4.2 实验指导

编译器里最核心的数据结构之一就是**中间代码（Intermediate Representation或IR）**。中间代码应包含哪些信息，这些信息又应有怎样的内部表示？这些问题会极大地影响编译器代码的复杂程度、编译器的运行效率、以及编译生成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数据结构都被称为中间代码（或可更精确地译为“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，都可视为中间代码。使用中间代码的主要原因是为了方便编写编译器程序的各种操作。如果我们在需要有关输入程序的任何信息时都只能去重新读入并处理输入程序源代码的话，编译器的编写将会变得非常麻烦，同时也会大大降低其运行效率。

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）。你可能会有疑问：为什么编译器不能把输入程序直接翻译成目标代码，而是要额外引入中间代码呢？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为前端和后端两个部分。当我们需要改变编译器的源语言或目标语言时，如果采用了中间代码，我们只需要替换原编译器的前端或后端，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那些复杂但相关性不大的任务分别放在前端和后端的各个模块中，这既简化了模块内部的处理，又使我们能单独对每个模块进行调试与修改而不影响其它模块。下文中，如果不特别说明，“中间代码”都是指狭义的中间代码。

4.2.1 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。不同编译器所使用的中间代码可能是千差万别的，即使是同一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接近，有的中间代码与目标语言更接近。编译器需要在不同的中间代码之间进行转换，有时为了处理的方便，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，然后将该树转换为另一种被称为GIMPLE的树形结构。在GIMPLE之上它建立静态单赋值式的中间代码之后，又会将其转换为一种非常底层的RTL（Register Transfer Language）代码，最后才把RTL转换为汇编代码。

```
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]
```

图4. 三种不同层次的中间代码示例。

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。从中间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：

1) **高层次中间代码（High-level IR或HIR）**：这种中间代码体现了较高层次的细节信息，因此往往和高级语言类似，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。我们所熟悉的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。

2) **中层次中间代码（Medium-level IR或MIR）**：这个层次的中间代码在形式上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为低级语言的代码。正是由于MIR的这个特性，它是三种IR中最难设计的一种。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操作。另外，对中层次中间代码可以进行绝大部分的优化处理，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。

3) **低层次中间代码（Low-level IR或LIR）**：低层次中间代码与目标语言非常接近，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，二者之间的转换也属于一趟指令选择就能完成的任务。前面提到的RTL就属于一种非常典型的低层次IR。

图4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）。

从表示方式来看，我们又可以将中间代码分成如下三类：

1) **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，以结点和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会很大，人们经常会使用特殊的图，例如树或**有向无环图（DAG）**。一个典型的树形中间代码的例子就是**抽象**

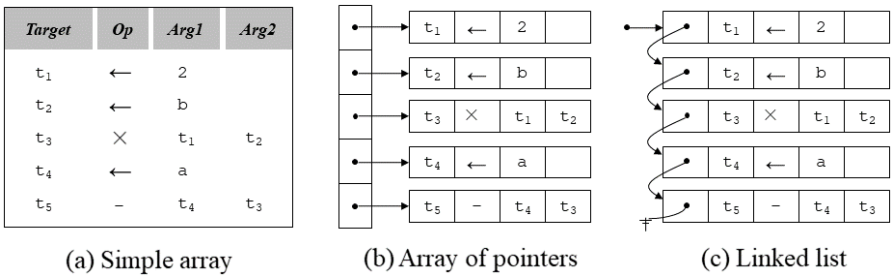

图5. 表示线性IR的三种基本数据结构。

语法树（Abstract Syntax Tree或AST）。抽象语法树中省去了语法树里不必要的结点，将输入程序的语法信息以一种更加简洁的形式呈现出来。其它树形中间代码的例子有GCC中所使用的GIMPLE。这类中间代码将各种操作都组织在一棵树中，在后面的实验四的指令选择部分我们会看到这种表示方式可以简化其中的某些处理。

2) **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就是线性关系。你可以将这种中间代码看成是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，而缺点就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很复杂。

3) **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并避免二者的缺点。例如，我们可以将中间代码组织成许多基本块，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。

在实验三中，你需要按照格式输出中间代码。虽然实验要求中规定的中间代码格式类似于线形的中层次中间代码，但这只是输出格式，而你的程序内部可以采用任何形式的中间代码，而这些中间代码中又体现了多少细节信息，则完全取决于你自己的设计。

4.2.2 中间代码的表示（线形）

在实验三中，你可能会一边对语法树进行处理一边把要输出的代码内容打印出来。这种做法其实并不好，因为当代码内容被打印出来的那一刻起，我们就已经失去了对这些代码进行调整和优化的机会。更加合理的做法是将所生成的中间代码先保存到内存中，等全部翻译完毕，优化也都做完后再使用一个专门的打印函数把在内存中的中间代码打印出来。既然生成好的中间代码会被放到内存中，那么如何保存这些代码以及为其设计怎样的数据结构就是值得考虑的问题了。我们下面对一种典型的线形IR的实现细节进行介绍，关于树形IR的实现细节我们将放到下一节介绍。

相对而言，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形数据结构来表示它们，如图5[1]所示。

其中图5(a)为一个大的静态数组，数组中的每个元素（图中的一行）就是一条中间代码。使用静态数组的好处是写起来编程方便，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5(b)同样为一个大数组，但数组中的每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5(a)的实现相比则大大减少了调换代码位置的开销。图5(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。链表以增加实现的复杂性为代价换得了极大的灵活性，可以进行高效的插入、删除以及调换位置操作，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：

```
 1  typedef struct Operand_* Operand;
 2  struct Operand_ {
 3    enum { VARIABLE, CONSTANT, ADDRESS, … } kind;
 4    union {
 5      int var_no;
 6      int value;
 7      …
 8    } u;
 9  };
10
11  struct InterCode
12  {
13    enum { ASSIGN, ADD, SUB, MUL, … } kind;
14    union {
15      struct { Operand right, left; } assign;
16      struct { Operand result, op1, op2; } binop;
17      …
18    } u;
19  }
```

那么，图5(a)中的实现可以写成：

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

图5(b)中的实现可以写成：

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

图5(c)中的（双向链表）实现可以写成：

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

要想打印出线形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域的不同值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

[1] 图片来源于：《Engineering a Compiler》，第2版，Keith D. Cooper和Linda Torczon著，Morgan Kaufmann出版社，第239页，2011年版。

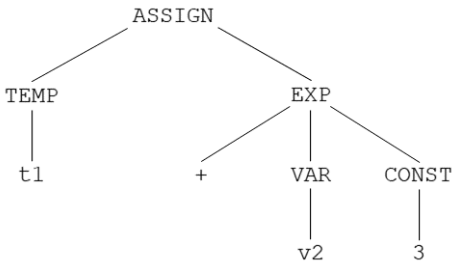

图6. 中间代码t1 := v2 + #3的树形结构表示。

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。

4.2.3 中间代码的表示（树形）

树形IR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IR一样直观。树形结构天然具有层次的概念，在靠近树根的高层部分的中间代码其抽象层次较高，而靠近树叶的低层部分的中间代码则更加具体。例如，中间代码t1 := v2 + #3可用树形结构表示为如图6所示。

我们也可以从另一个角度来理解树形IR。在实验一中，我们曾为输入程序构造过语法树，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，它所对应的程序设计语言则是我们的中间代码。源语言的语法相当复杂，但实验三要求我们输出的中间代码的语法规则却是简单的，因此树形结构的中间代码的设计与实现也会比语法树更简单。

之前我们已经做过有关语法树的实验，你对于树形结构该如何实现应当非常熟悉。正如前面所述，树形IR可以看作是一种基于中间代码的语法树（或抽象语法树），因此其数据结构以及实现细节与语法树非常类似。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，不会带来太多困难。

树形IR的打印要比线形IR复杂一些，该任务类似于给定一棵输入程序的语法树需要将该输入程序打印出来。你需要对树形IR进行（深度优先）遍历，根据当前结点的类型递归地对其各个子结点进行打印。从另外一个角度看，从之前实验中构造的语法树到实验三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。

4.2.4 运行时环境初探

在一个程序员眼里，程序设计语言中可以有很多机制，包括类型（基本类型、数组和结构体），类和对象，异常处理，动态类型，作用域，函数调用，名空间等等，而且每个程序似乎

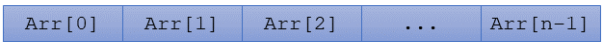

图7. C语言中一维数组的内存表示方式。

图8. Java语言中一维数组的内存表示方式。

都有使用不完的内存空间。但很显然，程序运行所基于的底层硬件不能支持这么多机制。一般来说，硬件只对32bits或64bits位整数、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。程序设计语言中的其它机制则需要编译器、汇编/链接器、操作系统等共同努力，从而让程序员们产生一种幻觉，认为他们眼中所看到的所有机制都是被底层硬件直接支持的。

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，为了能使用硬件直接支持的底层操作来表示这些高抽象层次的概念，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，使程序在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、但又确实存在于运行时刻的结构就被称为**运行时环境（Runtime Environment）**。运行时环境与源语言、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说明清楚，因此对于运行时环境这部分内容我们会分拆到实验三和实验四中逐步细化。在实验三中，我们以介绍原理为主，附带介绍一些简单结构（例如数组和结构体）的实现方式。在后面的实验四中，我们会更加细致地考察MIPS体系结构下的寄存器规约以及调用栈的布置。

高级语言中的char、short和int等类型一般会直接对应到底层机器上的一个、两个或四个字节，而float和double类型则会对应到底层机器上的四个和八个字节，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，但指针可以用四个字节（32bits位机器）或者八个字节（64bits位机器）整数表示，其内容即为指针所指向的内存地址。

以上是一些比较基本的类型，下面我们来考察一维数组的表示。最熟悉的表示数组的方式是C风格的（如图7所示），即数组中的元素一个挨着一个并占用一段连续的内存空间[1]。

[1] 这节中的图片均改自Stanford大学的编译原理课件：
http://www.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/11/Slides11.pdf。

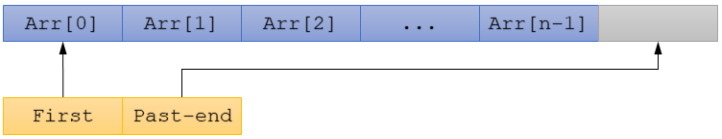

图9. D语言中一维数组的内存表示方式。

图10. C语言中多维数组的内存表示方式。

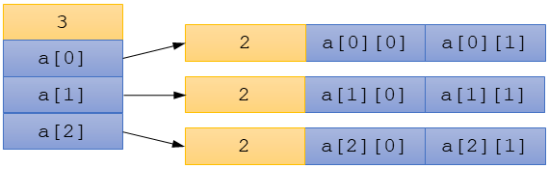

图11. Java语言中多维数组的内存表示方式。

图12. 结构体的内存表示示例。

当然这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类型数据也是这样保存的），如图8所示。

另外还有一种表示方式为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，如图9所示。

可以看出，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，如图10所示。

而Java中每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，如图11所示。

结构体的表示与数组类似，最常见的办法是将各个域按定义的顺序连续地存放在一起。比如struct { int a; float b[2]; }在内存中的表示如图12所示。

我们的实验三中只包含int和float两种类型，而这两种类型的宽度都是四字节，这省去了

图13. C--语言中结构体的内存表示的错误示例。

图14. C--语言中结构体的内存表示的正确示例。

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图13所示的表示方式吗？

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，如图14所示。

x86平台允许变量在存储时不对齐，但MIPS平台则要求对齐，这是我们需要注意的。

最后我们简单讨论一下函数的表示。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必须有地方能够保存下来，而保存这些信息的结构就称为**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。所以对于功能简单的函数来说，内联展开往往是一种有效的优化方法。在实验三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。但要注意的是，若数组或结构体作为参数传递，需谨记数组和结构体都要求采用**引用调用（Call by Reference）**的参数传递方式，而非普通变量的**值调用（Call by Value）**。

4.2.5 翻译模式（基本表达式）

实验三的任务比较简单，你只需根据语法树产生出中间代码，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？

最简单也是最常用的方式仍是遍历语法树中的每一个结点，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实验二中我们已经提到的工具：语法制导翻译（SDT）。具体到代码上，我们可以为每个主要的语法单元

表2. 基本表达式的翻译模式。

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(INT) return [place := #value][2]
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp_1 ASSIGNOP Exp_2[3] (Exp_1 → ID)	variable = lookup(sym_table, Exp_1.ID) t1 = new_temp() code1 = translate_Exp(Exp_2, sym_table, t1) code2 = [variable.name := t1] +[4] [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	

“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互相调用的过程。每种特定的语法结构都对应了固定模式的翻译“模板”，下面我们针对一些典型的语法树结构的翻译“模板”进行说明。这些内容你也可以在课本上找到，课本上介绍的翻译模式与下面我们介绍的可能略有不同，但核心思想是一致的[1]。

我们先从语言最基本的结构表达式开始。

表2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：语法树的结点Exp、符号表sym_table以及一个变量名place，并返回一段语法树当前结点及其子孙结点对应的中间代码（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：

1) 如果Exp产生了一个整数INT，那么我们只需要为传入的place变量赋值成前面加上一

[1] 使用模板并不是生成中间代码的唯一方法，也存在着其他方法（例如构造控制流图）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。

[2] 用方括号括起来的内容表示新建一条具体的中间代码。

[3] 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。

[4] 这里的加号相当于连接运算，表示将两段代码连接成一段。

表3. 语句的翻译模式。

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]

个“#”的相应数值即可。

2) 如果Exp产生了一个标识符ID，那么我们只需要为传入的place变量赋值成ID对应的变量名（或该变量对应的中间代码中的名字）即可。

3) 如果Exp产生了赋值表达式Exp1 ASSIGNOP Exp2，由于之前提到过作为左值的Exp1只能是三种情况之一（单个变量访问、数组元素访问或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在后面讨论，故这里仅列出当Exp1 → 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，然后对Exp2进行翻译（运算结果储存在临时变量t1中），再将t1中的值赋于ID所对应的变量并将结果再存回place，最后把刚翻译好的这两段代码合并随后返回即可。

4) 如果Exp产生了算术运算表达式Exp1 PLUS Exp2，则先对Exp1进行翻译（运算结果储存在临时变量t1中），再对Exp2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。

表4. 条件表达式的翻译模式。

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]

5) 如果Exp产生了取负表达式MINUS Exp1，则先对Exp1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从而实现对t1取负，最后将翻译好的这两段代码合并后返回。使用类似的翻译模式我们也可以对括号表达式进行翻译。

6) 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），我们则会调用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与跳转语句有关，表中并没有明确说明translate_Cond该如何实现，这一点我们在后面介绍。

4.2.6 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，它们的翻译模式如表3所示。

你可能注意到，无论是if语句还是while语句，表4中列出的翻译模式都不包含条件跳转。其实我们是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，其翻译模式如表4所示。

对于条件表达式的翻译，课本上已经花了较大的篇幅进行介绍，尤其是与回填有关的内容更是重点。不过，表4中没有与回填相关的任何内容。原因很简单：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，在这种情况下每当我们在条件

表5. 函数调用的翻译模式。

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]] + [place := #0] for i = 1 to length(arg_list) code2 = code2 + [ARG arg_list[i]] return code1 + code2 + [place := CALL function.name]

表6. 函数参数的翻译模式。

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

表达式内部需要跳转到外面时，跳转目标都已经从父结点那里通过参数得到了，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。

4.2.7 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，如表5所示。

由于实验要求中规定了两个需要特殊对待的函数read和write，故当我们从符号表中找到ID对应的函数名时不能直接生成函数调用代码，而是应该先判断函数名是否为read或write。对于那些非read和write的带参数的函数而言，我们还需要调用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的实现如表6所示。

4.2.8 翻译模式（数组与结构体）

C--语言的数组实现采取的是最简单的C风格。数组和结构体不同于一般变量的一点在于，访问某个数组元素或结构体的某个域需要牵扯到其内存地址的运算。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：

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

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的宽度4）。我们可以把一个有n个域的结构体看成为一个有n个元素的“一维数组”，它与一般一维数组的不同点在于，一般一维数组的每个元素的大小都是相同的，而结构体的每个域大小可能不一样。其地址运算的过程可以表示为：

$\text{ADDR}(\text{st}.field_n) = \text{ADDR}(\text{st}) + \sum_{t=0}^{n-1} \text{SIZEOF}(\text{st}.field_t)$。

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太难的事。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，然后再根据数组的下标定位该元素。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的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。这个过程中唯一需要关注的是，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，会导致代码的运行结果出错或者非法的内存访问。当然，上述访问方式需要经历多次地址计算，如果我们能通过其它手段将这多次地址计算合并成一次，那么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我们留给大家思考。

4.2.9 中间代码生成提示

实验三需要你在实验二的基础上完成。你可以在实验二的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；也可以将关于中间代码生成的所有内容写到一个单独的文件中，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之后，下一步就要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。完成之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，然后使用我们提供的虚拟机小程序简单地测试一下，以确保自己的数据结构和打印函数都能正常工作。准备工作完成之后，再继续做下面的内容。

接下来的任务是根据前面介绍的翻译模式完成一系列的translate函数。我们已经给出了Exp和Stmt的翻译模式，你还需要考虑包括数组、结构体、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太多，最关键的一点在于一定要读懂前面介绍的几个translate函数的意思。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，或者是对这些translate函数进行改进，都将是比较容易的。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，那么建议你还是先别急着动手写代码。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了你的编译器中，那么实验三的要求就差不多完成了。建议在这里多写几份测试用例检查你的编译器，如果发现了错误需要及时纠正。

最后，虚拟机小程序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中间代码的运行效率。因此如果要进行代码优化，重点应该放在精简代码逻辑以及消除代码冗余上。