从抽象到具体：一份面向初学者的编译器中间表示生成教程

引言：搭建编译器前后端的桥梁——中间表示

在深入探讨如何生成中间表示（Intermediate Representation, IR）之前，我们首先需要理解它在编译器中的核心地位。可以将一个完整的编译器想象成一条精密的“代码加工流水线”[1]。这条流水线的一端是程序员编写的、符合人类思维逻辑的源程序（例如用Java, C++, Tiger等语言编写的代码）；另一端则是计算机硬件能够直接执行的、冰冷的目标代码（即机器码）。

编译器的前端（Frontend）负责“理解”源码，它通过词法分析、语法分析和语义分析，将源程序从一串文本字符，转化为一种结构化的、能够体现程序内在逻辑的数据结构——通常是抽象语法树（Abstract Syntax Tree, AST）。这就像一个厨师读懂了一份复杂的菜谱，并将其中的步骤和食材关系在脑海中形成了一幅清晰的蓝图 [1]。

与之相对，编译器的后端（Backend）则负责“执行”这个蓝图，它将中间表示翻译成特定目标机器的指令集，并进行寄存器分配、指令调度等与硬件紧密相关的优化，最终生成可执行的机器码。这好比厨师根据脑中的蓝图，在某个具体的厨房里（特定的CPU架构），使用特定的厨具（指令集），完成菜品的烹饪 [1]。

而中间表示（**IR**），正是连接前端和后端的关键桥梁。它是一种独立于源语言和目标机器的“通用语言”或“通用配方”。

为什么需要**IR**？——模块化与可移植性的基石

引入中间表示并非多此一举，而是编译器工程化、模块化发展的必然选择。它主要解决了两大核心问题：

1. 降低复杂性：想象一下，如果没有IR，我们需要为 N 种不同的编程语言和 M 种不同的目标机器开发编译器。这意味着我们需要编写 N×M 个完全不同的编译器后端，工作量巨大且难以维护 [1]。
2. 提高复用性：引入IR后，情况大为改观。我们只需要编写 N 个前端，负责将各种源语言翻译成统一的IR；再编写 M 个后端，负责将这个IR翻译到不同的目标机器上。总工作量从 N×M 锐减到 N+M，大大提高了代码的复用性和项目的可扩展性 [1]。

这种设计体现了软件工程中“关注点分离”的核心思想。前端专注于理解源语言的语义，后端专注于目标机器的特性，而IR则作为它们之间清晰的接口。一个优秀的IR设计，本身就是一种精妙的权衡：它必须足够抽象，以摆脱源语言的语法细节；同时又必须足够具体，以便能高效地映射到不同目标机器的指令集上。这种“承上启下”的特性，是IR设计的核心挑战与魅力所在。

初识**IR**：三地址码

为了让初学者对IR有一个直观的认识，我们首先介绍一种经典且简单的线性IR——三地址

码（**Three-Address Code, TAC**）。顾名思义，三地址码的核心特点是每条指令最多只包含一个运算符和三个“地址” [1]。其通用形式为：

x=y op z
这里的“地址”可以是一个源程序中的变量名、一个常量，或者是编译器在翻译过程中生成的临时变量（temporary）[1]。三地址码将复杂的表达式拆解成一系列简单的、原子性的运算步骤，非常接近计算机的实际执行模型。

常见的三地址码指令类型包括 [1]：

序号	指令类型	指令形式
1	赋值指令	x=y op z 或 x=op y
2	复制指令	x=y
3	条件跳转	if x relop y goto n
4	非条件跳转	goto n
5	参数传递	param x
6	过程调用	call p, n
7	过程返回	return x

让我们通过一个计算阶乘的例子，直观感受一下从高级语言到三地址码的转换过程 [1]。

高级语言代码：

```
read x;
if 0 < x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1;
  until x = 0;
  write fact;
end
```

对应的三地址码序列：

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

这个例子清晰地展示了三地址码如何将源程序的结构“拉平”，变成一个线性的指令序列。这种形式虽然直观，但丢失了源程序原有的层次结构。接下来，我们将深入学习本教程的核心——一种保留了树状结构的IR：**IR Tree**。

第一部分：深入剖析IR Tree——一种树状中间表示

与三地址码这种线性表示不同，本教程将聚焦于一种特殊的树状中间表示——**IR Tree**。它在现代编译器（如Tiger编译器）中扮演着核心角色。

1.1 IR Tree的核心理念：介于AST与汇编之间

IR Tree的设计目标是在抽象语法树（AST）和底层汇编指令之间取得一个平衡点。一方面，它像AST一样保留了程序的层次化结构；另一方面，它的节点所代表的操作又非常接近于目标机器的指令，比如内存访问、二元运算等 [1]。

这种设计体现了编译器设计中一个重要的思想：“逐步降低抽象层次”（Lowering）。IR Tree没有像三地址码那样完全“拉平”代码结构，其保留的树状结构为后续的指令选择阶段（尤其是采用树匹配算法时 ）提供了极大的便利 [2]。可以说，IR Tree的结构本身就是为了简化从它到目标代码的映射过程而精心设计的。

IR Tree由两大类节点构成：用于计算值的表达式（**Expression, Exp**）和用于执行动作的语句（**Statement, Stm**） [1]。

1.2 IR Tree的“计算单元”：表达式 (Exp)

表达式（Exp）节点用于计算某个值，但计算过程中也可能产生副作用（Side effects），例如

修改内存或寄存器的内容 [1]。下面是IR Tree中所有表达式节点的详细介绍。

表格1：IR Tree表达式节点 (Exp) 概览 [1]
节点 (Node)	语法 (Syntax)	功能描述 (Description)	示例 (Example)
CONST(i)	CONST(i)	整数常量i。	CONST(42) → 值为42
NAME(n)	NAME(n)	符号常量n，通常是一个代码标签（Label），其值为该标签的内存地址。	NAME(L1) → 标签L1的地址
TEMP(t)	TEMP(t)	临时变量t。可以看作是一个虚拟寄存器，在寄存器分配前我们不关心其数量。	TEMP(t123) → 临时变量t123中存储的内容
BINOP(o,e1,e2)	BINOP(o,e1,e2)	对两个子表达式e1和e2应用二元操作o（如加减乘除、位运算等）。	BINOP(PLUS, TEMP(t1), CONST(1)) → 计算 t1 + 1
MEM(e)	MEM(e)	访问内存。e是一个计算地址的表达式。MEM节点既可以表示从内存加载数据（fetch），也可以表示向内存存储数据（store）。	MEM(TEMP(fp)) → 访问帧指针fp所指向地址的内存内容
CALL(f,l)	CALL(f,l)	调用函数f，参数列表为l。f通常是一个NAME节点，l是Exp节点的列表。	CALL(NAME(print),) → 调用 print(t1)
ESEQ(s,e)	ESEQ(s,e)	顺序组合。先执行语句s（为了其副作用），然后计算表达式e并返回其结果。	ESEQ(MOVE(TEMP(t), CONST(1)), TEMP(t)) → (t=1; t)，整个表达式的结果是t的值，即1

重点剖析：**ESEQ**与副作用

在所有表达式节点中，ESEQ最为特殊和精妙。它解决了在一个以“表达式求值”为核心的树状结构中，如何优雅地嵌入“产生副作用的语句”这一难题 [1]。

副作用（**Side effects**）是指在计算表达式之外，对程序状态产生的任何可观察的改变，最常见的就是修改某个内存单元或临时变量（寄存器）的值 [1]。例如，a=5这个赋值操作，其主要目的就是产生副作用（改变a的值），它本身不返回值。

ESEQ(s, e)的语义是：先执行语句s，s的执行是为了其副作用，s本身不返回值。执行完s后，再计算表达式e，整个ESEQ节点的值就是e的值。例如，ESEQ(MOVE(TEMP(a), CONST(5)), BINOP(PLUS, TEMP(a), CONST(5)))，会先执行MOVE语句使a的值变为5，然后计算a+5，最终整个表达式的结果是10。

ESEQ的存在使得IR Tree能够用统一的树形结构，既能表示纯粹的计算，也能表示带有状态更新的复杂命令式操作，是理解IR Tree的关键一环。

1.3 IR Tree的“动作指令”：语句 (Stm)

与表达式（Exp）不同，语句（Stm）节点用于执行动作、改变状态或控制程序的执行流程，它们本身不返回任何值 [1]。

表格2：IR Tree语句节点 (Stm) 概览 [1]
节点 (Node)	语法 (Syntax)	功能描述 (Description)	示例 (Example)

MOVE(d,s)	MOVE(d,s)	将源表达式s计算出的值，移动到目标位置d。d通常是TEMP或MEM节点。	MOVE(TEMP(t1), CONST(42)) → t1 = 42
EXP(e)	EXP(e)	计算表达式e，但完全忽略并丢弃其计算结果。这通常是为了利用e的副作用。	EXP(CALL(NAME(print),...)) → 调用print()函数，不关心其返回值，只为了其打印效果
JUMP(e,labs)	JUMP(e,labs)	无条件跳转。跳转到由表达式e计算出的地址。e通常是一个NAME节点。	JUMP(NAME(L1), [L1]) → goto L1
CJUMP(o,e1,e2,t,f)	CJUMP(o,e1,e2,t,f)	条件跳转。比较e1和e2（使用关系操作o），如果结果为真，则跳转到标签t；否则，跳转到标签f。	CJUMP(LT, TEMP(t1), CONST(0), L1, L2) → if t1 < 0 goto L1 else goto L2
SEQ(s1,s2)	SEQ(s1,s2)	语句序列。按顺序先执行语句s1，然后执行语句s2。	SEQ(MOVE(...), JUMP(...)) → 先执行一个赋值语句，再执行一个跳转语句
LABEL(n)	LABEL(n)	定义标签。将符号名n的值定义为当前指令在内存中的地址。它作为跳转指令的目标。	LABEL(L1) → 定义一个名为L1的标签，相当于汇编中的 L1:

1.4 综合示例：用**IR Tree**描述一个**while**循环

理论是枯燥的，让我们通过一个具体的例子，将前面介绍的节点组合起来，看看如何用IR Tree来描述一个简单的while循环。这个例子将成为连接理论与后续翻译实践的桥梁 [1]。

高级语言代码：

```
n := 0;
while (n < 10) do
  n := n + 1
```

对应的**IR Tree (**文本表示**):**

```
SEQ(MOVE(TEMP(n), CONST(0)),
    LABEL(HEAD),
    CJUMP(LT, TEMP(n), CONST(10), BODY, END),
    LABEL(BODY),
    MOVE(TEMP(n), BINOP(ADD, TEMP(n), CONST(1))),
    JUMP(NAME(HEAD)),
    LABEL(END))
```

对应的IR Tree (图形化结构):
!(IR Tree while循环示例)
[1]
这棵树的逻辑非常清晰：

- 最外层的SEQ节点将整个代码块串联起来。
- MOVE(TEMP(n), CONST(0))：对应循环前的初始化 n := 0。

- LABEL(HEAD)：定义了循环开始的标签，作为循环判断和返回的锚点。
- CJUMP(LT, TEMP(n), CONST(10), BODY, END)：这是循环的核心判断。如果 n < 10 为真，跳转到BODY标签；否则，跳转到END标签，退出循环。
- LABEL(BODY)：定义了循环体的入口。
- MOVE(TEMP(n), BINOP(ADD, TEMP(n), CONST(1)))：这是循环体的内容，对应 n := n + 1。
- JUMP(NAME(HEAD))：循环体执行完毕后，无条件跳转回HEAD标签，进行下一次循环判断。
- LABEL(END)：定义了循环结束后的代码位置。

通过这个例子，我们可以看到IR Tree如何用其节点精确地构建出高级语言的控制流结构。接下来的部分，我们将系统地学习如何从抽象语法树（AST）自动地生成这样一棵IR Tree。

第二部分：翻译的艺术——从抽象语法树到IR Tree

掌握了IR Tree的结构之后，本部分将进入教程的实践核心：系统性地讲解如何将源语言的各种语法结构（如变量访问、if语句、while循环、函数调用等）一步步地翻译成我们已经熟悉的IR Tree。

2.1 翻译的核心框架：Ex, Nx, Cx

在将AST节点翻译成IR Tree时，我们会发现一个问题：同一个语法结构，在不同的上下文中，其扮演的角色和我们期望的翻译结果是不同的。例如，x > 0这个表达式：

- 在 if (x > 0) then... 中，它充当一个条件（**Conditional**），我们希望它被翻译成一个条件跳转指令。
- 在 y := (x > 0) 中，它充当一个产生值的表达式（**Expression**），我们希望它被翻译成能计算出布尔值（通常是1或0）的IR。

为了优雅地处理这种上下文依赖性，我们引入一个翻译框架，将AST表达式分为三类 [1]：

- **Ex (Expression)**：一个能够计算出结果值的表达式。它应该被翻译成一个Tree.exp节点。例如，a + b。
- **Nx (No-result Expression / Statement)**：一个不产生任何结果值的语句。它应该被翻译成一个Tree.stm节点。例如，赋值语句 a := b 或一个 while 循环。
- **Cx (Conditional Expression)**：一个用于控制流程的条件表达式。它不直接计算出0或1，而是被翻译成一个能够根据条件真假跳转到不同目标标签的Tree.stm。最典型的例子就是 CJUMP。

这个框架的精髓在于，源语言的语法结构（如OpExp）和它在特定上下文中的语义角色（Ex, Nx, Cx）是解耦的。我们只需要为每种AST节点编写一个“标准”的翻译函数（通常是生成Cx或Ex），然后定义一组转换工具函数（如unEx, unNx, unCx）来将一种形式适配到另一种形式。

例如，当if语句需要一个条件（Cx），而我们提供了一个x+1（Ex）时，Ex -> Cx的转换会自动将x+1的结果与0比较，生成一个CJUMP。反之，当赋值语句需要一个值（Ex），而我们提供了一个x>0（Cx）时，Cx -> Ex的转换会生成一段代码，使用一个临时变量，当条件为真时给它赋值1，为假时赋值0，最后返回这个临时变量。

这种设计本质上是一种“适配器模式”，它极大地简化了翻译逻辑，避免了为每种可能的语法和上下文组合编写重复且复杂的代码，是实现一个清晰、可维护的翻译器的关键策略 [1]。

2.2 翻译变量与数据结构

简单变量访问

在函数中访问一个局部变量，实际上是访问它在当前函数**栈帧（Stack Frame）**中的存储位置。栈帧是函数调用时在栈上分配的一块内存区域，用于存放局部变量、参数等信息。我们通过一个特殊的寄存器——**帧指针（Frame Pointer, fp）**来定位当前的栈帧。

因此，访问一个距离帧指针偏移量为k的局部变量v，其IR Tree表示为 [1]：

MEM(BINOP(PLUS,TEMP(fp),CONST(k)))
这棵树的含义是：取fp寄存器的值，加上常量偏移k，得到变量的实际内存地址，然后通过MEM节点访问该地址的内容 [1]。

对于支持嵌套函数的语言（如Tiger），一个内部函数可能需要访问定义在外部函数中的变量。这通过静态链接（Static Link）实现。访问一个嵌套层级为n的变量，需要沿着静态链进行n次解引用，其IR Tree会形成一个嵌套的MEM结构，例如 [1]：

MEM(+(CONST kx,MEM(+(... MEM(+(CONST ksl,TEMP(fp)))...))))

这表示从当前帧指针fp开始，首先找到指向外层函数栈帧的静态链接，然后一层层回溯，直到找到目标变量所在的栈帧，最后加上变量在该帧内的偏移量k_x来访问它。
R值与L值 (R-Values and L-Values)

在深入翻译赋值和访存之前，理解**L值**和**R值**的概念至关重要 [1]。

- **R值 (R-value)**：指表达式的“值”，通常出现在赋值号的右边，如 a+3。它是一个可以被计算但不能被赋值的量。
- **L值 (L-value)**：指一个可被赋值的“内存位置”，通常出现在赋值号的左边，如变量x、数组元素a[i]。它也可以出现在右边，此时表示该位置存储的“内容”。

在IR Tree中，MEM节点的角色由其上下文决定：

- 当作为MOVE(d, s)语句的目标d时，MEM(...)代表一个**L值**，表示要向这个内存地址**存入（Store）**数据。
- 当出现在其他任何地方时，MEM(...)代表一个**R值**，表示要从这个内存地址**取出（

Fetch/Load）**数据。

标量与结构化**L**值 **(Scalar vs. Structured L-Values)**

不同语言对L值的处理方式有很大差异，这深刻影响了IR的生成 [1]。

- 标量**L**值 **(Scalar L-value)**：L值所代表的内存位置只包含一个不可再分的数据单元（如一个整数或一个指针）。Tiger语言的设计极大地简化了翻译，因为它规定所有变量都是标量。数组和记录变量本身只存储一个指针，这个指针是标量。
- 结构化**L**值 **(Structured L-value)**：L值代表一块包含多个组成部分的内存区域，例如C语言中的结构体或Pascal中的数组。对结构化L值的赋值（如 struct1 = struct2）意味着要拷贝整个数据结构的内容，而不仅仅是一个指针。

Tiger的“一切皆为标量指针”的设计哲学，意味着赋值操作永远是简单的单字（word-sized）拷贝，MEM(e)足以完成所有访存操作。而对于支持结构化L值的语言，编译器在生成IR时必须考虑数据的大小和对齐，可能需要生成特殊的块拷贝（block move）指令 [1]。

数组和记录

理解了Tiger中“变量是指针”的核心思想后，我们再来看数组和记录的翻译。

数组下标访问 **a[i]**

基于指针语义，访问数组元素a[i]需要两步：首先获取数组的基地址（即变量a中存储的指针值），然后计算元素的偏移地址。其IR Tree结构如下 [1]：

MEM(BINOP(PLUS,MEM(ea),BINOP(MUL,ei,CONST(W))))
让我们拆解这个复杂的结构：

1. e_a是表示变量a（一个L值）的表达式，通常是 MEM(+(TEMP(fp), CONST(k_a)))。
2. MEM(e_a)：第一次**MEM**访问。这是为了解引用指针，获取存储在变量a中的值（R值），也就是数组在堆上的基地址。
3. e_i是计算索引i的表达式。
4. BINOP(MUL, e_i, CONST(W))：计算索引偏移。将索引i乘以每个元素的大小W（Word size，例如4字节）。
5. BINOP(PLUS,...)：将基地址和索引偏移相加，得到目标元素的最终内存地址。
6. 最外层的MEM(...)：第二次**MEM**访问。访问上一步计算出的最终地址，这个MEM节点既可以作为L值（用于赋值 a[i] := ...），也可以作为R值（用于计算 ... := a[i]）。

记录字段访问 **r.f**

记录字段的访问与数组类似，但偏移量是固定的。如果字段f是记录的第n个字段（从0开始），其偏移量就是n * W。翻译结果为 [1]：

MEM(BINOP(PLUS,MEM(er),CONST(n*W)))

这里的MEM(e_r)同样是用于获取记录在堆上的基地址。

2.3 翻译算术与控制流语句

算术运算

算术运算的翻译相对直接 [1]：

- 二元运算：源语言中的加减乘除等二元运算符，直接映射到BINOP(op, e1, e2)节点。
- 一元运算：IR Tree中没有专门的一元运算节点。一元运算需要被转换为等价的二元运算。
  - 一元取负 -x 被翻译为 BINOP(MINUS, CONST(0), e_x)，即 0 - x。
  - 一元按位取反 ~x 被翻译为 BINOP(XOR, e_x, CONST(-1))，即 x XOR 0xFF..FF。
- 浮点数：本教程讨论的IR Tree不包含浮点运算。

if-then-else 语句

if语句的翻译是构建控制流图的典型例子。一个if test then arm1 else arm2语句会被翻译成如下结构的IR Tree [1]：

```
SEQ(CJUMP(op, e1, e2, t, f),
  SEQ(LABEL(t),
    SEQ(stm1,
      SEQ(JUMP(NAME(join)),
        SEQ(LABEL(f),
          SEQ(stm2,
            LABEL(join)))))))
```

这里的逻辑是：

1. CJUMP(...)：对test表达式进行翻译，生成条件跳转。如果为真，跳到t标签（then分支的入口）；如果为假，跳到f标签（else分支的入口）。
2. LABEL(t) 和 stm1：then分支的代码。
3. JUMP(NAME(join))：then分支执行完毕后，无条件跳转到join标签，跳过else分支。
4. LABEL(f) 和 stm2：else分支的代码。
5. LABEL(join)：then和else分支的汇合点，程序从这里继续往下执行。

逻辑运算符的短路求值

Tiger语言的逻辑与&和逻辑或|要求短路求值（Short-Circuit Evaluation）。这意味着一旦表达式的最终结果可以确定，就不再计算后续的部分 [1, 3]。例如，在a & b中，如果a为假，整个表达式必为假，此时不应再计算b。

这个语义深刻地影响了IR的生成。短路求值从根本上将一个逻辑运算问题转化为了一个控

制流问题。&和|并不对应于单一的BINOP节点，而是被翻译成一个由CJUMP和LABEL构成的代码片段 [4, 5, 6]。

a & b (逻辑与) 的翻译 [1]:

1. 创建一个新的内部标签 middle。
2. 翻译a，生成一个条件跳转：如果a为真，跳转到middle；如果为假，直接跳转到整个&表达式的false出口。
3. 在middle标签处，翻译b，生成另一个条件跳转：如果b为真，跳转到整个&表达式的true出口；如果为假，跳转到false出口。

a | b (逻辑或) 的翻译 [1]:

1. 创建一个新的内部标签 middle。
2. 翻译a，生成一个条件跳转：如果a为真，直接跳转到整个|表达式的true出口；如果为假，跳转到middle。
3. 在middle标签处，翻译b，生成另一个条件跳转：如果b为真，跳转到整个|表达式的true出口；如果为假，跳转到false出口。

while 循环

while循环的翻译模式相对固定 [1]：

```
SEQ(LABEL(test),
  SEQ(CJUMP(..., body, done),
    SEQ(LABEL(body),
      SEQ(body_stm,
        SEQ(JUMP(NAME(test)),
          LABEL(done))))))
```

1. LABEL(test)：循环判断的入口。
2. CJUMP(...)：对循环条件进行判断，若为真跳到body，否则跳到done退出循环。
3. LABEL(body) 和 body_stm：循环体的代码。
4. JUMP(NAME(test))：循环体执行完后，无条件跳回test进行下一次判断。
5. LABEL(done)：循环的出口。

对于循环中的break语句，其翻译非常直观：直接生成一个JUMP(NAME(done))指令，跳出循环 [1]。

for 循环

for i := lo to hi do body循环的一个直接想法是将其改写为等价的while循环 [1]：

let var i := lo, var limit := hi in

```
  while i <= limit do
    (body; i := i + 1)
end
```

然而，这种简单的转换存在一个严谨性问题：当hi的值恰好是机器能表示的最大整数时，循环最后一次执行i := i + 1会导致整数溢出 [1]。

一个更健壮的翻译方案是，在进入循环前先检查lo <= hi，并且将循环判断放在循环体的末尾，在增加i之前检查i < limit。这避免了对可能溢出的值进行递增操作。这个例子提醒我们，编译器设计充满了需要仔细考虑的边界情况和潜在陷阱。

2.4 翻译函数

函数调用

翻译一个函数调用f(a1,..., an)，除了要处理所有参数表达式的求值外，一个关键点是处理支持嵌套函数所必需的静态链接（**Static Link**）[1]。

翻译结果为：

CALL(NAME(lf),[sl,e1,...,en])
这里的sl就是静态链接。它是一个隐藏的、额外的参数，指向调用者的静态父级函数的栈帧 [7, 8]。当一个内层函数需要访问外层函数中定义的非局部变量时，它就通过静态链接（以及可能由多个静态链接组成的静态链）逐层回溯，找到定义该变量的栈帧，并从中读取数据 [8, 9]。

函数声明

一个完整的函数声明，在翻译后会形成一个代码段，逻辑上可以分为三部分：序言、函数体和尾声 [1]。这个过程清晰地展示了编译器如何管理程序的运行时状态，以及如何遵循与操作系统和硬件之间的“调用约定”（Calling Convention）。

1. 序言 **(Prologue)**：函数开始执行前的一系列准备工作 [1]。
    - 放置函数标签，使其可以被调用。
    - 调整栈指针（sp），为当前函数分配新的栈帧。
    - 将返回地址、旧的帧指针（fp）以及任何需要由被调用者保存的寄存器（callee-save registers）压入栈中，以便函数返回时恢复。
    - 处理传入的参数，包括将静态链接保存到当前栈帧的特定位置。
2. 函数体 **(Body)**：这是函数的核心逻辑。我们将函数内的表达式（在Tiger中，函数体就是一个表达式）翻译成IR Tree [1]。
3. 尾声 **(Epilogue)**：函数返回前的一系列清理工作 [1]。
    - 将函数的返回值（如果有的话）移动到约定的返回值寄存器中（如eax）。

- 从栈中恢复之前保存的被调用者保存寄存器、旧的帧指针和返回地址。
- 重置栈指针，释放当前函数的栈帧。
- 执行返回指令，跳转到之前保存的返回地址，将控制权交还给调用者。

结论：从**IR Tree**到真实世界

本教程带领大家完成了一段从抽象到具体的旅程。我们首先理解了中间表示（IR）在现代编译器架构中的必要性，它作为前端和后端之间的“通用语”，是实现模块化和可移植性的关键。

接着，我们深入剖析了IR Tree这一强大的树状中间表示。通过学习其Exp（表达式）和Stm（语句）两类节点，我们掌握了阅读和理解IR Tree的能力。

教程的核心部分，我们系统地学习了如何将源语言的各种结构翻译为IR Tree。我们掌握了Ex, Nx, Cx这一核心翻译框架，它能灵活地应对不同上下文对同一语法结构的不同语义要求。我们实践了对变量、数组、记录等数据结构的访问翻译，特别是理解了L值与R值、标量与结构化L值，以及Tiger语言中指针语义的重要性。我们还攻克了算术运算和控制流翻译的难点，如if语句的构建、while循环的标准模式，以及将逻辑运算（&, |）的短路求值巧妙地转化为控制流跳转。最后，我们通过函数调用和声明的翻译，窥见了编译器管理运行时栈帧和遵循调用约定的底层机制。

您现在掌握的IR生成技术，是连接编译器前端的语义理解和后端代码生成的关键枢纽，是整个编译过程中最富有创造性和挑战性的环节之一。但这并非编译之旅的终点。在生成IR Tree之后，编译器通常还会进行以下重要步骤：

1. 规范化 **(Canonicalization)**：生成的IR Tree中含有ESEQ和SEQ节点，使得树的结构比较复杂。规范化过程会将这些复杂的树“拉平”，转换成一个线性的基本块（**Basic Blocks**）序列。每个基本块内部没有跳转，只有一个入口和一个出口。这些基本块再被组织成踪迹（**Traces**），以优化跳转指令，提高指令缓存的命中率 [2, 10]。
2. 指令选择 **(Instruction Selection)**：经过规范化处理后，线性的IR指令序列将进入指令选择阶段。编译器会用目标机器的指令集模式去“覆盖”或“平铺”（Tiling）这些IR指令。这个过程就像用一套固定的瓷砖去铺满整个地面，目标是找到成本最低（如执行速度最快或代码体积最小）的平铺方案，从而将IR映射成使用虚拟寄存器（TEMP）的汇编指令 [2, 11, 12]。
3. 寄存器分配 **(Register Allocation)**：最后，指令选择生成的汇编指令使用的是无限的虚拟寄存器。寄存器分配阶段的任务，就是将这些数量众多的虚拟寄存器，高效地映射到目标机器有限的物理寄存器上。这是一个复杂的图染色问题，其结果直接影响最终代码的性能。

编译器的世界广阔而深邃。希望本教程能为您打下坚实的基础，并激发您继续探索编译器后端优化与代码生成等更深层次领域的兴趣。