<<编译原理 技术与工具>>

图书基本信息

书名：<<编译原理 技术与工具>>

13位ISBN编号：9787115099167

10位ISBN编号：7115099162

出版时间：2002-2

出版时间：人民邮电出版社

作者：美.阿霍

页数：795

字数：1009000

<<编译原理 技术与工具>>

内容概要

作为编译器设计的教程，本书重点主要放在解决在设计语言翻译器过程中所普遍面对的一些问题上而并不考虑源语言或者目标机器。
本书共12章：第一章介绍了编译器的基本结构；第二章给出了一个将前缀表达式转换成后缀表达式的编译器，主要使用本书的一些基本技巧来构建；第三章阐述了词法分析、正则表达式、有限自动机和扫描生成器工具，这章中的技术广泛应用于文本处理；第四章详细阐述了主要的分析技术，从适合手工实现的递归下降算法到在分析生成器中使用的LR算法；第五章介绍了语法制导翻译中的主要思想，本书的其它部分都用本章来说明和实现翻译；第六章提出了完成静态语义检查的主要思想，并对类型检查和类型的统一进行了详细的讨论；第七章讨论了支持应用程序运行时环境的存储组织；第八章从中间语言的讨论开始，说明了编程语言结构翻译成中间代码；第九章阐述了目标代码的生成，包含基本的on_the_fly代码生成方法、为表达式生成代码的优化方法、Peephole优化和代码生成器；第十章是代码优化的总述。
除了关于数据流分析方法的详细说明，还有关于如何进行全局优化的基本方法；第十一章讨论了在编译器实现过程中可能会产生的一些实际问题；第十二章提出一些使用本书中的技术构建的一些编译器的学习用例。
本书可作为高校计算机专业本科和研究生编译原理的教科书，也可供从事计算机软件开发的人员参考。

作者简介

Alfred V.Aho是美国AT&T贝尔实验室计算机原理研究员的负责人。
他在多伦多大学获得工程物理专业应用科学学士学位，在普林斯顿大学获得博士学位。

书籍目录

有一个额外的优点就是分析器构造过程可以显示出语法的二义性以及一些其它较难分析的结构，而那些结构很可能在一个语言和编译器的最初设计阶段是很难检查出来的。
l　恰当设计的文法将一个结构传给应用程序语言。
这个程序语言对于将源程序翻译成正确的目标代码以及进行错误的检测都是非常有用的。
工具有利于将基于文法的翻译描述转换成能用的应用程序。
l　随着时间的推移，语言增加了一些新的结构，同时又实现了一些别的工作。
这些新的结构可以更容易地添加到语言中，特别是当存在一个基于语言文法描述的实现时。
这一章描述了编译器中使用的分析方法。
我们提出的首先是基本概念；接着是适合手工实现的技术；最后是自动化工具中使用的算法。
由于程序可能包含语法错误，我们扩展了这个分析方法以使它们能够从产生的普通错误中恢复。
第5章　语法制导翻译5.1　语法制导定义5.2　语法树的构造5.3　S－属性定义的自下而上的求值5.4　L－属性义5.5　自顶向下的翻译5.6　继承性属性自下而上的求值5.7　递归求值5.8　编译时属性值的空间5.9　编译器时分配空间5.10　语法制导定义的分析小结：这一章发展了2.3节的主题：上下文无关文法制导的语言翻译。
我们将信息同程序语言构造联系起来，主要是通过将属性挂接到代表结构的文法象征上实现的。
属性的值由同文法产生式相关联的语义规则来计算。
将语义规则同产生式联系要注意两点：语法制导的定义和翻译模式。
语法制导定义是翻译的高层次的说明。
它们隐藏了许多实现细节从而将用户从不得不明确说明翻译发生的序列中解脱出来。
翻译模式表明语义规则求值的顺序，从而允许一些实现细节暴露出来。
我们在第六章使用这两点来说明语义检查，特别是类型判断，而在第八章使用它们产生中间代码。
概念上，使用语法制导定义和翻译模式，我们将分析输入标号流，建立分析树，接着根据需要遍历树从而对分析树节点上的语义规则求值，将信息保存在符号表中，发布错误信息，或者完成其它一些动作。
标号流的翻译是通过求值语义规则获得的结果。
输入字符串　分析树　依赖表　语义规则的求值顺序语法制导翻译的概念视图一个实现并不一定需要完全符合上图。
语法制导定义的特例也可以通过一遍来实现。
主要是通过对遍中的语义规则进行求值而不是明显的定义一个分析树或者一个图表来显示属性之间的依赖关系。
由于一遍的实现对于编译的高效性有重要影响，这一章主要研究这些特例。
一个重要的子类叫做L属性的算法包含了实际上所有的翻译，而这些翻译无需明确构造分析树就能实现的。
第6章　类型检查6.1　类型系统6.2　一个简单的类型检查器的说明6.3　类型表达式的等价性6.4　类型转换6.5和操作符的重载6.6　多态函数6.7　统一性的一个算法小结：编译器应该检查源程序是否遵循源语言的语法和语义。
这种检查，叫做静态检查（将它同目标程序执行时的动态检查区分开来），保证了特定类型的程序错误将被检测和报告出来。
静态检查的例子包括：1　类型检查。
比如如果一个操作符应用到一个不兼容的操作数中时，编译器应该报告出错；举例来说，如果一个数组变量和一个函数变量相加的话。
2　控制流检查。
存在导致一个控制流离开构造器的语句。
举例来说，C语言中的break语句将导致控制流离开最小的循环while，for和switch语句；如果这样一个包含语句不存在的话将导致错误。
3　唯一性检查。
有些情况下一个对象只能被定义一次。

举例来说，Pascal语言中，标识符应该被声明是唯一的，case语句中的标签也应该是唯一的。
4　相关名称检查。
有时候，同样的名字会多次出现。
举例来说，Ada中，循环或块中都将有一个名字同时出现在构造器的开始和结束。
编译器将检查同样的名字可以在两端被使用。
这一章中，我们着重于类型检查。
像上面的例子表示的，大多数静态检查都是惯例程序并且可以运用上一章的技术实现。
其中的一些可以被集成到其它一些行为中。
举例来说，当我们将名字的信息存到符号表中时。
我们可以确定名字的声明是唯一的。
许多Pascal编译器通过分析将静态检查和中间代码生成集成为一部分。
对于更多的复杂的结构，就象Ada的，很可能是在分析和中间代码生成之间有一个独立的类型检查的遍比较方便。
类型检查核实结构的类型同它期待的环境相匹配。
举例来说，Pascal中算术运算符mod需要整型操作数，因此一个类型检查器应该保证mod的操作数都是整数类型，同样的，类型检查器核实引用必须用到一个指针，索引只在数组上操作，而一个用户定义的函数应用时参数个数和类型必须正确等等。
一个简单的类型检测器的规则出现在6.2。
类型表示和什么时候两种类型匹配的问题将在6.3节讨论。
由类型检查器收集的类型信息可能是需要的，特别是当生成代码的时候。
举例来说，算术操作符像"＋"通常应用到每一个整数和实数上。
也有其它类型的可能，而且我们不得不观察一下"＋"的上下文来决定它导向的意思。
在不同的上下文中代表不同操作的符号叫做重载。
重载伴随着强制类型转换，编译器提供了一个操作符将操作数转换成上下文期待的类型。
重载的明确概念就是多态。
多态函数的主体是可以通过多种类型的参数来执行的。
这一章还包括推断多态函数的类型的统一算法。
第7章　运行时环境7.1　源语言问题7.2　存储组织7.3　存储分配策略7.4　非局部名称的访问7.5　参数传递7.表7.7　动态存储分配的语言工具7.8　动态存储分配技术7.9　Fortran中的动态存储分配小结：在考虑代码生成之前，我们需要将一个应用程序的静态源文本同运行时的行为联系起来来实现应用程序。
随着执行代码的继续，源文本中的同一个名字在目标机器中可能意味着不同的数据对象。
这儿就讨论名字和数据对象之间的关系。
数据对象的分配和回收由run-time support package进行管理，包括随生成的目标代码一起装载的例行程序。
Run-time support package的设计受到过程的语义影响。
像Fortran，Pascal和Lisp语言的支持包也可以使用本章中的技术来构建。
每一个程序的执行都可看作是这个过程的activation。
如果一个程序是递归的，那么它的几个动作应该是同时被激活的。
每一个过程的调用都有可能导致分配给它使用的数据对象的操作激活。
运行时数据对象的表示由它的类型决定。
经常的，像字符，整数和实数这样的元素数据类型，就是由目标机器中等价的数据对象表示的。
可是，像数组，字符串和结构等聚集，通常由原始对象的集合来表示。
第8章　中间代码的生成8.1　中间语言8.2　声明8.3　分配语句8.4　布尔表达式8.5　Case语句8.6　回填8.7　小结：编译器分析合成的模型中，前端将源程序翻译成中间表示，然后后端生成目标代码。
目标语言的细节尽可能地限制在后台。
尽管一个源程序能被直接翻译成目标语言。
但使用独立于机器的中间形式也是有好处的：1.　有利于重定位；不同机器的编译器可以通过将新机器

的后端挂接到现存的前端来创建。
2. 独立于机器的代码优化器可以被应用到中间代码的表示上。
这一章展示了第2章和第5章的语法制导方法如何将声明、赋值以及流控制语句等翻译成它们的中间形式的编程语言结构。
而为简单起见，我们假定源程序已经通过编译和静态检查，大多数语法制导定义都能通过自下向上或者自上而下的分析来实现，因此中间代码生成可根据具体需要合成到分析里。
第9章 代码生成9.1 代码生成器设计中的一些问题9.2 目标机器9.3 运行时存储管理9.4 基本块和流程图一步要使用的信息9.6 一个简单的代码生成器9.7 寄存器分配9.8 基本块的dag代表9.9 Peephole 优化9.10 从dags产生代码9.11 动态编程代码生成算法9.12 代码－生成生成器小结：编译器模型的最后阶段是代码生成器。
它将源程序的中间表示作为输入，从而产生等价的目标程序作为输出。
这章中使用的代码生成技术可以广泛应用，尽管在代码生成之前优化器阶段未必就会发生像在一些所谓的优化编译器中那样。
这样的阶段企图将中间代码翻译成表单因此会产生更加有效的目标代码。
代码优化将在下一章中详细讨论。
传统上对代码生成器的要求是很严格的。
输出代码应该是正确的并具有较高的质量，这意味着能更好的利用目标机器的资源。
还有，代码生成器本身是非常高效的。
可是数学上来说，生成优化代码的问题是不确定的。
实际上，我们应该满足于启发式技术来产生较好的，并一定是最高效的代码。
启发式的选择是重要的，特别是精心设计的代码生成算法能够很容易产生一些代码，它运行起来能比匆忙设计的算法产生的代码快好几倍。
第10章 代码优化10.1 简介10.2 优化的主要资源10.3 基本块的优化10.4 流程图的循环10.5 介绍全局数析10.6 数据流方程的交互策略10.7 代码优化的改造10.8 处理别名10.9 结构流图的数据流分析10.10 高效数据流算法10.11 数据流分析的工具10.12 类型的评估10.13 优化代码的符号性调试小结：理想中，编译器应该产生像手写的一样好的目标代码。
事实是这种目标只会在有限的用例中实现，而且难度还比较高。
可是，直接编译算法产生的代码通常运行的比较快或者使用较少的空间，或者两个特征同时具备。
这个改善是通过传统上叫做优化的程序改造来实现的，尽管"优化"这个术语是一个误导，因为实际上它很少能保证结果代码是尽可能优化的。
应用代码优化改造的编译器叫做优化编译器。
本章重点是独立于机器的优化，程序改造无须考虑目标机器的任何属性就可以改善目标代码。
而像寄存器分配和特殊的机器指令序列等这些依赖于机器的优化已经在9章中讨论了。
最少的努力获得最好的效果就是我们找出那些常用程序的执行部分，并使这些部分产生尽可能高的效果。
有一个比较流行的说法：大多数的应用程序在10%的代码上花费了90%的执行时间。
尽管实际的百分比会发生变化，通常情况下，小部分程序确会占据大多数的运行时间。
从输入代表性数据看应用程序运行时的执行时间可准确地找出了问题的吃重运行区域。
不幸的是，一个编译器通常没有输入数据的例子，因此它应该尽可能猜测问题热点所在。
实际上，应用程序的内在循环是改善应用程序的一个极好的侯选。
在一个强调像while和for控制结构的语言中，循环从程序的语法角度看可能是非常明显的；通常情况下，一个叫作控制流分析的过程在程序的流程图中找出循环。
这章是一个有用的优化改造和实现它们的技术的丰富集合。
编译器中进行什么样的改造是值得的决定这一点的最好的技术就是收集关于源程序的统计数据并且评估源程序典型例子上给定优化集合的好处。
第12章描述了在对不同语言的编译器优化中证明是有用的改造。
这章的主题是数据流分析，一个收集应用程序中使用变量方法的信息的过程。

一个应用程序中不同点收集的信息可以通过简单的集等价方程联合起来。
我们展示了几个使用数据流分析收集信息的算法并在优化中高效使用这些信息。
我们仍然考虑过程和指针等语言结构对优化的影响。
第11章　如何写编译器11.1　规划一个编译器11.2　编译器开发的方法11.3　编译器开发环境11.4　测试和维护
结：看了这些编译设计的原则，技术和工具，假定我们需要编写一个编译器：如果提前进行规划的话，我们可以进行的更快更好。
这章提出了一些编译器构建中出现的实现问题。
大多数讨论注意力集中在使用UNIX操作系统和它的工具来编写编译器上。
第12章　看一下一些编译器12.1 排版数学的预处理器，即EQN12.2 Pascal编译器12.3 C编译器12.4 Fortran H编译器12.5 Bliss／11编译器12.6 Modula－2优化编译器小结：讨论了Pascal、C、Fortran、Bliss和Modula 2的编译器。
我们的意图并不是支持某项设计而排除其它的，而是企图描述编译器实现过程中可能出现的各种情况。

版权说明