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。

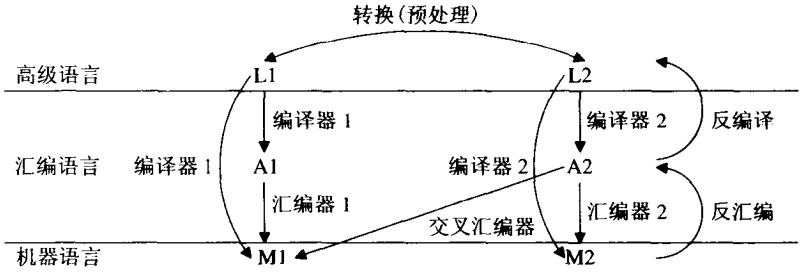


图 1.1 语言之间的翻译模式

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。

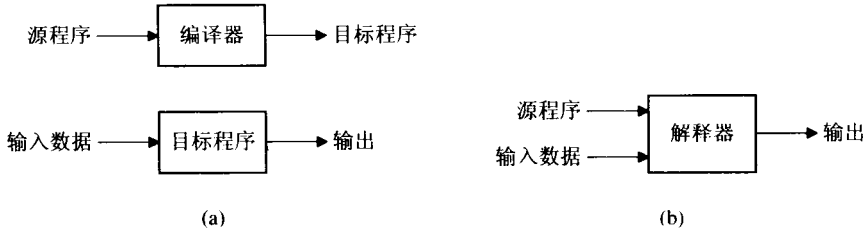


图 1.2 编译器与解释器工作方式的对比

(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

.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。

■

.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。

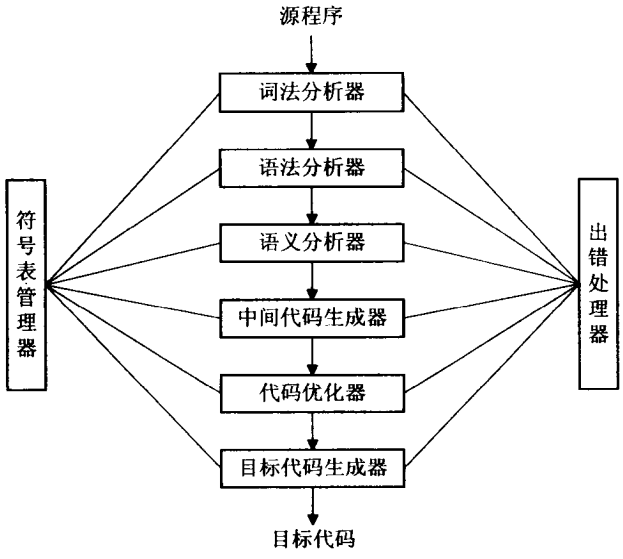


图 1.3 编译器的阶段

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。

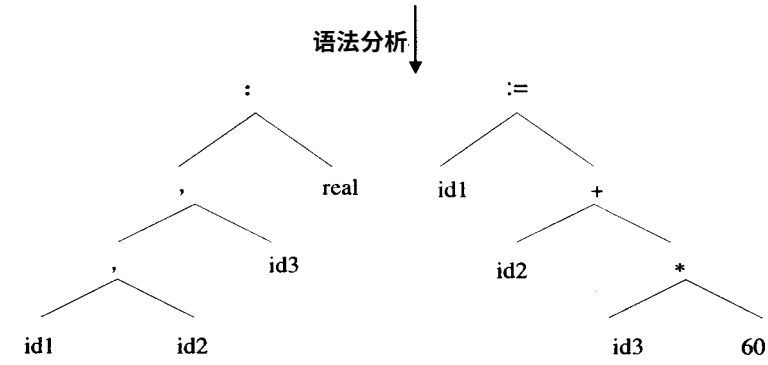
例 1.2 有一 Pascal 源程序语句如下所示：

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

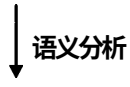
编译器从左到右扫描输入，首先进行的是词法分析。词法分析器的输入是源程序，输出是识别出的记号流。

```
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。

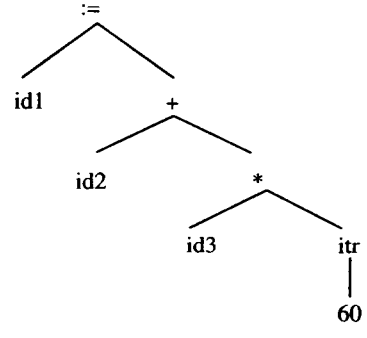


语义分析器根据语法分析器构造的语法树，进行适当的语义处理。对于声明语句，进行符号表的查填。下述符号表部分的内容中，每一行存放一个符号的信息，第一行存放标识符 x 的信息，它的类型是 real，为它分配的地址是 0。第二行存放 y 的信息，它的类型是 real，为它分配的地址是 4。由此可知，我们为每个实型数分配一个大小为四个单位的存储空间。对于可执行语句，检查结构合理的表达式运算是否有意义。由于变量 x，y，z 均是 real，而 60 被认为是 integer，因此，语义检查时需要进行把 60 转换为 60.0 的处理。反映在语法树上，就是增加了一个新节点 itr(将整型数转换为实型数)。



符号表部分内容

1	x	real	0
2	y	real	4
3	z	real	8
4			
	...	...	



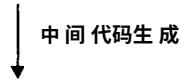
由于声明语句并不生成可执行的代码，所以到此为止，对声明语句的处理已经完成。下边开始的中间代码生成，仅涉及源程序中的赋值句。中间代码生成器对语法树进行遍历，并生成可以顺序执行的中间代码序列。最常用的中间代码形式是四元式，它的基本形式为：

(序号) (op, arg1, arg2, result)

操作符 左操作数 右操作数 结果

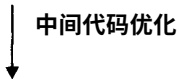
操作符也被称为算符，操作数也被称为算子。上式表示第（序号）个四元式，arg1 和 arg2 进行 op 运算，结果存进 result。如四元式 (+, x, y, T)表示的运算为 T := x + y。而四元式 (:=, x, , T)表示的运算为 T := x。为了表示上的直观，有时也把四元式直接表示为 T := x + y

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。



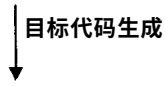

```
(1) (itr,   60,      ,   T1)
(2) (*,     id3,   T1,   T2)
(3) (+,     id2,   T2,   T3)
(4) (:=,    T3,      ,   id1)
```

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

有少部分与机器有关的优化（如局部的优化或寄存器的分配等）工作放在目标代码生成时进行。

优化实际上是一个等价变换，变换前后的指令序列完成同样的功能，但是，优化后的代码序列在占用的空间上和程序执行的时间上都更节省、更有效。

6. 目标代码生成

目标代码生成是编译器的最后一个阶段。在生成目标代码时要考虑以下几个问题：计算机的系统结构、指令系统、寄存器的分配以及内存的组织等。

编译器生成的目标程序代码可以有多种形式。

(1) 汇编语言形式(**Assembly Language Format**): 编译器生成汇编语言形式的代码序列。一般来讲，生成汇编指令代码比生成二进制代码序列在处理上要简单且易读，而且，由于汇编语言仍然是符号形式的，所以特别便于实现交叉编译。它的弱点是编译之后还要经过一次汇编。

(2) 可重定位二进制代码形式(**Relocatable Binary Format**): 这实际上是编译器常采用的一种目标代码。编译器生成二进制代码模块，模块内地址以模块首地址相对寻址，经过链接程序进行链接。链接时还需把程序中所引用的预定义标准例程和其它已编译过的模块包括进来，最后形成一个可直接运行的代码序列。

(3) 内存形式(**Memory-Image Format**): 编译器生成的代码序列直接被装入原编译器所在的位置并被立即执行，反映在外部也就是编译后马上运行。这类形式在英文中也被称为Load-and-Go。由于这种形式不生成以文件形式存放在磁盘上的目标代码，也没有被链接的过程，因而这种形式特别适合初学者或在程序的调试阶段使用。它的弱点是运行一次就需要编译一次。

由于这三种形式各有其它形式无法替代的特点，有些编译器同时提供这三种或者其中两种形式，用户可以根据需要选择使用。

7. 符号表管理

符号表的作用是记录源程序中符号的必要信息，并加以合理组织，从而在编译器的各个阶段能对它们进行快速、准确的查找和操作。符号表中的某些内容甚至要保留到程序的运行阶段。

8. 出错处理

由于例 1.2 中给出的是一个没有错误的源程序，因而出错处理是一个还未涉及的阶段。但是，用户编写的源程序中往往会有一些错误，这些错误大致被分为静态错误和动态错误两类。所谓动态错误，是指源程序中的逻辑错误，它们发生在程序运行的时候，也被称为动态语义错误，如变量取值为零时被作为除数，数组元素引用时下标出界等。静态错误又可分为语法错误和静态语义错误。语法错误是指有关语言结构上的错误，如单词拼写错、表达式中缺少操作数、 begin 和 end 不匹配等。静态语义错误是指分析源程序时可以发现的语言意义上的错误，如加法的两个操作数中一个是整型变量名，而另一个是数组名等。

静态错误应该在编译的不同阶段被检查出来，并且采用适当的策略修复它们，使得分析过程能够继续下去，直到源程序的结束。遇到一个错误就使编译器停止工作的做法是不负责任的，也是用户难以接受的

1.4.4 编译器的分析／综合模式

对于编译器的各个阶段，逻辑上可以把它们划分为两个部分，即分析部分和综合部分。从词法分析到中间代码生成各阶段的工作称为分析，而以后直到目标代码生成各阶段的工作被称为综合。分析部分也被称为编译器的前端，综合部分也被称为编译器的后端。图 1.4 所示是一个理想的分析／综合模式。在这里，中间代码起了分水岭的作用，由于中间代码是与机器无关的，因此它把编译器分成了与机器有关和无关的两部分，从而提高了编译器开发和维护的效率。例如，对于一种程序设计语言，可以开发一个共同的前端，再针对不同的机器设计不同的后端，并且语言结构的修改往往只涉及前端的维护。也可以针对某一机器开发一个后端，而对于不同的语言设计各自的前端，生成同一种中间代码，从而得到一个机器上的若干个编译器。当然，由于语言之间的差异，这些想法在实现上还存在着许多困难。另外，编译器和解释器的区别，也往往是形成中间代码之后开始：编译器从中间代码生成目标代码，而解释器解释中间代码得到运行结果。值得注意的是，编译器和解释器所需的中间代码形式可能有所不同。

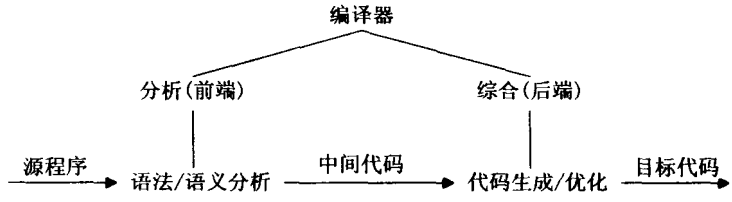


图 1.4 编译器的分析／综合模式

.4.5 编译器扫描的遍数

在图 1.3 所示的编译器模型中，编译器的每个阶段都是对以某种形式表示的完整程序进行一遍分析。我们把每个阶段将程序完整分析一遍的工作模式称为一遍扫描。例如，词法分析器对输入源程序进行第一遍扫描，把源程序分解成一串记号流，并进行必要的符号登记工作（由符号表管理器处理）。语法分析器进行第二遍扫描，它以词法分析器输出的记号流为输入，识别出语言结构，如赋值语句、过程定义等，并建立和输出对应的语法树。依此类推，最后生成目标程序。但是，这样一个阶段对应一遍扫描的工作方式，只是逻辑上的。由于多次扫描的方式需要大量的存储空间存放中间表示，并且也会增加一些不必要的输入输出操作，因此，编译器往往是把若干个阶段的工作结合起来，对应一遍扫描，从而减少对程序的扫描遍数。原理上希望扫描的遍数越少越好，这就必须保证两点：

(1) 为编译器的运行提供足够大的空间。由于若干阶段的工作合并在一遍中完成，所以处理各阶段工作的程序都随时准备运行，而且各阶段所需的信息也要同时放在内存中。随着计算机硬件技术的发展，这已不成为问题。

(2) 在语言的设计上和编译技术上为减少扫描遍数提供支持。在语言设计上，尽量使得编译器可以仅从已扫描过的内容就得到足够的信息。例如，许多程序设计语言都要求对标识符先声明后引用，这就保证了任何一个标识符出现时就可以确定它的性质，而不需要扫描标识符以后的程序部分。另外，也可以采用一些专门技术来达到类似目的。最典型的例

子是转移语句的翻译。大多数程序设计语言允许向前转移的 goto 语句，而遇到 goto 时，其具体转向并不知道，因而无法确定此语句的转向地址。对这种情况，可以采用一种称为“拉链／回填”的技术，把生成的转移指令中确定不了的转移地址先暂时空起，等到地址确定后再回填进去。

虽然从编译器工作效率的角度讲，一遍扫描是最好的。但是，由于各种原因，若干遍扫描也是不可少的。例如，由于中间代码界定了前端和后端，并且两个部分的工作有很大区别，因此，往往至少将前端作为一遍扫描。另外，为了生成高质量的目标代码，需要对中间代码进行优化，而全局性的程序流和数据流分析也应该是对中间代码的一遍扫描。总之，对一个具体的编译器，要确定用几遍扫描来完成，需要综合考虑各种因素，从中取得最佳折中。

1.5 编译器的编写

编译器本身也是一个程序，那么用什么编写编译器呢？早期人们用汇编语言编写编译器。众所周知，人工可以编写出效率很高的程序，但由于编译器本身是一个十分复杂的系统(如早期的 FORTRAN 用了 18 人年才完成），而用汇编语言编写编译器的效率很低，往往给实现带来很大困难。因此，除了特别需要，人们早已不再用汇编语言编写完整的编译器。现在常用通用程序设计语言编写编译器，它的效率比汇编语言要高得多。不过，用单纯程序设计的方法来对付编译器这样的庞然大物也显得不够。为此，需要一些专门的编译器编写工具来支持编译器某些部分的自动生成。比较成熟和通用的工具有词法分析器生成器和语法分析器生成器，如被广泛应用的 LEX 和 YACC。另外，还有一些工具，如语法制导翻译工具(用于语义分析)、自动的代码生成器(用于中间代码生成和目标代码生成)和数据流工具（用于优化）等。这些工具的共同特点是，仅需要对语言相应部分的特征进行描述，而把生成算法的过程隐蔽起来，同时所生成的部分可以很容易地并入到编译器的其它部分中。因此，这些工具往往与某程序设计语言联系在一起，如与 LEX 和 YACC 联系的程序设计语言是 C。

1.6 本 章 小 结

编译原理是一门理论和实践并重的课程，大部分同学都会感到学习这门课程十分困难。关键问题是应该掌握好的学习方法，在此我们强调两点：

①牢固掌握基本概念，这要进行大量的阅读，并通过阅读加深理解：

②灵活使用基本方法，这要在阅读理解的基础上做好习题和上机作业。

做到这两点，学好这门课程就不会成为难事，正所谓“难者不会，会者不难”。

本章介绍了有关程序设计语言和编译器的以下几个重要概念。

(1) 语言的翻译：

①面向人类的高级语言，如通用程序设计语言 Pascal、C/C++、Java、Ada 以及一些有特定应用领域的语言等；

②面向机器的低级语言，如汇编语言和二进制机器代码等；

③编译器与汇编器，把高级语言翻译成低级语言的程序被称为编译器（或解释器），把汇编语言翻译成机器代码的程序被称为汇编器。编译器与解释器，编译器首先把源代码翻译成目标代码，然后执行目标代码，解释器一边翻译源代码，一边执行解释后的代码。

(2）编译器的基本组成：以阶段划分编译器，阶段包括词法分析、语法分析、语义分析、中间代码生成、中间代码优化、目标代码生成、符号表管理以及出错处理。

(3) 编译器的分析一综合模式：把编译器分为前端和后端。前端称为分析，它的输出与机器无关；后端称为综合，以前端的输出为输入，其输出与具体机器指令密切相关。编译器的这种划分方式，有利于编译器的开发、维护与移植。

(4）编译器的扫描遍数：对程序（源程序、中间表示等）的一次完整的扫描称为一遍。影响扫描遍数的因素是多样的，减少扫描遍数的思路也是多样的。

(5) 编译器的编写工具：特别需要了解的是词法分析器和语法分析器的编写工具。

习　题

1.1　列举出你所使用过的所有计算机语言和所有的‘ 翻译 ’程序(编译、解释、汇编等)。

1.2　如果在 Pascal 源程序中出现这样一些情况：12x　2*/3　3.5+"end"　x/y（运行时 y=0 ）请指出它们分别是什么类型的错误。

1.3　从你使用过的编译器中选择一个最熟悉的，写出从编写到运行一个应用程序的全过程。

第2章 词法分析

在我们生活的社会中，为了维持社会的正常运转，必须制定各种法律并且有相应的执法机构保证法律的贯彻执行。程序设计语言的基本元素——单词的集合，也是这样的一个“社会”，必须为这一集合制定法律和设立相应的检查和执行机构。因此，词法分析在此处具有双重含义：

(1) 规定单词形成的规则，也被称为构词规则或词法规则。它的作用相当于立法，即规定什么样的输入序列是语言所允许的合法单词。

(2) 根据构词规则识别输入序列，也被称为词法分析。它的作用相当于执法，即根据规则识别出合法的单词和指出非法的输入序列。

本节首先简单介绍若干与词法分析有关的基本概念和相关问题，然后对单词形成的规则和根据这些规则构造词法分析器的方法进行理论上和方法上的详细讨论。

2.1 词法分析中的若干问题

2.1.1 记号、模式与单词

自然语言中的句子通常由一个个单词和标点符号组成，可以根据其在句子中的作用，将它们划分为动词、名词、形容词、标点符号等不同的种类。程序设计语言与此相类似，组成语句的基本单元也可根据其在句子中的作用分类。最基本分类有四类：

(1)关键字(保留字)这类单词在程序设计语言中有固定的意义,如 begin、end、while 等。若在程序设计语言中不允许用它们再表示其他的意思，则这类单词也被称为保留字。

(2) 标识符：标识符是程序设计语言中最大的一个类别，它的作用是为某个实体起一个名字,以便于今后称呼(引用),如 draw_line、 sort 等。可以用标识符来命名的实体包括类型、变量、过程、常量、类、对象、程序包、标号等，即类型名、变量名、过程名、常量名等。

(3) 字面量：字面量是指直接以其字面所表示的常量，如 25、true、"This is a string" 等。值得注意的是，字面量与常量是两个不同的概念，常量可以是一个字面量（直接表示），也可以是一个常量名(命名表示)。例如可以在 Pascal 中声明： const max_length = 25，显然 25 是一个常量， max_length 也是一个常量，我们称 25 为字面量，而不称 max_length 为字面量。根据字面量的内容，可以将它们再进行更细的划分，如常数字面量（包括整型字面量、实型

字面量、枚举字面量等）、字符串字面量等。

(4) 特殊符号：程序设计语言中的特殊符号，类似于自然语言中的标点符号，每个符号在程序设计语言中均有特殊用途。可以根据它们的用途，再细分为算符（如+、一、*、／等）、分隔符(如；”“等)。

显然，一个单词究竟是标识符、关键字，还是特殊符号，需要根据一定的构词规则来产生和识别。我们将产生和识别单词的规则称为模式（patten),按照某个模式(规则)识别出的元素称为记号(**token**)，而单词(lexeme)一词是指被识别出元素自身的值。

例 2.1　对于语句：position := initial + rate * 60，我们可以识别出下述序列：

标识符　特殊符号　标识符　特殊符号　标识符　特殊符号　数字字面量

其中 position、initial、rate 均被识别为标识符，因为它们均符合同一条规则，即以字母打头的字母数字串。记号至少含有两个信息：一个是记号的类别，如“标识符”；另一个是记号的值，如‘position”。显然，如果把记号看作是一个类型的话，则单词就是一个类型中的实例。由于我们总是说识别出一个标识符，而不说识别出一个 position 或 rate，因而将词法分析器识别出的序列称为记号流。

■

记号的类别、模式以及单词三者之间的关系可以用表 2.1 加以说明。其中，const 和 if 分别是被细分的关键字，它们的特点是一个记号类别仅对应一个单词； relation 表示关系运算符；id 表示标识符；num 表示数字字面量；literal 表示字符串字面量；comment 表示注释，它们的特点是一个记号类别可以对应若干个单词。由于语法分析及其后的阶段并不对注释进行分析，因而可在词法分析阶段中滤掉注释，即词法分析器可以不向语法分析器返回 comment。而其他的记号，均是源程序中的有效成分，需要返回给语法分析器。

表 2.1　记号、模式与单词

记号的类别	单词举例	模式的非形式描述
const(01)	const	const
if(03)	if	if
relation(81)	<，<=，=，<>，>，>=	<或<=或=...
id(82)	Pi，count, D2	字母打头的字母数字串
num(83)	3.1416, 0, 6.02E23	任何数值常数
literal(84)	"core dumped"	双引号之间的任意字符串
comment	{x is an integer}	括号之间的任意字符串

2.1.2　记号的属性

从例 2.1 中已经知道，记号至少包含两个部分：记号类别和记号的其他信息。可以看出，记号的类别唯一标识一类记号，例如所有的关系运算符均可以由 relation 来标识，而所有字符串字面量均可以由 literal 来标识。所以，记号的类别可以被认为是记号的名字或记号的代表，在不引起混淆的情况下，将记号的类别简称为记号。记号的其他信息被称为记号的属性。例如，num 可以取值 3.1416,则称 3.1416是 num 的属性,而 literal 可以取值“core dumped”（不含引号),则称“ core dumped ”是 literal 的属性。由此可见，记号的类别标识一类记号，

而记号的类别加属性标识一个记号实例。

在计算机内部，可以有不同的方式来表示记号的类别和属性。一般情况下，记号的类别可以用整型编码或枚举类型表示，如表 2.1 中每个记号类别可以用括号中的整型编码表示，如 01 表示 const，82 表示 id 等。根据记号类别的不同，记号的属性的值可以有不同的表示方法。relation 的属性值是一个有限可枚举集合，可以用每个属性值在集合中的位置来表示它，如 1 表示<，2 表示<=，依此类推。id 的属性值是一个无限可枚举集合，因此，只能用每个标识符的原始输入形式(字符串)来表示,如 pi、draw_line 等。字面量的属性根据情况，其表示方式也不同，如数字字面量可由转义后的实际值表示，如表示为 3.1416 而不是“ 3.1416”，而字符串字面量就无需转义。

例 2.2　表达式 mycount > 25 由表 2.2 的三个记号组成。其中标识符的属性值也可以由 mycount 在符号表中的入口(下标)来表示。

■

表 2.2　记号的表示

记号的类别	记号的属性
82	“mycount”
81	5
83	25

2.1.3　词法分析器的作用与工作方式

词法分析器是编译器中唯一与源程序打交道的部分，从某种意义说，也可以被认为是整个编译器的预处理器。它的主要工作包括：

(1) 滤掉源程序中的无用成分，如注释、空格、回车等。例如，表 2.1 中记号的种类除了 comment 之外，均有一个编码，表示需要递交给语法分析器进行后继的处理，而 comment 没有对应编码，表示注释成分可以过滤掉，不需要递交，因为语法分析及其之后的各个阶段已经不再需要它们。

(2) 处理与具体平台有关的输入。不同的操作系统或相关软件构成的平台，对某些特殊符号（如文件结束符等）可能有不同表示，因此需要在词法分析阶段分情况处理。

(3) 识别记号，并交给语法分析器。这是词法分析器的主要任务，本章将在各节中详细讨论。

(4) 调用符号表管理器或出错处理器，进行相关处理。词法错误是源程序中常见的错误，如出现非法字符、拼错关键字、多或少字符等。值得注意的是，词法错误往往不是由词法分析器检查出来的，而是由语法分析器发现的。这是因为，源程序中除了非法字符之外的大部分字符或字符串，都可以被词法分析器的某个模式所匹配，从而被识别成一个记号。而这些记号的正确与否，在没有上下文对照的情况下，是很难判断的。例如， 12x 被认为是一个非法的 Pascal 的标识符，但是，由于 12 可以被识别整型数的模式匹配，而 x 可以被识别标识符的模式匹配，因而词法分析器会分别识别出一个整型数和一个标识符，而不是报告一个错误。

根据编译器的总体需求，词法分析器在整个编译器中可以有不同的工作方式。

(1) 词法分析器作为语法分析器的子程序。最常采用，也最容易实现的工作方式是将词法分析器作为语法分析器的子程序，每当语法分析器需要一个记号时，就调用词法分析器，并得到一个识别出的记号。其工作方式如图 2.1 所示。

(2) 词法分析器进行单独的一遍扫描。另一种常用的工作方式，是安排词法分析器进行单独的一遍扫描，它以源程序为输入，输出是以记号流形式表示的源程序。其工作方式如图 2.2 所示。

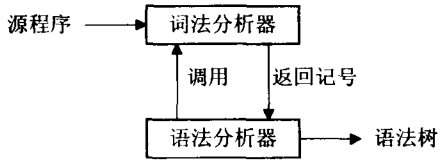


图 2.1 作为子程序的词法分析器

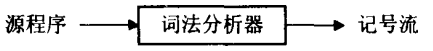


图 2.2 词法分析器进行单独一遍扫描

(3) 与语法分析器并行工作的模式。上述两种词法分析器的工作模式与语法分析器的关系均被认为是串行的。为了提高编译器的效率，可以通过一个队列，使词法分析器和语法分析器以生产／消费的形式并行工作。词法分析器将识别出的记号流输出到队列中，语法分析器从队列中取得记号，只要队列中有识别出的记号，则词法分析器和语法分析器就可以同时工作。其工作方式如图 2.3 所示。

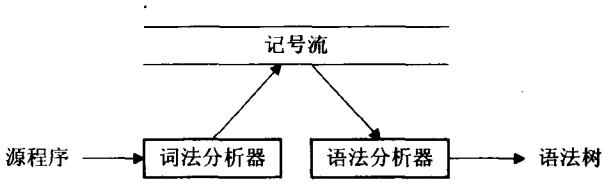


图 2.3 并行工作模式

2.1.4 输入缓冲区

词法分析器是编译器中读入源程序字符序列的唯一阶段，而相当可观的编译时间又消耗在词法分析阶段，所以，加快词法分析是设计编译器时要考虑的重要问题之一。可以通过设立输入缓冲区来加快读入源程序字符序列的速度。

如果使用词法分析器生成器编写词法分析器，则生成器会提供读入和缓冲输入序列的例程；如采用通用程序设计语言编写词法分析器，就需要显式地管理源程序的读取。

输入缓冲区一般被设计为一块与磁盘扇区大小成倍数关系的内存。若一个扇区为 1024 字节，则输入缓冲区可以取 1024、4096 或 8192 字节等。这样可以保证对缓冲区的一次输入所需的 I/O 操作次数尽可能少。

输入缓冲区的安排一般采用单缓冲区或双缓冲区（缓冲区对）的方式。下边所介绍的是单缓冲区方式，它也是词法分析器生成器 FLEX 所采用的方式。

图 2.4 是一个单缓冲区的示意图。有效输入序列从缓冲区的起始位置开始存放，最后添加一个特殊标记(此处用 #表示）：若缓冲区一次装不下整个源程序，它就表示缓冲区的结束，否则它紧跟在文件结束符 'eof)之后，表示整个输入源程序的结束。用两个指针 c_ptr 和 f_ptr