普通高等学校计算机专业特色教材

编译原理与技术

Compiler Principles and Techniques

张 昱 陈意云

高等教育出版社
HIGHER EDUCATION PRESS

普通高等学校计算机专业特色教材

编译原理与技术

Bianyi Yuanli yu Jishu

张　昱　陈意云

高等教育出版社·北京
HIGHER EDUCATION PRESS　BEIJING

内容提要

本书介绍基本的编译原理与编译技术，是2009年度普通高等教育精品教材《编译原理》（第2版）的精简版。本书主要内容包括词法分析、语法分析、类型检查、运行时存储空间的组织和管理、中间代码生成、代码生成和代码优化、编译系统和运行系统、面向对象语言的编译等。本书取材广泛、新颖，图文并茂；强调对编译技术的理解，淡化对相关理论的学习；强调对各种方法的把握，淡化对各个算法的掌握。

本书可作为高等学校计算机科学及相关专业的少学时编译原理课程教材，也可供计算机软件工程技术人员参考使用。

图书在版编目（CIP）数据

编译原理与技术/张昱，陈意云编著. —北京：高等教育出版社，2010.8

ISBN 978-7-04-029839-0

Ⅰ.①编… Ⅱ.①张…②陈… 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125466号

策划编辑	刘 艳	**责任编辑**	萧 潇	**封面设计**	于 涛	**责任绘图**	尹 莉
版式设计	范晓红	**责任校对**	刘 莉	**责任印制**	陈伟光		

出版发行	高等教育出版社	**购书热线**	010-58581118
社 址	北京市西城区德外大街4号	**咨询电话**	400-810-0598
邮政编码	100120	**网 址**	http://www.hep.edu.cn http://www.hep.com.cn
经 销	蓝色畅想图书发行有限公司	**网上订购**	http://www.landraco.com http://www.landraco.com.cn
印 刷	涿州市京南印刷厂	**畅想教育**	http://www.widedu.com
开 本	787×1092 1/16	**版 次**	2010年8月第1版
印 张	18.5	**印 次**	2010年8月第1次印刷
字 数	390 000	**定 价**	27.20元

物料号 29839-00

前　言

编译原理和技术对高等学校计算机专业的学生和软件工程技术人员来说是重要的基础知识之一。本书是作者所编写的2009年度普通高等教育精品教材《编译原理》(第2版)(高等教育出版社2008年出版)的一个精简版本,面向少学时编译原理课程的教学需求。作者认为,对于普通高等学校的学生来说,学习编译技术的主要目的是掌握基本的编译技术,用以提高学习编程语言及在软件开发中应用编程语言的能力,包括:

(1) 提高学习、理解和使用编程语言的能力。

(2) 提高程序排错的能力,即快速理解、定位和解决在程序开发与程序运行中遇到的问题的能力。

(3) 提高编写高质量代码的能力。

基于该观点,对原书的精简主要体现在下面几个方面:

(1) 忽略对各种LR分析表构造算法的把握,强调对各种语法分析方法的理解。

(2) 删除类型系统形式描述方法的介绍,保留体现类型系统在编程语言中重要地位的章节。

(3) 压缩语法制导翻译的理论介绍,加强积累翻译方案设计技巧的例题和习题。

(4) 淡化代码优化的数据流分析和控制流分析技术,用实例体现各种优化能达到的效果。

(5) 取消函数式语言编译和无用单元收集算法等内容,增加理论联系实际的习题。

(6) 附录中补充了用C语言写的PL/0语言编译器(编译成中间代码)和中间语言解释器,并给出基于PL/0语言的课程实践选题。

此外,在学时数有限时,可以略去带星号的章节内容。

本书具有如下特点:

(1) 强调对编译原理和技术的宏观理解和全局把握,而不把读者的注意力引导到理论和算法上。

(2) 鼓励读者用所学的知识去分析和解决实际问题。

与本书配套的习题解答是由作者编写、高等教育出版社2005年出版的《编译原理习题精选与解析》。课程实践可以采用本书附录给出的基于PL/0语言的课程实践选题,也可以采用由作者编写、高等教育出版社2009年出版的课程实践教材《编译原理实验教程》。作者讲授编译原理课程的教学课件和历年试题在作者教学网页(http://staff.ustc.edu.cn/~

yuzhang,http://staff.ustc.edu.cn/~yiyun)上。

限于时间和学识水平,书中难免存在不妥和错误之处。如果发现本书有任何错误或有任何建议,欢迎给作者发送电子邮件(yuzhang@ustc.edu.cn,yiyun@ustc.edu.cn)批评指正,作者将及时改正错误。

中国科学技术大学为本书的编写提供了足够的资金支持。谨向所有为本书提供支持与帮助的人致以最诚挚的谢意。

作者

于中国科学技术大学

2010年5月

目　录

第1章　引论 …… 1
1.1　编译器概述 …… 1
1.1.1　词法分析 …… 1
1.1.2　语法分析 …… 3
1.1.3　语义分析 …… 3
1.1.4　中间代码生成 …… 4
1.1.5　代码优化 …… 4
1.1.6　代码生成 …… 5
1.1.7　符号表管理 …… 5
1.1.8　阶段的分组 …… 5
1.1.9　解释器 …… 7
1.2　编译器技术的应用 …… 7
1.2.1　高级语言的实现 …… 7
1.2.2　针对计算机体系结构的优化 …… 8
1.2.3　新计算机体系结构的设计 …… 9
1.2.4　程序翻译 …… 10
1.2.5　提高软件开发效率的工具 …… 10
习题 …… 12
第2章　词法分析 …… 13
2.1　词法记号及属性 …… 13
2.1.1　词法记号、模式、词法单元 …… 14
2.1.2　词法记号的属性 …… 15
2.1.3　词法错误 …… 16
2.2　词法记号的描述与识别 …… 16
2.2.1　串和语言 …… 16
2.2.2　正规式 …… 17
2.2.3　正规定义 …… 19
2.2.4　状态转换图 …… 20
2.3　有限自动机 …… 23
2.3.1　不确定的有限自动机 …… 23
2.3.2　确定的有限自动机 …… 25
2.3.3　NFA 到 DFA 的变换 …… 26
2.3.4　DFA 的化简 …… 29
2.4　从正规式到有限自动机 …… 31
2.5　词法分析器的生成器 …… 35
习题 …… 38
第3章　语法分析 …… 41
3.1　上下文无关文法 …… 41
3.1.1　上下文无关文法的定义 …… 42
3.1.2　推导 …… 43
3.1.3　分析树 …… 44
3.1.4　二义性 …… 45
3.2　语言和文法 …… 46
3.2.1　正规式和上下文无关文法的比较 …… 46
3.2.2　分离词法分析器的理由 …… 47
3.2.3　验证文法产生的语言 …… 48
3.2.4　适当的表达式文法 …… 48
3.2.5　消除二义性 …… 50
3.2.6　消除左递归 …… 50
3.2.7　提左因子 …… 52
3.3　自上而下分析 …… 52
3.3.1　自上而下分析的一般方法 …… 52
3.3.2　LL(1)文法 …… 54
3.3.3　递归下降的预测分析 …… 55
3.3.4　非递归的预测分析 …… 56

3.3.5 构造预测分析表 …… 59
3.3.6 预测分析的错误恢复 …… 60
3.4 自下而上分析 …… 63
3.4.1 归约 …… 63
3.4.2 句柄 …… 64
3.4.3 用栈实现移进-归约分析 …… 65
3.4.4 移进-归约分析的冲突 …… 66
3.5 LR 分析器 …… 68
3.5.1 构造 SLR 分析表 …… 68
3.5.2 LR 分析算法 …… 73
3.5.3 其他 LR 分析表构造概述 …… 75
3.5.4 非二义且非 LR 的上下文无关文法 …… 77
3.6 语法分析器的生成器 …… 78
3.6.1 分析器的生成器 Yacc …… 78
3.6.2 用 Yacc 处理二义文法 …… 82
3.6.3 Yacc 的错误恢复 …… 84
习题 …… 86
第 4 章 类型检查 …… 91
4.1 语法制导的翻译 …… 91
4.1.1 翻译方案 …… 92
4.1.2 语法树 …… 93
4.1.3 构造语法树的翻译方案 …… 94
4.1.4 翻译方案中属性的自下而上计算 …… 95
4.1.5 设计翻译方案的一些技巧 …… 98
4.2 类型在编程语言中的作用 …… 99
4.2.1 执行错误和安全语言 …… 99
4.2.2 类型化语言和类型系统 …… 99
4.2.3 类型化语言的优点 …… 102
4.3 一个简单类型检查器的规范 …… 102
4.3.1 一个简单的语言 …… 103
4.3.2 类型表达式 …… 103
4.3.3 类型检查 …… 104
4.3.4 类型转换 …… 105
4.4 类型表达式的等价 …… 106
4.4.1 类型表达式的结构等价 …… 107
4.4.2 类型表达式的名字等价 …… 107
4.4.3 记录类型 …… 108
4.4.4 类型表示中的环 …… 109
习题 …… 109
第 5 章 运行时存储空间的组织和管理 …… 114
5.1 局部存储分配 …… 115
5.1.1 过程 …… 115
5.1.2 名字的作用域和绑定 …… 116
5.1.3 活动记录 …… 117
5.1.4 局部数据的布局 …… 117
5.1.5 程序块 …… 118
5.2 全局栈式存储分配 …… 119
5.2.1 运行时内存的划分 …… 120
5.2.2 活动树和运行栈 …… 120
5.2.3 调用序列 …… 122
5.2.4 栈上可变长度数据 …… 125
5.2.5 悬空引用 …… 126
5.3 非局部名字的访问 …… 127
5.3.1 无过程嵌套的静态作用域 …… 127
*5.3.2 有过程嵌套的静态作用域 …… 128
5.4 参数传递 …… 131
5.4.1 值调用 …… 131
5.4.2 引用调用 …… 132
5.4.3 换名调用 …… 132
*5.5 堆管理 …… 133
5.5.1 内存管理器 …… 133
5.5.2 计算机内存分层 …… 134
5.5.3 程序局部性 …… 136
5.5.4 手工回收请求 …… 136
习题 …… 137
第 6 章 中间代码生成 …… 149

6.1 中间语言 …… 150
6.1.1 后缀表示 …… 150
6.1.2 图形表示 …… 150
6.1.3 三地址代码 …… 151
6.1.4 静态单赋值形式 …… 153
6.2 声明语句 …… 154
6.2.1 过程中的声明 …… 154
6.2.2 作用域信息的保存 …… 155
6.2.3 记录的域名 …… 157
6.3 赋值语句 …… 158
6.3.1 符号表中的名字 …… 158
6.3.2 数组元素的地址计算 …… 159
6.3.3 数组元素地址计算的翻译方案 …… 160
6.3.4 类型转换 …… 163
6.4 布尔表达式和控制流语句 …… 164
6.4.1 布尔表达式 …… 164
6.4.2 控制流语句的中间代码结构 …… 165
6.4.3 布尔表达式的回填 …… 166
6.4.4 控制流语句的翻译 …… 168
6.4.5 开关语句的翻译 …… 169
6.4.6 过程调用的翻译 …… 171
习题 …… 172
第7章 代码生成和代码优化 …… 178
7.1 代码生成器设计中的问题 …… 178
7.1.1 目标程序 …… 178
7.1.2 指令选择 …… 179
7.1.3 寄存器分配 …… 180
7.1.4 计算次序选择 …… 180
7.2 目标语言 …… 181
7.2.1 目标机器的指令集 …… 181
7.2.2 指令代价 …… 182
7.3 基本块和流图 …… 184
7.3.1 基本块 …… 184
7.3.2 基本块的优化 …… 185
7.3.3 流图 …… 186
7.3.4 下次引用信息 …… 187
7.4 一个简单的代码生成器 …… 188
7.4.1 寄存器描述和地址描述 …… 189
7.4.2 代码生成算法 …… 190
7.4.3 寄存器选择函数 …… 190
7.4.4 为变址和指针语句产生代码 …… 192
7.4.5 条件语句 …… 193
7.5 代码优化概述 …… 194
7.5.1 优化的主要源头 …… 194
7.5.2 一个实例 …… 194
7.5.3 公共子表达式删除 …… 197
7.5.4 复写传播 …… 199
7.5.5 死代码删除 …… 199
7.5.6 代码外提 …… 200
7.5.7 强度削弱和归纳变量删除 …… 200
习题 …… 202
第8章 编译系统和运行系统 …… 217
8.1 C语言的编译系统 …… 217
8.1.1 预处理器 …… 218
8.1.2 汇编器 …… 219
8.1.3 连接器 …… 221
8.1.4 目标文件的格式 …… 222
8.1.5 符号解析 …… 224
8.1.6 静态库 …… 225
8.1.7 可执行目标文件及装入 …… 227
8.1.8 动态连接 …… 228
8.1.9 处理目标文件的一些工具 …… 230
8.2 Java语言的运行系统 …… 230
8.2.1 Java虚拟机语言简介 …… 231
8.2.2 Java虚拟机 …… 232
8.2.3 即时编译器 …… 233
习题 …… 235

*第 9 章 面向对象语言的编译 …… 239
9.1 面向对象语言的概念 …… 239
9.1.1 对象和对象类 …… 239
9.1.2 继承 …… 240
9.1.3 信息封装 …… 242
9.2 方法的编译 …… 242
9.3 继承的编译方案 …… 245
9.3.1 单一继承的编译方案 …… 246
9.3.2 多重继承的编译方案 …… 248
习题 …… 253
附录 …… 255
附录 1 PL/0 语言及其实现 …… 255
附录 2 基于 PL/0 语言的课程实践选题 …… 283
参考文献 …… 284

第1章 引 论

从理论上说,构造专用计算机来直接执行用某种高级语言编写的程序是可能的。但是,实际上目前的计算机能执行的都是非常低级的**机器语言**。那么,一个基本问题是:用高级语言编写的程序是怎样被变换成能在计算机上执行的机器语言程序的?

能够将一种语言的程序保语义地变换成另一种语言的程序的软件,被称为**翻译器**,这两种语言分别叫做该翻译器的**源语言**和**目标语言**。**编译器**是一种翻译器,它的特点是目标语言比源语言低级。

本章通过简要描述编译器的各个组成部分以及编译器技术的各种应用来介绍编译这个课题。该课题涉及编程语言、计算机体系结构、形式语言理论、类型论、算法和软件工程等方面的知识。

1.1 编译器概述

编译器的工作可以分成若干逻辑**阶段**,每个阶段把源程序从一种表示变换成另一种表示。编译过程的一种典型分解见图 1.1,图中左边一列的每个方框表示它的一个阶段。

本节以 C 语言的赋值语句

$$\text{position} = \text{initial} + \text{rate} * 60 \tag{1.1}$$

的翻译(假定变量都已声明为实型)为例,简要介绍编译的各个阶段。

1.1.1 词法分析

词法分析扫描构成源程序的字符流,按编程语言的词法规则把它们组成词法**记号**(token)流。对于一个词法单元,词法分析产生的记号是

〈记号名,属性值〉

二元组。记号名是同类词法单元共用的名称,而属性值是一个词法单元有别于同类中其他词法单元的特征值。赋值语句(1.1)的字符流在词法分析时被依次组成下面这些记号。

(1) 标识符 position 形成的记号是〈**id**,1〉,其中 **id** 是标识符的总称,1 代表 position 在符号表中的条目,符号表的条目用来存放标识符的各种属性,如它的名字和类型。

(2) 赋值号 = 形成的记号是〈**assign**〉,因为该记号只有一个实例,因此不需要属性值来区分实例。为了直观起见,下面直接用赋值号作为记号名,写成〈=〉。

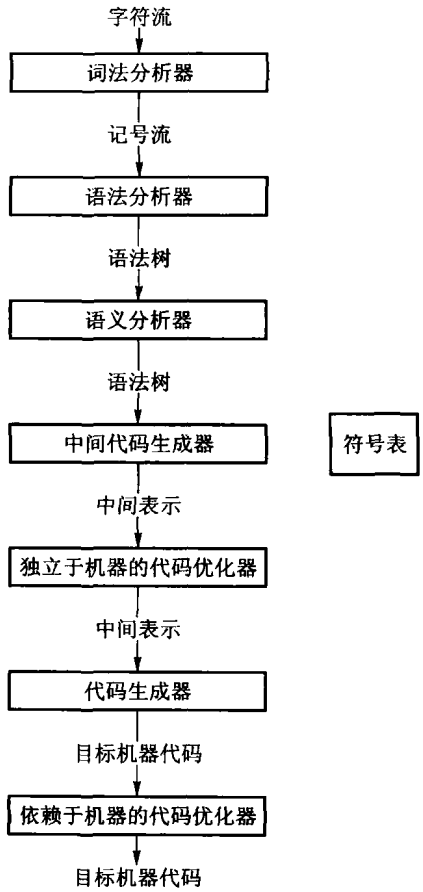

图 1.1 编译的各个阶段

(3) 标识符 initial 形成的记号是〈**id**,2〉。

(4) 加号+形成的记号是〈+〉。

(5) 标识符 rate 形成的记号是〈**id**,3〉。

(6) 乘号 * 形成的记号是〈 * 〉。

(7) 数 60 形成的记号是〈60〉。从技术上讲,程序中出现的常数也要放到符号表或单独的常数表中,形成记号〈**number**,60 在表中的条目〉。为直观起见,这里直接使用 60 在源程序中的字符序列作为记号名。

分隔单词的空格通常在词法分析时被删去。

于是,赋值语句(1.1)在词法分析后形成的记号流是

〈**id**,1〉〈=〉〈**id**,2〉〈+〉〈**id**,3〉〈 * 〉〈60〉 (1.2)

第 2 章讨论词法分析,词法分析也叫做**线性分析**或**扫描**。

1.1.2 语法分析

语法分析(syntax analysis)简称为**分析**(parsing),它按编程语言的语法规则检查词法分析输出的记号流是否符合这些规则,并依据这些规则所体现出的该语言的各种语言**构造**(construct,如函数、语句、表达式等)的层次性,用各记号建成一种树形的中间表示,这种中间表示用抽象语法的方式描绘了该记号流的语法情况。一种典型的中间表示是**语法树**,其中分支结点表示运算,它们的子结点代表该运算的运算对象。为记号流(1.2)建立的语法树见图 1.2(a)。

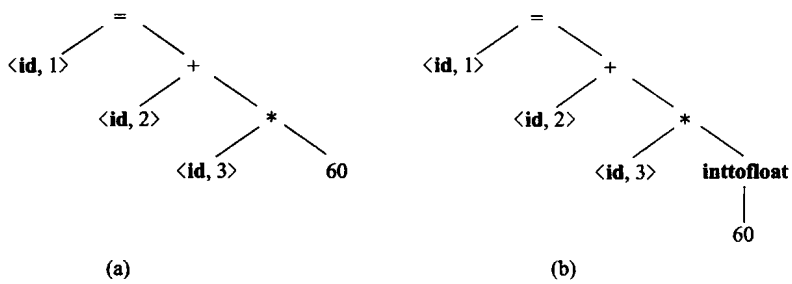

图 1.2 语义分析插入了类型转换

编译器后面各个阶段使用这种语法树,进一步分析源程序和生成目标代码。第 3 章将专门讨论语法分析算法,图 1.2 这种形式的语法树将在 4.1 节讨论。

1.1.3 语义分析

语义分析阶段使用语法树和符号表中的信息,依据语言定义来检查源程序的语义一致性,以保证程序各部分能有意义地结合在一起。它还收集类型信息,把它们保存在符号表或语法树中。

语义分析的一个重要部分是**类型检查**,编译器检查每个算符的运算对象,看它们的类型是否适当。例如,当实数作为数组的下标时,许多语言的定义都要求编译器报告错误。语言定义也可能允许运算对象的类型隐式转换,例如当二元算术算符作用于一个整数和一个实数时,编译器会把其中的整数转换为实数。

例 1.1 在机器内部,整数的二进制表示和实数的二进制表示是有区别的,不论它们是否有相同的值。在图 1.2 中,所有的变量都是实型,另外,由 60 本身可知它是整数。对图 1.2(a)进行类型检查会发现 * 作用于实型变量 rate 和整数 60,可以建立一个额外的算符结点 **inttofloat**(见图 1.2(b)),它显式地把整数转变为实数。 □

类型检查将在第 4 章讨论。

1.1.4 中间代码生成

经过语法分析和语义分析后，许多编译器为源程序产生更低级的显式中间表示，可以把这种中间表示想象成一种抽象机的程序。这种中间表示必须具有两个性质：易于产生，并且易于翻译成目标程序。

第6章主要采用叫做**三地址代码**的中间表示，它们像一种抽象机器的汇编语言，这种机器的每个存储单元的作用类似于寄存器。三地址代码由三地址指令序列组成，每条三地址指令最多有三个操作数，语句(1.1)的三地址代码序列可以如下：

```
t1 = inttofloat(60)
t2 = id3 * t1                    (1.3)
t3 = id2+t2
id1 = t3
```

这种中间形式有它的特点。首先，除了赋值算符外，每条指令至多还有一个算符，因此，在生成这些指令时，编译器必须决定运算完成的次序，语句(1.1)的乘优先于加。其次，编译器必须产生临时变量名，用以保存每条指令的计算结果。最后，某些三地址指令的运算对象不足三个，例如指令序列(1.3)的第一条和最后一条指令。

本书在第6章叙述编译器时用的主要是中间表示。通常，除了计算表达式外，这些中间表示还要做其他事情，例如有条件控制转移、无条件控制转移和过程调用。

1.1.5 代码优化

独立于机器的代码优化阶段试图改进中间代码，以便产生较好的目标代码。通常较好是指执行较快，但也可能期望其他目标，如目标代码较短或目标代码执行时能耗较低。

如果中间代码生成算法比较简单，那么它将给代码优化留下很多机会。例如，一个比较自然的中间代码生成算法，为语法树上的每个算符产生一条指令，因而得到(1.3)这样的中间代码。用这样的中间代码生成算法再跟上一个代码优化阶段是一种可行的办法，因为产生较优代码问题可以在代码优化阶段得以解决。例如，代码优化器会推断出，把60从整数转变为浮点数可以在编译时完成，从而用60.0代替60就可以把**inttofloat**运算删去。还有，t3只被引用一次，就是取它的值传给id1，因此用id1代替t3，把(1.3)的最后一条指令删除也是可以的。这样，优化器可以得到如下结果：

```
t1 = id3 * 60.0
id1 = id2+t1                     (1.4)
```

不同的编译器完成不同程度的优化，能完成大部分优化的编译器叫做“优化编译器”，但这时编译时间中相当可观的一部分都消耗在这种优化上。简单的优化也可以使目标程序的运行时间大大缩短，而编译速度并没有降低太多。第7章将通过例子来介绍独立于机器的优化。

本章所给出的这个例子不涉及依赖于机器的优化，本书也不打算介绍这方面的优化。

1.1.6 代码生成

代码生成取源程序的一种中间表示作为输入并把它映射到一种目标语言。如果目标语言是机器代码，则需要为源程序所用的变量选择寄存器或内存单元，然后把中间指令序列翻译为完成同样任务的机器指令序列。此阶段的一个关键问题是寄存器分配。

例如，使用寄存器 R1 和 R2，(1.4)的中间代码可以翻译成：

```
MOVF   id3,R2
MULF   #60.0,R2
MOVF   id2,R1                                    (1.5)
ADDF   R2,R1
MOVF   R1,id1
```

每条指令的第一个和第二个操作数分别代表源和目的操作数，每条指令的 F 告知指令处理浮点数。(1.5)的代码把地址 id3 的内容取出并存入寄存器 R2(在此认为指令中 id3 代表对象 id3 的地址，变量的存储分配将在第 5 章讨论)，然后把它乘上实数 60.0，#号代表 60.0 作为立即数处理。第三条指令把 id2 的内容存入寄存器 R1，第四条指令再把寄存器 R2 的值加上去，最后一条指令把寄存器 R1 的值存入地址 id1。这样，该段代码实现了语句(1.1)的赋值。代码生成在第 7 章讨论。

1.1.7 符号表管理

编译器的一项重要工作是记录源程序中使用的变量名字，并收集每个名字的各种属性。这些属性提供该名字有关存储分配、类型和作用域等信息。如果是过程名字，还有参数的个数、每个参数的类型、参数传递方式和返回值类型等。

符号表是为每个变量或过程名字保存一个记录的数据结构，记录的域是该名字的属性。该数据结构应该设计成允许编译器迅速地找到一个名字的记录，并在此记录中迅速地存储和读取数据。符号表将在第 6 章讨论。

图 1.3 中总结了语句(1.1)的编译过程。

1.1.8 阶段的分组

前面的介绍把编译器从逻辑上分成了 7 个阶段。最粗略的看法是把编译器分成分析和综合两大部分。分析部分揭示源程序的基本元素和它们所形成的层次结构，确定它们的含义，建立起源程序的中间表示，分析部分经常被称为**前端**。综合部分根据源程序的中间表示建立和源程序等价的目标程序，它经常被称为**后端**。

在实际的编译器中，几个阶段的活动可以组合在一起，这些阶段之间的中间表示也无须显式构造。几个阶段的活动常用一**遍**(pass)扫描来实现，一遍扫描包括读一个输入文件和

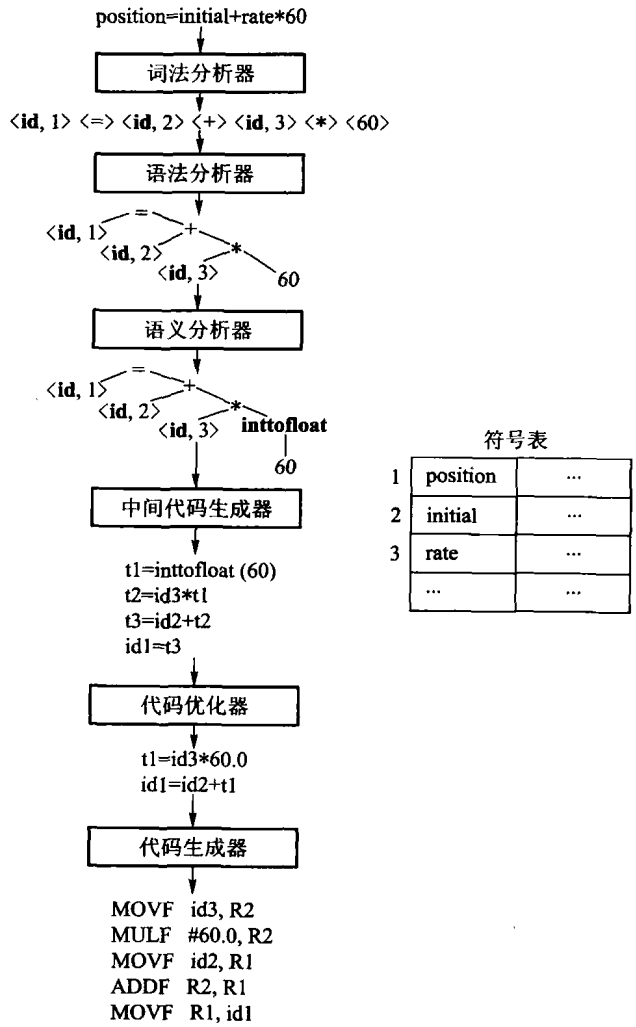

图 1.3 一个语句的翻译

写一个输出文件。例如,前端的词法分析、语法分析、语义分析和中间代码生成可以组成一遍,独立于机器的代码优化可以单独作为可选择的一遍,特定目标机器的代码生成可以作为一个后端遍。

取一个编译器前端,重写它的后端以产生同样源语言在另一种机器上的编译器已经是件普通的事情。如果原先的后端是经过仔细设计的,那么往往不需要对它作很多的重新设计。

把几种不同的语言编译成同一种中间表示,让不同的前端使用同一个后端,从而得到一

种机器上不同源语言的编译器,也已经有不少成功的例子。

1.1.9 解释器

解释器是不同于编译器的另一类语言处理器。解释器不像编译器那样通过翻译来生成目标程序,而是直接执行源程序所指定的运算。解释器也有和编译器类似的地方,它也需要对源程序进行词法分析、语法分析和语义分析等,这样它才有可能知道源程序指定了一些什么运算。

解释执行的效率比编译器生成的机器代码的执行效率低。对于编译方式来说,对源程序的词法分析、语法分析和语义分析只要进行一次。而对于解释执行来说,每次执行到源程序的某个语句,都要对它进行一次词法分析、语法分析和语义分析,确定了这个语句的含义后,才能执行它指定的运算。显然,反复分析循环体降低了解释执行的效率,所以解释执行都要寻找一种适合于解释的中间语言,以降低反复分析源程序需要的时间。

在20世纪80年代的BASIC语言阶段,解释器的功能是这样介绍的:它将高级语言的源程序翻译成一种中间语言程序,然后对中间语言程序进行解释执行。在那个年代,解释器的两个功能(编译和解释)是合在一个程序中的,因此这个程序被统称为解释器。进入Java语言时代,解释器的上述两个功能分在两个程序中,前一个程序叫编译器,它把Java语言的程序翻译成一种中间语言程序,这种中间语言叫做字节码;后一个程序叫做解释器,它对字节码程序进行解释执行。

1.2 编译器技术的应用

真正从事主流编程语言编译器设计的虽然只是极少数一部分人,但是编译器技术还有着其他重要的应用,这是学习编译器技术的主要理由。此外,编译器的设计影响着计算机科学的其他一些领域。本节回顾编译器设计与计算机科学的其他主要领域之间最重要的相互影响以及编译器技术的重要应用。

1.2.1 高级语言的实现

一种高级编程语言定义了一种编程抽象:程序员用这种语言表达算法,编译器将程序翻译到目标语言。一般来说,高级编程语言易于编程,但是所得到的程序运行较慢。用低级语言编程时可以在程序中实施更有效的控制方式,原则上说,能得到更有效的代码。不幸的是,低级语言程序难编写,易出错,更糟糕的是其难维护,缺乏移植性。优化编译器包括了很多改进所生成代码性能的技术,从而弥补了高级抽象导入的低效。

历史上,流行编程语言的大多数改变都是朝着提高抽象级别的方向。20世纪80年代C语言占据了系统编程的统治地位,90年代启动的很多项目选择C++,1995年问世的Java语言在20世纪90年代后期迅速流行。每一轮编程语言新特征的出现都促进了编译器优化的

新研究。下面回顾激励编译器技术显著推进的主要语言特征。

所有通用编程语言,包括C、FORTRAN和COBOL,都支持用户定义的聚合数据类型,如数组和记录,也都支持高级控制流,如循环和过程调用。如果逐句把程序直接翻译成机器代码,则结果程序非常低效。作为编译器优化主体的数据流分析和优化,它用来分析数据通过程序时的流动情况并进行多种优化,所生成代码的效率相当于有经验的程序员用低级语言写出程序的效率。数据流优化至今仍在被广泛研究。

面向对象的概念在1967年首次出现在Simula67语言中,以后逐步被加入各种语言中,如Smalltalk、C++、C#和Java。面向对象的主要概念是数据抽象和性质继承,它们都使得程序更加模块化并易于维护。面向对象的程序不同于用其他语言编写的程序,其中类的定义中可能包含许多较小的过程(在面向对象语言中称为方法)。因此,对于源程序中跨越过程边界的操作,编译器优化必须能够适当处理才能保证效率。在这个场合,用过程体替换过程调用的过程内联(inlining)特别有用。加速虚方法调遣(dispatch)的优化也一直是研究热点。

Java有很多使编程变得容易的特征,其中大部分已存在于先前的语言中。Java语言是类型安全的,它是指一个对象不会被当做一个不相关类型的对象来使用。所有数组访问都被检查以保证这些访问在相应数组的边界内。Java没有指针,也不允许指针算术运算。它用无用单元收集(俗称垃圾收集)机制来自动地回收那些不再使用的变量所占据的内存。这些特征虽然使编程变得容易,但是它们会引起运行开销的增加。相应地,出现了一些用于降低开销的编译器优化技术,例如,删除不必要的边界检查,把离开过程后不再访问的对象分配在栈上而不是堆上。极小化无用单元收集开销的算法也一直在开发。

此外,Java设计成能支持代码移植和代码移动。程序以Java字节码的形式分发,字节码必须被解释或动态地编译成本地代码。在运行时能够收集运行信息的场合,动态编译可用来生成较好地优化的代码。在动态优化中,很重要的一点是极小化编译所需时间,因为它是执行开销的一部分。现在通用的一种技术是仅编译和优化程序中经常执行的部分。

1.2.2 针对计算机体系结构的优化

计算机体系结构的迅速演化对编译器技术不断提出新的需求,几乎所有的高性能系统都在利用两种基本技术:并行化和内存分层。并行性可以在指令级和处理器级分别发掘;内存分层针对这样的基本局限:构造非常快的存储器或者非常大的存储器是可能的,但是构造不出既快又大的存储器。

所有现代微处理器都开拓指令级的并行。这种并行对程序员是隐蔽的,程序员理解为串行执行的指令序列,会被硬件动态地检查其中的相关性,并尽可能并行发出这些指令。在有些情况下,计算机包括一个硬件调度器,它可以改变指令的排序以增加程序中的并行。不管硬件是否重排指令的次序,编译器总可以重新整理指令,使得指令级的并行更有效。

指令级并行也可以显式出现在指令集中,超长指令字计算机有能够进行并行乘法运算