21世纪高等学校计算机规划教材
21st Century University Planned Textbooks of Computer Science

编译原理

Principles of Compiler

康慕宁 林奕 编著

- 注重编译器全系统工作原理的完整介绍
- 全面讲解编译器设计中的关键模型和算法
- 结合实例介绍实际目标代码结构与执行机理

精品系列

21世纪高等学校计算机规划教材

21st Century University Planned Textbooks of Computer Science

编译原理

Principles of Compiler

康慕宁 林奕 编著

精品系列

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

编译原理 / 康慕宁，林奕编著. -- 北京 : 人民邮电出版社，2010.10
21世纪高等学校计算机规划教材
ISBN 978-7-115-22405-7

Ⅰ. ①编… Ⅱ. ①康… ②林… Ⅲ. ①编译程序－程序设计－高等学校－教材 Ⅳ. ①TP314

中国版本图书馆CIP数据核字(2010)第077840号

内 容 提 要

本书系统地介绍了编译程序的基本结构、工作流程、关键算法与思想以及辅助设计工具。主要内容包括程序设计语言基本理论，词法分析、语法分析的主要模型和算法，语义分析和属性文法，语法制导的代码生成技术，运行时存储空间组织与管理以及代码生成和优化等。本书简洁明了地论述了编译器设计中采用的主要技术，并提供了大量例题及其解答。学习本书不仅可以使学生掌握编译思想和技术，而且也为加深对程序设计语言的理解和理解软件底层运行机理奠定了基础。书中每章都有难度适宜的习题，可以使学生更好地掌握所学知识。

本书可作为高等学校计算机及相关专业的教材，也可以作为考研学生的参考书。

21 世纪高等学校计算机规划教材

编 译 原 理

◆ 编　著　康慕宁　林　奕
责任编辑　贾　楠

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
大厂聚鑫印刷有限责任公司印刷

◆ 开本：787×1092　1/16
印张：13.5　　2010 年 10 月第 1 版
字数：348 千字　　2010 年 10 月河北第 1 次印刷

ISBN 978-7-115-22405-7

定价：26.00 元

读者服务热线：(010)67170985　印装质量热线：(010)67129223

反盗版热线：(010)67171154

出版者的话

计算机应用能力已经成为社会各行业最重要的工作要求之一，而计算机教材质量的好坏会直接影响人才素质的培养。目前，计算机教材出版市场百花争艳，品种急剧增多，要从林林总总的教材中挑选一本适合课程设置要求、满足教学实际需要的教材，难度越来越大。

人民邮电出版社作为一家以计算机、通信、电子信息类图书与教材出版为主的科技教育类出版社，在计算机教材领域已经出版了多套计算机系列教材。在各套系列教材中涌现出了一批被广大一线授课教师选用、深受广大师生好评的优秀教材。老师们希望我社能有更多的优秀教材集中地呈现在老师和读者面前，为此我社组织了这套“21世纪高等学校计算机规划教材-精品系列”。

“21世纪高等学校计算机规划教材-精品系列”具有下列特点。

（1）前期调研充分，适合实际教学需要。本套教材主要面向普通本科院校的学生编写，在内容深度、系统结构、案例选择、编写方法等方面进行了深入细致的调研，目的是在教材编写之前充分了解实际教学的需要。

（2）编写目标明确，读者对象针对性强。每一本教材在编写之前都明确了该教材的读者对象和适用范围，即明确面向的读者是计算机专业、非计算机理工类专业还是文科类专业的学生，尽量符合目前普通高等教学计算机课程的教学计划、教学大纲以及发展趋势。

（3）精选作者，保证质量。本套教材的作者，既有来自院校的一线授课老师，也有来自IT企业、科研机构等单位的资深技术人员。通过他们的合作使老师丰富的实际教学经验与技术人员丰富的实践工程经验相融合，为广大师生编写出适合目前教学实际需求、满足学校新时期人才培养模式的高质量教材。

（4）一纲多本，适应面宽。在本套教材中，我们根据目前教学的实际情况，做到“一纲多本”，即根据院校已学课程和后续课程的不同开设情况，为同一科目提供不同类型的教材。

（5）突出能力培养，适应人才市场要求。本套教材贴近市场对于计算机人才的能力要求，注重理论技术与实际应用的结合，注重实际操作和实践动手能力的培养，为学生快速适应企业实际需求做好准备。

（6）配套服务完善，共促提高。对于每一本教材，我们在教材出版的同时，都将提供完备的PPT课件，并根据需要提供书中的源程序代码、习题答案、教学大纲等内容，部分教材还将在作者的配合下，提供疑难解答、教学交流等服务。

在本套教材的策划组织过程中，我们获得了来自清华大学、北京大学、人民大学、浙江大学、吉林大学、武汉大学、哈尔滨工业大学、东南大学、四川大学、上海交通大学、西安交通大学、电子科技大学、西安电子科技大学、北京邮电大学、北京林业大学等院校老师的大力支持和帮助，同时获得了来自信息产业部电信研究院、联想、华为、中兴、同方、爱立信、摩托罗拉等企业和科研单位的领导和技术人员的积极配合。在此，人民邮电出版社向他们表示衷心的感谢。

我们相信，“21世纪高等学校计算机规划教材-精品系列”一定能够为我国高等院校计算机课程教学做出应有的贡献。同时，对于工作欠缺和不妥之处，欢迎老师和读者提出宝贵的意见和建议。

前 言

编译原理是计算机领域中一门较为“古老”和“偏僻”的课程。说它“古老”，是因为其中很多主要技术都源自20世纪80年代以前的老技术；说它“偏僻”，是因为很多人也许永远也不会参与一个编译器的设计项目。虽然这门课程既“古老”又“偏僻”，但编译原理课程并没有成为“古董”。实际上，除了对传统意义上的编译优化技术进行深入研究之外，编译技术可以用在所有使用程序设计语言的地方，如浏览器对HTML、XML的解析，Adobe Acrobat和AutoCAD对格式化文档的解析，对JSP、ASP、ActionScript等脚本语言的解析，分布式计算中DCOM、CORBA等接口定义语言（IDL）的翻译以及各种软件工具的设计。在虚拟机、多核编程、嵌入式程序设计等领域，编译技术也得到了广泛应用。因此，学好编译技术对于学生未来从事计算机软件相关的工作具有重要价值。

编译原理是一门理论和实践性都很强的课程。在我们的教学实践中，教师和学生普遍感觉这门课程难讲、难学。为了更好地帮助学生理解和学习本课程的内容，本书在保证编译理论较为完整的同时，提供了很多例题并给出了详细的解答。同时，为了保持内容的简洁和清晰，本书对所讲述的理论和技术进行了有针对性的选择。例如，在语法分析一章，本书以LR(1)这种在实际中使用的算法为主要介绍内容，而对LR(0)、SLR(1)等算法不作深入分析。在介绍编译器后端技术时，本书力图解决现有编译原理书籍仅局限于理论介绍的不足，结合实际软件系统的编译和操作系统运行环境，介绍了程序设计语言运行与执行的原理和实例。代码优化是现代编译技术研究的重点；限于篇幅，本书仅对优化的基本思想和原理进行介绍，给出了值编号、DAG图等基本优化算法详细论述，并给出了更多关于优化技术的参考文献。

本书共8章，第1章（1.6节）、第2章～第4章由康慕宁编写，第1章（1.1节～1.5节，1.7节）、第5章～第8章由林奕编写。

限于编者水平，书中难免存在不足乃至错误之处，敬请读者批评指正。

编 者

2010年5月

试读结束：需要全本请在线购买： www.ertongbook.com

目　录

第 1 章　绪论……1

1.1　汇编语言和高级程序设计语言……1
1.2　程序设计语言的编译技术……2
1.3　编译技术的基本构造与工作原理……4
1.4　程序设计语言的编译技术……7
1.4.1　词法分析……7
1.4.2　语法分析……8
1.4.3　语义分析……8
1.4.4　中间代码的生成……9
1.4.5　代码优化……10
1.4.6　目标代码生成……10
1.4.7　程序信息管理与错误检查和处理……11
1.5　编译程序的工作过程……11
1.6　文法及其分类……13
1.6.1　文法……13
1.6.2　文法及语言的 Chomskey 分类……15
1.6.3　规范推导……16
1.6.4　文法的二义性……17
1.7　本书内容简介……18
习题……18

第 2 章　扫描器与正规语言……21

2.1　正规表达式……21
2.1.1　正规表达式代数……22
2.1.2　正规表达式的性质……22
2.2　有限状态自动机……24
2.3　非确定的有限状态自动机……26
2.4　将正规文法转换为自动机……27
2.5　NFDA 的确定化及化简……29
2.6　从有限状态自动机转换到正规文法……35
2.7　有限自动机在计算机中的实现……36
2.8　扫描器实现中的特殊问题……37
2.8.1　输入符号表……37
2.8.2　扫描器自动机中的终止状态……37
2.8.3　删除空白符号与注释……38
2.8.4　输出单词……38
2.9　字符串表的实现……40
2.10　保留字……41
2.11　使用扫描器自动生成工具……41
2.12　例题解析……41
习题……43

第 3 章　语法分析与前后文无关文法……45

3.1　下推自动机……45
3.1.1　停机条件的等价性……46
3.1.2　从前后文无关文法 CFG 构造 PDA……47
3.2　LL(*k*)规范文法……48
3.2.1　FIRST 集与 FOLLOW 集……49
3.2.2　选择集合……50
3.3　文法的左递归性……51
3.4　公共左因子……52
3.5　用正规表达式运算符拓广 CFG……52
3.6　递归下降分析程序……53
3.7　作为下推自动机的递归下降分析程序……54
3.8　自底向上的语法分析器的构造……56
3.8.1　自底向上的语法分析……56
3.8.2　LR(*k*)分析法……59
3.9　语法分析器生成工具简介……70
习题……71

第 4 章　语法制导的代码生成……73

4.1　常见的中间语言简介……73
4.1.1　逆波兰表示……73

4.1.2 四元式 …… 74
4.1.3 其他表示法 …… 75
4.2 赋值语句的翻译 …… 75
4.3 布尔表达式的翻译 …… 76
4.4 程序流程控制语句的翻译 …… 81
4.4.1 常见控制结构的翻译 …… 81
4.4.2 语句标号及 GOTO 语句的翻译 …… 84
4.4.3 多分支语句的翻译 …… 87
4.5 含数组元素的算术表达式及赋值语句的翻译 …… 89
4.5.1 下标变量地址的计算 …… 89
4.5.2 含有下标变量的赋值语句的翻译 …… 91
4.6 过程说明和过程调用的翻译 …… 93
4.6.1 过程说明的翻译 …… 94
4.6.2 实参和形参间的信息传递 …… 94
4.6.3 过程语句的翻译 …… 95
4.6.4 关于形实结合的进一步讨论 …… 96
4.7 说明语句的翻译 …… 97
4.7.1 类型说明（变量及数组定义）语句的翻译 …… 97
4.7.2 数据类型定义语句的翻译 …… 99
习题 …… 101

第 5 章 符号表 …… 103

5.1 概述 …… 103
5.2 符号表的内容、用途与创建过程 …… 104
5.2.1 符号及其所表示的信息 …… 104
5.2.2 符号表 …… 105
5.2.3 符号表的用途 …… 105
5.3 设计符号表时需要考虑的几个问题 …… 107
5.4 符号表的创建和使用 …… 108
5.4.1 符号表的创建及其在语义分析中的使用 …… 108
5.4.2 符号表在内存分配和代码生成阶段的使用 …… 109
5.5 符号表的数据结构与算法 …… 110
5.5.1 符号表数据结构和算法的选择 …… 110
5.5.2 哈希表设计 …… 112
5.5.3 支持作用域的符号表 …… 115
习题 …… 122

第 6 章 静态与运行时的存储管理 …… 124

6.1 可执行程序的产生和执行过程 …… 124
6.2 可执行程序的存储布局与操作 …… 126
6.3 内存管理技术：静态内存管理 …… 128
6.4 动态内存管理 …… 132
6.5 栈式内存管理 …… 133
6.5.1 递归调用引起的问题 …… 133
6.5.2 栈 …… 134
6.5.3 活动树 …… 136
6.5.4 栈模型和活动树模型的比较 …… 140
6.5.5 栈式内存管理的基本思想和必须解决的问题 …… 140
6.5.6 活动记录和调用序列 …… 142
6.6 堆式管理和垃圾回收技术 …… 152
习题 …… 155

第 7 章 代码优化 …… 157

7.1 概述 …… 157
7.2 中间代码生成阶段的代码优化 …… 157
7.3 代码优化的基本原则、思路和范围 …… 158
7.4 基本块及其优化方法 …… 159
7.4.1 基本块、基本块划分算法和控制流图 …… 159
7.4.2 基于 DAG 模型的基本块优化技术 …… 162
7.4.3 基于值编号技术的基本块优化算法 …… 166
7.4.4 基本块中的其他优化 …… 170
7.4.5 更大范围的优化 …… 173
7.5 数据流分析 …… 173
7.5.1 基于数据流分析的冗余表达式删除 …… 174
7.5.2 活性分析 …… 178
7.5.3 过程间数据流分析 …… 182
7.6 循环优化 …… 182
7.6.1 循环的识别 …… 182

7.6.2　循环优化 ……185
习题 ……187

第 8 章　代码生成 ……189

8.1　代码生成的基本功能 ……189
8.2　代码生成的不同方式 ……189
8.3　代码生成的关键技术简介 ……190
8.3.1　代码生成技术需要考虑的内容 ……190
8.3.2　代码生成的关键技术 ……190
8.3.3　指令筛选技术简介 ……191
8.3.4　指令调度技术简介 ……198
8.3.5　寄存器分配技术简介 ……200
8.4　代码生成、软件调试和其他技术 ……202
习题 ……205

参考文献 ……206

第1章 绪论

程序设计语言为软件开发人员提供了有力的武器，使他们能够充分发挥自己的想象力，创造出成千上万种软件产品。为了更快、更好地设计出如此众多的庞大软件系统，程序设计师们需要使用比汇编语言等底层程序设计语言更为有效的高级程序设计语言。然而，在我们熟悉的高级程序设计语言和计算机底层硬件之间存在一条巨大鸿沟，编译技术正是使我们得以跨越鸿沟的技术桥梁。

1.1 汇编语言和高级程序设计语言

为早期计算机系统开发软件是一项非常复杂和繁琐的工作。那个时期的计算机系统远不像今天的个人计算机那样距离我们如此之近，为其开发软件则也必然只是少数顶级专家们的专利。最初的计算机程序只是由专家们编写出的一系列二进制数字，然后人们将其刻到纸带上，通过专门的纸带阅读设备将其输入计算机中。然而，随着需要编写的程序在数量和复杂性上的增长，人们发现直接面对二进制数据进行程序设计过于繁琐。为了解决这一问题，计算机软件设计者们设计了汇编语言和能够将汇编程序翻译为计算机可执行的二进制指令编码的汇编器。早期的汇编语言和 CPU 的硬件指令非常类似，因此实现这样一个简单汇编器的方式也通常比较直接。例如，可以将某条汇编语句直接映射到其对应的指令编码[1]。和直接理解二进制代码相比，汇编语言的出现，显然给那个时代的软件设计者们带来了巨大的方便和更高的生产力。

然而对于今天的大多数程序设计人员来说，汇编语言已经成为了一个既熟悉又陌生的概念。导致汇编语言使用逐渐减少的一个根本原因，就是出现了具有更为丰富表达能力的各种高级程序设计语言（High Level Programming Language，HLPL）。实际上，稍加比较我们就不难发现，用高级语言编写程序要比用汇编语言编写程序具有更多的优点。首先看一个例子。

例 1.1 对于 C 语言程序员来说，编写表达式 x=a+b+c+d 非常简单。但如果用汇编语言来写，则不得不写出多条语句（采用 Intel CPU 的汇编）：

```
mov  eax,dword ptr [ebp-8]
add  eax,dword ptr [ebp-0Ch]
add  eax,dword ptr [ebp-10h]
add  eax,dword ptr [ebp-14h]
```

[1] 现代汇编语言为了支持模块的动、静态链接和地址重定位等技术，通常都更为复杂。为了给汇编程序设计者提供更多方便，出现了各种“宏汇编”，以支持对复杂表达式等的直接表示。

mov   dword ptr [ebp-4],eax

汇编程序的第 1 条语句将变量 a 对应的值放入 eax 寄存器中。第 2 条语句将变量 b 的值和 eax 寄存器中的值相加，结果放入 eax 中（即执行 a+b）。第 3 条、第 4 条语句将 a+b 的结果与 c 相加，将 a+b+c 的结果与 d 相加，结果放在 eax 寄存器中。最后一条语句将最终结果（即 a+b+c+d）放入变量 x 的内存地址中。

对例 1.1 进行分析，不难得出如下结论。

（1）高级语言为程序员提供了更为方便、快捷的编程工具。毫无疑问，上面的 C 语言程序更为简洁紧凑，而且也不需要理解 Intel CPU 底层所使用的寄存器与内存访问指令。

（2）高级语言与硬件无关。汇编语言的定义和 CPU 的指令系统直接相关，因此不同 CPU 甚至同一系列不同型号 CPU 的指令系统都会不同。这就导致在 Intel 处理器下编写的汇编语言程序无法在非 Intel 处理器（如 IBM 的 Sparc 处理器或 ARM 处理器和 DSP 处理器）上执行。

与汇编语言相反，C 语言程序在大多数情况下对于不同的硬件环境来说具有相同的含义。这主要是由于该程序中只定义了 a、b、c、d 和 x 这些抽象的数学变量及其运算规则，而对于这些数学概念如何在 CPU 中实现则不做具体规定（例如，Intel CPU 不允许对两个内存变量进行直接计算，而必须至少使用一个寄存器）。

虽然不同的计算机硬件系统在实现这些抽象概念时采用的方式不同，但却能得到相同或相似的执行结果[2]。因此，只要能够将高级语言编写的程序等价地转换为特定硬件平台所支持的方式来实现（即汇编程序或机器指令序列），那么软件设计师就不必为每种硬件平台重新编写具有相同功能的软件（见图 1.1）。

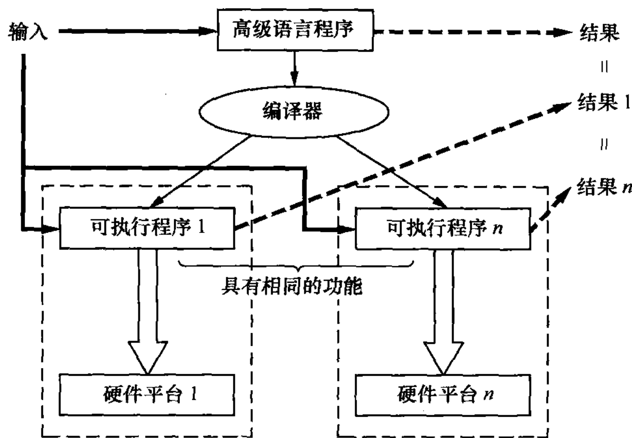

图 1.1　高级语言的可移植性

1.2　程序设计语言的编译技术

本节首先从编译器使用方式的角度介绍编译器的基本概念，然后对编译器、解释器和虚拟机

[2] 实际上，由于硬件不同会导致在计算时间、计算精度等方面的差别，因此 C、C++等语言编写的程序在移植到不同硬件平台时需要仔细考虑移植后程序的正确性。这种细微的差别可能使软件移植困难重重。Java 由于采用了虚拟机的概念，可以提供一个一致的虚拟计算平台，因此其可移植性明显好于 C、C++。

这 3 个重要概念进行介绍和分析。

1. 狭义的编译器概念

高级语言为程序员提供了比汇编语言更为方便的程序设计工具，使程序员的工作效率得到了极大的提高。然而，对于底层计算机硬件系统来说，CPU 仍然只能执行由二进制指令序列表示的程序代码。这样一来，高级语言编写的程序只有被翻译为具有相同功能的汇编语言程序或被直接翻译为可执行指令序列后，才能被 CPU 真正地执行。由于高级语言程序和 CPU 指令系统在形式和内容上的差别远比汇编语言和二进制指令的差别大得多，因此需要更为复杂的技术将其翻译为等价的可执行指令序列（或先翻译为汇编程序，再由汇编器将其转换为可执行指令序列）。编译器就是实现这种把高级语言编写的程序翻译为具有相同功能的底层指令序列或汇编程序的软件系统。编译器设计时采用的各种技术统称为编译技术。

一般来说，编译器需要具有两个基本的功能。

- 判断程序的合法性。识别输入的源程序是否符合语言定义的要求。例如，对于 C 语言程序来说，变量是否以字母开头，if 语句是否包含了合法的表达式等。
- 程序的等价翻译和错误提示。如果源程序合法，则可将其转换为另一种语言表示的程序（称为目标程序）。否则，将给出程序的出错提示，以方便软件开发人员查找和修改程序中的错误。

2. 广义的编译器概念

设计编译器的最初目的，是将高级语言程序等价地转换为具有相同功能的可执行程序或汇编程序。然而，随着编译技术的发展，编译程序的用途更为广泛。例如，在 C 语言出现以后，有一些编译器（如 GCC 系列的 Fortran 77 编译器 g77）就不再直接将 Fortran 语言翻译为可执行程序或汇编程序，而是首先翻译成 C 语言程序，然后再由 C 语言编译器完成最后的编译工作。又如，在分布式计算技术中，CORBA 和 DCOM 平台均定义了 IDL（Interface Definition Language）来描述程序的接口规范，并利用 IDL 编译器将模块的 IDL 描述翻译为特定语言的接口定义（如 C、C++ 的头文件，Java 的类定义等）。

上面这些编译器应用显然不再局限于产生可执行程序的范围。因此，不妨定义一个更为一般的广义的编译器概念：编译器就是将一种语言编写的程序翻译成由另一种语言编写的功能相同的程序的软件。

特别指出一点，广义的编译器定义并不要求所得到的程序一定是可执行的，只要该程序在功能和含义上与原来的程序一致就可以了。从这个角度看，编译器实际上起到的是一种程序转换器或程序生成器的作用。

3. 编译器和解释器

C 语言编译器是程序员最熟悉的编译器，负责将 C 语言程序翻译为可执行代码，符合以上对编译器的基本定义。然而，还有另外一种使用编译技术的方式。

我们熟悉的 Internet Explorer 可以对 HTML 网页文件进行解析，然后根据 HTML 文件的定义将文字和图片按指定格式显示在屏幕上。为了实现这一功能，Internet Explorer 必须首先“理解”HTML 网页文件的内容，然后按其规定调整与安排文字和图片的显示方式。在这一过程中，Internet Explorer 既没有将 HTML 文件变成 CPU 可以直接执行的二进制代码，也没有将其翻译为某种 CPU 的汇编语言程序，而是同时完成了对语言的理解和执行两件事情。首先，浏览器的 HTML 分析模块将 HTML 文件的内容进行分析和理解，得到一个内部的数据结构（记录了网页信息及其显示方式）。然后，浏览器的显示模块扫描该数据结构，根据格式要求将信息逐条显示在恰当的屏幕位置上。实际的 Internet Explorer 在完成上述工作时为了提高工作效率，可能不会按这么简单的方式工

试读结束：需要全本请在线购买： www.ertongbook.com

作，但基本原理与此相同。

以上这种对语言的处理方式称为语言的解释执行。实现这种语言解释执行功能的软件称为语言的解释器。

除了浏览器中的 HTML 解释器之外，BASIC 语言、TCL/Tk 语言、Lisp 语言等通常也采用解释器的方式来执行。和先编译后执行的编译模式相比，解释器的最大好处就在于程序编写灵活和使用方便。然而，高级语言和底层指令系统的差别较大，因此要实现对程序的解释执行，必须在程序解释执行的过程中进行程序的翻译。由于不是被 CPU 直接执行，而且还要进行很多额外的工作，因此解释执行的程序在性能上比编译执行的程序要慢很多。

4. 编译器和虚拟机

为了避免单纯解释执行在效率方面的不足，Java 语言引入了虚拟机的概念。Java 虚拟机是 Java 语言规范中的一部分，定义了一套标准的指令集合（称为字节码）。Java 编译器将 Java 程序翻译为具有相同功能的字节码程序，然后由 Java 虚拟机对字节码程序进行解释执行。由于字节码和 CPU 的二进制指令更为接近（虽然不同 CPU 的指令集合会有所不同，字节码和指令的差距远比 Java 程序和 CPU 指令的差距要小），因此对字节码的解释执行速度远比对高级语言程序直接进行解释要快。为了进一步提高执行效率，Java 编译器还采用了即时编译（Just-In-Time，JIT）等技术。

由于本书篇幅有限，因此对解释器、虚拟机和汇编器将不再进行更为详细的介绍。有兴趣的读者可以参考相关文献了解更多信息。

1.3 编译技术的基本构造与工作原理

理解编译器的工作原理，对于深入理解编译器设计技术中各种技术的内涵很有好处。本节将通过一个简单的例子，说明编译器的主要工作原理。

1. 编译器和语言的规格说明

在我们学习 C 语言的时候，手边通常都会放一本 C 语言的参考手册。查阅手册中的 C 语言文法规定，可以帮助我们判断自己的 C 语言程序是否存在格式错误（如是否少写了一个括号）和某些内容方面的错误（如变量是否重复定义）。同理，既然编译器需要能够判断程序的合法性，那么实际上在编译器内部也应当有一个关于被处理程序所用语言的“参考手册”，只不过这个“参考手册”是用程序来记录和查阅的罢了。那么，对于编译器来说，需要什么样的“语言参考手册”呢？

我们熟悉的 C 语言参考手册通常是一本厚厚的书。然而，编译器设计者们不可能让编译器去“读书”。因此，只有两种方式可以让编译器记住和理解 C 语言的语法规则。

第一种方式是由编译器设计者首先阅读厚厚的 C 语言手册（自然语言），然后根据手册中的规定编写编译器的代码。这些编译器代码负责执行对诸如语句结构是否完整、变量定义是否唯一等规则的判断。由于自然语言的描述通常不够精确，通常很难保障根据同一本 C 语言手册写出来的两个 C 语言编译器真的具有相同的功能。

第二种方式是用某种特定的数学模型来描述语言的语法规则。然后，程序员既可以根据精确的数学模型设计出功能统一的编译器，也可以考虑利用自动代码生成的方法直接将描述语言定义规则的数学模型转换成编译器的实现代码。从编译器设计技术的现状看，还不能把语言定义的所有内容都精确地用数学模型表示出来并方便地转换为代码实现。

2. 编译器的内部工作原理

有了语言的参考手册，下面让我们来研究一下根据手册如何理解程序的过程。首先让我们看一看程序员在阅读和理解程序时都完成了哪些工作。下面仍以一个 C 语言的程序为例进行说明。

例 1.2 C 语言程序的理解与翻译过程。

```
int a;
int add(int d1, int d2)
{
     return d1+d2;
}
int main(void)
{
     a=5;
     a=add(a, 6);
     return 0;
}
```

（1）人工翻译的过程。当程序员在阅读这段代码时，他们能够得到什么信息呢？又是怎样得到这些信息的呢？熟练的程序员通常会首先浏览整个程序的大致结构，发现整个程序包含 3 个相对独立的定义——即包含了一个全局变量定义和两个函数定义。然后，他会查看变量定义和函数定义的具体内容，获得以下的信息。①该程序包含一个全局变量定义 a，该变量是整型的。②包含两个函数定义。函数 add 的返回值是整型，且接收两个整型参数，并将两个输入参数的和作为返回值传递给调用方。③main 是主函数，没有参数且返回值是整型。main 函数中将变量 a 赋值为 5，并以 a 和整型常量 6 为参数调用 add 函数，结果赋值给变量 a。main 函数最后返回 0。

上面的这些信息已经足够我们理解该程序的含义。熟悉 C 语言和汇编语言的读者也许已经可以很容易地将该 C 语言程序翻译为具有相同功能的汇编程序。换言之，在程序员阅读和翻译上面的程序时，采用的是如图 1.2 所示的方式。

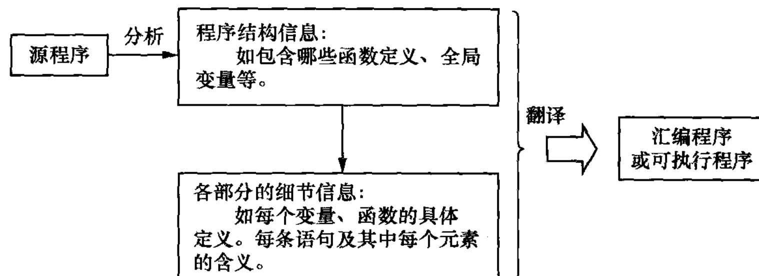

图 1.2 程序的人工翻译过程

图中所示的模型表明，程序员在进行翻译时完成了两个任务，即程序的分析（从源程序中提取必要的信息）和程序的转换（根据搜集的程序信息和对汇编语言或硬件指令集的理解，编写等价的结果程序）。这两个阶段通常又称为**程序的分析和程序的综合**。而从源程序中提取的相关信息是使这两个任务有机结合为一个整体的基础。程序员会采用多种灵活的方式来记录源程序中的信息，如可以列出关键函数表、全局变量表等。此外，一些细节信息则可以通过对源程序的搜索来回顾。例如，看见变量 a 之后如果想不起来其具体的定义形式，则可以回头去看 a 被定义的地方。

（2）程序的自动翻译过程。编译器在执行编译任务时，要完成的工作和人工翻译时非常类似。因此，编译器也需要通过对源程序的分析来搜集程序的结构与细节信息，然后根据这些信息和对

目标语言的理解产生编译结果。然而，要让编译器从程序源代码中自动提取出这些程序含义的信息却并不是一件容易的事情。实际上，如果将程序换一个写法，即使程序员来阅读这段程序也会感到非常困难。

```
int a;int add(int d1,int d2){return d1+d2;}int main(void){a=5;a=add(a, 6);return 0;}
```

这段程序和上面的程序在内容上完全一样，只是缩进、分行排版格式等被完全删除了。而这正是编译器所看到的程序的样子——纯粹的符号序列，没有可以提供直观视觉帮助的排版格式（前面的程序里可以一眼看出 int main(void)是函数的定义，但这里则不得不一个字一个字地去分析才能知道）。另一方面，一些对于程序员来说非常自然的任务如果让编译器来做，也会更为复杂。像记录函数定义这样的事情，程序员只需拿起纸和笔就可以了，而编译器必须为记录这些信息提供相应的数据结构。由于这些原因，编译器在分析和翻译时，通常是按与下面过程类似的方式工作的。

① 源程序的分析过程：首先，编译器读进一个单词 int[3]。根据对 C 语言的理解，该单词或者是一个全局变量定义的开头，或者是某个函数定义的开头。然后，编译器读进名字 a 和分号，并由此判断该语句是一个变量定义语句，定义了一个整型变量 a。为了保障全局变量 a 没有重复定义，编译器应记录所有变量定义信息以供检查。

编译器读进下一个 int、add，并遇到左括号“(”。此时，C 语言编译器可以确定遇到的是一个函数定义，函数名为 add，返回值是整型。而后，编译器识别出 add 的参数个数、每个参数的类型及名称等信息。当遇到“{”时，继续识别 add 函数内的各条语句。

识别完 add 函数后，即可继续识别后面的变量定义和函数定义等代码。

② 逐词分析和完整分析：这里需要特别指出一点，无论是程序员还是编译器，当看到“int add(”时由于还没有看到后面的内容，因此无法确定一定遇到的是完整、正确的函数定义（如程序员不小心写成“int add(int d1; int d2)”）。这样，只有当看到了完整的“int add(int d1, int d2)”时才能得出准确的结论。由此可见，编译器在对程序进行分析时不能看了后面忘记前面，即编译器不仅要能够知道正在看到的单词的含义，更要能够将后面遇到的内容和之前已经识别的内容合起来作为一个整体检查，才可以得到正确的结论。

在这种不断检查和匹配的过程中，编译器就可以知道程序、定义、语句等的结构是否完整，从而判断程序在格式上的合法性。

③ 信息搜集过程：另一方面，为了判断像变量重复定义、变量赋值类型不匹配等不属于格式错误的问题，编译器还需要在分析的过程中搜集有关变量、函数定义形式、函数参数个数及其类型等方面的信息，并以某种形式将这些信息保存下来。和上面的情况类似，由于编译器在识别程序时是一个符号一个符号进行分析的，因此要确定函数 add 定义的所有信息，就必须将“int add(int d1, int d2)”全部识别完后才能进行记录。而关于函数名“add”、函数返回值“int”、函数的两个参数“int d1”、“int d2”的识别都是分别完成的。因此，编译器设计者需要提供一种机制，将这些在不同步骤中识别出来的零散信息整合到一起，供后续识别和翻译使用。

④ 翻译过程：由于高级语言程序中的很多语言结构不能直接映射到 CPU 指令集合中，因此编译器必须知道每种语言成分应当如何用所需 CPU 的指令来实现。例如，C 语言的 if-then-else 语句可以用一系列的简单运算指令和几个条件转移指令共同实现（参见本书关于语义翻译部分的论述）。由此可见，翻译过程通常不是直接进行的。编译器通常不能仅仅依靠当前所看到的程序片段产生所需的翻译结果，而是要等到所需信息都知道后才能进行翻译（注意，编译器扫描程序是

[3] 单词的识别也不是简单地看看哪里有空格就可以的。对单词的正确性及其类别的识别同样需要一定的技术来提供支持。具体内容参见本书关于词法分析部分的介绍。

按顺序进行的，因此必须将前面扫描中识别的信息记到内部数据结构中，才能在后面的分析和综合过程中获得足够的信息）。这与信息搜集时的情况非常类似。

下一节中，将给出编译程序的基本结构，并结合编译器的工作原理对其进行说明。

1.4 程序设计语言的编译技术

编译程序的设计应当包括读取输入的源程序、分析源程序并提取/记录信息和生成目标程序几个部分。早期编译程序在设计时缺乏理论指导，通常将这几部分的工作糅合在一起。然而，对于编译器这种复杂程度很高的软件系统来说，采取模块化设计的思想更有利于提高其软件设计的质量。那么，编译程序应当被分解为哪些模块才比较恰当呢？在经过了对编译器设计的长期实践后，尤其是编译器设计理论不断完善后，对编译器的结构提出了业界较为公认的有效模块/功能划分方法。图 1.3 所示为编译程序的常见功能划分。

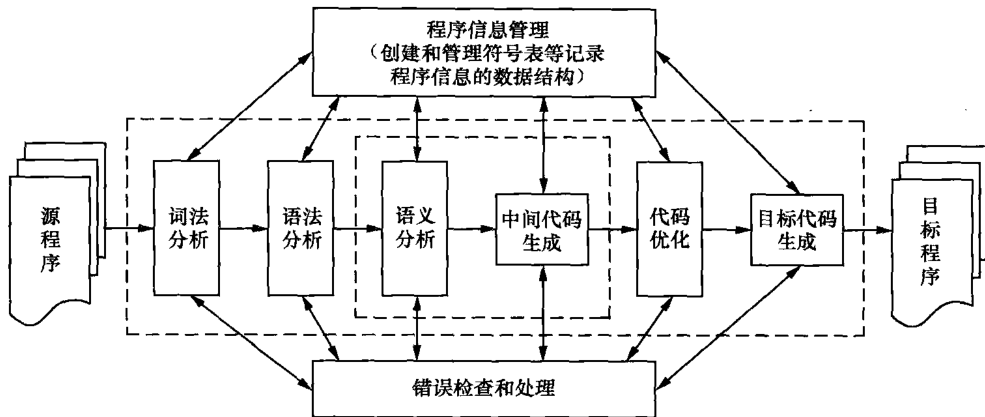

图 1.3 编译程序的逻辑结构和功能划分

图中的编译程序被划分为 6 个主要功能和两个辅助功能。以下各小节将对此分别进行介绍。

1.4.1 词法分析

编译器需要读取用户输入的源程序，因此必须能够通过文件读写或用户输入等操作获取源程序。如果让编译器直接读取源程序，则编译器将从输入中一个字符一个字符地读取信息，并将这些字符组织成具有独立边界的单词或符号。例如，例 1.2 中的“int”、“main”、“(”、“void”、“)”、“{”、“a”、“=”、“5”等将被词法分析程序逐一识别出来，然后分别给出每个符号所属的类别：“main”、“a”属于标识符，“int”、“void”是两个不同的关键字，“(”、“)”、“{”、“=”是运算符号，而“5”是整型常量。

通过词法分析程序对源程序的扫描（词法分析程序又称为扫描程序，这是因为其主要功能就是对源程序中的每个字符都进行线性读取），可以使编译程序的其他部分不必再直接访问源程序，而是根据词法分析器所识别出来的符号种类及其内容，对程序的格式和含义进行进一步的分析。

最后需要指出一点，在目前我们常见的高级程序设计语言中，通常都会将空格、制表符（Tab 键）等特殊符号作为程序中各种符号单元的分界符。例如，“abc xyz”之间虽然没有逗号、分号等标点符号，但仍会被编译器看作两个不同的符号串。但是，在 Fortran 这种出现较早的高级语言

中，空格并不作为单词的分界符。这导致 Fortran 词法、语法分析功能很难划分开来的问题，给编译器的实现带来了不少麻烦。

1.4.2 语法分析

编译器的一个重要工作，就是分析输入程序是否符合所使用语言的语法规范。但是，对于编译器设计者来说，语法规范这一说法的含义比我们通常理解的要更狭窄一些。举例来说，对于下面的 C 语言程序：

```
int func(void)
{
    int x, y;
    float x, z; //x 重复定义，不符合 C 程序的定义
}
```

该程序中，重复定义了变量 x，因此是不符合 C 语言规定的错误的程序。然而，编译器设计中所说的语法仅用于表明程序在结构上不符合语言的设计要求，至于每个符号的含义则属于语义的范畴。因此，对编译器设计者来说，上面这段程序在语法上是正确的，但在语义上却是错误的。

由上面的例子不难看出，所谓编译器的语法分析功能，其实就是判断一个程序是否在格式上符合其所用语言的要求，如 if 语句是否包含完整的条件表达式和真/假分支等。

语法分析通常不能采用词法分析中采用的线性扫描方式来设计。以 C 语言的 if-then-else 语句为例，要判断一个 if 语句是否完整，语法分析必须从该语句的第一个关键词 if 开始。然后，分析器逐个识别 if 语句的各个部分，如 if 的表达式、if 的 then 分支中的语句序列、else 分支的语句序列。只有在识别完一条 if-then-else 语句 else 部分的最后一个符号后，才能真正判断出该 if 语句是否完整。如果在这条 if 语句中嵌套了复杂的语句序列，则对该 if 语句的识别将不得不在所有这些语句都处理完成之后才能结束（如下面的程序）。

```
if (a>b) //语法分析器从识别出关键词 if 开始识别本语句
    x=5;
else {
        if(u= =v) { v=9;}
        else { ...
        }
        ...
    }//语法分析器在识别到这个“}”后才能完成
    //对本 if 语句的识别。此前需要先识别诸多其他内容。
```

根据我们前面对语言分析的介绍不难想见，语法分析的工作从某种角度看就是让编译程序自己完成“查阅语言参考手册”的工作。然而，如何让编译程序能够记住并查阅“语言手册”并不是一项简单的工作。

语法分析器的主要工作，一般是尝试为被分析的程序构造一棵语法树。如果成功，则表明被识别的程序符合语法，否则必然存在某种语法错误。语法树的构造需要根据对被分析语言的语法规则来建立。前后文无关文法（Context Free Grammar）或与之等价的 Backus-Naur 范式（BNF）是描述语法规则的有效模型，具有严格、精确的数学含义。对于语法规则的表示和如何根据语法规则构造相应的语法分析算法，是本书的一个核心内容。这也就是前面提到的让编译器记住并查阅“语言参考手册”的一项具体方法。

1.4.3 语义分析

程序的语法只给出了关于程序格式正确性的要求，而程序的语义则给出了程序含义正确性的

规定。换言之，语法规则规定的是句子级别的语言规则，而语义规则定义了比语法规则范围更广的约束条件。例如，C 语言中的标识符作用域规则给出了关于变量、函数名的有效作用范围的规则。由于变量、函数的定义和使用通常不是在一个程序语句中完成的，而是跨越多个程序语句（甚至位于程序的不同部分），因此对语义规则的识别通常需要在更大范围内进行处理。

语义分析有两项主要任务：①搜集和加工由词法和语法分析提供的基本信息，通过整合和计算后得到程序的语义信息；②将该程序的语义信息与语言的语义规则相匹配，从而对程序的语义正确性进行分析和判断。

一般来说，搜集和计算程序语义信息的工作需要对整个程序结构进行分析。因此，该项语义计算任务通常和语法分析结合在一起，**一边进行语法分析，一边进行语义计算**。这就是所谓的**语法制导的语义分析**。至于语义正确性的判断，则既可以在进行语义计算的同时进行，也可在得到了整个程序或部分程序的语义信息后再进行判断。前者的好处是能够根据分析的进度，及时给出程序中的错误位置；后者则更有利于对已经搜集到的信息进行全面分析。具体采用哪种方式处理，需要根据被编译语言的特点和需要进行哪些语义分析操作来进行选择。

下面是一些常见 C 语言编译器需要进行的语义处理。

- 识别变量是数组、指针、简单变量、结构体（struct）或共用体（union），并识别其类型和作用域。为每个变量建立“档案”以供后面的处理来检索和查阅。
- 识别函数定义并记录函数的信息，包括函数的返回值类型、函数名以及每个参数的信息和作用域等。
- 对变量、函数的每次访问是否合法进行检查，如变量类型是否匹配，函数参数个数是否正确，被访问的变量和函数是否在有效作用域内，被访问的结构体变量的分量是否在结构体中已经定义等。

1.4.4 中间代码的生成

从理论上讲，只要获得了足够充分的程序语义信息，就可以将其翻译为等价的汇编代码或二进制机器代码。考虑到实现语言和底层硬件平台的无关性是高级语言的一个重要目标，现代编译器通常不会直接将高级语言映射到低级语言或指令，而是首先将程序转换为某种形式的中间表示或中间代码；然后根据中间表示或中间代码进行性能、存储优化处理后，再将其翻译为最终的可执行代码。下面，对中间表示、中间代码的含义及其一般形式进行简要介绍。

1. 中间代码的基本概念

中间代码是一种和汇编或机器指令非常类似的“虚拟低级语言”，但独立于具体的硬件平台。因此，在中间代码级别进行优化，既可以避免直接生成低级语言目标代码时带来的平台依赖问题，也使优化算法能够更加直接地对存储器分配（如分配临时变量）、指令合并等问题进行检查和处理（这些问题在高级语言程序中被屏蔽了，但在汇编和指令的层面上却必须考虑）。不仅如此，中间代码也可方便地以临时文件的方式存储下来，从而使编译器可以不必一次将程序的所有信息都放在内存中。这对于提高编译器本身的性能很有益处。

常见中间代码包括逆波兰表示、三元式、四元式、P 代码等。

2. 中间表示

中间代码以低级语言的形式记录程序的信息。然而，在将程序转换为中间代码之前，编译器在进行词法、语法和语义分析的过程中需要提取和搜集关于程序结构、含义的信息，并将其记录到恰当的数据结构中。这样，编译器就可以在后续的语义分析中根据这些已经识别出来的信息进

试读结束：需要全本请在线购买： www.ertongbook.com