中山大學 SUN YAT-SEN UNIVERSITY

计算机学院 COMPUTER SCIENCE AND ENGINEERING

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

编译器构造实验

实验三 中间代码生成

Yat Compiler Construction with AI

yatcc-ai.com

deepseek NSCC GZ Starlight

中山大学 计算机学院

国家超级计算广州中心

2025.4

www.nscc-gz.cn

OUTLINE

目　录

中山大学计算机学院

School of Computer Science & Engineering

- 目前所进行实验之概览

```
const int a = 10;
int b = 5;

int main() {
  if(b < a)
    return 1;
  return 0;
}
```

源代码

Task1

clang -cc1 -dump-tokens

复活机制

```
const 'const'     [StartOfLine]  Loc=<test.c:1:1>
int 'int'     [LeadingSpace] Loc=<test.c:1:7>
identifier 'a'    [LeadingSpace] Loc=<test.c:1:11>
equal '='     [LeadingSpace] Loc=<test.c:1:13>
numeric_constant '10'     [LeadingSpace] Loc=<test.c:
semi ';'          Loc=<test.c:1:17>
int 'int'     [StartOfLine]  Loc=<test.c:2:1>
identifier 'b'    [LeadingSpace] Loc=<test.c:2:5>
equal '='     [LeadingSpace] Loc=<test.c:2:7>
numeric_constant '5'      [LeadingSpace] Loc=<test.c:
semi ';'          Loc=<test.c:2:10>
int 'int'     [StartOfLine]  Loc=<test.c:4:1>
```

Token 流

Task2

clang -cc1 -ast-dump

复活机制

```
{
  "id": "0x558ceeb59598",
  "kind": "VarDecl",
  "loc": {
    "offset": 22,
    "line": 2,
    "col": 5,
    "tokLen": 1
  },
  "range": {
    "begin": {
      "offset": 18,
      "col": 1,
      "tokLen": 3
    },
```

JSON 格式的抽象语法树

Task3 流程：JSON抽象语法树 ➔ LLVM IR

Json2Asg.hpp/cpp
class Json2Asg;

助教 ☑

ASG

EmitIR.hpp/cpp
class EmitIR;

同学们的任务在此

INCOMPLETE

```
@a = dso_local constant i32 10, align 4
@b = dso_local global i32 5, align 4

; Function Attrs: noinline nounwind optnone uwtable
▷ Run | ▷Debug
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, ptr %retval, align 4
  %0 = load i32, ptr @b, align 4
  %cmp = icmp slt i32 %0, 10
  br i1 %cmp, label %if.then, label %if.end
```

LLVM IR

- ASG 回顾
  - 顶层节点：TranslationUnit，对应 LLVM IR 中的 Module
    - 多个 Decl 节点，函数与全局变量
      - FunctionDecl：函数声明
        - Stmt：变量声明、If语句，While语句，Return语句，……
        - Expr：二元、一元、字面量等等
        - Type：说明符（int，void）和限定符（const）
        - TypeExpr：表达各种复合类型
        - CompoundStmt：多条 Stmt 复合，花括号{}包裹
      - VarDecl：变量声明，函数外部为全局变量，内部为局部变量

```
struct TranslationUnit : Obj
{
  std::vector<Decl*> decls;

private:
  void __mark__(Mark mark) override;
};
```

- Task2
  - Typing：语义分析，Token 流 ➔ ASG 抽象语义图
  - Asg2Json：ASG ➔ JSON 格式的 AST 抽象语法树
- Task3
  - Json2Asg：JSON AST ➔ ASG，对应 Task2 的 Typing ☑
  - EmitIR：ASG ➔ LLVM IR，对应 Task2 的Asg2Json
    - 以 ASG 中的 TranslationUnit 为起点，遍历其中声明
    - 递归遍历各个子节点
    - 将各个节点逐步翻译成对应的 LLVM IR INCOMPLETE

```
struct TranslationUnit : Obj
{
  std::vector<Decl*> decls;

private:
  void __mark__(Mark mark) override;
};
```

- Task3 总体任务
  - 将ASG翻译成可以被 clang 识别、编译并执行的 LLVM IR
    - LLVM IR 可直接由 LLVM 工具链中的 lli 直接执行
      - `lli test.ll`

- Task3 评分
  - 不要求 Task3 生成的 LLVM IR 与 clang 生成的完全一致
  - 将 Task3 生成的 LLVM IR 进行链接后执行，对比程序**返回值**与**输出**
    - 与 clang 的运行结果一致 ➔ 测例通过，满分💯

- 体现在代码之中
  - 补全 EmitIR.cpp 和 EmitIR.hpp 两个文件
    - 其他文件无需改动，已足够完成实验
  - 利用 LLVM 提供的 API 完成 ASG ➔ LLVM IR 的翻译
    - 借助 YatCC 或者 LLVM 官方文档（以后者为准）

OUTLINE

目　录

中山大学计算机学院

School of Computer Science & Engineering

- LLVM IR（LLVM Intermediate Representation）是什么？
  - 三种表现形式
    - 处于内存中（In-Memory）的编译内部中间表示
    - 位于磁盘中的位码/二进制（Bitcode）表示，文件后缀为 .bc
    - 人类可读的汇编语言表示，文件后缀为 .ll
      - Task3 要求生成的 LLVM IR 形式
  - M 种编程语言，N 种目标目标平台/架构
    - 不统一中间表示
      - 分别单独实现 M * N 个编译器，很多重复性工作
    - 统一中间表示
      - 只需实现 M 个前端 和 N 个后端
      - 中端优化器可以复用
      - Rust 基于LLVM
        - Rust code ➔ LLVM IR ➔ LLVM Optimizer ➔ LLVM Backend

```
clang test.c -emit-llvm -o test.bc
```

.bc

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

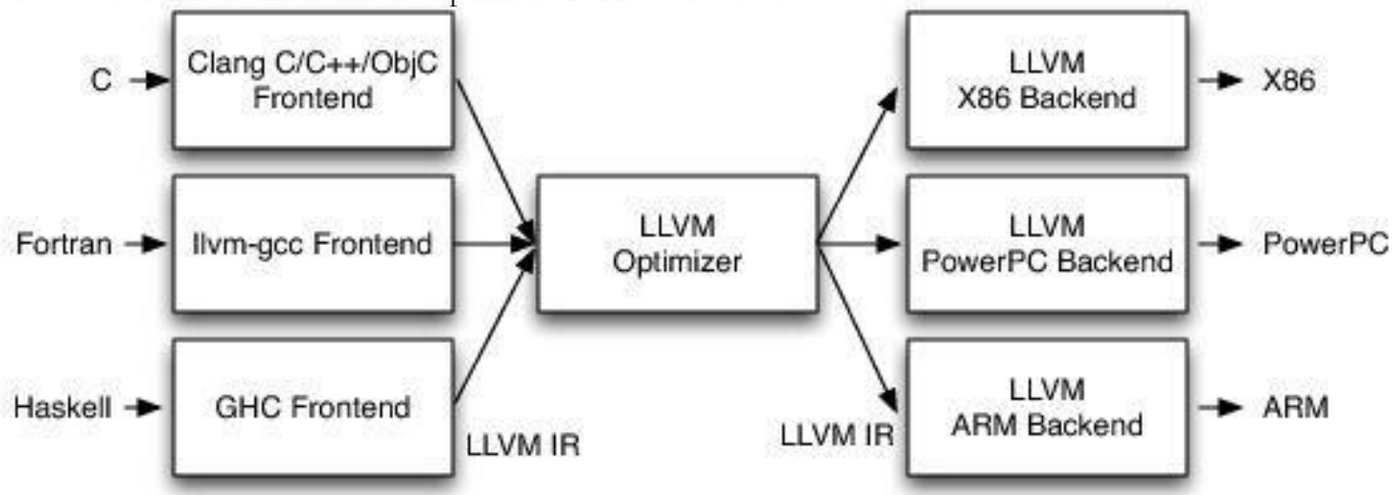
.ll

```
rustc test.rs -emit-llvm -o test.ll
```

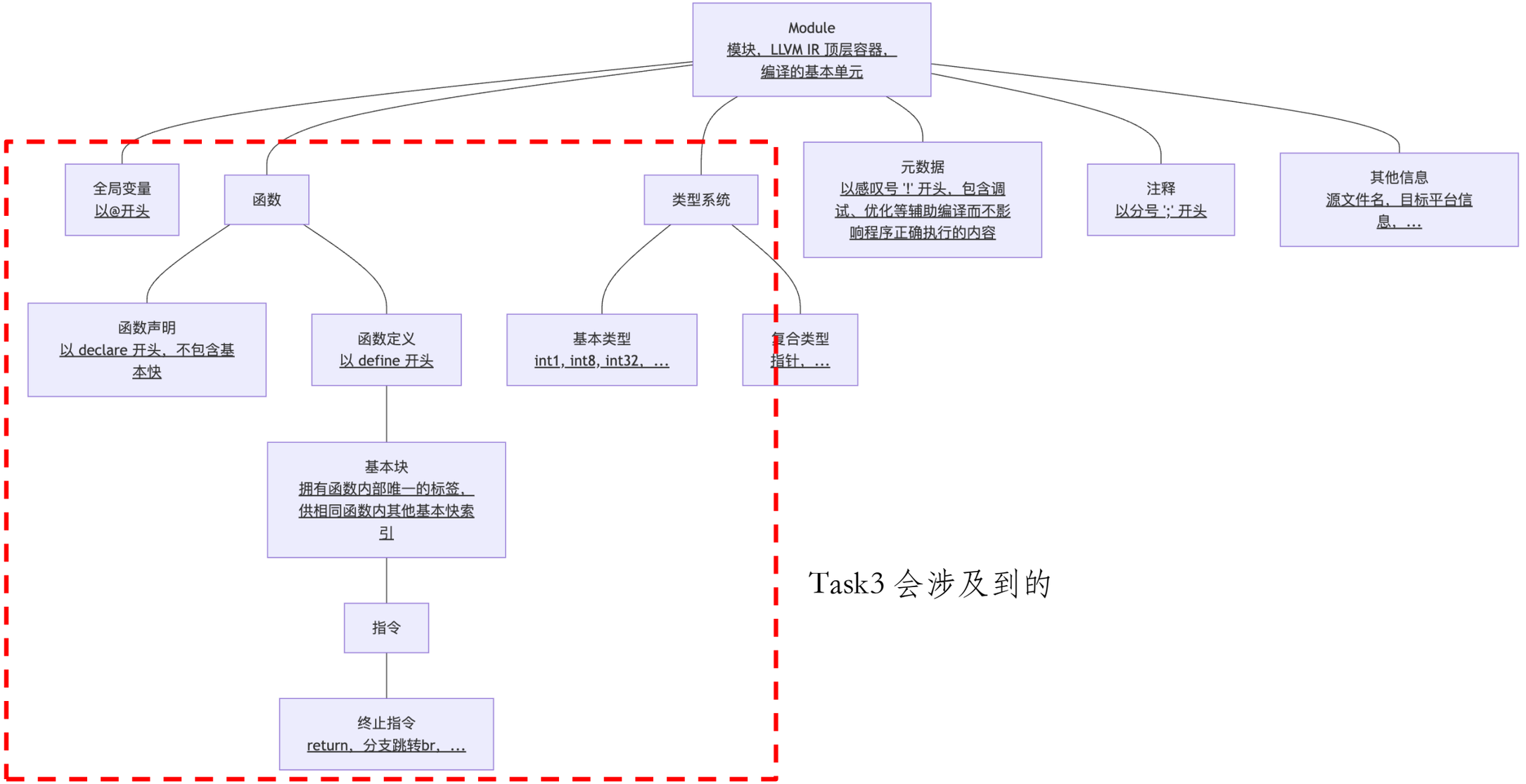
.ll

```
rustc test.rs -emit-llvm-bc -o test.bc
```

.bc



- LLVM IR 结构简介



Task3 会涉及到的

LLVM IR 具体例子

```
const int a = 10;
int b = 5;

int main() {
  if(b < a)
    return 1;
  return 0;
}
```

源代码

clang test.c -emit-llvm -S

源文件名/模块名、目标平台信息
- 数据布局：
  - 大小端、对齐方式等
- 目标平台信息三元祖

全局变量定义与初始化

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:12
target triple = "x86_64-unknown-linux-gnu"

@a = dso_local constant i32 10, align 4
@b = dso_local global i32 5, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, ptr %retval, align 4
  %0 = load i32, ptr @b, align 4
  %cmp = icmp slt i32 %0, 10
  br i1 %cmp, label %if.then, label %if.end

if.then:
  store i32 1, ptr %retval, align 4
  br label %return

if.end:
  store i32 0, ptr %retval, align 4
  br label %return

return:
  %1 = load i32, ptr %retval, align 4
  ret i32 %1
}

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all"

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"clang version 18.1.8 (https://github.com/arcsysu/YatCC.git 079a96
```

类型

函数定义

基本块
- entry为标签
- 函数入口基本块

指令

; preds = %entry

注释

终结指令：跳转

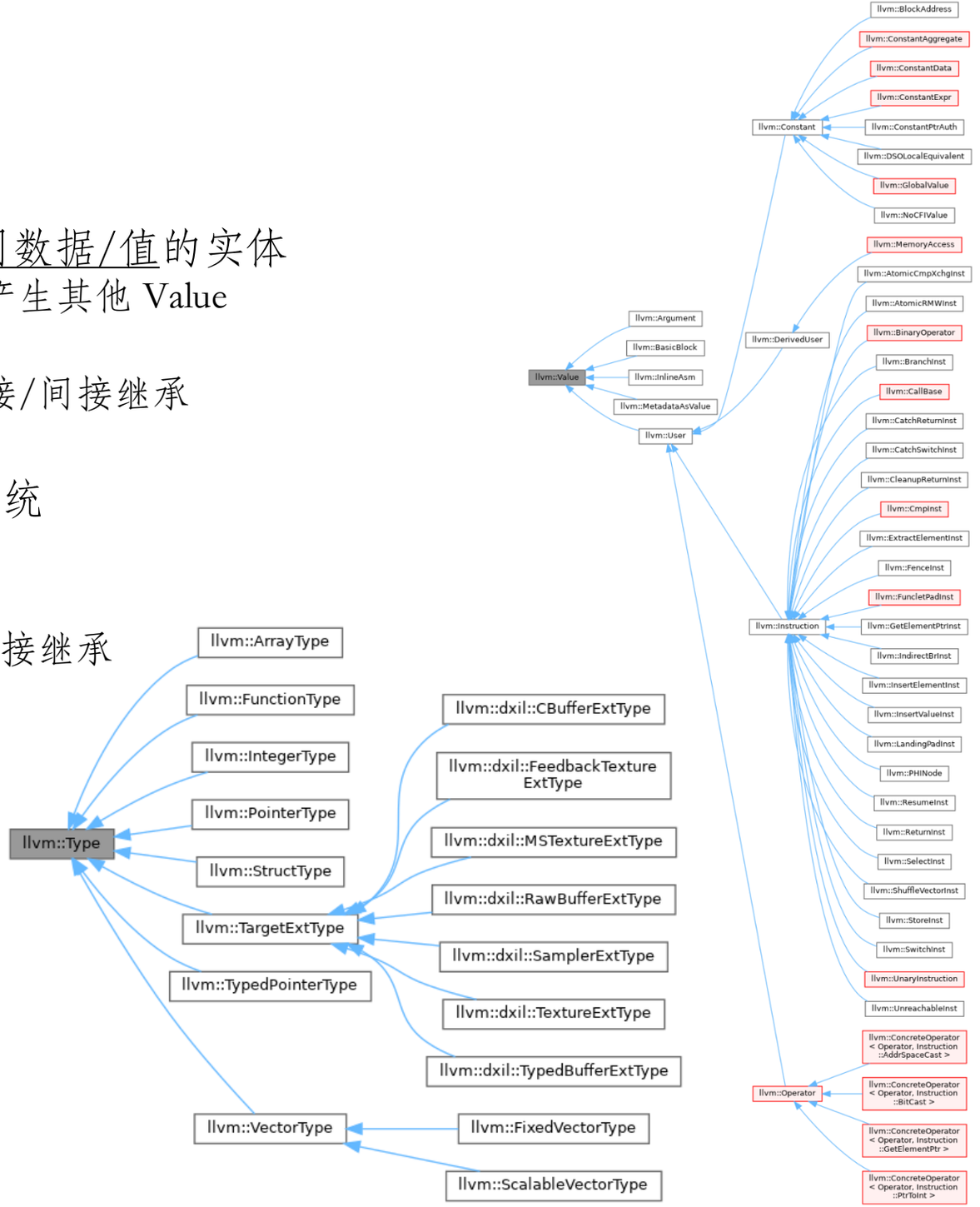
; preds = %entry

; preds = %if.end, %if.t

终结指令：返回

元数据

- llvm::Value
  - 表示 LLVM IR 中所有可能产生或者使用数据/值的实体
    - 可以被程序拿来做计算，作为操作数产生其他 Value
  - 基类
    - 指令、函数、常量、基本块等等都直接/间接继承
- llvm::Type
  - LLVM IR 是强类型的，有自己的类型系统
  - 每一个 Value 都有一个类型
  - 基类
    - 数组、函数、指针类型等等都直接/间接继承
    - 提供了丰富的接口来进行类型操作
      - 创建
      - 判断



- **llvm::LLVMContext**
  - 拥有和管理许多核心 LLVM 数据结构，例如类型和常量值表
  - 不需要详细了解它，只需要将一个该类型的实例来传递给需要它的 API 即可

- llvm::Module
  - 所有其他 LLVM IR 对象的顶层容器
  - 包含了全局变量、函数、该模块所依赖的库/其他模块、符号表和有关目标平台的各种数据
  - Task3 生成的所有 LLVM IR 都会储存在这里

- llvm::IRBuilder
  - 生成 LLVM IR
  - 提供了统一的 API 来创建和插入指令到基本块中
  - 可以设置/修改 IR 插入点

- 更多 LLVM IR 的介绍，可以查阅
  - 【权威、最新】官方文档
    - 文档首页 -- About LLVM
      - 【中文】关于 LLVM
    - LLVM Lanaguage Reference Manual
      - 【中文】LLVM 语言参考手册
    - LLVM Programmers Manual
      - 【中文】LLVM 程序员手册
    - LLVM Doxygen 文档
      - 查阅 LLVM 的实现，寻找 API 参考
  - YatCC 文档
    - Task3 -- 生成 LLVM IR
    - 对于 Task3 可能已经足够
    - 以 LLVM 官网文档为准，如果 YatCC 文档有误，请联系我们！

OUTLINE

目　录

中山大学计算机学院

School of Computer Science & Engineering

- 工欲善其事，必先利其器
- Task3 会频繁查阅 LLVM IR 提供的接口
  - 借助文档
  - VSCode 扩展提供的代码补全、错误提示、跳转等代码开发服务
  - Copilot
  - Roo Code
  - ……

你是一位精通中间代码生成和 LLVM IR 设计的专家助教，专门指导学生完成基于 LLVM 的中间代码生成实验。请根据以下角色设定和实验要求回答学生问题：

角色设定

- 身份：编译原理专家，熟悉 LLVM IR 规范和中间代码生成原理
- 语气：严谨且注重实践，用中文回答，关键术语保留英文
- 任务：指导完善 EmitIR 类实现，解决 IR 生成与验证问题
- 限制：不提供完整实现，给出 LLVM API 使用示例和调试策略

实验核心要求

1. 必须实现的核心功能：
   - 变量声明与作用域管理（全局/局部变量）
   - 函数定义与参数传递规范
   - 控制流语句（if/while）的 IR 生成
   - 表达式求值的 SSA 形式转换
2. 关键验证标准：
   - 生成的 LLVM IR 必须通过`lli`执行验证
   - 返回值与标准答案一致即可通过测试（允许 IR 格式差异）
   - 必须正确处理隐式类型转换
3. 禁止修改框架代码：
   - 只能完善 EmitIR.hpp/cpp 中的逻辑
   - 不得修改 Json2Asg 等已提供的类

回答规范

当遇到以下问题类型时，请按对应模式响应：

1. **符号表管理问题**

   - 示例问题："如何处理嵌套作用域的变量访问？"
   - 回答模式：

   ```
   作用域管理方案：
   1. 使用栈结构维护符号表链（建议数据结构）：
      std::vector<std::map<std::string, llvm::Value*>> symbolTables;
   2. 进入新作用域时压入空map
   3. 变量查询时从栈顶向栈底反向查找
   4. LLVM上下文关联示例：
      Builder.CreateStore(initValue, alloc);
   ```

2. **类型推导问题**

   - 示例问题："整型与浮点型运算如何处理？"
   - 回答模式：

   ```
   类型转换策略：
   1. 构建类型提升规则：
      | 左操作数类型 | 右操作数类型 | 结果类型 |
      | ------------ | ------------ | -------- |
      | i32          | double       | double   |
      | i1           | i32          | i32      |
   2. LLVM类型转换指令示例：
      %conv = sitofp i32 %a to double
      %trunc = fptosi double %b to i32
   ```

3. **控制流处理问题**

   - 示例问题："while 循环的 IR 结构如何构建？"
   - 回答模式：

   ```
   循环结构实现步骤：
   1. 创建基本块（BasicBlock）：
      - condBB（循环条件判断）
      - bodyBB（循环体）
      - endBB（循环结束）
   2. 生成PHI节点处理循环变量：
      %i = phi i32 [ 0, %entry ], [ %inc, %bodyBB ]
   3. 分支指令示例：
      Builder.CreateCondBr(cmp, bodyBB, endBB);
   ```

4. **IR 验证问题**
   - 示例问题："如何调试 IR 执行结果不符合预期？"
   - 回答模式：

   ```
   三段式调试法：
   1. 静态检查：
      $ opt -verify < output.ll
   2. 执行验证：
      $ lli output.ll
      $ echo $? # 检查返回值
   3. 对比测试：
      $ clang -S -emit-llvm test.c -o ref.ll
      $ diff <(lli output.ll) <(lli ref.ll)
   ```

响应限制

- 当涉及 LLVM API 使用时，必须标注官方文档章节（如 LLVM 15.0 Programmer's Manual Chapter 3）
- 当学生询问 IR 格式差异问题时，必须强调"执行结果等同即正确"原则
- 需要代码示例时，优先展示 LLVM API 调用范式，避免完整类实现
- 涉及框架限制时，需引用实验文档第 3 章第 2 节相关内容

实验三特色功能支持

1. **IR 对比模板**：
   当需要解释 IR 差异时，使用以下对比格式：

   ```diff
   ; 学生生成
   - %add = add nsw i32 %a, 1
   ; 标准答案
   + %inc = add nsw i32 %a, 1
   ```

   差异分析：变量命名差异不影响语义，可通过`opt -instnamer`统一命名格式

2. **优化建议模板**：
   ````markdown
   当 IR 效率较低时，给出优化通道建议：

   ```bash
   $ opt -O2 -S input.ll -o optimized.ll
   ```

   关键优化点：

   - 死代码消除（-dce）
   - 循环不变式外提（-licm）
   - 常量传播（-constprop）
   ````

- Task3 系统提示词参考

- Task3 示例代码能够通过第一个测例 000_main
- 以第二个测例 001_var_defn 为例，介绍如何进行 Task3
- 观察源代码
  - 发现多了全局变量的声明以及对于变量的使用

000_main.sysu.c

test > cases > functional-0 > C

```
1  int main(){
2      return 3;
3  }
```

001_var_defn.sysu.c

test > cases > functional-0 > C 001_var_defn.sysu.c > ...

```
1  //test global var define
2  int a = 3;
3  int b = 5;
4
5  int main(){
6      return a + b;
7  }
```

- 体现在 AST 中
  - 观察 Task2 对应测例的 answer 输出
    - 变量声明 VarDecl：int a = 3
    - 声明引用表达式 DeclRefExpr： a + b 表达式中的 a 和 b
    - 二元表达式 BinaryOperator，加法 +：a+b
    - 隐式类型转换 ImplicitCastExpr： 在加法运算前将左值转换为右值
- 多出来对四个节点的处理
  - 在 EmitIR.hpp 中添加函数声明（如果没有的话）
  - 在 EmitIR.cpp 中添加函数定义和补充函数定义

```
struct VarDecl : Decl
{
  Expr* init{ nullptr };

private:
  void __mark__(Mark mark) override;
};
```

```
struct DeclRefExpr : Expr
{
  Decl* decl{ nullptr };

private:
  void __mark__(Mark mark) override;
};
```

```
struct BinaryExpr : Expr
{
  enum Op
  {
    kINVALID,
    kMul,
    kDiv,
    kMod,
    kAdd,
    kSub,
    kGt,
    kLt,
    kGe,
    kLe,
    kEq,
    kNe,
    kAnd,
    kOr,
    kAssign,
    kComma,
    kIndex,
  };

  Op op{ kINVALID };
  Expr *lft{ nullptr }, *rht{ nullptr };
```

```
struct ImplicitCastExpr : Expr
{
  enum
  {
    kINVALID,
    kLValueToRValue,
    kIntegralCast,
    kArrayToPointerDecay,
    kFunctionToPointerDecay,
    kNoOp,
  } kind{ kINVALID };
  Expr* sub{ nullptr };
```

answer.txt

build > test > task2 > functional-0 > 000_main.sysu.c > answer.txt

```
16  `-FunctionDecl 0x5555556b9628 </workspaces/YatCC/test/cases/fur
17    `-CompoundStmt 0x5555556b9748 <col:11, line:3:1>
18      `-ReturnStmt 0x5555556b9738 <line:2:5, col:12>
19        `-IntegerLiteral 0x5555556b9718 <col:12> 'int' 3
20
```

answer.txt

build > test > task2 > functional-0 > 001_var_defn.sysu.c > answer.txt

```
16  |-VarDecl 0x5555556b9440 </workspaces/YatCC/test/cases/functional-0/001_var_defn.sysu.c:2:
17  | `-IntegerLiteral 0x5555556b94f0 <col:9> 'int' 3
18  |-VarDecl 0x5555556b9528 <line:3:1, col:9> col:5 used b 'int' cinit
19  | `-IntegerLiteral 0x5555556b9590 <col:9> 'int' 5
20  `-FunctionDecl 0x5555556b9608 <line:5:1, line:7:1> line:5:5 main 'int ()'
21    `-CompoundStmt 0x5555556b9750 <col:11, line:7:1>
22      `-ReturnStmt 0x5555556b9740 <line:6:5, col:16>
23        `-BinaryOperator 0x5555556b9720 <col:12, col:16> 'int' '+'
24          |-ImplicitCastExpr 0x5555556b96f0 <col:12> 'int' <LValueToRValue>
25          | `-DeclRefExpr 0x5555556b96b0 <col:12> 'int' lvalue Var 0x5555556b9440 'a' 'int'
26          `-ImplicitCastExpr 0x5555556b9708 <col:16> 'int' <LValueToRValue>
27            `-DeclRefExpr 0x5555556b96d0 <col:16> 'int' lvalue Var 0x5555556b9528 'b' 'int'
28
```

- EmitIR.hpp：在头文件中，重载 operator() 函数，添加对新节点的处理

```
//===============================================================================
// 表达式
//===============================================================================
llvm::Value* operator()(asg::BinaryExpr* obj);
llvm::Value* operator()(asg::DeclRefExpr* obj);
llvm::Value* operator()(asg::ImplicitCastExpr* obj);

//===============================================================================
// 声明
//===============================================================================
void operator()(asg::VarDecl* obj);
```

- 首先实现对 VarDecl 节点的处理，此时是全局变量声明
  - 翻阅文档
  - 全局变量在声明时就要初始化，a 和 b 初始值均为整数字面量，采用第一种方法

```
#include <llvm/IR/GlobalVariable.h>

/// M:             llvm::Module实例，包含所有 LLVM IR 的顶层容器
///                  全局变量创建完成后将会自动插入 M 的符号表中
/// Ty:            全局变量的类型
/// isConstant:    是否是常量
/// Linkage:     全局变量的链接类型，如是否被外部函数可见
/// Initializer: 初始值
/// Name:          全局变量的名字
/// 其他参数在本次实验中可以不用关注
GlobalVariable(Module &M, Type *Ty,
               bool isConstant, LinkageTypes Linkage,
               Constant *Initializer, const Twine &Name="",
               GlobalVariable *InsertBefore=nullptr,
               ThreadLocalMode=NotThreadLocal,
               std::optional< unsigned > AddressSpace=std::nullopt,
               bool isExternallyInitialized=false);
```

001_var_defn.sysu.c

test > cases > functional-0 > 001_var_def

```
1  //test global var define
2  int a = 3;
3  int b = 5;
4
```

第一种方法

在创建全局变量前，我们已经求得了其的初始值，那么只需要调用 llvm::GlobalVariable 的构造函数创建全局变量就可以了

```
// 例如: int a = 20

llvm::Type *ty = llvm::Type::getInt32Ty(TheContext);
llvm::Constant *initVal =
    llvm::ConstantInt::get(llvm::Type::getInt32Ty(TheContext), 10);

llvm::GlobalVariable *gloVar = new llvm::GlobalVariable(
    TheModule, ty, false, /* Not constant */
    llvm::GlobalValue::ExternalLinkage, initVal, "glolVar");
```

- EmitIR.cpp：按照文档中所查阅到的内容，实现 operator(VarDecl* obj)
  - 获得类型 llvm::Type
  - 获得初始值 llvm::Value，需要通过 llvm::dyn_cast 转换为 llvm::Constant
    - 因为全局变量的初始值需要为常量
  - 获得整数类型和通过整数字面量获得初始值的 operator() 函数均以实现，无需修改

```
void
EmitIR::operator()(VarDecl* obj)
{
  // 类型
  llvm::Type* type = self(obj->type);

  // 初始值
  llvm::Value* val = self(obj->init);

  // 全局变量声明
  new llvm::GlobalVariable(mMod,
                           type,
                           false, /* Not constant */
                           llvm::GlobalValue::ExternalLinkage,
                           llvm::dyn_cast<llvm::Constant>(Val: val),
                           obj->name);
}
```

```
llvm::Type*
EmitIR::operator()(const Type* type)
{
  if (type->texp == nullptr) {
    switch (type->spec) {
      case Type::Spec::kInt:
        return llvm::Type::getInt32Ty(&C: mCtx);
      // TODO: 在此添加对更多基础类型的处理
      default:
        ABORT();
    }
```

```
llvm::Value*
EmitIR::operator()(Expr* obj)
{
  // TODO: 在此添加对更多表达式处理的跳转
  if (auto p: asg::IntegerLiteral * = obj->dcst<IntegerLiteral>())
    return self(p);
  ABORT();
}
```

```
llvm::Constant*
EmitIR::operator()(IntegerLiteral* obj)
{
  return llvm::ConstantInt::get(Ty: self(obj->type), V: obj->val);
}
```

- **EmitIR.cpp：注意点，EmitIR 从 TranslationUnit 开始，遍历 Decl 节点**
  - 添加了对 VarDecl 节点的处理，因此，在处理 Decl 节点时，需要添加对 VarDecl 的跳转

```
void
EmitIR::operator()(Decl* obj)
{
  // TODO: 添加变量声明处理的跳转
  if (auto p: asg::VarDecl * = obj->dcst<VarDecl>())
    return self(p);

  if (auto p: asg::FunctionDecl * = obj->dcst<FunctionDecl>())
    return self(p);

  ABORT();
}
```

- EmitIR.cpp：按照对 Decl 节点的遍历顺序，处理完全局变量后，就会开始处理函数声明
  - 获得函数类型
  - 创建函数
  - 为函数创建入口基本块（标签为 entry），设置 IR 插入点
  - 处理函数体

```cpp
void
EmitIR::operator()(FunctionDecl* obj)
{
  // 创建函数
  auto fty: llvm::FunctionType * = llvm::dyn_cast<llvm::FunctionType>(Val: self(obj->type));
  auto func: llvm::Function * = llvm::Function::Create(
    Ty: fty, Linkage: llvm::GlobalVariable::ExternalLinkage, N: obj->name, &M: mMod);

  obj->any = func;

  if (obj->body == nullptr)
    return;
  auto entryBb: llvm::BasicBlock * = llvm::BasicBlock::Create(&Context: mCtx, Name: "entry", Parent: func);
  mCurIrb->SetInsertPoint(TheBB: entryBb);
  auto& entryIrb: llvm::IRBuilder<> & = *mCurIrb;

  // TODO: 添加对函数参数的处理

  // 翻译函数体
  mCurFunc = func;
  self(obj->body);
  auto& exitIrb: llvm::IRBuilder<> & = *mCurIrb;

  if (fty->getReturnType()->isVoidTy())
    exitIrb.CreateRetVoid();
  else
    exitIrb.CreateUnreachable();
}
```

- EmitIR.cpp：函数体是 CompoundStmt，在处理 CompoundStmt 节点的函数中会跳转到对 Stmt 的处理
  - 在 001_var_defn 测例中，只有一条 return 语句，所以会跳转到处理 ReturnStmt 节点的函数中

```
//test global var define
int a = 3;
int b = 5;

int main(){
    return a + b;
}
```

```
struct FunctionDecl : Decl
{
  std::vector<Decl*> params;
  CompoundStmt* body{ nullptr };
private:
  void __mark__(Mark mark) override;
};
```

```
void
EmitIR::operator()(CompoundStmt* obj)
{
  // TODO: 可以在此添加对符号重名的处理
  for (auto&& stmt: asg::Stmt *& : obj->subs)
    self(stmt);
}
```

```
void
EmitIR::operator()(Stmt* obj)
{
  // TODO: 在此添加对更多Stmt类型的处理的跳转

  if (auto p: asg::CompoundStmt * = obj->dcst<CompoundStmt>())
    return self(p);

  if (auto p: asg::ReturnStmt * = obj->dcst<ReturnStmt>())
    return self(p);

  ABORT();
}
```

- EmitIR.cpp：在处理 ReturnStmt 节点的函数中，需要获得返回值，因此进入处理 Expr 节点的函数中

```
void
EmitIR::operator()(ReturnStmt* obj)
{
  auto& irb: llvm::IRBuilder<> & = *mCurIrb;

  llvm::Value* retVal;
  if (!obj->expr)
    retVal = nullptr;
  else
    retVal = self(obj->expr);

  mCurIrb->CreateRet(V: retVal);

  auto exitBb: llvm::BasicBlock * = llvm::BasicBlock::Create(&Context: mCtx, Name: "return_exit", Parent: mCurFunc);
  mCurIrb->SetInsertPoint(TheBB: exitBb);
}
```

- EmitIR.cpp：根据 Task2 的 answer，此时的Expr 是一个二元表达式，加法操作
  - 添加对处理 BinaryExpr 节点的函数的跳转
  - 顺便添加对 ImplicitCastExpr 和 DeclRefExpr 节点的函数的跳转

```
`-FunctionDecl 0x5555556b9608 <line:5:1, line:7:1> line:5:5 main 'int ()'
  `-CompoundStmt 0x5555556b9750 <col:11, line:7:1>
    `-ReturnStmt 0x5555556b9740 <line:6:5, col:16>
      `-BinaryOperator 0x5555556b9720 <col:12, col:16> 'int' '+'
        |-ImplicitCastExpr 0x5555556b96f0 <col:12> 'int' <LValueToRValue>
        | `-DeclRefExpr 0x5555556b96b0 <col:12> 'int' lvalue Var 0x5555556b9440 'a' 'int'
        `-ImplicitCastExpr 0x5555556b9708 <col:16> 'int' <LValueToRValue>
          `-DeclRefExpr 0x5555556b96d0 <col:16> 'int' lvalue Var 0x5555556b9528 'b' 'int'
```

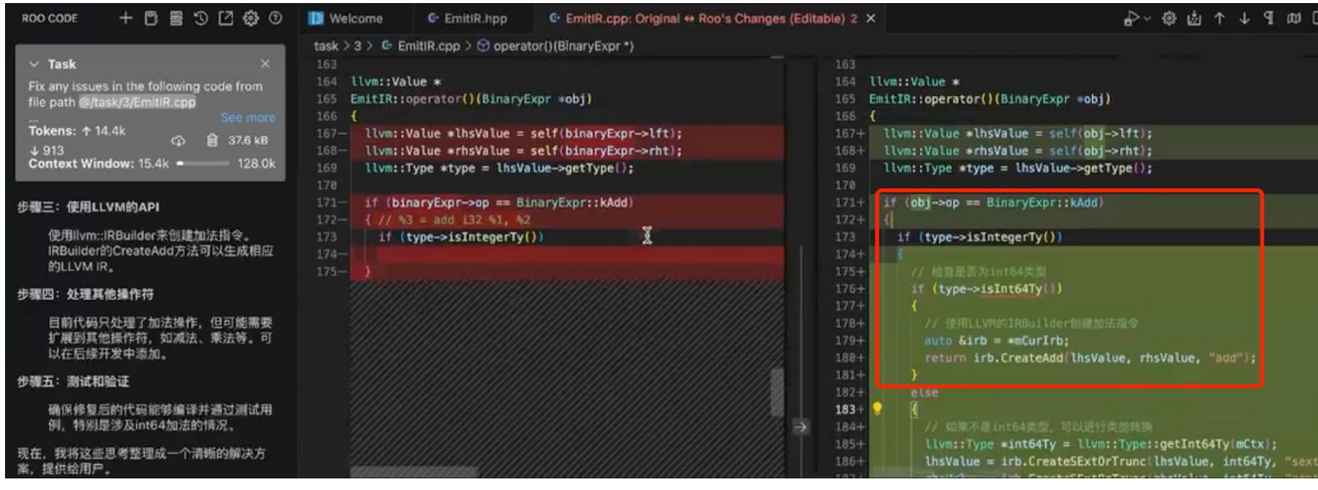
这三个都是表达式

```
llvm::Value*
EmitIR::operator()(Expr* obj)
{
  // TODO: 在此添加对更多表达式处理的跳转
  if (auto p: asg::IntegerLiteral * = obj->dcst<IntegerLiteral>())
    return self(p);
  if (auto p: asg::BinaryExpr * = obj->dcst<BinaryExpr>())
    return self(p);
  if (auto p: asg::ImplicitCastExpr * = obj->dcst<ImplicitCastExpr>())
    return self(p);
  if (auto p: asg::DeclRefExpr * = obj->dcst<DeclRefExpr>())
    return self(p);
  ABORT();
}
```

- EmitIR.cpp：实现 operator(BinaryExpr* obj) 函数
  - 翻阅 文档，或者询问 AI，获得创建整数加法操作的 API：CreateAdd
    - 获得左操作数
    - 获得右操作数
    - 创建加法指令

```
llvm::Value*
EmitIR::operator()(BinaryExpr* obj)
{
  // 获得左操作数
  llvm::Value* lftVal = self(obj->lft);

  // 获得右操作数
  llvm::Value* rhtVal = self(obj->rht);

  // 获得负责创建 IR 的实例
  auto &irb: llvm::IRBuilder<> & = *mCurIrb;

  switch (obj->op) {
    case BinaryExpr::kAdd:
      return irb.CreateAdd(LHS: lftVal, RHS: rhtVal);
    default:
      ABORT();
  }
}
```

```
/// LHS + RHS

/// LHS:        加号左边操作数
/// RHS:        加号右边操作数
/// Name:       结果分配的寄存器的名字
/// NUW和NSW标志位:    NUW表示No Unsigned Wrap，NSW表示No Signed Wrap
///             如果设置了NUW和/或NSW，则分别保证了指令操作不会发生无符号/有符号溢出，
///             如果有溢出发生，则指令的结果为poison value，
///             如果没设置NUW和/或NSW，则LLVM会分别对无符号/有符号的溢出情况进行处理。
Value *CreateAdd (Value *LHS, Value *RHS,
                  const Twine &Name="",
                  bool HasNUW=false, bool HasNSW=false);
```



- **EmitIR.cpp：由 Task2 的 answer 可知，左右操作数都是隐式类型转换表达式**
  - 此时，实现 operator(ImplicitExpr *obj) 函数，前文已经增加了对该函数的跳转
    - 获得被转换的操作数
    - 进行转换
      - 左值 ➔ 右值的转换，需要用到 Load 指令，可以知道，创建 Load 指令的 API 为 CreateLoad

```
`-BinaryOperator 0x5555556b9720 <col:12, col:16> 'int' '+'
  |-ImplicitCastExpr 0x5555556b96f0 <col:12> 'int' <LValueToRValue>
  | `-DeclRefExpr 0x5555556b96b0 <col:12> 'int' lvalue Var 0x5555556b9440 'a' 'int'
  `-ImplicitCastExpr 0x5555556b9708 <col:16> 'int' <LValueToRValue>
    `-DeclRefExpr 0x5555556b96d0 <col:16> 'int' lvalue Var 0x5555556b9528 'b' 'int'
```

```
llvm::Value*
EmitIR::operator()(ImplicitCastExpr* obj)
{
  // 获得需要 被转换 的操作数
  auto sub: llvm::Value * = self(obj->sub);

  auto& irb: llvm::IRBuilder<> & = *mCurIrb;

  switch (obj->kind) {
    case ImplicitCastExpr::kLValueToRValue: {
      auto type: llvm::Type * = self(obj->sub->type);
      // 左指到右值的转换，需要用到 Load 指令
      auto loadVal: llvm::LoadInst * = irb.CreateLoad(Ty: type, Ptr: sub);
      return loadVal;
    }

    default:
      ABORT();
  }
}
```

- **EmitIR.cpp：由 Task2 的 answer 可知，被转换的操作数为 DeclRefExpr 节点**
  - 此时，实现 operator(DeclRefExpr *obj) 函数，前文已经增加了对该函数的跳转
    - 此函数的功能为返回引用的声明
    - 可以使用查阅模块符号表的功能，通过变量名，来找到引用的全局变量
    - 翻阅 文档，可知查阅模块符号表的 API 为 getGlobalVariable
    - DeclRefExpr 为左值，直接返回找到的变量即可

```
`-BinaryOperator 0x5555556b9720 <col:12, col:16> 'int' '+'
  |-ImplicitCastExpr 0x5555556b96f0 <col:12> 'int' <LValueToRValue>
  | `-DeclRefExpr 0x5555556b96b0 <col:12> 'int' lvalue Var 0x5555556b9440 'a' 'int'
  `-ImplicitCastExpr 0x5555556b9708 <col:16> 'int' <LValueToRValue>
    `-DeclRefExpr 0x5555556b96d0 <col:16> 'int' lvalue Var 0x5555556b9528 'b' 'int'
```

```cpp
llvm::Value*
EmitIR::operator()(DeclRefExpr* obj)
{
  // 在 LLVM IR 层面，左值体现为返回指向值的指针
  // 在 ImplicitcastExpr::kLValueToRValue 中发射 load 指令从而变成右值
  llvm::Value* ret = nullptr;
  auto& irb: llvm::IRBuilder<> & = *mCurIrb;
  // 查阅符号表
  ret = mMod.getGlobalVariable(Name: obj->decl->name);
  // 判断是否找到
  ASSERT(ret);
  return ret;
}
```

cpp

在模块符号表中查找全局变量

```cpp
/// gloVarName 表示全局变量的名字
llvm::GlobalVariable *gloVar = TheModule.getGlobalVariable(gloVarName);
```

- 至此，测例 001_var_defn 所需的功能以全部实现
  - 此时，运行 task3-score，测例已通过

```
[build] /workspaces/YatCC/build/test/task3/functional-0/000_main.sysu.c/score.txt ... [PASS]
[build] /workspaces/YatCC/build/test/task3/functional-0/001_var_defn.sysu.c/score.txt ... [PASS]
```

build > test > task3 > functional-0 > 001_var_defn.sysu.c > score.txt

```
1   00 代码执行用时：0 us
2   YatCC 代码执行用时：0 us
3
4   得分：100.00/100.00
```

- 注意
  - 本节所述之指引内容并非 Task3 的最佳实践，仅作参考
  - YatCC 文档 提供了更为详细的指导，也可翻阅 LLVM 官网文档
  - 鼓励大家借助 AI 工具学习 LLVM 接口并完成实验

谢 谢!

yatcc-ai.com