★新世纪计算机类本科系列教材

编译原理基础

刘　坚　编著

西安电子科技大学出版社

2002

内 容 简 介

本书介绍程序设计语言和语言翻译的基本原理和技术，内容包括词法分析、语法分析、语义分析与中间代码生成、运行时的存储分配、以及目标代码的生成等。

本书可以作为工科院校计算机专业或非计算机专业的本科生教材，也可以作为软件技术人员或程序设计语言爱好者的参考书。

编译原理基础

刘 坚 编著

责任编辑 陈宇光 钟宏萍

出版发行 西安电子科技大学出版社(西安市太白南路 2 号)

电 话 (029)8227828 邮 编 710071

http: //www.xduph.com E-mail: xdupfxb@pub.xaonline.com

经 销 新华书店

印 刷 西安市第三印刷厂

版 次 2002 年 2 月第 1 版 2002 年 2 月第 1 次印刷

开 本 787 毫米×1092 毫米 1/16 印张 11.5

字 数 268 千字

印 数 1～4 000 册

定 价 14.00 元

ISBN 7-5606-1111-7/TP・0559

XDUP 1382001-1

前言

“编译原理”是国内高校计算机科学与技术专业的必修专业课之一，是一门理论与实践并重的课程，对引导学生进行科学思维和提高学生解决实际问题的能力有重要的作用。

“编译原理”课系统地介绍程序设计语言翻译的原理与技术，涉及的知识面比较广泛，国内大部分的编译教科书都存在越编越厚的现象，而由于授课时数的限制和学生接受能力的差异，教科书的内容往往并不能被充分利用，从而给学生带来不必要的经济负担。根据目前编译原理教学的实际情况，我们把“编译原理”课的内容分为基础篇和提高篇，其中基础篇的授课学时约为 50 学时，提高篇的授课学时约为 40 学时。本书是“编译原理”课的基础篇，供本科教学使用。我们编写的另一本教材《编译原理与技术》则是“编译原理”课的提高篇内容，可供研究生使用。

本书介绍程序设计语言翻译的基本原理与方法，全书分为六章。第一章引言，介绍有关程序设计语言和语言翻译的基本概念：高级语言与低级语言，编译与解释，编译器基本框架，构造编译器的方法与工具。第二章词法分析，从构词规则和词法分析两个方面讨论词法分析器的构造，内容包括：模式的描述与记号的识别，状态转换图与词法分析器，正规表达式与有限状态自动机。第三章语法分析，从原理上和方法上详细讨论了文法和不同的语法分析方法，内容包括：语法分析器在编译器中的位置和作用；上下文无关文法与上下文无关语言、文法的二义性及其消除；自上而下的 LL 分析和自下而上的 LR 分析。第四章语法制导翻译生成中间代码，介绍了语法制导翻译生成中间代码的一般方法，内容包括：语法与语义、属性与语义规则；中间代码的表现形式；名字信息的保存；声明性语句的语法制导翻译；可执行语句的语法制导翻译。第五章运行环境，内容包括：过程的动态特性、活动树与控制栈、名字的绑定；存储分配策略、栈式存储分配与非本地数据的访问。第六章代码生成，简单介绍了代码生成所需考虑的问题和在一个假想的计算机模型上如何生成基本块的目标代码。书中标有“*”的章节和习题是可选内容，可根据教学情况选择使用。

为配合编译教学的实施，本书作者提供由西安电子科技大学软件工程研究所开发的类 LEX/YACC 工具，XDCFLEX/XDYACC，其中的 XDCFLEX 可以接受部分中文描述。XDCFLEX/XDYACC 基于 C/C++，可分别运行在 PC 机的 DOS 和 Windows 环境，稍加修改，也可在其它环境，如 Unix 或 Linux 上运行。XDCFLEX/XDYACC 放

在西安电子科技大学的网站上边，具体下载地址如下：

http://www.xidian.edu.cn/soft/xdtools/xdtools.zip

或：ftp://ftp.xidian.edu.cn/soft/xdtools/xdtools.zip

本书的编写得到了西安电子科技大学研究生院和西安电子科技大学出版社的支持，龚杰民教授审阅了全书，郭强和张学敏等同学为 XDCFLEX/XDYACC 的研制开发作出了贡献，在此一并表示诚挚的谢意。

作者力图反映编译及其相关领域的基础知识与发展方向，并且力图用通俗的语言讲述抽象的原理。但是限于作者水平，书中难免有错误与欠妥之处，恳请读者批评指正。

作　者

2001 年 11 月

目　录

第1章　引　言

第2章　词 法 分 析

第3章 语 法 分 析

第 4 章　语法制导翻译生成中间代码

第5章 运行环境

第6章 代码生成

第1章 引 言

人类相互之间通过语言进行交流，人与计算机之间也通过语言进行交流。编译原理所讨论的问题，就是如何把符合人类思维方式的、用文字描述的意愿(源程序)翻译成计算机能够理解和执行的形式(目标程序)。具体实现从源程序到目标程序转换的程序，被称为编译程序或编译器。

1.1 从面向机器的语言到面向人类的语言

计算机的硬件只能识别由0、1字符串组成的机器指令序列，即机器指令程序。在计算机刚刚问世的年代，人们只能向计算机输入机器指令程序来指挥它进行简单的数学计算。机器指令程序是最基本的计算机语言。由于机器指令程序不易理解，用它编写程序既困难又容易出错，于是人们就用容易记忆的符号来代替0、1字符串。用符号表示的指令被称为汇编指令，汇编指令的集合被称为汇编语言，由汇编语言编写的指令序列被称为汇编语言程序。虽然汇编指令比机器指令在阅读和理解上有了长足进步，但是二者之间并无本质区别，它们均要求程序设计人员根据指令工作的方式思考、解决问题。因此，人们称这类语言为面向机器的语言或低级语言。

随着计算机应用需求的不断增长，人们希望能有功能更强、抽象级别更高的语言来支持程序设计，于是就产生了面向各类应用的程序设计语言。这些语言的共同特征是便于人类的理解与使用，因此被称为面向人类的语言或高级语言。表1.1列出了几种面向机器和面向人类的语言及其表现形式举例。

表1.1 面向机器和面向人类语言举例

分 类		语言表现形式举例
面向机器	机器指令	0000 0011 1111 0000
	汇编指令	add si, ax
面向人类	通用程序设计语言	x := a + b; sort(list); if c then a else b;
	数据查询语言	select id_no, name from student_table;
	形式化描述语言	E : E '+'E | E '*'E | id;

根据应用的不同，有着各种各样面向人类的高级语言，其中典型的有以下若干形式。

1. 通用程序设计语言

通用程序设计语言是继汇编语言之后发展起来的应用最广的一类语言，如人们常用的FORTRAN、Pascal、C/C++、Ada83/Ada95、Java等语言。这类语言的特征是：语言结构符合人类的思维特征，如直接使用表达式进行数学运算；具有很高抽象程度，如引入过程与

类等机制；程序设计中强调逻辑过程，即程序员要考虑事情的前因后果，不但要设计做什么，还要考虑怎么做，如条件或循环的判断等。

2. 数据查询语言

与通用程序设计语言相比，数据查询语言的抽象程度更高，它只要求程序员具有清晰的逻辑思维能力，设计好做什么，而忽略怎么做这样的实现细节，从而使得对大量复杂数据的处理变得轻松简单。

3. 形式化描述语言

形式化描述语言的代表之一是编译器构造中常用的工具 YACC 的语言。这类语言的核心部分是基于数学基础的产生式，设计人员只需利用产生式描述语言结构的文法，就可以构造出识别该语言结构的识别器。

4. 其他面向特定应用领域的语言

随着计算机应用领域的不断拓展，先后出现了多种面向特定应用领域的高级语言，如面向互联网应用的 HTML、XML，面向计算机辅助设计的 MATLAB，面向集成电路设计的 VHDL、Verilog，面向虚拟现实的 VRML 等等。这些形形色色、多不胜数的计算机语言推动了计算机应用的飞速发展，使得计算机成为人类生活中不可缺少的重要部分。

1.2 语言之间的翻译

尽管人类可以借助高级语言与计算机进行交往，但是计算机硬件真正能够识别的语言只是 0、1 组成的机器指令序列，这就需要在高级语言和机器语言之间建立若干桥梁，将高级语言逐步过渡到机器语言。换句话说，我们需要若干“翻译”，把人类懂的高级语言翻译成计算机懂的机器语言。由于应用的不同，语言之间的翻译是多种多样的。图 1.1 给出了一些常见的语言之间的翻译模式。在图 1.1 中，语言分为三个层次：高级语言、汇编语言、机器语言。虽然汇编语言和机器语言同属于低级语言，但是由于从汇编语言到可直接执行的机器指令之间也需要一层翻译，所以把它们分为不同的层次。设分别有两个高级语言 L1 和 L2，两个汇编语言 A1 和 A2，以及两个机器语言 M1 和 M2。高级语言之间的翻译，一般被称为**转换**，如 FORTRAN 到 Ada 的转换等，或者被称为**预处理**，如 SQL 到 C/C++的预处理等。高级语言可以直接翻译成机器语言，也可以翻译成汇编语言，这两个翻译过程被称为**编译**。从汇编语言到机器语言的翻译被称为**汇编**。高级语言是与具体计算机无关的，而汇编语言和机器语言均是与计算机有关的，因此，若将一个汇编语言汇编为可在另一机器上运行的机器指令，则称为**交叉汇编**，而建立在交叉汇编基础之上的编译模式，如首先将 L2 编译成 A2，再将 A2 汇编为 M1，有时也被称为**交叉编译**。上述这些翻译模式一般被认为是正向工程。在一些特定情况下需要逆向工程，如把机器语言翻译成汇编语言，或者把汇编语言翻译成高级语言，分别称它们为**反汇编**和**反编译**。值得一提的是，反编译是一件十分困难的事情。承担这些语言之间翻译任务的软件，一般被称为某某程序或某某器，为简单起见，本教材统一采用后一种方式，即将这些翻译软件称为转换器、编译器等。

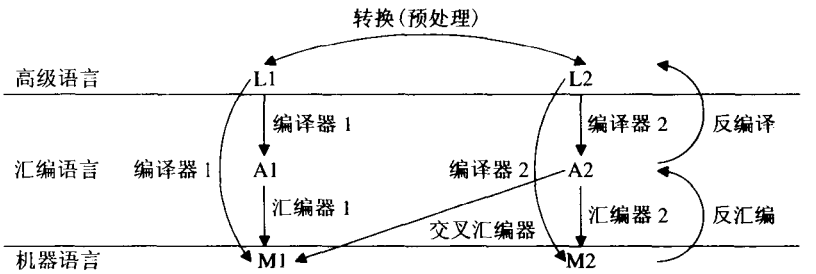


图 1.1 语言之间的翻译模式

上述语言之间的翻译虽然各不相同，但基本方法，特别是对源语言的分析方法是相同的。由于高级语言之间的转换和汇编语言到机器语言的翻译过程中，源程序和目标程序之间的结构变化不大，其处理方法相对编译器来讲一般比较简单，因此我们以编译器为例，讨论把高级语言中应用最广的通用程序设计语言翻译成汇编语言程序所涉及的基本原理、技术和方法。这些原理、技术和方法也同样适用于其他各类翻译器的构造，同时有些技术和方法也可以被用于其他软件设计。在后继讨论中，我们约定源程序是指通用程序设计语言程序，而目标程序是指汇编语言程序。

1.3 编译器与解释器

编译器(Compiler)一词是 Grace Murray Hopper 在 20 世纪 50 年代初提出来的，而被公认为最早的编译器是 50 年代末研制的 FORTRAN 编译器。

从用户的观点来看，编译器是一个黑盒子，如图 1.2(a)所示(为简明起见，图中忽略了对目标程序的汇编过程)。源程序的翻译和翻译后程序的运行是两个独立的不同阶段。首先是编译阶段，用户输入源程序，经过编译器的处理，生成目标程序。然后是目标程序的运行阶段，根据目标程序的要求进行适当的数据输入，最终得到运行结果。

解释器采用另一种方式翻译源程序。它不像编译器那样，把源程序的翻译和目标程序的运行分割开来，而是把翻译和运行结合在一起进行，翻译一段源程序，紧接着就执行它。这种方式被称为解释。在计算机应用中，凡是可以采用编译方式的地方，几乎都可以采用解释的方式，图 1.2(b)是一个解释器的工作模型。

假设有源程序：

```
read(x); write("x=",x);
```

则编译器的输入是此源程序。目标程序的输入如果是 3，则输出是 x=3。而对于解释器，则输入端既包括上述源程序，又包括 3，其输出同样是 x=3。

可以看出，编译器的工作相当于在翻译一本原著，计算机运行编译后的目标程序，相当于阅读一本译著，原著(或原作者)和译著者并不在场，主角是译著。而解释器的工作相当于在进行同声翻译，计算机运行解释器，相当于我们直接通过翻译听外宾讲话，外宾和翻译均需到场，主角是翻译。

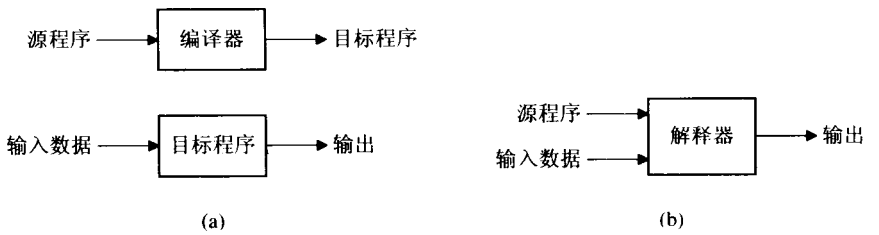


图 1.2 编译器与解释器工作方式的对比

(a) 编译器的工作方式；(b) 解释器的工作方式

解释器与编译器的主要区别在于：运行目标程序时的控制权在解释器而不在目标程序。因此，与编译器相比，解释器有以下两个优点:

(1) 具有较好的动态特性: 运行时，由于源程序也参与其中，因此数据对象的类型可以动态改变，并允许用户对源程序进行修改，且可提供较好的出错诊断，从而为用户提供了交互式的跟踪调试功能。

(2) 具有较好的可移植性: 解释器一般也是用某种程序设计语言编写的，因此，只要对解释器进行重新编译，就可以使解释器运行在不同的环境中。

由于解释器的动态特性和可移植特性，在有些特定的应用中必须采用解释的方法。典型的例子是数据库系统中的动态查询语句和 Java 的字节代码。前者利用了解释器的动态特性，在程序运行时根据输入动态生成查询语句，然后解释执行。后者利用了解释器的可移植特性，可在任何机器上对字节代码进行解释执行，习惯上称之为 Java 虚拟机。

但是，由于解释器是把源程序的翻译和目标程序的运行过程结合在一起，因此，与编译器相比，它在运行时间和空间上的损失较大，运行效率低：

(1) 时间上: 在运行过程中，解释器要时刻检查源程序。例如，每一次引用变量，都要进行类型检查，甚至需要重新进行存贮分配，从而大大降低了程序的运行速度。用早期 BASIC 编写的源程序，编译后运行和解释执行的时间比约为 1:10。

(2) 空间上：执行解释时，不但要有用户程序的运行空间，而且解释器和相应的运行支撑系统也要占据内存空间。

由于编译和解释的方法各有特点，因此，现有的一些编译系统既提供编译的方式，也提供解释的方式，或者是一种中和的方式。例如在 Java 虚拟机上发展的一种新技术，称为 compiling-just-in-time，它的基本思想是，当一段代码第一次运行时，首先对它进行编译，而在其后的运行中不再进行编译。这种方法特别适合一段代码多次运行的情况，而对于大多数代码仅运行一次的情况并不适用。

从翻译的角度来讲，两种方式所涉及的基本原理、方法与技术是相似的。

1.4 编译器的工作原理与基本组成

1.4.1 通用程序设计语言的主要成份

通用程序设计语言的典型特征之一是抽象，其抽象程度是以程序设计语言所支持的基

本结构为特征的，可以大致划分为三种形式：过程、模块（抽象数据类型，ADT）和类。以过程为基本结构的程序设计语言的典型代表有 C、Pascal 等；以 ADT 为基本结构的程序设计语言的典型代表是 Ada83；而以类为基本结构的程序设计语言包括当前流行的 C++、Java 和 Ada95 等。这三种形式经过了一个演变过程，每一次演变都使得程序设计语言的抽象程度得到一次提高，同时也为这些程序设计语言的编译器提出了新的要求。

类概念的引入，为利用程序设计语言构造类型提供了真正的支持，也是面向对象程序设计语言的重要特征之一。程序设计语言提供的机制与程序设计的风格有着密切关系，以过程为基本抽象的程序设计语言支持的是过程式的程序设计范型（paradigm），以类为基本抽象的程序设计语言支持的是面向对象的程序设计范型，以 ADT 为基本抽象的程序设计语言介于二者之间，一般被认为是面向过程的语言，但也被认为是基于对象的语言。有些面向对象的程序设计语言是由过程式的语言发展而来的，如 C++、Ada95 等，它们实质上是支持多范型的程序设计语言。

由于篇幅和授课时间所限，后继章节均以最简单的、以过程为基本结构的程序设计语言为背景进行讨论。因为无论何种形式的程序设计语言，均是由声明和操作这样两个基本元素构成的，所不同的是声明和操作的范围和复杂程度不同。

以过程为基本结构的程序设计语言的特征是把整个程序作为一个过程。过程的定义由两类语句组成：声明性语句和操作性语句。一般来讲，声明性语句提供所操作对象的性质，如数据类型、值、作用域等。而操作性语句确定操作的计算次序，完成实际操作。过程由过程头和过程体两个部分组成，对应的声明性和操作性语句用例 1.1 加以说明。

例 1.1 有一 Pascal 的过程如下所示：

```
(1)  procedure sample(y: integer);
(2)    var x : integer;
(3)    begin x := y;
(4)          if x>100 then x :=0
(5)    end;
```

(1)是过程头，它是一个声明性的语句，为使用者提供调用信息，包括过程名、参数、返回值(如果有的话)等。

(2)~(5)是过程体，它是一个语句序列，语句序列中既包括声明性语句也包括操作性语句。(2)是声明性语句，而(3)~(5)是操作性语句。对于编译器来讲，它对声明性语句的处理一般是生成相应的环境(存储空间)，而对操作性语句则是生成此环境中的可执行代码序列。为了便于编译器的处理，操作性语句中使用的每个操作对象，均应在使用前进行声明，即遵循先声明后引用的原则。

■

1.4.2 以阶段划分编译器

对于自然语言(如英语)的翻译，经历这样几个主要阶段：识别单词、识别句子、理解意思、译成中文并对译文进行合理的修饰。编译器对于计算机语言的翻译，也同样需要经历这样几个阶段：首先进行词法分析，识别出合法的单词；其次进行语法分析，得到由单词组

成的句子结构；然后进行语义分析，并且生成目标程序。为了使翻译工作更好地进行，编译器往往在语义分析之后先生成所谓的中间代码，并且可以对中间代码进行优化，最后从优化后的中间代码生成目标程序。每个阶段的工作在逻辑上由图 1.3 中的一个程序模块承担，其中的符号表管理器和出错处理器贯穿编译器各个阶段，为了统一，也把它们称为编译的两个阶段。

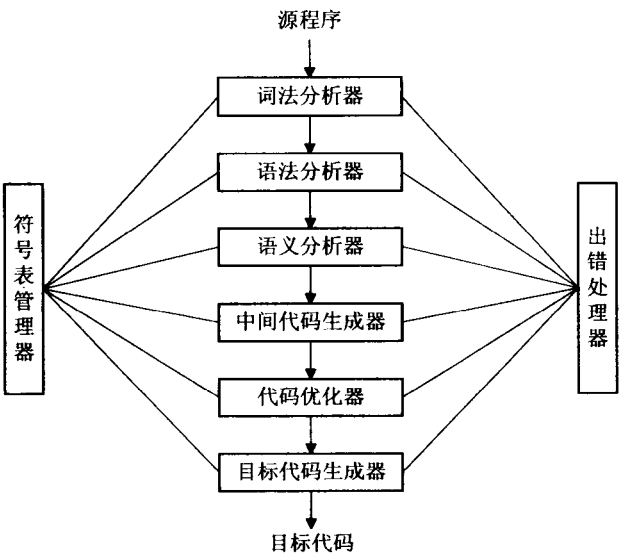


图 1.3 编译器的阶段

1.4.3 编译器各阶段的工作

我们以仅包含一条声明语句和一条可执行语句的 Pascal 源程序为例，说明编译器各个阶段处理的全过程。例中每个前一阶段的输出是后一阶段的输入。为了便于理解，叙述采用的是逻辑的和示意性的方法，其中表示变量名称的标识符用 id1、id2、id3 表示，目的是强调标识符的内部表示与输入序列的区别，而程序中的关键字和特殊符号以及像 60 这样的数字字面量等，均采用外部原来的表示，目的是为了直观。

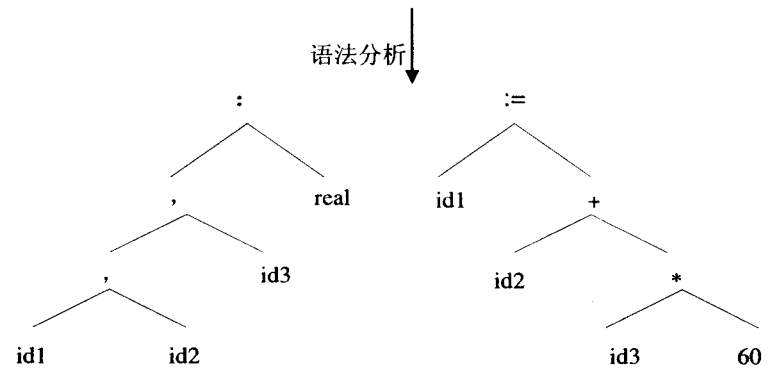
例 1.2 有一 Pascal 源程序语句如下所示：

```
var x, y, z : real;
x := y + z * 60;
```

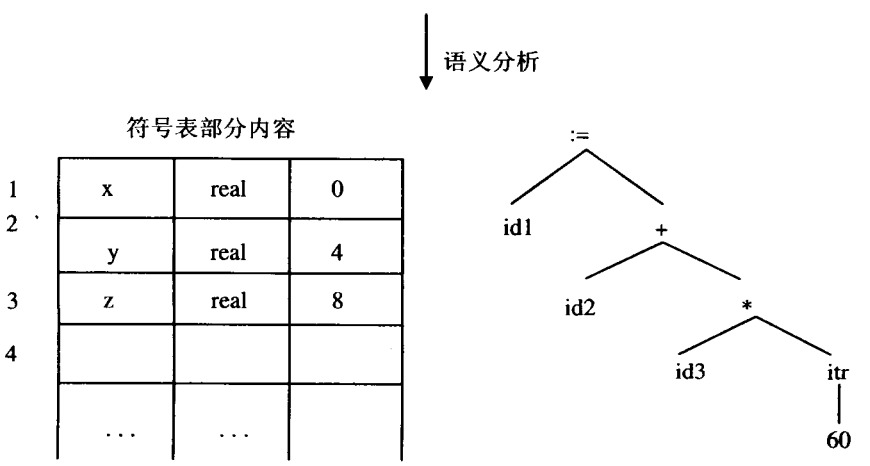
编译器从左到右扫描输入，首先进行的是词法分析。词法分析器的输入是源程序，输出是识别出的记号流。

```
var x, y, z : real;   x := y + z * 60;
                  ↓ 词法分析
var id1, id2, id3 : real;   id1 := id2 + id3 * 60;
```

语法分析器以词法分析器返回的记号流为输入构造句子的结构，并以树的形式表示出来，称之为语法树。



语义分析器根据语法分析器构造的语法树，进行适当的语义处理。对于声明语句，进行符号表的查填。下述符号表部分的内容中，每一行存放一个符号的信息，第一行存放标识符 x 的信息，它的类型是 real，为它分配的地址是 0。第二行存放 y 的信息，它的类型是 real，为它分配的地址是 4。由此可知，我们为每个实型数分配一个大小为四个单位的存储空间。对于可执行语句，检查结构合理的表达式运算是否有意义。由于变量 x，y，z 均是 real，而 60 被认为是 integer，因此，语义检查时需要进行把 60 转换为 60.0 的处理。反映在语法树上，就是增加了一个新节点 itr (将整型数转换为实型数)。



由于声明语句并不生成可执行的代码，所以到此为止，对声明语句的处理已经完成。下边开始的中间代码生成，仅涉及源程序中的赋值句。中间代码生成器对语法树进行遍历，并生成可以顺序执行的中间代码序列。最常用的中间代码形式是四元式，它的基本形式为:

(序号)　(op，　arg1，　arg2，　result)

操作符　左操作数　右操作数　结果

操作符也被称为算符，操作数也被称为算子。上式表示第(序号)个四元式，arg1 和 arg2 进行 op 运算，结果存进 result。如四元式 (+，x，y，T)表示的运算为 T := x + y。而四元式 (:= ，x，　，T)表示的运算为 T := x。为了表示上的直观，有时也把四元式直接表示为 T := x + y

和 T := x 的形式。这似乎与程序设计语言中的表达式在表示上没有什么区别，因此有时需要根据上下文来确定是算术表达式还是四元式。另外，四元式的一个特征是赋值号右边最多只有一个操作符和两个操作数。


```
(1) (itr,   60,     ,   T1)
(2) (*,     id3,   T1,   T2)
(3) (+,     id2,   T2,   T3)
(4) (:=,    T3,    ,    id1)
```

下一步工作就可以对中间代码进行优化了。分析上边的 4 个四元式可以看出，60 是编译时已经知道的常数，所以把它转换成 60.0 的工作可以在编译时完成，没有必要生成(1)号四元式。再看(4)号四元式，它的作用仅是把 T3 的值传给 id1(这样的运算被称为复写传播)，不难看出，这条四元式也是多余的。经过优化后，4 个四元式减少为两个。


```
(1) (*,    id3,   60.0,    T1)
(2) (+,    id2,   T1,      id1)
```

最后从优化后的中间代码生成目标代码。这里的目标代码是汇编指令，其中 MOVF、MULF 和 ADDF 分别表示浮点数的传送、乘和加操作。对于二元运算 MULF 和 ADDF，操作形式为 OP source，target，它表示 target := source OP target，即 sorce 与 target 进行 OP 运算，结果存进 target。对于一元运算 MOVF，操作形式为 MOVF source，target，它表示 target := source，即将 source 中的内容移进 target 中。


```
MOVF    id3,    R2
MULF    #60.0,  R2
MOVF    id2,    R1
ADDF    R2,     R1
MOVF    R1,     id1
```

归纳上述结果，我们把编译器的各阶段工作总结如下。

■

1. 词法分析

词法分析器根据词法规则识别出源程序中的各个记号(token)，每个记号代表一类单词(lexeme)。源程序中常见的记号可以归为以下几大类，其中每一类均可再细分。

(1) **关键字**: 如 var、begin、end ...，它们在源程序中均有特定含义，一般不作它用，在这种情况下也被称为**保留字**。

(2) **标识符**: 如 x、y、z、sort ...，它们在源程序中被用作变量名、过程名、类型名和标号等所有对象的名称。

(3) **字面量**: 如 60、"Xidian University" ...，一般表示常数或字符串常量，它们也可以被细分为数字字面量、字符串字面量等。

(4) **特殊符号**: 如":="、"+"、";" ...，它们在源程序中均有特定含义，根据它们的作用，也可以被细分为运算符、分隔符等。

2. **语法分析**

语法分析器根据语法规则识别出记号流中的结构(短语、句子等)，并构造一棵能够正确反映该结构的语法树。以后我们会看到，除了反映语言结构外，有些语法树也反映语法分析的关键步骤。因此，语法树可以是隐含的，也可以确有其“树”。语法树的数据结构一般采用典型的二叉树结构，因为任何形态的树均可以转化为二叉树。

3. **语义分析**

语义分析器根据语义规则对语法树中的语法单元进行静态语义检查，如类型检查和转换等，其目的在于保证语法正确的结构在语义上也是合法的。

当分析到声明语句时，语义分析器将相应的环境信息记录在符号表中，以便在后继操作语句中使用。如例 1.2 中的三个变量都是 real 类型。而 60 被默认为 integer 类型。不同类型的数所占用的存贮空间不同，例如 real 类型占用 4 个存贮单元，则三个变量被分配的地址分别为 0、4、8。

当分析到操作性语句时，可以根据符号表中的信息判断各操作数是否合法，由于三个变量均为 real，而 60 是 integer 类型，因此，此时的语义分析要增加一个操作 itr，把 60 转换成 60.0。

4. **中间代码生成**

中间代码生成器根据语义分析器的输出生成中间代码。中间代码可以有若干种形式，它们的共同特征是与具体机器无关。最常用的一种中间代码是三地址码，它的一种实现方式是四元式。三地址码的优点是便于阅读、便于优化。

值得一提的是，无论是对于解释器还是编译器，到中间代码生成以前的各阶段(即完成语义分析)是完全一样的。语义分析完成以后，语法树已经形成，执行计算的基本元素已经具备，因此，对于解释器来讲，此时就可以直接形成计算步骤并且进行计算，没有必要再做中间代码生成和其后的工作。或者，解释器在语义分析完成以后，生成某种中间代码，统一对此中间代码进行解释执行。由于语法树和中间代码均不依赖于任何机器，因此解释器是可移植的。典型的例子是 Java 字节代码与 Java 虚拟机。

5. **中间代码优化**

优化是编译器的一个重要组成部分，由于编译器将源程序翻译成中间代码的工作是机械的、按固定模式进行的，因此，生成的中间代码往往在时间上和空间上有很大浪费。当需要生成高效目标代码时，就必须进行优化。

优化过程可以在中间代码生成阶段进行，也可以在目标代码生成阶段进行。由于中间代码是不依赖于机器的，在中间代码一级考虑优化可以避开与机器有关的因素，把精力集中在对控制流和数据流的分析上。因此，优化的大部分工作在目标代码生成之前进行，只