中间代码生成　实习指导

许畅 陈嘉 朱晓瑞

> It is easy to ignore responsibility when one is only an intermediate link in a chain of action.
>
> —— Stanley Milgram

编译器里核心的数据结构之一就是中间代码（Intermediate representation，简称 IR）。中间代码应当包含哪些信息，这些信息又应当有怎样的内部表示将会极大地影响到编译器的代码复杂程度、编译器的运行效率以及编译出来的目标代码的运行效率。

广义地说，编译器中根据输入程序所构造出来的绝大多数结构都被称为中间代码（或者更精确地译作“中间表示”）。例如，我们之前所构造的词法流、语法树、带属性的语法树等等，都可以看作是一种中间代码。使用中间代码的主要原因当然是为了方便我们在编程时的各种操作，毕竟如果我们在需要有关输入程序的任何信息时都只能重新去读入并处理输入程序的源代码的话， 不仅会使得编译器的编写变得麻烦，也会大大减慢其运行效率。况且经过前面两次实习我们都应该清楚，输入程序里包含的绝大部分编译器需要的信息都没有被显式地表达出来。

狭义地说，中间代码是编译器从源语言到目标语言之间采用的一种过渡性质的代码形式（这时它常被称作 intermediate code）。看到这里你可能会产生疑问：为什么编译器不能直接把输入程序直接翻译成目标代码，而是要额外地多一道手续，引入中间代码呢？这难道不是自己给自己找麻烦吗？实际上，引入中间代码有两个主要的好处：一方面，中间代码将编译器自然地分成了前端和后端两个部分。当我们需要改变编译器的源语言或者目标语言时，如果采用中间代码，那么我们只需要替换原有编译器的前端或者后端，而不需要重写整个编译器。另一方面，即使源语言和目标语言是固定的，采用中间代码也有利于编译器的模块化。人们将编译器设计中的那些复杂但相关性又不是很大的任务分别放在前端和后端的各个模块之中，这样既简化了模块内部的处理，又使得我们能单独对每个模块进行调试与修改而不影响到其它模块。下文中，如果不是特别说明，那么所有出现的“中间代码”都指的是本段所介绍的狭义的中间代码。

中间代码的分类

中间代码的设计可以说更多的是一门艺术而不是技术。不同的编译器所使用的中间代码可以是千差万别，而就算是同一个编译器内部也可以使用多种不同的中间代码——有的中间代码与源语言更加接近，有的中间代码与目标语言更加接近。编译器需要在不同的中间代码之间进行转换，有时候为了处理的方便甚至会在将中间代码 1 转换为中间代码 2 之后，对中间代码 2 进行优化然后又转换回中间代码 1。这些不同的中间代码虽然对应了同一个输入程序，但它们却体现了输入程序不同层次上的细节信息。举一个实际的例子：gcc 内部首先会将输入程序转换成一棵抽象语法树，然后将这棵树转换成另一种被称为 GIMPLE 的树形结构，在 GIMPLE 之上建立静态单赋值（SSA）式的中间代码以后，又会将它转换为一种非常底层的 RTL（Register Transfer Language）代码，最后才把 RTL 变为汇编代码。

我们可以从不同的角度对现存的这些花样繁多的中间代码进行分类。从代码所体现出的细节上，我们可以将中间代码分为如下三类：

- 高层次中间代码（High-level IR, HIR）

  这种中间代码更多地体现了比较高层次的细节信息，因此往往和高级语言比较类似，保留了不少包括数组、循环在内的源语言的特征。高层次中间代码常在编译器的前端部分使用，并在之后被转换为更加低层次的中间代码。高层次中间代码常被用于进行相关性分析(dependence analysis)和解释执行。我们熟悉的 Java bytecode、Python .pyc bytecode 以及目前使用得非常广泛的 LLVM IR 都属于高层次 IR
- 中层次中间代码（Medium-level IR, MIR）

  这个层次的中间代码的形式介于源语言和目标语言之间。它既体现了许多高级语言的一般特性，又可以被方便地转换成低级语言的代码。正是由于 MIR 的这个特性，我们认为它是三种 IR 中最难设计的一种 IR。在这个层次上，变量和临时变量可能已经有了区分，控制流也可能已经被简化为无条件跳转、有条件跳转、函数调用和

函数返回四种。另外，对中层次中间代码可以进行绝大部分优化处理，例如公共子表达式消除（common-subexpression elimination）、代码移动（code motion）、代数运算简化（algebraic simplification）等。

- 低层次中间代码（Low-level IR, LIR）

  低层中间代码基本上与目标语言已经非常接近，它在变量的基础上可能会加入寄存器的细节信息。事实上，LIR 中的大部分代码和目标语言中的指令往往存在着一一对应的关系。即使没有对应，二者之间的转换也应该属于一趟指令选择就能完成的任务。RTL 就属于一种非常典型的低层次 IR。下图给出了一个完成相同功能的三种 IR 的例子（从左到右依次为 HIR、MIR 和 LIR）：

```
t1 = a[i][j+2]        t1 = j + 2            r1 = [fp - 4]
                      t2 = i * 20           r2 = r1 + 2
                      t3 = t1 + t2          r3 = [fp - 8]
                      t4 = 4 * t3           r4 = r3 * 20
                      t5 = addr a           r5 = r4 + r2
                      t6 = t5 + t4          r6 = 4 * r5
                      t7 = *t6              r7 = fp - 216
                                            f1 = [r7 + r6]
```

从表示方式来看，我们又可以将中间代码分成如下三类：

- 图形中间代码（Graphical IR）

  这种类型的中间代码将输入程序的信息嵌入到一张图中，以节点、边等元素来组织代码信息。由于要表示和处理一般的图代价稍大，人们经常会使用特殊的图，例如树或者有向无环图（DAG）来代替一般的图。

  一个典型的树形中间代码的例子就是抽象语法树（Abstract Syntax Tree, AST）。抽象语法树中省去了语法树里那些不必要的节点，将输入程序的语法信息以一种更加简洁的形式呈现出来，对于我们的处理来说是十分方便的。其它树形代码的例子有 GCC 里所使用的 GIMPLE。这种中间代码将各种操作都组织在一棵树里，在最后一次实习的指令选择部分我们会看到这种表示方式会简化其中的某些处理。

- 线形中间代码（Linear IR）

  线形结构的代码我们见得非常多，例如我们经常使用的 C 语言、Java 语言和汇编语言中语句和语句之间也就是一个线性关系。你可以将这种中间代码看成某种抽象计算机的一个简单的指令集。这种结构最大的优点是表示简单、处理高效，而缺点是代码和代码之间的先后关系有时会模糊整段程序的逻辑，让某些优化操作变得复杂。

- 混合型中间代码（Hybrid IR）

  顾名思义，混合型中间代码主要混合了图形和线形两种中间代码，期望结合这两种代码的优点、避免二者的缺点。例如，我们可以将中间代码组织成一个个基本块，块内部采用线形表示，块与块之间采用图表示，这样既可以简化块内部的数据流分析，又可以简化块与块之间的控制流分析。

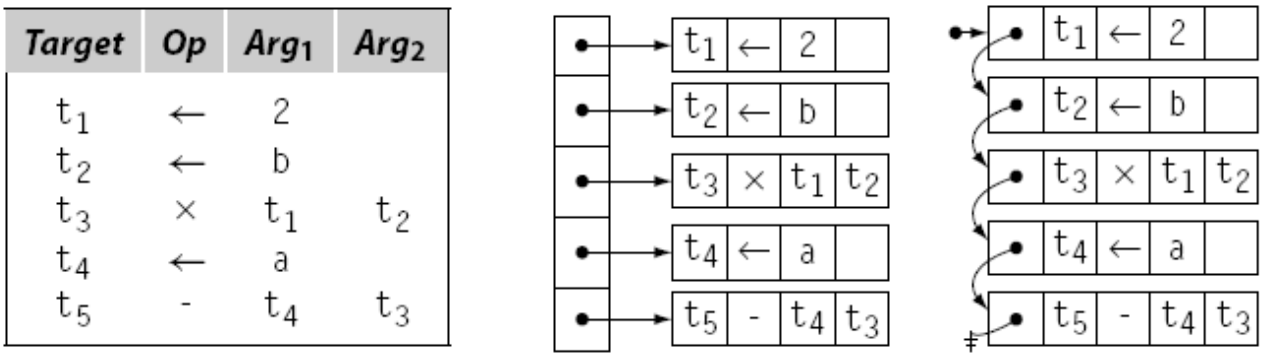
本次实习中，你被要求按照格式输出一系列已经设计好了的内容。虽说实习要求中规定的这个代码格式类似于线形的中层次中间代码，但不要据此就认为我们在限制你对于中间代码的选择——实习要求里仅仅是一个输出格式，而你的编译器只要能按要求输出规定的内容即可。至于程序内部采用何种形式的中间代码，而这些中间代码中又体现了多少细节信息，都完全取决于你自己的喜好。

中间代码的表示

刚拿到实习任务时，你可能会想要一边对语法树进行处理一边使用 fprintf()函数把要输出的代码内容打印出来。这种做法其实并不好，因为当你将代码内容打印出来的那一刻你就已经失去了对这些代码进行进一步调整、优化的机会。更加合理的做法应该是将所生成的中间代码先保存到内存中，等到全部翻译完毕、优化也都做完再使用一个专门的打印函数把内存中的中间代码打印出来。既然生成好的中间代码会放到内存里，如何保存这些代码、为其设计怎样的数据结构就是值得我们考虑的问题了。本节将对一种典型的线形 IR 和一种典型的树形 IR 的实现细节进行介绍，希望这些内容能够给你一些启发。

相对而言，使用线形 IR 是实现起来最简单，而且打印起来最方便的中间代码形式。由于代码本身是线形的，我

们可以使用几种最基本的线性数据结构来表示它们，如下图[1]所示：



其中图(a)中开了一个大的静态数组，数组中的每一个元素（图中的每一行）就是一条中间代码。使用静态数组的好处是写起来方便，但缺点是灵活性不足，中间代码的最大行数受限，而且代码的插入、删除以及调换位置的代价较大。图(b)中同样开了一个大数组，但数组中的每一个元素并不是一条中间代码，而是一个指向中间代码指针。虽然采用这种实现时代码行数也会受限，不过它和图(a)的实现相比大大减少了调换代码位置的开销。图(c)是一个纯链表的实现，图中画出来的链表是单向的但我们更建议使用双向循环链表。链表以增加实现复杂性为代价换得了极大的灵活性，可以进行高效地插入、删除、调换位置，并且几乎不存在代码最大行数的限制。

假设单条中间代码的数据结构定义为：

```
typedef struct Operand_* Operand;
struct Operand_ {
    enum { VARIABLE, CONSTANT, ADDRESS, … } kind;
    union {
        int var_no;
        int value;
        …
    } u;
};

struct InterCode
{
    enum { ASSIGN, ADD, SUB, MUL, … } kind;
    union {
        struct { Operand right, left; } assign;
        struct { Operand result, op1, op2; } binop;
        …
    } u;
}
```

[1] 图片来自 K. D. Cooper 和 L. Torczon 所著的 *Engineering a Compiler 2nd Edition*

那么上图(a)中的实现可以写成：

```
InterCode codes[MAX_LINE];
```

图(b)中的实现可以写成：

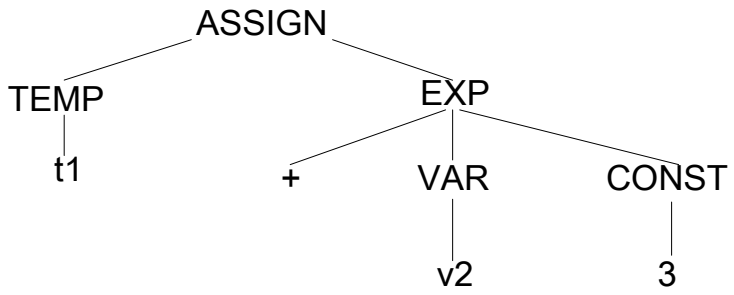
```
InterCode* codes[MAX_LINE];
```

图(c)中的（双向链表）实现可以写成：

```
struct InterCodes { InterCode code; struct InterCodes *prev, *next; };
```

想要打印出线形 IR 非常简单，只需从第一行代码开始逐行访问，根据每行代码 kind 域的不同按照不同的格式打印即可。对于数组，逐行访问其实就意味着一个 for 循环；对于链表，逐行访问则意味着以一个 while 循环顺着链表的 next 域进行迭代。

树形 IR 乍看上去可能让人感到复杂，但仔细想想就会发现其实它与线形代码一样直观。树形结构天然地具有层次的概念，在靠近树根的高层部分的中间代码其抽象层次较高，而在靠近树叶的低层部分的中间代码则更加具体。举个例子，一条中间代码 t1 := v2 + #3 可以用树形结构表示为：



我们也可以从另一个角度来理解树形 IR。在第一次实习中，我们曾经为输入程序构造过语法树或是抽象语法树，这棵语法树所对应的程序设计语言是编译器的源语言；而在这里，树形结构同样可以看作是一棵语法树或是抽象语法树，而它所对应的程序设计语言则是我们的中间代码。源语言的语法本身是相当复杂的，但本次实习要求我们输出的中间代码的语法规则却是简单的，因此我们也可以想象得到树形结构的中间代码的设计与实现也应该要比语法树更简洁一些。

之前我们已经做过有关语法树的实习，你对于树形结构如何实现应当是非常熟悉的。正如前面所说，树形 IR 可以看作一种基于中间代码的抽象语法树，因此其数据结构以及实现细节应该与语法树非常类似。有了写语法树的经验，写树形 IR 只需在原有基础上稍加修改即可，不会带来太多额外的困难。

树形 IR 的打印要比线形 IR 的打印复杂一些，该任务类似于给你一棵输入程序的语法树让你将输入程序打印出来。你需要对树形 IR 进行（深度优先）遍历，根据当前节点的节点类型递归地对当前节点的各个子节点进行打印。从另外一个角度看，从之前实习中构造的语法树到本次实习要求输出的中间代码之间总要经历一个由树形到线形的转换，使用线形 IR 其实就是将这步转换提前到构造 IR 时，而使用树形 IR 则是将这步转换推后到输出时才进行。

运行时刻环境初探

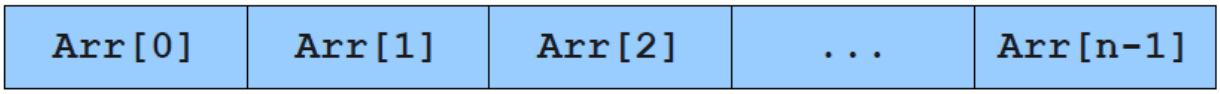
在程序员眼里，程序设计语言中可以有很多类型（基本类型、数组、结构体），有类和对象，有异常处理，有动态类型，有作用域的概念，有函数调用，有名空间，等等，而且每个程序似乎都有使不完的内存空间。但很显然，程序运行所基于的底层硬件决不会支持这么多的特性。一般来说，硬件只对 32bits/64bits 整数、IEEE 浮点数、简单的算术运算、数值拷贝，以及简单的跳转提供直接的支持，并且其存储器的大小也是有限的、结构也是分层的。程序设计语言里其余的特性则需要编译器、汇编/链接器、操作系统等共同作用，从而让程序员产生一种幻觉，认为他们眼里所看到的所有特性都是被底层硬件直接支持的。因此从某种意义上讲，编译器的主要任务就是帮助底层硬件向程序员诉说一个又一个美丽的谎言。

使用程序设计语言所书写出来的变量、类、函数等都是些抽象层次比较高的概念，为了使用硬件能够直接支持的底层操作表示这些高抽象层次概念，仅靠编译时刻字面上的代码翻译是远远不够的，我们需要能够生成额外的目标代码，使我们在程序的运行时刻可以维护起一系列的结构以支撑起程序设计语言中的各种高级特性，这些程序员一般不可见、但确实存在于运行时刻的结构就被称为运行时刻环境（runtime environment）。运行时刻环境与源语言、

目标语言和目标机器都紧密相关，由于其中包含的很多细节并不是一两句话就能够一次说清楚的，因此对于运行时刻环境这部分内容我们会分拆在两次实习任务中逐步细化。本次实习以介绍原理为主，以及一些简单结构（例如数组和结构体）的实现方式，下次实习中我们会更加细致地来考察 MIPS 体系结构下的寄存器规约以及调用栈的布置。另外，关于面向对象语言中如何布置运行时刻环境以及代码生成的问题将会作为补充材料放到实验课网站上，这些补充内容实习中不会用到，但确实很有意思，供有兴趣的同学阅览，拓展一下自己的知识面。

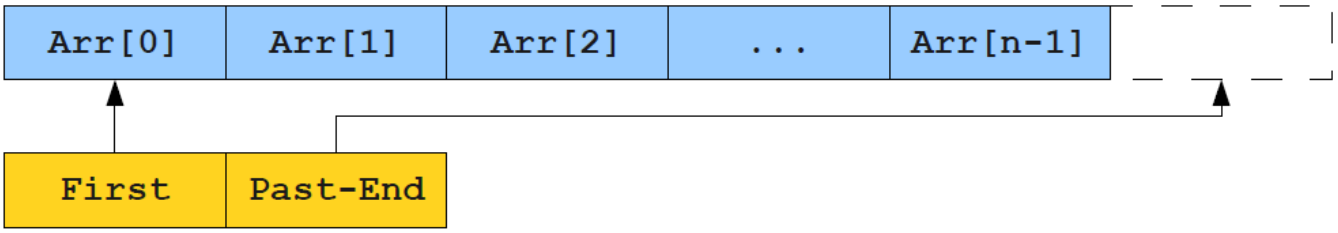
高级语言里的 char、short、int 等类型一般会直接对应到底层机器上的一个、两个或者四个字节，而 float 和 double 类型则会对应到底层机器上的四个、八个字节，这些类型都可以由硬件直接提供支持。底层硬件中没有指针类型，但指针可以用四字节（32bits 机器）或者八字节（64bits 机器）整数表示，其内容即为指针所指向的内存地址。

以上都是一些比较基本的类型，下面来考察一维数组的表示。我们最熟悉的表示数组的方式是 C 风格的，即数组中元素一个挨着另一个并占用一段连续的内存空间[2]：

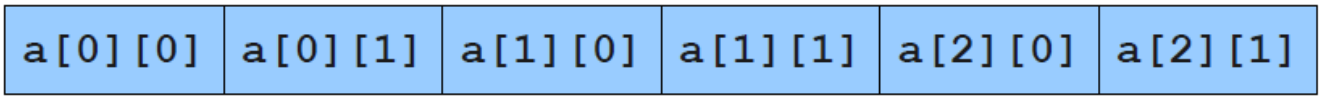


这当然不是唯一的表示方法。Java 在编译 bytecode 时就会采取另外一种布局，将数组长度放在起始位置（Pascal 中的 string 类型也是这样保存的）：

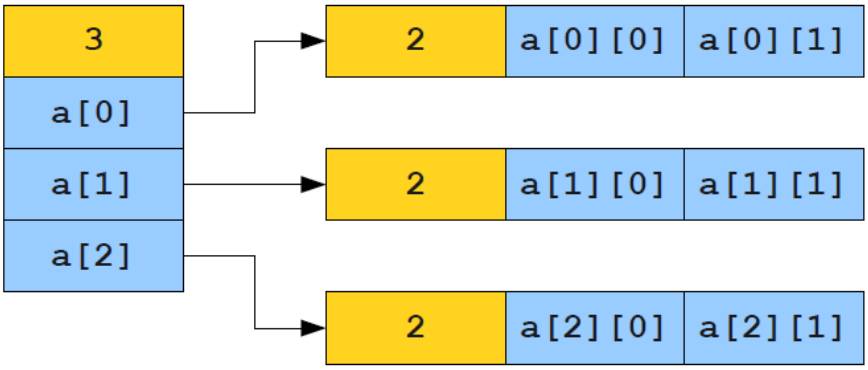
还有另外一种表示方式为 D 语言所采用：数组变量本身仅由两个指针组成，一个指向数组的开头，另一个指向数组的末尾之后，数组的所有信息存在于另外一段内存之中：



可以看到，无论是哪一种表示方式，数组元素在内存中总是连续存储的，这当然是为了使数组的访问能够更快（只需计算基地址+偏移量，然后取值即可）。多维数组可以看作一维数组的数组，C 风格的表示方法仍然是使用一段连续的内存空间：

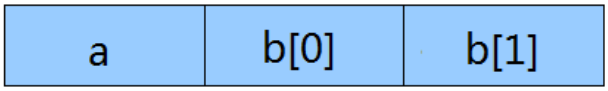


而 Java 中每一个一维数组是一个独立的对象，因此多维数组中的各维一般不会聚在一起：



结构体的表示和数组类似，最常见的办法是将各域按定义的顺序连续地放在一起。比如 struct { int a; float b[2];} 在内存中可以表示为：

[2] 这一小节里所有的图片都取自 Stanford 编译课件。



我们的实习里只包含 int 和 float 两种类型 。两种类型的宽度都是四字节，这省去了我们许多的麻烦。如果 C--语言允许其它宽度的类型存在又会如何呢？例如，struct { int a; char b; double c;} 这个结构体在内存中会排成如下这样吗：

答案是不会。如果没有特别指定，gcc 总会将结构体中的域对齐到字边界。因此在 char b 和 double c 之间会有 3 字节的空间被浪费掉，如下图所示：

x86 平台允许变量在存储时不经对齐，但 MIPS 则要求必须对齐，这是我们需要注意的一点 。

最后简单地来谈一谈函数的表示。众所周知，在调用函数时我们需要进行一系列的配套工作，包括找到函数的入口地址、参数传递、为局部变量申请空间、保存寄存器现场、返回值传递和控制流跳转等等。在这一过程中我们需要用到的各种信息都需要有地方能够保存下来，而保存这些信息的结构就被称为活动记录（activation record）。因为活动记录经常被保存在栈上，故它往往也被称为栈帧（stack frame）。活动记录的建立可以说是维护运行时刻环境的重点，编译器一般也会为它生成大段的额外代码——这意味着函数调用的开销一般会很大，所以对于那些功能简单的函数来说，内联展开往往是一种非常有效的优化方法。本次实习中我们并不需要关心活动记录是如何建立的，只需要压入相应的参数然后调用 CALL 语句即可。唯一要注意的是，若数组/结构体作为参数传递，需谨记数组和结构体都要求采用 call by reference 的参数传递方式，而非普通变量的 call by value。

活动记录一定要保存在堆栈上吗？对于这个问题的回答取决于程序设计语言本身的特性。下表是从 Berkeley 的编译课件中摘录下来的，它很好地说明了不同语言特性与活动记录布置之间的对应关系。限于篇幅，表中内容的具体含义这里就不详细解释了。

Problem	Solution
1) No recursion, no nesting, fixed-sized data with size known by compiler, first-class function values	Use inline expansion or static variables to hold return addresses, locals, etc.
2) #1 + recursion	Need stack
3) #2 + variable-sized unboxed data	Need to keep both stack pointer and frame pointer
4) #3 - first-class function values + nested functions, up-level addressing	Add static link (access link) or global display
5) #4 + function values with properly nested accesses: functions passed as parameters only	Static link, function values contain their link (global display doesn't work so well)
6) #5 + general closure (first-class functions returned from functions or stored in variables)	Store local variables and static links on heap
7) #6 + continuations	Put everything on heap

翻译模式——基本表达式

本次实习的任务说起来很简单：你只需要根据语法树产生出一段段的中间代码，然后将中间代码按照输出格式打印出来即可。中间代码如何表示以及如何打印我们都已经讨论过了，现在需要解决的问题是：如何将语法树变成中间代码呢？

最简单也是最常用的方式仍是遍历语法树中的每一个节点，当发现语法树中有特定的结构出现时，就产生出相应的中间代码。和语义分析一样，中间代码的生成需要借助于上次实习中我们已经提到过的工具——语法制导翻译（SDT）。具体到代码上，我们可以为每一个主要的语法单元 X 都设计相应的翻译函数 translate_X，对语法树的遍历过程也就是这些函数之间的互相调用的过程。每一种特定的语法结构都对应了固定模式的翻译“模板”，下面我们

将针对一些典型的语法树的结构的翻译“模板”进行说明，这些内容你也可以在教材上找到，理论课上应该也会讲。教材上介绍的翻译模式可能与本文中的有些许不同，但核心思想是一致的[3]。

我们先从语言最基本的结构——表达式开始：

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(**INT**) return [*place := #value*][4]
ID	variable = lookup(sym_table, **ID**) return [*place := variable.name*]
Exp_1 ASSIGNOP Exp_2[5] (Exp_1 → ID)	t1 = new_temp() variable = lookup(sym_table, **Exp_1.ID**) code1 = translate_Exp(**Exp_2**, sym_table, t1) code2 = [variable.name *:= t1*] +[6] [*place :=* variable.name] return code1 + code2
Exp_1 PLUS Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(**Exp_1**, sym_table, t1) code2 = translate_Exp(**Exp_2**, sym_table, t2) code3 = [*place := t1 + t2*] return code1 + code2 + code3
MINUS Exp_1	t1 = new_temp() code1 = translate_Exp(**Exp_1**, sym_table, t1) code2 = [*place := #0 - t1*] return code1 + code2
Exp_1 RELOP Exp_2	label1 = new_label() label2 = new_label() code0 = [*place := #0*] code1 = translate_Cond(**Exp**, label1, label2, sym_table) code2 = [*LABEL label1*] + [*place := #1*] return code0 + code1 + code2 + [*LABEL label2*]
NOT Exp_1	
Exp_1 AND Exp_2	
Exp_1 OR Exp_2	

上表列出了与表达式相关的一些结构的翻译模式。假设我们有函数 translate_Exp()，它接受三个参数：语法树的节点 Exp、符号表 sym_table 以及一个变量名 place，并返回一段语法树当前节点及其子孙节点对应的中间代码（或是一个指向存储中间代码内存区域的指针）。根据语法单元 Exp 所采用的产生式的不同，我们将生成不同的中间代码：

- 如果 Exp 产生了一个整数 INT，那么我们只需要为传入的 place 变量赋成前面加上一个‘#’ 的相应的数值即可。
- 如果 Exp 产生了一个标识符 ID，那么我们只需要为传入的 place 变量赋成 ID 对应的变量名（或者该变量对应到中间代码中的名字）即可。
- 如果 Exp 产生了赋值表达式 Exp_1 ASSIGNOP Exp_2，由于之前提到过作为左值的 Exp_1 只能是三种情况之一（单个变量访问、数组元素访问或者是结构体特定域的访问），而对于数组和结构体的翻译模式我们将放在后文讨论，故这里仅列出当 Exp_1 -> ID 时应该如何进行翻译。我们需要通过查表找到 ID 对应的变量，再对 Exp_2 进行翻译（运算结果储存在临时变量 t1 中），最后将 t1 中的值赋于 ID 所对应的变量并将结果再存回 place，然后把刚翻译好的这两段代码合并随后返回即可。
- 如果 Exp 产生了算数运算表达式 Exp_1 PLUS Exp_2，则先对 Exp_1 进行翻译（运算结果储存在临时变量 t1 中），再

[3]使用模板并不是唯一的生成中间代码的方法，也存在着其他方法（例如构造控制流图，Control Flow Graph）可以完成翻译任务，只不过使用模板是最简单和被介绍得最为广泛的方法。

[4] 用方括号括起来的内容代表新建一条具体的中间代码，下同。

[5] 这里 Exp 的下标只是用来区分产生式 Exp→Exp ASSIGNOP Exp 中多次重复出现的 Exp，下同。

[6] 这里的加号相当于连接运算，意为将两段代码连接成一段，下同。

对 Exp_2 进行翻译（运算结果储存在临时变量 t2 中），最后生成一句中间代码 place := t1 + t2，并将刚翻译好的这三段代码合并随后返回即可。使用类似的翻译模式我们也可以对减法、乘法和除法表达式进行翻译。

- 如果 Exp 产生了取负表达式 MINUS Exp_1，则先对 Exp_1 进行翻译（运算结果储存在临时变量 t1 中），再生成一句中间代码 place := #0 - t1 从而对 t1 取负，最后将翻译好的这两段代码合并并返回。使用类似的翻译模式我们也可以对括号表达式进行翻译。
- 如果 Exp 产生了条件表达式（包括与或非运算以及比较运算的表达式），我们会调用 translate_Cond()函数进行（短路）翻译。如果条件表达式为真，那么为 place 赋值 1；否则，为其赋值 0。由于条件表达式的翻译可能和跳转语句有关，上表中并没有明确 translate_Cond()应该如何实现，这一点我们将放在后文介绍。

翻译模式——语句

C--的语句包括表达式语句、复合语句、返回语句、跳转语句和循环语句，它们的翻译模式如下表所示：

translate_Stmt(Stmt, sym_table) = case Stmt of	
Exp SEMI	return translate_Exp(**Exp**, sym_table, NULL)
CompSt	return translate_CompSt(**CompSt**, sym_table)
RETURN Exp SEMI	t1 = new_temp() code1 = translate_Exp(**Exp**, sym_table, t1) code2 = [*RETURN t1*] return code1 + code2
IF LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() code1 = translate_Cond(**Exp**, label1, label2, sym_table) code2 = translate_Stmt(**$Stmt_1$**, sym_table) return code1 + [*LABEL label1*] + code2 + [*LABEL label2*]
IF LP Exp RP $Stmt_1$ ELSE $Stmt_2$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(**Exp**, label1, label2, sym_table) code2 = translate_Stmt(**$Stmt_1$**, sym_table) code3 = translate_Stmt(**$Stmt_2$**, sym_table) return code1 + [*LABEL label1*] + code2 + [*GOTO label3*] + [*LABEL label2*] + code3 + [*LABEL label3*]
WHILE LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(**Exp**, label2, label3, sym_table) code2 = translate_Stmt(**$Stmt_1$**, sym_table) return [*LABEL label1*] + code1 + [*LABEL label2*] + code2 + [*GOTO label1*] + [*LABEL label3*]

细心的你可能已经注意到了，无论是 if 语句还是 while 语句，上表中列出的翻译模式都不包含条件跳转。如果没有条件跳转，如何对 if 和 while 语句的条件表达式进行判断呢？其实我们可以在翻译条件表达式的同时生成这些条件跳转语句。translate_Cond()函数负责对条件表达式进行翻译，其翻译模式为：

translate_Cond(Exp, label_true, label_false, sym_table) = case Exp of	
Exp_1 RELOP Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(**Exp_1**, sym_table, t1) code2 = translate_Exp(**Exp_2**, sym_table, t2)

	op = get_relop(**RELOP**); code3 = [*IF t1 op t2 GOTO label_true*] return code1 + code2 + code3 + [*GOTO label_false*]
NOT Exp_1	return translate_Cond(**Exp_1**, label_false, label_true, sym_table)
Exp_1 AND Exp_2	label1 = new_label() code1 = translate_Cond(**Exp_1**, label1, label_false, sym_table) code2 = translate_Cond(**Exp_2**, label_true, label_false, sym_table) return code1 + [*LABEL label1*] + code2
Exp_1 OR Exp_2	label1 = new_label() code1 = translate_Cond(**Exp_1**, label_true, label1, sym_table) code2 = translate_Cond(**Exp_2**, label_true, label_false, sym_table) return code1 + [*LABEL label1*] + code2
(other cases)	t1 = new_temp() code1 = translate_Exp(**Exp**, sym_table, t1) code2 = [*IF t1 != #0 GOTO label_true*] return code1 + code2 + [*GOTO label_false*]

对于条件表达式的翻译，理论课上可能会花比较长的时间进行介绍，尤其是与回填有关的内容更是重点。不过，无论你如何仔细地阅读上表，你仍然找不到和回填相关的任何内容。原因其实非常简单：上表中我们将跳转的两个目标 label_true 和 label_false 作为继承属性（函数参数）进行处理，这种情况下每当我们在条件表达式内部需要跳转到外面时，跳转目标都已经从父节点那里通过传参数得到了，直接填上即可。所谓回填，只用于将 label_true 和 label_false 作为综合属性处理的情况，注意这两种处理方式的区别。

思考题 1：对于 if 和 while 语句，除上表列出的翻译模式之外有没有其他的模式？不同的翻译模式之间孰优孰劣？请使用 gcc 对 if 语句、while 语句和 do-while 语句生成 x86 或 MIPS 汇编代码，看看 gcc 对于代码的结构是如何组织的。

思考题 2：如果 C--语言允许 for 语句出现，那么 for 语句的语法和翻译模式应该是怎样的？

思考题 3：如果 C--语言允许 break 语句和 continue 语句出现，这两种语句的翻译模式应该是怎样的？

思考题 4：如果 C--语言允许 switch 语句出现，那么如何为 switch 语句设计翻译模式呢？将 switch 语句看成多条 if-then-else 语句的组合并按照 if 语句的翻译模式进行翻译当然是可以的，但有没有其他更高效的做法呢？

翻译模式——函数调用

函数调用是由语法单元 Exp 推导而来的，因此为了翻译函数调用表达式我们需要继续完善 translate_Exp()：

translate_Exp(Exp, sym_table, place) = case Exp of	
ID LP RP	function = lookup(sym_table, **ID**) if (function.name == "read") return [*READ place*] return [*place := CALL function.name*]
ID LP Args RP	function = lookup(sym_table, **ID**) arg_list = NULL code1 = translate_Args(**Args**, sym_table, arg_list) if (function.name == "write") return code1 + [*WRITE arg_list[1]*] for i = 1 to length(arg_list) code2 = code2 + [ARG arg_list[i]] return code1 + code2 + [*place := CALL function.name*]

由于实习要求中规定了两个需要特殊对待的函数 read()和 write()，故当我们从符号表里找到 ID 对应的函数名时不能直接生成 CALL，而是应该判断函数名是否是 read 或者 write。对于那些非 read()和 write()的带参数的函数而言，我们还需要 translate_Args()函数将计算实参的代码翻译出来，并构造这些参数所对应的临时变量列表 arg_list。translate_Args()的实现如下：

translate_Args(Args, sym_table, arg_list) = case Args of

Exp	t1 = new_temp() code1 = translate_Exp(**Exp**, sym_table, t1) arg_list = t1 + arg_list return code1
Exp COMMA Args$_1$	t1 = new_temp() code1 = translate_Exp(**Exp**, sym_table, t1) arg_list = t1 + arg_list code2 = translate_Args(**Args$_1$**, sym_table, arg_list) return code1 + code2

翻译模式——数组和结构体

C--语言的数组实现采取的是最简单的 C 风格。数组以及结构体不同于一般变量的一点在于，访问某个数组元素或者访问结构体的某个域需要牵扯到内存地址的运算。以三维数组为例，假设有数组 int array[7][8][9]，为了访问数组元素 array[3][4][5]，我们首先需要找到三维数组 array 的首地址（直接对变量 array 取地址即可），然后找到二维数组 array[3]的首地址（array 的地址加上 3 乘以二维数组的大小 8×9 再乘以 int 类型的宽度 4），然后找到一维数组 array[3][4]的首地址（array[3]的地址加上 4 乘以一维数组的大小 9 再乘以 int 类型的宽度 4），最后找到整数 array[3][4][5]的地址（array[3][4]的地址加上 5 乘以 int 类型的宽度 4）。整个运算过程可以表示为：

$$\mathrm{ADDR(array[i][j][k])} = \mathrm{ADDR(array)} + \sum_{t=0}^{i-1} \mathrm{SIZEOF(array[t])} + \sum_{t=0}^{j-1} \mathrm{SIZEOF(array[i][t])} + \sum_{t=0}^{k-1} \mathrm{SIZEOF(array[i][j][t])}$$

上式很容易推广到任意维数组的情况。

结构体的访问方式与数组非常类似。例如，假设要访问结构体 struct { int x[10]; int y,z; } st 中的域 z，我们首先找到变量 st 的首地址，然后找到 st 中域 z 的首地址（st 的地址加上数组 x 的大小 4×10 再加上整数 y 的大小 4）。我们可以把一个拥有 n 个域的结构体看成一个有 n 个元素的一维数组，它跟数组惟一的不同点就在于，数组的每一元素大小都是相同的，而结构体的每一个域大小可能不一样。其地址运算的过程可以表示为：

$$\mathrm{ADDR(st.field\_n)} = \mathrm{ADDR(st)} + \sum_{t=0}^{n-1} \mathrm{SIZEOF(st.field\_t)}$$

将数组的地址运算和结构体的地址运算结合起来也并不是什么难事。假如我们有一个结构体，该结构体的某个元素是数组。为了访问这个数组中的某个元素，我们需要先根据该数组在结构体中的位置定位到这个数组的首地址，然后再根据数组下标定位该元素。反之，如果我们有一个数组，该数组的每个元素都是结构体。为了访问某个数组元素的某个域，我们需要先根据数组下标定位要访问的结构体，再根据域的位置寻找要访问的内容。这个过程中惟一需要重点关注的是，我们应记录并区分在访问过程中使用到的临时变量哪些代表地址、哪些代表内存中的数值。如果弄错，有可能会导致代码的运行结果出错或者非法的内存访问（这些错误可不是那么好检查的）。当然，上述访问方式需要经历多次地址计算，而如若我们能通过其它手段将这多次地址计算合并成一次，则得到的中间代码的效率就会得到一定的提高。

数组和结构体的翻译模式，以及其它语法单元的翻译模式留给大家自己思考。

思考题 1：ANSI C 标准里，数组各维度的大小必须是一个常数或者常数表达式。程序员如果想要使用变量大小的数组必须借助于 malloc()等方式完成，非常麻烦。在 ISO C99 标准里，数组大小终于可以是一个变量了，而且 gcc 也早已经对此提供了支持。如何让我们的 C--也能使用变量确定数组大小？在中间代码的生成与优化上与之前的静态数组又会有什么区别呢？

思考题 2：C 语言中的联合体（union）机制想必大家都非常熟悉，在之前的语法实习中你甚至还可能会大量地使用到它。union 为我们提供了一种既可以节省空间又能绕过 C 语言类型机制的有效手段。如何为我们的 C--语言也添加 union 机制？你认为添加 union 机制与添加 struct 机制相比哪一个难度更大一些？

如何完成本次实习

本次实习需要你在上一次实习的基础上完成。你可以在上次实习语义分析部分添加中间代码生成的内容，使编译器可以一边进行语义检查一边生成中间代码；也可以将中间代码生成的所有内容写到一个单独的文件里，等到语义检查全部完成并通过之后再生成中间代码。前者会让你的编译器效率高一些，后者会让你的编译器模块性更好一些。

确定了在哪里进行中间代码生成以后，下一步就要动手实现中间代码的数据结构（最好能写一系列可以直接生成一条中间代码的“构造函数”以简化后面的实现），然后再按照输出格式的要求自己编写函数将你的中间代码打印出来。完成以后建议先自行写一个测试程序，在这个测试程序中使用构造函数人工构造一段代码并将其打印出来，然后使用虚拟机小程序简单地测试一下，一方面确保自己的数据结构和打印函数都能正常工作，另一方面也顺带检查一下虚拟机程序是否存在 bug。准备工作完成后，再继续做下面的内容。

接下来的任务是根据前文所给出的翻译模式完成一系列 translate 函数。本文中已经给出了 Exp 和 Stmt 应该如何翻译，你还需要自行考虑包括数组（多维数组选做）、结构体（选做）、数组与结构体定义、变量初始化、语法单元 CompSt、语法单元 StmtList 在内的翻译模式。需要你自己考虑的内容实际上并不多，但最关键的一点在于一定要读懂上面的几个 translate 函数的意思——而这件事情其实要比你想象中的更困难一些。如果读懂了，那么无论是将它们实现到你的编译器中还是书写新的 translate 函数以及对这些 translate 函数进行改进都将是触类旁通的。反之如果没读懂，例如没想明白某些 translate 函数中出现的 place 参数究竟有什么用，那么我建议你还是先别急着动手写代码，静下心来思考才是上策。如果顺利完成了所有 translate 函数，并将其连同中间代码的打印函数添加到你的编译器中，那么本次实习的基本要求就差不多完成了。建议在这里多写几份测试数据测一下你的编译器，若发现了错误则需及时纠正。

最后，虚拟机将以总共执行过的中间代码条数为标准来衡量你的编译器所输出的代码的运行效率，因此如果要进行代码优化，重点应该在精简代码逻辑以及消除代码冗余上做文章，我们设计的测试数据也会着重考察这两点。由于课程安排的关系，理论课的进度应该还没讲到代码优化，因此如果想拿到这两个选做内容的分数你可能要提前自行阅读教材中那些比较简单的机器无关代码优化的机制。另外，别忘了这两个选做内容是竞争性的——你要做的不是超越某个固定的要求，而是超越你的同班同学，同时你的同学也可能在为了拿到这些分数而想方设法超越你。

最后，本次实习的过程中你会使用到陈嘉编写的虚拟机小程序。这个虚拟机程序有五百行左右的 Python 代码，其中肯定有一些 bug 在实习公布之时仍没有被发现。为了避免这些 bug 给你带来更多的麻烦，首先请不要蓄意编写奇怪的代码试探虚拟机程序的鲁棒性，其次当你发现该程序存在任何问题时请及时发邮件给我们。

祝你好运！

> The machine does not isolate man from the great problems of nature but plunges him more deeply into them.
>
> —— Antoine de Saint-Exupery