7 线性 IR

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 熟悉 LLVM IR
- ⋆⋆⋆ 能够将 Tea 语言代码翻译为 LLVM IR
- ⋆ 了解解释执行

7.1 线性 IR

本章将系统介绍一种线性中间表示（Intermediate Representation, IR）的定义方式及其基本使用方法。这里所采用的 IR 是 LLVM IR [1] 的一个简化子集，专门为 Tea 语言的编译过程设计。选择 LLVM IR 作为中间表示具有多方面优势。首先，LLVM 生态成熟，拥有完善的工具链支持，使得 IR 不仅是中间产物，同时也是一个可观察、可执行、可优化的程序表示形式。例如，可以通过现有的 C 语言编译器 `clang` 将高级语言程序转换为 LLVM IR，从而直观理解语言结构与底层表示之间的映射关系；同时，还可以借助解释器 `lli` 直接执行 IR 代码，方便调试和验证编译结果。这种可执行的中间表示大大降低了编译器开发与教学的门槛。

```
; 生成中间代码hello.ll
clang -emit-llvm -S hello.c
; 执行中间代码
lli hello.ll
```

代码 7.1: 生成与执行 LLVM IR 的基本命令

代码 7.2 给出了一个简单的 LLVM IR 示例。该示例包含一个全局变量声明 `@g`，以及两个函数定义 `@foo` 和 `@main`。通过这个例子，可以初步观察 LLVM IR 的基本结构：包括全局区、函数定义、指令序列以及显式的内存操作。

```
@g = global i32 10                ; 声明全局变量@g：类型为i32，初始值为10
define i32 @foo(i32 %0) {         ; 定义函数foo：类型为i32->i32，参数为%0
    %x = alloca i32               ; 在栈上分配i32空间，返回指针%x
    store i32 %0, ptr %x          ; 将%0写入%x指向的内存
    %g0 = load i32, ptr @g        ; 加载全局变量@g的内容到%g0
    ret i32 %g0                   ; 返回%g0
}
define i32 @main() {              ; 定义函数foo：类型为 void->i32
    %r0 = call i32 @foo(i32 1)    ; 调用函数foo，返回值保存为%r0
    ret i32 %r0                   ; 返回%r0
}
```

代码 7.2: LLVM IR 代码示例

接下来，我们将围绕 Tea 语言所使用的 IR 子集，对相关指令和核心概念进行逐一讲解。

7.1.1 类型

LLVM IR 是一种强类型语言，每一条指令都需要显式标注数据类型。Tea 语言中涉及的主要类型包括以下几类：

- **标量类型**：表示基本数值数据，主要是不同位宽的有符号整数，如`i32`（32 位整数）、`i8`（8 位整数）和`i1`（1 位布尔值）。
- **指针类型**：用于表示内存地址。在 LLVM 17 之后，引入了统一的`ptr`类型，不再需要区分具体指向的数据类型（例如`i32*`）。
- **数组类型**：用于表示定长序列，例如`[2 x i32]`表示包含两个`i32`元素的数组。
- **自定义类型**：可以通过`type`关键字定义结构体类型，例如`%mytype = type {i32, i32}`。

7.1.2 标识符

在 LLVM IR 中，变量名、函数名以及基本块名称统称为标识符，并分为两类：

- **局部标识符**：以`%`开头，仅在当前函数内部有效，例如`%r1`或`%0`。局部标识符既可以表示临时变量，也可以表示基本块标签（如`%bb1`）。

  需要特别注意的是，当局部标识符采用纯数字形式（如`%0`、`%1`等）时，必须满足以下约束：

  - 编号必须从`%0`开始；
  - 编号必须连续递增，不能跳号或重复；
  - 所有局部标识符（包括变量和基本块标签）共享同一编号空间，即它们的编号是统一递增的，而不是分别独立编号。

  如果违反上述规则，解释器 `lli` 可能无法正确解析或执行 IR 代码。

- **全局标识符**：以`@`开头，在整个程序范围内可见，例如`@g`或`@main`。全局标识符用于表示全局变量或函数名称，其命名不受局部编号规则的限制。

LLVM IR 采用一种重要的设计原则：静态单赋值形式（SSA, Static Single Assignment）。在这种形式下，每个变量只能被赋值一次，即每个标识符只会出现在一条定义语句的左侧。这种约束使得数据流关系更加清晰，也为后续优化提供了便利。关于 SSA 的详细内容将在后续章节展开。

7.1.3 内存分配和数据存取

本节主要讨论函数栈帧上的内存分配与访问机制。在 LLVM IR 中，内存操作是显式的：与高级语言中“变量即存储”的抽象不同，IR 不会隐式为变量分配空间，而是需要通过指令明确地完成“分配—写入—读取”的全过程。

基本机制 局部变量通常分配在函数栈帧中，其内存空间通过`alloca`指令申请。该指令返回一个指向所分配内存的指针。对内存的访问则通过`store`和`load`指令完成：前者用于写入数据，后者用于读取数据。需要注意的是`alloca`按字节分配内存，因此通常不会用于`i1`这样的位级类型。

基本示例 代码 7.3 展示了典型的“分配—写入—读取”流程：

```
; <result> = alloca <type> [, <num_elements>] [, align <alignment>]
; 在栈上分配内存，返回指向该内存的指针
%x = alloca i32               ; 分配一个 i32 空间，返回指针 %x

; store <type> <value>, ptr <pointer> [, align <alignment>]
; 将 value 写入 pointer 指向的内存
store i32 1, ptr %x           ; 向 %x 指向的地址写入整数 1

; <result> = load <type>, ptr <pointer> [, align <alignment>]
; 从 pointer 指向的内存读取数据
%t1 = load i32, ptr %x        ; 从 %x 读取 i32，结果存入 %t1
```

代码 7.3: LLVM IR 代码示例：内存分配和数据存取

变量与临时值 在 Tea 语言的源程序中，每个变量在 IR 中通常对应一块内存单元，需要通过`load/store`访问。而在 LLVM IR 中，还会引入大量临时变量（如`%t1`、`%t2`），这些变量不对应具体内存位置，而更类似于“虚拟寄存器”，用于保存中间计算结果。

类型转换 LLVM IR 提供了显式的类型转换指令，用于在不同位宽之间转换数据。例如：

- `zext`（zero extension）：将小类型扩展为大类型，高位补零；
- `trunc`（truncate）：将大类型截断为小类型，仅保留低位。

```
; <result> = zext <from_type> <value> to <to_type>
; 零扩展（zero extend）：将较小整数类型扩展为较大整数类型，高位补 0
%t2 = zext i1 %t1 to i32          ; 将 i1 扩展为 i32，高位补 0，结果存入 %t2

; <result> = trunc <from_type> <value> to <to_type>
; 截断（truncate）：将较大整数类型转换为较小整数类型，保留低位，丢弃高位
%t3 = trunc i32 %t2 to i8         ; 将 i32 截断为 i8，仅保留低 8 位
```

代码 7.4: LLVM IR 代码示例：类型转换

复合类型的访问 对于数组和结构体等复合类型，内存访问不仅仅是“读/写”，还需要先进行地址计算。在 LLVM IR 中，这一过程通过`getelementptr`（GEP）指令完成，然后再结合`load/store`进行数据访问。可以将`getelementptr`理解为一种“基于类型信息的地址偏移计算”：它不会访问内存，而只是根据给定的类型结构和索引，计算出目标元素的地址。

代码 7.5展示了数组元素访问的两种典型情况。一种是基于元素指针的线性访问，另一种是基于数组类型的分层访问。

```
; <result> = getelementptr <ty>, ptr <ptrval>{, <idx_ty> <idx>}*
; ===== 情况1：线性内存（指向单一元素类型）=====
; 在%p的基础上按i32大小偏移1个元素，得到下一个元素地址
%t1 = getelementptr i32, ptr %p, i32 1
; 读取该地址中的值
%t2 = load i32, ptr %t1

; ===== 情况2：数组对象访问 =====
; 在栈上分配一个长度为10的i32数组
%a = alloca [10 x i32]
; 第一个0：从数组对象本身开始（不跳过数组）
; 第二个1：访问数组下标为1的元素（即第2个元素）
%t3 = getelementptr [10 x i32], ptr %a, i32 0, i32 1
; 将99写入该元素位置
store i32 99, ptr %t3
```

代码 7.5: LLVM IR 代码示例：数组元素存取

当 GEP 操作的是“指向元素类型的指针”（如`ptr %p`指向`i32`）时，只需要一个索引，表示按元素大小进行线性偏移。当操作的是“数组类型指针”（如`ptr %a`指向`[10 x i32]`）时，需要两个索引：第一个索引通常为`0`，表示从当前数组对象本身开始；第二个索引表示数组中具体元素的下标。因此，`getelementptr [10 x i32], ptr %a, i32 0, i32 1`的含义是：在数组`%a`中，取第 1 号元素（即第二个元素）的地址。

结构体的访问方式与数组类似，同样通过 GEP 进行逐层定位。代码 7.6给出了示例。其中，第一个索引`0`表示从结构体对象本身开始；第二个`0`表示访问结构体的第 0 个字段。

```
; 定义结构体类型：包含两个i32字段
%mystruct = type { i32, i32 }
; 在栈上分配一个结构体对象，返回指针%st
%st = alloca %mystruct
; 第一个0：从结构体对象本身开始
; 第二个0：访问第0个字段（第一个成员）
%t1 = getelementptr %mystruct, ptr %st, i32 0, i32 0
; 将整数1写入该字段
store i32 1, ptr %t1
```

代码 7.6: LLVM IR 代码示例：结构体域数据存取

可以看出，数组和结构体在 GEP 中的访问方式本质是一致的：都是通过“逐层索引”的方式，在类型结构中逐级定位目标元素。GEP 支持多层级的嵌套索引，可用于访问多维数组或嵌套结构体等复杂数据结构。更详细的说明可参考官方文档[1]。

[1] https://llvm.org/docs/GetElementPtr.html

7.1.4 算数运算

Tea 语言使用的 LLVM IR 中的算数运算指令均为有符号数运算，包括add、sub、mul 和 sdiv，不涉及无符号数运算。为了简化讨论，本教材暂不考虑整数运算溢出的情况。

```
; <res> = add <res type> <oprand 1>, <oprand 2>
; oprand1和oprand2也必须和<res type>一致
%t3 = add i32 %t1, %t2      ; 加法运算: %t3 = %t1 + %t2
%t4 = sub i32 %t1, %t2      ; 减法运算: %t4 = %t1 - %t2
%t5 = mul i32 %t1, %t2      ; 乘法运算: %t5 = %t1 * %t2
%t6 = sdiv i32 %t1, %t2     ; 有符号的除法运算: : %t6 = %t1 / %t2
```

代码 7.7: LLVM IR 代码示例：算数运算

7.1.5 关系运算

IR 中支持的关系运算指令是icmp，可通过参数设置区分不同的比较模式。

```
; <res> = icmp <mod> <operand type> <oprand1>, <oprand2>
; <mod>是比较模式，包括: eq, neq, sgt, sge, slt, sle
%t3 = icmp eq i32 %t1, %t2      ; 等于
%t4 = icmp neq i32 %t1, %t2     ; 不等于
%t5 = icmp sgt i32 %t1, %t2     ; 大于
%t6 = icmp sge i32 %t1, %t2     ; 大于等于
%t7 = icmp slt i32 %t1, %t2     ; 小于
%t8 = icmp sle i32 %t1, %t2     ; 小于等于
```

代码 7.8: LLVM IR 代码示例：比较运算

7.1.6 控制流

控制流描述程序执行过程中基本块（basic block）之间的跳转关系。在 LLVM IR 中，基本块以标识符加冒号定义，例如“bb1:”。每个基本块必须以一条“终结指令”（terminator）结束，例如 br 或 ret，用于显式指定控制流的去向。

LLVM IR 中最常见的跳转指令是 br，分为无条件跳转和条件跳转两种形式：

```
bb0: ; 定义代码块 bb0
    ; br label <dst block> ; 无条件跳转
    br label %bb1

bb1: ; 定义代码块 bb1
    %t3 = icmp eq i32 %t1, %t2
    ; br i1 <cond>, label <true block>, label <false block> ; 条件跳转
    br i1 %t3, label %bb0, label %bb2

bb2: ; 定义代码块 bb2
    ...
```

代码 7.9: LLVM IR 代码示例：控制流

此外，LLVM IR 中还有一条与控制流密切相关的指令 phi，用于在不同控制流路径汇合时选择变量的值。该指令将在下一章介绍静态单赋值（SSA）形式时详细讲解，这里仅给出基本形式：

```
; <res> = phi <type> [<value 1>, <label 1>], [<value 2>, <label 2>], ...
; 若来自 <label 1>, 则取 <value 1>; 若来自 <label 2>, 则取 <value 2>
%t3 = phi i32 [%t1, %bb1], [%t2, %bb2]
```

代码 7.10: LLVM IR 代码示例：phi 指令

7.1.7 逻辑运算

LLVM IR 中没有专门的逻辑运算指令。逻辑运算可以通过位运算指令xor、and和or来实现。

```
; 实现逻辑非运算：%b = !%a
; <res> = xor <type> <operand 1> <operand 2>
%b = xor i1 %a, true ;
; 实现逻辑与运算：%r = %b && %a
; <res> = and <type> <operand 1> <operand 2>
%r = and i1 %a, %b
; 实现逻辑或运算：%r = %b || %a
; <res> = or <type> <operand 1> <operand 2>
%r = or i1 %a, %b
```

代码 7.11: LLVM IR 代码示例：通过位运算实现逻辑运算

此外，逻辑“与”和“或”运算通常通过控制流指令以短路方式实现等效功能。

```
bb1:
    br i1 %a, label %bb2, label %bb3
bb2:
    br label %bb3
bb3:
    %r = phi i1 [false, %bb1], [%b, %bb2]
```

代码 7.12: LLVM IR 代码示例：通过控制流指令实现%a && %b

```
bb1:
    br i1 %a, label %bb3, label %bb2
bb2:
    br label %bb3
bb3:
    %r = phi i1 [true, %bb1], [%b, %bb2]
```

代码 7.13: LLVM IR 代码示例：通过控制流指令实现%a || %b

7.1.8 函数

在 LLVM IR 中，定义函数使用`define`语句；如果仅声明该函数，则使用`declare`语句。在同一个 LLVM IR 文件中，不允许对同一个函数同时进行声明和定义。如果需要在一个 IR 文件中调用另一个 IR 文件中定义的函数，应先在当前 IR 文件中进行声明，并使用 llvm-link 工具进行链接。函数调用相关的指令主要包括调用指令`call`和返回指令`ret`。

```
; define <return type> <function ID> (<arg1 type> <arg1>, <arg2 type> <arg2>) {...}
define i32 @foo(i32 %0) {    ; 定义函数foo：类型是i32->i32
    ret i32 %0
}
; declare <return type> <function ID> (<arg1 type> <arg1>, <arg2 type> <arg2>)
declare void @bar(i32 %0)    ; 声明函数bar：类型是i32->void
define i32 @main() {         ; 定义函数main：类型是void->i32
    ; <return value> = call <return type> <function ID>(<arg type> <arg value>)
    %r0 = call i32 @foo(i32 1)
    ; ret <return type> <return value>
    ret i32 %r0
}
```

代码 7.14: LLVM IR 代码示例：函数声明、定义和调用

7.2 AST 翻译线性 IR

将 AST 翻译为 IR 代码的整体思路如下：

1) 遍历顶层 AST，生成函数及全局变量的 IR 表示；

2) 对每个函数的 AST 进行递归下降遍历，构建基本块（代码块）及其跳转关系；

3) 遍历各个基本块，逐条翻译其中的语句为 IR 指令。

该翻译过程的主要难点体现在两个方面：其一是基本块的划分及其控制流关系的构建，其二是指令参数的定义-使用（def-use）关系的处理。

7.2.1 基本块的构建及跳转关系

在 LLVM IR 中，每个基本块必须以终结指令（如`br`或`ret`）结束。因此，在递归下降遍历 AST 的过程中，需要在控制流发生分叉或回跳的位置显式划分基本块，并建立清晰的跳转关系。主要包括以下几种结构：

- **函数定义**：创建入口基本块`%bb0`，并预先插入返回指令 `ret <type> %tobeDetermined` 作为占位。
- **if 语句**：创建三个基本块：`%bb-true`、`%bb-false` 以及后继块`%bb-after`。在当前基本块中插入条件分支指令：`br i1 %tobeDetermined, label %bb-true, label %bb-false`。同时，将当前基本块中原有的终结指令移动至`%bb-after`，作为其终结指令。在`%bb-true` 和`%bb-false` 中分别添加无条件跳转至`%bb-after` 的指令。
- **while 语句**：创建三个基本块：`%bb-cond`（条件判断块）、`%bb-body`（循环体）以及后继块`%bb-after`。在当前基本块中插入跳转至`%bb-cond` 的指令，并将原有终结指令移动至`%bb-after`。在`%bb-cond` 中生成条件分支指令：`br i1 %tobeDetermined, label %bb-body, label %bb-after`；在`%bb-body` 中添加回跳至`%bb-cond` 的指令以形成循环。

在 `while` 结构中，还需要额外考虑 `break` 和 `continue` 语句对控制流的影响。`continue` 语句应直接跳转至`%bb-cond`，以开始下一轮循环；`break` 语句则跳转至后继块`%bb-after`，从而退出循环。因此，在构建循环体内部基本块时，需要根据语句类型动态确定其后继基本块。

需要注意的是，`break` 和 `continue` 的影响范围取决于其所在位置：当它们出现在嵌套的内层 `while` 中时，仅影响该内层循环的控制流，对外层结构没有影响；而当它们出现在当前 `while` 内部的 `if` 分支中时，会改变该 `if` 语句后续基本块的控制流走向（例如提前跳转至`%bb-cond` 或`%bb-after`），从而影响后续基本块的构建。因此，在实现时需要结合语法结构，对不同层级的跳转目标进行区分和维护。

伪代码 1展示了函数内部的代码块生成逻辑。该设计能够正确表达 Tea 语言中的控制流结构，包括 `if-else` 与 `while` 的嵌套情况。在实现时，基本块编号建议采用`%bb` 后接递增数字的形式；不推荐使用纯数字编号，以避免编号不连续导致的问题，从而影响 `lli` 的正确执行。

算法 1 基于 AST 的基本块构建与 IR 生成

input: AST statement $stmt$
output: IR in basic blocks
procedure GenIR($stmt, curBB, breakTarget, continueTarget$) ▷ 在当前基本块 $curBB$ 中为语句 $stmt$ 生成 IR
  match $stmt$:
    case If($cond, thenStmt, elseStmt$) ⇒
      $trueBB \leftarrow$ newBB() ▷ then 分支
      $falseBB \leftarrow$ newBB() ▷ else 分支
      $afterBB \leftarrow$ newBB() ▷ 汇合块
      $v \leftarrow$ GenExpr($cond$, $curBB$) ▷ 计算条件表达式，得到 i1 值
      emit($curBB$, br(v, $trueBB$, $falseBB$)) ▷ 根据条件跳转
      GenIR($thenStmt, trueBB, breakTarget, continueTarget$) ▷ 生成 then 分支
      emit_if_no_term($trueBB$, br($afterBB$)) ▷ 若未终结，则跳转到汇合块
      if $elseStmt \neq$ null **then**
        GenIR($elseStmt$, $falseBB, breakTarget, continueTarget$) ▷ 生成 else 分支
        emit_if_no_term($falseBB$, br($afterBB$)) ▷ 保证 else 分支正确终结
      else
        emit($falseBB$, br($afterBB$)) ▷ 无 else 时直接跳转
      end if
    case While($cond, body$) ⇒
      $condBB \leftarrow$ newBB() ▷ 条件判断块
      $bodyBB \leftarrow$ newBB() ▷ 循环体
      $afterBB \leftarrow$ newBB() ▷ 循环结束后
      emit($curBB$, br($condBB$)) ▷ 进入循环前跳转到条件判断
      $v \leftarrow$ GenExpr($cond$, $condBB$) ▷ 在条件块中计算条件
      emit($condBB$, br(v, $bodyBB$, $afterBB$)) ▷ 根据条件进入循环体或退出
      GenIR($body, bodyBB, afterBB, condBB$) ▷ 更新 break/continue 目标
      emit_if_no_term($bodyBB$, br($condBB$)) ▷ 若未终结，则回跳形成循环
    case Break ⇒
      emit($curBB$, br($breakTarget$)) ▷ 跳转到当前循环的退出块
    case Continue ⇒
      emit($curBB$, br($continueTarget$)) ▷ 跳转到当前循环的条件判断块
    case _ ⇒
      translate($stmt$, $curBB$) ▷ 普通语句在线性附加到当前基本块
  end match
end procedure

7.2.2 指令参数的定义和使用

在翻译每条 IR 指令时，首先需要明确其操作数的来源。理想情况下，应尽量复用已经保存在寄存器中的计算结果，而不是频繁地从局部变量中通过 load 指令重新读取。然而，实际情况中，参数往往定义于不同的基本块，且可能存在多重定义（如控制流汇合处），若在 IR 翻译阶段直接处理这些问题，将显著增加实现复杂度。

因此，在当前阶段的翻译过程中，我们暂不考虑跨基本块的优化，而是将参数的定义与使用关系限制在单个基本块内部，避免直接依赖来自其它基本块的寄存器值。具体策略如下：

- 所有局部变量在使用前必须先通过 load 指令读入寄存器；
- 所有局部变量在更新后立即通过 store 写回内存。

代码 7.15 与 7.16 分别给出了阶乘函数的 Tea 语言源代码及其对应的 IR 表示。这种方式虽然牺牲了一定的执行效率，但能够有效简化实现逻辑，并保证语义的正确性。

```
fn fac(n:i32) -> i32 {
    let r = 1;
    while (n>0) {
        r = r * n;
        n = n-1;
    }
    return r;
}
```

代码 7.15: Tea 语言代码

```
define i32 @foo(i32 %0) {
bb0:
    %n = alloca i32 ; 参数内存单元
    %r = alloca i32
    store i32 %0, ptr %n ; 保存参数值
    store i32 1, ptr %r
    br label %bb1
bb1:
    %t1 = load i32, ptr %n ; 使用变量的值前先load，限制临时变量%t1仅在当前代码块使用
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, ptr %r ; 使用变量的值前先load，避免与其它代码块中的%r值耦合
    %t4 = load i32, ptr %n ; 使用变量的值前先load，避免与其它代码块中的%n值耦合
    %t5 = mul i32 %t3, %t4 ; 限制临时变量%t5仅在当前代码块使用
    store i32 %t5, ptr %r ; 立即更新%r的内存单元，保证后续指令可以load到最新的数值
    %t6 = load i32, ptr %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, ptr %n ; 立即更新%n的内存单元，保证后续指令可以load到最新的数值
    br label %bb1
bb3:
    %t8 = load i32, ptr %r
    ret i32 %t8
}
```

代码 7.16: 代码 7.15 对应的 IR

7.3 解释执行

线性 IR 通过消除 `if-else` 和 `while` 等语法结构，将程序转换为由基本块（Basic Block）和显式跳转指令组成的形式，其结构已经非常接近底层汇编代码。因此，可以从程序入口函数开始，按照控制流顺序逐条解释执行 IR 指令。

指令加载与解析 在执行之前，需要将文本形式的 LLVM IR 加载并解析为内部数据结构。解析过程首先对 IR 进行词法与语法分析，识别每条指令的组成部分。例如，`%t5 = mul i32 %t3, %t4` 可以解析为：

- 指令（opcode）：`mul`
- 类型信息：`i32`
- 操作数：`%t3`、`%t4`
- 结果寄存器：`%t5`

在此基础上，需要进一步按基本块对指令进行组织。每个基本块包含一段顺序执行的指令序列，并以终结指令（如 `br`、`ret`）结束。从而方便解释执行过程中根据标签（如 `bb1`、`bb2`）进行跳转寻址。

虚拟机 解释执行的核心问题在于：如何保存并传递指令执行过程中产生的中间结果。为此，需要引入虚拟机来模拟程序运行环境。根据执行模型的不同，虚拟机通常可以分为两类：

- **栈虚拟机（Stack-based VM）**：操作数通过操作数栈隐式传递，指令通常不显式给出操作数位置。例如在 `Java Bytecode` 或 `Python` 字节码中，`add` 指令会从栈顶弹出两个操作数并将结果压回栈中。这类模型实现简单，但指令间的数据依赖较为隐式。
- **寄存器虚拟机（Register-based VM）**：操作数显式存储在“寄存器”中，指令直接引用这些寄存器。LLVM IR 本质上是一种寄存器虚拟机模型，其 SSA 变量（`%t1`, `%t2` 等）可以看作数量无限的虚拟寄存器。因此，在实现解释器时，更自然的方式是采用寄存器模型，而非栈模型。

在解释执行过程中，虚拟机通常需要维护以下几类状态：

- **寄存器**：用于存储 SSA 变量的当前值，可实现为“变量名 → 数值”的映射表。
- **内存模型**：对应 `alloca` 分配的栈空间，可以用线性内存或哈希表模拟地址到数值的映射。
- **函数栈桢**：每次函数调用时创建新的栈桢，包含局部变量、参数以及返回地址等信息。
- **程序计数器**：指示当前执行的基本块及其内部指令位置。

基于上述结构，解释执行的基本流程为：从入口基本块开始，顺序执行指令；遇到普通指令时更新寄存器或内存状态；遇到跳转指令时根据条件切换基本块；遇到函数调用指令时创建新的栈桢并进入被调函数执行；遇到返回指令时结束当前函数执行并将结果返回给调用者。该过程在语义上与真实机器执行类似，但所有状态均由虚拟机在软件中维护。

参考文献

[1] LLVM 语言参考文档-指令部分, `https://llvm.org/docs/LangRef.html#instruction-reference`.

练习

1) 使用控制流指令，通过短路求值方式改写以下代码中的 and 指令，并使用 lli 工具验证其执行结果。

```
define i32 @foo(i32 %0, i32 %1) {
    %t1 = alloca i32
    %t2 = alloca i32
    store i32 %0, ptr %t1
    store i32 %1, ptr %t2
    %t3 = load i32, ptr %t1
    %t4 = load i32, ptr %t2
    %t5 = icmp sgt i32 %t3, %t4
    %t6 = load i32, ptr %t1
    %t7 = icmp ne i32 %t6, 0
    %t8 = and i1 %t5, %t7
    %t9 = zext i1 %t8 to i32
    ret i32 %t9
}
define i32 @main() {
    %1 = call i32 @foo(i32 2, i32 1)
    ret i32 %1
}
```

代码 7.17: LLVM IR 代码片段

2) 将下列 Tea 语言代码翻译为线性 IR，并使用 lli 工具进行测试。

```
let a[i32;10] = {1,3,5,7,9,2,4,6,8,10};
fn binsearch(x:i32) -> i32 {
    let high:i32 = 9;
    let low:i32 = 0;
    let mid:i32 = (high + low)/2;
    while a[mid] != x && low < high {
        mid = (high + low) / 2;
        if x < a[mid] {
            high = mid-1;
        } else {
            low = mid +1;
        }
    }
    if x == a[mid] {
        return mid;
    }
    else {
        return -1;
    }
}
fn main() -> i32 {
    let r = binsearch(2);
    return r;
}
```

代码 7.18: Tea 语言代码片段