实验四：代码生成

主讲教师：陈安龙

实验4内容简介

- 查看指令手册 LLVMRef.mht
- 用 clang 学习 llvm 虚拟指令：

  ① 编写简单的 C 语言程序 test.c

  ② 用 clang -emit-llvm -S test.c -o test.ll

  ③ 生成该文件对应的 llvm 指令
- 学习示例程序，理解代码生成过程

实验4内容简介

- 完成算术表达式、逻辑表达式、赋值语句、条件语句、循环语句对应的代码生成，函数命名分别为：

  ① genArithmeticExpr：生成算术表达式的LLVM IR指令。

  ② genLogicExpr：生成逻辑表达式的LLVM IR指令。

  ③ genAssignStmt：生成赋值语句的LLVM IR指令。

  ④ genIfStmt：生成条件语句的LLVM IR指令。

  ⑤ genWhileStmt：生成循环语句的LLVM IR指令。

- 函数参数都为：(past node, char* result)
- node 为相关类型的结点
- result 为用来保存LLVM指令，每行只放一条指令；

实验安排要求

尽量通过全部测试用例

需完整总结从词法分析到代码生成各步骤的

技术要点及相关设计，并体现在实验报告中

提交方式

icoding 平台提交，包括实验4全部代码及实验报告

提交截止日期:

全年级统一，以icoding平台为准

LLVM IR简介

LLVM IR（Intermediate Representation）是LLVM编译器基础设施中的一种中间表示，旨在为多种编程语言提供一个平台无关的低级表示。LLVM IR具有强大的表达能力，支持多种优化和代码生成。

LLVM IR特点

① 平台无关性：LLVM IR不依赖于特定的硬件架构，使得编译器可以生成适用于多种平台的代码。

② 静态单赋值（SSA）形式：每个变量在其作用域内只被赋值一次，这使得数据流分析和优化变得更加简单。

③ 强类型系统：LLVM IR具有强类型系统，支持基本数据类型（如整数、浮点数、指针等）和复合数据类型（如结构体、数组等）。

④ 丰富的指令集：LLVM IR提供了丰富的指令集，支持算术运算、逻辑运算、控制流、内存操作等。

LLVM IR的结构

LLVM IR可以以文本格式或二进制格式表示。文本格式的LLVM IR通常以.ll文件扩展名保存，结构如下：

① 模块（Module）：LLVM IR的顶层结构，包含全局变量、函数和其他定义。

② 函数（Function）：模块中的每个函数都有一个名称、返回类型和参数类型。

③ 基本块（Basic Block）：函数由多个基本块组成，每个基本块是一个顺序执行的代码序列，包含控制流指令。

④ 指令（Instruction）：基本块中的每条指令执行特定的操作，如算术运算、内存访问、控制流等。

LLVM IR的基本语法

- 注释：以;开头的行是注释。
- 类型：LLVM IR支持多种数据类型，如i32（32位整数）、float（浮点数）、double（双精度浮点数）、i8*（指向8位整数的指针）等。
- 指令格式：指令通常以操作符开头，后跟操作数。例如：

```
%result = add i32 %a, %b
```

算术指令

- 加法：add指令用于两个操作数的加法。

```
%result = add i32 %a, %b
```

- 减法：sub指令用于两个操作数的减法。

```
%result = sub i32 %a, %b
```

- sdiv指令用于执行有符号整数除法；udiv指令用于执行无符号整数除法。

```
%result_sdiv = sdiv i32 %a, %b  ; 计算 %a / %b（有符号除法）

%result_udiv = udiv i32 %c, %d  ; 计算 %c / %d（无符号除法）
```

浮点运算指令

- 浮点加法：fadd指令用于执行浮点加法。

```
%result = fadd float %a, %b  ; 计算 %a + %b
```

- 浮点减法：fsub指令用于执行浮点减法。。

```
%result = fsub float %a, %b  ; 计算 %a - %b
```

- fdiv指令用于执行浮点除法；fmul指令用于执行浮点乘法。

```
%result = fdiv float %a, %b  ; 计算 %a / %b

%result = fmul float %a, %b  ; 计算 %a * %b
```

逻辑指令：

- 与运算：and指令用于按位与运算。

  %result = and i32 %a, %b

- 或运算：or指令用于按位或运算。

  %result = or i32 %a, %b

- not指令用于执行按位非运算。

  %result_not = not i1 %condition ；计算 %condition 的非

比较运算：

- 整数比较：icmp指令用于整数比较。

```
%result_eq = icmp eq  i32 %a, %b  ; 等于比较

%result_gt = icmp gt i32 %a, %b  ; 大于比较

%result_ge = icmp ge i32 %a, %b  ; 大于等于比较

%result_lt = icmp lt i32 %a, %b   ; 小于比较

%result_le = icmp le i32 %a, %b  ; 小于等于比较
```

浮点比较运算：

- fcmp指令用于比较两个浮点数。
- 语法：%result = fcmp <predicate> <type> <operand1>, <operand2>
- <predicate>：比较条件，如oeq（等于）、ogt（大于）、oge（大于等于）、olt（小于）、ole（小于等于）等。
- 示例：

① %result = fcmp oeq float %a, %b

; 如果 %a 等于 %b，则 %result_eq 为 true (1)，否则为 false (0)

② %result = fcmp ogt float %a, %b

; 如果 %a 大于 %b，则 %result_gt 为 true (1)，否则为 false (0)

控制流指令：

- 条件跳转：br指令用于条件跳转。

```
br i1 %condition, label %true_block, label %false_block
```

- 无条件跳转：br指令用于无条件跳转。

```
br label %next_block
```

内存操作指令：

- 加载：load指令用于从内存中加载值。

① 语法：%value = load <type>, <type>* <pointer>

② 示例：%loaded_value = load i32, i32* %ptr ；从 %ptr 指向的内存位置加载值

- 存储：store指令用于将值存储到内存中。

① 语法：store <type> <value>, <type>* <pointer>

② 示例：store i32 %value, i32* %ptr ；将值存储到 %ptr 指向的内存位置

内存分配指令：

- alloca指令用于在栈上分配内存。它的语法如下：

① 语法：%ptr = alloca <type>, <alignment>

<type>：要分配的类型。

<alignment>（可选）：内存对齐。

② 示例：%ptr = alloca i32 align 4; 在栈上分配一个32位整数

函数声明：

- 函数声明用于定义函数的签名，包括返回类型、函数名称和参数类型。函数声明不会包含函数的实现。

① 语法：declare <return_type> @<function_name>(<parameter_types>)

<return_type>：函数的返回类型。

@<function_name>：函数的名称，前面带有@符号。

<parameter_types>：函数参数的类型列表，多个参数用逗号分隔。

② 示例：declare i32 @add(i32, i32); 简单的函数声明，声明一个返回i32类型并接受两个i32类型参数的函数。

函数定义：

- 函数定义包含函数的实现，包括函数体和指令。

① 语法：define <return_type> @<function_name>(<parameter_types>) {

<function_body>

}

② 示例：简单的函数定义，定义一个返回两个整数和的函数：

```
define i32 @add(i32 %a, i32 %b) {
entry:
    %sum = add i32 %a, %b
    ret i32 %sum
}
```

函数调用：

- 函数调用使用call指令。调用指令用于执行已声明或定义的函数，并获取返回值。

① 语法：%result = call <return_type> @<function_name>(<arguments>)

%result：存储函数返回值的变量。

<return_type>：函数的返回类型。

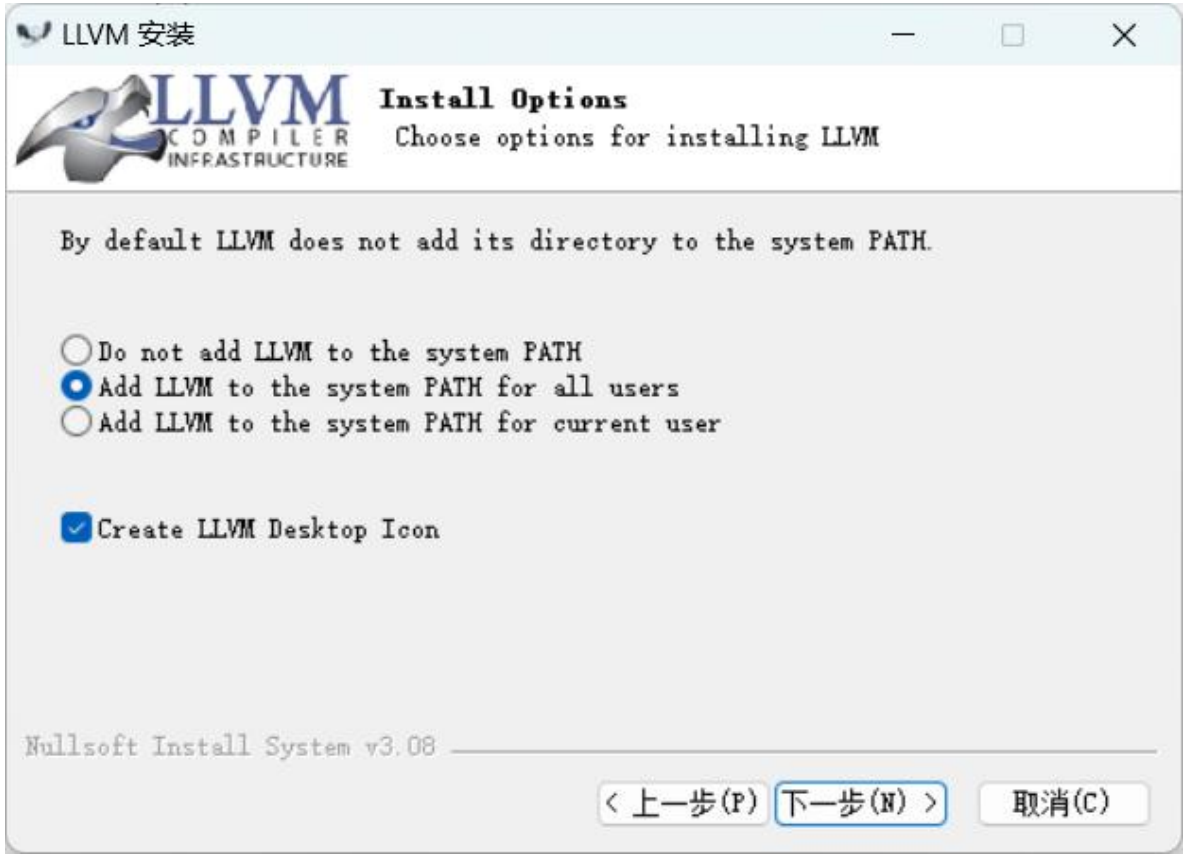
@<function_name>：要调用的函数名称。

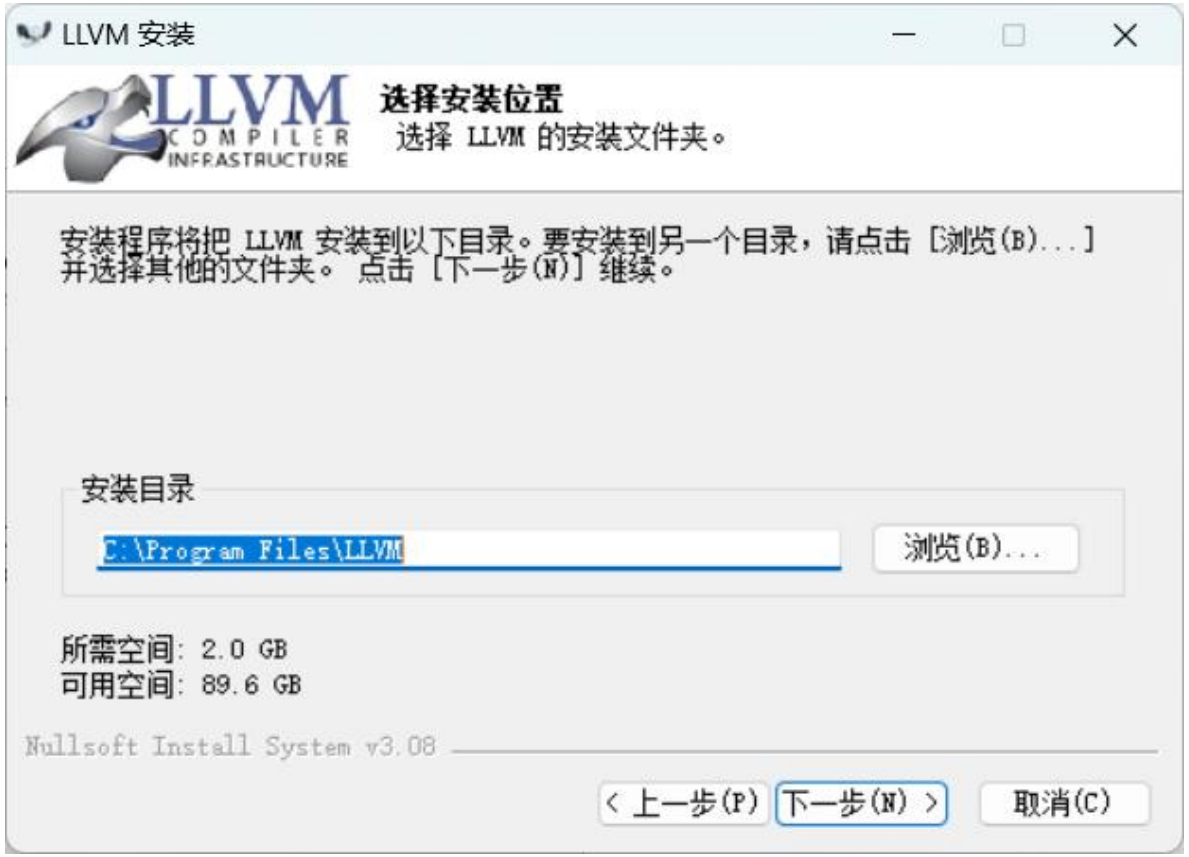
<arguments>：传递给函数的参数列表。}

② 示例：调用之前定义的add函数的示例：

```
%result = call i32 @add(i32 5, i32 10) ；调用 add 函数，传递 5 和 10
```

安装LLVM


LLVM 安装
Install Options
Choose options for installing LLVM
By default LLVM does not add its directory to the system PATH.
Do not add LLVM to the system PATH
Add LLVM to the system PATH for all users
Add LLVM to the system PATH for current user
Create LLVM Desktop Icon
Nullsoft Install System v3.08
< 上一步(P)
下一步(N) >
取消(C)


LLVM 安装
选择安装位置
选择 LLVM 的安装文件夹。
安装程序将把 LLVM 安装到以下目录。要安装到另一个目录，请点击 [浏览(B)...] 并选择其他的文件夹。 点击 [下一步(N)] 继续。
安装目录
C:\Program Files\LLVM
浏览(B)...
所需空间: 2.0 GB
可用空间: 89.6 GB
Nullsoft Install System v3.08
< 上一步(P)
下一步(N) >
取消(C)

安装LLVM


LLVM 安装
选择开始菜单文件夹
选择开始菜单文件夹，用于创建程序的快捷方式。
选择开始菜单文件夹，用于创建程序的快捷方式。你也可以输入自定义名称，创建新文件夹。
LLVM
360安全浏览器
360安全中心
Accessibility
Accessories
Administrative Tools
Android Studio
C-Free 5
C-Free5
CMake
Eclipse
EV录屏
不要创建快捷方式(N)
Nullsoft Install System v3.08
< 上一步(P)
安装(I)
取消(C)

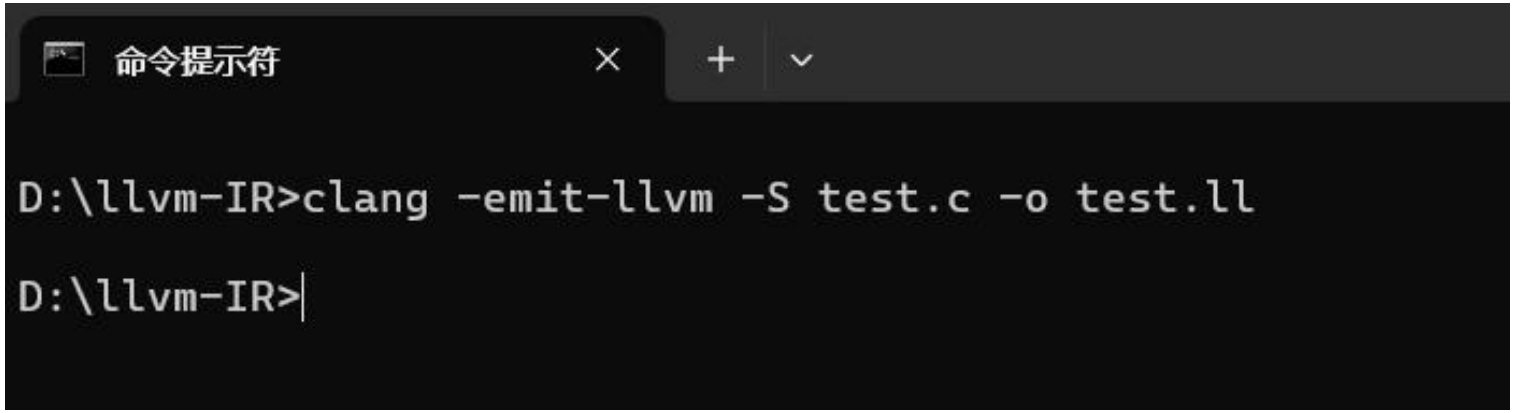

LLVM 安装
正在安装
LLVM 正在安装，请稍候。
解压缩: clang++.exe... 38%
显示详情(D)
Nullsoft Install System v3.08
< 上一步(P)
下一步(N) >
取消(C)

假如编写test.c程序

```
int mytest()
{
    inta = 5;
    intb = 3;
    intc;
    if (a > b)
    {
        c = a * b;
        a = b + 2;
    }
    else
        c = a + b;
    while (a <= c)
    {
        a = a + b;
        b = b - 1;
    }
    returnc;
}
```

clang 生成中间代码

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



test.c的中间代码如下：

test.ll

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-windows-msvc19.42.34321"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @mytest() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 5, ptr %1, align 4
  store i32 3, ptr %2, align 4
  %4 = load i32, ptr %1, align 4
  %5 = load i32, ptr %2, align 4
  %6 = icmp sgt i32 %4, %5
  br i1 %6, label %7, label %13

```

```
int mytest()
{
    inta = 5;
    intb = 3;
    intc;
    if (a > b)
    {
        c = a * b;
        a = b + 2;
    }
    else
        c = a + b;
    while (a <= c)
    {
        a = a + b;
        b = b - 1;
    }
    returnc;
}
```

test.c的中间代码如下：

```
7:                                                ; preds = %0
  %8 = load i32, ptr %1, align 4
  %9 = load i32, ptr %2, align 4
  %10 = mul nsw i32 %8, %9
  store i32 %10, ptr %3, align 4
  %11 = load i32, ptr %2, align 4
  %12 = add nsw i32 %11, 2
  store i32 %12, ptr %1, align 4
  br label %17

13:                                               ; preds = %0
  %14 = load i32, ptr %1, align 4
  %15 = load i32, ptr %2, align 4
  %16 = add nsw i32 %14, %15
  store i32 %16, ptr %3, align 4
  br label %17

17:                                               ; preds = %13, %7
  br label %18
```

```
int mytest()
{
    inta = 5;
    intb = 3;
    intc;
    if (a > b)
    {
        c = a * b;
        a = b + 2;
    }
    else
        c = a + b;
    while (a <= c)
    {
        a = a + b;
        b = b - 1;
    }
    returnc;
}
```

test.c的中间代码如下：

```
34
35  17:                                               ; preds = %13, %7
36    br label %18
37
38  18:                                               ; preds = %22, %17
39    %19 = load i32, ptr %1, align 4
40    %20 = load i32, ptr %3, align 4
41    %21 = icmp sle i32 %19, %20
42    br i1 %21, label %22, label %28
43
44  22:                                               ; preds = %18
45    %23 = load i32, ptr %1, align 4
46    %24 = load i32, ptr %2, align 4
47    %25 = add nsw i32 %23, %24
48    store i32 %25, ptr %1, align 4
49    %26 = load i32, ptr %2, align 4
50    %27 = sub nsw i32 %26, 1
51    store i32 %27, ptr %2, align 4
52    br label %18, !llvm.loop !5
53
54  28:                                               ; preds = %18
55    %29 = load i32, ptr %3, align 4
56    ret i32 %29
57  }
58
```

```
int mytest()
{
    inta = 5;
    intb = 3;
    intc;
    if (a > b)
    {
        c = a * b;
        a = b + 2;
    }
    else
        c = a + b;
    while (a <= c)
    {
        a = a + b;
        b = b - 1;
    }
    returnc;
}
```

示例程序

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedefstruct_ast *past;
enum_node_type
{
    STRUCT_DECL = 2,
    UNION_DECL = 3,
    ENUM_DECL = 5,
    FIELD_DECL = 6,
    ENUM_CONSTANT_DECL = 7,
    FUNCTION_DECL = 8,
    VAR_DECL = 9,
    PARM_DECL = 10,
    TYPEDEF_DECL = 20,
    TYPE_ALIAS_DECL = 36,
    MEMBER_REF = 47,
    LABEL_REF = 48,
    OVERLOADED_DECL_REF = 49,
    VARIABLE_REF = 50,
    UNEXPOSED_EXPR = 100,
    DECL_REF_EXPR = 101,
    MEMBER_REF_EXPR = 102,
    CALL_EXPR = 103,
    BLOCK_EXPR = 105,
    INTEGER_LITERAL = 106,
    FLOATING_LITERAL = 107,
    STRING_LITERAL = 109,
    CHARACTER_LITERAL = 110,
    PAREN_EXPR = 111,
    UNARY_OPERATOR = 112,
    ARRAY_SUBSCRIPT_EXPR = 113,
    BINARY_OPERATOR = 114,
    COMPOUND_ASSIGNMENT_OPERATOR = 115,
    CONDITIONAL_OPERATOR = 116,
    CSTYLE_CAST_EXPR = 117,
    COMPOUND_LITERAL_EXPR = 118,
    INIT_LIST_EXPR = 119,
    ADDR_LABEL_EXPR = 120,
    UNEXPOSED_STMT = 200,
    LABEL_STMT = 201,
    COMPOUND_STMT = 202,
    CASE_STMT = 203,
    DEFAULT_STMT = 204,
    IF_STMT = 205,
    SWITCH_STMT = 206,
    WHILE_STMT = 207,
    DO_STMT = 208,
    FOR_STMT = 209,
    GOTO_STMT = 210,
    INDIRECT_GOTO_STMT = 211,
    CONTINUE_STMT = 212,
    BREAK_STMT = 213,
    RETURN_STMT = 214,
    NULL_STMT = 230,
    DECL_STMT = 231,
    TRANSLATION_UNIT = 300,
};
```

示例程序

```
typedef enum _node_type node_type;
struct _ast
{
    intivalue;          // 整数值
    floatfvalue;        // 浮点值
    char *svalue;        // 字符串值
    node_type nodeType; // 节点类型
    pastleft;           // 左子树
    pastright;          // 右子树
    pastif_cond;        // 条件
    pastnext;           // 下一个节点
};
typedef struct _ast *past;
```

示例程序

```
// 生成唯一标签
char *new_label()
{
    char *label = (char *)malloc(20);
    sprintf(label, "label%d", label_counter++);
    returnlabel;
}
// 生成算术表达式的LLVM IR
voidgenArithmeticExpr(pastnode, char *result)
{
    chartemp[256];
    if (node->nodeType == BINARY_OPERATOR)
    {
        // 生成左操作数的LLVM IR
        genArithmeticExpr(node->left, result);
        // 生成右操作数的LLVM IR
        genArithmeticExpr(node->right, result);
        // 生成算术操作的LLVM IR
        sprintf(temp, "%%temp%d = add i32 %%result1, %%result2\n", label_counter);
        strcat(result, temp);
        sprintf(result + strlen(result), "%%result = %%temp%d\n", label_counter);
    }
    elseif (node->nodeType == INTEGER_LITERAL)
    {
        sprintf(temp, "%%result = add i32 %d, 0\n", node->ivalue);
        strcat(result, temp);
    }
}
```

示例程序

```
// 示例主函数
int main()
{    charresult[1024] = {0};
     pastnode1 = (past)malloc(sizeof(struct_ast));
     node1->nodeType = BINARY_OPERATOR;
     node1->left = (past)malloc(sizeof(struct_ast));
     node1->left->nodeType = INTEGER_LITERAL;
     node1->left->ivalue = 5;
     node1->right = (past)malloc(sizeof(struct_ast));
     node1->right->nodeType = INTEGER_LITERAL;
     node1->right->ivalue = 10;
     genArithmeticExpr(node1, result);
     printf("%s\n", result);
     // 清理内存
     free(node1->left);
     free(node1->right);
     free(node1);
     return0;
}
```