第 4 章 中间代码生成

引言故事：

中间代码生成是编译器工作过程中的一个关键阶段，其主要任务是将源代码转换为中间表示形式，以便后续进行机器无关的优化和目标代码生成。结合我国传统文化，我们可以用文言文翻译来类比这个过程。编译器工作过程中的中间代码生成和文言文翻译都需要将某种源语言转化为目标语言，因此二者存在一定的相似性。

在中间代码生成的过程中，源代码作为源语言，是一种高级语言，而中间代码则是目标语言，同时也是一种中级语言。在编译过程中，中间代码是对源代码进行词法分析、语法分析与语义分析等操作之后的中间产物，以便后续阶段进行目标代码生成。与之相似，将文言文翻译为现代汉语，也是一个从源语言到目标语言的翻译过程。文言文翻译也需要对原文进行语法分析、语义分析、翻译校对和修改等操作，以便将其转化为现代汉语或其他目标语言，并且符合正确的表达习惯。另外，翻译的结果只是作为一个中间产物，如果我们想要更好的理解这段文字中蕴涵的丰富感情、表达意图以及题材类别，我们还需要将其凝练成为最终的相关词汇。这相当于编译过程后期使用中间代码生成目标代码的过程。例如《诗经·秦风·无衣》中的“岂曰无衣，与子同裳”翻译为现代汉语是“谁说我没有衣服可穿？与你共穿那战裙”，而更深层次的意思是借助共同穿战裙这件事情表达共御外敌的英雄主义精神。 再如诗仙李白的千古名句“床前明月光，疑是地上霜”翻译为现代汉语是“床前明亮的月光， 让人误以为地上已经结霜了”。作者借助“结霜”这个比喻，既形容了月光的皎洁，又表达了季节的寒冷，还烘托出诗人漂泊他乡的孤寂凄凉之情。

除此之外，中间代码生成的过程还可以类比成中国传统文化中的书法创作过程。在书法创作的过程中，书法家需要掌握一定的技法和规则，例如笔画线条的粗细、笔画的顺序、毛笔的不同使用技法等，这些技法和规则是书法家可以借鉴和运用的经验与工具。同样地，编译器需要运用一系列的算法和规则，例如词法分析、语法分析和语义分析等，这些算法和规则是编译器生成中间代码时可以借鉴和运用的　　。类比于书法创作，这个过程则类似于草书的写作，首先书法家需要有一份清晰、准确的草稿，而中间代码生成也需要先有一份正确、完整的源代码；在草书写作过程中，书法家需要对每个字的形状、笔画顺序和排列位置有清晰的认识和把握，才能将草书写得漂亮、流畅。

总之，中间代码是高级语言代码的翻译结果，但其不依赖于任何特定的目标机器，而是依赖于编译器本身的语义规则。编译器的中间代码生成是将源代码转换为目标代码的重要阶段之一。在此阶段，编译器将源代码翻译成中间代码表示形式，以供后续优化和代码生成阶段使用。也就是说，中间代码生成使得编译器的优化更容易，提高了编译器的灵活性，具有承上启下的作用。

本章要点：

中间代码生成是编译器工作流程中的核心阶段之一，其核心目标是将源代码翻译成 对应的中间代码，以便实现后续的机器无关优化和目标代码生成。为了形成完整的中间代码生成步骤，本章将首先介绍编译器中的运行时环境、存储组织，以及栈帧设计方法。然后，从中 间代码的层次和表示方式两方面分别介绍中间代码的表示形式。此外，本章还将讨论类型和 声明的相关概念（例如，类型表达式、类型等价、局部变量名的存储布局、类型声明和类型 记录等）。在表达式翻译部分，本章将重点讨论赋值语句、增量翻译和数组引用的翻译；在 控制流与回填部分，本章将着重介绍布尔表达式和控制流语句的翻译，以及布尔表达式和控 制流语句的回填理论方法。

此外，本章也将讨论中间代码生成理论方法所对应的技术实践内容，在所提供的技术指导下，编写一个程序为基于 C-- 语言的源代码生成中间代码，并打印出相应的中间代码表示结果。该实践任务要求使用 GNU Flex、GNU Bison，以及 C 语言来实现中间代码的生成，结合本章前面部分介绍的理论方法，编写一个中间代码的生成程序将是一件轻松愉快的事情。

思维导图(理论方法部分):


第四章
中间代码生成
运行时环境
目标代码的运行环境
为程序设计语言的高级特性提供支持
存储组织
编译器为目标代码分配和管理的逻辑地址空间
分配方式
静态存储分配 — 通过目标代码的文本确定内存分配策略
动态存储分配 — 通过运行目标代码确定内存分配策略
栈帧
活动记录的保存区域
栈指针与帧指针
寄存器
中间代码的表示形式
基于层次的分类
高层次中间代码 — 保留了较多语言特性
中层次中间代码 — 介于源语言和目标语言之间
低层次中间代码 — 与目标语言非常接近
基于表示方法的分类
线形中间代码
简单指令集
三地址代码
图形中间代码
用结点和边组织代码信息
抽象语法树
混合形式 — 同时结合线形和图形结构
类型与声明
类型表达式
基本类型
基于类型构造符的新类型
类型等价
结构等价
名等价
局部变量名的存储分布
基本类型
数组类型
类型声明
类型记录
表达式的翻译
赋值语句的翻译 — 为表达式求值生成相应的三地址码
增量翻译 — code属性为长字符时的翻译
数组引用的翻译 — 数组元素的寻址
控制流与回填
布尔表达式的翻译 — 改变语句中控制流的条件表达式
控制流语句的翻译
if语句的翻译
if-else语句的翻译
while语句的翻译
布尔表达式的回填 — 回填技术 — 为跳转指令填充目标标号
控制流语句的回填
if语句的回填
if-else语句的回填
while语句的回填

4.1 中间代码生成的理论方法

4.1.1 运行时环境概要

程序设计语言中通常包括基本类型、数组和结构体、类和对象、函数调用以及作用域等抽象概念。然而，程序运行所基于的底层硬件只能支持有限的抽象概念。现代计算机的底层硬件，只对32位或64位的整数、IEEE浮点数、基本算术运算、数值复制，以及简单的跳 转提供直接支持。为了准确地表示和实现这些抽象层次的概念，编译器创建并管理一个**运行时环境（Runtime Environment）**。通过运行时环境编译出的目标代码运行在该环境中。该环境可以为程序设计语言中的各种高级特性提供支持。例如，为在源代码中命名的对象分配和 安排存储位置，确定目标代码访问变量时所使用的机制、参数传递机制、系统调用、输入 / 输出设备与其他代码之间的接口等。我们将在后续的实践技术部分中详细介绍运行时环境如何表示基本类型、数组和结构体以及函数调用这三种抽象概念。

4.1.2 存储组织与栈帧设计方法

1. 存储组织

为了保障目标代码的运行，编译器通常会为目标代码分配和管理一个逻辑地址空间。图4. 1[1]展示了运行时存储空间划分的一个示例。一段目标代码在逻辑地址空间中运行时，其存储空间由低到高依次为：代码区、静态区、堆区、空闲内存和栈区。

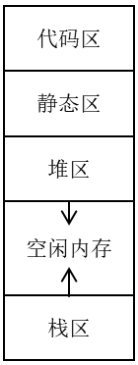

图 4. 1 运行时存储空间划分的一个示例（上方是低地址内存区，下方是高地址内存区）

[1] 图片来源于：《编译原理》，Alfred V. Aho 等著，赵建华、郑滔和戴新宇译，机械工业出版社，第 275 页，2009 年。

其中，代码区主要用于存储可运行的目标代码，通常位于逻辑地址低端的空间。由于可运行的目标代码的大小在编译时已经确定，因此代码区是一个可被静态确定的区域。同时，静态区还存储目标代码的数据对象，例如，全局变量和编译器生成的数据等。数据对象通常由静态分配，这是因为数据对象的地址可以被编译到目标代码中。堆区和栈区位于剩余 的逻辑地址空间中，这两个区域的大小会根据目标代码的运行而改变。当编译器无须运行目标代码，仅通过目标代码的文本即可确定内存分配时，该方式称为**静态存储分配**。当编译器必须通过运行目标代码才能确定内存分配策略时，该方式称为**动态存储分配**。编译器一般通过组合栈式存储和堆式存储两种策略来实现动态存储分配。栈区通常是从高地址位向低地址位增长的一块连续内存区域，主要存放函数的参数值、局部变量、返回地址信息，以及调用信息等数据；这些数据信息通常将在函数调用结束时被自动清理释放。而堆区通常处于静态区及栈区中间的区域，目前的常见实现是通过链表索引存储空闲地址；堆区主要用于动态分配内存，与栈区相比，堆区上的内存空间需要程序员手动申请，并且不会在函数调用结束时被自动销毁；在C/C++等语言中，堆上已分配的内存空间需要程序员手动释放，而在 Java/ Python 等语言中，则是通过垃圾回收器的运行完成释放。此外，运行时环境中数据位置的布局和分配也是存储组织的关键问题。

图 4. 2 栈帧结构（上方是低地址内存区，下方是高地址内存区）

2. 栈帧

程序在进行调用函数时需要进行一系列的配套工作，例如，找到函数的入口地址、传递参数、为局部变量申请空间、保存寄存器现场、以及实现返回值传递和控制流跳转等。栈中用于保存上

述信息的区域称为该函数的**活动记录（Activation Record）**。由于活动记录经常被保存在栈上，因此活动记录又称为**栈帧（Stack Frame）**。

图4.2展示了编译器中一种常见的栈帧结构。该栈帧中局部变量部分存放在栈帧中，部分保存在寄存器中。在当前函数调用其他函数时，可在传出参数空间传递参数。在允许声明嵌套函数的语言中，内层函数可以使用外层函数声明的变量。当调用函数*f*时，*f*的活动记录中将保存一个指针，指向程序结构上包含*f*的直接外层函数的活动记录，该指针被称为**静态链（Static Link）**。静态链的存在可以用于访问函数*f*中用到的非局部变量信息。而**动态链（Dynamic Link）**是指向调用者活动记录指针，也称为**控制链**，表示了函数在运行时刻的调用嵌套情况。

寄存器（Register）主要负责存放函数的局部变量，表达式的中间结果和其他值。通过这种方式，可以快速运行编译生成的代码。一台计算机中通常包含一组寄存器，然而，不同的函数和过程往往需要同时使用寄存器。假设函数*f*调用函数*g*，并且两个函数需要使用同一个寄存器*r*。在这种情况下，函数*g*在使用寄存器*r*之前必须先将*r*的旧值保护在栈帧中。在函数*g*返回时，将寄存器*r*的旧值恢复。此时，如果必须由调用者*f*来保护和恢复寄存器*r*，我们称*r*为**调用者保护的寄存器**；如果这是由被调用者*g*负责，则称*r*是**被调用者保护的寄存器**。

栈是一种特殊的数据结构。栈帧包含两个关键的指针：**栈指针（Stack Pointer）**和**帧指针（Frame Pointer）**。栈指针用来标识栈的顶端，栈指针之后的所有位置都视为可用存储空间，而位于栈指针之前的所有位置都视为已分配的存储空间。帧指针是一个单独的寄存器，用来标识当前栈帧的起始位置，其位置可通过栈指针减去栈帧长度的方法获取。

返回地址通常情况下通过call指令产生，主要负责告知当前函数调用结束后应该返回的正确地址。例如，当函数*f*调用函数*g*时，被调用函数*g*必须知道其在执行完后需要返回的位置，假设*f*调用*g*的call指令位于地址*a*，则函数*g*的返回位置应位于call指令的下一条指令处，也即*a*+1。该地址则为**返回地址（Return Address）**。

目前函数调用过程中的参数、返回地址、大部分局部变量和表达式中间结果的传递均需要寄存器实现。然而，仍然存在一些需要将变量存放至栈帧中的情况：变量作为传地址参数时，值太大以至于不能存放至单个寄存器；局部变量和临时变量个数太多，以至于无法全部放入寄存器；

过程嵌套时，变量被当前过程内嵌套的过程访问；需要引用其它元素进行地址运算的数组变量；以及存储变量的寄存器有其他特殊用途等。此外，若把数组或结构体作为参数传递，需要采用**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）**。

4.1.3 中间表示

中间表示（Intermediate Representation，IR）是面向目标机器特点的一种抽象表示形式，它主要用于表示目标机器的操作，屏蔽目标机器过多的复杂细节。实践中，IR 可以有多种形式，例如代码形式、树形结构、线形结构等。引入 IR 可以将源代码转为中**间代码（Intermediate Code）**，有利于编译任务的模块化和移植，便于分析与优化程序，易于生成目标机器指令。例如，一个编译器通常包含前端和后端。前端负责进行语法分析、静态检查， 以及生成中间代码。后端则将中间代码翻译为目标机器的指令，这种设计可以极大地提升编译器的可移植性。如果没有IR，当需要编译*N*种不同的源语言，并为*M*台不同的目标机器生成代码时，则需要实现$N \times M$个完整编译器，这将是一项庞大的工程。而如果将源语言翻译成IR，再将IR转换成目标机器语言，则只需要*N*个前端和*M*个后端，大大降低了复杂度。此外，将编译器设计中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，不仅可以简化模块内部的处理，而且便于单独对每个模块进行调试与修改而不影响其他模块。 接下来我们将分别从IR的层次和表示方式两个方面介绍IR。

1. 基于层次的中间表示分类

在将给定源语言翻译成特定的目标机器语言代码的过程中，一个编译器可能构造一系列IR。从IR所体现出的细节上，我们可以将IR分为如下三类：

（1） **高层次中间表示（High-level IR或HIR）**：这种IR体现了较高层次的程序细节信息，因此往往与高级语言类似，保留了包括数组和循环在内的源语言的特征。HIR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IR。HIR常被用于进行**相关性分析（Dependence Analysis）**和解释执行。例如，Java bytecode、Python bytecode和目前使用非常广泛的LLVM IR等均属于HIR。

（2） **中层次中间表示（Medium-level IR或MIR）**：这个层次的IR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表示。正是由于MIR的这个特性，它具有一定的设计难度。在这个层次上，变量和临时变量已经被区分，

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MIR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）**、以及**代数运算简化（Algebraic Simplification）**等。

（3） **低层次中间表示（Low-level IR，LIR）**：LIR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中的大部分代码与目标语言中的指令往往存在着一一对应的关系。即便不存在完美的对应关系，二者之间的转换通常也可以通过一趟指令选择就能完成。

2. 基于表示方式的中间表示分类

另一方面，从IR的表示方式来看，可以将IR分成如下三类：

（1）**线形中间表示（Linear IR）**：线性IR可以看作是一个简单的指令集。这种结构表示简单、处理高效，但指令之间的先后关系有时会模糊整段指令的逻辑。常用的线性IR为**三地址代码（Three-Address Code）**。三地址代码的一般形式为*x*=*y op z*，其中包含三个地址（每个分量代表一个地址）。在三地址代码中，一条指令的右侧不允许出现组合的算术表示式，也即最多只有一个运算符。例如，表达式*x*+*y**z将被翻译为*t*1=*y**z和t2=*x*+*t*1，其中*t*1和*t*2是编译器生成的临时名字。常用的三地址代码指令有赋值指令*x*=*y op z*、*x*=*op y*、*x*=*y*、无条件转移指令goto *L*、条件转移指令if *x* relop *y* goto *L*、过程调用指令param *x*和call *p, n*、过程返回指令return *y*、索引赋值指令*x*=*y*[*i*]和*x*[*i*]=*y*、以及地址和指针处理指令*x*=&*y*、*x*=**y*和**x*=*y*等。

（2）**图形中间表示（Graphical IR）**：图形IR将输入的源代码信息嵌入到一张图中，以结点和边等元素来进行组织。其中各个结点代表了源代码中的构造，一个结点的所有子结点则反映了该结点对应构造的有意义的组成成分。由于图的表示和处理代价会很大，我们经常会使用一些较为特殊的图表示形式，例如树或**有向无环图（Directed Acyclic Graph，DAG）**。一个典型的基于DAG的树形IR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分析树里不必要的结点，将输入程序的语法信息以一种更加简洁的形式呈现出来。

（3）**混合型中间表示（Hybrid IR）**：IR还可以采用混合图形和线形的表达形式。这种方式可以同时结合这两种形式的优点并避免二者的缺点。例如，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。

4.1.4 类型与声明

1. 类型表达式

类型表达式旨在表示类型的结构，其主要包含以下两种类型：基本类型和基于类型构造算符生成的类型表达式。当前，常见的基本类型通常有boolean、char、integer、float等。基于类型构造算符生成的类型表达式一般包含以下五种：

（1）**对于数组构造算符array**。当T是类型表达式时，则array(I, T)表示一个类型表达式（I是一个整数）。

（2）**对于指针构造算符pointer**。当T是类型表达式时，pointer(T)表示一个指针类型的类型表达式。

（3）**对于笛卡尔乘积构造算符×**。当$T1$和$T2$是类型表达式时，笛卡尔乘积$T1 \times T2$是一个类型表达式。

（4）**对于函数构造算符→**，$T1$、$T2$、…、Tn和R是类型表达式时，则$T1 \times T2 \times \ldots \times Tn \rightarrow R$是一个类型表达式。

（5）**对于记录构造算符record**，当有标识符$N1$、$N2$、…、Nn与类型表达式$T1$、$T2$、…、Tn时，则record$((N1 \times T1) \times (N2 \times T2) \times \ldots \times (Nn \times Tn))$是一个类型表达式。

2. 类型等价

当两个类型表达式满足以下任何一种条件时，我们认为两个类型表达式之间**结构等价（Structurally Equivalent）**：（1）两个类型是相同的基本类型；（2）两个类型是将相同的类型构造符用于结构等价的类型生成的类型表达式；（3）一个类型是另一个类型的类型表达式的别名。如果类型表达式的名字只代表其自身，那么上述条件中的前两个条件表示类型表达式**名等价（Name Equivalent）**。

3. 局部变量名的存储布局

类型的**宽度**是指该类型的一个对象所需的存储单元数量。在编译过程中，编译器可以使用类型的宽度为每个类型名字分配一个相对地址。类型的名字和类型相对地址将保存在对应的符号表中。基本类型和数组类型的翻译方案如图4. 3[1]所示。其中，*type*和*width*表示类型表达式*T*、符号*B*和符号*C*的综合属性。变量*t*和*w*旨在将类型和宽度信息从语法分析树中的*B*结点传递到*C*结点。类型表达式*T*的生成包含符号*B*和*C*以及一个动作。*B*和*C*之间的动作是将*t*设置为*B.type*，并将*w*设置为*B.width*。当解析到*B*→*int*时，*B.type*则设*integer*，*B.width*则为4。类似地，当解析到*B*→*float*时，*B.type*则为*float*，*B.width*则为8。*C*的产生式体决定了*T*是一个基本类型还是数组类型。例如，当解析到*C* →ε时，*t*则为*C.type*，*w*则为*C.width*。当解析到*C* →[*num*]C_1时，则表示将类型构造算符array用于*num.value*和C_1*.type*，从而得到*C.type*。

$T \rightarrow B$	$t=B.type; w=B.width;$
C	$T.type=C.type; T.width=C.width;$
$B \rightarrow int$	$B.type=interger; B.width=4;$
$B \rightarrow float$	$B.type=float; B.width=8;$
$C \rightarrow \varepsilon$	$C.type=t; C.width=w;$
$C \rightarrow [num]C1$	$C.type=array(num.value, C1.type);$ $C.width=num.value * C1.width;$

图 4. 3 计算类型和宽度

4. 类型声明

Java 和 C 语言都可以在语法分析的过程中将所有声明作为一个序列进行处理。为此，可以使用变量 *offset* 来跟踪下一个可用的相对地址。在考虑第一个声明之前，*offset* 被设置为 0。当处理一个变量 *x* 时，*x* 被加入符号表中，它的相对地址被设置为 *offset* 的当前值。随后，*x* 的类型的宽度被加到 *offset* 上。

$P \rightarrow \{offset=0\} D$

$D \rightarrow T\ id;\ \{ST.Put(id.lexeme, T.type, offset);$
$offset=offset+T.width\} D1$

$D \rightarrow \varepsilon$

[1] 图片来源于：《编译原理》，Alfred V. Aho 等著，赵建华、郑滔和戴新宇译，机械工业出版社，第 240 页，2009 年。

图 4. 4[1]展示了声明序列的计算，其中 *T id* 表示一个声明的序列。对于 *D*→*T id*; *D*1 首先需要执行 *T.Put*(*id.lexeme*, *T.type*, *offset*)，其中 *SymbolTable*（即 *ST*）代表当前的符号表，*ST.Put* 表示为 *id.lexeme* 创建一个符号项，该符号项的数据区中存放了类型 *T.type* 和相对地址 *offset*。

$$P \rightarrow \{offset=0\}D$$

$$D \rightarrow T\ id;\ \{ST.Put(id.lexeme, T.type, offset);\ offset=offset+T.width\}D1$$

$$D \rightarrow \varepsilon$$

图 4. 4 计算声明的序列

5. 记录类型

类型构造算符record可以用于处理记录和类的类型，即*T*→*record* ‘{‘ *D* ‘}‘。图4. 5[2]展示了一个记录类型的示例，其中类型表达式包含以下两个动作：

（1）在*D*之前，保存*ST*所代表的符号表，并赋予*ST*新的符号表。这是因为当处理完新符号表中的类型表达式后，编译器可能需要继续处理旧符号表中其他类型表达式。例如，*Env.push*(*ST*)表示将*ST*所表示的当前符号表压入一个栈中。然后，变量*ST*被设置为指向一个新的符号表。类似的，*offset*被推入名为*Stack*的栈中，并且*offset*变量被重置为0。

$$T \rightarrow record\ \text{‘{‘}\quad Env.push(ST);\ ST=\textbf{new}\ Env();\ Stack.push(offset);\ offset=0;$$

$$D\ \text{‘}\}\text{‘}\quad T.type=record(ST);\ T.width=offset;\ ST=Env.pop();\ offset=Stack.pop();$$

图 4. 5 计算记录字段

（2）*D*的声明会保存记录中所有字段的类型和相对地址，并提供存放所有字段所需的存储空间。例如，将*T.type*设为*record*(*ST*)，并将*T.width*设为*offset*。然后，变量*ST*和*offset*将被恢复为原先被压入栈中的值，以完成这个记录类型的翻译。

4.1.5 表达式的翻译

[1] 图片来源于：《编译原理》，Alfred V. Aho 等著，赵建华、郑滔和戴新宇译，机械工业出版社，第 241 页，2009 年。

[2] 图片来源于：《编译原理》，Alfred V. Aho 等著，赵建华、郑滔和戴新宇译，机械工业出版社，第 242 页，2009 年。

1. 赋值语句的翻译

赋值语句的翻译旨在为表达式求值并生成相应的三地址中间代码，我们将在后续的实践技术部分中详细介绍赋值语句翻译的具体实现。这里我们假设*Statement*表示语句，*Expression*表示表达式。*Statement*的属性包含*C*（即 *Code*），*Expression*的属性包含*A*（即 *Address*）和*C*（即 *Code*）。*ST.Get*(*id.lexeme*)表示从符号表*SymbolTable*（即 *ST*）中获取变量*id*所对应的符号信息，*NewTemp*()表示生成一个新的临时变量名，*Gen*(*code*)表示生成一条中间代码。图4. 6展示了常见的赋值语句的翻译规则。

1. *Statement → id = Expression*
```
{
    Statement.C=Expression.C||
    Gen(ST.Get(id.lexeme) ' :=
    'Expression.A);
}
```

2. *Expression → - Expression1*
```
{
    Expression.A = NewTemp();
    Expression.C = Expression1.C
    ||
    Gen(Expression.A ' := '
    ' -' Expression1.A);
}
```

3. *Expression→ (Expression1)*
```
{
    Expression.A = Expression1.A;
    Expression.C = Expression1.C;
}
```

4. *Expression→ id*
```
{
    Expression.A
    =ST.Get(id.lexeme);
    Expression.C = ' ';
}
```

5. *Expression → Num*
```
{
    Expression.A = Num.lexeme;
    Expression.C = ' ';
}
```

6. *Expression→Expression1 + Expression2*
```
{
    Expression.A = NewTemp();
    Expression.C=Expression1.C||Expression2.C||
    Gen(Expression.A' := ' Expression1.A ' + '
    Expression2.A);
}
```

7. *Expression→ Expression1 - Expression2*
```
{
    Expression.A= NewTemp();
    Expression.C=Expression1.C||Expression2.C||
    Gen(Expression.A' := ' Expression1.A ' - '
    Expression2.A);
}
```

8. *Expression→Expression1 * Expression2*
```
{
    Expression.A= NewTemp();
    Expression.C= Expression1.C||Expression2.C||
    Gen(Expression.A' := ' Expression1.A ' * '
    Expression2.A);
}
```

9. *Expression→Expression1 /Expression2*
```
{
    Expression.A = NewTemp();
    Expression.C=Expression1.C||Expression2.C||
    Gen(Expression.A ' := 'Expression1.A ' /
    'Expression2.A);
}
```

10. *Expression →Expression1 % Expression2*
```
{
    Expression.A = NewTemp();
    Expression.C = Expression1.C || Expression2.C ||
    Gen(Expression.A ' := ' Expression1.A ' % '
    Expression2.A);
}
```

图 4. 6 常见的赋值语句的翻译规则

对于*Statement → id = Expression*，其翻译规则是将一个赋值语句翻译为一个中间代码序列。该序列首先通过符号表获取变量名对应的地址，然后将该地址与右侧表达式的值赋给该变量，最后生成一个赋值语句的中间代码。

对于*Expression* → - *Expression1*，其翻译规则是将一个取负表达式翻译为一个中间代码序列。该序列先递归地生成右侧表达式的中间代码，然后生成一个新的临时变量并将其赋给新的中间代码行，该行的代码为将右侧表达式的值取负，并将其存储在新的临时变量中。

对于*Expression* → (*Expression1*)，其翻译规则是将一个括号内的表达式翻译为一个中间代码序列。该序列递归地生成括号内表达式的中间代码，然后直接将该表达式的值赋给新的中间代码行，并且不需要再生成新的临时变量。

对于*Expression* → *id*和*Expression* → *Num*，其翻译规则是将一个变量名或常量值翻译为一个中间代码行，分别获取变量名对应的地址和将常量值直接存储在中间代码中。

对于*Expression* → *Expression1* + *Expression2*，其翻译规则是将一个加法表达式翻译为一个中间代码序列。该序列先递归地生成左侧表达式的中间代码，并且递归地生成右侧表达式的中间代码，然后生成一个新的临时变量并将其赋给新的中间代码行，该行的代码为将左侧表达式的值加上右侧表达式的值，并将其存储在新的临时变量中。

对于*Expression* → *Expression1* - *Expression2*、*Expression* → *Expression1* * *Expression2*、*Expression* → *Expression1* / *Expression2*、*Expression* → *Expression1* % *Expression2*，上述表达式分别对应减法、乘法、除法和求余运算，其翻译规则可以参考加法表达式的翻译规则。

2. 增量翻译

由于属性*C*（即 *Code*）可能是很长的字符串，赋值语句通常采用增量的方式进行翻译。在增量翻译中，函数*Gen*不仅需要生成新的三地址指令，还需要将其添加至目前已生成的指令序列之后。其中，指令序列可以暂时存放在内存中，也可以增量地输出。例如，当采用增量翻译方式翻译赋值语句*Expression* → *Expressio1* + *Expression2*时，可直接调用函数*Gen*来产生一条加法指令。在此之前，增量翻译已经生成指令序列，依次计算*Expression1*的值并放入*Expression1.A*，计算*Expression2*的值并放入*Expression2.A*。**图**4. 7展示了常用赋值语句的增量翻译规则。

3. 数组引用的翻译

为了将数组引用翻译成三地址代码，首先需要确定数组元素的存放地址，也即数组元素的寻址。因此，在介绍数组引用的翻译前，我们介绍数组元素的寻址。

对于一维数组，假设每个数组元素的宽度是*w*，则数组元素*a*[*i*]的相对地址是*base+i*w*，其中，*base*是数组的基地址，*i*w*是偏移地址。

对于二维数组，假设一行的宽度是*w1*，同一行中每个数组元素的宽度是*w2*，则数组元素*a*[*i1*][*i2*]的相对地址是：*base+i1*w1+i2*w2*，其中*i1*w1+i2*w2*是偏移地址。

```
1. Statement → id = Expression
{
    Gen(ST.Get(id.lexeme) ' := ' Expression.A);
}

2. Expression → - Expression1
{
    Expression.A = NewTemp();
    Gen(Expression.A ' := ' ' -' Expression 1.A);
}

3. Expression → (Expression1)
{
    Expression.A = Expression1.A;
}

4. Expression→ id
{
    Expression.A = ST.Get(id. lexeme);
}

5. Expression→ Num
{
    Expression.A = Num.lexeme;
}

6. Expression → Expression1 + Expression2
{
    Expression.A = NewTemp();
    Gen(Expression.A ' := 'Expression1.A '+ '
    Expression2.A);
}
7. Expression→ Expression1 - Expression2
{
    Expression.A = NewTemp();
    Gen(Expression.A ' := ' Expression1.A ' - '
    Expression2.A);
}
8. Expression→Expression1 * Expression2
{
    Expression.A = NewTemp();
    Gen(Expression.A ':= ' Expression1.A ' * '
    Expression2.A);
}
9. Expression→Expression1 /Expression2
{
    Expression.A = NewTemp();
    Gen(Expression.A ' := 'Expression1.A ' / '
    Expression2.A);
}
10. Expression →Expression1 % Expression2
{
    Expression.A = NewTemp();
    Gen(Expression.A ' := 'Expression1.A ' % '
    Expression2.A);
}
```

图 4.7 赋值语句的增量翻译规则

以上二维数组的偏移地址计算可以推广到*k*维的情况，对于*k*维数组，数组元素*a*[*i1*][*i2*]…[*ik*]的相对地址是*base+i1*w1+i2*w2*+…+*ik*wk*，其中*i1*w1+i2*w2*+…+*ik*wk*是偏移地址，*w1*为*a*[*i1*]的宽度, *w2*为*a*[*i1*][*i2*]的宽度，*wk*是*a*[*i1*] [*i2*]…[*ik*]的宽度。

在了解了数组元素的寻址方法后，我们进一步介绍数组引用的翻译规则，如图4.8所示。其中，*Statement*表示语句，*Expression*表示表达式，*ArrayRef*表示数组引用。*ArrayRef*具以下三种属性：*ArrayRef.T*表示数组元素的类型，*ArrayRef.A*表示一个临时变量，该临时变量用于累加公式中的*ij*wj*项，从而计算数组引用的偏移量。*ArrayRef.S*表示数组在符号表中其条目的信息。*ST*表示符号表*SymbolTable*，*ST.Get*(*id.lexeme*)函数表示获取*id*对应的符号信息。

```
1. Statement → id = Expression
{
    Gen(ST.Get(id.lexeme):=Expression.A);
{
2. Statement → ArrayRef = Expression
{
    Gen(ArrayRef.S.BA '['ArrayRef.A']' := Expression.A);
{
3. Expression → ArrayRef
{
    Expression.A = NewTemp();
    Gen(Expression.A :=ArrayRef.S.BA '['ArrayRef.A']');
{
4. ArrayRef → id[Expression]
{
    ArrayRef.S = ST.Get(id.lexeme);
    ArrayRef.T = ArrayRef.S.T.ET;
    ArrayRef.A = NewTemp();
    Gen(ArrayRef.A:= ArrayRef.S.BA + Expression.A * W);
{
5. ArrayRef → ArrayRef1[Expression]
{
    ArrayRef.S = ArrayRef1.S;
    ArrayRef.T = ArrayRef1.T.ET;
    ArrayRef.A = NewTemp();
    t = NewTemp();
    Gen(t := Expression.A * ArrayRef.T.W);
    Gen(ArrayRef.A:= ArrayRef1.A+ t);
{
```

图 4.8 数组引用的翻译规则

对于*Statement → ArrayRef = Expression*，其翻译规则是将表达式*Expression*的值存放在数组引用所对应的内存位置。其中，*ArrayRef.S.BA*表示数组的基地址，即0号元素所对应的地址。*ArrayRef.S.BA*[*ArrayRef.A*]则表示数组引用的位置，该指令将地址*Expression.A*的右值放至内存位置。

对于*Expression → ArrayRef*，其翻译规则将首先生成一个新的临时变量作为该表达式的地址，并用*Expression.A*表示。然后，通过*ArrayRef.S.BA*和*ArrayRef.A*确定数组元素的地址，其中，*ArrayRef.S.BA*是数组的基地址，而*ArrayRef.A*是数组下标对应元素相对于数组起始地址的偏移值。最后，使用*Gen*函数生成一条指令，将*Expression.A*设置为数组元素的地址。

对于*ArrayRef → id*[*Expression*]，其翻译规则首先通过符号表*ST.Get*函数获取到*id*对应的数据类型，存储位置等符号信息，并将其存储在*ArrayRef.S*中。然后，通过*ArrayRef.S*获取该数组元素的数据类型信息，并存储在*ArrayRef.T*中。接着，继续生成一个新的临时变量名并存储在*ArrayRef.A*中，

用于存储数组元素相对于数组起始地址的偏移值。最后，根据数组元素的基地址、偏移值、以及数组元素的宽度（即 *ArrayRef.T.W*），计算出数组元素的地址并存储到*ArrayRef.A*中。

对于*ArrayRef* → *ArrayRef1*[*Expression*]，其翻译规则首先将*ArrayRef1*的符号表信息赋值给*ArrayRef.S*。然后，将*ArrayRef1*的元素类型赋值给*ArrayRef.T*，并分别为*ArrayRef.A*和 *t* 生成新的临时变量。接着，继续计算当前维度的偏移量。由于数组在内存中是连续存储的，每一维的长度是一维元素个数的乘积，所以需要乘上元素的宽度。最后，计算当前元素在内存中的地址，即前一维的地址加上当前维的偏移量。

我们将在后续的实践技术部分中详细介绍数据和结构体翻译的具体实现技术。

4.1.6 控制流与回填

1. 布尔表达式的翻译

if、if-else和while等控制流语句与布尔表达式通常结合在一起使用。布尔表达式可以用于改变语句中控制流。例如，对于if (*BoolExpression*) *Statement*，如果运行至语句*Statement*，则表示表达式*BoolExpression*的取值为真。此外，布尔表达式还可以用于计算逻辑值。布尔表达式的值可以表示为True或者False，然后参考算术表达式的翻译规则。

布尔表达式的翻译规则如图4.9所示，其中*BoolExpression*和*Statement*均包含综合属性*C*（即*Code*）。*BoolExpression.T*和*BoolExpression.F*表示存放了*BoolExpression*为True或者False时控制流指令的跳转目标所在的地址。*NewlLabel*()用于生成一个用于存放标号的新的临时变量。

对于*BoolExpression* → *BoolExpression1* || *BoolExpression2*，如果*BoolExpression1*为真，*BoolExpression*则为真。因此，*BoolExpression1.T*和*BoolExpression.T*相同。如果*BoolExpression1*为假，则需要对*BoolExpression2*进行求值。因此，*BoolExpression1.F*设置为*BoolExpression2.C*代码的第一条指令的标号。*BoolExpression2*的真假目标地址分别等于*BoolExpression*的真假目标地址。

对于*BoolExpression* → *BoolExpression1* && *BoolExpression2*，其翻译与上述翻译规则类似。

1. *BoolExpression → BoolExpression1 || BoolExpression2*
```
{
        BoolExpression1.T = BoolExpression.T;
        BoolExpression1.F=NewLabel();
        BoolExpression2.T = BoolExpression.T;
        BoolExpression2.F = BoolExpression.F;
        BoolExpression.C = BoolExpression1.C||
        Label(BoolExpression1.F)||BoolExpression2.C
}
```
2. *BoolExpression → BoolExpression1 && BoolExpression2*
```
{
        BoolExpression1.T =NewLabel();
        BoolExpression1.F= BoolExpression.F;
        BoolExpression2.T = BoolExpression.T;
        BoolExpression2.F = BoolExpression.F;
        BoolExpression.C =
        BoolExpression1.C||Label(BoolExpression1.T)||
        BoolExpression2.C;
}
```
3. *BoolExpression → !BoolExpression1*
```
{
        BoolExpression1.T = BoolExpression.F;
        BoolExpression1.F = BoolExpression.T;
        BoolExpression.C=BoolExpression1.C;
}
```
4. *BoolExpression → Expression1 < Expression2*
```
{
        BoolExpression.C = Expression1.C ||
        Expression2.C
        ||Gen(' if ' + Expression1.A+
        ' < '+Expression2.A+'goto' BoolExpression.T)
        ||Gen('goto' BoolExpression.F);
}
```
5. *BoolExpression → Expression1 <= Expression2*
```
{
        BoolExpression.C = Expression1.C ||
        Expression2.C
        ||Gen(' if ' + Expression1.A+
        ' <=
        '+Expression2.A+'goto' BoolExpression.T)
        ||Gen('goto' BoolExpression.F);
}
```
6. *BoolExpression → Expression1 > Expression2*
```
{
        BoolExpression.C = Expression1.C ||
        Expression2.C
        ||Gen('if' + Expression1.A+ ' > '
        +Expression2.A+'goto' BoolExpression.T)
        ||Gen('goto' BoolExpression.F);
}
```
7. *BoolExpression → Expression1 >= Expression2*
```
{
        BoolExpression.C = Expression1.C ||
        Expression2.C
        ||Gen('if' + Expression1.A+ ' >='
        +Expression2.A+ 'goto' BoolExpression.T)
        ||Gen('goto' BoolExpression.F);
}
```
8. *BoolExpression → Expression1 == Expression2*
```
{
        BoolExpression.C = Expression1.C ||
        Expression2.C
        ||Gen('if' + Expression1.A+ ' == '
        +Expression2.A+ 'goto' BoolExpression.T)
        ||Gen('goto' BoolExpression.F);
}
```
9. *BoolExpression → Expression1 != Expression2*
```
{
        BoolExpression.C = Expression1.C ||
        Expression2.C
        ||Gen('if' + Expression1.A+ ' != '
        +Expression2.A+'goto' BoolExpression.T)
        ||Gen('goto' BoolExpression.F);
}
```
10. *BoolExpression → BoolFactor*
```
{
        BoolExpression.T = BoolFactor.T;
        BoolExpression.F = BoolFactor.F;
        BoolExpression.C = BoolFactor.C;
}
```

图 4.9 布尔表达式的翻译规则

对于*BoolExpression → !BoolExpression1*，其翻译规则不需要生成新的代码。通过对换*BoolExpression*中真假目标地址，即可以得到*BoolExpression1*的真假目标地址。

对于*BoolExpression → Expression1 relop Expression2*，其中*Expression1 relop Expression2*是一个关系表达式。*Expression1*和*Expression2*为算术表达式，*relop*为关系运算符。*relop*通常包含以下六种：<、<=、>、>=、==和!=。

对于*BoolExpression → BoolFactor*，*BoolFactor*表示常量为True或False时，目标分别为*BoolExpression.T*和*BoolExpression.F*的跳转指令。

此外要说明的是，&&和||是左结合的，优先级从高到低依次为：!、&&和||。另外，在跳转代码中，逻辑运算符&&、||和!被翻译成跳转指令，运算符本身不出现在代码中，布尔表达式的值通过代码序列中的位置来表示。

1. ***Statement*** **→ if (** ***BoolExpression*** **)** ***Statement1***

{

BoolExpression.T =NewLabel();

BoolExpression.F =Statement1.N=Statement.N;

Statement.C=BoolExpression.C || *Label*(*BoolExpression.T*) || *Statement1.C*;

}

2. ***Statement*** **→ if (** ***BoolExpression*** **)** ***Statement1*** **else** ***Statement2***

{

BoolExpression.T = NewLabel();

BoolExpression.F = NewLabel();

Statement1.N = Statement2.N= Statement.N;

Statement.C=BoolExpression.C || *Label*(*BoolExpression.T*) || *Statement1.C*||

Gen('goto' *Statement.N*) || *Label*(*BoolExpression.F*) ||
Statement2.C;

}

3. ***Statement*** **→ while (** ***BoolExpression*** **)** ***Statement1***

{

Begin=NewLabel();

BoolExpression.T =NewLabel();

BoolExpression.F=Statement.N；

Statement1.N=Begin();

Statement.C=Label(*Begin*) || *BoolExpression.C*|| *Label*(*BoolExpression.T*) ||

Statement1.C Gen('goto' *Begin*);

}

4. ***Statement*** **→** ***Statement1 Statement2***

{

Statement1.N=NewLabel();

Statement2.N=Statement.N

Statement.C=Statement1.C || *Label*(*Statement1.N*) || *Statement2.C*

}

图 4. 10 控制流语句的翻译规则

2. 控制流语句的翻译

接下来我们介绍if、if-else和while三种控制流语句的翻译规则，如图4. 10所示。

对于*Statement* → if (*BoolExpression*) *Statement1*,该语句的翻译规则首先初始化新的标号*BoolExpression.T*，并将其关联至*Statement1*生成的第一条三地址指令。因此，跳转至*BoolExpression.T*的指令将转到*Statement1*对应的代码位置。此外，为了确保*BoolExpression*的值为假时，控制流将跳过*Statement1*的代码，*BoolExpression.F*被设置为*Statement*的下一条指定，即 *Statement.N*。

对于*Statement* → if (*BoolExpression*) *Statement1* else *Statement2*，在初始化新的标号*BoolExpression.T*和*BoolExpression.F*后，如果*BoolExpression*的值为真，则跳转至*Statement1.C*的第一条指令。否则跳转至*Statement2.C*的第一条指令。然后，控制流将从*Statement1*或*Statement2*转至*Statement.C*之后的三地址指令。此外，*Statement1.C*之后的goto *Statement.N*语句负责将控制流跳过*Statement2.C*。由于*Statement2.N*即为*Statement.N*，因此*Statement2.C*不需要额外的goto语句。

1. *BoolExpression → BoolExpression1 || Marker BoolExpression2*
```
{
    Marker.instruction=next_instruction
    BackPatch(BoolExpression1.FL, Marker.instruction);
    BoolExpression.TL= merge
    (BoolExpression1.TL,BoolExpression2.TL );
    BoolExpression.FL= BoolExpression2.FL;
}
```
2. *BoolExpression → BoolExpression1 && Marker BoolExpression2*
```
{
    Marker.instruction=next_instruction
    BackPatch( BoolExpression1.TL,Marker.instruction);
    BoolExpression.TL = BoolExpression2.TL;
    BoolExpression.FL= merge( BoolExpression1.FL,
    BoolExpression2.FL);
}
```
3. *BoolExpression → !BoolExpression1*
```
{
    BoolExpression.TL = BoolExpression1.FL;
    BoolExpression.FL= BoolExpression1.TL;
}
```
4. *BoolExpression → (BoolExpression1S*
```
{
    BoolExpression.TL = BoolExpression1.TL ;
    BoolExpression.FL= BoolExpression1.FL;
}
```
5. *BoolExpression → Expression1 < Expression2*
```
{
    BoolExpression.TL = MakeList (next_instruction );
    BoolExpression.FL= MakeList (next_instruction+1 );
    Gen('if' + Expression1.A+'<' + Expression2.A+'goto _');
}
```
6. *BoolExpression→Expression1 <= Expression2*
```
{
    BoolExpression.TL =MakeList (next_instruction);
    BoolExpression.FL=MakeList(next_instruction+1);
    Gen('if' + Expression1.A+'<=' + Expression2.A+'goto _');
}
```
7. *BoolExpression → Expression1 > Expression2*
```
{
    BoolExpression.TL =MakeList (next_instruction );
    BoolExpression.FL=MakeList(next_instruction+1);
    Gen('if' + Expression1.A+'>' + Expression2.A+'goto _');
}
```
8. *BoolExpression → Expression1 >= Expression2*
```
{
    BoolExpression.TL =MakeList (next_instruction);
    BoolExpression.FL=MakeList(next_instruction+1);
    Gen('if' + Expression1.A+'>=' + Expression2.A+'goto _');
}
```
9. *BoolExpression → Expression1 == Expression2*
```
{
    BoolExpression.TL =MakeList (next_instruction);
    BoolExpression.FL=MakeList(next_instruction+1);
    Gen('if' + Expression1.A+'==' + Expression2.A+'goto _');
}
```
10. *BoolExpression → Expression1 != Expression2*
```
{
    BoolExpression.TL =MakeList (next_instruction);
    BoolExpression.FL=MakeList(next_instruction+1);
    Gen('if' + Expression1.A+'!=' + Expression2.A+'goto _' );
}
```
11. *BoolExpression → True*
```
{
    BoolExpression.TL =MakeList(next_instruction);
    Gen('goto _');
}
```
12. *BoolExpression → False*
```
{
    BoolExpression.FL= MakeList (next_instruction);
    Gen('goto _');
}
```

图 4. 11 布尔表达式的回填翻译规则

3. 布尔表达式的回填

回填技术是编译器中负责处理跳转指令的一种技术。回填技术的核心在于允许暂时不指定该跳转指令的目标标号，这样的指令被放入由跳转指令组成的列表中；同一个列表中的所有跳转指令具有相同的目标标号，当可以确定正确的目标标号时，即可填充这些指令的目标标号。例如，在处理if-else语句时，编译器可能无法确定else语句对应的跳转位置。如果不使用回填技术，编译器可能需要多次遍历代码才能确定else语句的跳转位置。但如果采用回填技术，编译器可以先生成占位符，并在确定else语句的跳转位置后再回填该地址，从而增加控制流语句处理的灵活性并提升编译效率。

图4.11展示了布尔表达式的回填规则，其中*BoolExpression*分别包含以下两个综合属性：*BoolExpression.TL*和*BoolExpression.FL*，它们分别表示布尔表达式*BoolExpression*的跳转指定列表*TrueList*和*FalseList*。*MakeList*(*i*)函数可创建一个只包含跳转指令*i*的列表，并返回指向该新创建列表的指针。*Merge*(*p1*, *p2*)表示将*p1*和*p2*指向的列表进行合并，返回指向合并后的列表的指针。*BackPatch*(*p*, *i*)表示将*i*作为目标标号插入到*p*所指列表中的各指令中作为其跳转目标。

例如，对于$BoolExpression \rightarrow BoolExpression1 \,||\, Marker\ BoolExpression2$，如果*BoolExpression1*为真，*BoolExpression*则为真，此时，*BoolExpression1.TL*中的跳转指令则成为了*BoolExpression.TL*的一部分。如果*BoolExpression*1的值为假，*BoolExpression1.FL*中的跳转指令的目标为*BoolExpression2*代码的起始位置。该位置通过标记符*Marker*获取，*Marker*负责为回填操作提供一个标记位置，以便在需要时确定跳转指令的目标标号。在生成*BoolExpression2*代码之前，*Marker*将生成下一条指定的序号，并存放入*Marker*的综合属性*Marker.instruction*中。其中，*Marker.instruction* = *next_instruction*，而*next_instruction*保存了下一条指令的序号。图4.11中其它布尔表达式的回填规则与上述表达式类似。

4. 控制流语句的回填

图4.12展示了控制流语句的回填规则。其中*Statement*和*StatementList*分别表示语句和语句列表。*Assignment*表示赋值语句，*BoolExpression*表示布尔表达式，*TL*和*FL*分别表示*TrueList*和*FalseList*。*Marker*表示跳转目标代码的位置。*BoolExpression.TL*和*BoolExpression.FL*分别表示布尔

表达式*BoolExpression*的跳转指定列表*TrueList*和*FalseList*。*Statement.NL*或*StatementList.NL*包含了所有按运行顺序跳转至*Statement*或*StatementList*代码之后的指定条件或者无条件跳转指令。

例如，对于 *Statement* → if (*BoolExpression*) then *Marker1 Statement1 NextMarker* else *Marker2 Statement2*，如果*BoolExpression*的值为真，则采用*Marker1.instruction*回填跳转指令，并指向*Statement1*的起始位置。如果*BoolExpression*的值为假，则采用*Marker2.instruction*回填跳转指令，并指向*Statement2*的起始位置。*Statement.NL*则包含了所有从*Statement1*和*Statement2*中跳出的指令。图4. 12中其他控制流语句的回填规则与上述语句相似。

```
1. Statement → if (BoolExpression) then Marker Statement1
{
    Marker.instruction=next_instruction;
    BackPatch(BoolExpression.TL,Marker.instruction);
    Statement.NL=Merge(BoolExpression.FL, Statement1.NL);
}
2. Statement → if (BoolExpression) then Marker1 Statement1 NextMarker
else Marker 2 Statement2
{
    Marker1.instruction=next_instruction1;
    Marker2.instruction=next_instruction2;
    NextMarker.NL ={MakeList(next_instruction);Gen('goto'); }
    BackPatch(BoolExpression.FL, Marker1.instruction);
    BackPatch(BoolExpression.TL, Marker2.instruction);
    temp=Merge (Statement1.NL, NextMarker.NL);
    Statement.NL = Merge (temp,Statement2.NL);
}
3. Statement → while Marker1 (BoolExpression) Marker2 Statement1
{
    Marker1.instruction=next_instruction1;
    Marker2.instruction=next_instruction2;
    BackPatch(Statement1.nextlist, Marker1.instruction);
    BackPatch(B.truelist, Marker2.instruction);
    Statement.NL = BoolExpression.FL;
    Gen('goto' Marker1.instruction);
}
4. Statement → {StatementList}
{
    Statement.NL =StatementList.NL;
}
5. Statement → Assignment Statement
{
    Statement.NL =null;
}
6. StatementList → StatementList1 Marker Statement
{
    BackPatch(StatementList1.NL, Marker1.instruction);
    StatementList.NL =Statement.NL;
}
7. StatementList → Statement
{
    StatementList.NL =Statement.NL;
}
```

图 4. 12 控制流语句的回填翻译规则

4.2 中间代码生成的实践技术

4.2.1 线形中间表示

为了实现线形IR，可以采用**四元式**、**三元式**、或者**间接三元式**来表示三地址指令。我们接下来将分别介绍以上三种三地址指令的表示形式。

四元式的格式如下：*op arg1 arg2 result*。其中，*op*为运算符的内部编码，*arg1*、*arg2*和*result*为地址。例如， 对于*x=y+z*而言，*op*存放+操作，*arg1*为*y*，*arg2*为*z*，*result*为*x*。需要注意的是，单目运算符不使用*arg2*，*param*运算不使用*arg2*和*result*，条件转移或者非条件转移将目标标号放在*result*字段。三元式的格式如下：*op arg1 arg2*。在三元式的表示形式中，三元式的位置将被用来引用三元式的运算结果。间接三元式包含了一个指向三元式的指针的列表。通过对列表进行操作，完成优化功能。此外，间接三元式在操作时不需要修改三元式中的参数。

除了这三种表示之外此外，**静态单赋值形式（Static Single Assignment）**是一种便于某些代码优化的IR形式。静态单赋值形式和三地址代码主要区别有两点：（1）所有赋值指令都是对不同名字的变量的赋值；（2）对于同一个变量在不同路径中定值的情况，可以使用φ函数来合并不同的定值。例如，对于赋值语句*if* (*temp*) *a*=100；*else a*=-100；*c=a*b*。基于静态单赋值的表示为*if* (*temp*) *a1*=100；*else a2*=-100；*a3*=φ (*a1, a2*)；静态单赋值使用函数φ将*a*的两处定值合并，根据到达φ的赋值语句的不同控制流路径，φ返回不同的参数值。例如，如果控制流经过上述赋值语句的真分支，φ (*a1, a2*)的值为*a1*；如果经过假分支，φ (*a1, a2*)的值为*a2*。

4.2.2 图形中间表示

树形结构是使用较为广泛的一种图形IR形式。树形结构具有层次的概念，在靠近树根的高层部分的IR其抽象层次较高，而靠近树叶的低层部分的IR则更加具体。 对于树形的IR，可以看作是一种语法树（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。 为了进一步展示树形IR，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。

4.2.3 运行时环境简介

与运行时环境相关的理论内容，本书已经在前面的理论方法部分第4.1.1章节进行了详细介绍，接下来将具体介绍基本类型、数据组和结构体以及函数调用的表示。

对于基本类型，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double类型则会对应到底层机器上的八个字节，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（32位机器）或者八个字节（64位机器）整数表示，其内容即为指针所指向的内存地址。

对于数组类型，各种语言的编译器有不同的设计。在表示一维数组时，C语言中的数组元素一个挨着一个并占用一段连续的内存空间（如图4. 13[1]所示）；Java语言则将数组长度放在起始位置（如图4. 14所示）；而D语言将采用两个指针表示数组，其中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4. 15所示）。多维数组的表示可以看作是其中元素是低维数组的数组。结构体的表示与数组类型类似，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。

图 4. 13 C 语言中一维数组的内存表示方式

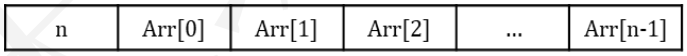

图 4. 14 Java 语言中的一维数组表示方式

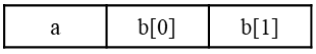

图 4. 15 D 语言中的结构体的表示方式

对于函数调用的表示，将通过活动记录（栈帧）存放函数调用过程中的各种信息，并需要压入相应的参数然后调用call指令。与活动记录（栈帧）相关的理论内容可参考前面的理论方法部分。

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

4.2.4 翻译模式（基本表达式）

我们将采用语法制导的翻译，为每个主要的语法单元“X”设计相应的翻译函数“translate_X”。其中，对语法树的遍历过程可以看作是对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”。表4.1总结了基本表达式的翻译模式。

表 4.1 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][1]
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp_1 ASSIGNOP Exp_2[2] (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] +[3] [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	

[1] 用方括号括起来的内容表示新建一条具体的中间代码。

[2] 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。

[3] 这里的加号相当于连接运算，表示将两段代码连接成一段。

函数translate_Exp()为基本表达式的翻译函数，返回值为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）。该函数的参数分别为语法树的结点*Exp*、符号表sym_table、以及一个变量名*place*。

（1）当表达式*Exp*为INT时，其翻译模式是为*place*变量赋值添加一个“#”。

（2）当表达式*Exp*为ID时，其翻译模式是为*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）。

（3）当表达式*Exp*为赋值表达式*Exp1* ASSIGNOP *Exp2*时，左值的*Exp1*的为以下三种情况之一：单个变量访问、数组元素访问或结构体特定域的访问。关于数组和结构体的翻译模式，将在后面部分详细介绍。对于*Exp1* → ID时，其翻译模式为通过查表找到ID对应的变量，对*Exp2*进行翻译（运算结果储存在临时变量*t*1中）。然后，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。

（4） 当表达式*Exp*为算术运算表达式*Exp1* PLUS *Exp2*时，其翻译模式先对*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。

（5） 当表达式*Exp*为取负表达式MINUS *Exp1*，其翻译模式先对*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。

4.2.5 语句的翻译模式（语句）

我们接下来介绍表达式语句、复合语句、返回语句、跳转语句和循环语句的翻译模式，其翻译模式如表4. 2所示。

对于循环语句的翻译，由于我们在翻译条件表达式的同时生成条件跳转语句，因此，这部分的翻译模式将不包含条件跳转。translate_Cond函数负责对条件表达式进行翻译。

表 4.2 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]

对于条件表达式的翻译，其翻译模式如**表4.3**所示。这里没有使用回填表示，而是将跳转的两个目标*label_true*和*label_false*作为继承属性（函数参数）进行处理。表达式内部需要跳转到外面时，跳转目标从父结点通过参数直接填上即可。与回填相关内容可参考前面的理论方法部分。

4.2.6 函数调用的翻译模式

函数调用的翻译模式通过translate_Exp实现，具体翻译模式如**表4. 4**所示。与回填相关内容可参考前面的理论方法部分。假定需要翻译函数read和write时，当从符号表中找到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，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4. 5**所示。

表 4. 3 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]

表 4.4 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 4.5 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

4.2.7 数组与结构体的翻译模式

数组和结构体的翻译模式需要设计到其内存地址的运算。对于数组，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]，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：

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

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

对于结构体，其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*，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）。对于一个有*n*个域的结构体，可以将其看作是一个有*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：

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

对于同时结合数组和结构的情况， 当一个结构体的元素为数组时，首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。当一个数组的元素为结构体时，首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。值得注意的是，在上述过程中，需记录并区分在访问过程中代表地址的临时变量和代表内存中的数值的临时变量。

最后，具体的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。

4.3 中间代码生成的实践内容

4.3.1 实践要求

在本章节的实践内容需要在符合以下假设的前提下， 将源代码翻译为中间代码。具体假设如下：

（1）**假设 1**：不会出现注释、八进制或十六进制整型常数、浮点型常数或者变量。

（2）**假设 2**：不会出现类型为结构体或高维数组（高于一维的数组）的变量。

（3）**假设 3**：任何函数参数都只能为简单变量，也就是说，结构体和数组都不会作为参数传入函数中。

（4）**假设 4**：没有全局变量的使用，并且所有变量均不重名。

（5）**假设 5**：函数不会返回结构体或数组类型的值。

（6）**假设 6**：函数只会进行一次定义（没有函数声明）。

（7）**假设 7**：输入文件中不包含任何词法、语法或语义错误（函数也必有 return 语句）。

此外，在翻译为中间代码过程中，需要符合**表 4.6** 中的操作规范，具体内容如下：

（1）标号语句 LABEL 用于指定跳转目标，注意 LABEL 与 *x* 之间、*x* 与冒号之间都被空格或制表符隔开。

（2）函数语句 FUNCTION 用于指定函数定义，注意 FUNCTION 与 *f* 之间、*f* 与冒号之间都被空格或制表符隔开。

（3）赋值语句可以对变量进行赋值操作（注意赋值号前后都应由空格或制表符隔开）。赋值号左边的 *x* 一定是一个变量或者临时变量，而赋值号右边的 *y* 既可以是变量或临时变量，也可以是立即数。如果是立即数，则需要在其前面添加“#”符号。例如，如果要将常数 5 赋给临时变量 *t1*，可以写成 *t1* := #5。

（4）算术运算操作包括加、减、乘、除四种操作（注意运算符前后都应由空格或制表符隔开）。赋值号左边的 *x* 一定是一个变量或者临时变量，而赋值号右边的 *y* 和 *z* 既可以是变量或临时变量，也可以是立即数。如果是立即数，则需要在其前面添加“#”符号。例如，如果要将变量 *a* 与常数 5 相加并将运算结果赋给 *b*，则可以写成 *b* := *a* + #5。

（5）赋值号右边的变量可以添加“&”符号对其进行取地址操作。例如，*b* := &*a* + #8 代表将变量 *a* 的地址加上 8 然后赋给 *b*。

（6）当赋值语句右边的变量 *y* 添加了“*”符号时，代表读取以 *y* 的值作为地址的那个内存单元的内容，而当赋值语句左边的变量 *x* 添加了“*”符号时，则代表向以 *x* 的值作为地址的那个内存单元写入内容。

（7）跳转语句分为无条件跳转和有条件跳转两种。无条件跳转语句 GOTO *x* 会直接将控制转移到标号为 *x* 的那一行，而有条件跳转语句（注意语句中变量、关系操作符前后都应该被空格或制表符分开）则会先确定两个操作数 *x* 和 *y* 之间的关系（相等、不等、小于、大于、小于等于、大于等于共 6 种），如果该关系成立则进行跳转，否则不跳转而直接将控制转移到下一条语句。

（8）返回语句 RETURN 用于从函数体内部返回值并退出当前函数，RETURN 后面可以跟一个变量，也可以跟一个常数。

（9）变量声明语句 DEC 用于为一个函数体内的局部变量声明其所需要的空间，该空间的大

小以字节为单位。这个语句是专门为数组变量和结构体变量这类需要开辟一段连续的内存空间的变量所准备的。例如，如果我们需要声明一个长度为 10 的 int 类型数组 *a*，则可以写成 DEC *a* 40。对于那些类型不是数组或结构体的变量，直接使用即可，不需要使用 DEC 语句对其进行声明。变量的命名规范与之前的要求相同。另外，在中间代码中不存在作用域的概念，因此不同的变量一定要避免重名。

（10）与函数调用有关的语句包括 CALL、PARAM 和 ARG 三种。其中 PARAM 语句在每个函数开头使用，对于函数中形参的数目和名称进行声明。例如，若一个函数 func 有三个形参 *a*、*b*、*c*，则该函数的函数体内前三条语句为：PARAM *a*、PARAM *b* 和 PARAM *c*。CALL 和 ARG 语句负责进行函数调用。在调用一个函数之前，我们先使用 ARG 语句传入所有实参，随后使用 CALL 语句调用该函数并存储返回值。仍以函数 func 为例，如果我们需要依次传入三个实参 *x*、*y*、*z*，并将返回值保存到临时变量 *t1* 中，则可分别表述为：ARG *z*、ARG *y*、ARG *x* 和 *t1* := CALL func。注意 ARG 传入参数的顺序和 PARAM 声明参数的顺序正好相反。ARG 语句的参数可以是变量、以#开头的常数或以&开头的某个变量的地址。注意：当函数参数是结构体或数组时，ARG 语句的参数为结构体或数组的地址（即以传引用的方式实现函数参数传递）。

（11）输入输出语句 READ 和 WRITE 用于和控制台进行交互。READ 语句可以从控制台读入一个整型变量，而 WRITE 语句可将一个整型变量的值写到控制台上。

除以上说明外，注意关键字及变量名都是大小写敏感的。例如，“*abc*”和“*AbC*”会被作为两个不同的变量对待，上述所有关键字（例如 CALL、IF、DEC 等）都必须大写，否则虚拟机小程序会将其看作一个变量名。

表 4.6 中间代码的形式及操作规范

语法	描述
LABEL x :	定义标号 x。
FUNCTION f :	定义函数 f。
x := y	赋值操作。
x := y + z	加法操作。
x := y - z	减法操作。
x := y * z	乘法操作。
x := y / z	除法操作。
x := &y	取 y 的地址赋给 x。
x := *y	取以 y 值为地址的内存单元的内容赋给 x。
*x := y	取 y 值赋给以 x 值为地址的内存单元。
GOTO x	无条件跳转至标号 x。
IF x [relop] y GOTO z	如果 x 与 y 满足[relop]关系则跳转至标号 z。
RETURN x	退出当前函数并返回 x 值。
DEC x [size]	内存空间申请，大小为 4 的倍数。
ARG x	传实参 x。
x := CALL f	调用函数，并将其返回值赋给 x。
PARAM x	函数参数声明。
READ x	从控制台读取 x 的值。
WRITE x	向控制台打印 x 的值。

我们可以考虑在后续实践内容中，需要在符号表中预先添加 read 和 write 这两个预定义的函数。其中 read 函数没有任何参数，返回值为 int 型（即读入的整数值），write 函数包含一个 int 类型的参数（即要输出的整数值），返回值也为 int 型（固定返回 0）。添加这两个函数的目的是让 C−−源程序拥有可以与控制台进行交互的接口。在中间代码翻译的过程中，read 函数可直接对应 READ 操作，write 函数可直接对应 WRITE 操作。

除此之外，还可以选择完成以下部分或全部的要求：

（1）**要求 4.1**：修改前面对 C−−源代码的假设 2 和 3，使源代码中：

1） 可以出现结构体类型的变量（但不会有结构体变量之间直接赋值）。

2） 结构体类型的变量可以作为函数的参数（但函数不会返回结构体类型的值）。

（2）**要求 4.2**：修改前面对 C−−源代码的假设 2 和 3，使源代码中：

1） 一维数组类型的变量可以作为函数参数（但函数不会返回一维数组类型的值）。

2） 可以出现高维数组类型的变量（但高维数组类型的变量不会作为函数的参数或返回类值）。

4.3.2 输入格式

实践内容要求程序的输入是一个包含 C 源代码的文本文件，也即程序需要能够接收一个输入文件名和一个输出文件名作为参数。例如，假设程序名为 cc、输入文件名为 test1、输出文件名为 out1.ir，程序和输入文件都位于当前目录下，在 Linux 命令行下运行./cc test1 out1.ir 即可将输出结果写入当前目录下名为 out1.ir 的文件中。

4.3.3 输出格式

运行结果需要输出到文件。输出文件要求每行一条中间代码，每条中间代码的含义如前文所述。如果输入文件包含多个函数定义，则需要通过 FUNCTION 语句将这些函数隔开。FUNCTION 语句和 LABEL 语句的格式类似，具体例子见后面的样例。

4.3.4 验证环境

程序将在如下环境中被编译并运行：

（1）GNU Linux Release: Ubuntu 20.04, kernel version 5.13.0-44-generic；

（2）GCC version 7.5.0；

（3）GNU Flex version 2.6.4；

（4）GNU Bison version 3.5.1。

一般而言，只要避免使用过于冷门的特性，使用其它版本的 Linux 或者 GCC 等，基本上不会出现兼容性方面的问题。

4.3.5 提交要求

具体提交内容如下：

(1) Flex、Bison以及C语言的可被正确编译运行的源代码程序。

(2) 一份PDF格式的报告，内容包括：

a) 提交的程序实现了哪些功能？简要说明如何实现这些功能。清晰的说明有助于助教对提交的程序所实现的功能进行合理的测试。

b) 提交的程序应该如何被编译？可以使用脚本、makefile或逐条输入命令进行编译，请详细说明应该如何编译提交的程序。无法顺利编译将导致助教无法对提交的程序所实现的功能进行任何测试，从而丢失相应的分数。

c) 报告的长度不得超过三页！所以报告中需要重点描述的是提交的程序中的亮点，是提交者认为最个性化、最具独创性的内容，而相对简单的、任何人都可以做的内容则可不提或简单地提一下，尤其要避免大段地向报告里贴代码。报告中所出现的最小字号不得小于五号字（或英文11号字）。

4.3.6 样例（必做部分）

本样例包括**必做内容样例**与**选做要求样例**两部分，分别对应于必做内容和选做要求。请仔细阅读样例，以加深对实践要求以及输出格式要求的理解。这节列举必做内容样例。

样例1：

输入：

```
int main()
{
  int n;
  n = read();
  if (n > 0) write(1);
  else if (n < 0) write (-1);
  else write(0);
  return 0;
}
```

输出：

这段程序读入一个整数n，然后计算并输出符号函数sgn(n)。它所对应的中间代码可以是这样的：

```
FUNCTION main :
READ t1
v1 := t1
t2 := #0
IF v1 > t2 GOTO label1
GOTO label2
LABEL label1 :
t3 := #1
```

```
 9  WRITE t3
10  GOTO label3
11  LABEL label2 :
12  t4 := #0
13  IF v1 < t4 GOTO label4
14  GOTO label5
15  LABEL label4 :
16  t5 := #1
17  t6 := #0 - t5
18  WRITE t6
19  GOTO label6
20  LABEL label5 :
21  t7 := #0
22  WRITE t7
23  LABEL label6 :
24  LABEL label3 :
25  t8 := #0
26  RETURN t8
```

需要注意的是，虽然样例输出中使用的变量遵循着字母后跟一个数字（如t1、v1等）的方式，标号也遵循着label后跟一个数字的方式，但这并不是强制要求的。也就是说，程序输出完全可以使用其它符合变量名定义的方式而不会影响虚拟机小程序的运行。

可以发现，这段中间代码中存在很多可以优化的地方。首先，我们将0这个常数赋给了t2、t4、t7、t8这四个临时变量，实际上赋值一次就可以了。其次，对于t6的赋值我们可以直接写成t6 := #-1而不必多进行一次减法运算。另外，程序中的标号也有些冗余。如果提交的程序足够“聪明”，可能会将上述中间代码优化成这样：

```
 1  FUNCTION main :
 2  READ t1
 3  v1 := t1
 4  t2 := #0
 5  IF v1 > t2 GOTO label1
 6  IF v1 < t2 GOTO label2
 7  WRITE t2
 8  GOTO label3
 9  LABEL label1 :
10  t3 := #1
11  WRITE t3
12  GOTO label3
13  LABEL label2 :
14  t6 := #-1
15  WRITE t6
16  LABEL label3 :
17  RETURN t2
```

样例2：

输入：

```
1  int fact（int n）
2  {
3    if （n == 1）
```

```
4     return n;
5   else
6     return （n * fact（n - 1））;
7  }
8  int main（）
9  {
10   int m, result;
11   m = read（）;
12   if （m > 1）
13     result = fact（m）;
14   else
15     result = 1;
16   write（result）;
17   return 0;
18 }
```

输出：

这是一个读入m并输出m的阶乘的小程序，其对应的中间代码可以是：

```
1  FUNCTION fact :
2  PARAM v1
3  IF v1 == #1 GOTO label1
4  GOTO label2
5  LABEL label1 :
6  RETURN v1
7  LABEL label2 :
8  t1 := v1 - #1
9  ARG t1
10 t2 := CALL fact
11 t3 := v1 * t2
12 RETURN t3
13
14 FUNCTION main :
15 READ t4
16 v2 := t4
17 IF v2 > #1 GOTO label3
18 GOTO label4
19 LABEL label3 :
20 ARG v2
21 t5 := CALL fact
22 v3 := t5
23 GOTO label5
24 LABEL label4 :
25 v3 := #1
26 LABEL label5 :
27 WRITE v3
28 RETURN #0
```

这个样例主要展示如何处理包含多个函数以及函数调用的输入文件。

4.3.7 样例（选做部分）

样例1：

输入：

```
1  struct Operands
2  {
```

```
 3   int o1;
 4   int o2;
 5 };
 6
 7 int add（struct Operands temp）
 8 {
 9   return （temp.o1 + temp.o2）;
10 }
11
12 int main（）
13 {
14   int n;
15   struct Operands op;
16   op.o1 = 1;
17   op.o2 = 2;
18   n = add（op）;
19   write（n）;
20   return 0;
21 }
```

输出：

样例输入中出现了结构体类型的变量，以及这样的变量作为函数参数的用法。如果程序需要完成要求4.1，样例输入对应的中间代码可以是：

```
 1 FUNCTION add :
 2 PARAM v1
 3 t2 := *v1
 4 t7 := v1 + #4
 5 t3 := *t7
 6 t1 := t2 + t3
 7 RETURN t1
 8 FUNCTION main :
 9 DEC v3 8
10 t9 := &v3
11 *t9 := #1
12 t12 := &v3 + #4
13 *t12 := #2
14 ARG &v3
15 t14 := CALL add
16 v2 := t14
17 WRITE v2
18 RETURN #0
```

样例2：

输入：

```
 1 int add（int temp[2]）
 2 {
 3   return （temp[0] + temp[1]）;
 4 }
 5
 6 int main（）
 7 {
 8   int op[2];
 9   int r[1][2];
10   int i = 0, j = 0;
11   while （i < 2）
```

```
12   {
13     while （j < 2）
14     {
15       op[j] = i + j;
16       j = j + 1;
17     }
18     r[0][i] = add（op）;
19     write（r[0][i]）;
20     i = i + 1;
21     j = 0;
22   }
23   return 0;
24 }
```

输出：

样例输入中出现了高维数组类型的变量，以及一维数组类型的变量作为函数参数的用法。如果程序需要完成要求4.2，样例输入对应的中间代码可以是：

```
1  FUNCTION add :
2  PARAM v1
3  t2 := *v1
4  t11 := v1 + #4
5  t3 := *t11
6  t1 := t2 + t3
7  RETURN t1
8  FUNCTION main :
9  DEC v2 8
10 DEC v3 8
11 v4 := #0
12 v5 := #0
13 LABEL label1 :
14 IF v4 < #2 GOTO label2
15 GOTO label3
16 LABEL label2 :
17 LABEL label4 :
18 IF v5 < #2 GOTO label5
19 GOTO label6
20 LABEL label5 :
21 t18 := v5 * #4
22 t19 := &v2 + t18
23 t20 := v4 + v5
24 *t19 := t20
25 v5 := v5 + #1
26 GOTO label4
27 LABEL label6 :
28 t31 := v4 * #4
29 t32 := &v3 + t31
30 ARG &v2
31 t33 := CALL add
32 *t32 := t33
33 t41 := v4 * #4
34 t42 := &v3 + t41
35 t35 := *t42
36 WRITE t35
37 v4 := v4 + #1
38 v5 := #0
39 GOTO label1
```

```
40  LABEL label3 :
41  RETURN
```

如果程序不需要完成要求4.2，将不能翻译该样例输入，程序可以给出如下的提示信息：

```
 Cannot translate: Code contains variables of multi-dimensional array type or
parameters of array type.
```

4.4 本章小结

本章介绍了编译器中生成中间代码的过程。我们提供了中间代码生成的基础理论，其中包括运行时环境、存储组织、栈帧设计方法、中间代码的层次和表示方式、类型和声明的相关概念、表达式翻译（赋值语句、增量翻译、数组引用）和控制流与回填（布尔表达式、控制流语句）的翻译方法。基于上述理论方法，本章介绍了中间代码的实践技术，其中包含线性和图形的 IR 形式、基本表达式 、语句、函数调用、数组和结构体的翻译模式。通过本章的学习，读者可以掌握中间代码生成的完整流程和方法，为后续代码生成和优化提供支持。

习题

4.1 有如下 C 语言代码：

```
void fun(int x, int y)
{
   int a, b, c;
   …
}
int main()
{
   int i, j;
   fun(i, j);
   …
   return 0;
}
```

其文法为 $G[S]$：$S \rightarrow L, L \rightarrow L, id \mid T\ id, T \rightarrow int \mid float$。

写出调用函数 *fun* 的活动记录栈帧。

4.2 有如下语言代码，其中 *a, b, c* 是全局变量，*i, j* 是局部变量。

```
a, b, c : int;
int main()
{
   i, j : int;
   …
   return 0;
}
```

其文法为 $G[S]$：$S \rightarrow id\ L, L \rightarrow, id\ L \mid : T, T \rightarrow int \mid float$。变量 *a, b, c, i, j* 各放在哪个数据区？请选择数据区的一个基地址，给出各变量相对基地址偏移量。

4.3 有以下翻译模式：

(1) $P \rightarrow MD$　　$D \rightarrow D; D$

(2) $M \rightarrow \varepsilon$　　{*offset*=0;}

(3) $D \rightarrow id\ L$　　{*enter*(*id.name*, *L.type*, *offset*); *offset*=*offset*+*L.width*;}

(4) $L \rightarrow id\ L1$　　{*enter*(*id.name*, *L*1.*type*, *offset*); *offset*=*offset*+*L1.width*;

L.type=*L*1.*type*; *L.width*=*L*1.*width*;}

(5) $L \rightarrow :T$　　{*L.type*=*T.type*; *L.width*=*T.width*;}

(6) *T*→*integer* {*T.type*=*integer*; *T.width*=4;}

(7) *T*→*real* {*T.type* =*real*; *T.width*=8;}

根据该翻译模式，写出如下声明语句的符号表：*a, b, c*: *real*

4.4 有以下翻译模式：

(1) *S*→*id*=*E* {*gen*(*ST.Get*(*id.lexeme*) ':=' *E.A*) ;}

(2) *E*→*E*1+*E*2 {*E.A*=*newtemp*; *gen*(*E.A*':='*E*1.*A*'+'*E*2.*A*) ;}

(3) *E*→*E*1-*E*2 {*E.A*=*newtemp*; *gen*(*E.A*':='*E*1.*A*'-'*E*2.*A*) ;}

(4) *E*→*E*1**E*2 {*E.A*=*newtemp*; *gen*(*E.A*':='*E*1.*A*'*'*E*2.*A*) ;}

(5) *E*→*E*1/*E*2 {*E.A*=*newtemp*; *gen*(*E.A*':='*E*1.*A*'/'*E*2.*A*) ;}

(6) *E*→-*E*1 {*E.A*=*newtemp*; *gen*(*E.A*':= ' '-'*E*1.*A*) ;}

(7) *E*→*id* {*E.A* = *ST.Get*(*id.lexeme*) ;}

根据该翻译模式翻译如下句子：

x = (*a* + *b*) * -(*a* + *b*)

4.5 有以下翻译模式：

(1) *S*→*L*=*E* {if *L.offset*=*null gen*(*L.A*':='*E.A*); else *gen*(*L.A*[*L.offset*]':='*E.A*);}

(2) *E*→*E*1+*E*2 {*E.A*=*newtemp*; *gen*(*E.A*':='*E*1.*A*'+'*E*2.*A*) ;}

(3) *E*→*E*1-*E*2 {*E.A*=*newtemp*; *gen*(*E.A*':='*E*1.*A*'-'*E*2.*A*) ;}

(4) *E*→*E*1**E*2 {*E.A*=*newtemp*; *gen*(*E.A*':='*E*1.*A*'*'*E*2.*A*) ;}

(5) *E*→*E*1/*E*2 {*E.A*=*newtemp*; *gen*(*E.A*':='*E*1.*A*'/'*E*2.*A*) ;}

(6) *E*→-*E*1 {*E.A*=*newtemp*; *gen*(*E.A*':= ' '-'*E*1.*A*) ;}

(7) *E*→(*E*1) {*E.A*=*E*1.*A*;}

(8) *E*→*L* {if *L.offset*=*null* *E.A*=*L.A*;

else { *E.A*=*newtemp*; *gen*(*E.A*':='*L.A*'['*L.offset*']'); } }

(9) *L*→[*Elist*] {*L.A*=*newtemp*;*gen*(*L.A*':='*Elist.array*);

L.offset=*newtemp*; *gen*(*L.offset*':='*Elist.A*'*'*w*); } // *w* 为字宽

(10) *L*→*id* {*L.A*=*ST.Get*(*id.lexeme*);*L.offset*=*null*;}

(11) *Elist*→*Elist*1, *E* {*t*=*newtemp*; *m*=*Elist.ndim*+1; // 用一次维度+1

gen(*t*':='*Elist*1.*A*'*'*limit*(*Elist*1.*array*,*m*)); *gen*(*t*':='*t*'+'*E.A*);

Elist.array=*Elist*1.*array*; *Elist.A*=*t*; *Elist.ndim*=*m*;}

(12) *Elist*→*id*[*E*] {*Elist.A*=*E.A*; *Elist.ndim*=1; *Elist.array*=*id.place*;}

根据该翻译模式翻译如下句子：

x = *a*[*i*, *j*]，其中 *a* 的两个维度分别为 10 和 20。

4.6 第 4.4 题的翻译模式，将加减乘除的运算修改为：

E→*E*1 θ *E*2

{*E.A*=*newtemp*;

if *E*1.*type*= =*integer* && *E*2.*type*= =*integer* {*gen*(*E.A*':='*E*1.*A*'*θ*'*E*2.*A*); *E.type*=*integer*; }

else if *E*1.*type*= =*real* && *E*2.*type*= =*real* {*gen*(*E.A*':='*E*1.*A*'*θ*'*E*2.*A*); *E.type*=*real*;}

else if *E*1.*type*= =*integer* && *E*2.*type*= =*real* {*u*=*newtemp*; *gen*(*u*':='*int*2*real*('*E*1.*A*));

gen(*E*.A':='*u* '*θ*'*E*2.*A*,); *E.type*=*real*;}

else if *E*1.*type*= =*real* && *E*2.*type*= =integer {*u*=*newtemp*; *gen*(*u*':='*int*2*real*(*E*2.*A*));

gen(*E.A*':='*E*1.*A*'*θ*'*u*); *E.type*=*real*;}

else *E.type*=*type_error*;}

其中 θ 表示加减乘除运算。根据该翻译模式翻译如下句子：

b = *a* * (*i* + *j*)，其中 *a*、*b* 为 *rea*l 类型，*i*、*j* 为整型。

4.7 有以下翻译模式：

(1) $E \rightarrow E1 \vee E2$ {*E.A=newtemp*; *gen*(*E.A*':='*E1.A*∨*E2.A*);}

(2) $E \rightarrow E1 \wedge E2$ {*E.A=newtemp*; *gen*(*E.A*':='*E1*.A∧*E2.A*);}

(3) $E \rightarrow \neg E1$ {*E.A=newtemp*; *gen*(*E.A*':=¬'E*1.A*);}

(4) $E \rightarrow (E1)$ {*E.A*=E*1.A*;}

(5) $E \rightarrow$id {*E.A=ST.Get*(*id.lexeme*);}

(6) $E \rightarrow id1\theta id2$ {*E.A=newtemp*; *gen* (if (*ST.Get*(*id*1.*lexeme*) '*θ*' *ST.Get*(*id*2.*lexeme*)) goto *nxq*+3)

gen(*E.A*':=0'); *gen*(goto *nxq*+2); *gen*(*E.A*':=1');}

其中 θ 为关系运算符，*nxq* 表示将要生成但尚未生成的三地址码代码编号。

根据该翻译模式翻译如下句子：$x<y \vee s \leq t \wedge a$，假设 *nxq* = 100。

4.8 有以下翻译模式：

(1) $E \rightarrow E1 \vee ME2$ {*backpatch*(*E*1.*falselist*,*M.quad*);

E.truelist=merge(*E*1.*truelist*,*E*2.*truelist*); *E.falselist=E*2.*falselist*;}

(2) $E \rightarrow E1 \wedge ME2$ {*backpatch*(*E*1.*falselist*,*M.quad*);

E.falselist= merge(*E*1.*falselist*,*E*2.*falselist*); *E.truelist=E*2.*truelist*;}

(3) $M \rightarrow \varepsilon$ {*M.quad =nxq*;}

(4) $E \rightarrow \neg E1$ {*E.truelist=E*1.*falselist*; *E.falselist=E*1.*truelist*;}

(5) E→(*E*1) {*E.truelist=E*1.truelist; *E.falselist=E*1.*falselist*;}

(6) $E \rightarrow id1\theta id2$ {*E.truelist=mklist*(*nxq*); *E.falselist=mklist*(*nxq*+1);

gen('if' *ST.Get*(*id*1.*lexeme*)'*θ*'*ST.Get*(*id*2.*lexeme*) 'goto 0'); *gen*('goto 0');}

(7) $E \rightarrow id$ {*E.truelist=mklist*(*nxq*); *E.falselist=mklist*(*nxq*+1);

gen('if' *ST.Get*(*id.lexeme*)'goto 0'); *gen*('goto 0');}

(8) *S*→if *E* then *M*1 *S*1 *N* else *M*2 *S*2

{*backpatch*(*E.truelist*,*M*1.*quad*); *backpatch*(*E.falselist*,*M*2.*quad*);

S.nextlist=*merge*(*S*1.*nextlist*,*N.nextlist*,*S*2.*nextlist*);}

(9) *M*→ε {*M.quad*=*nxq*;}

(10) *N*→ε {*N.nextlist*=*mklist*(*nxq*);*gen*('goto 0');}

(11) *S*→if *E* then *M S*1

{*backpatch*(*E.truelist*,*M.quad*); *S.nextlist*=*merge*(*E.falselist*,*S*1.*nextlist*);}

(12) *S*→while *M*1 *E* do *M*2 *S*1

{*backpatch*(*S*1.*nextlist*,*M*1.*quad*);*backpatch*(*E.truelist*,*M*2.*quad*);

S.nextlist=*E.falselist*;*gen*('goto' *M*1.*quad*);}

(13) *S*→begin L end {*S.nextlist*=*L.nextlist*;}

(14) *S*→*A* {*S.nextlist*=mklist(); } // A 是赋值语句

(15) *L*→*L*1;*MS* {*backpatch*(*L*1.*nextlist*,*M.quad*); *L.nextlist*=*S.nextlist*;}

(16) *L*→*S* {*L.nextlist*=*S.nextlist*;}

根据该翻译模式翻译如下句子：if *x*<*y*∧*b* then *x*=*y*+*z*

4.9 根据第 4.8 题的翻译模式翻译如下句子：if *x*<*y*∧*b* then *x*=*y*+*z* else *x*=*y*-*z*

4.10 根据第 4.8 题的翻译模式翻译如下句子：while *x*<*y*∧*b* do *x*=*y*+*z*