目录

[第1章 引论 3](#_Toc30330889)

[1.1 语言处理器 3](#_Toc30330890)

[1.2 一个编译器的结构 3](#_Toc30330891)

[1.3 程序设计语言的发展历程 4](#_Toc30330892)

[1.4 构建一个编译器的相关科学 4](#_Toc30330893)

[1.5 编译技术的应用 4](#_Toc30330894)

[1.6 程序设计语言基础 5](#_Toc30330895)

[第2章 一个简单的语法制导翻译器 6](#_Toc30330896)

[2.1 引言 6](#_Toc30330897)

[2.2 语法定义 6](#_Toc30330898)

[2.3 语法制导翻译 6](#_Toc30330899)

[2.4 语法分析 7](#_Toc30330900)

[第3章 词法分析 7](#_Toc30330901)

[3.1 词法分析器的作用 7](#_Toc30330902)

[3.2 输入缓冲 8](#_Toc30330903)

[3.3 词法单元的规约 8](#_Toc30330904)

[3.4 词法单元的识别 8](#_Toc30330905)

[3.5 词法分析器生成工具Lex 9](#_Toc30330906)

[3.6 有穷自动机finite automata 9](#_Toc30330907)

[3.7 从正则表达式到自动机 10](#_Toc30330908)

[3.8 词法分析器生成工具的设计 11](#_Toc30330909)

[3.9 基于DFA的模式匹配器的优化 11](#_Toc30330910)

[第4章 语法分析 12](#_Toc30330911)

[4.1 引论 12](#_Toc30330912)

[4.2 上下文无关文法 13](#_Toc30330913)

[4.3 设计文法 14](#_Toc30330914)

[4.4 自顶向下的语法分析 15](#_Toc30330915)

[4.5 自底向上的语法分析 15](#_Toc30330916)

[4.6 LR语法分析技术介绍：简单LR技术 16](#_Toc30330917)

[4.7 更强大的LR语法分析器 17](#_Toc30330918)

[4.8 使用二义性文法 18](#_Toc30330919)

[4.9 语法分析器生成工具 18](#_Toc30330920)

[第5章 语法制导的翻译 18](#_Toc30330921)

[5.1 语法制导定义 18](#_Toc30330922)

[5.2 SDD的求值顺序 19](#_Toc30330923)

[5.3 语法制导翻译的应用 20](#_Toc30330924)

[5.4 语法制导的翻译方案 20](#_Toc30330925)

[5.5 实现L属性的SDD 21](#_Toc30330926)

[第6章 中间代码生成 21](#_Toc30330927)

[6.1 语法树的变体 21](#_Toc30330928)

[6.2 三地址代码 21](#_Toc30330929)

[6.3 类型和声明 22](#_Toc30330930)

[6.4 表达式的翻译 23](#_Toc30330931)

[6.5 类型检查type checking 23](#_Toc30330932)

[6.6 控制流 24](#_Toc30330933)

[6.7 回填backpatching 25](#_Toc30330934)

[6.8 switch语句 25](#_Toc30330935)

[6.9 过程的中间代码 25](#_Toc30330936)

[第7章 运行时刻环境run-time environment 25](#_Toc30330937)

[7.1 存储组织 26](#_Toc30330938)

[7.2 空间的栈式分配 26](#_Toc30330939)

[7.3 栈中非局部数据的访问 26](#_Toc30330940)

[7.4 堆管理 27](#_Toc30330941)

[7.5 垃圾回收概述 28](#_Toc30330942)

[7.6 基于跟踪的回收的介绍 29](#_Toc30330943)

[7.7 短停顿垃圾回收 30](#_Toc30330944)

[7.8 垃圾回收中的高级论题 31](#_Toc30330945)

[第8章 代码生成 31](#_Toc30330946)

[8.1 代码生成器设计中的问题 32](#_Toc30330947)

[8.2 目标语言 32](#_Toc30330948)

[8.3 目标代码中的地址 33](#_Toc30330949)

[8.4 基本块和流图 33](#_Toc30330950)

[8.5 基本块的优化 33](#_Toc30330951)

[8.6 一个简单的代码生成器 34](#_Toc30330952)

[8.7 窥孔优化peephole optimization 35](#_Toc30330953)

[8.8 寄存器分配和指派 35](#_Toc30330954)

[8.9 通过树重写来选择指令 35](#_Toc30330955)

[8.10 表达式的优化代码的生成 36](#_Toc30330956)

[8.11 使用动态规划的代码生成 37](#_Toc30330957)

[第9章 机器无关优化 37](#_Toc30330958)

[9.1 优化的主要来源 37](#_Toc30330959)

[9.2 数据流分析简介data-flow analyze 38](#_Toc30330960)

[9.3 数据流分析基础 39](#_Toc30330961)

[9.4 常量传播 40](#_Toc30330962)

[9.5 部分冗余消除 41](#_Toc30330963)

[9.6 流图中的循环 42](#_Toc30330964)

[9.7 基于区域的分析region-based analysis 43](#_Toc30330965)

[9.8 符号分析 44](#_Toc30330966)

[第10章 指令级并行性 45](#_Toc30330967)

[10.1 处理器体系结构 45](#_Toc30330968)

[10.2 代码调度约束 46](#_Toc30330969)

[10.3 基本块调度 47](#_Toc30330970)

[10.4 全局代码调度global scheduling 47](#_Toc30330971)

# 第1章 引论

## 1.1 语言处理器

源语言——编译——目标语言

编译器compiler：“翻译”源程序（高级语言编写的）至目标语言（亦即机器语言）的程序

解释器interpreter：不“翻译”源程序，逐个语句执行源程序，慢但是纠错能力强

预处理器preprocessor：把不同位置的源程序聚合在一起，并替换宏定义的程序

汇编器assembler：将汇编语言程序转化为机器语言的程序，有时是某些编译器的一部分或中间步骤（先编译成汇编语言程序便于调试，后编译为目标语言）

链接器linker：连接不同位置的目标文件至一起的程序

加载器loader：把所有可执行目标文件放到内存中执行

## 1.2 一个编译器的结构

编译过程包括：分析和综合

分析analysis：编译器的前端front end；检查语法，创建一个中间结构表示源程序

综合synthesis：编译器的back end后端，将中间结构和符号表构造成目标程序

编译过程或步骤phase：p3/20

编译过程举例p4/21

1.2.1 词法分析lexical analysis

也称扫描scanning，将源程序代码字符转化为词法单元流

词素lexeme：源代码中的有意义的字符实体，如变量名、数值、运算符

词法单元token：一种由单个词素构建的中间数据结构，包含词法单元名称与属性值（对应符号表项）

1.2.2 语法分析syntax analysis

也称解析parsing，将词法单元流转化为语法树，表达语句逻辑

语法树syntax tree：一种由词法单元作为节点，能够表达语句逻辑的树形数据结构。内部节点表示运算，其子节点表示运算分量。语句的逻辑顺序从下至上最后到根节点。

1.2.3 语义分析semantic analyzer

检查并细化语法树的内容

重要组成部分是类型检查type checking，即运算分量和运算符是否满足类型的限制

自动类型转换coercion

1.2.4 中间代码生成

经过词法语法语义分析后，编译器生成的一个确切的低级的中间代码，这种中间代码具有汇编语言的特征，易于生成和转化为机器语言。

三地址代码three-address code：一种中间表示形式，每个指令最多只有3个运算分量

1.2.5 代码优化

分为机器无关和机器相关的优化，用来缩短指令长度以提高运行速率

1.2.6 代码生成

机器语言：广义上讲，目标语言不一定是机器语言，这个词应该作为字面意义通过上下文来理解其含义。机器语言的特点是对寄存器和内存位置的直接操作。

1.2.7 符号表管理

符号表symbol table：用于存储源程序中和变量、函数相关信息的数据结构，比如名称、作用域、参数类型与数量等。

1.2.8 将多个步骤组合成趟

趟pass：多个步骤phase的组合

1.2.9 编译器构造工具

一些用于构造编译器的编程工具p7/24

## 1.3 程序设计语言的发展历程

1.3.1 走向高级程序设计语言

语言分类——世代：  
第一代：机器语言  
第二代：汇编语言  
第三代：高级程序设计语言  
第四代：为特定应用设计的语言  
第五代：基于逻辑和约束的语言

语言分类——程序命令方式：  
强制式imperative  
声明式declarative

语言分类——模型：  
冯诺依曼von Neumann language语言  
面向对象语言object-oriented language  
脚本语言scripting language

1.3.2 对编译器的影响

## 1.4 构建一个编译器的相关科学

1.4.1 编译器设计和实现中的建模

1.4.2 代码优化的科学

## 1.5 编译技术的应用

1.5.1 高级程序设计语言的实现

1.5.2 针对计算机体系结构的优化

并行parallelism  
非常长指令字VLIW，Very Long Instruction Word

内存层次结构memory hierarchy

1.5.3 新计算机体系结构的设计

现代编译器在计算机处理器设计阶段就进行开发，可以编译并运行于模拟器上，用以评估设计的处理器。

精简指令集计算机RISC，Reduced Instruction-Set Computer：只有少量指令可供使用的计算机体系。指令集中的指令不是高级语言，也不是汇编语言，而是机器语言或处理器本身可以执行的最基本任务的指令。

复杂指令集计算机CISC，Complex Instruction-Set Computer

通用处理器体系结构：  
PowerPC, SPARC, MIPS, Alpha, PA-RISC, x86

专用体系结构：计算机的体系结构概念（处理器结构）  
数据流机器、向量机、VLIW，SIMD（单指令多数据）处理器阵列、心动阵列（systolic array）、共享内存的多处理器、分布式内存的多处理器

1.5.4 程序翻译

①二进制翻译

将一个机器的二进制代码翻译成另一个机器的二进制代码，需要应对不同处理器体系结构之间的指令集区别

②硬件合成

高级硬件描述语言：用于硬件设计的语言，通常是描述寄存器组合逻辑的形式，即在寄存器传输层（RTL，register transfer level），如Verilog，VHDL（very high-speed integrated circuit hardware description language）甚高速集成电路硬件描述语言

硬件合成工具可以将RTL翻译成为门电路，进而翻译为晶体管物理布局，当然也包括电路优化。

③数据查询解释器

SQL的编译和解释

④编译然后模拟

1.5.5 软件生产率工具

即软件工程开发中使用的代码错误探测器和技术

①类型检查

②边界检查

缓冲区溢出问题

③内存管理工具

垃圾收集机制

## 1.6 程序设计语言基础

1.6.1 静态和动态的区别

静态static-编译时刻compile time

动态dynamic-运行时run time

静态作用域static scope-词法作用域lexical

动态作用dynamic scope

1.6.2 环境与状态

环境environment：从一个变量名称到一个内存位置的映射

状态state：从一个内存位置到一个值的映射

1.6.3 静态作用域和块结构

块结构block structure的静态作用域规则p18/35

1.6.4 显式访问控制

1.6.5 动态作用域

例子：宏扩展和多态性

1.6.6 参数传递机制

①值调用call-by-value

②引用调用call-by-reference

③名调用（已弃用）

1.6.7 别名alias

# 第2章 一个简单的语法制导翻译器

## 2.1 引言

略

## 2.2 语法定义

上下文无关文法：也称BNF（Backus-Naur）范式

产生式production：一种语句的构造规则，包括头部或左部（一个非终结符号），一个箭头，体或右部组成（非终结符号与终结符号的序列），亦即理解为某个非终结符号的可能的细分结构或书写形式 p26/43

终结符号terminal：亦即词法单元（实际上，词法单元还包括符号表关联的值，但是在语法分析时用不到，因此近似二者定义等价），关键字、运算符号、变量等无法细分的词法元素

非终结符号nonterminal：也称语法变量，表示终结符号的序列，即可以再细分的符号，如表达式

2.2.1 文法定义

上下文无关文法context-free grammar的组成元素：  
终结符号集合  
非终结符号集合  
产生式集合  
开始符号：一个指定的非终结符号，从该非终结符号开始用产生式推导可得整个语言

空串empty string：零个终结符号组成的串

2.2.2 推导

2.2.3 语法分析树

语法分析的任务：以一个终结符号串作为输入，即源程序语句，然后从文法的开始符号开始推导试图得到该串，否则报告语法错误。

语法分析树的定义：p28/45

结果yield：一棵语法分析树的叶子结点从左向右构成

文法语言是指任何能够由某棵语法分析树生成的符号串的集合。

2.2.4 二义性ambiguous

多棵语法分析树能够生成同一个给定终结符号串，而不同的语法分析树显然代表不同的含义，即便具有相同的终结符号串，运算顺序也不同。这显然是错误的

2.2.5 运算符的结合性

加减乘除是左结合运算符，即若一个变量左右两侧都有+号，该量属于左边的运算符。

指数、赋值系列运算符是右结合的。

结合的左右性规定了连续的同级运算符之间的运算顺序。

2.2.6 运算符的优先级

使用不同的非终结符号表示不同优先级运算符的运算式，低级运算符对应的非终结符号可分解为更高级非终结符号之间的低级运算。p31/48

## 2.3 语法制导翻译

概述：通过向一个文法的产生式附加规则或程序片段而得到的一种翻译方案translation scheme

2.3.1 后缀表示postfix notation

定义见p33/50

后缀表示不需要括号定义运算顺序，运算符的顺序就代表了运算顺序

2.3.2 综合属性

详见后文

2.3.3 简单语法制导定义simple

即语法制导定义的语义规则和产生式本身非常相似的情况——这种相似体现在符号串的出现顺序，以及没有除了添加简单额外字符外的复杂操作。

2.3.4 树的遍历

略

2.3.5 翻译方案

略

## 2.4 语法分析

p54

# 第3章 词法分析

## 3.1 词法分析器的作用

1）扫描阶段：完成一些不需要生成词法单元的简单处理，对注释、空白的处理

2）词法分析阶段：生成词法单元

3.1.1 词法分析及语法分析

把二者区分的目的：  
简化编译器设计  
提高编译器效率  
增强编译器可移植性

3.1.2 词法单元、模式和词素

词法单元：一个词法单元名和一个可选的属性值组成。不同的词法单元类型的属性值一般定义也不同，对于标识符id的词法单元是一个指向符号表中对应条目的指针；数值型number词法单元的属性值是一个指向代表该数值的字符串的指针，关系运算符relop的词法单元其属性值是符号常量，如LT，LE，GT，GE等。简而言之，属性是精确定义词法单元大类中的具体含义的量。

模式：一个词法单元的词素可能具有的形式

词素：源程序中的一个字符序列

·词法单元和词素的例子：p69/86

·所有程序设计语言中典型的词法单元：p70/87

3.1.3 词法单元的属性

略

3.1.4 词法错误

简单的或交互式计算环境下词法错误的自动恢复策略：p71/88

·这些策略的核心思想是，一般的词法范畴错误都只涉及一个字符

## 3.2 输入缓冲

3.2.1 缓冲区对

使用一对输入缓冲区和两个指针进行词法单元识别，双缓冲的意义在于不让缓冲区的结尾掐断任何一个可能的词法单元

3.2.2 哨兵标记sentinel

使用一个eof标志作为缓冲区结尾的哨兵可以优化算法

·注意，词法分析器最好但并不一定需要解决语句中的全部定义解析，语法分析器可以后续处理一些二义性。

## 3.3 词法单元的规约

3.3.1 串和语言

【优化】switch多路分支的实现优化：p74/91

串string，也称句子和字。串相关的术语解析，前缀prefix，后缀suffix，子串substring，真子串，子序列subsequence，连接concatenation

串也可以基于视连接为乘法操作而定义指数操作，注意指数为0的串是空串，因为空串和任何串的乘积都是其本身

3.3.2 语言上的运算

最重要的语言运算是并集、连接和Kleene闭包：p75/92

3.3.3 正则表达式

使用语言符号和语言运算符组成的表达式，用以表达一种语言模式或一个词法单元的形式。可以仅用一个正则表达式定义的语言叫做正则集合regular set。

·注意，正则表达式和上下文无关文法都可以描述一种语言构造模式，但是后者功能更强或者说包含了前者能够表达的语言集合（当然上下文无关文法也只是全部文法的子集）。正则表达式只是用于词法单元识别为的是速度和效率起见。

归纳基础：p76/93

归纳步骤：p76/93

正则表达式的代数定律：p77/94

3.3.4 正则定义regular definition

将一些只包含基本语言符号的正则表达式用一个标识符（斜体）代替，从而简化更加复杂的正则表达式书写。

正则定义：使用基本符号组成的正则表达式和标识符化的正则表达式组合定义的语言规约。

p77/94

3.3.5 正则表达式的扩展

介绍一些正则表达式的扩展表示法：p78/95

## 3.4 词法单元的识别

词法单元等于文法中的终结符号，这是不同视角下的区分。

·让关键字（如if，else，do等）作为保留字，即不能作为标识符，非常有助于简化词法单元的识别。

3.4.1 状态转换图transition diagram

转换图的重要约定：p82/99

·接受状态或最终状态

·forward指针回退

·开始状态或初始状态

3.4.2 保留字和标识符的识别

两种识别方法：p83/100

·将保留字填入符号表，并赋予特殊的属性

·为每个关键字建立单独的状态转换图，必须定义识别保留字的优先级更高

3.4.3 完成我们的例子

略，更多的状态转换图例子

3.4.4 基于状态转换图的词法分析器的体系结构

代码角度来说，状态转换图由多路分支语句switch主导实现。一般会将所有的状态转换图（每个词法单元一个）合并成为统一的一个图进行分析，虽然这种策略也有一定的难度。

## 3.5 词法分析器生成工具Lex

名称：Lex编译器

功能：使用正则表达式描述各个词法单元的模式，给出一个词法分析器的规约

输入表示方法：Lex语言

3.5.1 Lex的使用

Lex的工作流程：p89/106

3.5.2 Lex程序的结构

Lex语言描述的程序结构形式：p89/106

·声明部分：变量和明示常量manifest constant，以及正则定义

·转换规则：按照“模式 {动作}”的形式定义，模式及词法单元模式，动作是对应的触发代码

·辅助函数：各个动作所需要的函数，是可选的一种分离定义

Lex中的非字母和非数字符号往往带有特殊功能，如果需要使用符号本身的表示，用\反斜线转义即可，这和UNIX类似

3.5.3 Lex中的冲突解决

两个规则

·总是选择最长的前缀匹配

·如果仍然冲突，总是选择先列出的模式（一般保留字放在前面）

3.5.4 向前看运算符p92/109

定义：有时（尤其是关键字不是保留字的情况）词法单元的匹配需要向前检查若干个字符才能确定，这些若干个字符并非词法单元的一部分，仅仅相当于附加模式。此时可以使用斜线/（不是或|）作为向前看运算符使用。

## 3.6 有穷自动机finite automata

识别器recognizer：对每个输入串只能回答是或否，有穷自动机是此类

有穷自动机分为两类：

不确定的有穷自动机nondeterministic finite automata，NFA

确定的有穷自动机deterministic finite automata，DFA

两种有穷自动机能识别的语言集合是相同的（二者可以相互转换，即用不同的写法描述），而且和正则表达式能够描述的语言集合相同，这种语言集合也称正则语言regular language

3.6.1 不确定的有穷自动机

NFA的组成：p93/110

特点：

·状态转换不确定，即从一确定状态根据一个确定输入可以得出不同的状态

·允许输入为空串

不确定的有穷自动机的转换图和前文的状态转换图不一样，前文的状态转换图实际上是DFA的转换图。

3.6.2 转换表transition table

举例：p94/111

优点：清晰

缺点：占据大量空间

3.6.3 自动机中输入字符串的接受

只要有一条路径（开始状态到接受状态）上的连续输入和字符串相符即可。注意，开始状态是唯一确定根本不可能为接受状态的状态。

3.6.4 确定的有穷自动机

DFA是NFA的一个特例，满足条件：

·不许空串输入

·对每个状态和每个输入符号，有且只有一条对应该符号的边离开该状态

DFA模拟算法，即用DFA检验字符串的算法流程（非常简单）：p95/112

move函数的实现基本都基于图数据结构和图搜索算法。

## 3.7 从正则表达式到自动机

DFA因其确定性是构造词法分析器来实现的首选。实际上NFA可由正则表达式直接得来，NFA和DFA也可以转换，且NFA也可以直接实现（用于转换为DFA更加费时的情况）。

3.7.1 从NFA到DFA的转换

理论上，DFA的状态数目应该为对应的NFA状态数目的指数倍，但是词法分析领域的实践中，二者的状态数目差不多。

子集构造subset construction：一种由NFA转换到DFA的算法，p97/114

包括辅助函数的算法

3.7.2 NFA的模拟

一个NFA的实现算法（简单的图搜索算法）p99/116

3.7.3 NFA模拟的效率

从具体实现代码的角度分析时间复杂度

3.7.4 从正则表达式构造NFA

McMaughton-Yamada-Thompson算法：p101/118

基本规则：对空集和不包含运算符的子表达式的转换

归纳规则：对包含“或|，和，以及任意数量\*”的子表达式的转换

本节有示例：此算法需要利用语法分析树拆解正则表达式来指导构造过程

3.7.5 字符串处理算法的效率

DFA和NFA处理字符串和创建开销汇总表：p104/121

应用选择：

·频繁使用的字符串处理程序应使用DFA模拟，其额外代价可以被均摊

·偶尔使用的字符串处理器使用NFA直接模拟即可。

·折中混合：仍然先生成NFA，但是仅针对当前的字符串生成对应的DFA，而非一次性全部生成，之后再次遇到同类字符串的输入可以直接查表

## 3.8 词法分析器生成工具的设计

3.8.1 生成的词法分析器的结构

Lex程序组件的内容：p105/122

3.8.2 基于NFA的模式匹配

略，解释型例子

·最长匹配+最先匹配（根据分支语句代码或声明的正则表达式的顺序） = 最终确定匹配

3.8.3 词法分析器使用的DFA

略

·仍然遵从最长+最先匹配的规律。

3.8.4 实现向前看运算符

向前看运算符状态s的定义和性质：p108/125

DFA中的死状态讨论：p108/125

死状态：在所有输入符号串上都转向自己的非接受状态。

## 3.9 基于DFA的模式匹配器的优化

本节介绍的三个算法：

·正则表达式直接构造DFA

·优化合并DFA状态

·生成更加紧凑的转换表

3.9.1 NFA的重要状态important state

重要状态：如果一个NFA状态有一个标号非的离开转换，则为重要状态。

·重要状态和正则表达式的运算分量（非运算符|，\*等的符号）一一对应。

·抽象语法树中的非的叶节点和重要状态一一对应。

两个NFA状态集合的一致性判断：p109/126

结束标记符#：为了将接受状态作为一个重要状态，后加一个#。这样的正则表达式也称增广（或扩展augment）正则表达式

3.9.2 根据抽象语法树计算得到的函数

为了直接构造DFA，需要先构造正则表达式的抽象语法树，以及4个函数：p111/128

·nullable(n)：判断节点n是否可能为空串

·firstpos(n)：节点n下的子树中，可接受的串的允许起始位置，这是一个集合

·lastpos(n)：节点n下的子树中，可接受的串的允许终止位置，也是集合

·followpos(p)：允许跟随在位置p后面的符号位置，是一个集合

注意：n代表抽象语法树的所有可能节点，p代表语法树中的位置，亦即重要状态或非运算符符号。

3.9.3 计算nullable，firstpos及lastpos

基本规则和归纳规则情况下的计算：p111/128

3.9.4 计算followpos

计算followpos表的两个规则：p112/129

followpos函数表可以转换为有向图（位置作为状态，follow关系表示为有向边），该有向图经过添加边的标号、标记开始状态和接受状态，就会变成不含转换的NFA。

3.9.5 根据正则表达式构建DFA

算法流程：p113/130

3.9.6 最小化一个DFA的状态数

任何正则语言都有一个唯一的状态数目最少的DFA

可区分的distinguishable：状态的可区分性，两个不同的状态，存在某相同的串路径使二者到达不同的状态，且只有一个是接受状态，则二者可区分。反之，所有的串从两个状态起始到最后都是均可接受或不接受，则二者不可区分。

最小化状态数算法及其原理：p115/132

·空串可以区分任意的非接受状态和接受状态，因此这也是算法初始化的划分由来

3.9.7 词法分析器的状态最小化

DFA状态最小化算法的初始分划的扩展标准：p116/133

·若不使用扩展标准，有时DFA的接受状态集合无法细分

3.9.8 DFA模拟中的时间和空间权衡

压缩DFA转换表的方法：

·使用转换链表来代替二维数组（类似用邻接链表代替邻接矩阵）

·一个巧妙的数据结构p117/134

利用状态之间的转换相似性，default数组用以定义状态的默认参考状态，亦即若没有特别定义该状态的某转换，则该转换和默认参考状态的一致。

# 第4章 语法分析

语法描述方法：

上下文无关

BNF（巴库斯-瑙尔范式）

以下是一个概念类比：

语法分析树=DFA或NFA的一个路径

文法或上下文无关文法=正则表达式

## 4.1 引论

4.1.1 语法分析器的作用

作用：接受词法单元流，生成语法分析树（可以不显式生成），验证该流可以由源语言的文法生成（符合某一种句式）。实际上语法分析器和其他的一些检查与翻译过程可以同时进行并且整合为一个模块。

语法分析器三种类型：

·通用：Cocke-Younger-Kasami算法和Earley算法，可以对任意文法进行语法分析，但效率低，不能用于编译器产品

·自顶向下

·自底向上

高效的语法分析器只能处理某些文法子类，其中的一部分子类（LL，LR）足以囊括现代程序设计语言的大部分构造（其余的交由后续模块处理检查）。手工实现的语法分析器通常使用LL，LR一般由自动化工具构造，因为较大。

4.1.2 代表性的文法

略

4.1.3 语法错误的处理

错误种类：

·词法错误：拼写，符号标注（如引号）错误等

·语法错误：语法结构的书写错误，如分号、大括号和分支循环语句的书写

·语义错误：运算分量或函数返回类型等的不匹配问题

·逻辑错误：其他任何一般性的编程错误

错误处理程序的要求与目标：p123/140

4.1.4 错误恢复策略

①恐慌模式的恢复

遇错则跳，最简单的“恢复”

同步词法单元synchronizing token：

②短语层次的恢复

遇到错误后对余下输入进行就地局部修正，修正方法是自定义的

③错误产生式

根据常见错误构造错误文法以及分析器，反向工作

④全局纠正

开销太大，仅限于理论分析

·最接近正确不一定就是符合需求的程序，评估标准的选择也是必要的

## 4.2 上下文无关文法

4.2.1 上下文无关文法的正式定义

上下文无关文法，简称文法，包括：

·终结符号：即词法单元

·非终结符号：由终结符号组成的集合

·开始符号：文法描述种首个列出的非终结符号，以此来逐层定义文法

·产生式：描述文法规则的式子，包括头（非终结符号），箭头或::=，产生式体

4.2.2 符号表示的约定

常用符号约定：p125/142

4.2.3 推导

定义：简而言之是将文法按照产生式逐渐替换至一个特殊的情况，推导出来的式子或串显然是符合文法的。推导的一般定义见p126/143

句型sentential form：由开始符号推导出的一个串，可能包含终结或非终结符号

句子sentence：不含非终结符号的句型。所有句子的集合即该文法生成的语言。

上下文无关语言context-free language：可以由文法生成的语言，换言之，不能用文法生成的语言必须和上下文有关来进行解释。

最左推导leftmost derivation：总是最先替换最左边的非终结符号进行推导

最右推导rightmost derivation：同理，也称规范推导canonical derivation

最左句型left-sentential form：按照最左推导获得的句型

4.2.4 语法分析树和推导

语法分析树即是推导的图形表示形式，但是无法完全准确表达替换的顺序，有时省略非终结符号代表的节点作为另一种形式。

语法分析树的叶节点从左至右排列就可以得到一个句型，也称结果yield或边缘frontier

用推导过程构建语法分析树的一般归纳步骤：p128/145

4.2.5 二义性ambiguous

定义：一个文法可以为某个句子生成多棵语法分析树或多个推导方式

二义性可以利用以带来方便（此时必须要有消二义性规则disambiguating rule），但是本质上来说必须确定唯一的语法分析树才能进行之后的工作。

例子中就是利用二义性的方便结合运算符优先级作为消二义性规则，否则必须为加减和乘除运算定义不同的标识符来代表运算项。

4.2.6 验证文法生成的语言

略，证明文法所产生语言的特性

4.2.7 上下文无关文法和正则表达式

根据NFA（正则表达式）构造上下文无关文法的方法：p130/147

有穷自动机或正则表达式不能计数（这也许和其定义有关），因此也不能支持形如anbn形式的结构（可以让a与b自由重复，但是无法保证二者相等且同时可自由重复），也没有乘幂运算符。

## 4.3 设计文法

计算机中的语法称为syntax，文法称为grammar。语言学领域里的语法不做讨论。

文法和语法的区别：文法（一般特指上下文无关文法）是一种按照特定规则来描述的行文规律，而语法更加宽泛，指代语言相关的一切规则。显然完全意义上的语法包括文法，而文法是其中最富有规律性和可抽象性的部分。极端情况下，一个“毫无规律”的语言注定还是有语法的，但是可能无法提炼出文法。

文法可以改写成等价的其他形式，以便于语法分析器的工作与编写。文法更加像是一种严谨的数学描述，而且针对的是独立的句子翻译，而计算机中的语法则类似语言学中的概念，也要考虑上下文的关系（声明，定义域）等等。

4.3.1 词法分析和语法分析（再次讨论二者的区分）

为何使用正则表达式定义词法？

·正则表达式简单高效，面对词法单元足够用，比如标识符，名称，关键字

·语法分析，尤其是嵌套语句结构等，也只能使用更加复杂的文法描述

4.3.2 消除二义性

消除二义性和消二义性规则举例：p133/150

·消除二义性的方法通俗的来说就是将产生不同语法分析树分支的转换分别用不同的非终结符号重新定义。当然消除二义性的文法方法和消二义性规则虽然是两种方法，但是文法的设计选择本身就带有消二义性规则。

4.3.3 左递归的消除left recursive

立即左递归（递归只经过一步推导，如A→Aα）的一般消除规则：p134/151

原理：这里的递归和其他问题中的递归不一样，实则各具独特性。这里的左递归就是在句子的左边不断增加相同的内容，直到某一符号为止。可以反向理解为在该某一符号之后，句子向右开始扩展增加相同的内容，直到另一个终结符号——空串。

一般消除左递归的算法（没有环和产生式）：p135/152

环：能够推导出自身

产生式：生成空串的产生式

·以上两者都可以从文法中系统地消除

4.3.4 提取左公因子

定义：一种文法转换方法，实际上就是一种改写方式，便于某些语法分析实现技术

原理：p136/153

·和乘法项的提取公因子类似

4.3.5 非上下文无关语言的构造

比如变量的先声明后使用，比如函数的形参和实参的对应，都不是文法可以描述的问题，亦即也不在语法分析阶段解决，而是在语义分析阶段。

## 4.4 自顶向下的语法分析

自顶向下语法分析：从语法分析树的根节点开始，按照先根次序或称深度优先次序创建语法分析树的各个节点。这实际上对应了最左推导的过程。

递归下降语法分析：自顶向下语法分析的通用形式，可能需要回溯来确定产生式的选择

预测分析技术：前者的一个特例，不需要回溯。通过向前读取一个符号来判断和预测产生式的选择，当然这种技术对于文法本身的特点也有要求。

4.4.1 递归下降的语法分析

一个非终结符号对应的过程算法：p139/156

·对于多个产生式的情况，需要进行多一层的循环

※自顶向下语法分析不能应用于左递归的文法是因为每次使用上述算法的一个过程，都必然因为首先遇到一个非终结符号而继续递归展开，从而产生无限循环。

4.4.2 FIRST和FOLOW

两个常用函数的定义：

FIRST(α)：可从α推导得到的串的首符号（必须是终结符号）的集合，可能包括空集。FIRST函数可用于预测或选择产生式用于推导，前提是该非终结符号的产生式已经经过提取左公因子

结束标记：$虽然不是任何文法的符号，但是在文法表达中表示句子或串的结束

FOLLOW(A)：合法情况下可能出现在紧跟A右边的终结符号的集合

两个函数的计算方法：p140/157

4.4.3 LL(1)文法

名称：第一个L——表示从左向右扫描输入，第二个L——产生最左推导，1或任意值表示每步中只需要向前看一个输入符号来预测

预测分析器的状态转换图（不同于词法分析）：p141/158

LL(1)文法的定义和准则：p142/159

构造预测/语法分析表的算法：p142/159

·之所以可以使用预测分析，尤其是向前一位，因为已经提取左公因子

LL(1)文法已经足以描述大部分程序设计语言构造，但是不支持左递归和二义性。

4.4.4 非递归的预测分析

分析表驱动的基于手动堆栈的语法分析算法实现：p144/161

4.4.5 预测分析中的错误恢复

①恐慌模式

一些启发式规则：p145/162

②短语层次的恢复

## 4.5 自底向上的语法分析

含义：从叶节点开始逆推导直到归于一个单一的根节点（开始符号）

4.5.1 归约reduction

归约是指自底向上逆推导的过程，它的正推导实际上是一个最右推导

4.5.2 句柄剪枝

句柄：注意这里的含义和其他计算机领域内的不同，是和某个产生式体匹配的子串。严格来说，这不是一个专有或特别的定义，只是一个提法。另外，句柄的完全定义也包括该对应的产生式

4.5.3 移入-归约语法分析技术shift-reduce

原理：将输入字符不断移入语法符号栈，然后对符合某个产生式体的子串进行替换，直到输入字符为空，而栈中只有开始符号

句柄总是出现在栈的顶端，而非中间

4.5.4 移入-归约语法分析中的冲突

存在两种冲突：

移入/归约冲突：不知道是应该继续移入还是归约

归约/归约冲突：不知道使用哪一种归约

一般不能使用移入-归约语法分析，或不符合LR(k)文法的文法特点有：

二义性——虽然不是所有的情况都无法处理，但是能处理的情况必增加了消二义性规则

·移入-归约分析有时不仅仅需要向前看若干个字符，也需要使用栈中而不仅仅是栈顶的信息来判断归约方式。因为有的文法中两个非终结符号的产生式体一致，该选择哪个用来归约必须查看其它部分的信息，或者是符号表等以区分。

## 4.6 LR语法分析技术介绍：简单LR技术

简单LR也称SLR，L表示从左至右扫描输入符号，R表示反向构造一个最右推导序列，道理类似LL

·LL文法类是LR文法类的一个真子集，虽然LR文法需要消除左递归才能应用LL语法分析

4.6.1 为什么使用LR语法分析器

LR分析器比LL分析器更加高级，优点更多，只是工作量太大需要自动构造

其他优势：p153/170

4.6.2 项和LR(0)自动机

项：项是针对文法而言的，概念上它表示一种状态，确切的说是文法的产生式再加上一个位于体中的一个点p154/171

规范LR(0)项集族canonical LR(0) collection：可以用于构建一个确定有穷自动机，其构建方法和定义见p157/174

项集的闭包：定义见p155/172

项的分类：

内核项：包括初始项以及点不在最左端的所有项

非内核项：除了初始项之外的点在最左端的所有项，非内核可以由内核项闭包运算得出

GOTO函数：这里不是特指某个编程语言，GOTO函数用于定义某项集状态根据下一步输入转换得到的新状态/项集

LR(0)自动机的用法：见例子p157/174

4.6.3 LR语法分析算法

LR语法分析表的结构：

ACTION：4种动作，移入，归约，接受，报错

GOTO：实际上就是个转换表，ACTION的构造也要基于GOTO

LR语法分析器的格局configuration：

包括栈中的内容（状态或文法符号），输入缓冲内容

LR语法分析器的行为：算法p159/176

4.6.4 构造SLR语法分析表

不同的LR分析技术的区别只在于语法分析表的构造不同，其行为模式都是一致的。

SLR语法分析表的构造算法：p161/178

·准确的说，SLR是SLR(1)的简写，SLR(1)作为一个术语可以形容分析表的构造方法（SLR分析技术）、使用该分析表的SLR分析器，以及能够构造该分析表的SLR文法。

※每种语法分析技术都存在某些无二义性文法使其产生语法分析冲突，不过程序设计语言领域内可以避免使用这样的文法。

4.6.5 可行前缀viable prefix

定义：一个最右句型的前缀，并且没有越过该句型的最右句柄的右端。

有效项：定义见p163/180。大意是可以使当前前缀（栈中内容）按照其进行正确与合理地操作判定的项。这个定义的对立面具有参考意义——无效的项虽然一定程度看似对应当前前缀，但是按照它进行操作是错误的或引致冲突。

## 4.7 更强大的LR语法分析器

规范LR方法——基于LR(1)项集，分析表较大

LALR方法——也称向前看LR，基于LR(0)项集族，分析表不大

4.7.1 规范LR(1)项

比LR(0)项多一个元素，即产生式之后的预测输入符号。

LR(1)项对一个可行前缀有效的条件：p166/183

※SLR(1)语法分析方法基于LR(0)项集，规范LR(1)方法基于LR(1)项集。这说明括号中的数字对于不同的概念并非是统一的，项集LR(1)的1是指第二个元素（终结符号串）的长度。而SLR(1)语法分析方法的1表示向前看一个输入字符。虽然两种情况下括号内数字都有“向前看”的意义，但是项集中的向前是指产生式或句柄之后，而分析方法中的向前是指当前输入流的位置之后。

4.7.2 构造LR(1)项集

构造算法：p167/184

·这里的GOTO函数返回的是一个项集

4.7.3 规范LR(1)语法分析表

构造规范LR(1)语法分析表的算法：p169/186

·这里的GOTO函数返回的是一个状态

·每个SLR(1)文法都是LR(1)文法，但是后者的状态更多。

4.7.4 构造LALR语法分析表

·LALR语法分析方法是实践中最实用和常用的

项集的核心core：是针对LR(1)项集而言的，即它的第一个分量或元素，具有相同的核心的项集实际上在LR(0)项集中同属一个项集。

·同核心的项集进行合并后不会产生原来没有的移入/归约冲突，因为移入动作仅由核心决定。但是可能会产生归约/归约冲突。

不常用的构造LALR分析表的方法：p171/188（因为先构造LR(1)项集，再合并，这样的方法本质上就很麻烦）

·同理，LALR(1)也是形容分析表、分析技术、文法（如果没有冲突）、项集族等术语的通用术语。

·LALR(1)分析器和对应的LR(1)分析器在处理正确的输入时行为一致，但是处理错误的输入时LALR分析器可能需要更多的步骤（只有归约）和时间来报错，即它也绝对不会比LR分析器移入更多的符号来发现错误。

4.7.5 高效构造LALR语法分析表的方法

向前看符号的传播和自发生成检测（或确定向前看符号）的算法：p174/191

LALR(1)项集族的内核的高效计算算法：p174/191（先构造LR(0)项集族的内核，再计算向前看符号）

有了内核之后，求闭包即可得LALR(1)项集，将这些项集当做LR(1)项集族求语法分析表即可。

4.7.6 LR语法分析表的压缩

压缩语法分析表的一些策略：p176/193

## 4.8 使用二义性文法

有时结合使用二义性文法和消二义性规则可以起到事半功倍的效果，但是本质上文法根本不可能解析二义性的内容

4.8.1 用优先级和结合性解决冲突

利用运算符的优先级和结合性（左结合或右结合）可以在表达式型文法中适当利用二义性

4.8.2 “悬空-else”的二义性

条件语句的匹配是使用二义性简化文法的另一个例子

4.8.3 LR语法分析中的错误恢复p181

错误恢复动作可以像分析动作（移入/规约）一样加入分析表中

## 4.9 语法分析器生成工具

4.9.1 语法分析器生成工具Yacc

Yacc的.y文件的编写范式：p183/200

·Yacc语法分析器里包含了语义定义，因为语法分析在实践中并非完全独立的模块

·辅助的C语言例程必须包含一个名为yylex()的词法分析器

4.9.2 使用带有二义性文法的Yacc规约

描述如何使用Yacc定义和使用二义性，以及解决冲突

4.9.3 用Lex创建Yacc的词法分析器

具体的一些编译和连接方法

4.9.4 Yacc中的错误恢复

基于错误产生式的错误恢复机制

# 第5章 语法制导的翻译

·编译原理中的“翻译”有很多层的含义，虽然道理上和字面意思区别不大，但语境不同便目标不同。翻译的源语言一定是编程语言，但是目标语言可以存在（如另一种编程语言、中间表示形式或中间代码），也可以不存在（仅仅根据语义动作执行一个操作）。

## 5.1 语法制导定义

语法制导定义syntax-directed definition，SDD：一个上下文无关文法和属性及语义规则semantic rules的结合。说白了语法只是语言的规矩，语义则是语言的含义，语法制导即以语法（文法或产生式）为基础来进行语义的描述。

注意这里的“定义”不是“语法制导”的定义，“语法制导定义”是一个技术的名称术语。SDD是语法制导类型的翻译方法中的一个（虽然没有明确给出实现代码的方式），后文中的SDT是另一个，其中的“翻译方案”也是名称术语的一部分，不是通用的名词。

5.1.1 继承属性和综合属性

语义分析中赋予非终结符号两种属性——

综合属性synthesized attribute：由该非终结符号自身的产生式所关联的语义规则定义的属性。因此该属性只由该非终结符号自身和其子节点表示。

继承属性inherited attribute：由该非终结符号的父节点的产生式所关联的语义定义的属性。因此只包含父节点、自身和兄弟节点。

·不允许某节点的继承属性通过其子节点上的属性定义，但是允许其综合属性通过该节点的继承属性来定义。

·终结符号只能有综合属性，不能有继承属性，实际上其属性值是词法分析器直接提供的。

S属性SDD（S-attribute SDD）：一个只包含综合属性的SDD

属性文法attribute grammar：没有副作用的SDD的别称，因为其规则仅仅只是通过其他属性定义一个新的属性（这里谈论的是语义，不知为何使用文法这个术语）

5.1.2 在语法分析树的结点上对SDD求值

注释语法分析树annotated parse tree：显示了属性值的语法分析树

·继承属性可以说是一种将语义定义脱离语法定义而活用的存在，S属性SDD仅能提供完全仿照产生式结构的赋值语义。而且继承属性顾名思义可以用来保存和传递数据。

继承属性的使用例子（仅供初始理解之用）：p197/214

## 5.2 SDD的求值顺序

5.2.1 依赖图dependency graph

定义：描述语法分析树中属性之间的信息流，或者求值顺序。

详细定义：（太抽象）p198/215，可参考例子

5.2.2 属性求值的顺序

·必须是无环的依赖图才能够进行SDD求值，S属性SDD和L属性SDD是两种不可能生成有环依赖图的SDD，然而其他种类的SDD不见得满足这个要求。

获取拓扑排序的一般算法：p200/217（不是构造依赖图的算法）

5.2.3 S属性的定义

定义略

·S属性SDD可以按照语法分析树的任何自底向上顺序来计算属性值，最常用和简单的就是后序遍历（每个节点都在其子节点全部遍历后再执行）。也因此它适合和自底向上语法分析工作结合，一边分析语法一边求属性值或执行其他动作。

5.2.4 L属性的定义L-attributed definition

详细定义：p200/217

·简单说来，L表示left左边，L属性SDD允许包含继承属性，但是该属性的定义必须依赖其对应的非终结符号左侧的其他非终结符号的属性。左侧是指产生式体或分析树的结构中位于左侧。当然该属性的定义也可以依赖产生式头的属性（也算左侧）以及自身的其他综合或继承属性（至少不算右边）。

·类似于S属性SDD可以按照任何自底向上顺序计算属性，LL文法（典型的自顶向下语法分析文法）上的每个L属性SDD都可以在自顶向下语法分析中实现，但是对于LR文法上的L属性SDD无法在语法分析过程中实现（必须先分析之后再翻译）。但这不意味着L属性SDD必须按照自顶向下顺序计算（见后文实现L属性SDD的部分）。也就是说语法制导翻译虽然常用来和语法分析结合进行，但二者是两种独立的技术，没有任何天生必然的绑定关系。只是为了实现方便，翻译技术的发展路线侧重于将二者结合。

5.2.5 具有受控副作用的语义规则

副作用含义：说白了就是非属性求值性的语义动作或程序逻辑，如打印、添加至表格等等

副作用的添加要有一定限制和控制：p201/218

·副作用语义也是绑定在产生式或文法上的，它可以视为一种哑属性的定义，因为它一般不会是为该属性赋值的语义，这种“假想”的数据流仍然可以体现在依赖图中。

·实际上，带有副作用的SDD并非SDT，算作一种属性文法和SDT之间的平衡，这也说明SDD和SDT是两种独立的方法。另外，SDT不一定使用属性来存储数据并完成和SDD相同的翻译过程，当然SDT也没有规定不允许使用属性，只要最终输出结果正确即可。

## 5.3 语法制导翻译的应用

本节中使用SDD构造抽象语法树，因为有时抽象语法树是部分编译器的中间表示形式，但实际上可以直接使用一些SDD生成一种中间代码而非显式构造该树

·注意，抽象语法树是个实际的程序数据结构，不是语法分析树

5.3.1 抽象语法树的构造

S属性SDD和L属性SDD两种构造方法举例：p203/220

5.3.2 类型的结构

一个声明数组或变量类型的语义示例p206/223

## 5.4 语法制导的翻译方案

语法制导的翻译方案syntax-directed translation scheme：SDT和SDD是独立的两种概念或语法制导翻译的具体方法，SDD针对每个产生式对应定义若干个属性与语义规则，而SDT是将语义动作semantic action（程序片段）嵌入产生式体中。二者看起来都是针对产生式的某种语义赋予过程，所以都是语法制导类型的翻译方法。

·不同于S属性SDD必然可以实现于后序遍历语法分析树（或和移入-归约语法分析一起实现），SDT不一定可以与语法分析过程一起实现，然而任何SDT必然可以实现于前序遍历语法分析树（个人感觉任何深度优先均可，因为代码段子节点必然没有更多子节点）。虽然语法分析过程也是遍历分析树，但是那只是结果的表象，分析过程的代码是根据语法判断进行的，最终形成了某种遍历，而不是强制按照某个顺序进行遍历。

·语义翻译的过程实际上是和语法分析独立的步骤（举例而言，中缀表达式书写的文法可以被后缀SDT翻译成后缀表达式，即翻译不见得和语法一定有关。注意两个后缀的含义是独立的）。

判别SDT方案是否可以在语法分析过程中实现的方法：p207/224（将程序片段作为只能产生空串的非终结符号即可）

5.4.1 后缀翻译方案

定义：所有动作（程序片段）都在产生式最右端的SDT，实际上就是S属性的SDD

·后缀SDT和S属性SDD一样可以和自底向上语法分析同步执行。

5.4.2 后缀SDT的语法分析栈实现

即在真实的程序中，属性值计算和执行动作都是在原有的语法分析栈上进行的。

举例：p208/225

5.4.3 产生式内部带有语义动作的SDT

产生式体内部SDT的执行规则：p209/226

不能实现于任何语法分析过程的SDT举例：p209/226

实现任何SDT均可使用的方法：p210/227（将语义动作视为语法分析树结点的前序遍历）

·说白了SDT能够和语法分析过程一起实现是不一定的。

5.4.4 从SDT中消除左递归

两种情况：

·若SDT的动作全部都是非属性求值型命令，转化时将动作当做终结符号处理

·若SDT的动作包括属性求值，见p211/228（条件是所有属性转化前均为综合属性）

5.4.5 L属性定义的SDT

L属性SDD转化为SDT的规则：p212/229

两个转化的例子：p212/229

## 5.5 实现L属性的SDD

几乎每个之前介绍的语法分析技术都可以和L属性SDD结合。

5.5.1 在递归下降语法分析

方法&举例：p217/234

5.5.2 边扫描边生成代码

边扫描边生成代码的实现要素：p219/236

主属性：主属性是该算法实现的必备要素，所谓的边生成的内容也必须是主属性。

L属性SDD转化为边扫描边生成的SDT的又一例：p220/237

5.5.3 L属性的SDD和LL语法分析

方法描述：p220/237

动作记录action-record：即将被执行的语义动作

综合记录synthesize-record：保存非终结符号的综合属性值

例子1：边扫描边生成代码方式

例子2：存储代码至综合属性的方式

5.5.4 L属性的SDD的自底向上语法分析

·LL文法虽然可以使用自顶向下语法分析，但是不代表不能使用自底向上方法。当然需要经过一定变换，毕竟LL文法是LR文法的真子集。

将LL文法的L属性SDD改写为可以在LR语法分析过程中实现的方法：p224/241

举例：p225/242

# 第6章 中间代码生成

编译器前端的逻辑结构：p229/246

·中间表示形式也有高低层之分，一般语法树形式的中间表示属于高层，高层接近于源程序逻辑而非目标机器逻辑；而三地址代码形式的中间表示根据所选运算符不同，可以是高层形式的也可以是低层的

中间代码或中间形式非常依赖编译器的选择与设计，中间代码甚至可以是一门编程语言，如C++编译可以先翻译为C。

·中间表示形式的举例：三地址表示方式（四元式、三元式、间接三元式），虚拟机表示方式（字节代码和堆栈机代码），线性表示方式（后缀表示），语法树和DAG图形表示

## 6.1 语法树的变体

6.1.1 表达式的有向无环图directed acyclic graph，DAG

语法树对于重复的内容必须重复构造节点，而DAG可以简化这个问题

生成DAG的语法制导定义（只是在生成语法树节点的基础上先检查是否已存在该节点）：p230/247

6.1.2 构造DAG的值编码value number方法

值编码定义：即使用一种特殊的数据结构存储DAG，p231/248。也可以狭义的指某个节点的代表。

构造DAG的节点的值编码方法：p231/248（用散列表保存）

## 6.2 三地址代码

三地址代码的一条指令最多只有一个运算符。每个指令对应于语法树或DAG的一个内部节点。

6.2.1 地址和指令

常见地址形式：p233/250（这里的地址可以指一个字母变量，或对应符号表的指针）

常见指令形式：p234/251

6.2.2 四元式表示quadruple

定义：使用四个字段（最多四个）表示一个三地址指令的方式，p235/252

6.2.3 三元式表示triple

含义：只用三个字段表示三地址指令的方式，实际上用指令的位置或编码来代指运算的中间结果，从而节省一个字段

三元式和表达式的DAG表示是等价的，但是三地址和语法树不等价

·注意一条三地址指令不一定只对应一条三元式条目，可能需要更多

·常用的翻译算法很多时候会复制赋值指令的运算，即出现a=x1样式的退化型三地址指令。这是因为算法的作用机理导致的必须检测到=才能确定赋值运算的参数，否则一律使用中间临时变量保存结果。上一步的中间临时变量在下一步检测到=时已经生成完毕，这只能靠复杂的算法或优化手段改进。

·优化编译器中，普通三元式如果遇到指令的迁移，那么其中间运算结果的引用全部需要修改，很麻烦。

间接三元式indirect triple：额外包含一个指向三元式的指针的列表，用于移动三元式的序列，从而不会导致三元式内的指令引用错乱失效。

6.2.4 静态单赋值形式SSA

含义：可以说SSA是一种特殊的类似三地址代码形式的中间表示形式，其要求是每个指令的赋值对象都必须为不同的变量。SSA有利于实现某些类型的代码优化

SSA中的φ函数p237/254

## 6.3 类型和声明

类型在编程中的应用分为两类：

类型检查type checking——保证运算符和分量的类型匹配

翻译——确定变量运行时刻占用的空间、指针、函数调用等

6.3.1 类型表达式type expression

含义：一种解释类型声明语句的表达式标准，或可视为一种针对类型声明指令的中间表示形式

详细定义：p238/255

·递归类型：即类型的声明体中包含对该类型本身的使用

6.3.2 类型等价

结构等价structurally equivalent：判定方法p239/256

名等价name equivalent：名等价是结构等价的一种特殊形式，即两个类型完全相同且被赋予相同的值编码。

6.3.3 声明

一个简化的类型声明文法：p239/256

6.3.4 局部变量名的存储布局

计算类型存储布局参数的SDT p240/257

6.3.5 声明的序列

按照一个序列进行声明的翻译SDT p241/258

6.3.6 记录和类中的字段

记录类型或类中的数据声明的翻译SDT p242/259

## 6.4 表达式的翻译

6.4.1 表达式中的运算

生成表达式的三地址代码的SDD（只包含+，和-负号运算）：p243/260

6.4.2 增量翻译

生成表达式的三地址代码的SDT（采用边翻译边输出而非存储代码的方式）：p244/261

·上面的方案只改写语义动作即可构造语法树

6.4.3 数组元素的寻址

数组索引相关的计算公式：p245/262

C语言使用按行存放方式存储多维数组，而Fortran使用按列方式。前者按照地址从前向后遍历时数组最右边的维度变化最频繁，后者正好相反，最左边的下标维度变化最频繁

·为了达到中间代码独立于目标机器的效果，应该使用可定义变量来表示（基本）类型的长度，由目标代码生成器来根据目标机器决定其具体数值。

6.4.4 数组引用的翻译

带有处理数组引用的生成表达式的三地址代码的SDT p246/263

举例：p247/264

## 6.5 类型检查type checking

类型系统type system：一组逻辑规则，用于检测语言翻译后的类型表达式的合法性

强类型：编译器可以保证其程序在运行时刻不会发生类型错误，即可以在编译时完成类型检查。其他弱类型的语言在程序真正执行前进行类型检查，但不是编译器完成的，具体规则取决于其类型系统的设计。

6.5.1 类型检查规则

类型检查的两种形式：

（1）类型综合type synthesis

根据子表达式的类型构造出本表达式的类型，要求所有元素名字先声明再使用。同时也包括根据上下文的某些信息来确定类型的情况，如重载函数之类型的确定。

类型综合规则的形式(6.8)：p248/265

（2）类型推导type inference

根据一个语言结构的使用方式来确定结构的类型，不需要对名字进行声明。注意类型推导不一定是弱类型语言，如ML

类型推导规则的形式(6.9)：p249/266

6.5.2 类型转换

拓宽widening和窄化narrowing：两种类型转换规则，前者是将原类型替换为空间更宽的数据类型，后者相反。

隐式转换：也称自动类型转换coercion，编译器自动完成的类型转换，一般仅限于拓宽

显式转换：也称强制类型转换cast，由程序员代码明示。

举例：表达式求值中引入类型转换的SDT p250/267

6.5.3 函数和运算符的重载overload

针对重载函数的类型综合规则：p251/268

·C和Java中可以通过仅查看函数参数类型来解决重载区分，但是有些编程语言仍需更多信息解决重载

6.5.4 类型推导和多态函数

参数多态parametric polymorphism：通过参数和类型变量刻画的多态性

Ɐ：全称量词wiversal quantifier，读作“对于任意类型”，作用于受限的bound类型变量。其在函数类型表达式中的使用如(6.12) p251/268

受限变量：可以被任意重命名，但是必须将所有同名变量一起更改

多态类型：带有Ɐ符号的类型表达式

合一unify：判断能否通过将两个类型表达式中的变量替换为某些表达式，使得两个类型表达式相同。测试表达式是否等价是合一的一个特殊情况。

置换substitution：书中定义是从类型变量到类型表达式的映射。类型表达式包括类型变量，也包括类型常量，即整型、浮点型等。所以正确的理解是置换是将一个类型表达式中的一个或多个类型变量替换为另一组类型变量或常量的操作。注意置换操作对同名类型变量必须统一。

实例instance：被置换处理后的结果即为原类型表达式的一个实例，尤其是置换为类型常量的情况

最一般化的合一替换most general unifier：p252/269，粗略的理解为是所有合一替换中被置换为常量的变量个数最少的。

多态函数的类型推导算法：p252/269

举例：p253/270

6.5.5 一个合一算法

·合一算法是检验两个类型表达式是否等价或可合一的算法

算法：p255/272

等价类：即等价的类型表达式（一个类型变量和一个常量可以等价也可以不等价，不等价的条件是该类型变量已经和另一个类型常量等价），包括等价的内部节点（类型构造算子及其子节点）

## 6.6 控制流

布尔表达式有两种用法：改变控制流（条件语句等）和计算逻辑值（作为一个数值类型进行算术计算）。不同的用法在编译中的翻译方式不同。

6.6.1 布尔表达式

定义：由作用于布尔变量（取值为false或true的变量）或关系表达式（由比较运算符和算术表达式组成的，如a>b等）的布尔运算符（表示AND、OR和NOT等的运算符）构成的

·NOT运算优先级高于AND，AND高于OR

6.6.2 短路代码

定义：一种翻译布尔表达式控制流的方式，不显式计算出运算符结果，而直接根据逻辑推导将控制流跳转到对应语句位置。P257/274

6.6.3 控制流语句

控制流语句的语法制导定义：p258/275

·本定义中可能会产生大量标号，可以使用回填技术或优化技术来减轻此问题

6.6.4 布尔表达式的控制流翻译

布尔表达式生成三地址代码的SDD：p260/277（此SDD不是最优化的）

6.6.5 避免生成冗余的goto指令

对上一节SDD的语义规则进行改进：p261/278

6.6.6 布尔值和跳转代码

对布尔表达式的两种使用方法的“简单”翻译策略：p262/279（不是详细算法）

①两趟处理法

②语句一趟，表达式两趟处理

·因为存在不同的翻译方法，所以对相同的产生式必须应用不同的语义动作（函数代码）。虽然求值过程类似，但是控制流语句中要增加更多的跳转指令。

·布尔表达式的两趟处理需求在于第一趟需要用于构建语法树，第二趟用于翻译。这是使SDT翻译策略必定可行的方法，并不是布尔表达式有两种翻译所导致的必须需求。具体讨论书中本节不详。

## 6.7 回填backpatching

原理：先生成跳转指令但是其目的地（语句标号）由后面的代码回填，因为初次生成跳转指令时无法确定其目的地，除非两次扫描。

6.7.1 使用回填技术的一趟式目标代码生成

原理详述以及承担主要功能的函数

6.7.2 布尔表达式的回填

布尔表达式的翻译方案SDT（自底向上语法分析过程）：p264/281

·看例子学习这个算法的含义更明白些

6.7.3 控制转移语句

以回填为基础的控制转移语句的SDT：p267/284

·回填过程并不被所有跳转语句需要，实际上只对向后不确定的跳转有意义，向前跳转由于已经扫描并确定了语句标号，可以直接生成完整跳转语句。

6.7.4 break语句、continue语句和goto语句

另外两种跳转语句的实现策略，和goto大同小异，区别是break和continue很可能需要当做符号表标识符（并非中间代码的一个形式）来进行处理。

## 6.8 switch语句

6.8.1 switch语句的翻译

描述性的switch语句翻译策略（几种）：对照关系表/条件跳转指令序列、散列表、桶数组

·注意此处的翻译策略并非只针对中间代码，也包含最终代码级别的实现策略。如中间代码可能简单采用条件跳转表示多路分支语句，而最终代码视情况允许采用效率较高的桶数组实现。

6.8.2 switch语句的语法制导翻译

两种翻译方法的直接结果示例：p270/287

·可以使用case x V L形式的三地址代码来代替条件跳转指令if x = V goto L，x是存储被检测的变量，V是数值，L是对应于V数值的分支语句标号。这样区别对待switch语句和条件if语句的目的是为了让最终代码生成器可以有针对性的对多路分支语句进行优化（见6.8.1中的策略）。

## 6.9 过程的中间代码

简单的过程调用三地址代码示例：p271/288

增加了过程定义和调用的文法示例：同上

过程相关的翻译策略简述：类型检测、符号表（每个函数重新用一个符号表作为新作用域，但和原表相关联）、调用——p271/288

# 第7章 运行时刻环境run-time environment

运行时刻环境是一种编译器创建并管理的功能，用于处理多种目标程序在运行时面对的问题，如存储位置分配、过程（函数）绑定（链接）、参数传递和I/O接口。

## 7.1 存储组织

·数据的对齐一般和目标机器的寻址约束有关，每个机器都有一个机器字，32-bit或64-bit。

静态和动态存储分配：编译时刻决定的存储分配为静态，运行时刻的称为动态

·运行时刻内存划分的典型方式：p275/292

栈式存储：用于分配给过程或函数及其局部元素的方式

堆存储：动态数据分配方式，最大的特点是可以回收

·栈存储和堆存储不是独立的，二者往往结合进行工作

## 7.2 空间的栈式分配

7.2.1 活动树activation tree

定义：用于描述函数嵌套调用组织结构的数据结构

·活动树的构成和构造特点决定了可以使用栈来模拟

7.2.2 活动记录activation record

也称为帧frame，是控制栈control stack的每个单元

·根据书写习惯的上下或左右原则，一般会将栈顶写于右侧或下方，虽然这和视觉上的顶端相悖。

活动记录可能包含的数据内容：p279/296

7.2.3 调用代码序列calling sequence

定义：实现过程调用的代码段，注意不是过程的函数体，而是底层实现函数调用和参数传递以及栈构建的目标代码。

返回代码序列return sequence：对应功能的类似事物

调用代码序列和活动记录布局的设计原则：p280/297

·活动栈的栈顶指针有时不一定指向栈顶，尤其是活动记录包含变长的部分，此时往往将变长部分靠近栈顶存放，而将所谓的栈顶指针指向最后的不可变长部分的地址。因为此时所有偏移量均为已知或可预测，因而可以根据该指针进行任意位置的访问。

调用代码序列的任务：p281/298

·调用代码序列实质上是由调用者和被调用者共同分工完成的。

返回代码序列的任务：p281/298

7.2.4 栈中的变长数据

·编译时刻不能决定大小的对象完全可以分配在堆区，存于栈区的优点在于不必垃圾回收，但是该对象只能存活在当前局部作用域。

变长数据存于栈中的策略：p282/299

·使用两个指针来确定数据分布

## 7.3 栈中非局部数据的访问

一阶对象：也称第一类对象，可以作为函数参数或返回值的对象

7.3.1 没有嵌套过程时的数据访问

即不允许在函数或过程内部声明其他的过程，如C语言

（1）全局变量被分配在静态区

·过程的任何非局部变量也是所有过程的非局部变量

·被声明的过程可以作为参数传递，也可以作为结果返回（C中可以使用函数指针），但是不需要对数据访问策略做出改变。

（2）其他变量一定是栈顶活动的局部变量

7.3.2 和嵌套过程相关的问题

略，见访问链

7.3.3 一个支持嵌套过程声明的语言

支持嵌套过程声明的语言：Alogl 60（C语言前身之一），Pascal，ML

ML语言的变量、函数定义示例：p285/302

7.3.4 嵌套深度nesting depth

不内嵌的函数深度为1，其余的依次+1

7.3.5 访问链access link

定义：一个保存在活动记录中的指针，用于记录栈中被执行的过程的嵌套关系。注意嵌套关系是嵌套定义，不是调用关系。访问链的形式是由某过程的活动记录指向其外围且最靠近它的嵌套函数的活动记录。一条访问链路标记了一组嵌套结构中的作用域关系。

7.3.6 处理访问链

两个过程通过显式调用产生关系时，如何确定访问链的方法：p287/304

比较两者嵌套深度的3种关系=<>

7.3.7 过程型参数的访问链

解法：当过程被用作参数的时候，调用者除了传递过程参数的名字，也要传递这个参数对应的访问链。这一切机制可以通过调用代码序列来实现，而不是在高级编程语言中体现。

7.3.8 显示表

原理：为了解决访问链可能随着程序规模增长而过长的问题，使用另一个数据结构来存储活动记录的组织关系。按照嵌套深度不同将同深度的活动记录保存至一个列表来进行查找。实际上显式表和访问链是按照不同的维度进行存储的数据结构而已，显示表适于深度大而独立的不同嵌套关系链不多时，访问表适于深度小而独立的不同嵌套关系链较多时。

显式表的维护方法：每个活动记录需要保存和其同深度的最近前一次活动记录的指针，用于在该活动记录返回撤销后可以恢复显示表至上一个同深度的且最接近栈顶的活动记录指针。换句话说，同深度活动记录按照从栈顶到底的顺序维护了一个列表，列表头的指针在显示表对应深度的项中。

## 7.4 堆管理

7.4.1 存储管理器

存储管理器的两个基本功能：分配与回收（对于自动释放弃用资源的语言，如Java，存储管理器还需要一个子系统——垃圾回收器，来确定需要回收的内容）

存储管理器的期望特性：

空间效率——内存碎片处理

程序效率——存储位置和内存访问的非随机性聚集性

低开销overhead——执行存储管理器自身功能的速度

7.4.2 一台计算机的存储层次结构memory hierarchy

典型的内存层次结构配置举例：p293/310

高速缓存线cache line：主存和高速缓存之间的数据传输单位块，大小为32-256Byte

页page：硬盘和主存之间的数据传输单位块，大小为4-64KB

7.4.3 程序中的局部性locality

时间局部性temporal locality：同一存储位置在短时间内被重复访问

空间局部性spatial locality：临近位置在短时间内被重复访问

·两种局部性在一般的程序中均存在，而对于指令和数据两种内容的访问规律却不尽相同。只能根据需要动态调整高速缓存。

静态与动态RAM简介p294/311

①利用存储层次结构的优化

不如说是利用局部性特点的优化

高速缓存的体系结构存储方式简介p294/311

7.4.4 碎片整理

（1）best-fit和next-fit对象放置

荒野块wilderness chunk：存储块中最靠近可扩展边界的一个，也被视为是“最大容量”的块，该块的扩充不能靠存储器协调而是请求操作系统分配更多空间

next-fit策略：在刚刚分割过的存储块中优先查看是否可用于分配

（2）管理和接合空闲空间

支持相邻空闲块接合的数据结构：

·边界标记——标记使用情况和大小

·双重链接嵌入式空闲列表

相邻空闲块接合和相关数据结构的示例：p296/313

7.4.5 人工回收请求

（1）人工回收带来的问题

①内存泄漏memory-leak：未能删除不能被引用的数据

②悬空指针引用dangling-pointer-dereference：引用已经被删除的数据

③访问非法地址——访问一个数组界限之外的元素

（2）编程规范和工具

①静态生命周期对象的指针所有者ownership

②动态生命周期对象的引用计数reference counting

③基于区域的分配region-based allocation

## 7.5 垃圾回收概述

7.5.1 垃圾回收器的设计目标

增变者mutator：一个用户程序的别称，因为可以修改堆区中的对象集合

（1）基本要求：类型安全

类型安全typesafe：如果任何数据分量的类型都是可确定的，不论是动态运行时确定还是静态编译时确定

（2）性能度量

度量标准：

总体运行时间

空间使用——回收结果的效益

停顿时间——导致程序待机的时间

程序局部性——垃圾回收器可以改变程序内容分布

·没有完美的垃圾回收器，只有适合某种类型程序的情况。另外将数据分配于栈区可以避免产生垃圾，虽然也有其局限性需要考虑。

7.5.2 可达性

根集root set：不需要对指针解引用就可以被直接访问的数据。一般如所有的静态字段成员和栈中的所有局部变量，亦即所有的非堆区数据和对象。

·优化编译器可能会对程序进行某些处理从而在目标代码中改变了源语言的特征，为了能够和垃圾回收器工作，可以考虑的几点设计：p301/318

程序改变可达对象集合的4种操作：

对象分配

参数传递和返回值

引用赋值

过程返回

7.5.3 引用计数垃圾回收器

引用计数的维护机制：p302/319

对象分配

参数传递

引用赋值

过程返回

可达性的传递

缺点：不能回收孤立的循环数据结构，且开销较大

优点：

·及时回收垃圾，较低空间使用量

·以增量方式完成，即可以将本来较大的运算负担（总体运行时间）分摊到多步计算完成（因为基于递归算法），从而可以用来降低停顿时间

延期引用计数：将更新引用计数的操作有计划地拖延从而在某些角度上提高效率的方法。比如，引用计数不包括程序根集的引用，用来降低一些时间开销。但可能在一些阶段需要统一全面扫描计数为0的对象在根集中的引用情况。

## 7.6 基于跟踪的回收的介绍

7.6.1 基本的标记-清扫式mark-and-sweep回收器

此类算法是一种直接的全面停顿的算法。

一个未优化的标记-清扫式垃圾回收算法：p304/321

7.6.2 基本抽象

基本标记-清扫式算法中的状态描述：p306/322

算法中的数据结构和对象状态关系：p306/322（图7-24）

7.6.3 标记-清扫式算法的优化

也称Baker的标记-清扫式算法：p306/322

·多了一个在分配时记录所有已分配对象的列表，这样在回收不可达对象时有的放矢

7.6.4 标记并压缩mark-and-compact的垃圾回收器collector

亦即对内存碎片进行重新定位relocating的回收器，这样做优点很多，而且基于标记跟踪的算法本来就需要遍历所有指针和引用，因此可以同时进行指针修改。另外，由于不必特意回收不可达对象，因此标记阶段也无需Baker优化。

标记并压缩的垃圾回收器算法：p308/325（标记可达对象、计算新地址、更新指针和存储块拷贝）

·此算法是在本地进行重定位，可以降低存储需求，但是不如拷贝回收器高效

7.6.5 拷贝回收器copying collector

原理：预先通过划分（半劈）堆区，将其分为使用区和空闲区，从使用区向空闲区重定位对象不必严格遵循从左至右的顺序，因为必然不存在覆盖的问题。

C. J. Cheney的拷贝回收器算法：p309/326

·此算法还有一个优势，不必为对象添加多余的链接指针（用于存储一些对象至一容器），也无需标记变量。

·重定位算法的缺点在于必须移动所有的对象，而对于大型对象这个代价较高。

7.6.6 开销的比较

p311/328

## 7.7 短停顿垃圾回收

·短停顿垃圾回收算法不必和上一节中的算法冲突，可以结合可以视为一种改进。

7.7.1 增量式垃圾回收incremental collection

含义：按照时间来分割工作任务，使得垃圾回收和增变者程序交错运行

漂浮垃圾floating garbage：每次回收后留下的垃圾

（1）增量回收的准确性

略

·根集处理和空间回收部分都不可被中断。正确性依赖于增量式回收器一定不会遗漏在回收周期开始时就已经不可达的垃圾。

（2）简单的增量式跟踪p312/329

介绍一个简单拙劣的算法策略

7.7.2 增量式可达性分析

一个会导致让可达对象被误认为不可达的情况：p313/330

3种解决该问题的策略：p313/330

（1）写关卡的实现

·这是最优的策略

可以通过记录所有的写入引用的操作或只记录写操作的大致位置来确定需要重新扫描的集合。两种方法都有一定不准确性。

卡片card：将地址空间分成固定大小的块，概念上和页page类似，但是这里只是用于粗略记录区域位置。或者说是一种和页大小不相等的划分区域的统称。

（2）结合增量和拷贝技术

·前述的增量算法对于标记-清扫式回收算法已经足够，但若用于改进拷贝回收，还需要考虑地址重定位的顺序和问题。p314/331

7.7.3 部分回收概述partial collection

·最好的方法是对年轻对象使用世代回收管理，成熟对象使用列车算法独立管理

目标集target：在一轮部分回收中被回收的对象集合，注意不是所有要被释放的对象集合，而是在部分回收算法中，每一步要被执行检测和回收操作的那一部分，其余部分在当前步骤中不被检测和释放。

稳定集stable：相对于目标集的其他对象

根集（新定义）：在部分回收算法中，根集包括寄存器、栈和全局变量，还包括所有指向目标集对象的稳定集对象。这可以理解为每一步将不被检测和回收的稳定集视为根集的一部分。

被记忆集合remembered set：保存了从稳定对象到目标对象的引用的对象集合，可以采用只记录卡片或页的方式粗略记录。

7.7.4 世代垃圾回收generational garbage collection

原理：首先改变堆分配的策略，将堆区划分为多个子区域，一开始只在首个区域分配空间，填满后进行回收并将可达对象移至下个区域，以此类推。注意，每个区域进行回收的情况一定是它之前的所有区域都在同一步中进行了一次回收。

·本节中的阐释有些混淆，世代垃圾回收的原理定义决定了它的“连锁反应式”的工作特性，这种特性在此处被解释得像是某种事先选择。可以将其理解为对世代垃圾回收的原理的可行性解释。

·很明显，世代垃圾回收在某些时候会执行完整的扫描回收，因而并非完全意义上的部分回收或短停顿算法。

7.7.5 列车算法

原理：以车厢和列车的表观方式将堆区以2维分割进行部分回收的算法，对第一辆列车的第一节车厢的定位和世代算法中的第0个区域类似，是回收的首要区域。被回收后将可达对象移走，进而删除该车厢，下一节或下一辆首节车厢替代其位置。

车厢car：固定大小的区域

列车train：堆区组织形态p316/333

（1）被记忆集

每节车厢有一个被记忆集，包含指向该车厢内部的所有来自更高序号车厢（包括后续列车）的引用。

每辆列车有一个被记忆集，包含指向该列车内部的所有来自更高序号列车的引用。列车的检测用于尽早检测出循环引用。

（2）管理列车

略

·新列车的创建时机一般是约定出来的规则，但是一定是给新对象使用的

（3）单节车厢的垃圾回收p317/334

核心理念就是尽可能将对象和引用它的对象放在同一辆车中或同一节车厢中，以期便于检测循环引用的垃圾

（4）恐慌模式p318/335

无效futile垃圾回收：针对第一节车厢无法在一步回收之后删除部分对象或转移至其他列车中的情况。此时需要改变算法策略，以期促使算法能够往下进行处理其他列车中的内容。

## 7.8 垃圾回收中的高级论题

7.8.1 并行和并发垃圾回收

并行性parallel：使用多线程或类似机制同时完成多个任务，多任务之间彼此一般较独立或有同步时刻的情况，但绝对不会出现彼此干扰。如回收器的两个以上线程分担工作。

并发性concurrent：两个任务都会对相同数据产生更改，彼此干扰，需要解决冲突。如回收器和增变者。

一个并行、并发垃圾回收算法的描述：p320/337

7.8.2 部分对象重新定位p321/338

拷贝世代垃圾回收器：

一个混合型的回收方案：

7.8.3 类型不安全的语言的保守垃圾回收

基于C++和地址数据扫描识别的垃圾回收器原理：p321/338

7.8.4 弱引用weak reference

弱引用不会被垃圾回收器跟踪，因而只被弱引用指向的对象会被认为是不可达的。这种技术在需要将可达但不再使用的对象回收时很有用，原本只能依靠手动回收，现在垃圾回收器可以“识别”这些对象。

# 第8章 代码生成

·代码优化和代码生成统称为编译器后端back end

代码生成器的3个主要任务：

指令选择、寄存器分配与指派、指令排序

## 8.1 代码生成器设计中的问题

8.1.1 代码生成器的输入

代码生成过程以编译器前端生成的中间表示和相关的符号表信息作为输入，产生目标程序。

8.1.2 目标程序

指令集体系结构：

·RISC（精简指令集）——较多寄存器、三地址指令、简单寻址

·CISC（复杂指令集）——较少寄存器、两地址指令、多种寻址和寄存器类型、可变长度指令和具有副作用指令

·基于堆栈——JVM，除此之外已无实用性

即时Java编译器：JIT，just-in-time，用以在运行时刻翻译Java字节码至机器码的编译器，这是一种非常规提升Java程序性能的手段

目标模块object module：可重定位的机器语言程序块，目标模块通常需要彼此链接和加载至一起形成可执行程序。不可重定位或使用绝对地址的程序虽然速度快但是灵活性差。

·使用汇编程序代码（也称符号指令）作为代码生成器的输出（此时不再是一步到位的代码生成）可以简单化代码生成过程，然后使用汇编器的宏机制可以生成目标代码。代价是多了一个步骤。

8.1.3 指令选择

指令选择的影响因素：

·IR的层次——中间代码层次越低，代码生成效率和质量越高，反之可能必须优化

·指令集体系结构——机器给出的可选指令不同自然结果不同

·目标代码的质量

最简单的代码生成器就是使用一个类似代码模板的替换方式来翻译IR的每个语句模式。这种逐句无脑替换自然会在特定上下文中产生各种优化问题。

8.1.4 寄存器分配

两个子问题：

分配和指派

·寄存器分派是一个NP完全问题

·本节例子8.1中有一个笔误，使用寄存器对进行乘除操作的指令的第一个参数都不是任何参与运算的元素，而是用于运算的寄存器对的偶数号寄存器（一般来说寄存器对就是一个偶数号寄存器和其后续相邻的奇数号寄存器）。这样的解释会形成一个统一局面：乘法中的结果和除法中的被除数均占据全部寄存器对，乘法中的第一个被乘数和除法中的商均占据奇数号寄存器。

8.1.5 求值顺序

·这也是一个NP完全问题

## 8.2 目标语言

8.2.1 一个简单的目标机模型

可用指令举例：p330/347（以汇编语言表示）

·寄存器名和变量名一样都是代表一个地址。但是两者的寻址方式和代价是不一样的。

·注意很多指令都是采用“关键字 第一个地址 第二个地址”的模式或类似模式的，然而解释时往往是“第二个地址中的内容”存于“第一个地址”。也可以理解为，解释时在第二个地址前有一个省略的解引用的符号。书中给出的a(r)或100(r)的写法解释都是在第二个地址位置上，而实际上a(r)的含义就仅仅是a + contents(r)，即只是一个地址。

寻址模式举例：（寻址模式即目标语言中的地址或变量的表示方式）p331/348

8.2.2 程序和指令的代价

一些有关指令执行代价的约定：p332/349

·指令长度和指令代价此处假设是正相关的，而代价在这里的文中就是对应指令的字长（一个字4字节）。寄存器寻址模式字长为0，指令不包括参数本身占用字长为1，这些将会决定指令序列的地址分布。

·寄存器寻址模式或参数不占用字长，可能是因为寄存器数量不太多以至于可以和指令本身占用同一个字的缘故。

## 8.3 目标代码中的地址

·本节针对静态和栈式内存分配为过程调用和返回来生成代码

8.3.1 静态分配p334/351

即跳转地址和过程的活动记录在编译时可确定，这并不现实或常用，只是个引出性例子。

8.3.2 栈分配p335/352

使用相对地址保存活动记录，该记录是动态分配位置的，只有在运行时寄存器中方可得到其地址。

8.3.3 名字的运行时刻地址p337/354

所有代码中的名字最终在目标代码里都必须替代为对应的存储位置，可以是静态或动态确定的。

·IR中的名字可以是指向符号表条目的指针，这便于后续代码生成的处理

## 8.4 基本块和流图

基本块basic block：实际上就是三地址指令序列的若干条相邻指令组成的块单位，该块中的控制流只能从首指令进入和从尾指令离开，具体条件和标准见p338/355

流图flow graph：即控制流图，基本块就是流图的一个结点

·基本块和流图可以给出控制流的真实走向，这可以用于优化目标代码，去除那些单靠模板式翻译造成的多余指令。

8.4.1 基本块

基于先寻找所有基本块的首指令leader的基本块划分算法：p339/356

8.4.2 后续使用信息

有关名字的“使用use”和“活跃live”的概念：p340/357

对一个基本块的语句确定名字活跃性和后续使用信息的算法：p340/357

·本节中对“活跃”概念的定义可能有笔误，应该是在语句j处活跃（或从i到j均是活跃的）

8.4.3 流图

入口entry和出口exit：流图的虚拟起止点

8.4.4 流图的表示方式

略，p341/358

8.4.5 循环

流图中的循环——结点集合的定义：p341/358

## 8.5 基本块的优化

本节侧重对基本块本身进行局部优化，这本身可以实质性优化代码性能。全局优化见后文

8.5.1 基本块的DAG表示

基本块的DAG构造方法：p343/360

8.5.2 寻找局部公共子表达式

局部公共子表达式：具有同样运算符结点和同样子结点（顺序也相同）的表达式，在DAG中可以使用相同的结点合并。

·局部公共子表达式只从形式上还无法优化所有情况，必须借助其他技术才能发现所有可优化的“多余”代码。

8.5.3 消除死代码

消除为不活跃变量赋值操作的语句：p344/361

·本小节例子是在假设情况下做出的示例，需要注意。

8.5.4 代数恒等式的使用

本节实际上在介绍一些代数和计算层面的优化，包括：

·代数恒等式：数值上相等可化简的运算

·局部强度消减reduction in strength：用代价低的运算代替代价高的同结果运算（如乘法代替低阶幂运算）

·常量合并constant folding：编译时对常量表达式求值，然后用该值替换原表达式（实际上是先编译常量表达式然后在目标机执行以保证得到和运行时相同的结果）

·代数层面的优化不一定全部可行，一方面是受到计算机本身的特点限制（溢出等），另一方面来自于源语言的标准规定。

8.5.5 数组引用的表示

数组引用在DAG中的表示法：p345/362

·因为数组表示一组值，而数组索引不确定，所以造成了特殊性

8.5.6 指针赋值和过程调用

局部情况下，缺少对指针和过程调用真实作用域的了解，所以一般必须将所有可能的名字全部处理。换句话说，指针和过程调用的存在会严重阻碍基本块和局部公共子表达式的优化（只能借助更广泛层面的优化）。

8.5.7 从DAG到基本块的重组

本节介绍如何从DAG重新回归到基本块的优化结果

优化重组后的基本块语句顺序：p347/364

## 8.6 一个简单的代码生成器

8.6.1 寄存器和地址描述符register and address descriptor

代码生成器所需的数据结构描述

8.6.2 代码生成算法p349/366

算法按照下列分类描述：

①运算的机器指令

②复制语句的机器指令

③基本块的收尾处理

④管理寄存器和地址描述符

8.6.3 函数getReg的设计

函数的算法描述：p352/369

溢出操作spill：准确翻译应为吐出，是指寄存器不够存储数据后需要将部分数据存于内存的操作，以便于空出寄存器来容纳新数据

## 8.7 窥孔优化peephole optimization

利用在目标代码指令上的一个滑动窗口来进行局部优化代码的技术，也可以用于中间代码优化。这只是一种简单的优化技术，也更适用于较原始的代码生成算法。

8.7.1 消除冗余的加载和保存指令

本小节的内容疑似存在问题：

如果R0中的数据被修改，即a有了新值，则需要将新值储存回a的原本地址

8.7.2 消除不可达代码

紧跟在无条件跳转之后的不带标号的指令可以删除

8.7.3 控制流优化

几个经验例子p354/371

8.7.4 代数化简和强度消减

几个经验例子p355/372

8.7.5 使用机器特有的指令

一般的机器特有指令均为硬件优化某些操作而设，使用它们代替普通指令按道理一定可以优化代码

## 8.8 寄存器分配和指派

分配：哪个值应该存放于寄存器中

指派：值应该放在哪个寄存器

·虽然专用寄存器的存在（如专用于基地址和栈顶指针的寄存器）会导致整体上寄存器利用率的降低，但是这是实际规律与理论规律的不同或不全面吻合之处。专用寄存器保证某些功能或操作的执行一定顺利，反之可能会造成某些关键操作无法执行而陷入类似死锁的境地。

8.8.1 全局寄存器分配

策略之一：分配固定多个寄存器存放每个内部循环中最活跃的值

8.8.2 使用计数

计算某个变量使用寄存器存储的效益的公式（近似）：p356/373

·通过计算以上效益，可以得出分配寄存器的最优变量选择。

8.8.3 外层循环的寄存器指派

类似内层循环，使用相似过程与方法

8.8.4 通过图着色方法进行寄存器分配

寄存器冲突图register-interference graph：用于描述名字或符号活跃期冲突的图，进行可用寄存器数目的着色即可确立一个寄存器分配方案。但是只能使用经验方法，因为原问题是NP完全的。

·如果需要spill寄存器至内存，如何指派寄存器进行spill也是使用启发式规则，总原则是避免在内循环中引入spill操作，因为内循环是执行次数最多且性能最关键的部分。

## 8.9 通过树重写来选择指令

DAG匹配可以产生比树匹配更好的代码，但是也更加复杂。

·本节中介绍的技术不仅仅是针对三地址代码或其他中间表示形式，而是有可能可以用于为源代码直接生成目标代码的情况。

8.9.1 树翻译方案tree-translation scheme

输入：在代码中插入运行时刻地址后的树形表示形式

输出：目标代码的指令序列（寄存器的分配和指派问题另行解决）

·树翻译方案也称树重写规则，即将中间代码树归约转化或翻译为目标指令的方案。这其实类似于前端的自底向上语法制导翻译。树的重写或归约过程也称替换和覆盖tiling，最终树会变为一个根节点。

8.9.2 通过覆盖一个输入树来生成代码

一个示意性的例子，并非介绍算法

8.9.3 通过扫描进行模式匹配

利用LR语法分析方法来实现树匹配：

·树模式或树重写规则替换SDT的产生式与对应动作

·树模式或指令模板采用前序遍历（或前缀表示）进行记录来代替产生式体

·使用LR语法分析器来表达这个SDT

使用LR语法分析方法来生成代码的优劣势：p363/379

·缺乏指令代价信息时，经验策略倾向于执行较大的归约，即尽量生成较少的目标代码进行最多的运算。

8.9.4 用于语义检查的例程

略，介绍了一个使用断言来让同一个产生式为不同情况生成不同代码的例子

8.9.5 通用的树匹配方法

方法：将树模式模板写成从根到叶子节点的串集合形式，采用串模式匹配的自顶向下算法进行匹配。

·此处内容有些复杂但是却没有给出真正的实例与代码。只是一笔带过如何实现通用意义上的树匹配。也许串模式匹配较LR方法更优的原因是串模式匹配倾向于发现最长的匹配，即可能发现最节省指令的匹配。但是具体算法书中未给出。

## 8.10 表达式的优化代码的生成

使用条件：一个基本块只包含单一的表达式求值，或逐次处理各个表达式的方式生成代码

8.10.1 Ershov数

含义：表示若不将任何临时值放回内存，则计算该表达式需要多少寄存器

计算方法：p365/382

·理解方法是每个叶子节点代表一个值，显然需要1个寄存器。只有一个子节点的运算不会需要更多的寄存器。两个相同标号的节点需要等量的寄存器，最理想情况下第一个节点完成计算后便只占用一个寄存器来存储，而第二个节点完成计算过程中总占用寄存器数势必要比标号多1。而不同标号节点则同理，标号小的节点计算过程最多比标号大的节点少占用1个寄存器。

8.10.2 从带标号的表达式树生成代码

算法：p365/382

·这是一个从根节点递归的算法

8.10.3 寄存器数量不足时的表达式求值

改进算法：p366/383

·因为寄存器使用数目的上升（至少从2开始）其实就是因为多用一个寄存器存储二目运算符的一个运算分量，如果将该运算分量存于内存再提取，那么寄存器数目将永远不会上升（每次运算将使用固定数目的寄存器）。当然，尽可能利用多的寄存器少交换内存可以提高计算效率。

## 8.11 使用动态规划的代码生成

8.11.1 连续求值

定义：按照从左到右或从右到左的顺序先计算一些结果并存于内存，再进行后续其他的运算（没有特殊需求的情况下这也是最一般的运算顺序）。反之非连续求值的顺序是比较复杂的，比如使用奇偶寄存器对进行乘除运算的机器，可能不按照从左到右或从右到左的单向顺序，也可能先计算出的结果是存储于寄存器中的。

·可连续求值的条件（由机器特性决定）是使用动态规划的先决条件，否则不同子问题之间存在彼此制约。

8.11.2 动态规划的算法

算法：p369/386

·注意：

①此算法假设机器存在op R, R, M的寻址方式。

②因为运算符并不都存在交换律，且指令op R, R, M显然有运算顺序，所以此算法统一认定只能将右子树置于内存。若将左子树置于内存则必定多一加载指令，而使用内存的情况下因为左右子树均可使用同等数目的寄存器，右子树的计算不会节省更多代价。

·问题：（本节有诸多问题和混淆之处）

①指令的代价和寻址方式相关，为何op R, R, M和op R, R, R的代价一致？

②只有一个可用寄存器的情况下，为何可以选择使用两个寄存器的最优解？

# 第9章 机器无关优化

## 9.1 优化的主要来源

9.1.1 冗余的原因

·有时冗余是高级语言编程的副产物，但是并非是坏事，由编译器去发现一些固定模式的冗余并将这些内容对程序员透明，可以起到很好的封装和效率作用。

9.1.2 一个贯穿本章的例子：快速排序

略

9.1.3 保持语义不变的转换

略

9.1.4 全局公共子表达式common subexpression

略，公共子表达式可以省略求值，但是必须确保中间的值没有改动

9.1.5 复制传播

复制语句copy statement

复制传播转换：基本思想是在u=v之后的语句中，尽可能用v来替代u，从而构造消除u的机会。一般来说光靠这个技术无法明显优化代码，可以说这是一个“中间优化”的优化，为了其他进一步的优化而做的改变。

9.1.6 死代码消除

常量折叠可以说是复制传播的一种特殊例子。

9.1.7 代码移动code motion

原理：针对循环不变计算而进行的通用型改进，将这些循环不变计算式在循环外进行单次计算，在循环中只使用计算结果来工作。

9.1.8 归纳变量和强度消减strength reduction

归纳变量：对于一个变量x，若存在一个常数c，使得每次x的值总是增加c，则x就是归纳变量。或者说常见的就是循环变量。

·多个归纳变量之间如果存在相同的变化规律，那么将其缩减至一个，可以起到这样的作用——用该变量的低强度运算（加减）代替被缩减的变量的高强度运算（乘除）。

## 9.2 数据流分析简介data-flow analyze

9.2.1 数据流抽象

执行路径execution path：简称路径，定义p383/400

程序点：类似于断点的位置，位于代码语句之间。

9.2.2 数据流分析模式

数据流值data-flow value：某个程序点可能观察到的所有程序状态的集合的抽象表示。这种抽象表示仅随需求而定，并非是某种确定的数值标准，另外也没有完全完备的程序状态的表达方式（也不需要）。

域domain：所有可能的数据流值的集合

（1）传递函数transfer function

即能够直接改变和重新计算数据流值的语句，抽象为函数

（2）控制流约束

从控制流获得的数据流值信息

9.2.3 基本块上的数据流模式

略

9.2.4 到达定值reaching definition

含义：可能沿着某条路径到达某个程序点的定值

·本节中的IN和OUT在下一节中会有不同含义，此处是定值集合，下节是活跃变量集合

·本章内容假设仅仅处理没有别名的程序变量，即不通过过程参数、数组和间接引用访问的变量。

（1）到达定值的传递方程

略，见书中对基本块的gen和kill的推导

·kill集合包括语句顺序上后续的同变量定值语句，这是因为在有循环等不定控制流的情况下，后续语句可能会在某次执行中率先执行。然而其他情况下kill的全局屏蔽效果还是会被gen重新补齐的。

（2）控制流方程

略

交汇运算meet operator：将数据流在基本块的上下文中整合时使用的运算，到达定值采用并集运算，而后面的可用表达式采用交集。

（3）到达定值的迭代算法

最小不动点least fixedpoint：方程组的多个解是多个彼此包含的集合时，其中最小的集合代表最小规模的解。这也是到达定值分析所需要的结果。为何需要这样的选择完全取决于对到达定值信息的处理需求，因为一般求取到达定值都是看到底哪些值可能会到达该处。所以使用并集来交汇，方程组的理想结果是某个最小集合，其余的结果均是保守的“过多估计”。

算法：p388/405

·检测迭代结果变化的算法被省略

9.2.5 活跃变量分析live-variable analysis

算法：基本上是到达定值分析算法的逆向p391/408

9.2.6 可用表达式

·可用表达式分析的结果是最大不动点，因此决定了它的收敛是从大到小。之所以可以采用这样的方法而不产生问题，是因为可用表达式的交汇运算采用了交集，保证可用表达式一定可用而非可能。

算法：p393/410

9.2.7 小结

对三种数据流分析问题的总结p393/410

## 9.3 数据流分析基础

数据流分析框架包含的元素：

D——数据流方向，包括前向FORWARD或逆向BACKWARD

半格——V值集和交汇运算Λ

F——从V到V的传递函数族

9.3.1 半格semilattice

·本节部分中的偏序<=和交汇运算都是一个普遍抽象概念，交汇运算可能实质上是并集运算，而偏序也只是用一个符号（书中用了<=）代表某种实际的比较关系，实质上可能是包含关系（即左边集合更大）。进一步讲，所谓的顶元素或者最大下界不一定代表实质上最大的集合，只是偏序上定义的极值而已，可能和真实情况相反。因此思考这部分问题时，有时不要具体化思考，而是关注于公式和符号的抽象化规律。

半格的定义：满足一定条件的集合与二元交汇运算，p396/413

顶元素：对集合中所有的元素交汇结果为该元素

底元素：对集合中所有元素的交汇结果为底元素

①偏序partial order

定义：满足一定性质的某种比较关系p396/413

偏序集：partially ordered set, poset。一个集合和定义在其上的一个偏序

②半格的偏序

利用半格的交汇运算性质可以定义一个偏序

幂集power set：一个全集U的所有子集组成的集合，即为U的幂集

③最大下界greatest lower bound，glb

·这里的最大最小显然是参照偏序来讲的。

最小上界least upper bound，lub

交半格meet semilattice

并半格join semilattice

④格图

集合V中的元素各自为节点，每个偏序关系“从大到小”产生一条有向边。部分长边可以用短边路径来省略（基于偏序传递性）。

⑤乘积格

定义：格的乘积，结果的交汇运算与偏序也存在对应定义，p398/415

⑥半格的高度height

上升链ascending chain：一个满足严格递增顺序的偏序序列

定义：上升链中递增关系的个数的最大值，即比最长上升链元素个数少1，p399/416

9.3.2 传递函数

性质：p399/416

①单调的框架monotone

框架单调定义：p399/416

②可分配的框架

可分配条件distributivity condition：即单调性中的特殊情况，p400/417

·到达定值框架满足可分配条件

9.3.3 通用框架的迭代算法

算法：p400/417

算法性质与证明：p401/418

如果单调，则结果为最大不动点Maximum Fixed Point，MFP

半格如果单调且有穷高度，则必定收敛

9.3.4 数据流解的含义

①理想解

略

②基于路径交汇运算的解

·数据流分析中，寻找所有的可能执行路径是不可判定问题。实际上，所有的路径是可以发现的，只是因为各种原因无法确定实际中真正可行的执行路径，即势必包含不可行的执行路径。但是这种“保守”的路径集合并不会导致错误。

③最大不动点和MOP解

MOP定义：基于所有路径交汇计算所得结果，p403/420

·最大不动点或最小不动点MFP可以理解是算法执行的所有结点的交汇计算结果，算法一般按照结点编号顺序访问并交汇，和MOP计算方式的区别就是交汇的顺序。二者在可分配框架下结果一致，否则算法结果相对于MOP讲更加保守。

## 9.4 常量传播

9.4.1 常量传播框架的数据流值

常量传播框架为乘积格，每个格或分量对应一个程序数据名（不一定是常量或变量），每个格的信息或值域含有：

·所有符合该数据类型的可能的常量取值（不是程序中出现过的，而是数据定义允许的全部取值范围，一般也是无穷界的）

·NAC值：not a constant，非常量值，底元素

·UNDEF值：未确定值，顶元素

映射：常量传播框架中的数据流值不再是某个值或值的集合，而是程序中的各个变量到其对应半格中的某个常量值的映射。这个映射不是单个变量的，而是全部变量的共同映射或映射集合。换句话说映射m可以以任意变量v为参数，而非只有一个映射值。而m的偏序则由格图定义，UNDEF>所有常量值>NAC。

·常量传播框架和前面所述框架有相同也有不同之处，需要分开理解。

9.4.2 常量传播框架的交汇运算

定义，P405/422

9.4.3 常量传播框架的传递函数

定义，p405/422-406/423

9.4.4 常量传递框架的单调性

论证，略

9.4.5 常量传播框架的不可分配性

论证，因为此框架较简单无法发现不同路径中恒等变量的存在。

9.4.6 对算法结果的解释

·算法的保守假设是将可能取值为同一个常量值的变量视为任何情况下均相同的常量，即便有些路径上该变量没有定值。这在一般情况下合法，除非语言规定未定值变量需要有特殊的定值。

## 9.5 部分冗余消除

部分冗余问题本身可能需要4种不同的数据流分析才能完成。

9.5.1 冗余的来源

①全局公共子表达式

·有时寻找公共子表达式的算法只能单次发掘字面上相同的表达式，此时可以重复应用该算法来发现化简后出现的新公共子表达式。或者也可以使用不同的框架来分析数据流。

②循环不变表达式

·为了避免出现有时控制流提前离开循环或完全未进行循环时，循环不变代码移动造成的反向性能提升，编译器会将while循环改写为条件语句，将循环不变式计算置于条件和循环体之间，从而避免了循环条件不满足时仍提前计算了循环不变式的情况。

③部分冗余表达式

·需要放置新的表达式指令

9.5.2 可能消除所有冗余吗

关键边critical edge：所有从一个具有多个后继的结点到另一个具有多个前驱的结点的边

·消除全部冗余不一定导致代码最优，结果可能是没有冗余但代码反变长很多。因此对于需要复制代码的消余操作往往禁止，只进行添加并移动代码块的操作。

9.5.3懒惰代码移动问题

含义：以尽可能延迟计算为目标的部分冗余消除优化

①完全冗余

略

②部分冗余

略

9.5.4 表达式的预期执行

割集cut set：一个图中的一个边集合，若将该集合从图中删除则得到分割的两个部分，且该边集合的任意真子集都做不到分割的结果。

预期执行anticipated：若从程序点p出发的所有路径最终都会计算表达式e的值，且那时e中各个变量的值和在p处的时候完全相同，则表达式e在p上被预期执行。

·详细算法细节最好参考后文例子学习，文字叙述非常抽象

9.5.5 懒惰代码移动算法

4步，4个数据流分析问题

①算法概览

略

部分冗余消除中的4个数据流分析框架：p415/432

②预处理步骤

略

③预期执行的表达式

略

④可用表达式

略

·分析技术可分为4种，按照两个二分来分类：前向或逆向，并集或交集作为交汇运算

⑤可后延postponable表达式

略

⑥被使用的used表达式

实际上就是表达式的活跃性分析。

·这部分内容中，在程序点p处“被使用”“活跃”“预期执行”等概念都是违反汉语语言常理的含义（可能和翻译有关）。实质上，往往都是指变量或表达式的某个特点值在p处有效（在其他地方该变量或表达式的值可能改变或无意义），而真正使用和计算这些变量与表达式的具体操作并不是概念描述的主要对象。

⑦综合全部步骤

懒惰代码移动算法：p417/434

⑧总结

## 9.6 流图中的循环

9.6.1 支配结点dominate

定义：每一条从入口出发到结点n的路径如果都经过结点d，则d dom n

支配结点树dominator tree：表示支配信息的树结构

直接支配结点immediate dominator：支配结点树中的直接父节点

寻找支配结点的算法：（基于数据流框架）p420/437

Dom的性质：p420/437

9.6.2 深度优先排序

深度优先排序和后序遍历的顺序完全相反，注意后序遍历和前序遍历并非反向的。

深度优先生成树和深度优先排序算法：p421/438

·在这里的流图中并不在乎子节点的左右顺序，只要保证一致即可。

9.6.3 深度优先生成树中的边

3类：

·前进边advancing edge：从结点指向生成树中真后代（不是单纯指向顺序编号较大的结点），DFST深度优先生成树中所有的边都是前进边，但是反之不一定成立

·后退边retreating edge：从结点回退指向深度优先生成树中的祖先

·交叉边cross edge：两个结点互相不为祖先

m→n是后退边的充要条件是m的深度优先排序编号dfn不小于n。证明见p423/440。这里有一个理解难点，若m到n的边存在，则n的编号不可能小于m且n为深度小于m的非祖先结点。这也是交叉边的性质所在，交叉边一定是从小编号结点指向大编号结点。

9.6.4 回边和可归约性

回边：一条头结点支配尾结点的边（边的头就是箭头方向的指向）。回边必定为后退边，反之不成立。

可归约性reducible：若流图的任何DFST中的所有后退边都是回边

·如果一个流图可归约，则其所有DFST的后退边集合是相同的，且为回边集合。反之，不可归约的流图的DFST含有非回边的后退边，且因DFST构造时的顺序选择不同而不同。

·不可归约的另一个充要条件：删除流图中所有回边后得到的图仍带有环。

·循环结构的流图可归约时的特点为：循环只有一个入口，即对循环内其他结点起支配作用的结点。因此该循环的流图一定有回边，删除后，循环便可消失。

9.6.5 流图的深度

定义：各条无环路径上的后退边数目中的最大值。注意，无环路径不是说流图中无环，而且路径中不一定只有后退边。

9.6.6 自然循环natural loop

定义与性质：

1）具有唯一入口结点，称为循环头header

2）必然有一条进入循环头的回边

回边的自然循环natural loop of the edge：回边的头加上那些不经过头就能到达尾的结点集合。

构造回边的自然循环的算法：p424/441

·具有相同循环头的循环，要么是分离的，要么是嵌套的。分离的同头循环彼此不包含，一般可以将控制流最终合并在一起，当做一个循环。

最内层循环innermost loop：不包含其他循环的循环

9.6.7 迭代数据流算法的收敛速度

·到达定值、可用表达式和活跃变量分析都具有问题沿无环路径传播足矣的特点（即便考量有环情况，也可以只执行一次循环体而不影响结果），常量传递不可以，有时必须通过循环反复赋值才能得到真正的解。

·以深度优先顺序访问结点或根据情况逆序访问，可以达到加速收敛的效果。因为深度优先顺序往往代表了最长控制流路径的大致方向，所以按照该顺序或逆序可以加速数据流分析的推导过程。

## 9.7 基于区域的分析region-based analysis

9.7.1 区域

区域region：即一个特殊的流图结点集合，正式定义见p429/446

·区域是结点和边的集合，结点就是基本块。但区域的定义决定了它不一定是大基本块，实际上只是规定了控制流只能从首节点进入（且所有结点相连），而不一定从尾结点离开。

9.7.2 可归约流图的区域层次结构

叶子区域：每个基本块本身看做一个区域的初始化情况

循环体区域body region：不包括回到循环头的回边的条件下，循环中的结点和边的集合

循环区域loop region：包括所有回边时的循环的结点和边集合

构造一个可归约流图的自底向上的区域序列算法：p431/448

可归约流图的其他定义：p431/448

9.7.3 基于区域的分析技术概述

出口基本块exit block：区域中具有离开区域的边的基本块

略，对后续算法的文字描述

9.7.4 有关传递函数的必要假设

·组合：结点序列的传递函数串联

·交汇：相同尾结点的不同路径交汇

·闭包：执行一个循环0到n次之后的结果

9.7.5 一个基于区域的分析算法

基于区域的数据流分析算法：p433/450

·书中给出了一个超级详细的例子

9.7.6 处理不可归约流图

·处理不可归约的流图不能直接使用基于层次结构的方法，可以使用较早章节中的迭代算法，或者本节中的结点分割方法。

·本节中对于结点分割方法的描述非常混淆，结合例子说明：选取R2作为被复制结点，它有两个前驱，一个后继；两个副本的后继沿用原图，即均指向R3；原图的两个前驱各自链接到不同的副本上。

## 9.8 符号分析

参考变量：用来表示其他变量的值的变量，即不再关心所求变量的具体值，而是关心所求变量相对于参考变量的关系，即可用于分析程序的各种流以达到优化目的。比如编译时即可发现一些多余的比较关系，或是发现不同代码处所用数组下标的绝对差异从而可进行并行寻址优化。

9.8.1 参考变量的仿射表达式

仿射表达式：定义p438/455

线性表达式：属于仿射表达式的特殊情况，但是定义有分歧

①归纳变量induction variable

定义：通俗的说也就是循环变量，即所有在循环中随着循环次数等差递增的变量。归纳变量也可以作为参考变量。注意归纳变量并不一定在程序中存在或已定义，可以在分析中按照实际需要任意引入。

·所有可以被归纳变量仿射表达的变量均可以只通过加法运算求值，这也是强度消减。

②其他参考变量

·一般选取的参考变量的值都不是某种表达式形式的，否则一般它们也可以用该表达式中的变量表示。常见的参考变量一般选取外部输入值，随机值和其他复杂函数的返回结果，后文中的NAA变量就会被赋予别名用作参考变量供后续代码分析使用。

9.8.2 数据流问题的公式化

本节中将前文数据流问题中的数值流分析公式化为符号表达式流的分析。

①数据流值：符号化映射

顶元素，底元素的新定义：p440/457

非仿射表达式——NAA

②单个语句的传递函数

定义：p441/458

·本节中一个易混淆点是，传递函数是以映射为自变量和函数值的函数，而映射本身也是一个函数，以程序中的变量名为自变量，映射的结果仍然是一个表达式。可以说如果以具体数值为最终的函数值标准，那么这里蕴含三层函数。

③传递函数的组合

定义：p441/458

④数据流问题的解决方法

略

·符号化数据流分析只能依靠基于区域的算法，而不能使用迭代法。具体细节见后文。

9.8.3 基于区域的符号化分析

①传递函数的交汇运算

p442/459

②带参数的函数组合

程序中的变量的四种类型以及符号化映射定义：

基本归纳变量basic induction variable

符号化常量symbolic constant

非基本归纳变量——可由基本归纳变量和符号化常量的仿射表达式表示

非仿射表达式NAA

·循环的分析非常依赖循环的执行次数，尤其是循环是否至少能执行一次。另外是否确定循环的执行次数也很重要，不确定次数的循环会导致除了常量外的变量都无法定值，甚至用于符号化分析。当然这是从循环外的角度来看的，循环内变量之间的关系可能仍保持一致且不随执行次数影响。

③一个基于区域的算法

基于区域的符号化分析算法：p443/460

# 第10章 指令级并行性

大部分非数值应用程序难以使用指令级并行性，因为执行过程非常依赖输入和上一步的结果。只有数值计算应用程序遵循相对统一固定的执行过程，可以善加利用并行性。

## 10.1 处理器体系结构

10.1.1 指令流水线instruction pipeline和分支延时

指令流水线的步骤：（只是大致步骤，实际情况可能更加复杂，且部分步骤长度明显较长）

获取指令IF

解码ID

执行EX

访问内存MEM

写回结果WB

分支延时指处理器无法预测或错误预测分支语句的目标位置从而中断流水线的情况，因为只有彻底执行完毕后才能准确知道分支语句的目标。

10.1.2 流水线执行

流水线化pipelined：一条指令的后继指令在不需要本指令的计算结果时即可立刻向下执行，则本指令称为被流水线化。能否流水线化明显取决于技术限制和实际限制等因素。

·本节中对于流水线执行的定义说明的并不清晰，实际上既非定义也非准确描述。上述指令流水线的5个步骤只是通用的广泛例子，具体的指令流水线步骤因指令的硬件执行细节（或算法）而异。执行每个步骤的硬件部分一般称作某某单元，在流水线化的处理器中它们肯定是独立的个体。流水线化的单元的工作时间往往是确定并且一致的，一般均为1个时钟周期。

·一个指令是否流水线化不是高级语言程序决定的，而是该指令的硬件执行算法决定的。若该指令做不到任何情况下各个步骤的时间均统一，则显然它不是流水线化的。比如浮点数除法，部分步骤执行时间和输入相关，而且占用时间可长达数十个周期，则执行该指令时无法准确预测它所占用的单元何时会被释放，因而无法流水线化。

·另外，不同的指令就算是都流水线化，也存在步骤数不同的情况，然而每个周期一定会释放一个单元的资源，并不会在硬件资源上产生相互依赖。指令调度的目标之一就是确保指令运算结果的数据依赖性，因为流水线条件下不同的指令因步骤数不同而运行时间不同。

10.1.3 多指令发送

可同时执行的指令数目=指令发送宽度（即每时钟周期发送指令数）\*流水线中平均阶段数目

VLIW：非常长指令字very-long-instruction-word机器，依靠软件管理并发性的机器，在编译时就决定运算的并行性，并以此明确地使用复杂的机器代码来操作。

超标量superscalar机器：利用硬件管理并发性的机器，一般只用普通的指令集和顺序执行语义，在运行时自动检测指令之间的依赖关系。

指令字：要在同一时钟周期内发送的一条或多条指令的编码

·硬件管理情况可以使用先行静态指令调度来增加收益。

## 10.2 代码调度约束

代码调度要遵守的约束：

·控制依赖约束——必须执行原程序执行的所有运算

·数据依赖约束——和原程序结果相同

·资源约束——不能超额使用资源

·代码优化尤其是改变代码顺序的优化行为都会给调试过程带来困难。

10.2.1 数据依赖data dependence

3种类型的数据依赖：

·真依赖——先写再读

·反依赖——先读再写

·输出依赖——写之后再写

存储相关的依赖storage-related dependence：即反依赖和输出依赖，可以通过将值放在不同位置来消除这两种依赖

10.2.2 寻找内存访问之间的依赖关系

①数组的数据依赖分析

略，见后文11.6

②指针别名分析

互为别名aliased：两个指针指向同一对象

略，见12.4

③过程间分析

略，见12章

10.2.3 寄存器使用和并行性之间的折衷

硬件寄存器重命名hardware register renaming：一种对单纯的寄存器使用优化和并行性折衷的方法。寄存器分配算法实则也可考虑一定并行性的。

·寄存器的最少分配算法和并行化之间存在冲突，并行需要尽可能多的同时使用寄存器，而非先后使用。

10.2.4 寄存器分配阶段和代码调度阶段之间的顺序

两个阶段的顺序甚至穿插顺序依赖具体实用情况下的代码特性。

10.2.5 控制依赖control-dependent

定义：一个指令的是否执行取决于另一个指令的结果，即常见的各种条件表达式

10.2.6 对投机执行的支持speculatively

投机执行：对于可能会执行的指令在特定条件下提前执行，以获取可能的提速性。特定条件是该种指令近乎不需要耗费资源来执行。如内存加载指令。

①预取指令prefetch

高性能处理提供的一种特殊功能，支持投机性内存访问，且会自动处理访问错误的情况

②毒药位poison bit

寄存器的额外指示位，用于标记其是否加载了非法内存位置，从而可将加载时的合法性检测移至使用数据时（如果程序正确，这些数据不会被使用）p457/474

③带断言的执行predicted instruction

消除或者转化分支代码为普通的代码，但是对于性能的要求仍不小。p457/474

10.2.7 一个基本的机器模型

简单机器模型RT：p457/474

资源预约表resource-reservation table：二维表，记录某运算指令开始执行后某时钟周期占用某种资源的单元数

## 10.3 基本块调度

10.3.1 数据依赖图data-dependence graph

构造方法 p459/476

10.3.2 基本块的列表调度方法list scheduling

基本块列表调度算法：p460/477

·本算法可能存在书写错误，且描述并不全面：

①RT[s+i]可能应该写成RT[s][i]

②R应该是资源向量的ri

③s是时钟周期为单位的时间值，S则是调度算法给出的对应每个节点的运行时刻记录

10.3.3 带优先级的拓扑排序

关键路径critical path：数据依赖图中的最长路径

·优先级的度量选取可以是结点高度，或资源使用/可用数量比值。高度和比值越高，优先级越高。二级优先级（解决一级优先级中的平级问题）是源代码中的出现顺序。

## 10.4 全局代码调度global scheduling

基本块内部调度由于条件限制一般都不能完全利用机器的全部并行性能，在全局代码调度的层面下，一个基本块执行时的剩余性能会被另一个并行基本块使用。

10.4.1 基本的代码移动

P479 ++5