源代码，并编译产生 `*.ll` `*.bc` 等文件后再产生 `*.out` 可执行文件。

- quicksort.cmm

```
int array[10000];

int quick_sort(int left, int right)
{
    int i, j, x;
    if (left < right)
    {
        i = left;
        j = right;
        x = array[i];
        while (i < j)
        {
            while(i < j && array[j] > x)
                j = j - 1; /* 从右向左找第一个小于x的数 */
            if(i < j) {
```

```

        array[i] = array[j];
        i = i + 1;
    }

    while(i < j && array[i] < x)
    i = i + 1; /* 从左向右找第一个大于x的数 */
    if(i < j) {
        array[j] = array[i];
        j = j - 1;
    }
}
array[i] = x;
quick_sort(left, i-1); /* 递归调用 */
quick_sort(i+1, right);
}
return 0;
}

int main()
{
    int i;
    int N;
    int left, right;
    scan(N);
    i = 0;
    while(i<N){
        scan(array[i]);
        i = i + 1;
    }

    left = 0;
    right = N - 1;

    quick_sort(left, right);

    i = 0;
    while (i<=right){
        print(array[i]);
        i = i + 1;
    }

    return 0;
}

```

- 结果

```
>>> 🔥 quicksort ./linux-amd64 ./quicksort.out
fixed case 0 (size 0)...pass!
fixed case 1 (size 1)...pass!
fixed case 2 (size 2)...pass!
fixed case 3 (size 2)...pass!
fixed case 4 (size 3)...pass!
fixed case 5 (size 3)...pass!
fixed case 6 (size 3)...pass!
fixed case 7 (size 3)...pass!
fixed case 8 (size 3)...pass!
fixed case 9 (size 4)...pass!
fixed case 10 (size 9)...pass!
fixed case 11 (size 9)...pass!
fixed case 12 (size 10000)...pass!
fixed case 13 (size 10000)...pass!
fixed case 14 (size 4096)...pass!
randomly generated case 0 (size 10000)...pass!
randomly generated case 1 (size 10000)...pass!
randomly generated case 2 (size 10000)...pass!
randomly generated case 3 (size 10000)...pass!
randomly generated case 4 (size 10000)...pass!
randomly generated case 5 (size 10000)...pass!
randomly generated case 6 (size 10000)...pass!
randomly generated case 7 (size 10000)...pass!
randomly generated case 8 (size 10000)...pass!
randomly generated case 9 (size 10000)...pass!
-----
2021-21-14 09:29:18.666
>>> 🔥 quicksort
```

- matrix-multiplication.cmm

```
int arrayA[400]; // 20 * 20
int arrayB[400];
int arrayC[400]; // result

int main()
{
    int MA; // row_A
    int NA; // col_A
    int MB; // row_B
    int NB; // col_B
    int MC; // row_C
    int NC; // col_C
    int i; // row
    int j; // col
    int i_a;
    int i_b;

    MA = 0; // row_A
    NA = 0; // col_A
```

```

MB = 0; // row_B
NB = 0; // col_B
MC = 0; // row_C
NC = 0; // col_C
i = 0; // row
j = 0; // col
i_a = 0;
i_b = 0;

// read matrix A
scan(MA, NA);
while (i<MA){
    while (j<NA){
        scan(arrayA[i*NA+j]);
        j = j + 1;
    }
    j = 0;
    i = i + 1;
}
// whether can mul or not
scan(MB, NB);
if (NA!=MB) {
    print("Incompatible Dimensions");
    return 0;
}

// read matrix B
i = 0;
j = 0;
while (i<MB){
    while (j<NB){
        scan(arrayB[i*NB+j]);
        j = j + 1;
    }
    j = 0;
    i = i + 1;
}

MC = MA;
NC = NB;
i = 0;
j = 0;

while (i<MC){
    while (j<NC){
        // arrayA的第i行和arrayB的第j列相乘
        i_a = 0;
        i_b = j;
        while (i_a<NA){

```

```

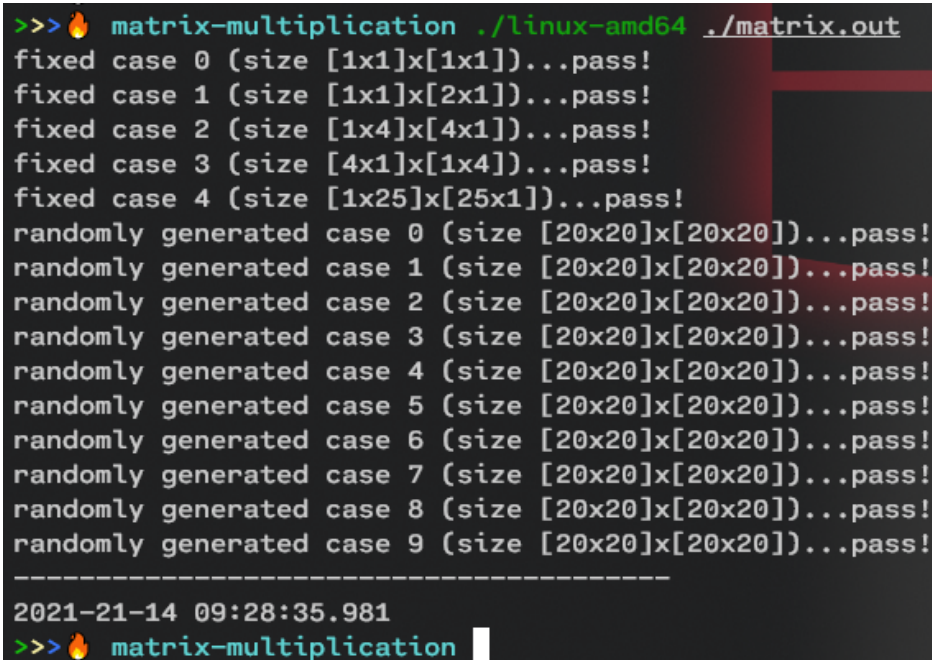
        arrayC[i*NC+j] = arrayC[i*NC+j] + arrayA[i*NA+i_a]*arrayB[i_b];
        i_a = i_a + 1;
        i_b = i_b + NB;
    }
    j = j + 1;
}
j = 0;
i = i + 1;
}

i = 0;
j = 0;
while (i<MC){
    while (j<NC){
        printf("%10d", arrayC[i*NC+j]);
        j = j + 1;
    }
    print("");
    j = 0;
    i = i + 1;
}

return 0;
}

```

- 结果



```

>>> 🔥 matrix-multiplication ./linux-amd64 ./matrix.out
fixed case 0 (size [1x1]x[1x1])...pass!
fixed case 1 (size [1x1]x[2x1])...pass!
fixed case 2 (size [1x4]x[4x1])...pass!
fixed case 3 (size [4x1]x[1x4])...pass!
fixed case 4 (size [1x25]x[25x1])...pass!
randomly generated case 0 (size [20x20]x[20x20])...pass!
randomly generated case 1 (size [20x20]x[20x20])...pass!
randomly generated case 2 (size [20x20]x[20x20])...pass!
randomly generated case 3 (size [20x20]x[20x20])...pass!
randomly generated case 4 (size [20x20]x[20x20])...pass!
randomly generated case 5 (size [20x20]x[20x20])...pass!
randomly generated case 6 (size [20x20]x[20x20])...pass!
randomly generated case 7 (size [20x20]x[20x20])...pass!
randomly generated case 8 (size [20x20]x[20x20])...pass!
randomly generated case 9 (size [20x20]x[20x20])...pass!
-----
2021-21-14 09:28:35.981
>>> 🔥 matrix-multiplication

```

- auto-advisor.cmm

由于相关功能在前两个代码中均已展现，故不再冗述，详见文件夹。

- 结果

```
>>> 🔥 auto-advisor ./linux-amd64 ./auto-advisor.out
fixed case 0...pass!
fixed case 1...pass!
fixed case 2...pass!
fixed case 3...pass!
fixed case 4...pass!
randomly generated case 0...fail!
input curriculum:
c21|2||C
c11|4|c3,c4,c8,c2,c6,c9;c9,c8,c2,c4,c1,c10,c7|F
c3|4||D
c67|1|c2,c25,c52,c28,c43;c63,c22,c41,c13,c50,c31;c26,c21,c36,c23,c40;c24,c37,c12|B
c37|2|c29,c4,c14,c20,c5;c15;c2,c13,c17,c19,c9,c15,c29;c21|C
c77|5|c40,c1;c26,c29,c0;c31,c27,c18,c7;c65;c18,c30|
c78|2|c57|B
```

5.2 常规测试

- testcode0.c 基础类型及功能测试

```
int a;

int main() {
    float b;
    b = 10;
    a = 0;
    a = a + 1;
    a = a + b;
    print("a: ", a, " b: ", b);
}
```

- 中间代码 testcode0.ll

```
; ModuleID = 'main'
source_filename = "main"

@a = private global i32 0
@_Const_String_ = private constant [4 x i8] c"a: \00"
@_Const_String_.1 = private constant [5 x i8] c" b: \00"
@.str = constant [11 x i8] c"%s%d%s%lf\0A\00"

declare i32 @printf(i8*, ...)

declare i32 @scanf(...)

define i32 @main() {
entrypoint:
    %b = alloca float, align 4
    store float 1.000000e+01, float* %b, align 4
    store i32 0, i32* @a, align 4
    %tmpvar = load i32, i32* @a, align 4
    %addtmpi = add i32 %tmpvar, 1
```

```

store i32 %addtmpi, i32* @a, align 4
%tmpvar1 = load i32, i32* @a, align 4
%tmpvar2 = load float, float* %b, align 4
%tmptypecast = sitofp i32 %tmpvar1 to float
%addtmpf = fadd float %tmptypecast, %tmpvar2
%tmptypecast3 = fptosi float %addtmpf to i32
store i32 %tmptypecast3, i32* @a, align 4
%tmpvar4 = load i32, i32* @a, align 4
%tmpvar5 = load float, float* %b, align 4
%tmpdouble = fpext float %tmpvar5 to double
%printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8], [11
x i8]* @.str, i32 0, i32 0), i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@_Const_String_, i32 0, i32 0), i32 %tmpvar4, i8* getelementptr inbounds ([5 x i8],
[5 x i8]* @_Const_String_.1, i32 0, i32 0), double %tmpdouble)
}

```

- testcode1.c 数组、字符串、指针测试

```

int a[5];

int main() {
    float b;
    char str[];
    b = 10;
    str = "hahahaha";
    a[0] = 0;
    a[2] = a[0] + 1;
    a[4] = a[0] + b;
    print("a: ", a[3], " b: ", b);
    printf("%s", str);
}

```

- 中间代码 testcode1.ll

```

; ModuleID = 'main'
source_filename = "main"

@a = private global [5 x i32] zeroinitializer
@_Const_String_ = private constant [9 x i8] c"hahahaha\00"
@_Const_String_.1 = private constant [4 x i8] c"a: \00"
@_Const_String_.2 = private constant [5 x i8] c" b: \00"
@.str = constant [11 x i8] c"%s%d%s%lf\0A\00"
@_Const_String_.3 = private constant [3 x i8] c"%s\00"

declare i32 @printf(i8*, ...)

declare i32 @scanf(...)

```



```

define i32 @main() {
entrypoint:
    %str = alloca i8*, align 8
    %b = alloca float, align 4
    store float 1.000000e+01, float* %b, align 4
    store i8* getelementptr inbounds ([9 x i8], [9 x i8]* @Const_String_, i32 0, i32 0), i8** %str, align 8
    store i32 0, i32* getelementptr inbounds ([5 x i32], [5 x i32]* @a, i32 0, i32 0), align 4
    %tmpvar = load i32, i32* getelementptr inbounds ([5 x i32], [5 x i32]* @a, i32 0, i32 0), align 4
    %addtmpi = add i32 %tmpvar, 1
    store i32 %addtmpi, i32* getelementptr inbounds ([5 x i32], [5 x i32]* @a, i32 0, i32 2), align 4
    %tmpvar1 = load i32, i32* getelementptr inbounds ([5 x i32], [5 x i32]* @a, i32 0, i32 0), align 4
    %tmpvar2 = load float, float* %b, align 4
    %tmptypecast = sitofp i32 %tmpvar1 to float
    %addtmpf = fadd float %tmptypecast, %tmpvar2
    %tmptypecast3 = fptosi float %addtmpf to i32
    store i32 %tmptypecast3, i32* getelementptr inbounds ([5 x i32], [5 x i32]* @a, i32 0, i32 4), align 4
    %tmpvar4 = load i32, i32* getelementptr inbounds ([5 x i32], [5 x i32]* @a, i32 0, i32 3), align 4
    %tmpvar5 = load float, float* %b, align 4
    %tmpdouble = fpext float %tmpvar5 to double
    %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8], [11 x i8]* @.str, i32 0, i32 0), i8* getelementptr inbounds ([4 x i8], [4 x i8]* @Const_String_.1, i32 0, i32 0), i32 %tmpvar4, i8* getelementptr inbounds ([5 x i8], [5 x i8]* @Const_String_.2, i32 0, i32 0), double %tmpdouble)
    %tmpvar6 = load i8*, i8** %str, align 8
    %printf7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @Const_String_.3, i32 0, i32 0), i8* %tmpvar6)
}

```

- 错误代码测试（以重定义为例）

```

int a;
int a;

int main() {
    print(a);
}

```

- 结果

```
>>> 🔥 test ./parse < testcode2.c > testcode2.ll
terminate called after throwing an instance of 'std::logic_error'
  what():  Redefined variable: a
[1]    955 abort      ./parse < testcode2.c > testcode2.ll
>>> 🔥 test
```