程序分析基础

Foundations of Program Analysis

周晓聪(isszxc@mail.sysu.edu.cn)
中山大学计算机科学系, 广州 510275

2018 年 3 月 25 日

第一章　程序要素

程序分析的对象是程序，因此我们首先要对程序本身及其性质进行了解。这里我们讨论程序的要素(elements)，或说程序的基本构成(component)，并收集一些与程序及程序分析相关的术语，这些术语广泛地用于软件开发、软件测试乃至计算机科学中。

1.1　程序的基本构成

1.1.1　什么是程序?

什么是程序？按照维基百科的解释[1]，计算机程序是为了使计算机执行某个特定任务而编写的指令序列，也即计算机程序是计算机能够执行的指令序列，但当前的程序通常有两种形式的指令序列：一种称为程序的源代码(source code)，由程序员编写并能为人所理解的形式；一种称为程序的机器代码(machine code)，可由计算机理解并直接运行的形式。编译器或解释器负责自动地将程序的源代码翻译成（等价的）机器代码。

将程序看作是指令序列这是从计算机角度的低层抽象，从程序员的角度看，程序是对执行某个特定任务的一种描述，这种描述用某种程序设计语言完成。程序设计语言是一种人工设计的语言，用来描述计算机如何执行某个特定任务，程序是使用程序设计语言进行描述的产品，是对计算机所执行任务的一种抽象。

因此，讨论程序的基本构成实际上是要讨论程序设计语言提供了怎样的描述机制，不同的程序设计语言机制产生程序中不同的要素，或说构件。

一个程序设计语言提供哪些用于描述计算机怎样完成任务的机制首先取决于这种语言所采用的“思维模式”，或者程序设计理论教材所说的“程序设计语言风范(paradigm)”，例如过程式(procedural)、面向对象式(object-oriented)、逻辑式(logical)以及函数式(functional)等等。不同的风范实际上体现在对如何描述计算机完成任务有不同的理解，或者说提供了不同的描述任务如何由计算机完成的思维方式。不同风范的程序设计语言所编写的程序也就提供了计算机所完成任务的不同角度的抽象。因此不同的风范也就是我们理解程序的纲，或说最基本的切入点：不同的风范决定了我们在编写、调试和测试程序时有不同的思维方式和基本方法。

过程式和面向对象式的程序设计语言风范都属于命令式(imperative)风范，其核心是通过详细给出计算机完成任务的具体步骤来描述任务。过程式风范更侧重于对完成任务的步骤本身，以及这些步骤的实施顺序，或者用计算机的术语就是侧重于描述完成任务的算法，从而过程式程序设计语言编写的程序是完成任务的算法的实现。面向对象式风范进一步考察完成任务的步骤由谁实施，考

[1] http://en.wikipedia.org/wiki/Computer_program

察实施这些步骤的实体及它们之间的关系，在此基础上再描述这些实体如何完成所要完成的任务。因此面向对象程序可看作是完成任务的实体在计算机中的抽象。

而逻辑式和函数式的程序设计语言风范都属于声明式(declarative)风范，其核心是通过给出计算机所完成任务的基本性质或所要满足的约束来描述任务。函数式风范描述完成任务可用的资源（输入）与任务的目标（输出）之间的函数关系及其性质，因此函数式程序就是将任务抽象成函数之后在计算机中的表示。逻辑式风范描述任务所要满足的约束，因此逻辑式程序就是任务的约束规则在计算机中的表示。

我们这里只讨论命令式的程序设计语言风范及其程序，因为命令式风范所采用的思维模式与我们用自然语言描述完成某项任务的方式相近。根据上面的讨论，我们可将命令式程序设计语言编写的**程序理解为完成某项任务的实体和过程在计算机中的抽象**。

注意到，我们很难对“任务(task)”、“实体(entity)”和“过程(procedure)”等做更深入的阐述，因为这些名词的含义过于宽泛。不过在描述现实世界的任务我们通常会给出：(1) 任务的目标，这是描述任务最基本的要素；(2) 可用的资源，完成任务所需要的人力、物力、财力及信息等；(3) 任务的背景，任务需要在怎样的环境下实施与完成。

最后，可以使用表1.1所示的现实世界概念和计算机世界概念的对应方式给出我们对程序的基本理解。

表 1.1 程序的基本理解

现实世界	计算机世界
任务	程序的需求
完成任务的实体和过程	程序
任务的目标	程序的输出
可用的资源	程序的输入
任务的背景	程序开发与运行的环境

1.1.2 程序要素的分类与理解角度

在上述对程序的基本理解的基础上，我们还可从计算机世界角度对“实体”和“过程”进行解读：我们可认为实体是指一切能编码到计算机从而能存储起来的东西的抽象，而过程是指一切改变计算机所存储东西的内容的操作或活动的抽象。这样实体和过程实际上给出了对程序要素最基本的分类，实体对应程序中作为**数据**(data)**抽象**的程序要素，而过程对应程序中作为**控制**(control)**抽象**的程序要素。

上面的模型还给出了我们的一个基本观点：我们在考虑程序的性质时，总是从两个抽象层次来考虑：一是与现实世界相关的高层抽象，二是与计算机世界相关的底层抽象。在现实世界里，我们使用任务、实体、过程、功能等这样的名词，而在计算机世界里，我们使用存储、数据、控制、操作、指令这样的名词。

例如，在现实世界我们说“将一堆电话号码按字典顺序排序”，而在计算机世界里，我们可能是“将一堆字符串按字典顺序排序”，这里“电话号码”是实体，而“字符串”是“电话号码”在计算

机中一种可能的存储表示；虽然我们在现实世界（程序需求）和计算机世界（程序实现）中都说“排序”，但在现实世界里我们更多的是想到“按顺序排好的电话号码”，而在计算机世界里我们可能更多地是想到“使用比较和交换来完成排序”。

在软件开发或程序设计中，我们通常将离计算机世界比较近的东西称为“**物理的**(physical)”，而将离现实世界比较近的东西称为“**逻辑的**(logical)”。程序本身是离计算机世界比较近的东西，是物理的，但有经验的程序员在编写程序时，往往想到的是程序所对应的现实世界中的东西。例如，我们在编写程序实现“将一堆电话号码按字典顺序排序”时，有经验的程序不会将自己的思维局限在操作字符串（电话号码的表示），而是认为在操作电话号码本身。这一点很重要，初学编程的人员往往容易陷入程序的一些与计算机相关的细节，而无法看到程序与现实世界相关的抽象。我们在这里讨论这两种层次的抽象也是希望我们在考虑程序要素的时候能够思考这些要素所对应的现实世界的抽象是什么，从而更深入地理解程序的实质。

除了从物理和逻辑的角度看程序要素之外，还有两种角度对理解程序要素也非常重要：一种是**语法**(syntax)和**语义**(semantic)的角度；一种是**静态**(static)和**动态**(dynamic)的角度。

我们说程序是完成某项任务的实体和过程在计算机中的描述，语法和语义分别指这种描述的形式和内容，语法指怎样的程序要素，或者程序要素怎样组合具有正确的描述形式，而语义则规定我们如何去理解程序要素的含义，或者说如何将程序要素与现实世界中的东西，或者对现实世界所建立的其他模型（例如描述现实世界的数学模型）中的东西相关联。

程序要在计算机上运行，任务要实施以达到任务目标，静态和动态分别指是程序要素在程序运行前（任务实施前）还是程序运行过程中（任务实施过程中）所表现出来的性质。静态考察程序要素在程序运行之前表现出来的性质，或者说是在程序运行之前就存在的程序要素，而动态则考察程序要素在程序运行之中才能表现出来的性质，或者说是在程序运行之中才能存在的程序要素。

综上，我们首先将程序的要素分为作为数据抽象的要素和作为控制抽象的要素，分别对应现实世界中完成任务的实体和过程，其次我们提出可从逻辑层/物理层、语法/语义以及静态/动态的角度理解程序要素。我们可使用表1.2总结这些对程序要素进行分类和理解的角度。

表 1.2 理解程序要素的角度

角度种类	角度	含义
从模型构成类型出发	数据抽象	与完成任务的实体相关的程序要素
	控制抽象	与完成任务的过程相关的要素
从模型抽象层次出发	物理层次	与计算机紧密相关的程序要素
	逻辑层次	与现实世界紧密相关的程序要素
从模型表示方式出发	语法层次	与描述模型的形式相关的程序要素
	语义层次	与描述模型的含义相关的程序要素
从程序运行与否出发	静态	与程序动态运行无关的程序要素
	动态	与程序动态运行相关的程序要素

当然，上述分类与理解角度只是帮助我们更好地理解程序要素，而并没有要强制地将所有程序要素做对应的划分，有些程序要素可能并不容易将其归到哪一类。

下面我们将详细展开讨论程序的要素，首先从语法的角度讨论程序的要素，因为我们通常也是

首先注意到程序的表达形式，然后我们分数据抽象和控制抽象讨论各种程序要素，这时主要从语义的角度去解释这些要素，并讨论这些要素可能在逻辑层的解释，以及从静态和动态的角度对这些要素的基本性质进行讨论。

1.1.3 程序语法与程序要素

我们知道，程序无论是解释执行还是编译执行，都会经过词法分析(lex analysis)和语法分析(syntax parse)两个阶段。词法分析将源程序分解成单词(token)序列，而语法分析根据程序的语法规则构成语法分析树(parse tree)。因此，从语法层次讨论程序要素也可从这两个角度进行讨论。

程序的词法要素

从词法分析的角度，程序就是单词的序列，**单词是构成程序的最基本单位**。当然，我们还可从更底层次的角度说程序就是字符串，但这对我们理解程序没有意义，所以单词是构成程序的最基本单位，从理解程序出发，单词已经不可再分解更细的要素。

构成程序的单词可进一步分为**保留字**(reserve words)、**常量**(constants)、**标识符**(identifiers)和**特殊符**(special symbols)。程序中用来分隔单词的符号称为**空白符**(blank symbols)，包括空格(space)、制表符(table)、换行(new line)以及注释(comments)。对于程序的词法分析，注释与空格、制表符换行等作用相同，都是用来分隔单词。

程序 1.1 HelloPrinter.java

```
1 public class HelloPrinter {
2     public static void main(String[] args) {
3         // Display a greeting in the console window
4         System.out.println("Hello, World!");
5     }
6 }
```

例如，Java程序1.1 HelloPrinter.java的单词如表1.3。

表 1.3 程序HelloPrinter.java中的单词

单词	单词种类
public class static void	保留字
HelloPrinter main String args System out println	标识符
"Hello, World!"	常量
{ } ( ) [ ] . ;	特殊符

保留字是程序设计语言预先确定了含义的一些单词，在识别程序语法结构中起着关键作用，不能由程序员改变其用途。例如，看到程序中的关键字if就知道这是分支语句等。在有些程序设计语言中保留字也称为关键字(keywords)。

常量也称为**文字**(literal)，用来表示程序可以处理的一些基本数据类型的值，如整数、浮点数、字符、字符串等。不同类型的常量有不同形式的语法结构。例如，整数通常是数字串，但语言还可能

支持八进制、十六进制整数；浮点数可支持科学记数法，字符通常使用单引号分隔，字符串通常使用双引号分隔等。

标识符是程序中使用得最多的单词，通常由程序员定义其含义，用来命名变量、对象、类、类型、方法、模块等程序要素。在多数程序设计语言中，标识符是以字母开头的字母、数字串，但有些程序设计语言也支持其中使用一些数字、字母之外的字符，例如多数语言都允许使用下划线。Java语言支持标识符中出现美元符号，甚至Unicode中任意可见的字符（包括汉字、韩文、德文等）。

特殊符是程序中除保留字、常量、标识符和空白之外的单词，它们也可像空白一样用来分隔保留字、常量和标识符。通常特殊符有两大类：一类是运算符，用来在程序的表达式中表示最基本的数学运算，例如加号(+)、减号(-)等；一类是分隔符，用于将程序要素组织成某种语法结构，并与其他的程序要素分隔开来，例如分号通常用来分隔不同的程序语句，园括号用来组织表达式，或函数/方法的参数，花括号用来组织块语句等。

综上，从词法分析的角度看，程序是由空白符分隔的单词的序列，其中最重要的单词是标识符，它用于命名程序中的各种要素。我们将在后面章节对标识符，在那里我们称为**名字**(name)的各种属性做更深入的讨论。

程序语法分析树节点类型

对于程序分析而言，我们需要从更高的语法层次来看待程序的要素，这些程序要素是一些单词的组合，组成了更有意义的语法结构。程序中有哪些这样的语法结构，或说程序要素呢？我们可从语法分析的结果–语法分析树的节点类型来理解这些语法要素。

Eclipse的JDT(Java Development Tools)提供了一组API让软件开发人员操作Java语言源代码程序。Eclipse JDT实际上提供了对Java软件的一个模型：一个Java软件处在一个工作空间(work space)中，工作空间有多个项目(project)，一个项目有多个Java 文件，每个Java文件在该模型中称为一个编译单元(compilation unit)。Eclipse JDT提供了类`ASTNode` 描述Java程序的语法分析树的节点，每个Java文件通过语法分析之后可构造以一个称为编译单元（对应类`CompilatoinUnit`）的节点为根的语法分析树。

图1.1是程序`HelloPrinter.java`的语法分析树。图1.1的每一行给出了该语法分析树的一个节点，每一行最前面的数字给出了这个节点在程序源代码中的行号。例如，第一行是语法分析树的根，它是一个类型为`CompilationUnit`的节点，它只有一个儿子节点，该节点的类型是`TypeDeclaration`，即类型声明节点，等等。一个节点的儿子节点将相对于该节点缩进显示，兄弟节点在相同的缩进位置上。图1.1以一种简单的文本形式展示了程序`HelloPrinter.java`（程序1.1）的语法分析树。

我们可通过分析Eclipse JDT 所提供的语法分析树节点类型来理解Java程序的语法结构，即从语法角度看Java程序的要素，其他语言的程序虽然在细节上有一些差别，但只要是面向对象程序，包括C++, C#程序等，基本的程序要素没有太大的差别，至少我们可通过熟悉Java程序的语法分析树而更容易其他程序设计语言的程序的语法结构。

表1.4给出了类`ASTNode`的一些直接子类。当然它还有一些其他的直接子类，但是其他子类都与Java语言的标注(notation)有关，而标注机制在其他语言还比较少见，因此我们暂时不对它们进行分析。在标注机制标准化（或说称为现代语言密不可分的一部分）之前，程序分析还不能依赖于标注机制。当然表1.4 也给出了一些Java比较特有的东西，如匿名类，包声明和包导入等，但这些东西

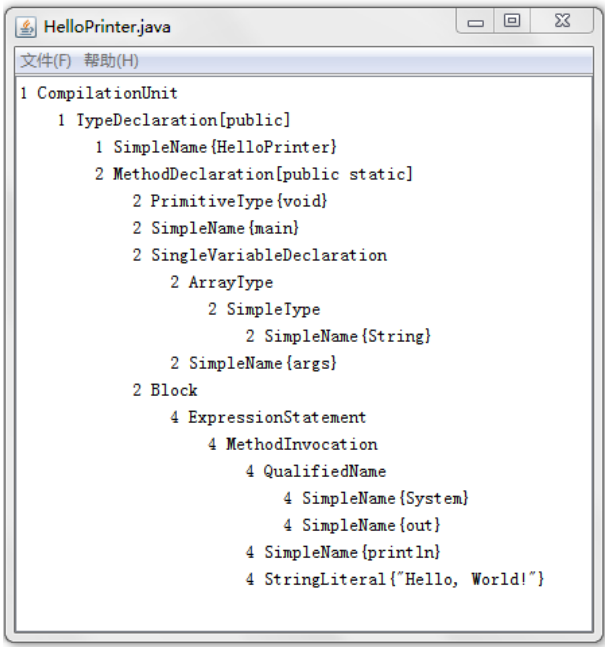


图 1.1 程序HelloPrinter.java的语法分析树

在其他语言也往往有对应物，例如C++语言的嵌套类、include机制以及名字空间等。

表 1.4 类ASTNode的直接子类

AST 节点类型	含义	有否子类
AnonymousClassDeclaration	匿名类声明，匿名类是Java程序特有的语法要素	否
BodyDeclaration	体声明，作为某种类型声明的体的语法要素	有
CatchClause	异常捕获声明	否
CompilationUnit	编译单元，这是语法分析树的根节点类型	否
Expression	表达式	有
ImportDeclaration	包导入声明	否
Modifier	成员修饰符，例如public等	否
PackageDeclaration	包声明	否
Statement	语句	有
Type	类型	有
TypeParameter	类属参数	否
VariableDeclaration	变量声明	有

表1.4给出的语法树节点类型中，体声明、表达式、语句、类型和变量声明还有子类，这也说明了它们是Java程序最重要的语法要素。表1.5给出了类BodyDeclaration的子类。从这个表可以看到，Java语言的体声明是指枚举、接口和类声明及可放在它们之中的一些重要程序语法要素。注意，我们这里给出的子类关系，而不是组成关系，即体声明这个类抽象了枚举声明、接口声明、类声明、枚举常量声明、属性域声明和方法声明的一些共性，并未说明这些节点之间的组成关系。我们将在给出所有的语法树节点类型之后，简单讨论一些复杂语法节点的构成。

表达式是Java程序中比较复杂的语法要素。表1.6给出了类Expression的所有子类，由于从类名

表 1.5 类BodyDeclaration的子类

AST 节点类型	含义
AbstractTypeDeclaration	TypeDeclaration和EnumDeclaration的父类
EnumDeclaration	枚举声明
TypeDeclaration	接口声明或类声明
EnumConstantDeclaration	枚举常量声明
FieldDeclaration	类或接口的属性域声明
Initializer	（不在任何方法中的）初始化语句块
MethodDeclaration	类或接口的方法（包括构造方法）的声明

比较容易理解其对应的表达式（不容易理解的将在后面解释），因此表中没有给出子类的含义，但我们将这些子类进行了简单的分类，以方便理解Java程序中表达式的种类。

表 1.6 类Expression的子类

表达式种类	AST 节点类型
名字	Name
文字	BooleanLiteral, CharacterLiteral, NumberLiteral, StringLiteral, NullLiteral, TypeLiteral
基本表达式	Assignment, CastExpression, ConditionExpression, InstanceofExpression, VariableDeclarationExpression
数组访问等	ArrayAccess, ArrayCreation, ArrayInitializer
成员访问等	ClassInstanceCreation, FieldAccess, MethodInvocation, SuperFieldAccess, SuperMethodInvocation, ThisExpression
表达式构造	InfixExpression, ParenthesizedExpression, PostfixExpression, PrefixExpression

在类Expression的子类中，类TypeLiteral表示形如String.class这样的表达式，这种表达式用于Java 的反射机制中，用来返回一个代表String的类对象（类Class的对象）。类CastExpression表示类型转换表达式，例如(int)3.0。类ConditionExpression表示C++和Java中特有的条件表达式，例如(a > b)?a:b。类VariableDeclarationExpression表示变量声明并进行了初始化的表达式，例如for语句中的初始化表达式，for (int i = 0; i < 10; i++)中的int i = 0。

表1.6将表达式分为名字、文字、基本表达式、数组访问、成员访问和表达式构造等几种类型。名字可以是类名（例如类型转换表达式中的类名）、对象名、变量名（基本数据类型的变量）、方法名（出现在访问调用中）等。由于Java语言的特殊性，类name有两个子类：类QualifiedName和类SimpleName，前者是包含点运算符(.)的限定名，而后者是简单的名字。例如从图1.1可看到，System.out是限定名，而println是简单名。

表1.6中的文字表示各种类型的常量，包括布尔常量、字符常量、数值常量、字符串常量、null以及类型文字。表1.6将赋值(Assignment)、类型转换、条件表达式、instanceof表达式、变量声明

表达式归于基本表达式，或者也可称为表达式杂类，实际上是Java程序中一些不太好归类的，但比较基本的表达式。数组访问实际上是与数组有关的表达式，包括数组访问（如a[i]）、数组创建（如new int[3]）和数组初始化（如用于声明数组并初始化的表达式{3，4，5}）。成员访问实际上是与类（或接口、枚举）的创建与属性域访问、方法调用相关的表达式。表达式构造是指利用圆括号、中缀、后缀、前缀运算符构造表达式。

语句是Java程序中另外一个比较复杂的程序语法要素。表1.7给出了类Statement的所有子类。表中也对这些子类进行了分类。

表 1.7 类Statement的子类

语句种类	AST 节点类型
基本语句	AssertStatement，ConstructorInvocation，EmptyStatement，ExpressionStatement，SuperConstructorInvocation，SynchronizeStatement，TypeDeclarationStatement，VariableDeclarationStatement
块语句	Block
分支语句	IfStatement，SwitchCase，SwitchStatement
循环语句	DoStatement，EnhanceForStatement，ForStatement，WhileStatement
控制转移	BreakStatement，ContinueStatement，LabelStatement，ReturnStatement
异常处理	ThrowStatement，TryStatement

表1.7的基本语句也是给出了Java程序一些难以归类但比较基本的语句种类，包括断言语句、构造方法调用、空语句、表达式语句、超类构造方法调用、同步语句、类型声明语句和变量声明语句等。断言语句是形如assert (i > 0)这样的语句。构造方法调用是形如this("hello")这样的语句，用于在某个类中调用该类的某个构造方法。类似地，超类构造方法调用是形如super("hello")这样的语句，用于在某个类中调用该类的超类的某个构造方法。类型声明语句这种语法分析树节点主要用于将类型声明包装成语句，实际上就代表类型声明。类似地，变量声明语句也是将变量声明（例如int i）包装成语句（例如int i;），实际上就是变量声明。

表1.7中的其他语句都是常见的语句，包括块语句、分支语句、循环语句，以及与控制转换相关的语句（break,continue,return和带标号语句），和与异常处理相关的语句（异常抛出(ThrowStatemnt)、异常捕获(TryStatement)）。注意SwitchCase表示Switch语句中的case分支，而异常捕获语句实际上由放在try的语句块以及一些CatchClause构成。

语法树节点Type表示对应的程序要素代表一个类型。表1.8给出类Type的所有子类。实际上，其中的QualifiedType和SimpleType引用的就是由程序员自己定义的类、接口和枚举名。在后文我们将进一步讨论程序的类型系统。

程序语法要素的结构

上面给出了Java程序语法分析树可能的节点类型，并详细讨论了一些种类比较多的节点，如Expression，Statement的子类，从而让我们对Java程序语法要素的类型有了比较全面的了解。下面我们则从组成结构上进一步讨论这些语法要素之间的关系。

表 1.8 类Type的子类

AST 节点类型	含义
ArrayType	数组类型，例如：int[]
ParameterizedType	类属化类型，例如：List<String>
PrimitiveType	原子类型，包括boolean，byte，char，int，short，long，double，float，void
QualifiedType	带限定的类型名，例如：java.lang.String
SimpleType	简单类型名，例如：String
WildcardType	用于类属机制的通配类型，例如：？ extend Student

前面已经提到，Eclipse JDT将一个Java文件看作一个**编译单元**(CompilationUnit)，因此该Java文件所对应的Java程序的语法分析树的根节点类型是CompilationUnit。

通常一个Java程序由**包声明**(PackageDeclaration)、**包导入声明**(ImportDeclaration)以及一个或多个**类/接口声明**(TypeDeclaration)或**枚举声明**(EnumDeclaration)构成。

枚举声明通常有一个或多个**枚举常量声明**(EnumConstantDeclaration)构成。枚举类型是Java 1.5之后引入的新机制，它也可像类一样有属性字段和方法，但通常应该只是声明一些枚举常量。枚举常量简单地来说就可看作是一些名字。

类或接口声明中的主要程序要素是**初始化块**(Initializer)、**方法声明**(MethodDeclaration)以及**属性字段声明**(FieldDeclaration)。构造方法的语法形式与普通方法基本相同。初始化块实际上就是一个块语句(Block)，只是它直接隶属于类或接口，而不是在某个方法中。

属性字段声明本质上与**变量声明**(VariableDeclaration)相同。类VariableDeclaration有两个子类：一是SingleVariableDeclaration；一是VariableDeclarationFragment。前者表示单个变量声明，后者是变量声明片段。实际上前者可看作是变量类型(Type)后跟一个变量声明片段。因此变量声明片段用于多个具有相同类型的变量的声明中。多数情况下，我们看到的语法分析树节点都将给出变量类型和变量声明片段这两种节点。变量声明片段由名字(Name)后跟一个初始化表达式(Expression)构成。

方法声明中的主要程序要素是**返回类型**(Type)、**方法名**(Name)、**参数声明列表**以及**方法体**。参数声明列表是由逗号分隔的多个**参数声明**，而每个参数声明本质上就是**单个变量声明**(SingleVariableDeclaration)。方法体实际上一个块语句(Block)，而块语句是由花括号分隔的零个或多个语句(Statement)构成。

表1.9给出了Java程序的基本语法结构，从编译单元开始，给出每种语法要素的主要组成成分。当然，这里并不是要严格讨论Java程序的语法，因此只是给出一些主要的组成成分让我们对Java程序的语法结构有一个简单的认识，像修饰符（诸如static，public）等一些不太重要的组成成分没有提到。

1.1.4 数据抽象、控制抽象和程序要素

上面从词法与语法的角度讨论了程序的语法要素，由于是语法要素因此也主要是物理的（程序本身的角度）和静态的（与程序运行无关）的角度讨论了程序的要素。这一节我们从程序数据抽象（主要对应现实世界的实体）以及控制抽象（主要对应现实世界的过程）来讨论程序的要素，因此

表 1.9 Java程序的基本语法结构

AST 节点类型	主要组成成分
编译单元(CompilationUnit)	包声明(PackageDeclaration) 包导入声明(ImportDeclaration) 类声明(TypeDeclaration) 接口声明(TypeDeclaration) 枚举声明(EnumDeclaration)
枚举声明(EnumDeclaration)	枚举常量声明(EnumConstantDeclaration)
类或接口声明(TypeDeclaration)	初始化块(Initializer) 属性字段声明(FieldDeclaration 方法声明(MethodDeclaration)
属性字段声明(FieldDeclaration)	变量声明(VariableDeclaration)
方法声明(MethodDelcaration)	返回类型(Type) 参数声明(SingleVariableDeclaration) 方法体(Block)

也可以说主要是从语义的角度，并兼顾静态和动态的角度对程序要素展开讨论。

数据和控制这两个概念的含义本身也很广泛，不过我们在这里并是要讨论这两个概念的含义，也不是讨论什么是数据抽象与控制抽象，而是从数据抽象和控制抽象这两个角度去讨论程序要素，所以我们只需在直观理解的基础上明确一下如何从两个角度对程序要素进行归类即可。

直观地，我们将具有如下特性的程序要素归为从数据抽象角度进行理解与讨论的一类：(1) 从逻辑层面看，是值及对值的分类，我们用这些值去度量或刻划现实世界的实体或实体在某个时刻的状态；(2) 从物理层面看，是程序要处理和操作的目标；(3) 从静态和动态角度看，是可以静态存在的，在每个时刻点都有它的含义（即可建立与现实世界事物的映射），而且可以在程序动态运行过程中被改变。

对应地，我们将具有如下特性的程序要素归为从控制抽象角度进行理解与讨论的一类：(1) 从逻辑层面看，是操作及操作的组合，我们用这些操作去描述现实世界的任务完成的步骤与顺序；(2) 从物理层面看，是对程序中的数据进行操作以及操作顺序的描述；(3) 从静态和动态角度看，需要从时间的延续中理解它的含义，在某个时刻点是没有意义的，静态的描述只是辅助人们想象与理解它的动态含义。

按照这样的标准，表1.10给出了程序`HelloPrinter.java`（程序1.1）在数据抽象与控制抽象这一层次我们所关心的的程序要素及其归类：

简单地说，(1) 在这个层次我们只关系程序员使用名字所命名的程序要素，即标识符（或说名字）所对应的程序要素；(2) 我们将类、类型、对象、原子类型的变量（我们后面简称为**原子变量**）以及常量归为从数据抽象进行讨论的程序要素；(3) 我们将方法、运算符、表达式、控制结构、语句等归为从控制抽象进行讨论的程序要素。

程序的数据抽象

我们将用于刻划实体或实体状态的值及其分类归为从数据抽象角度讨论的程序实体。实际上，

表 1.10 程序HelloPrinter.java中的数据抽象与控制抽象要素

类别	程序要素	备注
数据抽象	HelloPrinter	类名，对现实世界实体进行分类的抽象
	String[]	类型名，对现实世界实体及其结构进行分类的抽象
	args	参数名，现实世界实体的抽象
	System	类名，对现实世界实体进行分类的抽象
	out	对象名，现实世界实体的抽象
	"Hello, World!"	数据值，现实世界实体状态的抽象
控制抽象	main	主方法，程序运行的起点
	println	打印方法，现实世界中操作的抽象

我们如何刻划实体或实体状态呢？我们是利用值及值之间的结构关系来进行刻划。不同的实体首先是具有不同的结构，其次是它能完成不同的功能。因此从数据抽象讨论程序要素，我们主要从两个方面进行：一是考察这些程序要素的**分类层次**；一是考察这些程序要素的**组成结构**。当然，这两个方面并不是孤立的，组成结构的不同通常是划分成不同类的最重要的标准之一。

从程序语法的角度看，常量、原子变量、对象、类型和类（以及与类相似的接口、枚举、结构、联合体等）属于程序的数据抽象要素，注意，类、接口、枚举、结构、联合体本身也是类型。从分类层次角度看，有两个层次，一是**对值进行分类**，即将常量、原子变量、对象归为不同的类型，常量、原子变量、对象称为某个类型的实例；一是**类型之间的层次关系**，即一种类型可看作另外一种类型的子类型，或者说一种类型的实例都是另外一种类型的实例。

常量、原子变量、对象与类型最大的区别在于它们在程序运行过程中必然是存在的，而且除常量外，原子变量的值，对象的状态随着程序的运行而变化。类型在程序运行过程中通常是不存在的，类型主要用于在程序运行之前数据之前是否匹配的检查，以避免运行时的错误。程序在运行过程中通常不能对类型本身进行操作，例如不能撤销、创建、修改某个类型，但是可以撤销、创建、修改某个对象或原子变量。

所以，对于常量、原子变量和对象我们主要关心它们的动态特性，例如它们的作用域或说在什么时候可使用，它们的生存期（什么时候创建，什么时候撤销），它们的存储表示和布局等，而对于类型我们主要关心它们的静态特性，例如各个类型与类型之间的关系（两个类型是否匹配，两个类型之间能否转换，一个类型是否是另外一个类型的子类型等）。我们在后文中分别就名字（主要是用于命名原子变量和对象的名字）与类型系统展开详细的讨论。

从数据抽象要素的组成结构角度看，首先这些要素可分为两类：一是**原子型**(primitive)，包括原子类型及其实例（原子变量或原子常量）；一是**复合型**(composite)，包括类、接口、枚举、结构、联合体、数组、指针、引用等及其实例（对象或说变量，以及常量）等。

原子类型、原子变量和原子常量是数据抽象的基本单位，不可再分解为更基本的单位。通常程序中的原子类型包括布尔类型(`boolean`)、字符类型(`char`)、各种整数类型（短整数(`short`)、整数(`int`)、长整数(`long`)等）、各种实数类型（浮点数(`float`)、双精度数(`double`)等）。

程序的复合型数据抽象要素的特点是能够在结构上进行分解，它由其它的原子型或复合型程序程序数据抽象要素组成，复合型要素是对它的组成要素的一个总体上的命名。实际上，这也是所谓的数据抽象的真实含义，即在某个层次上，我们可只关心复合型数据抽象要素的整体性质（整体上

的存储表示，有什么可用的操作以及满足怎样的约束等），而忽视它的组成要素（即不关心有哪些组成成分，这些成分的存储表示、可用的操作及应该满足的约束等）。

例如，我们可在整体上关注“学生”这样的类提供怎样的操作，在存储中的表示，以及满足的约束，而无需关心学生类中的细节，例如它有哪些组成成分（是否一定有学号、姓名、年龄、出生日期？），这些组成成分的存储表示等等。

程序的复合型数据抽象要素还可根据组成成分与整体的耦合性进一步分为两类：一类具有紧密耦合性，组成成分与整体是紧密耦合的，组成成分脱离整体不能单独存在，或者单独存在时具有与整体完全无关的含义；一类具有松散耦合性，组成成分与整体是松散耦合的，组成成分可脱离整体单独存在，整体是多个组成成分的组合，单独存在的组成成分与只有一个组成成分的整体没有本质的差别。

简单地说，结构或类这种复合型数据抽象其组成成分与整体是紧密耦合的，而数组这种复合型数据抽象其组成成份与整体是松散耦合的。结构或类的组成成份（即其字段）是不能脱离整体单独存在的，例如，“名字”不能脱离“学生”单独存在，单独存在的“名字串”与“学生”这个整体没有关系。而数组成份与整个数组是松散耦合的，每个数组成份可以单独存在，单独存在的数组成份可看作长度为1的数组。

从分类的角度看，紧密耦合性的复合型数据将组成成份组合在一起之后形成了一个全新的类型，与单个组成成份的类型没有关系，而松散耦合性的复合型数据将组成成份组合在一起虽然也形成了一个新的类型，但是以原类型为基类型的类型，新类型可用的操作与原类型紧密相关。

从实现（物理层）的角度，紧密耦合性的复合型数据抽象通常完全由用户定义它的组成成份，因此也称为**用户自定义类型**(user defined type)，这包括类(`class`)、接口(`interface`)、结构(`struct`)、联合体(`union`)、枚举(`enum`)等；而松散耦合性的复合型数据抽象通常是程序设计语言提供的类型定义机制，从基类型派生出新的类型，因此也称为**派生类型**(derived type)，这包括数组(`array`)、指针(`pointer`)以及引用(`reference`)等。

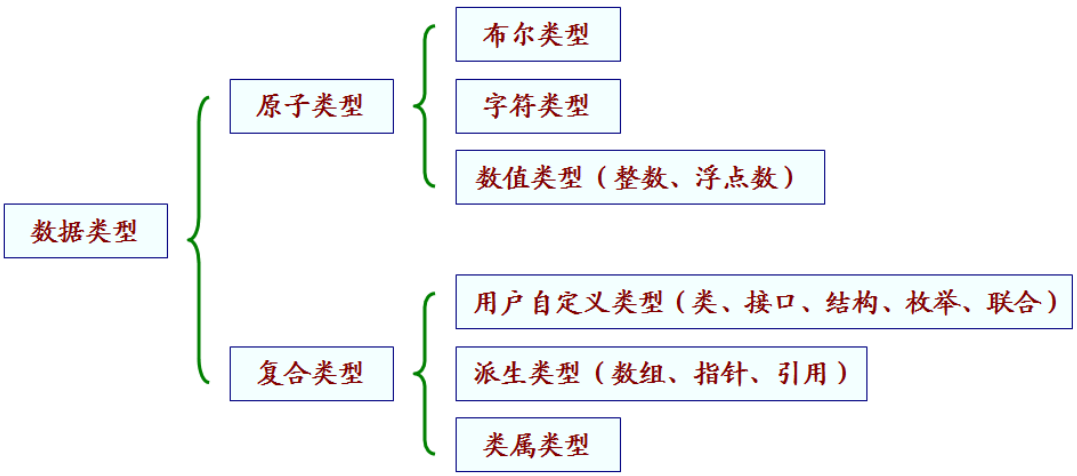


图 1.2 程序设计语言的数据类型基本架构

图1.2给出了多数程序设计语言中数据类型的基本架构，其中的类属类型实际上也是程序设计语言提供的从基类型派生新类型的一种机制，因此也是一种松散耦合性的派生类型机制，不过由于它比较新，而且有更复杂的特性，因此我们将其单列。例如，对于任意类型`T`，`List<T>`给出了以`T`类

型为基类型的链表类型。在理论上，类属类（即带有类型参数的类）List本身已经不是类型，而是用来构造类型的类型算子(type operator)，只有List<T>才是类型（才能用来声明变量）。实际上，数组([])、指针(*)、引用(&)也是类型算子。

对于复合型数据抽象要素，我们还需要了解它的**成员访问**机制，即如何从一个复合型数据抽象要素的名字访问到它的成员。通常用户自定义类型和派生类型的成员访问机制不相同。程序设计语言通常提供成员访问运算符（例如Java语言的.，以及C++语言中额外的->）访问用户自定义类型变量的成员。对于派生类型，程序设计语言通常提供一种遍历机制，例如数组的随机访问、指针的加减运算等。

复合类型的成员访问可能会是非常复杂的，因为它的成员结构可能是非常复杂的。我们可将复合类型的成员分为**直接成员**和**间接成员**两种。直接成员则是直接声明在该复合类型之中的成员，而间接成员则是声明在该复合类型的某个直接或间接成员中的成员。如果复合类型的某个直接或间接成员的类型是该复合类型本身，则该复合类型是一种**递归类型**(recursive type)。递归类型是实现递归数据结构（如链表、树）等的基本方法。

程序的控制抽象

命令式(imperative)程序设计语言程序的核心是描述计算机如何完成任务，或从计算机世界的角度说，是描述如何使用程序语句改变变量的值，从而将输入数据转换输出数据。因此计算与改变变量的值是命令式程序的基础。计算是要根据变量已有的值计算新的值，这在程序中通过**表达式**来实现，而改变变量的值则通过**赋值语句**实现。因此也可以说，表达式和赋值是命令式程序中最基本的控制抽象。

前面也看到，表达式是Java程序（C++程序也类似）中最复杂的语法要素之一。从概念上说，表达式由**操作数**(operand)与**运算符**(operator)组成。可以看作基本操作数的程序语法要素包括表1.6中的名字（变量名）、所有的文字（常量）、所有的成员访问表达式、以及数组访问(ArrayAccess)等。表达式本身是递归定义的，因为一个表达式也可看作（另一个）表达式的操作数。

运算符提供了对数据的最基本的操作，通常程序设计语言都会提供**算术运算符**(arithmetic)、**关系运算符**(relational)和**逻辑运算符**(logical)，Java和C++语言还提供了**按位运算符**(bitwise)。算术运算符包括通常的算术运算，如+，-，*，/，%等，通常还包括移位运算符，Java和C++语言还提供了著名的增量和减量运算符(++，--)。关系运算符用于比较两个操作数，通常包括大于、等于、小于、大于等于、小于等于、以及不等于等比较。逻辑运算符包括与、或、非，通常用来与关系运算符一起表示复杂的逻辑条件表达式。按位运算符严格来说也算是算术运算符，只是将操作数看作二进制位，从而进行按位非、按位与和按位或的操作。因此简单来说，使用运算符的表达式的结果通常是数值型数据或逻辑型数据（真或假），字符型数据通常不参与这种运算，或者参与也是被看作数值型数据。

对于表达式，我们需要关注表达式是如何计算的以及表达式计算结果的数据类型。表达式的计算方式取决于运算符的优先级与结合性。程序员在书写表达式，由于省略园括号而使得一个操作数可能成为两个运算符的操作数时，例如a+b*c中的操作数既可能是运算符+的操作数，也可能是运算符*的操作数。运算符的结合性，以及它们之间的优先级决定了在这种情况下，这样的操作数到底是哪个运算符的操作数，例如由于*的优先级通常比+高，所以a+b*c中的b应该是*的操作数，即优先级高的运算符的操作数。如果两个运算符的优先级相同，则由运算符的结合性决定，运算符是从左至

右结合，则属于左边的运算符，若是从右至左结合，则属于右边的运算符。

表达式计算结果的数据类型取决于运算符以及参与运算的操作数的类型。显然，关系运算符和逻辑运算符的计算结果是逻辑型的数据，而算术运算符的计算结果取决于操作数的类型，如果操作数的类型都相同，则计算结果与操作数的类型相同。如果操作数的类型不尽相同，则操作数会发生隐式类型转换，这种类型转换的基本思想在于不丢失数据，因而总是将取值范围窄的数据类型的数据为取值范围宽的数据类型，最后表达式的结果将是所有操作数的类型中取值范围最宽的数据类型。

赋值语句是命令式程序的核心，因为它是改变变量值的最基本方法。能够放在赋值语句左边的表达式称为**左值表达式**(left value expression)，相应地，放在赋值语句右边的表达式称为**右值表达式**。所有的表达式都可作为右值表达式，但并不是所有表达式都能够作为左值表达式，例如`a+b`就不能作为左值表达式。简单地说，在Java语言中，只有变量名字(`name`)、属性字段访问(`FieldAccess`和`SuperFieldAccess`)以及数组访问(`ArrayAccess`)才能作为左值表达式。

众所周知，**顺序**(sequence)、**分支**(branch)和**循环**(loop)是描述语句执行顺序的三种基本控制结构。人们早就证明了，只要有这三种控制结构便能描述计算机能完成的任何任务。因此所有的命令式程序设计语言都能表示这三种基本控制结构。程序语句隐含地就是顺序执行，而对于分支和循环则提供了各种表示分支和循环的语句。分支语句一般有`if`语句表示两个分支，而`switch`或`case`语句表示多个分支语句。循环语句一般有`while`语句、`do...while`语句或`repeat...until`语句，以及`for`语句。

目前几乎所有的命令式程序设计语言在表示这三种基本控制结构的时候都是采用单入口，即除了循环语句的第一条语句可能有多个前导语句之外，其他语句，包括分支语句都只有唯一的前导语句。我们称语句S_1是语句S_2的前导，当且存在程序的某次运行是在执行语句S_1之后紧接着执行S_2。这一点将在后面讨论程序的控制流图(control flow graph, CFG)时可以看得更清楚。简单地，说语句（或语句块）是单入口的，如果该语句（语句块）所对应的CFG节点的入度为1。由于分支和循环语句中存在条件表达式的计算，而且要根据其计算结果决定下一语句的执行，因此我们在讨论程序执行顺序时不能以语句为基本单位，而是需要两种基本单位：一种是只有单入口和单出口的一条语句或多条语句，称为**基本块**(basic block)；一种是条件表达式（或称谓词）的计算。我们在后面讨论CFG时还要详细讨论程序执行顺序的描述。

注意，程序执行的基本块与块语句(block statement)并不相同，基本块是在程序执行时只有单入口和单出口的一条或多条语句，是一个动态的概念，而块语句则用于在语法上将多条语句组合成一条语句，从而简化语句结构的语法描述，是一个静态的，语法上的概念。

在语句层次之上的控制抽象就是**方法**(method)（或**过程**(procedure)、**函数**(function)、**模块**(module)）。方法可看作是被命名了的块语句，即将多条语句组合在一起，并且赋予一个名字，作为方法名。方法也可作一种表达式，方法名是运算符，而方法的参数是其操作数。因此，方法不仅是有名字的块语句，而且拥有参数，从而具有更大的灵活性。作为一种有名块语句，它划分了名字的可见区域，从而使得名字的**作用域**变得十分重要。作为一种表达式，它所处理的参数在运行时是可动态变化的，从而我们需要理解方法的**参数传递**方法。后文将对方法的参数传递做进一步的讨论。

在现代几乎所有的高级语言中，方法（或过程、函数、模块(sub-module)）被看作是组织程序的基本单位。实际上，方法才是真正意义上的、最基本的控制抽象，即将用程序语句描述的程序执行顺

序的细节进行屏蔽，而只是使用方法名对程序所完成的基本任务进行抽象。结构化的命令式程序设计语言的程序从某种意义上来说就是由一个或多个方法（或过程）组成。而对于面向对象程序设计语言，在方法之上，还有**类**作为组织程序的单位。类是现实世界实体的抽象，不仅用属性域字段描述实体的结构，而且使用方法描述实体可完成的基本任务（操作）。对于Java语言，在类之上还有**程序包**(package)作为组织程序的单位。也就是说，程序包、类和方法给出了Java程序在描述如何完成任务的控制结构方面的模块架构。

程序的这种模块架构是一种逻辑架构，因为它对应着现实世界任务的控制结构。对于任何的程序设计语言，我们都需要理解这种语言所提供的这种模块架构，例如，对于Java语言是程序包、类和方法，而对于C++语言是类和函数等。我们需要了解这种模块架构的基本组织单位，这些组织单位之间组成关系（例如程序包有多个类，类有多个方法），同一种组织单位之间的关系（例如程序包与子程序包、类与子类、类之间的嵌套等），以及每种组织单位对外界提供怎样的接口。

程序的这种模块架构还往往对应着程序的物理组织，即程序源代码文件在操作系统中的文件组织方式。例如，Java语言的程序包对应着组织程序源文件的目录，程序包与子程序包主要是对应目录与子目录之间的关系，在名字的作用域上并不形成嵌套关系。而在C++语言中，没有对应目录的模块组织单位，其基本的物理组织单位是文件。虽然Java程序的一个文件中可以有多个类，但每个类在编译成字节码后都会对应一个物理文件，而C++语言一个文件编译后对应一个目标代码文件。

1.1.5 程序基本构成小结

这一节我们首先从逻辑层面讨论了程序的基本模型，即程序是现实世界中为完成某项任务的实体和过程在计算机中的表示。然后我们指出可以从逻辑/物理、语法/语义、数据/控制以及静态/动态等几个角度讨论程序的要素，即可从不同的角度讨论程序的组成成份及其性质。

在这一节我们首先从词法的层面讨论了程序的组成成份，即程序是由空白符分隔的单词序列。空白符包括空格、制表符、换行以及注释，而单词可分为保留字、标识符（名字）、文字（常量）以及特殊符。特殊符包含运算符和分隔符两种。

进一步，我们从语法的层面讨论了程序的组成成份，以Java语言为例，给出了Eclipse JDT中语法分析树(AST)节点类型，这些节点类型给出了Java程序的语法要素，其中复杂的语法分析树节点包括表达式、语句、类型等。

从语法组成结构角度看，一个Java程序是一个编译单元，而编译单元中有程序包声明、程序包导入声明以及多个类（接口或枚举）声明。类（接口或枚举）声明中包括属性域字段声明和方法声明，属性域声明实际上是变量声明，方法声明包括返回类型、参数列表以及方法体，方法体是一个块语句，其中有多条语句。

在这一节我们还从数据抽象和控制抽象的角度讨论程序的要素，其中数据抽象有两个层次，一个层次将变量（对象）归类为类型，一个层次是类型之间的层次关系。

通常的程序设计语言都提供原子类型和复合类型，原子类型包括布尔类型、字符类型和数值类型等，而复合类型又可分为用户自定义类型和派生类型，用户自定义类型包括类、接口、枚举、结构、联合等，而派生类型包括数组、指针、引用以及类属类型。

程序控制抽象的基本要素是运算符和表达式，它们可描述如何计算数据值，而赋值语句是命令式程序的核心，它描述如何改变变量的值。三种基本控制结构–顺序、分支和循环可描述计算机能完成的任务的执行顺序，对应地程序会提供分支语句、循环语句来表示这些结构。程序使用方法（过

程、函数或模块）来从控制结构角度组织程序，这些是程序模块架构的基本组织单位，在面向对象程序中，类也是组织程序的一种模块层次，在Java语言中还使用程序包来组织程序。方法、类和程序包构成了Java程序的基本模块组织架构。

在这一章的剩下章节我们还需要对程序要素的三个问题做更深入的讨论：

1. 名字。从语法角度看名字是最重要的语法要素，它可对类型（类、接口、枚举）、方法、对象（变量）进行命名。与名字相关的问题包括：(1) 名字的定义和使用；(2) 变量（对象）的值、地址与类型；(3) 变量（对象）的作用域；(4) 变量（对象）的生存期。

2. 类型系统。类型系统给出了程序对所要处理的数据的基本抽象。我们需要进一步讨论：(1) 类型系统与类型表达式；(2) 类型与子类型；(3) 类型匹配与类型转换。

3. 程序的模块架构。模块架构给出程序从抽象控制结构的角度如何组织程序。我们需要进一步讨论：(1) 程序模块的逻辑架构与物理组织；(2) 参数传递；(3) 模块的调用与运行环境。

1.2 名字

名字(name)是程序中最重要的语法要素。我们可以将名字分为三种类别：**变量名字**、**类型名字**和**方法名字**。变量名字可能是在程序中出现频率最高的名字，它是程序运行过程中某个存储数据的抽象。类型名字是用户自定义的类型的名字，在Java语言中包括类名、接口名、枚举名以及类属参数名。类型用于程序编译时的类型检查，类型名字在程序运行过程中通常作为变量名字的属性存在。

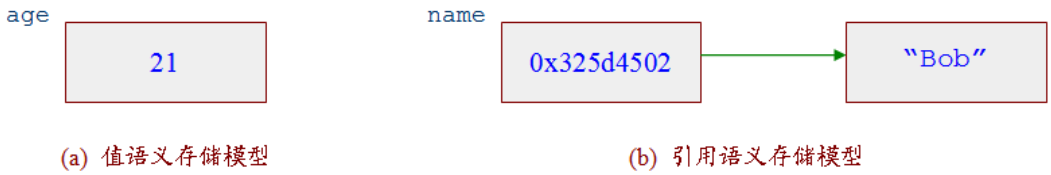
不同的名字有不同的属性，确定名字的某个属性的取值的过程称为名字与该属性之间的**绑定**(binding)。名字与属性的绑定可能是静态的，即在程序运行之前确定属性的取值，也可能是动态的，即在程序运行之后确定属性的取值。实际上，这一节就是要讨论不同名字的各种属性及其绑定方式。

1.2.1 名字的属性

为更好地理解名字，我们先总结一下各种名字可能的属性。对于变量名字，我们首先将其分为**原子变量名**(primitive variable name)和**对象名**(object name)。原子变量名是指其类型为原子类型的变量的名字，对象名是指其类型为复合类型的变量的名字。在这里我们只所以要采用这样的区分，只是因为在Java语言中这两类变量名字采用不同的语义模型。原子变量名采用的是**值语义**(value semantic)，而对象名采用的是**引用语义**(reference semantic)。

所谓的值语义就是名字所对应的存储区域存储的就是变量的值，而引用语义就是名字所对应的存储区域存储的是另外一片存储区域的引用（地址），另外一片存储区域存储的才是程序员真正想要访问的值。例如，在Java程序中，变量声明`int age = 21`与`String name = "Bob"`的存储模型如图1.3所示。

因此对于原子变量名，我们关心的属性是它的**类型**(type)、**值**(value)、**存储地址**(address)、**作用域**(scope)和**生存期**(lifetime)。而对于对象名，我们关心的属性是它**引用的对象的类型**、**它的值**，也即它**引用的对象的地址**、它**自己的地址**、它**自己的作用域**、它**自己的生存期**，以及它**引用的对象的生存期**。总而言之，对于变量名，我们都需要关心它的类型、值、存储地址、作用域和生存期这五个属性，而对于采用引用语义存储模型的变量名（Java语言的对象名，以及C++语言中的指针、引用



(a) 值语义存储模型　　(b) 引用语义存储模型

图 1.3 Java程序的变量存储模型

等）实际上我们要关注两个变量（或准确地，两片存储区域），一是该变量名本身所抽象的存储区域，一是该变量所指向的存储区域，这个存储区域可抽象为一个匿名变量。

类型名字都是用来命名程序员自定义的复合类型，在Java语言中，包括类(class)、接口(interface)与枚举(enum)，在其他语言中可能还有记录(record)、结构(struct)、联合(union)等。类型名字也有其作用域，即可以使用的范围，但通常类型名字的作用域要比变量名字的作用域简单很多。类型名字通常没有生存期，因此类型本身通常不会在程序运行过程中发生变化。对于类型名字，我们更关注的是它的**成员**(member)，因为类型名通常命名的是复合类型。在面向对象程序中，我们还可能需要关注类型之间的**子类型关系**，例如在Java语言中，一个类继承了哪个类，以及实现了哪些接口。

方法名，或者其他语言中的函数名、过程名或子程序名也有作用域，但通常也比变量名要简单。同样地，我们也不关心方法的生存期，因为方法本身在程序运行过程中通常不会被改变。方法名也有对应的地址，即调用该方法时的入口地址，或者说方法的第一条语句的存储地址，但程序员通常使用方法名来调用方法，对于方法具体的存储地址不关心，而且该地址通常在程序运行过程中也不会改变。因此，对于方法，我们主要关注的是该方法的**返回类型**(return type)、**参数列表**(argument list)以及**方法体**(method body)，其中参数列表的属性包括**参数个数**，以及每个位置上的**参数类型**。

表 1.11 名字的属性

名字种类	属性
原子变量名	类型、值、地址、作用域、生存期
对象名	本身的地址、值（所引用对象的地址）、作用域、生存期 所引用对象的类型、所引用对象的生存期
类型名	成员、作用域
方法名	返回类型、参数类表、方法体、作用域

表1.11总结了我们所关注的名字属性，下面对这些名字及其属性，以及名字与属性之间的绑定作进一步的讨论。

1.2.2 名字的定义、声明和使用

在多数程序设计语言中，名字必须先**声明**(declaration)或**定义**(definition)，然后才能**使用**(use)。所谓的声明或定义实际上建立名字与其属性之间的绑定，而使用则是根据这种绑定去获取名字的某种属性。声明和定义的差别取决于名字与哪些属性绑定，通常名字的定义要比名字的声明绑定更多

的属性。Java语言只有名字的定义，而C++语言则区别了名字的声明与定义。下面分不同的名字种类对名字的定义、声明和使用做进一步的讨论。

对于变量名（包括原子变量名好对象名），最关键的属性是类型和（存储地址），因此如果一个语句在运行时会建立变量名与地址之间的绑定，则该语句是该变量名的定义。通常，在静态类型声明的语言，例如C++和Java等，该语句也会建立变量名与类型的绑定。在C或C++语言中，还区分变量的声明与定义，变量声明通常只是建立变量名与类型的绑定，从而供编译器检查变量的使用是否符合类型匹配的要求，变量声明语句通常不会被编译成可执行语句，因此也不建立变量名与地址之间的绑定。变量声明同时确定了该变量名的作用域，而变量定义则确定了该变量的所有属性，包括类型、地址、(初始）值、作用域与生存期。

除变量声明或定义之外的变量名出现都是变量的使用。变量的使用实际上就是使用变量及其属性的绑定。编译器要对变量的每次使用根据作用域规则确定该变量使用是使用何处定义（或声明）的变量，从而获得与变量绑定的属性。通常编译器要根据变量的类型检查该变量使用是否符合类型匹配规则。对于程序员而言，变量使用可分为**左值**(left value)使用和**右值**(right value)。所谓左值使用就是使用变量的地址，而所谓右值使用就是使用变量的值。从形式上看，左值使用是指变量名可以出现在赋值运算符左边的使用，而右值使用是指变量名可以放在赋值运算符右边的使用。

类型名的定义是指定义该类型的成员。在区分类型定义和声明的语言（例如C++语言）中，类型的声明则是指仅仅给出类型的类别（类、结构或联合体），而不给出类型的成员。当然原子类型(primitive type)名通常是语言的关键字，由语言设计者预先定义。类型名的使用主要有两种，一是用于定义或声明变量（或参数）；一是用于类型转换表达式。在面向对象语言中，类名还可能用于访问类的静态成员。

方法名（或一般地，子程序名）的定义是指给出该方法的方法体，当然也包括方法的返回类型、参数列表，而方法的声明则只是给出方法的返回类型、参数列表，不给出方法的方法体。方法名的使用则是指使用实际参数调用该方法。

表1.12给出了程序`HelloPrinter.java`中的名字的定义和使用情况。

表 1.12 程序HelloPrinter.java中名字的定义和使用

名字	类别	名字的定义还是使用?
HelloPrinter	类型名	类型HelloPrinter的定义
String	类型名	使用，String是Java可重用构件库预先定义的类型名
args	变量名	定义，定义字符串数组类型的参数
System	类型名	使用，System是Java可重用构件库预先定义的类型名
out	变量名	使用，out是Java可重用构件库预先定义的对象名
main	方法名	定义
println	方法名	使用，println是Java可重用构件库预先定义的方法名

1.2.3 变量的类型、值与地址

像Java, C++等语言是强类型语言，在这种语言中，程序员必须使用变量声明或定义语句声明变量的类型，以便编译器根据类型匹配规则对程序进行语义检查。也有一些弱类型语言，例

如Python等一些脚本型语言，程序员无需声明变量的类型。通常这些语言都是解释执行的，解释器在运行程序时根据变量所存储的值确定变量的类型，并进行类型匹配检查。

类型本质上来说是值的集合，或者说是对程序能处理的各种数据的一个划分，因此变量的类型决定变量的取值范围，即变量只能取属于该类型的值。类型的取值范围也与该类型的值在内存中的表示，或说编码方式密切相关。变量的类型还有一个更重要的作用是确定变量可用的运算（操作），例如不能将字符类型的值用于加减乘除等算术运算。不同类型的变量适用不同的运算是程序设计语言中类型匹配规则的重要组成部分。因此，简单地说，变量类型的作用是确定变量的**取值范围**、**内存表示**与**可用操作**。

变量实际上存储区域的抽象，变量的值是该存储区域所存放的数据的值，变量的地址通常是指该存储区域在计算机硬件或操作系统的某种地址编码方式中的编码。程序员通常不关心变量地址的具体值。程序员之所以对变量的地址属性感兴趣是因为有些变量的值本身是另外某个变量的地址。在C++语言中，引用变量或指针变量本身的值就是另外某个变量的地址，而在Java语言中，所有不是原子变量的变量都是另外某个变量的地址。

若变量`ptr`的值是变量`obj`的地址，我们称变量`ptr`**指向**(point to)变量`obj`。这种指向关系是程序变量之间最重要的关系之一，也是程序分析的核心内容之一。当变量`ptr`指向变量`obj`时，程序员往往更关心变量`obj`的类型，或者说更关心`ptr`能指向哪些类型的变量。通常`ptr`的类型是指针类型或引用类型，而`obj`的类型可以是任意类型，并称为`ptr`的**基类型**(base type)。通常程序员在声明变量`ptr`时，指定它的基类型。在面向对象程序中，变量`ptr`能指向类型是`ptr`的基类型或者是`ptr`的基类型的子类型的变量，这是面向对象程序具有多态性的重要基础。

变量之间的指向关系是一种动态关系，即在程序运行过程中建立和变化。一个变量在程序运行过程中可能指向不同的变量，而且在某个运行时间段，某个变量也可能被多个变量指向，这时这多个变量互为**别名**(alias)。别名现象是导致程序容易出错的主要原因之一。变量指向分析(point analysis)的主要目的之一就是要找出所有可能的别名。在Java语言中，被指向的变量（对象）都是匿名变量，因此只有（引用）变量之间发生别名，而在C++语言中，被指向的变量也可能具有变量名，这时被指向变量的变量名与指向该变量的指针（或引用）变量名也发生别名现象。

1.2.4 名字的作用域

作用域是程序源代码的一段区域，在这个区域里某些名字可以被使用，而在这个区域之外这些名字则不可被使用，因此这个区域称为这些名字的作用域。或者从另外一个角度看，作用域用来区分某个名字的使用到底是使用在哪里声明或定义的名字。

在绝大多数语言中，作用域是静态建立的，即在程序运行之前可以确定每个名字的作用域。**静态作用域**(static scoping)也称为**词法作用域**(lexical scoping)，因为根据程序的词法形式可确定每个名字的作用域。极少数语言，像早期的Lisp，以及目前比较常用的脚本语言Perl，采用**动态作用域**(dynamic scoping)，即在程序运行时才能确定某些名字的作用域，或者说，一个名字的使用在不同的子程序调用中可能使用在不同地方定义的（同名）变量。动态作用域使得程序更加难以理解，因此在绝大多数语言中使用静态作用域，我们这里也只考虑静态作用域。

Java语言有几种方式定义作用域：

(1) **包作用域**：通过包声明语句可声明一个Java源文件属于某个包，一个包可包含多个Java源文件。包的声明将Java程序划分成不同的区域，每个区域对应一个包作用域。

(2) **类作用域**：类（或接口、枚举）的声明也将Java程序划分成不同的区域，每个类（或接口、枚举）对应一个类作用域。

(3) **块作用域**：Java程序中花括号界定的方法体或块语句将程序划分为不同的块，每个块对应一个块作用域。对于方法体对应的块作用域从方法参数声明算起，而对于`for`语句对应的块作用域从其初始化语句算起。块作用域也称为**局部作用域**。

因此，Java语言的作用域规则可以说比较简单，只有包作用域、类作用域和块作用域三种。这些作用域是可以嵌套的，即类可定义在某个Java源文件，也即某个包作用域（或者说缺省作用域）中，还可定义在另外一个类中形成嵌套类，或定义在一个块中形成局部类，而块作用域定义在类或另外一个块中。一个作用域构成了某些名字的名字空间，而作用域及其之间的嵌套构成了名字空间的层次化组织。作用域机制使得程序员在不同的名字空间定义相同的名字，从而在不影响程序可理解性的前提下减轻程序员对标识符的命名工作。

在Java程序中要确定一个名字的作用域也比较简单。定义在任何类作用域或块作用域外的类名（接口名、枚举名）具有包作用域，而定义在某个类（接口、枚举）之内，但在任何块作用域之外的类名（接口名、枚举名）具有类作用域，定义在某个方法之类的类名（接口名、枚举名）具有局部作用域。定义在类（接口、枚举）之内，但在任何块作用域之外的方法名、变量名具有类作用域，也是这个类（接口、枚举）的成员。定义在块作用域类的变量名（包括方法参数、`for`语句初始化语句定义的变量名具有块作用域。

每个名字所在的作用域可根据源代码书写顺序，从定义该名字的语句回溯到碰到的第一个左花括号，或者Java源文件的开头而确定。如果该左花括号是块语句的开头，则该名字具有由该左花括号及其配对的右花括号界定的块作用域；如果是方法体的开头，则该名字具有该方法对应的块作用域；如果是类（接口、枚举）声明的开头，则该名字具有这个类（接口、枚举）对应的类作用域。如果该名字没有定义在任何花括号之内，则具有该Java源文件所在的包对应的包作用域。

对于包作用域和类作用域中定义的名字而言，该名字可在整个包作用域（类作用域）内都可使用，而对于块作用域中定义的名字，该名字只能在定义该名字的语句之后的区域使用，在定义该名字的语句之前的区域不能使用该名字。

对于名字的使用，Java语言（以及多数面向对象语言）还提供了**成员访问**机制来控制一个名字在该名字定义所在的作用域以外的程序区域是否可以被使用。Java语言提供了公有访问控制、受保护访问控制、包级访问控制和私有访问控制几种级别的访问控制。公有访问控制的名字可在任何地方都可使用，受保护访问控制的名字在同一个包内都可使用，但在不同包内只能在定义该名字的类的后代类中使用。包级访问控制的名字只能在同一个包内使用，在其他包中不可使用。私有访问控制的名字不能在定义该名字的作用域以外的区域中使用。

在名字定义所在的作用域以外访问名字需要通过名字解析机制进行访问，因为在不同的作用域中可定义相同的名字。Java语言的名字解析机制是使用点运算符(.)来构成**受限名字**(qualified name)。受限名字可看作一个名字的全名，例如在包`myPackage`的类`MyClass`中定义的方法名`myMethod`的受限名是`myPackage.myObj.myMethod`，其中`myObj`是类型为`MyClass`的对象。对应地`myMethod`称为**简单名字**(simple name)。受限名字实际上将简单名字用所在的层次化作用域（包作用域或对象名）进行限定。为了简化名字的使用，Java语言提供了包引入语句（`import`语句）。在引入某个包之后，使用该包中定义的名字，在不产生冲突的情况下，可使用去掉包作用域之后的名字。

下面以一个例子来说明名字的作用域。注意，程序1.2只是一个程序片段，并没有给出完整的、

程序 1.2 replaceOne.java

```
1  public class replaceOne {
2      static class InOutInt {
3          int value;
4      }
5
6      static boolean getline(char[] s, int maxsize) {
7          int result=0;
8          try {
9              result = in.read(s);
10         } catch (IOException e) {
11             System.err.println(e);
12             System.exit(-1);
13         }
14         return result!= -1;
15     }
16
17     static int makesub(char[] arg, int from, char delim, char[] sub) {
18         int  result;
19         InOutInt i = new InOutInt();
20         InOutInt j = new InOutInt();
21
22         j.value = 0;
23         i.value = from;
24         if (arg[i.value] != delim) result = 0;
25         else {
26             boolean junk = addstr(ENDSTR, sub, j, MAXPAT);
27             if ((!junk)) result = 0;
28             else result = i.value;
29         }
30         return result;
31     }
32 }
```

完全符合语法定义的程序。下面是这个程序片段定义的一些有效作用域（即是某个名字的作用域的程序区域）：

(1) 类作用域replaceOne，包含的程序源代码区域从第1行到第30行，即整个程序源代码；

(2) 类作用域InOutInt，包含的程序源代码区域从第2行到第4行，即类InOutInt的定义；

(3) 方法getline对应的块作用域，包含程序源代码区域从第6行到第15行，即方法getline的方法体及其参数声明；

(4) 方法makesub对应的块作用域，包含程序源代码区域从第17行到第31行，即方法makesub的方法体及其参数声明；

(5) 第9行的catch语句的块语句对应的块作用域，包含程序源代码区域从第10行到第13行；

(6) 第23行的else语句的块语句对应的作用域，包含程序源代码区域从第25行到第29行。

表 1.13 程序replaceOne.java中定义的名字

程序行	定义的名字	名字的作用域
第1行	类名replaceOne	Java源文件replaceOne.java所在的包对应的包作用域
第2行	类名InOutInt	类作用域replaceOne
第3行	原子变量名value	类作用域InOutInt，value是类InOutInt的成员
第6行	方法名getline	类作用域replaceOne
第6行	原子变量名s，maxsize	方法getline对应的块作用域，它们是该方法的参数
第7行	原子变量名result	方法getline对应的块作用域中从第7行到15行部分
第10行	对象名e	第9行catch语句的块语句对应的块作用域
第17行	方法名makesub	类作用域replaceOne
第17行	变量名arg，from，delim，sub	方法makesub对应的块作用域，它们是该方法的参数
第18行	原子变量名result	方法makesub对应的块作用域中从第16行到29行部分
第19行	对象名i	方法makesub对应的块作用域中从第17行到29行部分
第20行	对象名j	方法makesub对应的块作用域中从第18行到29行部分
第26行	原子变量名junk	第23行的else语句的块语句对应的作用域

表1.13给出了程序1.2中定义的名字及其作用域。程序的第20、21行使用了名字value，该处并不属于value的作用域范围内，因此使用受限名字j.value，i.value进行访问。除上述指出的名字定义之外，名字的其他出现都是名字的使用。注意名字in，addstr，ENDSTR，MAXPAT的定义没有出现在该程序片段中。

1.2.5 变量的生存期

通常我们只关心变量的生存期，因为变量是内存区域的抽象，在程序运行过程中会发生改变，而类名、方法名通常在程序运行过程中不发生改变，讨论类与方法的生存期意义不大。

变量的**生存期**(life time)是指从为变量分配存储空间从而在程序运行过程中存在，直到变量失去存储空间这一段程序运行的时间段。详细地说，变量的生存期可分为如下几个阶段：

(1) **分配存储空间**，即将变量名与存储空间建立绑定，从而使得变量获得地址等属性；

(2) **初始化**，即对变量赋初值，从而建立变量与其（有效）值之间的绑定；

(3) **变量使用**。只有在变量与地址绑定之后才能使用变量，在Java语言中，编译器还会在变量使用之前检查是否已经初始化；

(4) **变量收尾**，这也可看作是变量使用的一部分，是一种特殊的变量使用，是在变量失去内存之前所进行的一些必要的工作以保证程序逻辑上的正确性；

(5) **释放存储空间**，即解除变量与存储空间的绑定。

变量的生存期与程序运行时的存储区域布局密切相关。简单地，我们可将一个程序运行时所占据的空间分为**代码区**和**数据区**。代码区是指程序编译之后的机器代码所占用的存储空间，数据区则是程序所要处理的数据，即常量和变量所占用的存储空间。数据区又可大体上分为**静态数据区**、**栈区**(stack)和**堆区**(heap)几个部分。静态数据区用来存放常量和具有静态生存期的变量，栈和堆用来存放具有动态生存期的变量。通常栈区存储空间的分配和释放是隐式的，即由编译器自动生成的目标代码负责，而堆区存储空间的分配，乃至释放都是由程序员通过`new`运算符，或`malloc()`之类的函数显式完成的。

简单地说，变量的生存期可分为**静态生存期**和**动态生存期**两种。静态生存期可理解为在整个程序运行期间都存在，从程序运行开始就分配存储空间，到程序运行结束时才释放存储空间。文字常量实际上是值不变的变量，它的生存期可理解为静态生存期。还有一些变量，例如在C++语言声明为静态(static)的变量，或者在C++语言和Java语言中声明为类的静态成员等都可理解为具有静态生存期。

程序中多数变量都具有动态生存期。Java程序主要有两种变量具有动态生存期：

(1) 声明在方法中具有局部作用域的变量，这种变量通常在栈区分配存储空间。在方法调用时，方法的参数及局部变量获得存储空间并进入生存期。在方法调用完毕返回时，这些变量失去存储空间，结束生存期。

(2) 在程序中使用`new`运算符创建的对象，这种对象通常在堆区分配存储空间。在Java程序中，这些对象都是匿名对象，只能通过指向它的变量进行访问。例如，语句`MyClass myObj = new MyClass()`实际上创建了两个变量：一是`myObj`，它是一个引用变量；另一个变量由`new`运算符创建，是匿名变量，在堆区分配内存，其存储空间由类`MyClass`的非静态数据成员的存储空间构成。

这两个变量通常具有不同的生存期。通常上述语句属于某个方法体，因此`myObj`通常具有局部作用域，它的生存期到该方法调用完毕返回时结束。但它指向的匿名变量的生存期并不随着方法调用完毕而结束。在C++程序中，程序员需要显式使用`delete`运算符释放这种对象的存储空间，维持这种存储空间分配与释放的正确性是程序员一项重要工作，也是导致C++ 程序容易出错的主要原因。在Java程序中，程序员无需显式释放这种对象的存储空间，它们的存储空间由Java虚拟机的垃圾收集机制在适当的时候回收，这大大减轻了程序员的负担。

具有局部作用域变量的初始化和收尾通常都比较简单。程序员可使用初始化语句在声明这种变量的同时对这种变量进行初始化，例如语句`int i = 0`则声明整数型变量`i`并将其初始化为`0`，而上述语句`MyClass myObj = new MyClass()`则是将对象变量`myObj`初始化为指向使用`new`运算符创建的匿名变量。如果程序员没有对这种变量显式初始化，则可理解为使用随机值对它们进行初始化。Java编译器会检查这种变量是否在初始化之后才引用它的（右）值，这也降低了程序出错的可能性。

我们知道，一个类的数据成员可分为**静态数据成员**和**非静态数据成员**。静态数据成员由类的所有对象共享，在Java程序运行过程中是在由Java虚拟器装载类的时候分配存储空间和进行初始化，

可认为这些静态数据成员具有静态生存期（不考虑虚拟机对类的动态装载和卸载）。类的不同对象具有不同的非静态数据成员，在使用new运算符创建对象时分配存储空间和初始化。

从另外一个角度看，类的数据成员可分为**自身数据成员**和**继承数据成员**。自身数据成员是指声明在这个类的定义中的数据成员，而继承数据成员是指从这个类的祖先类继承下来的成员。注意，自身的非静态数据成员和继承的非静态数据成员都是在使用new运算符创建对象时分配存储空间和初始化。

使用new运算符创建的匿名对象通常在分配存储空间之后的初始化比较复杂。Java语言提供了多种方式初始化对象的非静态数据成员，包括：在声明这些成员时同时提供初始化语句；使用初始化块语句进行初始化；使用构造函数进行初始化。我们知道，new运算符后面实际上是调用类的某个构造函数。根据Java语言的规定，类的每个构造函数的第一条语句是以super(...)的形式调用其直接父类合适的构造函数，如果没有这种语句，则调用直接父类的缺省构造函数（没有任何参数的构造函数）。

Java语言采用单继承，只能从一个父类继承数据成员，而不能从多个父类继承数据成员，因此也不会使得一个类的继承数据成员过于复杂。在使用new运算符创建对象时，根据继承链，从根类(Object)开始到这个类本身，每个类执行如下存储分配和初始化工作：

(1) 为这个类自身声明的非静态数据成员分配存储空间，若程序员在声明非静态数据成员还提供了初始化语句，则执行初始化语句对该存储空间进行初始化，否则以全0进行填充；

(2) 然后，如果这个类有非静态初始化语句块，则执行非静态初始化语句块；

(3) 然后再调用合适的构造函数。根据new运算符后面指定的构造函数可确定这个类本身以及它的所有祖先类这时应该执行哪个构造函数。

简单地说，在使用new运算符创建某个类对象时，根据这个类的继承链，从而根类到这个类本身，依次执行初始化语句、初始化语句块、构造函数来初始化该对象的所有非静态数据成员（包括自身数据成员和继承数据成员）。

从逻辑上来说，变量的收尾和释放存储空间应该按照对称的顺序来进行，即根据继承链，先对这个类自身的数据成员实施收尾和释放存储空间，然后再对其父类，一直到根类。变量的收尾工作是指当在变量初始化时，如果占用了除存储空间以外的资源，例如打开了某个文件、建立了某个数据库连接等，那么需要在释放变量存储空间之间确保这些资源已经被释放可，例如所打开的文件已经关闭，数据库连接已经释放等，这些工作称为变量的收尾工作。正确完成这些收尾工作是保证程序正确的前提。C++语言提供了析构函数，程序员可在析构函数中描述这种收尾工作。程序员在使用delete运算符释放对象时自动调用析构函数以完成相应的收尾工作。Java语言不提供可自动调用的析构函数，对象变量的存储空间由垃圾回收机制自动回收，从而像关闭文件、释放数据库连接这些额外的收尾工作程序员通常需要编写另外的方法，并由程序员在适当的时候显式调用以完成这种收尾工作。

变量的生存期与程序的运行过程密切相关，完整、清晰地理解每个变量的生存期，包括其存储空间的分配、初始化以及在程序运行过程中变量值的变化是理解和分析程序的重要基础。但对于稍微有点规模的程序，其中使用的变量可能比较多，而在运行过程中这些变量的值的变化（包括变量之间的指向关系的变化）也可能会非常、非常复杂，要将它们全部理解（最好能以可视化方式展示出来）是非常困难的，因此如何选择合适的视图（例如选择一些关键的变量）来展示变量的生存期及其在生存期中值的变化是程序理解和分析的重要研究问题，也是我们后面要重点思考和研究的问题。

1.2.6 名字小结

名字或说标识符是程序中最重要的要素。通常，程序中的名字有类型名（类名、接口名、枚举名等）、方法名（或说函数名、子程序名、子模块名等）和变量名。绑定是指建立名字及其属性之间的联系。类型名我们最关心其成员、作用域等属性，方法名我们关心其返回类型、参数列表、方法体及其作用域等属性，而变量我们主要关心它的类型、值、地址、作用域和生存期等属性。

变量是存储空间的抽象，变量的地址是存储空间的地址，变量的值是存储在空间中的数据值。引用变量（或指针变量）的值是另外一个变量的地址，这样两个变量建立了一种指向关系。Java语言的对象变量都是引用变量，指向由new运算符创建的该对象变量所属类的一个匿名变量。若两个引用变量指向同一个变量时，这两个引用变量发生了别名。别名现象是导致程序容易出现错误的重要原因之一。

所有的名字都有作用域。作用域是指一段源代码区域，在这段代码区域中某些名字可以使用，而在这个代码区域之外这些名字不可使用，因此这个代码区域称为这些名字的作用域。目前绝大多数语言都是采用静态作用域，或说词法作用域，即在程序运行之前，或者说根据程序的静态文本就能确定每个名字的作用域。从另外一个角度看，作用域确定了名字的使用和名字的定义（或声明）之间的绑定关系，即在该作用域中某个名字的使用到底是使用哪里定义（声明）的名字。

只有变量的生存期才是有意义的。变量的生存期是一个动态的概念，是指在程序运行过程中该变量所存在的时间段。变量所占用的存储空间可分为静态数据区和动态数据区，动态数据区又可细分为栈区和堆区。静态数据取用来存放常量以及类的静态数据成员。栈区用来存放在方法调用时，方法内部声明的局部变量。堆区用来存放使用new运算符创建的匿名对象。变量的生存期可细分为分配存储空间、初始化、变量使用、变量收尾以及释放存储空间这几个阶段。

1.3 程序的类型系统

类型用于将程序要处理的数据进行分类，不同的类型是不同的数据集合。通过数据的类型，程序员可确定数据在计算机存储中的表示，以及数据上可用的操作，因此类型也可看作是数据及其可用操作的一种抽象。对于原子类型数据上可用的操作往往是隐式说明的，而类类型则将数据及其可用的操作都显式地说明，并将有关数据的细节隐藏，形成抽象数据类型。抽象数据类型通过操作及其操作之间应该满足的约束来规范数据的性质，使得程序员可在更高抽象层次对数据进行操作。

类型是程序设计语言中最重要的概念之一，这里对一般程序中有哪些类型，以及类型之间的关系做初步的讨论。

1.3.1 类型表达式

前面已经提到，一般来说，程序设计语言中的类型可分为**原子类型**和**复合类型**两种。原子类型是指不用基于其他类型或者自身类型就已经定义的类型。原子类型通常有布尔类型（逻辑类型）、数值类型（整数类型、浮点类型、双精度类型等）和字符类型。

复合类型是指需要基于其他类型甚至自身类型才能定义的类型。复合类型又可分为用户自定义类型、派生类型和类属类型。用户自定义类型包括类、结构、联合等类型，这些类型的数据都具有比较复杂的结构，可以分解为更多类型的数据，即成员数据。这些类型被成为用户自定义类型是因为

它们是通过由程序员定义其成员数据的类型而定义的。Java语言没有提供结构和联合等类型，但它所提供的接口和枚举类型可看作一种特殊的类类型，我们也将它们归为用户自定义类型。

派生类型包括数组类型、引用类型和指针类型。派生类型和类属类型都是通过类型算子作用到其他类型上而定义得到的类型。数组、引用和指针可看作是程序设计语言预定义的类型算子，而类属类型则是由用户自定义的类型算子。

派生类型和类属类型使得类型可能不是只用一个名字就能表达，而是需要使用**类型表达式**。例如，`int[]`，`LIST<String>`都不是仅使用一个名字表达的类型，而是类型表达式。在C++语言中，由于有指针类型，因此有非常复杂的类型表达式，例如语句`int* ptr_array[10]`声明的变量`ptr_array`具有类型`int*[]`，是一个指向数组的指针，而语句`int (*array_ptr)[10]`声明的变量`array_ptr`具有类型`int(*)[]`，是一个指针数组。

相对而言，Java语言的类型表达式比较简单，特别是若我们不考虑Java语言的类属参数中允许使用`extends`，`super`以及通配符`*`对类属参数进行约束，则其基本的类型表达式仅仅具有两种形式，一种是像`int[]`这样的多维数组；一种是像`List<String>`这样的类属类型。

因此，简单地说，Java语言的类型表达式可定义为：(1) 一个类型名字是类型表达式；(2) 若`T`是类型表达式，则`T[]`是类型表达式；(3) 若`G`是有n个类属参数的类属类型，`T1`，`T2`，$\cdots$，`Tn`是类型表达式，则`G<T1, T2, ···, Tn>`是类型表达式。

不过要注意的是，Java语言目前不允许创建类属类型的数组，例如语句

```
List<String>[] myArray = new List<String>[10]
```

是错误的，不能使用`new`运算符创建类属类型的数组，因此也无法使用貌似类型为`List<String>[]`这样的数组`myArray`。Java语言设计者通过这样的约束简化类属类型的使用，因为他们希望程序员使用Java语言提供的容器类（例如`ArrayList`）来持有多个类属类型的对象，例如可声明`ArrayList<List<String>> obj`这样的变量。

1.3.2 子类型关系

在程序中，类型与类型之间最重要的关系是子类型(subtype)关系。称类型`T`是类型`S`的**子类型**，如果凡是需要使用类型`S`的变量的地方都可使用类型`T`的变量代替，这也是著名的**里氏替换原则**(Liscov substitution principle)。或者从另外一个角度说，只有任意使用类型`S`的变量的地方都可使用类型`T`的变量替换，才能将类型`T`看作类型`S`的子类型。例如，从这个意义上说，整数类型可看作浮点类型的子类型，因为任何用到浮点类型变量的地方都可使用整数类型的变量替换。

前面提到，类型实际上是对数据的划分，某个类型实际上一些数据构成的集合，或者简单地说，类型就是集合。从这个意义上来说，子类型关系就是子集关系，即类型`T`是类型`S`的子类型就意味着类型`T`对应的数据集合是类型`S`对应的数据集合的子集。例如，整数类型是浮点类型的子类型，意味着整数集是浮点数集的子集。

在面向对象程序中，继承可产生类型之间的子类型关系。如果类`A`继承`B`，则类`A`可以认为是类`B`的子类型。不过在面向对象程序中，类不仅仅是类型，它还是一个模块，是程序模块组成的基本构成单位之一，继承也不仅仅产生子类型关系，还是模块的扩充与重用。类`A`继承`B`除了意味着`A`的对象也拥有从类`B`继承下的数据成员之外，也意味着类`A`可重用类`B`的方法，这种重用也不仅仅是增加方法，还可能重定义类`B`中已有的方法。

子类型与这种模块扩充与重用，特别是子类重定义父类的方法可能产生冲突，也即，这种重定义使得子类的对象并不一定能在任何地方都可替换父类的对象，至少子类对父类在方法上的扩充与重定义使得子类对象在替换父类对象方面显得更为复杂，这也是继承机制在面向对象程序设计中比较难以掌握与使用的原因之一。

在Java语言程序（也包括在C++语言程序）中，要检查类型T是否是类型S的子类型，只要将类型T的变量赋值给类型S即可，如果这种赋值不产生语法错误，则说明T是S的子类型，否则T不是S的子类型。

简单地，在Java语言程序中，子类型关系可通过以下几种方法产生：

(1) 如果类T继承类S，或者接口T继承接口S，或者类T实现接口S，则类型T是类型S的子类型；

(2) 如果类型T是类型S的子类型，则类型T[]也是类型S[]的子类型；

对于类属类型，程序员若使用类似下面的方式进行声明

```
class GSubClass<T> extends GSuperClass<T> { ... }
```

则意味着对对任意的类型表达式T，类型GSubClass<T>都是类型GSuperClass<T>的子类型，例如GSubClass<String>是GSuperClass<String>的子类型等等。当然这里的类属类型都可含有多个类属参数，甚至父类与子类可以有不同的类属参数，但只有使用相同的类型去实例化父类与子类中相同的类属参数时才有可能建立子类型关系。

另外有一种情况程序员要特别注意：设G是只有一个类属参数的类属类型，且类型T是类型S的子类型，但是类型G<T>不是类型G<S>的子类型。对于多个类属参数也类似，即同样的类属类型作用到具有子类型关系的类型后得到的两个类型并不具有子类型关系。

1.3.3 类型匹配与类型转换

类型匹配是指在将一个变量的值赋给另外一个变量时，编译器需要检查这两个变量的类型是否满足合适的语义约束规则。具体来说，对于赋值语句A = B，这种语义约束规则本质上就是要保证变量A的存储空间能够存放变量B的值而不会发生数据丢失。特别地，因为Java语言的对象变量采用引用语义，赋值语句A = B使得变量A也指向变量B所指向的对象，这就要求变量B所指向的对象能够替换变量A原本应该指向的对象，也即变量B所指向的对象的类型应该是变量A所指向的对象的类型的子类型。

通常在程序中有两种情况将变量的值赋给另外一个变量：(1) 赋值语句A = ...计算赋值运算符右边的表达式的值并将其赋给变量A；(2) 方法调用时计算实际参数表达式的值并将其赋给形式参数所声明的变量。前一种情况是显式的赋值，后一种情况是隐式的赋值。这两种情况都要对表达式值的类型与赋值目标变量的类型进行匹配。

类型转换是指当两个不同类型的变量在赋值或其他运算时，将一个变量的类型转换成另外一个变量的类型以保证运算结果具有确定的、唯一的类型。简单来说，主要在两种情形下需要进行类型转换：(1) 在赋值语句或参数传递时，当值的类型与赋值目标变量的类型不相同时将值的类型转换为目标变量的类型；(2) 在计算表达式时，某个二元运算符两边操作数的类型不相同时需要发生类型转换。

类型转换可分为隐式类型转换和显式类型转换两种。隐式类型转换由编译器自动生成相应的代码来完成，而显式类型转换由程序员使用类型转换表达式完成，类型转换表达式通常具

有(TypeExpression)valueExpression的形式，即将类型表达式作用于值表达式，例如(int)3.5。

编译器自动生成代码进行隐式类型转换的基本原则就是要保证不发生数据丢失。对于两个类型T和S，只有在满足以下条件之一时才发生隐式类型转换：(1) 若T和S都是原子类型，且其中一个类型的取值范围要比另一个的取值范围宽，这时将取值范围窄的类型的变量转换为取值范围宽的类型；(2) 若T和S都是类（接口）类型，且它们之间存在子类型关系，例如T是S的子类型，则将T的变量转换为S类型。

在赋值时，如果值表达式的类型不能隐式类型转换为赋值目标变量的类型，则程序员需要使用显式类型转换，否则编译器会报告操作。对于显示类型转换，如果源类型与目标类型都是原子类型，且目标类型的取值范围比源类型的取值范围宽，则编译器不会有任何警告，否则若目标类型的取值范围比源类型的取值范围窄，则编译器会警告程序员可能发生数据的丢失。如果源类型与目标类型都是类（接口、枚举）类型，则源类型与目标类型之间必须具有子类型关系，否则编译器报告错误。若源类型是目标类型的子类型，这时源类型的变量也可隐式转换为目标类型；如果目标类型是源类型的子类型，即这时必须使用显式类型转换，至于是否可将源类型的变量转换为目标类型则需要在运行时检查，如果源类型的变量在运行时刻实际指向的对象的类型是目标类型的子类型，则这种转换是可行的，否则会抛出与类型转换有关的异常。

在Java语言中，将取值范围窄的类型的变量转换为取值范围宽的类型，或者将一个类型的子类型的变量转换为该类型的这两种情况的类型转换称为**向上转换**(up-casting)，而将取值范围宽的类型的变量转换为取值范围窄的类型，或者将一个类型的的变量转换为它的子类型的这两种情况的类型转换成为**向下转换**(down-casting)。简单地说，向上转换是可以隐式进行的，不会发生数据丢失或类型转换异常，而向下转换需要显式类型转换，而且可能发生数据数据丢失或类型转换异常，因此编译器会进行警告，或在运行时进行动态检查。

在计算表达式时，当一个二元运算符两边的操作数类型不一致时，这时的操作数通常都是原子类型的变量，因此对两个操作数中属于取值范围比较窄的类型的那个操作数变量转换为取值范围比较宽的类型，也即这时的原则也是要保证不发生数据丢失。

1.4 程序的模块架构

程序的模块架构是指从程序的控制结构，或者说从完成现实世界任务的角度看程序可以分成哪些组成部分，这种组成部分称为程序的**模块**(module)，是现实世界中子任务的抽象。

这一节首先简单讨论程序设计语言通常提供哪些程序模块划分机制，以及如何在物理上组织程序模块，即在计算机存储中如何组织和存储程序模块所对应的源代码。然后我们简单讨论由于模块的划分而带来的相关问题，包括模块之间参数的传递以及模块之间的调用关系与模块的运行环境。

1.4.1 程序的物理与逻辑组织

Eclipse JDT实际上给出了程序的基本模型：首先所有的程序在某个工作空间(workspace)，一个工作空间可能有多个项目(project)，而一个项目有多个程序包(package)，每个程序包有多个程序源文件，每个源文件可能有多个类，每个类可能有多个方法。这个模型也就是Java程序的模块架构，同时也对应Java程序的物理组织：工作空间是存放程序的起始路径，每个项目对应该文件路径下的

一个子路径，而每个程序包又对应项目路径下的一个子路径，该程序包中的多个源文件都存放在该子路径下。

Java语言还提供了**子程序包**的概念，但要注意的是，子程序包并不Java程序模块逻辑组织上的概念，而是物理组织上的概念。说子程序包不是逻辑组织上的概念是因为某个程序包的子程序包中的类并不属于该程序包，例如用语句`package com.java`声明的程序包是用语句`package com`声明的程序包的子程序包，但是在程序包`com.java`中声明的类并不属于程序包`com`，或者更明确地说，如果类`MyClass`是在程序包`com.java`中声明的具有包级访问控制的类，则在程序包`com`中不能使用类`MyClass`。

实际上，子程序包是Java程序在物理组织上的概念，程序包和子程序包对应Java源文件存放时路径与子路径的概念。例如，程序包`com`中的Java源文件存放在路径com下面，那么它的子程序包`com.java`的源文件则存放在路径com/java下面。而从逻辑的角度看，程序包中的类与子程序包中的类处于一种并列关系，它们完完全全属于不同的程序包。

通常来说，一个项目下的所有程序源文件构成了通常意义下的一个程序（或说一个软件的源代码部分），这样程序包、文件（编译单元）、类、方法是Java程序的基本模块类型。不同模块之间主要存在两种关系：一种是组成关系，一种是调用关系。组成关系是指程序包、文件、类、方法之间有一种层次关系，即一个程序包可能由多个文件构成、一个文件可能由多个类构成、一个类可能有多个方法等等。

模块之间的调用关系是指一个模块可能要使用另外一个模块所提供的接口，或说服务（为了与Java语言中的接口类型相区别，我们下面主要是使用服务这一词汇）。对于程序包和文件而言，它对外界提供的服务主要由类来体现，即外界能使用这个程序包或文件中定义的哪些类。对于程序包而言，它为外界提供的服务由在这个程序包定义的所有公有类构成。对于文件，它为自己所在程序包不同的程序包提供了一个公有类，为自己所在的程序包则还提供了受保护与包级访问控制的类。

对于类，它为外界提供的服务主要由方法来体现，即外界能使用这个类中定义的哪些方法。这个外界又分为与这个类所在程序包不同程序包的非后代类、不同程序包的后代类、同一程序包但不同文件的类、同一文件的不同类几个层次。类为外界提供的服务还与本身的访问控制有关，即与类是公有类、包级访问控制类还是私有类有关。表1.14总结了各种类所提供的服务所包含的（不同访问控制类型的）方法。

表 1.14 各种类提供的服务所包含的方法

提供服务的类	不同程序包的非后代类	不同程序包的后代类	同一程序包不同文件的类	同一文件的不同类
公有类	公有方法	公有方法 受保护方法	公有方法 受保护方法 包级访问控制的方法	公有方法 受保护方法 包级访问控制的方法
包级访问控制的类	无	无	同上	同上
私有类	无	无	无	同上

对于方法，它是为外界提供服务的基本单元，但它与外界的通信主要通过方法的参数以及方法的返回类型来体现。在现代程序设计语言中，异常处理机制已经成为语言的一部分，从而对于方法

来说，该方法可能抛出的异常也是方法与外界通信的一种手段，或者说可将方法抛出的异常看作方法返回的一种方式。

1.4.2 方法的参数传递

方法是程序模块（程序包、类或说对象）为外界提供服务的基本单元，外界通过调用方法而完成程序所要完成的基本任务。方法的调用者与方法本身之间的通信主要通过方法参数完成。方法调用者在调用时提供的参数称为**实际参数**(actual parameter)，而方法定义时给出的参数列表称为**形式参数**(formal parameter)。

在方法调用时将实际参数与形式参数匹配的过程称为参数传递。为了完成参数传递，编译器通常要进行前面所说的类型匹配，即实际参数的类型是否与形式参数的类型相匹配。在程序运行时，参数传递使得方法可以通过某种方式处理方法调用者所提供的数据，而方法调用者通过传递不同的参数使得方法的每次调用可完成相似但不完全相同的功能。

目前程序设计语言主要有**按值传递**(call by value)、**按引用传递**(call by reference)、**按名传递**(call by name)以及**拷进拷出**(copy in and copy out)几种参数传递机制。简单来说，按值传递就是将实际参数的值赋给形式参数，而按引用传递就是将实际参数的值所存放的地址传递给形式参数，使得形式参数指向实际参数的值，按名传递则不计算实际参数（表达式）的值，而是将表达式本身传递给方法，由方法在必要的时候才进行计算，在一些数据库语言或脚本语言中的“宏替换”机制与按名传递有些类似。而拷进拷出的传递机制则在进行方法时将实际参数的值拷贝给形式参数，然后在方法返回时，又将形式参数的值拷贝回给实际参数。

对于Java语言而言实际上只提供了按值传递一种参数传递机制，但由于Java语言的对象变量本身是采用引用语义的，因此对于对象变量而言，这种按值传递本质上与按引用传递相同，即使得实际参数与形式参数指向相同的对象。

1.4.3 方法的调用与运行环境

除了方法的定义之外，方法名的所有出现都是方法调用。方法调用可看作是在程序运行过程中，方法名与方法定义进行绑定的过程。在现代程序设计语言中，允许对方法进行**重载**(overloading)与**重定义**(overriding)。方法重载是指相同的方法名通过不同的形式参数类型而区别方法的不同实现。方法重定义是指子类根据需要重新实现在祖先类中定义的方法。目前的面向对象程序设计语言只有当子类的方法在形式参数类型、返回类型与父类方法完全相同时才构成方法的重定义。

由于方法的重载和重定义，因此在方法调用时需要确定到底是调用哪个方法，这个过程称为**方法名解析**(resolving)。对于Java程序，通过调用方法的对象可确定所调用的方法来自哪个类，然后再根据实际参数类型在这个类中所有同名的互为重载的方法（包括这个类自身的以及继承的方法成员）中确定真正调用的方法。这通过将实际参数类型与各个互为重载方法的形式参数类型进行匹配而确定，形式参数类型与实际参数类型最匹配的方法作为真正调用的候选方法。但这还不是最后调用的方法，因为还需要根据调用方法的对象变量在运行时时真正指向的对象类型在重定义的方法中进行选择，如果运行时指向的对象类型是声明时指向的对象类型的子类型，而且在该子类中又重定义了候选方法，那么真正调用的方法是子类中重定义的方法。

每次方法调用都对应一个**活动记录**(activation record)。活动记录在栈区分配内存，包括方法的

形式参数、方法声明的局部变量、方法的调用者（即在哪个模块中调用该方法）及返回地址，以及在方法中计算表达式所需的一些临时变量等。对于面向对象程序设计语言而言，方法调用都是通过对象变量来完成（或通过类名调用静态方法），因此活动记录还需记录该对象变量所指向的对象，或者也可将这个对象变量看作方法的第一个参数。

方法的**运行环境**(runtime environment)是指在调用方法时该方法可访问的所有变量及其绑定。在一些程序设计语言，如Ada中，需要通过计算方法的调用环境和定义环境来确定该方法中的变量使用到底是在哪里定义的变量。方法的调用环境是指哪个程序模块调用了该方法，以及由此形成的调用链中所可能访问到的变量，而方法的定义环境是指在哪个程序模块中定义了方法，而且在某些程序设计语言中，由于方法可以嵌套定义，内部定义的方法可访问外围方法中定义的变量，这样构成了定义环境中的变量访问链。对于Java语言而言，方法是不允许嵌套定义的，因此方法的定义环境很简单，就是定义该方法的类。进一步方法的运行环境也比较简单，除了方法本身定义的局部变量之外，还有调用方法的对象所在类的成员，以及在这个方法中可访问的类的公有静态成员等，例如程序`HelloPrinter.java`中的`System.out`。

1.5 文献及注释

这一章对程序设计语言的基本概念进行了讨论，以建立我们对程序及其构件的基本理解，作为我们考察程序分析技术的基础。我们这里主要是从程序的语法构成的角度讨论程序中的名字、类型系统和模块架构，有关程序设计语言原理的更多内容可参考相关的教材，例如[1, 2]等。这一章关于Java语言的抽象语法树节点类型的介绍来自开源开发平台Eclipse的帮助文档。

第二章　程序视图

这一章讨论为了分析程序，我们首先需要提取哪些与程序有关的视图，例如通过静态分析可以得到的程序的抽象语法树、符号表程序控制流程图、程序依赖图、模块调用图等，以及需要通过动态分析才能得到的程序状态转换图等。

2.1　程序抽象语法树

抽象语法树是程序最重要的视图，从某种意义上来说，抽象语法树比程序源代码文本更为重要，因为它更适用于程序分析。要获得像Java这样的通用程序设计语言编写的程序的抽象语法树并不是一件容易的事情，因为这几乎要实现一个编译器中从词法分析到语法分析的所有功能。

幸运的是，我们不需要从零开始，而可以利用已有的工具来获得程序的抽象语法树。早期有LEX和YACC来帮助程序员构建自己的编译器，近期有ANTLR(Another Tool for Language Recognition)。ANTLR是Terence Parr博士领导的团队开发的一个用于构造编译器的工具，具体的细节浏览网站http://www.antlr.org，以及参看Tenrence Parr博士的书[3, 4]。

不过对于Java语言而言，我们还可选择Eclipse平台提供的Eclipse JDT来得到Java程序的抽象语法树，因为虽然ANTLR是一个很优先的编译器构造工具，然而还是需要程序员提供所编译语言的文法，虽然Java语言的文法也可从互联网上找到，但要完全熟悉Java 语言的文法并应用到ANTLR上显然要比直接使用Eclipse JDT花费的时间要更多。

Eclipse JDT将Java程序看作它要处理的文档，因此类似XML的解析，Eclipse对Java程序进行了建模，并称为它的文档对象模型(Document Object Model, DOM)，实际上也就是Java程序的抽象语法树。Eclipse JDT的程序包org.eclipse.jdt.core.dom提供了用于构造和访问Java程序抽象语法树的类与API。程序员可利用这些类构建、访问与修改已有Java程序的抽象语法树，乃至构造新的Java程序并编译与运行它们。

这一节我们先介绍如何构建已有Java程序的抽象语法树，然后简要介绍Eclipse JDT的每个AST节点所对应的类，然后讨论如何利用Eclipse JDT提供的类访问抽象语法树。注意，这一章所讨论的类除非有特别说明都来自Eclipse JDT的程序包org.eclipse.jdt.core.dom。

2.1.1　构建抽象语法树

在Eclipse JDT中使用类`ASTParser`可Java程序的抽象语法树。程序2.1给出的代码片段可构建某个Java源程序文件的抽象语法树，其中假定我们将该源程序文件的内容读入到字符串变量`sourceCode`之中。

程序 2.1 构建Java程序抽象语法树的程序片段

```
1  String sourceCode = null;
2  // Read a java program file to sourceCode
3
4  ASTParser parser = ASTParser.newParser(AST.JLS3);
5  parser.setKind(ASTParser.K_COMPILATION_UNIT);
6  parser.setSource(sourceCode.toCharArray());
7  rootNode = (CompilationUnit)parser.createAST(null);
```

为构建某个Java源程序文件的抽象语法树，我们首先要使用类ASTParser提供的静态对象工厂方法newParser()获得一个ASTParser的对象，该方法需要一个int类型的参数level来指定该Java源程序文件是根据哪个版本的Java语言规范编写的，类AST提供的静态常量JLS2和JLS3分别表示第二版和第三版的Java语言规范。注意第二版的Java语言规范不支持类属类型，所以都应该使用使用常量JLS3。

类ASTParser的方法setKind(int kind)设置要编译的源代码类型。类ASTParser定义了如下四个常量表示四种类型：

(1) K_COMPILATION_UNIT表示编译单元，即一个完整的Java源程序文件，这样方法createAST()得到的抽象语法树的根节点类型是CompilationUnit。如果程序员在调用createAST()方法之前不调用方法setKind()则缺省编译的源代码类型是编译单元；

(2) K_CLASS_BODY_DECLARATIONS表示源代码仅仅包含类声明，这样方法createAST()得到的抽象语法树的根节点类型是TypeDeclaration；

(3) K_STATEMENTS表示源代码是语句序列，这样方法createAST()得到的抽象语法树的根节点类型是Block；

(4) K_EXPRESSION表示源代码是表达式，这样方法createAST()得到的抽象语法树的根节点类型是类Expression的子类。

也就是说，类ASTParser不仅可以用于分析一个完整的Java程序文件，也可用于分析像类声明、语句序列，乃至表达式这样的Java代码片段。

类ASTParser的方法setSource()设置要编译的源代码。有四个互为重载的setSource()方法，每个方法都只有一个参数，类型分别是char[]、ICompilationUnit、IClassFile和ITypeRoot，分别对应由字符数组表示的源代码文件，或者是一个编译单元对应的源代码文件，或者是一个字节码文件对应的源代码文件，或者是这两者之一（即ITypeRoot实际上是ICompilationUnit和IClassFile的父接口）。后面三个接口类型所指向的对象都Eclipse JDT所定义的Java元素(IJavaElements)，与仅用字符数组表示的源代码文件相比，它们包含了更多的信息，从这样的对象不仅可访问到相应的源代码文件，而且可确定该Java程序所在的项目、工作空间等等，从而在编译的时候有更多可用的信息，例如可建立不在该源代码文件中定义的名字的绑定，从而在程序员访问得到的抽象语法树时可获得有关这些名字的更多信息。

最后调用类ASTParser的方法createAST()可得到所设置的源代码文件的抽象语法树。方法createAST()可接受一个类型为IProgressMonitor的进程监控器对象。接口IProgressMonitor定义在程序包org.eclipse.core.runtime中，使得可在用户界面中显示进度条以表示编译的进度，这通常

用于编译长文件时才使用，避免用户长时间等待过程中用户界面无任何变化。多数情况下无需使用这样的对象，直接用null调用方法createAST()即可。

方法createAST()的返回类型是类类型ASTNode，表示所编译的源代码的抽象语法树的根节点。程序员需要根据方法setKind()所设置的源代码类型将方法createAST()返回的对象转换为合适的抽象语法树节点类型。下一小节我们将对类ASTNode及其子类做更多的介绍。

简单地来说，上述步骤可构建一个Java源程序的抽象语法树。类ASTParser还提供了一些其它方法来定义这个构建过程，其中一些方法我们简单地介绍如下。方法createASTs()可用于编译多个Java源程序文件，方法setResolveBindings()要求在编译的同时建立程序中名字的绑定，即确定该名字及相关的一些信息。如果一个抽象语法节点对应于名字（主要是程序包名、类型名、成员名、方法名、变量名），则它会提供方法resoveBinding()返回相应的绑定信息，这些信息可能对于我们分析程序有更大的帮助。类ASTParser的方法createBindings()可对一系列的Java元素建立绑定信息。建立绑定信息可能需要花费更多的运行时间，因此Eclipse JDT建议在非必需的情况下不要这样做。不过名字的绑定信息对于程序的分析可能会非常有用，因此我们需要对相关的方法与类做进一步的研究。

类ASTParser的方法setCompilerOptions()可设置编译选项，而方法setSourceRange()可设置只构建部分源代码的抽象语法树。类ASTParser也提供了方法setFocalPosition()来设置编译时所关注的程序点以生成简化的抽象语法树，即与所要关注的程序点密切相关的部分将给出完整的抽象语法树，而其他地方则会简化甚至省略，这样使得程序员可更快速地访问所关注的程序点相关的抽象语法树。有关这些方法以及类ASTParser的更多信息可参考Eclipse开发平台的帮助中的JDT Plug-in Developer Guide 部分。

2.1.2 抽象语法树节点概述

Eclipse JDT所建立的抽象语法树节点的基类型是类ASTNode，所有表示抽象语法树节点的类都是类ASTNode的子类。每个抽象语法树节点表示Java源代码的一个语法构造单元，例如名字、类型、表达式、语句或声明等。每个抽象语法树节点都唯一地属于某个类AST的对象，也即Eclipse JDT还提供了类AST对应于（整棵）抽象语法树的概念。实际上，一个抽象语法树节点及其儿子都必然属于同一个类AST的对象，这与同一个抽象语法树的节点属于同一个抽象语法树对应，或者说这意味着，抽象语法树根节点所属的类AST对象代表了这棵抽象语法树。

每个抽象语法树节点都自动维护一个指向它父亲节点的指针（对象引用），使得程序员可以自下而上地访问整个抽象语法树，当然程序员也可以自上而下地访问。新创建的抽象语法树节点的父亲节点指针指向null，当用合适的方法将其设置为某个抽象语法树节点的儿子节点时，该方法将自动设置相应的父亲节点指针，并将该抽象语法树节点原来的儿子的父亲节点指针指向null。

简单地说，一个抽象语法树中的节点的父亲节点指针是自动维护的，而儿子节点则根据节点类型的不同而进行添加，不同的节点提供了不同的添加儿子节点的方法。在一个抽象语法树中的节点不允许出现环（正如树是无环的图），所有添加儿子节点的方法都会进行检测是否可能产生环，如果可能产生环，则添加不会成功。

程序员可以自己构造一棵抽象语法树，类AST提供了许多工厂方法创建新的抽象语法树节点对象（即类ASTNode的子类的对象），若节点类名为NodeType，则相应的工厂方法为newNodeType()。例如对于节点类CompilationUnit，则类AST有方法newCompilationUnit()。

更多的时候，程序员是像上一节所说的那样使用类ASTParser对Java源程序文件进行编译生成一棵抽象语法树，这时树中的节点还存有与源代码的一些对应信息，例如该节点对应源文件中什么位置的源代码等。

每个抽象语法树节点还含有一些二进制位标志，用来表示关于该节点的更多信息，例如编译器可能用某些标志来表示一些出错信息等。新创建的抽象语法树节点没有设置标志位。每个抽象语法树节点还可能包含更多的可以由程序员自己定义的性质，实际上节点的儿子信息也可作为一种性质添加到节点中。新创建的节点不包含任何性质。

为了管理每个抽象语法树节点的性质，Eclipse JDT设计了类StructuralPropertyDescriptor描述节点可能具有的三种性质：

(1) 简单性质，指节点的这个性质的值不是子节点或子节点列表，例如，一个简单名字节点（类SimpleName的对象）使用简单性质存放该节点对应的名字串本身；

(2) 儿子性质，指节点的这个性质是另外一个节点（即是它的儿子节点），例如，一个方法声明节点（类MethodDeclaration的对象）使用儿子性质来存放它的方法名所对应的节点；

(3) 儿子列表性质，指节点的这个性质是另外一系列节点（即是它的一系列儿子节点），例如，一个块节点（类Block的对象）的多个语句对应的节点是该块节点的儿子列表性质。

对应这三种性质，类StructuralPropertyDescriptor有三个子类SimplePropertyDescriptor、ChildPropertyDescriptor和ChildListPropertyDescriptor。每个性质都有一个标识，因此类StructuralPropertyDescriptor有方法String getId()返回对应性质的标识。这个类还提供了方法Class getNodeClass()返回是哪种类型的节点有当前性质描述符所描述的性质。这个类的方法isSimpleProperty()、isChildProperty()和isChildListProperty()返回当前性质描述符所描述的性质类型。

类SimplePropertyDescriptor的方法Class getValueType()返回所描述性质的取值类型，而类ChildPropertyDescriptor的方法Class getChildType()返回儿子节点的类型。这两个类都有方法boolean isMandatory()来返回所描述的性质是否是节点所必有的性质，节点必有的性质不能取值为null。类ChildListPropertyDescriptor的方法class getElementType()返回列表中的儿子节点类型。

实际上，性质描述符类用来统一地管理节点的性质，例如统一地实现accept()方法，在无需知道节点具体类型的情况下，配合访问者对象（ASTVistor的对象）对抽象语法树进行遍历等。对于具体的节点而言都已经提供了更明确的方法来设置该节点的各种性质或添加儿子，因此程序员通常不会用到这些性质描述符类。

每个抽象语法树节点有父亲节点指针，有二进制标志位，可以有各种不同的性质等构成了抽象语法树节点的共性，这些也反映在基类ASTNode所提供的属性与方法中。表2.1给出了类ASTNode的一些属性字段与方法。注意，类ASTNode定义了一系列的静态整型常量来表示节点类型，例如常量COMPILATION_UNIT表示是编译单元，即实际类型是CompilationUnit。当然表2.1并没有给出类ASTNode的所有方法，更多的细节请参看Eclipse的帮助文件。

为了更好地介绍抽象语法树节点类，我们先根据Java语言程序的一般结构介绍相应的抽象语法树节点，必要时讨论其中某些节点的父类，然后再讨论对应程序中一些很具体的元素，主要是表达式、语句和类型等的抽象语法树节点类及其子类。希望这样能让大家能从抽象语法树节点的角度，既对对Java程序的结构有大致的把握，又对一些细节能够有比较深入的了解。

表 2.1 类ASTNode的一些方法

int getNodeType() 保留字返回对象的实际节点类型，用相应的整型常量表示
Class nodeClassForType(int type) 静态方法，获得有type指定类型的节点所对应的类对象
int getFlags() 返回以整数表示的标志
void setFlags(int) 设置标志
Map properties() 返回该节点所拥有的性质，每个性质包含字符串String表示的性质名，以及类型为Object的值。注意，返回的性质不能进行修改
void setProperties(String,Object) 设置该节点的一个性质
Object getStructuralProperty(StructuralPropertyDescriptor) 返回性质描述符所描述的性质
void setStructuralProperty(StructuralPropertyDescriptor,Object) 设置性质描述符所描述的性质
List structuralPropertyForType() 返回当前节点类型有哪些性质，使用性质描述符列表表示
int getStartPosition() 返回节点对应的程序片段在源程序中以字符计的起始下标位置
int getLength() 返回节点所对应的程序片段以字符计的长度
ASTNode getParent() 返回该节点的父亲节点
StructuralPropertyDescriptor getLocationInParent() 返回该节点的父亲节点用何种性质描述符描述该节点
ASTNode getRoot() 返回该节点所在抽象语法树的根节点
AST getAST() 返回该节点所属的抽象语法树对象
void delete() 将当前节点从其父亲节点中删除
String toString() 将节点转换为字符串，该字符串非常接近节点所对应的源程序代码，但是不保证会符合Java语言的语法规则，此方法多数用于调试
void accept(ASTVistor) 按照访问者模式接受访问者（ASTVistor的对象）的访问，下一节详细讨论抽象语法树的访问

表 2.2 Java程序的基本语法结构

AST 节点类型	主要组成成分
编译单元(CompilationUnit)	包声明(PackageDeclaration) 包导入声明(ImportDeclaration) 类声明(TypeDeclaration) 接口声明(TypeDeclaration) 枚举声明(EnumDeclaration)
枚举声明(EnumDeclaration)	枚举常量声明(EnumConstantDeclaration)
类或接口声明(TypeDeclaration)	初始化块(Initializer) 属性字段声明(FieldDeclaration 方法声明(MethodDeclaration)
属性字段声明(FieldDeclaration)	变量声明(VariableDeclaration)
方法声明(MethodDelcaration)	返回类型(Type) 参数声明(SingleVariableDeclaration) 方法体(Block)

表2.2给出了Java程序的基本语法结构，因此我们也首先从类CompilationUnit开始介绍，这个类也是使用类ASTParser编译一个Java源程序文件所得到的抽象语法树根节点类。

下面几节讨论主要的一些抽象语法树节点类（注意，我们不讨论与标注、注释以及类属类型相关的类），这些类都是类ASTNode的后代类。我们先讨论编译单元及其基本组成部分（包导入与声明、修饰符、体声明、类型声明、字段声明、方法声明等）对应的节点类及其基本方法，然后展开讨论几种复杂的抽象语法树节点类，包括类型、语句和表达式。

我们将看到，这些类的设计基本上对应Java语言的语法，提供了相应的get和set来返回节点类所对应的语法单元的组成成分，如果组成成分是一个列表，则返回可用于遍历和修改的列表。在必要时这些类还会提供一些额外的方法来获得一些额外的信息。熟悉Java语言的语法不难理解下面这些类的设计，而熟悉这些类的方法可帮助我们更好地理解Java语言的语法。

2.1.3 编译单元及其基本构成的AST节点

编译单元节点

编译单元节点类CompilationUnit是类ASTNode的直接子类。这个类主要是三种性质，对应一个编译单元的三种主要组成成份，即包声明、包导入声明和类型声明。因此类CompilationUnit声明了三个公有静态常量，即三个性质描述符来描述这三种性质：

(1) public static final ChildPropertyDescriptor PACKAGE_PROPERTY：该性质是可选的，描述符标识串是"package"，性质类型是PackageDeclaration（该描述符中使用这个类的类对象，即PackageDeclaration.class来表示）。对应地，类CompilationUnit有方法PackageDeclaration getPackage()返回包声明子节点，而方法void setPackage(PackageDeclaration)设置（即添加）包声明子节点。

(2) public static final ChildListPropertyDescriptor IMPORTS_PROPERTY：描述符标识串是"imports"，性质类型是ImportDeclaration。对应地，类CompilationUnit有方法List imports()

返回包导入声明子节点列表。程序员还可通过对此列表进行添加或删除来改变该编译单元的包导入声明。

(3) public static final ChildListPropertyDescriptor TYPES_PROPERTY：描述符标识串是"types"，性质类型是AbstractTypeDeclaration，即实际上可以是类、接口或枚举类型。对应地，类CompilationUnit有方法List types()返回类型声明子节点列表。程序员还可通过对此列表进行添加或删除来改变该编译单元的类型声明。

类CompilationUnit还提供了一些方法来维持抽象语法树中的节点与程序源代码之间的位置关系。方法int getExtendedLength(ASTNode)返回一个节点（通常应该是当前编译单元节点的后代节点）在源代码中的长度，而方法int getExtendedStartPosition(ASTNode)返回一个节点在源代码中的起始位置。这两个方法与类ASTNode的方法getLength()和getStartPosition()的不同之处在于它们会将源代码中的注释和空白考虑在内。方法int getLineNumber(int)和int getColumnNmber(int)返回以字符计的指定位置在源代码文件中的行与列，对程序员来说行与列更容易定位源程序。对应地，方法int getPosition(int line, int column)则将行列位置转换为以字符计的位置。

类CompilationUnit还提供了方法来返回使用类ASTParser对源文件编译时出现的警告信息和错误信息。方法Message[] getMessages()返回编译器报告的信息，而方法IProblem[] getProblems()则返回编译时碰到的问题（包括警告和错误）的更多信息。类Message提供了方法String getMessage()、int getLength()和int getSourcePosition()分别返回问题信息，出现问题的以字符计的源代码起始位置和长度。接口IProblem则对编译出现的问题进行了分类，从而提供与该问题相关的更多信息，其中方法String getMessage()也可返回（更人性化）的信息，而且方法int getSourceLineNumber()还可返回该问题对应的源代码行。

如果程序员要改变类ASTParser编译Java源程序文件而构建的以类CompilationUnit的对象为根的抽象语法树，则需要调用类CompilationUnit的方法void recordModification()使得该根节点可记录对它及其后代节点的改变。

包声明和包导入声明节点

包声明节点类PackageDeclaration和包导入声明节点类ImportDeclaration都是类ASTNode的直接子类。它们最重要的性质都是NAME_PROPERTY，这是一种儿子节点性质（即是性质描述符类ChildPropertyDescription的对象），分别记录声明的（所属的）程序包名字以及导入的类名字。相应地，这两个类都提供了方法Name getName()和方法void setName(Name)来分别获得和设置名字，这里类Name是用于表示名字的抽象语法树节点类，下面将作详细的介绍。

由于在导入程序包时可使用星号(*)来表示在需要的时候才导入某个程序包的类，因此类ImportDeclaration还有一个简单性质ON_DEMAND_PROPERTY（类SimplePropertyDescriptor的对象），而这个类相应地提供了方法boolean isOnDemand()和方法setOnDemand(boolean)来获得和设置这个导入声明是否使用了星号。

通过对编译单元、包声明和包导入声明节点的介绍，我们已经基本清楚每个节点怎样使用性质描述符类来描述该节点所具有的性质（包括儿子节点），所以在下面的介绍中我们将不再给出描述节点性质的常量，而只是讨论该节点所提供的访问和设置这些性质的方法，以及一些相关的方法。

体声明和修饰符节点

类或接口声明（类TypeDeclaration）以及枚举类型声明（类EnumDeclaration）都是体声明（类BodyDeclaration）的子类，所以我们这里先介绍体声明节点类。实际上，还有许多节点类也是体声明节点类的子类，表2.3给出了它的一些子类（我们暂时省略了与Java标注有关的类）。

表 2.3 类BodyDeclaration的子类

AST 节点类型	含义
AbstractTypeDeclaration	TypeDeclaration和EnumDeclaration的父类
EnumDeclaration	枚举声明
TypeDeclaration	接口声明或类声明
EnumConstantDeclaration	枚举常量声明
FieldDeclaration	类或接口的属性域声明
Initializer	（不在任何方法中的）初始化语句块
MethodDeclaration	类或接口的方法（包括构造方法）的声明

类BodyDeclaration是类ASTNode的直接子类，它主要负责管理上述子类的与修饰符相关的性质，因为类、接口、每句、字段、方法，乃至初始化语句块的一个共同特征是都有static，public等这样的修饰符。当然这些语法单元还有一个共同特征是可以有文档化的注释（JavaDoc可处理的注释），但是我们目前也将与注释有关的内容暂时省略。

因此类BodyDeclaration最重要的方法是List modifiers()，返回该体声明节点的修饰符列表，程序员可遍历并该表这个列表。该列表的元素具有接口类型IExtendedModifer。修饰符类Modifier实现了这个接口，还有一些标注类实现了这个接口（但我们这里不讨论这些标注类）。实际上，接口IextendedModifier只有两个方法boolean isModifier()和boolean isAnnotation()来判断当前对象是标注还是修饰符。

修饰符类Modifier对应像public，private，protected，static，final等这样的修饰符，它也是类ASTNode的直接子类。这个类目前（规范第三版）实际上融合了两种设计，在第二版时是使用一个整数来记录所有的修饰（整数不同的位代表不同的修饰），因此这个类提供了很多静态方法来判断一个整数中是否含有某个修饰符，例如方法static boolean isPublic(int)判断一个整数是否含有public修饰符。在这种设计下，类BodyDeclaration提供了方法int getModifiers()和void setModifiers(int)来获得和设置修饰符，后者（设置方法）在第三版中已不提倡使用。

在第三版中，修饰符类使用一个简单性质来区别不同的修饰符，这个性质对应（从而也记录）该修饰符的关键字。因此类Modifier有一个公有嵌套类Modifier.ModifierKeyword来记录对应的关键字。这时类Modifier提供非静态方法来判断当前修饰符类对象是否是某个修饰符，例如方法boolean is Public()判断当前对象是否是public修饰符。类Modifier的方法Modifier.ModifierKeyword.getKeyword()可返回当前对象的关键字，而类ModifierKeyword的方法String toString()可得到该关键字串（例如"public"）。

类ModifierKeyword还提供了方法int toFlagValue()将该关键字对应的修饰符转换为第二版所设计的整数，以及静态方法static Modifier.ModifierKeyword fromFlagValue(int)及static Modifier.ModifierKeyword toKeyword(String)分别从整数和字符串得到修饰符的关键字对象。

类型声明节点

类声明和接口声明都是类型声明，类型声明的抽象语法树节点类是TypeDeclaration，这个类是AbstractTypeDeclaration的直接子类，而类AbstractTypeDeclaration才是类ASTNode的直接子类。类AbstractTypeDeclaration的另外一个直接子类是EnumDeclaration，因为枚举也是一种类型，在这里我们暂时不讨论枚举声明节点类。

类AbstractTypeDeclaration维护类型声明的两个重要性质，一个是类型名，一个是类型体声明（或者也可以是类型的成员声明）。这个类的方法List bodyDeclarations()返回体声明列表，该列表的元素类型是类BodyDeclaration，程序员可遍历和改变这个列表从而访问和改变类型声明的成员。为了让程序员了解某个类型声明中的体声明到底是什么，类AbstractTypeDeclaration的方法ChildListPropertyDescriptor getBodyDeclarationsProperty()返回当前类型声明节点（对象）的体声明的性质描述符对象，利用这个描述符对象，程序员可得到有关体声明列表的更多类型信息。

类AbstractTypeDeclaration的方法SimpleName getName()和方法void setName(SimpleName)分别获得和设置类型名信息，而方法ChildPropertyDescriptor getNameProperty()可获得有关类型名性质描述符对象。

类AbstractTypeDeclaration还提供了一些方法来帮助程序了解有关类型声明的一些信息。方法boolean isLocalTypeDeclaration()判断当前类型声明是否是一个局部类声明，方法boolean isMemberTypeDeclaration()判断是否是成员类声明（即这个类声明节点的父亲节点也是一个类型声明或匿名类声明），方法boolean isPackageMemeberTypeDeclaration()判断是否包成员的类型声明，即是否是顶层类声明。

类型声明节点类TypeDeclaration主要维护类型声明的属性字段、方法和成员类声明这些性质。方法FieldDeclaration[] getFields()返回类型声明的属性字段声明数组，方法MethodDeclaration[] getMethods()返回类型声明的方法声明数组，而方法TypeDeclaration[] getTypes()返回类型声明中的（嵌套）类声明数组。不过这些返回的数组不能被修改，程序员需要通过AbstractTypedeclaration的方法bodyDeclarations()返回体声明节点列表来修改类型声明的成员。

类TypeDeclaration的方法boolean isInterface()来判断当前类型声明是否是接口声明，而方法void setInterface(boolean)设置当前类型声明为接口声明。方法Type getSuperclassType()返回当前类型声明所声明的父类，而方法void setSuperclassType(Type)设置所声明的父类。方法List superInterfaceTypes()返回当前类型声明中要实现的接口列表，其元素类型也是Type，程序员可遍历和修改这个列表。类Type是对应类型的抽象语法树节点类，我们将在后面做进一步的介绍。

属性字段声明与变量声明节点

在Eclipse JDT中，一个属性字段声明节点（类FieldDeclaration的对象）将多个变量声明片段（类VariableDeclarationFragment的对象）组合成一个体声明（类FieldDeclaration是类BodyDeclaration的子类），这多个变量声明片段实际上可能声明了类的多个属性字段，但它们共享相同的修饰符和类型。属性字段声明的基本形式如下（其中中括号表示0或1个、花括号表示0或多个）：

[Javadoc] {ExtendedModifier} Type VariableDeclarationFragment
{, VariableDeclarationFragment };

也即，一个属性字段声明节点由多个修饰符节点，一个类型节点，以及一个或多个变量声明片段节点组成，这也是属性字段声明节点的性质构成。属性字段声明节点类FieldDeclaration继承了类BodyDeclaration中处理修饰符节点的方法，自己还提供了方法List fragments()返回变量声明片段列表，程序员可遍历和修改这个列表，列表元素类型是VariableDeclarationFragment。类FieldDeclaration的方法Type getType()和void setType(Type)分别获得和设置属性字段声明中的类型节点。

类VariableDeclarationFragment和单个变量声明节点类SingleVariableDeclaration都是变量声明节点类VariableDeclaration的直接子类。变量声明节点类则是类ASTNode的直接子类。变量声明节点类VariableDeclaration是一个抽象类，除了一些与性质描述符相关的方法之外都是抽象方法，因此程序员很少使用。

变量声明片段与单个变量声明之间的最大区别是，变量声明片段只维持变量名、初始化表达式等性质，而单个变量声明则除变量名、初始化表达式之外还有修饰符和类型。变量声明片段的基本形式如下：

Identifier {[]} [= Expression]

而单个变量声明的基本形式如下：

{Modifier} Type Identifier {[]} [= Expression]

放在花括号的中括号是因为Java语言在声明数组时，中括号既可以放在类型名后面也可以放在变量名后面，例如，

```
int[] a, b[];
```

在这个变量声明中，a是一个一维数组，而b是一个二维数组。

VariableDeclarationFragment的方法SimpleName getName()和void setName(SimpleName)获得/设置变量名节点，而Expression getInitializer()和void setInitializer(Expression)分别获得和设置变量声明中的初始化表达式（如果没有则返回null）。表达式节点类Expression是最复杂的节点，我们下面将做更多的介绍。类VariableDeclarationFragment还提供了方法int getExtraDimensions()和方法void setExtraDimensions(int)分别获得和设置在跟在变量后面给出的额外的数组维数。实际上，抽象类VariableDeclaration除了与性质描述符相关的方法之外，也就是声明了这些抽象方法。

类SingleVariableDeclaration除了具有上述与变量名和初始化表达式相关的方法之外，还提供了方法List modifiers()返回变量声明的修饰符列表，这个列表可被修改。同时方法int getModifiers()可返回整数形式的修饰符。最后还有方法Type getType()和方法void setType(Type)分别获得和设置变量声明中的类型。

方法声明与初始化语句块节点

方法声明节点可能是一般成员方法的声明也可能是构造方法的声明，一般成员方法的声明通常具有如下形式（不考虑类型参数和Javadoc注释）：

```
{Modifier} (Type|void) Identifier([parameter {, parameter}])
                                    [ throws TypeName {, TypeName}] (Block|;)
```

其中(Type|void)表示要么是一个类型，要么是关键字void，而(Block|;)表示方法体要么是一个块语句，要么是一个分号（空语句，抽象方法没有方法体）。构造方法的声明通常具有如下形式（也不考虑参数和Javadoc注释）：

```
{Modifier} Identifier([parameter {, parameter}])
                        [ throws TypeName {, TypeName}] Block
```

也即，构造方法的声明不用声明返回类型，而方法体也必须是块语句。

类MethodDeclaration是方法声明的抽象语法树节点类，它是BodyDeclaration的子类。这个类的方法SimpleName getName()和void setName(SimpleName)可获得和设置方法名节点。方法Type getReturnType2()和void setReturnType2(Type)可获得和设置方法的返回类型（之所以有2为后缀，是因为对应规范第二版的没有2为后缀的方法已被丢弃，在规范第二版返回类型是必须的，但在第三版是可选的）。

类MethodDeclaration的方法Block getBody()和void setBody(Block)可返回和设置方法体。类Block是语句节点类Statement的子类，我们后面会做进一步的介绍。类MethodDeclaration的方法List parameters()可获得形式参数列表，列表元素类型是SingleVaraibleDeclaration，程序员可遍历和修改这个列表。类MethodDeclaration的方法List throwExceptions()可获得所声明抛出的异常列表，列表元素类型是Name，程序员也可遍历和修改这个列表。

类MethodDeclaration的方法boolean isConstructor()可判断是否是构造方法声明，而方法boolean isVarargs()可判断是否可变参数方法（Java语言规范第三版增加了对可变参数的支持），最后方法int getExtraDimensions()和void setExtraDimensions(int)可返回和设置方法返回类型的数组额外维数。与变量声明类似，Java语言在方法返回数组类型时，允许将表示数组维数的中括号放在形式参数列表（的右圆括号）后面（方法体之前）。

初始化语句块由一个可选的静态修饰符(static)以及一个块语句构成。类Initializer是初始化语句块的抽象语法树节点，它也是体声明BodyDeclaration的子类，因此继承了其中处理修饰符的方法。类Initializer的方法Block getBody()和void setBody(Block)可返回和设置初始化语句块中的块语句。

至此我们已经将与Java程序的基本结构（编译单元、类型声明、属性字段声明、方法声明）相关的抽象语法树节点介绍完毕，下面我们进一步介绍在构造Java程序中到处都可能出现的语法构造，这主要是类型、语句和表达式，其中名字是最基本的表达式，我们也将在表达式一节中介绍。

2.1.4 类型节点

类型节点类Type实际上对应类型名的使用，或者更准确地说对应类型表达式。表2.4给出了类Type的子类。通过这个表可以看出，Java语言的类型表达式比较简单，如果不考虑类属类型（参数化类型），那么只有数组类型（类ArraryType）、原子类型（类PrimitiveType）和带限定的类型名（类QualifiedType）或不带限定的简单类型名（类SimpleType），实际上后两者相当于用户自定义类型的类型名（类名、接口名或枚举名）。

表 2.4 类Type的子类

AST 节点类型	含义
ArrayType	数组类型，例如：int[]
ParameterizedType	类属化类型，例如：List<String>
PrimitiveType	原子类型，包括boolean，byte，char，int，short，long，double，float，void
QualifiedType	带限定的类型名，例如：java.lang.String
SimpleType	简单类型名，例如：String
WildcardType	用于类属机制的通配类型，例如：？ extend Student

类Type的方法很简单，提供了几个返回类型为布尔类型的方法来判断当前节点到底是什么类型，例如isArrayType()判断是否是数组类型，isPrimitiveType()判断是否是原子类型，isQualifiedType()判断是否是带限定的类型名，isSimpleType()判断是否是简单类型名等。

类PrimitiveType定义嵌套类PrimitiveType.Code来编码byte，short，char，int，long，float，double，boolean，void这九种原子类型。类PrimitiveType的方法PrimitiveType.Code getPrimitiveTypeCode()和void setPrimitiveTypeCode()可分别获得和设置当前类型节点所对应的原子类型编码，而静态方法static PrimitiveTypeCode toCode(String)可将用字符串表示的原子类型名（例如"byte"）转换为相应的编码。而类PrimitiveType.Code的方法toString()则可将编码转换回为原子类型名字符串。

类ArrayType对应一维或多维数组类型，它的方法Type getComponentType()返回数组类型的构件类型，而方法Type getElementType()返回数组类型的元素类型，对于一维数组而言这两者返回的类型是相同的，但是对于多维数组而言，构件类型仍然可能是数组类型，而元素类型是指最基本的那个类型，肯定不是数组类型。例如对于数组类型int[][]，他的构件类型是int[]，而元素类型是int。类ArrayType的方法void setComponentType(Type)设置数组的构件类型（注意构件类型可以是非数组类型，也可以是数组类型，因此无需设置元素类型的方法）。方法int getDimensions()可返回数组的维数，同样无需设置维数的方法，因为可以根据所设置的构件类型而自动计算。

Java程序中可使用点运算符连接多个字符串表示一个类型名字，例如PrimitiveType.Code就是一个带限定的类型名。在抽象语法树中，节点类QualifiedType用来表示这种类型名。不过由于点运算符也可能用于构件对象名字，例如System.out，所以编译器即使在确定第一个名字是一个类型名的情况下，也不能确定整个名字是一个类型名，因此像PrimitiveType.Code这样的名字在编译后得到的抽象语法树中可能是如下两种表示中的一个：

1. QualifiedType(SimpleType(SimpleName("PrimitiveType")), SimpleName("Code"))
2. SimpleType(QualifiedName(SimpleName("PrimitiveType"), SimpleName("Code")))

第一种形式以类QualifiedType的对象为子树的根节点，并有两个直接儿子，一个的类型是SimpleType，一个是SimpleName，而第二形式以类SimpleType的对象为子树的根节点，只有一个类型为QuanlifiedName的直接儿子。在编译得到的抽象语法树中，这两种形式都有可能，而且通常以第二形式更为常见，程序员在处理这种名字的时候需要注意。

类QualifiedType的方法Type getQualifier()来获得该限定类型名的限定名，注意该限定名

还有可能也是带限定名的类型名，方法SimpleName getName()来获得该限定类型名（最后的那个）简单名。对应地方法void setQualifier(Type)和void setName(SimpleName)分别设置该限定类型名（前面的）限定名和（最后的）简单名。

类SimpleType则只有方法Name getName()来返回该类型的名字，这个名字可能是QualifiedName，也可能是SimpleName，后两个类都是类Name的子类。对应地，类SimpleType的方法void setName(Name)设置类型的名字。

2.1.5 语句节点

在抽象语法树节点类中，用于表示语句的节点类是Statement，这个类在语言规范第三版中已经没有有用的方法（先前的两个用于处理注释的方法已被丢弃）。类Statement是类ASTNode的直接子类，其他各种表示语句的抽象语法树节点类都是类Statement的子类（除了与try语句中的catch部分对应的节点类CatchClause外，类CatchClause是类ASTNode的直接子类）。表2.5列出了类Statement的各种子类。

表 2.5 类Statement的子类

语句种类	AST 节点类型
基本语句	AssertStatement, ConstructorInvocation, EmptyStatement, ExpressionStatement, SuperConstructorInvocation, SynchronizeStatement, TypeDeclarationStatement, VariableDeclarationStatement
块语句	Block
分支语句	IfStatement, SwitchCase, SwitchStatement
循环语句	DoStatement, EnhanceForStatement, ForStatement, WhileStatement
控制转移	BreakStatement, ContinueStatement, LabelStatement, ReturnStatement
异常处理	ThrowStatement, TryStatement

类AssertStatement是断言语句对应的节点类，它的基本形式如下：

assert Expression [:Expression]

即关键字assert后跟两个表达式，前一个表达式称为断言表达式（则要检验的是否满足该断言的表达式），后一个表达式是信息表达式，则断言不满足的时候显示该表达式生成的字符串。因此类AssertStatement的方法Expression getExpression()和Expression getMessage()分别返回这两种表达式，对应地方法void setExpression(Expression)和void setMessage(Expression)则设置这两种表达式。

类ConstructorInvocation是构造函数调用对应的节点类，它的基本形式是this(参数表)。因此这个类的最重要方法是List arguments()返回调用的实际参数列表，列表元素类型是Expression，程序员可遍历和修改这个列表。类SuperConstructorInvocation是超类构造调用对应的节点类，它的基本形式是[Expression.]super(参数表)，这里关键字super之前的表达式用来当超类是嵌套类的时候选择合适的外围对象，具体含义可参见Java语言规范。因此这个类除了方法List

arguments()返回调用的实际参数列表（列表元素类型也是Expression，程序员也可遍历和修改这个列表）外，还有方法Expression getExpressio()和void setExpression(Expression)返回和设置super前面的表达式。

类EmptyStatement没有对程序员处理抽象语法树有用的方法，仅仅是提供一个支持空语句存在的节点类而已。类SynchronizedStatement是同步语句对应的节点类。同步语句的基本形式是synchronized (Expression) Block，即在关键字synchronized后跟一个表达式，然后是一个块语句。因此类SynchronizedStatement的方法Block getBody()和void setBody(Block)返回和设置其中的块语句，而方法Expression getExpression()和void setExpression(Expression)返回和设置其中的表达式。

类ExpressionStatement将表达式包装成语句，对应Java语言中表达式加分号构成表达式语句。因此这个类的方法Expression getExpression()和void setExpression(Expression)返回和设置其中的表达式。类似地，类TypeDeclarationStatement将类型声明包装成语句，这个类的方法AbstractTypeDeclaration getDeclaration()和void setDeclaration(AbstractTypeDeclaration)返回和设置其中的类型声明。

类VariableDeclarationStatement将多个共享相同类型和修饰符的变量声明片段包装成语句。因此这个类的方法Type getType()和void setType(Type)返回和设置变量声明语句中的类型，而方法List modifiers() 返回其中的修饰符列表，列表元素的类型是IExtendedModifier，程序员可遍历和修改这个列表。这个类也提供了方法int getModifiers()返回以整数表示的修饰符。方法List fragements()返回其中的变量声明片段列表，列表元素类型是VariableDelcarationFragment，程序员也可遍历和修改这个列表。

类Block是块语句的节点类，它的方法List statements()返回其中的语句列表，列表元素类型是Statement，程序员可遍历和修改这个列表。类ThrowStatement是异常排除语句的节点类，这种语句的基本形式是throw Expression，因此这个类的方法Expression getExpression()和void setExpression(Expression)返回和设置其中的表达式。

类TryStatement是异常捕获语句的节点类，这种语句的基本形式是：

try Block {CatchClause} [finally Block]

即try后面跟一个块语句，然后跟0个或多个catch句子，最后还可能有0个或1个finally语句块。因此，这个类的方法Block getBody()和void setBody(Block)返回和设置try后面的语句块，方法Block getFinally()和void setFinally(Block)返回和设置finally后面的语句块。而方法List catchClauses()返回其中的catch句子列表，列表元素类型是CatchClause，该列表可被程序员遍历与修改。

catch句子的基本形式是catch(形式参数) Block，即关键字catch后面跟一个形式参数说明（即声明要捕获的异常类型），然后后面是块语句（该形式参数的作用域仅限于该块语句）。类CatchClause是这种句子的节点类，它是ASTNode的直接子类。这个类的方法Block getBody()和void setBody(Block)可返回和设置其中的块语句可，而方法SingleVariableDeclaration getException()和void setException(SingleVariableDeclaration)可返回和设置其中声明的要捕获的异常类型，这实际上是一个单个变量声明。

类IfStatement是if语句的节点类，它的方法Expression getExpression()返回其中的条件表达式，而Statement getThenStatement()返回其中的then分支语句，而Statement getElseStatement()返

回其中的else分支语句。这些方法也都有对应的set方法，读者很容易给出这些方法的基调(signature)，这里不再赘述。

类SwitchStatement是switch语句的节点类，它的方法Expression getExpression()返回其中的条件表达式，同时也有相应的设置方法。方法List statements()返回其中用来处理各种情况的语句序列，列表元素类型Statement，按照书写的顺序给出其中的所有语句，其中的case Expression:以及default:等用来标记不同情况语句（实际上是标号）以类SwitchCase的对象表示，从而将switch中的语句进行分组。类SwitchCase也是Statement的子类，它的方法Expression getExpression()返回case后面的表达式（对应的设置方法可设置其中的表达式），而方法boolean isDefault()判断是否是default的情况。

类DoStatement是do语句的节点类，而类WhileStatement是while语句的节点类，这两个类提供的方法相同，都有方法Expression getExpression()返回其中的循环表达式，而方法Statement getBody()返回其中的循环体（语句），相应的设置方法则可设置循环表达式和循环体。

类ForStatement是for语句的节点类，它提供方法List initializers()返回其中的初始化表达式序列，列表可被遍历与修改，列表元素类型是Expression。类ForStatement的方法Expression getExpression()返回其中的循环表达式，而方法Statement getBody()返回其中的循环体（语句），这两个方法有相应的设置方法可设置循环表达式和循环体。for循环语句还有一部分被称为循环计数器修改表达式序列，类ForStatement的方法List updaters()返回该表达式序列，列表元素类型也是Expression，该列表可被遍历与修改。

Java语言规范第三版还提供了增强型的for语句，它的基本形式是：

```
for（形式参数声明：表达式）语句
```

例如下面的语句可输出整数数组intArray中的所有整数：

```
for (int a:  intArray) System.out.printf("%4d", a);
```

类EnhancedForStatement是这种语句的节点类，它的方法Statement getBody()返回循环体语句，而方法SingleVariableDeclaration getParameter()返回其中的形式参数声明，这也说明这时只能声明单个变量，而方法Expression getExpression()可返回其中的表达式。这些方法也有对应的设置方法，以设置循环体、形式参数声明和表达式。

Java语言允许在break和continue后面使用标号指定要跳转的地方，因此类BreakStatement和类ContinueStatement都有方法SimpleName getLable()和void setLable(SimpleName)分别返回和设置这种语句的标号。类LabeledStatement是带标号语句的节点类，其基本形式“标号:语句”，它可作为break和continue语句的跳转目标。这个类的方法SimpleName getLabel()返回其中的标号，而方法Statement getBody()返回其中的语句。这两个方法也有相应的设置方法。

最后，类ReturnStatement是return语句的节点类，这种语句的基本形式是return 表达式，因此类ReturnStatement提供了方法Expression getExpression()和void setExpression(Expression)分别返回和设置其中的表达式。

2.1.6 表达式节点

表达式是Java语言（C++语言也类似）中最复杂的语法单元，类Expression是表达式的抽象

语法树节点类。这个类本身只有一些与绑定解析相关的方法，我们这里暂不讨论。表2.6给出了类Expression的一些直接子类。

表 2.6 类Expression的子类

表达式种类	AST 节点类型
名字	Name
文字	BooleanLiteral, CharacterLiteral, NumberLiteral, StringLiteral, NullLiteral, TypeLiteral
基本表达式	Assignment, CastExpression, ConditionExpression, InstanceofExpression, VariableDeclarationExpression
数组访问等	ArrayAccess, ArrayCreation, ArrayInitializer
成员访问等	ClassInstanceCreation, FieldAccess, MethodInvocation, SuperFieldAccess, SuperMethodInvocation, ThisExpression
表达式构造	InfixExpression, ParenthesizedExpression, PostfixExpression, PrefixExpression

名字是最基本的表达式。Java语言中的名字也分为带限定的名字和简单名字。例如myVar是一个简单名字，而myObject.myField是一个带限定的名字。类Name是名字的抽象语法树节点类，它有两个子类：类QualifiedName和SimpleName。类Name的方法boolean isQualifiedName()判断当前的名字是否是带限定的名字，而方法boolean isSimpleName()判断是否是简单名。方法String getFullyQualifiedName()可返回当前名字的全名字串。

带限定的名字被认为具有形式Name.SimpleName，前面的Name称为名字的限定，而最后是一个简单名。因此类QualifiedName的方法Name getQualifier()返回前面的限定，而SimpleName getName()返回最后的简单名。这两个方法有相应的设置方法，分别设置其限定和简单名。类SimpleName的方法String getIdentifier()返回该名字串，而方法void setIdentifier(String)设置名字串。类SimpleName还提供了方法isDeclaration()来判断该简单名的出现是定义（声明）还是引用。注意，名字的限定一定是使用，而不可能是声明。

类BooleanLiteral是表示布尔文字常量（即true或false）的节点类，它的方法boolean booleanValue()可返回该文字常量所表示的布尔值，而方法void setBooleanValue(boolean)设置布尔值。类NullLiteral表示文字常量null，没有对程序员特别有用的方法。

类CharacterLiteral是表示字符文字常量的节点类，它的方法char charValue()返回该常量所表示的字符，而方法void setCharValue(char)设置其字符值。这个类还提供了方法String getEscapedValue()返回该字符常量的字符串值，例如字符'a'的字符串值是"á"，即在程序中用来表示该文字常量的整个串。该方法有对应的设置方法设置该常量的字符串值。

类StringLiteral是表示字符串常量的节点类，它的方法String getLiteralValue()返回该常量所表示的字符串，例如字符串常量"Hello"所对应的类StringLiteral的节点使用这个方法返回的就是"Hello"。这个类还提供了方法String getEscapedValue()返回在程序中用来表示该常量的整个串，例如对于字符串常量"Hello"，这个方法返回的是"Ḧello"。这两个方法都有对应的设

置方法。

类NumberLiteral是表示数值常量的节点类，它的方法String getToken()返回表示该数值常量的串，例如常量123所对应的类NumberLiteral的节点使用这个方法返回的是"123"。程序员需要利用其他的信息来判断该数值常量的真正类型（整数还是双精度数），从而将该字符串解析成真正的常量。

类TypeLiteral表示的是类型常量。在Java语言中可使用Type.class甚至void.class来表示一个类型为Class的常量，例如int.class是一个用来表示整数类型的Class对象。因此类TypeLiteral的方法Type getType()可返回这种常量中的类型部分（从而知道是哪种类型的类型常量，即Class的哪个实例）。

类Assignment是赋值表达式的节点类，它的方法Expression getLeftHandside()返回赋值运算符左边的表达式（左值表达式），而Expression getRightHandSide()返回赋值运算符右边的表达式（右值表达式）。由于Java语言（像C++语言一样）支持多种赋值运算符，因此类Assignment定义了一个嵌套类Assignment.Operator来记录所使用的赋值运算符，这个嵌套类的方法String toString()返回当前对象所表示的赋值运算符串，例如赋值运算符+=对应的对象使用这个方法返回"+="，而静态方法Assignment.Operator toOperator(String)则可将相应的字符串转换为这个类的对象。类Assignment的方法Assignment.Operator getOperator()可返回当前赋值表达式的赋值运算符。类Assignment也提供了相应的设置方法来设置左值表达式、右值表达式和赋值运算符。

类CastExpression是类型转换表达式的节点类，Java语言的类型转换表达式的基本形式是(Type)Expression，因此这个类的方法Expression getExpression()返回其中（待转换）的表达式，而Type getType()返回目标类型，这两个方法有相应的设置方法设置表达式及类型。

类ConditionalExpression是三元运算符?:构成的条件表达式的节点类，其方法Expression getExpression()返回其中的条件表达式（即?之前的表达式），而Expression getThenExpression()返回其中问号与冒号之间的表达式，而Expression getElseExpression()返回冒号之后的表达式。这三个方法都有相应的设置方法设置这三个表达式。

Java语言中的instanceof用来判断某个表达式的类型是否是某个类型的（子类型），其基本形式是Expression instanceof Type，因此这个类的方法Expression getLeftOperand()返回其中的表达式，而Type getRightOperand()返回其中的类型，相应的设置方法用来设置表达式和类型。

类VariableDeclarationExpression是变量声明表达式的节点类。变量声明表达式是变量声明语句除了分号剩下的部分，它由修饰符（可选）、类型及（一个或多个）变量声明片段组成。因此这个类的方法List modifiers()返回其中的修饰符列表，这个列表可被遍历与修改，其元素类型是IExtendedModifier，而方法Type getType()和void setType(Type)分别返回和设着其中的类型声明，最后方法List fragments()返回其中的变量声明片段列表，该列表可被遍历与修改，其元素类型是VariableDeclarationFragment。

类ArrayAccess是数组元素访问表达式的节点类，其基本形式是Expression[Expression]，前一个表达式应该是一个数组，而中括号内的表达式是下标表达式。这个类的方法Expression getArray()和Expression getIndex()分别返回这两个表达式，而对应的设置方法可设置这两个表达式。

类ArrayCreation是创建并初始化数组的节点类，其可能的形式如下：

```
new PrimitiveType [Expression] {[Expression]}{[]}
new TypeName [<Type {, Type}>] [Expression] {[Expression]}{[]}
new PrimitiveType [] {[]} ArrayInitializer
new TypeName [<Type {, Type}>] [] {[]} ArrayInitializer
```

即是关键字new后面跟一个类型名字，然后是一个或多个中括号括住的维数表达式，最后还可跟零个或多个中括号（因为Java语言允许暂时不确定数组（最后）的某些维数，例如new int[2][]，但空的中括号不能放在有维数的中括号之前，而且没有确定的维数可以通过数组初始化表达式自动确定）。

类ArrayCreation的方法ArrayType getType()和void setType(ArrayType)返回和设置其中的数组类型名。方法List dimensions()返回其中的维数表达式列表（即去掉中括号之后维数表达式序列），该列表的元素类型是Expression，可被遍历和修改。方法ArrayInitializer getInitializer()返回其中的数组初始化表达式，这个方法也有相应的设置方法。

类ArrayInitializer是数组初始化表达式的节点类，Java语言中的数组初始化表达式是由花括号括住而用逗号分隔的多个表达式。这个类的方法List expressions()返回其中的表达式列表，该列表可被遍历和修改，列表元素的类型是Expression。

类ClassInstanceCreation是创建类实例（即对象）的节点类。Java语言中创建类实例的基本形式如下：

```
[expression.] new [<Type {, Type}>] Type([Expression {, Expression}])
                                        [AnonymousClassDeclaration]
```

注意，这里的表达式用于选择合适的外围对象（当创建是嵌套类的实例时），而尖括号中的类型列表是当构造函数本身也是类属函数时需要给出的实际类型参数，而圆括号之前的类型才是构造函数所在的类的类名（这个名字中还可以含有类型参数，当这个类本身是类属类型时），圆括号中的表达式是构造函数的实际参数。当创建的是匿名类实例时还可能给出匿名类的声明（即这个类的字段和方法等）。

类ClassInstanceCreation的方法Expression getExpression()返回new之前的表达式，而Type getType()返回其中的类型，AnonymousClassDeclaration getAnonymousClassDeclaration()返回其中的匿名类声明。这三个方法也有相应的设置方法以设置表达式、类型和匿名类声明。类ClassInstanceCreation的方法List arguments()返回调用构造函数的实际参数列表，该列表可被遍历与修改，其元素类型是Expression。

类FieldAccess是属性字段访问表达式的节点类，其基本形式为Expression.Identifier，因此这个类的方法Expression getExpression()和SimpleName getName()分别返回其中的表达式和名字（字段的标识符）。这两个方法相应的设置方法设置表达式和名字。

类SuperFieldAccess是访问超类属性字段的表达式的节点类，其基本形式为

```
[ClassName.]super.Identifier
```

即可选的类名后跟super再加上字段名，其间使用点运算符连接。这个类的方法Name getQualifier()和SimpleName getName()分别返回其中的类名和字段名，相应的设置方法可设置这两个名字。

类ThisExpression是this表达式的节点，这种表达式用来选择合适的对象，其形式是

[ClassName.]this

即可选的类名后跟关键字this，中间用点运算符连接。这个类的方法Name getQualifier()和void setQualifier(Name)分别返回和设置其中的类名。

当使用类ASTParser对Java程序进行编译构造抽象语法树时，有几种表达式都类似属性字段访问表达式：

(1) 像foo.this这样的表达式只能被表示为一个this表达式，即一个包含类名（简单名）foo的ThisExpression表达式。这种表达式以关键字this结尾，不是以标识符结尾，因此不是属性字段访问表达式；

(2) 像this.foo这样的表达式只能被表示为一个属性字段访问表达式，它包含一个this表达式及一个简单名；

(3) 含有关键字super的表达式只能是超类字段访问（或方法调用）；

(4) 像foo.bar这样的表达式可能被表示为一个带限定的名字（类QualifiedName的对象），也可能是一个属性字段访问表达式。在没有更多绑定信息的情况下，编译器很难区分这两种形式，因此返回这两种形式都是可能的；

(5) 其他以标识符结尾的表达式，例如foo().bar都只能被表示成属性字段访问表达式（也即类FieldAccess的对象）。

类MethodInvocation是方法调用表达式的节点类，其基本形式是：

[Expression.] [<Type {, Type}>] Identifier([Expression {, Expression}])

其中的表达式的结果通常是一个对象，而尖括号中的类型列表是调用类属方法时需要给出的实际类型参数，圆括号中的表达式列表是调用方法的实际参数。因此这个类的方法Expression getExpression()返回其中的对象表达式，方法SimpleName getName()返回方法名。这两个方法也有对应的设置方法，可设置其中的对象表达式和方法名。类MethodInvocation的方法List arguments()返回实际参数表达式列表，程序员可遍历和修改该列表，列表元素的类型是Expression。

类SuperMethodInvocation是调用超类方法表达式的节点类，其基本形式为

[ClassName.]super.Identifier([实际参数列表])

即可选的类名后跟super再加上方法名及实际参数，其间使用点运算符连接。这个类的方法Name getQualifier()和SimpleName getName()分别返回其中的类名和字段名，相应的设置方法可设置这两个名字。这个类的方法List arguments()返回实际参数表达式列表，程序员可遍历和修改该列表，列表元素的类型是Expression。

类ParenthesizedExpression表示圆括号括号住的表达式，其方法Expression getExpression()和void setExpression(Expression)可返回和设置其中的表达式。类PostfixExpression表示使用后缀运算符构造的表达式，这个类定义嵌套类PostfixExpression.Operator表示其中的后缀运算符（实际上只有++和--两个后缀运算符）。类PostfixExpression的方法Expression getOperand()可返回其中的操作数表达式，而方法PostfixExpression.Operator getOperator()可返回其中的后缀运算符，相应的设置方法可设置操作数和运算符。类PrefixExpression表示使用前缀运算符构造的表达式，也有相同的设计，定义了嵌套类PrefixExpression.Operator表示

其中的前缀运算符（++，--，+，-，～，!等可作为前缀运算符），而这个类的方法Expression getOperand()和PrefixExpression.Operator getOperator()分别返回其中的操作数和运算符，相应的设置方法设置操作数和运算符。

类InfixExpression是中缀运算符构造的表达式，它也定义了嵌套类InfixExpression.Operator来表示中缀运算符。上述三个以Operator命名的嵌套类与前面提到的Assignment.Operator都有相同的设计，提供静态方法toOperator(String)将表示运算符的字符串转换为这种类的对象（不同类的方法的返回类型当然不同），而方法String toString()将这种对象转换为表示运算符的字符串。

类InfixExpression比表示前缀和后缀表达式的两个类都稍微复杂一些，因为当某中缀运算符满足结合律时，可将连续使用该中缀运算符连接的多个操作数编译成一个深度更少的子树，例如表达式(((a+b)+c)+d)不用编译成高度为4的子树，而是编译成树高为2的子树，将a看作左操作数，b为右操作树，而{c，d}为扩展操作数列表。因此这个类的方法InfixExpression.Operator getOperator()、方法Expression getLeftOperand()和方法Expression getRightOperand()分别返回运算符、左操作数和右操作数，而方法boolean hasExtendedOperands()判断是否有扩展操作数，如果有可使用方法List extendedOperands()返回扩展操作数列表，这个列表可被遍历和修改，其元素类型是Expression。

2.1.7 抽象语法树的访问

有时我们需要遍历某个Java程序的整个抽象语法树，例如，我们可能需要遍历整个抽象语法树以获得程序所定义和使用的所有名字列表。Eclipse JDT使用访问者模式来遍历整个抽象语法树，其访问者类是ASTVistor。

类ASTVistor提供了一系列的方法来实现对抽象语法树节点的方法，其中：

(1) 为每种具体的节点类（即类ASTNode的子类）都提供了一个visit()方法，例如，对于类Type有方法boolean visit(Type)。所有visit()的方法的返回类型都是boolean，按照设计者的意图，方法visit()对当前节点进行访问，在访问完毕之后，如果还需要继续访问当前节点的儿子节点，则该方法应该返回true，否则返回false（当某些节点的儿子节点比较简单的情况下，可以在该节点的visit()方法本身完成儿子节点的访问，从而无需进一步访问其儿子节点）。

(2) 为每种具体的节点类还提供了一个endVisit()方法，这个方法将在访问完当前节点的所有儿子节点之后被调用，可完成一些在儿子节点访问完毕之后的一些收尾工作。

(3) 类ASTVisitor还提供了两个方法void preVisit(ASTNode)和void postVisit(ASTNode)，这两个方法可给出一些与具体节点类无关的访问工作，其中方法void preVisit(ASTNode)在调用visit()之前调用，而void postVisit(ASTNode)在调用endVisit()之后调用。实际在规范第三版的实现中，还增加了一个方法boolean preVisit2(ASTNode)，目前该方法的缺省实现是调用preVisit()方法，然后返回true。

类ASTNode提供了方法void accept(ASTVisitor)配合访问者来访问抽象语法树节点，该方法的实现如程序2.2，其中调用的方法accept0()是一个包级访问控制的方法，由每个具体的节点类实现，其实现的模板基本上是，先调用访问者针对该节点编写的visit()方法，在该方法返回true的情况下，访问该节点的所有儿子节点，然后调用访问者针对该节点编写的endVisit()方法。

例如程序2.3给出了类CompilationUnit中accept0()方法的实现。当一个节点的性质仅仅是儿子性质而不是儿子列表性质时，调用类ASTNode提供的void acceptChild()方法来访问该儿子节

程序 2.2 类ASTNode的accept()方法的实现

```
/**
 * Accepts the given visitor on a visit of the current node.
 *
 * @param visitor the visitor object
 * @exception IllegalArgumentException if the visitor is null
 */
public final void accept(ASTVisitor visitor) {
    if (visitor == null) {
        throw new IllegalArgumentException();
    }
    // begin with the generic pre-visit
    if (visitor.preVisit2(this)) {
        // dynamic dispatch to internal method for type-specific
        // visit/endVisit
        accept0(visitor);
    }
    // end with the generic post-visit
    visitor.postVisit(this);
}
```

点，而如果是儿子列表性质，则调用类ASTNode提供的void acceptChildren()方法来访问该儿子列表节点。由于在类ASTNode中可通过节点性质描述符来获得该节点儿子的信息，因此这两个方法的实现可与具体的节点无关，从而放在类类ASTNode中实现，这两个方法本身的实现细节不重要，因此在这里不再给出。

程序 2.3 类CompilationUnit的accept0()方法的实现

```
void accept0(ASTVisitor visitor) {
    boolean visitChildren = visitor.visit(this);
    if (visitChildren) {
        // visit children in normal left to right reading order
        acceptChild(visitor, getPackage());
        acceptChildren(visitor, this.imports);
        acceptChildren(visitor, this.types);
    }
    visitor.endVisit(this);
}
```

从上面可以看到，对于每个抽象语法树节点，当进入该节点时将调用访问者提供的preVisit()方法，然后调用访问者提供的visit()方法，然后在需要的时候按照相同的模式访问该节点的所有儿子节点，然后调用访问者提供的endVisit()方法，最后再调用访问者提供的postVisit()方法。

当使用类ASTParser构建某个Java源程序的抽象语法树之后，假设得到的根节点是root，则调用root.accept(aVisitor)则可从根节点root开始遍历整棵抽象语法树。

为了在遍历中完成某项功能，例如找出所有的名字定义和引用，程序员可编写自己的访问者类，例如编写一个类NameVisitor，这个类需要继承类ASTVisitor，然后实现其中的preVisit()，postVisit()方法，并为每个具体的节点类实现visit()和endVisit()方法。

当然有些具体的节点类可实用类ASTVisitor的缺省实现，例如对于那些肯定与名字定义与使用无关的节点（例如对应文字常量的节点类NullLiteral）在类NameVisitor中可能就无需实现其对应的visit()和endVisit()方法。类ASTVisitor对方法preVisit()和postVisit()的缺省实现都是不做任何事情，对于所有节点类的visit()方法都是不做任何事情而直接返回true，对于所有节点的endVisit()方法也都是不做任何事情。

2.2 符号表

符号表是指一个Java程序中用到的所有名字及其相关属性构成的表，符号表在编译过程起着很大的作用，它通常在语法分析阶段构造，然后会用于语义分析、中间代码生成等许多后续阶段。同样地，为了分析一个程序，我们也需要像编译器内部一样构造一个符号表，以供后续的分析使用。

2.2.1 名字的作用域

根据上一章的分析，每个名字都有它其作用的区域，即该名字的作用域，在不同的作用域里可声明相同的名字，因此要构建符号表，首先需要找出程序中的作用域，所以我们先讨论Java程序中作用域的分析。

Java程序的作用域实际上比较简单，具体来说有以下程序区域可被看作作用域：

(1) **程序包**(Package)：多个Java文件可属于同一个程序包，公有类型声明（类、接口或枚举）和包级访问控制的类型声明都具有包级作用域，即它们可声明在某个包中，并可在这个包中直接使用（意指可使用简单名而不是限定名即可使用）。

(2) **类型声明**(TypeDeclaration)：类、接口或枚举的声明都构成了一个作用域，其成员的作用域则是这个类型声明的整个区域。

(3) **方法体**(MethodBody)：每个方法的方法体都构成了一个作用域，是方法的形式参数及方法中声明的局部变量的作用域；

(4) **块语句**(Block)：Java语言允许在块语句中声明变量，因此某些块语句也是作用域。

对于每个作用域，我们可能需要关心如下属性：

(1) **作用域标识**(Id)：每个作用域应该有唯一的标识。对于类型声明和方法体可采用类型名/方法名+所在源文件名+该作用域的其实源代码行号作为作用域的标识，例如"MyClass@MyFile:116"，其中MyClass是声明的类名，MyFile是Java源程序文件名，如果考虑到不同子目录可能还有同名文件，那么可允许文件名中也包括子目录，使用斜杠分隔，最后的116表示源程序行号，因为可能在同一个文件中定义同名的方法。对于块语句形成的作用域，可统一使用"Block@MyFile:118"这种标识符形式，源代码行号足以区分不同的块。对于程序包，则直接使用包名作为作用域的标识，因为程序包不属于某个文件，但对于每个Java项目来说，程序包名应该是唯一的。

(2) **作用域类型**(Type)：表示该作用域是程序包、类型声明、方法体还是块语句。当然这实际上是一个冗余信息，因为如果按照上面的作用域标识规则，通过作用域标识已经可知道作用域的类型。但作用域标识的编写方法以后可能会变化（例如我们可能选择为作用域随机地生成一个唯一的标识），所以还是需要设置独立的作用域类型属性。

(3) **作用域范围**，我们需要记录作用域所在的源文件名(SourceFile)、作用域的起始行号(StartLine)和终止行号(EndLine)；

(4) **上层作用域**，作用域是可以嵌套的，因此我们需要记录上层作用域。通过这个信息我们也可搜索到嵌套在某个作用中的所有作用域。

2.2.2 名字的定义

如上一章所给出的，Java语言的名字主要有程序包名、类型名、方法名和变量名，程序包名通常在程序源文件的开头声明（定义），类型名的作用域可能是某个程序包，也可能是声明在某个类之中（即嵌套类），而方法名只能声明（定义）在某个类型声明中，变量名可以声明在某个类型声明中（作为这个类型的成员），也可能声明在方法（甚至是方法的块语句）中。

简单地来看，除了程序包名和类型名之外，其他的名字要么是声明为某个类型的成员，要么是声明在方法中（包括方法的形式参数中）而作为局部变量。因此，为了列出某个程序中所定义名字，我们只需要构造两张表：

(1) 类型名表：对于类型名，我们关心它如下属性：(a) 定义该类型名的源代码文件及源代码行号；(b) 类型名字；(c) 种类（及是类、接口还是枚举等）；(c) 该类型名的作用域；(d) 超类名；(e) 所实现的接口名列表；(f) 数据成员和方法成员列表，对于每个成员，我们还要关心定义该成员的源代码文件及源代码行号、成员名字、是否是静态成员、是否是最终成员(final)。对于数据成员，我们还关心该成员的类型，对于方法成员，我们还关心该方法的返回类型、形式参数类型列表。由于我们的目的是做程序分析，因此我们无需关心成员的访问控制。对于方法成员，这里我们也无需关心它的形式参数名，只关心其类型。

根据类型名及源代码文件及行号可唯一地确定一个类型名，而对于成员，同样根据成员名及源代码文件及行号也可唯一地确定。因为合法的Java程序不可能在同一行定义相同的类型名或成员名。

(2) 局部变量名表：对于局部变量（包括方法的形式参数），我们关心它如下属性：(a) 定义该变量名的源代码文件及源代码行号；(b) 种类（是局部变量还是形式参数）；(c) 该变量名的作用域；(d) 该变量的类型。同样，根据源代码文件及源代码行号可唯一地确定一个局部变量名。

2.2.3 名字的使用

为了后续的程序分析工作，也许我们需要构造有关名字的使用信息。名字的使用信息可以帮助我们了解程序的复杂程度，以及每个名字的重要程度，进一步也许我们可以分析名字的使用分布与程序错误的分布之间的关联关系等。

对于每个名字的使用，我们可能关心这样的信息：(1) 所使用的名字；(2) 名字使用点，使用源代码文件名加行号确定，如果一个名字在同一行有多处使用，我们可以在行号后面加.1表示第一次使用，.2表示第二次使用等，例如116.1表示第116行的该名字的第1次出现等；(3) 名字的使用类型，

例如是名字的左值使用、还是右值使用等。有关名字的使用类型我们需要在对更多的Java 源程序做调研的基础上才能进一步确定；(4) 该名字使用所对应的名字定义点（也使用源代码文件名加行号确定），注意，有些名字可能找不到定义（例如在Java API 中定义的String等）。

简单地来说，在获得一个Java项目（即多个Java源程序一起构成的一个软件）中作用域定义、类型定义、局部变量名定义以及名字的使用信息之后，程序员基本上可了解到在Java项目中所使用到的名字的各个方面的信息，对于后续的程序分析有比较重要的作用。有关名字的这些信息应该可以通过遍历Java源程序的抽象语法树获得。

2.3 程序控制流图

程序控制流图描述程序中的控制流，从静态的角度给出了程序运行时控制转移的情况。这一节对程序控制流图的基本概念，以及如何构造程序控制流图进行较为详细的讨论。

在Java语言中，由于方法是不允许嵌套定义的，因此我们可以针对每个类的每个方法构建控制流图，从而这里对于控制流图的讨论也局限于一个方法的方法体的控制流图的定义与构建。

2.3.1 程序的可执行点

从源程序代码的角度看，程序执行的基本单位是语句，但是语句可能是复合语句（块语句），因此我们需要引入程序**可执行点**(executable point)的概念。程序的一个可执行点是指程序中的一个位置，这个位置的语句的执行或表达式的计算可能改变程序的控制流，相应地，程序控制流就是程序可执行点构成的串。

我们可以根据程序语句的类型定义程序可执行点，前面的表2.5给出了Eclipse JDT的抽象语法树节点类`Statement`的子类，根据这些子类我们可对Java程序的语句进行分类并确定其对应的程序可执行点：

(1) 每一条表达式语句、构造方法调用语句、超类构造方法调用语句、断言语句或异常抛出语句都对应一个可执行点，而变量声明语句中的每个变量声明及其初始化对应一个可执行点；

(2) 每一条控制转移语句（`break`, `continue`, `return`语句及标号语句）都对应一个可执行点；

(3) 块语句根据其中的每条语句计算其对应的可执行点；

(4) `if`语句中的表达式计算对应一个可执行点，而真分支和假分支的块语句根据其中的每条语句计算对应的可执行点；

(5) `switch`语句中的表达式计算对应一个可执行点，`switch`的每种情况(以`case`开头的标号也分别对应一个可执行点，每种情况的其他语句根据其中的每条语句计算对应的可执行点；

(6) `do`语句中的`while`后面的表达式计算对应一个可执行点，循环体根据其中的每条语句计算其对应的可执行点；

(7) `while`语句中的表达式计算对应一个可执行点，循环体根据其中的每条语句计算其对应的可执行点；

(8) `for`语句中的初始化、循环条件表达式计算及循环计数器的改变分别对应可执行点，循环根据其中的每条语句计算其对应的可执行点；

(9) 增强型`for`语句中的变量声明对应一个可执行点，循环根据其中的每条语句计算其对应的可执行点；

(10) try语句中的每一个catch中的异常声明对应一个可执行点，而其块语句根据其中的每条语句计算其对应的可执行点；

(11) throw语句对应一个可执行点。

在程序可执行点可定义前导和后继关系。称可执行点s是可执行点t的**直接后继**(direct successor)，对应地，t 称为s的**直接前驱**(direct predecessor)，如果存在程序的运行使得程序在经过可执行点t之后控制流可以到达可执行点s。利用这种关系我们可将程序的可执行点分为两大类：一类可执行点只有唯一的直接后继，而另一类可执行点可以有多个直接后继：

(1) 没有方法调用的表达式语句、控制转移语句（break，continue，return语句及标号语句）、switch中的case标号、for循环中的初始化和计数器改变、throw语句等都只有唯一的直接后继；

(2) if，while，do，for等语句中的表达式（这种表达式在文献中通常称为**谓词**）计算都可能有两个直接后继，相应表达式的值为真有一个直接后继，为假时有另一个直接后继；

(3) 类似地，增强型for语句中的变量声明也有两个直接后继，如果该变量遍历完容器有一个直接后继，没有遍历完进入循环体有另一个直接后继；

(4) try语句及其catch子句和throw语句涉及到异常处理，它与正常的控制流有差别。严格来说，放在try中的每条语句都可能有多个直接后继，根据它所抛出的异常被哪个catch子句捕获而定。而catch中的异常声明也可能有两个不同的直接后继，捕获了相应的异常将执行该catch子句中的块语句，否则则忽略该块语句。有关异常处理的控制流分析需要进一步参考文献[5]；

(5) 因此方法调用本身也隐含地可能有多个直接后继，例如该方法可能抛出异常，甚至可能直接终止程序等。

表 2.7 程序PowerCalculator.java中方法main()的可执行点

标识	可执行点类型	程序源码
4:9	方法调用	`int x = readAInt("Please input x:  ");`
5:9	方法调用	`int y = readAInt("Please input y:  ");`
6:9	常规语句	`int power;`
7:12	谓词	`(y < 0)`
7:20	常规语句	`power = -y;`
8:14	常规语句	`power = y;`
9:9	常规语句	`double z = 1;`
10:15	谓词	`(power != 0)`
11:13	常规语句	`z = z * x;`
12:13	常规语句	`power = power - 1;`
14:12	谓词	`(y < 0)`
14:20	常规语句	`z = 1/z;`
15:9	方法调用	`System.out.println("x^y = " + z);`

由于程序员书写程序的习惯等原因，程序可执行点与源程序行可能没有一一对应关系，例如，通常for语句中的初始化、条件表达式以及计数器的改变等语句都在同一行。因此我们可能需要使用源程序行以及起始列，即“line:column”的形式来标识一个可执行点。

按照上面的讨论，可执行点也可大致分为：方法调用(method call)、常规语句(normal statement)、谓词(predicate)等三大类，其中方法调用是含有方法调用的表达语句或其他构造方法调用语句，而谓词包含分支语句、循环语句中的条件表达式计算，以及断言语句、catch子句中的变量声明，以及增强for语句中的变量声明等，这些都含有为真（或说属于容器、属于所声明的异常等）和为假两个可能的分支，其他的可执行点都可归结为常规语句（都只有唯一的直接后继）。表2.7给出了程序2.4的方法main()的可执行点。

程序 2.4 PowerCalculator.java

```
1  import java.util.Scanner;
2  public class PowerCalculator {
3      public static void main(String[] args) {
4          int x = readAInt("Please input x: ");
5          int y = readAInt("Please input y: ");
6          int power;
7          if (y < 0) power = -y;
8          else power = y;
9          double z = 1;
10         while (power != 0) {
11             z = z * x;
12             power = power - 1;
13         }
14         if (y < 0) z = 1/z;
15         System.out.println("x^y = " + z);
16     }
17
18     static int readAInt(String promptMsg) {
19         Scanner in = new Scanner(System.in);
20         System.out.print(promptMsg);
21         int value = in.nextInt();
22         return value;
23     }
24 }
```

2.3.2 方法的控制流图

确定程序的可执行点之后，我们可定义程序中每个方法的控制流图。

定义 2.3.1 给定程序的方法$\mathcal{M}$，其控制流图是有向图$CFG_{\mathcal{M}} = (V, E)$，其中$V$ =方法$\mathcal{M}$的可执行点构成的集合$\cup \{\textbf{Start}, \textbf{End}\}$，即控制流图的顶点是该方法所有可执行点构成的集合再加上两个特殊顶点**Start**, **End**。图G的两个顶点u, v有有向边$\langle u, v\rangle$当且仅当v是u的后继，而且对方法$\mathcal{M}$的每个入口（可执行点）u都有边$\langle \textbf{Start}, u\rangle$，而对方法$\mathcal{M}$的每个出口（可执行点）$u$也都有边$\langle u, \textbf{End}\rangle$。

注意，所谓方法的入口就是方法的第一条语句所对应的可执行点（如果第一条语句是分支或循

环语句，则是其中的谓词所对应的可执行点），而方法的出口是方法的最后一条语句所对应的可执行点，或者是return语句、throw语句对应的可执行点。

图2.1给出了程序PowerCalculator.java的方法main()的控制流图。

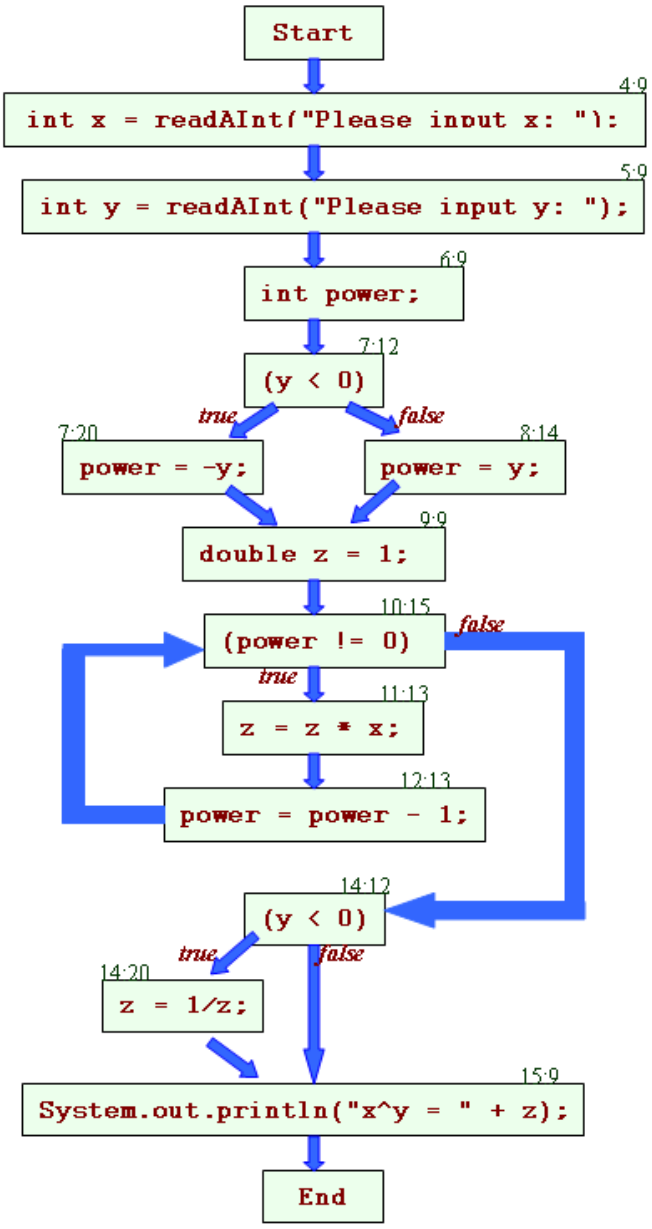


图 2.1 程序PowerCalculator.java的方法main()的控制流图

对于方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}} = (V, E)$，图中的有向路径是顶点$p_1 p_2 \cdots p_n$的序列，使得对任意的$i = 1, n-1$都有$\langle p_i, p_{i+1} \rangle \in E$。若$p_1 =$ **Start**且$p_n =$ **End**，则称该路径为完整路径。显然方法$\mathcal{M}$的运行迹是其控制流图的某条完整路径，但有些完整路径不一定是方法的运行迹，是运行迹的完整路径称为可达完整路径。若控制流图中存在可执行点u到可执行点v的有向路径，则称u是v的前驱，而v是u的后继。

2.3.3 基本块

为了简化程序（方法）的控制流图，通常使用基本块作为控制流图的顶点。

定义 2.3.2 基本块(basic block)是可执行点的一个序列$p_1p_2\cdots p_n$，且满足：(1) 对任意的$i=1,\cdots,n-1$，p_{i+1}是p_i的唯一直接后继，而p_i是p_{i+1}的唯一直接前驱；(2) 如果p_1不是方法的入口，则它存在多个直接前驱或它不是它的直接前驱的唯一直接后继；(3) 如果p_n不是方法的出口，则p_n存在多个直接后继或它不是它的直接后继的唯一直接前驱。

也就是说，基本块是可执行点的一个极长序列，除了第一个和最后一个可执行点外，其它的可执行点都只有唯一的前驱和后继，且第一个可执行点有唯一的后继，而最后一个可执行点有唯一的前驱。或者更简单地说，基本块只有唯一的入口点（第一个可执行点）和出口点（最后一个可执行点），但第一个可执行点可能有多个直接前驱，而最后一个可执行点可能有多个直接后继。

表 2.8 程序PowerCalculator.java中方法main()的基本块

基本块	可执行点序列	备注
1	[4:9, 5:9, 6:9, 7:12]	可执行点7:12有两个直接后继
2	[7:20]	
3	[8:14]	
4	[9:9]	可执行点9:9有两个直接前驱，所以不能归入基本块2或3
5	[10:15]	可执行点10:15有两个直接前驱也有两个直接后继
6	[11:13, 12:13]	
7	[14:12]	可执行点14:12有两个直接后继
8	[14:20]	
9	[15:9]	可执行点15:9有两个直接前驱，因此不能归入基本块8

对于两个基本块$B_1=p_1p_2\cdots p_n$和$B_2=q_1q_2\cdots q_n$，称B_2是B_1的直接后继，如果可执行点q_1是执行点p_n的直接后继。若基本块B_2是B_1的直接后继，则对应地，B_1是B_2的直接前驱。基本块$B=p_1p_2\cdots p_n$称为方法的入口，如果可执行点p_1是方法的入口。基本块$B=p_1p_2\cdots p_n$称为方法的出口，如果p_n是方法的出口。

方法$\mathcal{M}$的以基本块为顶点的控制流图$CFG^B_{\mathcal{M}}=(V,E)$，其中$V$包含方法$\mathcal{M}$的所有基本块以及特殊顶点**Start**和**End**，而对于$u,v\in V$，$\langle u,v\rangle\in E$当且仅当u是v的直接前驱，或者u是**Start**，而v是方法的入口，或者v是**End**，而u是方法的出口。

图2.2给出了程序PowerCalculator.java的方法main()以基本块为顶点的控制流图。不难看出，从图论的角度看，这个控制流图是以可执行点为顶点的控制流图中合并了连续的2度（入度和出度都为1的）顶点而得到有向图。

2.3.4 决定者与后决定者

控制流图中的决定者与后决定者的概念在程序分析和测试中非常有用。首先我们假定方法的控制流图都是常规的(normal)，即对任意的不是**Start**也不是**End**的顶点u，都存在从**Start**到u的路径（否则u对应该方法中不可执行的语句），也存在从u到**End**的路径（否则该方法内部存在死循环而不能退出该方法的执行）。

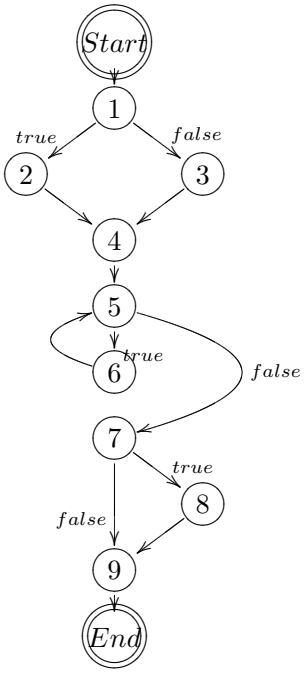


图 2.2 程序PowerCalculator.java的方法main()的控制流图

定义 2.3.3 给定方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}}=(V,E)$。设$u,v\in V$是两个不同的顶点，如果对任意从**Start**到v的路径都包含u，则称u**决定**(dominate)v，或者说u是v的**决定者**(dominator)，记为$dom(u,v)$。

类似地，若对任意的从v到**End**的路径都包含u，则称u**后决定**(post-dominate)v，或者说u是v的**后决定者**(post-dominator)，记为$pdom(u,v)$。

注意，上述定义排除了一个顶点决定或后决定该顶点自己。由上述定义立即可得顶点**Start**决定所有（从**Start**可达的）顶点，而顶点**End**后决定所有（可达**End**的）顶点。

引理 2.3.1 显然，决定和后决定关系都是传递关系，即若u决定/后决定v，且v决定/后决定w，则u决定/后决定w。而且，决定和后决定关系都是反对称关系，即若u决定/后决定v，则v不可能再决定/后决定u。 □

定义 2.3.4 给定方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}}=(V,E)$，$u,v\in V$。称u是v的**直接决定者**(immediate dominator)，如果u是v的决定者，而且不存在顶点w是的u决定w且w决定v。

类似地，称u是v的**直接后决定者**(immediate post-dominator)，如果u是v的后决定者，而且不存在顶点w是的u后决定w且w后决定v。

引理 2.3.2 显然，若u是v的直接决定者，则从**Start**到v的每条路径，u和v之间的顶点（不含u,v）都不是v的决定者，也即u是从**Start**到v的某条路径上的“最后一个v的决定者”（实际上也是每条从**Start**到v的路径上的“最后一个v的决定者”。

对应地，若u是v的直接后决定者，则从v到**End**的每条路径，v和u之间的顶点（不含u,v）都不是v的后决定者，也即u是从v到**End**的某条路径上的“第一个v的后决定者”（实际上也是每条从v到**End**的路径上的“第一个v的决定者”。 □

引理 2.3.3 给定方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}} = (V, E)$。设$u \neq$ **Start**，则u有唯一的直接决定者。类似地，若$u \neq$ **End**，则u有唯一的直接后决定者。

证明 当$u \neq$ **Start**，显然u至少有一个决定者（**Start**必定是它的决定者之一），而且若v和w都是u的决定者，则必有v决定w，或者w决定v。对于后决定关系也有类似的性质。 □

因为除**Start**之外的每个顶点都有唯一的直接决定者，因此顶点之间的直接决定关系的关系图是一棵树，其根是**Start**。类似地，顶点之间的直接后决定关系的关系图也是一棵树，其根是**End**。有关如何根据控制流图生成决定者和后决定者关系的算法可参见文献[6]。

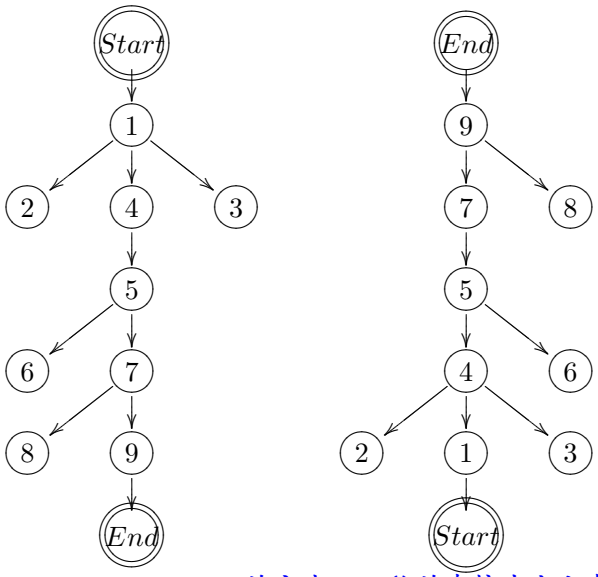


图 2.3 程序PowerCalculator.java的方法main()的直接决定和直接后决定关系图

图2.3给出了程序PowerCalculator.java的方法main()的直接决定和直接后决定关系图，这些图都以基本块为顶点。不难看到，若$B = p_1 p_2 \cdots p_n$是基本块，则对任意的$i = 1, \cdots, n-1$，p_i是p_{i+1}的直接决定者，且p_{i+1}是p_i的直接后决定者，而且对任意的$i, j \in \{1, \cdots, n\}$，若$i < j$，则$p_i$决定$p_j$，而$p_j$后决定$p_i$。

2.3.5 控制流图的可约简性

给定方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}} = (V, E)$，它的一个强连通子图称为该控制流图的一个**环**(loop)。注意，根据图论知识，一个有向图是强连通图当且仅当它存在一个经过所有顶点的有向回路。在程序的控制流图中，循环语句会产生环。

对于控制流图的环C，边$\langle u, v\rangle$称为环C的**入边**(entry edge)，如果$u \notin C$但$v \in C$，这时v称为环C的**入口顶点**(entry node)。对应地，$\langle u, v\rangle$称为环C的**出边**(entry edge)，如果$u \in C$但$v \notin C$，这时u称为环C的**出口顶点**(exit node)。

但是有向图中的一个强连通子图不一定都对应现代结构化程序设计语言中的循环语句。实际上，结构化的循环语句都对应控制流图中的自然环(natural loop)。**自然环**是指控制流图中存在唯一的入口顶点的环，这个入口顶点称为自然环的**头节点**(head node)。由于头节点是自然环的唯一入口顶点，因此头节点决定自然环中所有其他顶点。

自然环中必存在回边(backward edge)。**回边**是指控制流图中目标顶点决定源顶点的边。由于自然环的头节点决定环中所有其他顶点，而且由于环是强连通图，因此其他顶点必存在到头节点的有向路径，因此也存在以头节点为目标顶点的有向边，这条边就是回边，自然环中回边的源顶点称为该环的**尾节点**。

实际上，控制流图中每条回边$\langle u,h\rangle$都对应一个自然环，该自然环由顶点u、顶点h及那些不经过顶点h可到达顶点u的所有顶点构成。显然顶点h决定所有不经过顶点h可到达顶点u的所有顶点，因为若w是这样的一个顶点，那么所有从**Start**到w的路径是从**Start**到u的路径的一部分，而h决定u，且h不属于从w到u的路径，因此h属于从**Start**到w的每条路径，也即h决定w。由于这样的顶点都可到达u，而u可达h，h又可达所有的这些顶点，因此顶点u、顶点h及那些不经过顶点h可到达顶点u的所有顶点导出的子图是强连通的，因此这个子图是自然环。

图2.4给出了带break或continue的循环的控制流图，左边给出了代码示意图，中间给出了其代码中语句$S3$后面是语句break时的控制流图，而右边给出了其代码中语句$S3$后面是语句continue时的控制流图。

```
S1;
while (C1) {
    S2;
    if (C2) {
        S3;
        break/continue;
    }
    S4;
}
S5;
```

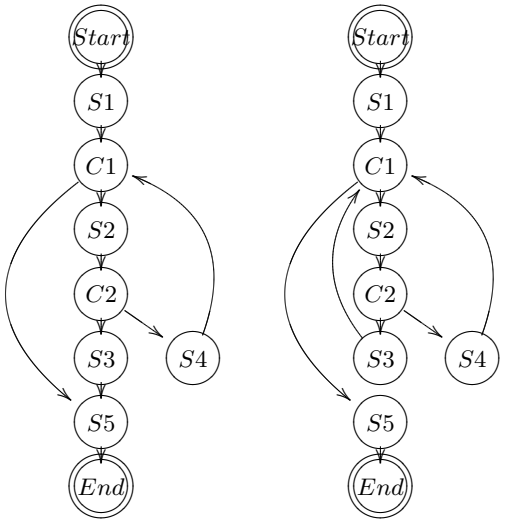


图 2.4 带break/continue结构循环的控制流图

可以看到，带break的循环语句的控制流图只有一个环（$C1 \to S2 \to C2 \to S4 \to C1$），其中$C1$既是入口顶点，也是出口顶点（正常的循环都是循环表达式计算为入口和出口顶点），而边$\langle S4,C1\rangle$是回边，$S4$是尾节点，这个环只有一个入口顶点$C1$，因此是自然环，而且是回边$\langle S4,C1\rangle$对应的自然环。。

而带continue的循环语句的控制流图严格来说有三个环：$C1 \to S2 \to C2 \to S3 \to C1$，$C1 \to S2 \to C2 \to S4 \to C1$，以及$C1,S2,C2,S3,S4$这五个顶点构成的强连通子图，这些环都是自然环，$C1$都是它们唯一的入口顶点，而$S3,S4$分别是前两个环的尾节点，$\langle S3,C1\rangle,\langle S4,C1\rangle$分别是它们的回边，而这两个自然环也分别是这两条回边的自然环。由$C1,S2,C2,S3,S4$这五个顶点构成的自然环不是某条回边对应的自然环，但它是前面两个自然环的并。

定义 2.3.5 只含有自然环的控制流图称为**可约简**(reducible)的控制流图。

我们看到结构化程序只会得到自然环，从而都是可约简的控制流图。可约简的控制流图之所以称为可约简的，是因为这样的控制流图通过两个简单的变换总可变换为一个节点，具体的含义可参

考文献[7]，由于这两个变换对我们来说不重要，所以在这里不做介绍。

可约简的控制流图有如下重要性质：

引理 2.3.4 控制流图$CFG = (V, E)$是可约简的当且仅当将其中所有回边删除之后得到的是没有环的有向图。

证明 ($\Longrightarrow$)：设控制流图是可约简的，如果将其中的所有回边删除之后还有环，设这个环的一个入口顶点是u，显然这个环中有以u为目标顶点的边$\langle w, u\rangle$，由于$\langle w, u\rangle$不是回边，即u不决定w，从而存在从**Start**到w的不经过u的路径（注意控制流图在删除所有回边之后，顶点**Start**仍可达到所有其他顶点），从而这条路径上也必存在这个环的另一个入口顶点，因此这个环不是自然环，这与原来的控制流图只含有自然环矛盾!

($\Longleftarrow$)：类似地，如果控制流图删除回边之后得到的是没有环的有向图，那么加上回边之后得到的环必然是自然环，因为若处回边的目标之外该环还有其他入口顶点，那么回边的目标顶点不可能决定其源顶点。 □

一个基本的不可约简的控制流图具有图2.5的形式，其中顶点2和3构成了一个环，但它有两个入口，因此不是自然环，顶点2不决定3，顶点3也不决定顶点2，它们之间的边也都不是回边。显然，在结构化的程序设计语言中无法写出对应这样的控制流图的程序，只有在非结构化程序中使用`goto`语句可得到这类程序。

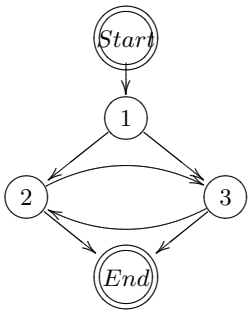


图 2.5 不可约简控制流图的基本形式

可约简的控制流图的重要性在于许多程序分析算法只能针对这种控制流图，而且那些能处理不可约简控制流图的程序分析算法在处理不可约简控制流图时的运行效率也会比处理可约简控制流图时要低很多。

一个可约简控制流图的**深度**(depth)定义为各条无环路径上回边数目中的最大值，多数优化的程序分析算法的运行效率与该控制流图的深度线性相关。可以证明，这个深度永远不会大于直观上循环嵌套的深度。

对于可约简控制流图中的两个自然环，除非它们具有相同的头节点（像带`continue`循环语句的控制流图中的两个环），否则它们要么是分离的，要么一个完全嵌套在另一个中（即一个环的头节点在另外一个环中，即被另外一个环的头节点所决定）。

2.4 程序依赖图

程序中各个成份之间的依赖关系分析是程序分析的基础之一，程序成份之间的依赖分析结果可用于优化程序或者将程序并行化。程序依赖图是表示一个方法内的各个成份之间的数据依赖和控制

依赖关系的图形方法，在程序分析中，特别是程序的过程内分析中被广泛使用。

2.4.1 控制依赖

给定方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}}=(V,E)$，我们假定其中的每个顶点至多有两个直接后继（即每个顶点的出度小于等于2）。我们将出度为2的顶点称为控制流图中的谓词（顶点）(predicate)，且假定以谓词为源点的边都使用了`true/T`或`false/F`标记。

定义 2.4.1 *给定方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}}=(V,E)$，及两个顶点$u,v\in V$。称v**控制依赖于***(control dependent on)u，*如果：(1) 存在从u到v的长度大于等于1的有向路径P，且v是P中除u,v之外所有顶点的后决定者；(2) v不是u的后决定者。*

注意，若$u=v$，则上述定义中的条件(2)平凡地满足（前面定义的后决定者关系是反自反关系），因此条件(1)中要求存在u到v的长度大于1的有向路径，这既排除了任意顶点都控制依赖于自己的这种平凡情形，又可将在循环时循环入口控制依赖于自己的情况包括在内。而条件(1)当控制流图中有边$\langle u,v\rangle$时平凡地满足，因此当v不是u的后决定者时总有v控制依赖于u。

从上述定义还可以看到，若v控制依赖于u，则u必然是谓词（出度为2的顶点），而且存在一条从u到**End**的路径，v不属于该路径。因为若u出度为1，则它的直接后继顶点是它的后决定者，从而不可能满足条件(1)，而v不是u的后决定者意味着u有不包含v的到**End**的路径。实际上，v控制依赖于u的直观含义是，**v是否被执行取决于u的执行结果**。

文献[8]给出了一个找出控制流图中所有控制依赖关系的方法。该方法进一步假定在方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}}=(V,E)$中增加一个虚拟入口谓词**Entry**，它有一条标记为`true`的有向边指向顶点**Start**，且有一条标记为`false`的有向边指向顶点**End**。该谓词用来模拟该方法是否被外界调用而运行。

令$S\subseteq E$是控制流图中所有那些目标顶点不是源顶点的后决定者的边构成的集合。对S中的每条边$\langle u,v\rangle$，设w是在后决定者树中u,v的最近公共祖先，即w要么就是u（v不是u的后决定者，但u可能是v的后决定者），要么同时是u和v的后决定者，且对任意同时是u和v的其他后决定者t，w是t的后决定者。实际上，文献[8]证明了：

引理 2.4.1 *w要么是u，要么是u在后决定树中的父亲顶点，即w是u的直接后决定者。*

证明 设t是u在后决定树中的父亲顶点，因为v不是u的后决定者，因此$t\neq v$。首先可证明t后决定v，因为否则的话存在一条从v到**End**的路径不经过t，但是将边$\langle u,v\rangle$加入到该路径中得到一条从u到**End**的路径，而且也不经过t，从而t不是u的后决定者，这与t是u的直接后决定者！因此t也是v的后决定者。从而由于u的直接后决定者也是v的后决定者，因此在后决定树中，u和v的最近公共祖先要么是u，要么是u的直接后决定者。 □

这样对S中的每条边$\langle u,v\rangle$，设w是在后决定者树中u,v的最近公共祖先，则在后决定者树中从w到v的唯一路径上所有的顶点，包括v，且当$w=u$时也包括u，但当$w\neq u$时不包括w都控制依赖于u。这是因为：

(1) 对w到v的路径上的所有不是v也不是u的顶点t，由于t是v的后决定者，因此存在从v到t的有向路径，而且t是这条路径上所有顶点的后决定者，加上边$\langle u,v\rangle$就得到u到t的一条长度大于等于1的路径，且t是这条路径上除u,t外的所有顶点的后决定者，而且t不是u的后决定者（因为要

么$u=w$是t的后决定者，要么u不在w到t在后决定树的路径中而且w是u的直接后决定者，因此t也不可能是u的后决定者），所以根据控制依赖的定义，t控制依赖于u。

(2) 当$u=w$时，u自己也控制依赖于u，因为这时u是v的后决定者，从而存在v到u的路径，而且u是这条路径上所有顶点的后决定者，加上边$\langle u,v\rangle$就得到了u到u的长度大于等于1的路径，因此u控制依赖于u自己。

文献[8]指出，考察S中的所有边即可得到控制流图中的所有控制依赖关系，也即有：

引理 2.4.2 给定方法$\mathcal{M}$的控制流图$CFG_{\mathcal{M}}=(V,E)$，及两个顶点$u,v\in V$。v控制依赖于u当且仅当存在边$\langle u,t\rangle$，t不是u的后决定者，设w是t和u在后决定者树中的最近公共祖先，则v或者是w到t在后决定树中的唯一路径上除t,w之外的顶点，或者$v=w=u$。

证明 根据上面的讨论，我们只要证明当v控制依赖于u时存在满足条件的边$\langle u,t\rangle$。根据定义，当v控制依赖于u时存在从u到v的长度大于等于1的路径P，而且v是这条路径上除u,v外所有顶点的后决定者，因此有边$\langle u,t\rangle\in P$，若$t=v$，则u是v的后决定者，从而引理成立。若$t\neq v$，则v是t的后决定者，则设w是u,t在后决定树中最近的公共祖先，则由于v是t的后决定者，而且不是u的后决定者，这就必有w是v的后决定者，因为显然v不可能是w的后决定者（否则v就会也是u的后决定者），而若w也不是v的后决定者的话，由v和w都是t的后决定者会得到t有两个直接后决定者，这与直接后决定者的唯一性矛盾！这样，v就在w到t的后决定树路径中。 □

图2.6给出了程序PowerCalculator.java的方法main()的控制流图（左边）和后决定树（右边），为了方便与下面要讨论的数据依赖，这里给出了可执行点为顶点的控制流图和后决定树，而且根据文献[8]的方法添加了**Entry**顶点。

在图2.6中控制流图的所有边中目标顶点不是源顶点的后决定者的边集

$$S=\{\langle\textbf{Entry},\textbf{Start}\rangle,\langle 7{:}12,7{:}20\rangle,\langle 7{:}12,8{:}14\rangle,\langle 10{:}15,11{:}13\rangle,\langle 14{:}12,14{:}20\rangle\}$$

表2.9给出了这些边所识别的控制依赖。

可以看到，可执行点**Start**, 4:9, 5:9, 6:9, 7:12, 9:9, 10:15, 14:12, 15:9都依赖顶点**Entry**，其含义是只要该方法被调用，那么这些可执行点必然被执行（从而如果这些可执行点不存在数据依赖的话就可以被并行化）。表2.9的“标记”列给出了“识别的顶点”这一列的可执行点在“依赖的顶点”这一列（都是谓词）为真还是为假的时候执行，例如可执行点7:20控制依赖于7:12，且在可执行点7:12取值为真时必然执行。注意，表2.9的第5行表明可执行点10:15（循环条件表达式计算）自己控制依赖于自己，这是因为只有在上一次循环条件表达式的值为真时，这次的循环条件表达式才会被计算。

表 2.9 程序PowerCalculator.java的方法main()中的控制依赖

检查的边	识别的顶点	依赖的顶点	标记
⟨**Entry**, **Start**⟩	**Start**, 4:9, 5:9, 6:9, 7:12, 9:9, 10:15, 14:12, 15:9	**Entry**	T
⟨7:12, 7:20⟩	7:20	7:12	T
⟨7:12, 8:14⟩	8:14	7:12	F
⟨10:15, 11:13⟩	10:15, 11:13, 12:13	10:15	T
⟨14:12, 14:20⟩	14:20	14:12	T

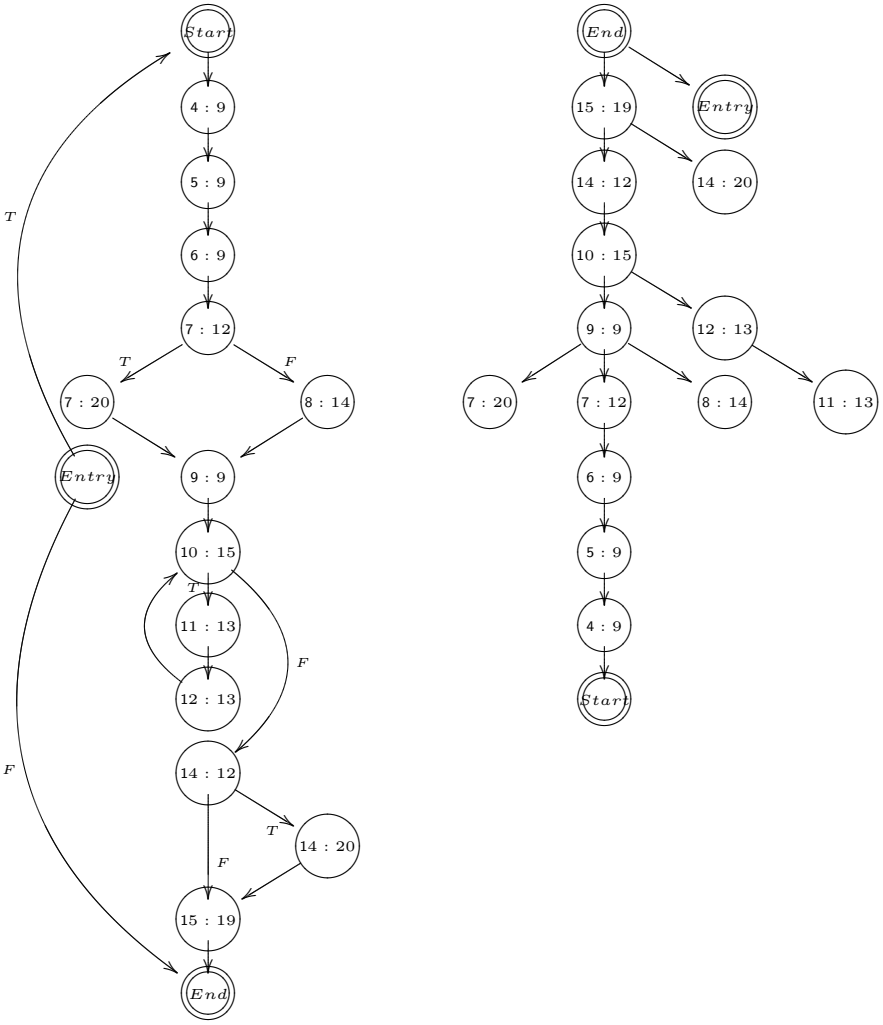


图 2.6 程序PowerCalculator.java的方法main()的控制流图和后决定树

为了更好地理解控制依赖以及程序依赖图中控制依赖子图的构造方法，我们再考察文献[9]中给出的一个例子。图2.7的左边给出该例子中的控制流图，在这个控制流图中，存在出度大于2的顶点，不过这对控制依赖的计算不会产生实质的影响。我们假定每条边都有标记（而不仅仅是出度为2的顶点的出边有T或F的标记），图中给出了其中少数边的标记（这些标记在后面会用到）。这个控制流图还将顶点**Entry**和**Start**合二为一了，顶点**Start**直接有一条边到**End**。注意，这个控制流图是可约简的，在实际的程序中，A点相当于一个多分支语句（例如`switch`语句）的分支点，而C, D点指向**End**的边相当于使用`return`语句直接返回。

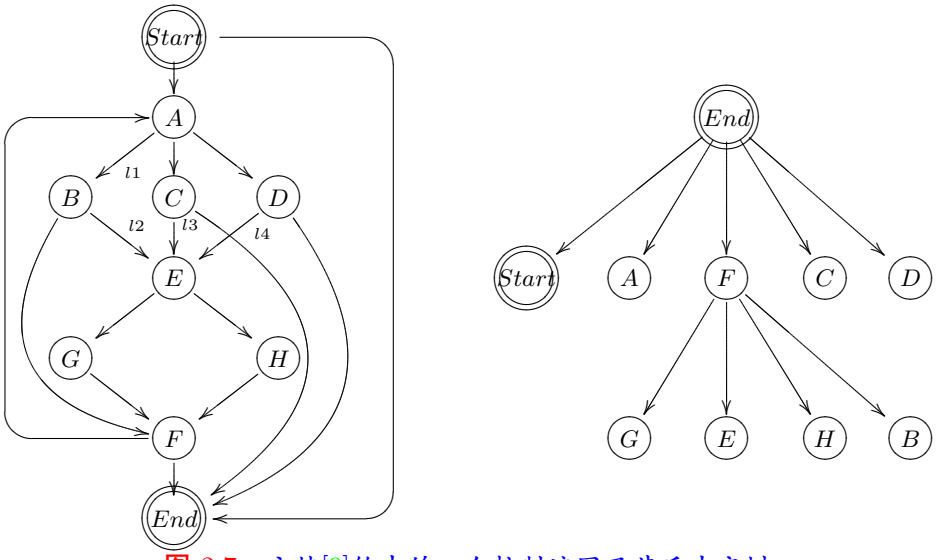


图 2.7 文献[9]给出的一个控制流图及其后决定树

图2.7的右边给出了该控制流图的后决定树。表2.10给出了该控制流图中存在的控制依赖，注意标记列实际上等于所检查的边的标记。

表 2.10 文献[9]给出的控制流图中的控制依赖

检查的边	识别的顶点	依赖的顶点	标记
$\langle$**Start**, $A\rangle$	A	**Start**	
$\langle A, B\rangle$	B, F	A	$l1$
$\langle A, C\rangle$	C	A	
$\langle A, D\rangle$	D	A	
$\langle B, E\rangle$	E	B	$l2$
$\langle C, E\rangle$	E, F	C	$l3$
$\langle D, E\rangle$	E, F	D	$l4$
$\langle E, G\rangle$	G	E	
$\langle E, H\rangle$	H	E	
$\langle F, A\rangle$	A	F	

2.4.2 控制依赖子图的区域节点

上述对控制依赖的定义可直接用于构造程序依赖图的控制依赖子图，但在多数情况下，将程序

依赖图中的顶点根据其是否共享相同的控制条件而划分成不同区域会非常有用，例如同一区域下的顶点所代表的语句如果没有之间不存在数据依赖则可并行执行。控制流图中区域的划分使用使用控制依赖子图中的区域顶点来表示。

控制依赖子图中的**区域顶点**(region node)是一种新的顶点（即不对应程序可执行点或基本块，也即不属于控制流图中的顶点），用于表示某些控制条件(control conditions)的一个集合。这里的控制条件可理解为“⟨顶点, 标号⟩”，即顶点和标号构成的有序对，通常这里的顶点都是谓词顶点，而标号用来标记谓词的取值。因此这里控制条件的直观含义就是谓词及其取值的组合，或者对应控制流图来说就是对应其中某个分支点的出边。

某个区域顶点的所有儿子节点可认为属于同一区域，在该区域顶点所对应的条件成立时这些儿子节点所对应的程序代码就会被执行。区域顶点的儿子节点可以是对应程序语句或谓词的顶点，也可以是另外的区域顶点。谓词顶点的儿子节点只能是区域顶点，而不能是语句顶点或谓词顶点，而且谓词的一个取值仅仅只有一个儿子节点（到达该儿子的边使用该取值标记），因为区域顶点的作用就是概括具有相同控制条件的一组节点。

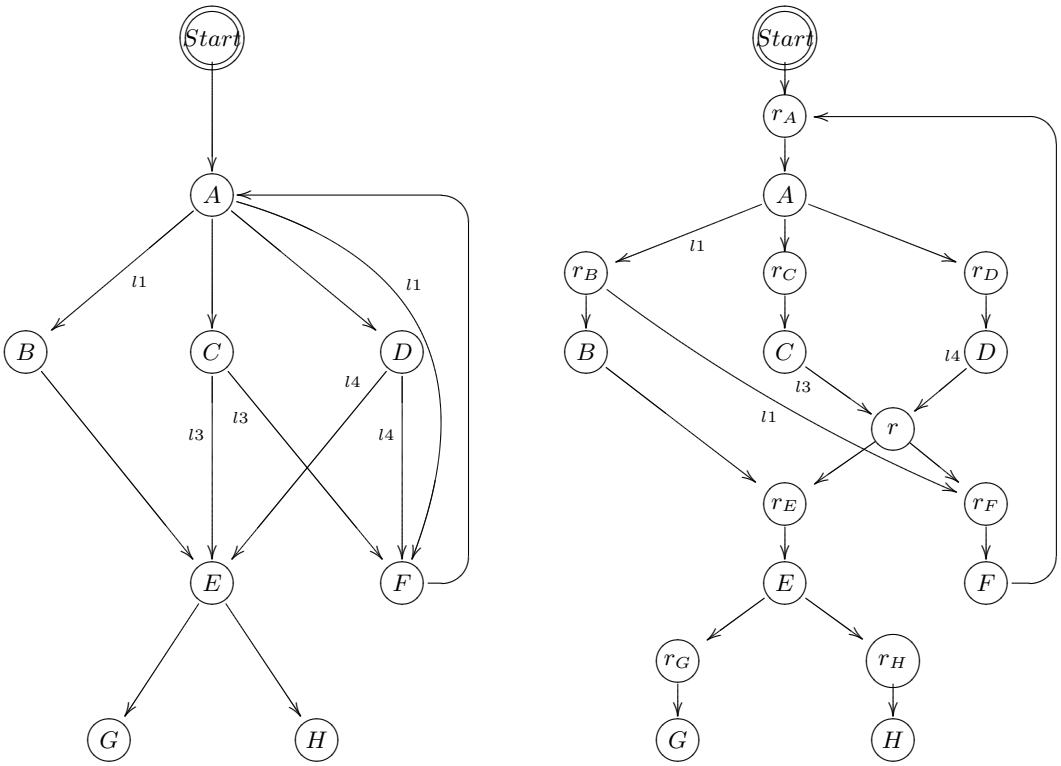


图 2.8 文献[9]中控制流图的控制依赖及区域顶点的插入

图2.8给出了文献[9]中控制流图的控制依赖及区域顶点的插入，我们所给出的带区域顶点的控制依赖图与文献[9]中的图稍有差别，原图有一条从顶点A到区域顶点r_F的带标记$l1$的边，但我们根据文献[8]的插入区域顶点的方法，发现这条边应该是从区域顶点r_B到r_F，因为区域顶点r_B概括了控制条件$\langle A, l1\rangle$，这是顶点B和顶点F都依赖的控制条件。

我们发现不同的文献对于如何插入区域顶点有一些差别，例如文献[8]与文献[9]就有一些差别，这增加了我们理解区域顶点及其作用的难度。这里我们介绍文献[8]所给出的插入区域顶点的基本方法。该方法分为两步，第一步插入的区域顶点用于概括一组控制条件，并将依赖这一组控制条件的

节点组织在一起。为了实现这一步，后序遍历后决定者树，对每个顶点N，确定其所依赖的控制条件集CD：

(1) 若CD对应的区域顶点R已经存在，则在原来的控制依赖图中删除以N为目标顶点的边，而增加一条从R到N的边，然后考虑后序遍历后决定者树的下一顶点；

(2) 若CD对应的区域顶点不存在，则创建一个新的区域顶点R（并记录其对应的控制条件），同样删除原来的控制依赖图以N为目标顶点的边，而代之以从R到N的一条边，并从这些边的每个源顶点引一条边到R。然后考虑N在后决定树中的每个儿子节点M，设其所依赖的控制条件为CD'，其对应的区域顶点为R'（按照构造，儿子节点所依赖控制条件的区域顶点应该已经存在），处理下面两种情况：

(a) 若$CD' \cap CD = CD$，即M所依赖的控制条件包含了N所依赖的控制条件，则删除对应CD中每个控制条件的以R'为目标的边，代之以从R到R'的边；

(b) 若$CD' \cap CD = CD'$，即N所依赖的控制条件包含了M所依赖的控制条件，则删除对应CD'中每个控制条件的以R为目标的边，代之以从R'到R的边。

注意，若这两种情况都不成立，则直接考虑后序遍历后决定者树的下一顶点。不难看到，经过这一步，每个谓词或语句顶点都只可能是某一个区域顶点的儿子，而不会是多个区域顶点的儿子，也不可能是其他谓词或语句顶点的儿子。对于两个谓词或语句顶点，它们具有相同的父亲（区域）节点，当且仅当它们所依赖的控制条件完全相同。

每个区域顶点可能是（多个）谓词顶点或区域顶点的儿子（但不可能是语句顶点的儿子，因为语句顶点不表示控制条件，不会是依赖的目标），这时该区域顶点所代表的控制条件是它的这些父亲节点所代表的控制条件的并集（注意，谓词顶点及其出边的标记表示一个控制条件），而这意味着在程序运行时，只要父亲节点所代表的控制条件之一成立，该区域顶点的儿子节点中的谓词顶点及语句顶点就一定会被执行。

我们发现，虽然不同的文献在插入区域顶点或构造带区域顶点的控制依赖图的方法会稍有吧同，但所得到的控制依赖图都具有上述两个基本性质：(1) 两个谓词或语句顶点有相同的父亲（区域）顶点，当且仅当它们所依赖的控制条件完全相同；(2) 每个区域顶点所代表的控制条件是它的父亲节点所代表的控制条件的并集。

在图2.8中，各个区域顶点所对应的控制条件集如下：

$$
\begin{aligned}
r_A &\text{ 对应 } \{\langle \mathbf{Start}, - \rangle, \langle F, - \rangle\} \\
r_B &\text{ 对应 } \{\langle A, l_1 \rangle\} \\
r_C &\text{ 对应 } \{\langle A, - \rangle\} \\
r_D &\text{ 对应 } \{\langle A, - \rangle\} \\
r &\text{ 对应 } \{\langle C, l3 \rangle, \langle D, l4 \rangle\} \\
r_E &\text{ 对应 } \{\langle B, - \rangle, \langle C, l3 \rangle, \langle D, l4 \rangle\} \\
r_F &\text{ 对应 } \{\langle A, l1 \rangle, \langle C, l3 \rangle, \langle D, l4 \rangle\} \\
r_G &\text{ 对应 } \{\langle E, - \rangle\} \\
r_H &\text{ 对应 } \{\langle E, - \rangle\}
\end{aligned}
$$

在图2.8中，有些控制依赖边没有标记，因此上面以$-$表示，但假定这些边都有互不相同的标记。在

这个图中没有两个谓词或语句顶点有相同的控制依赖条件集，但是我们看到，每个区域顶点所代表的控制条件是它的父亲节点所代表的控制条件的并集，例如$r_E = \{\langle B, -\rangle\} \cup r, r_F = r_B \cup r$等。

文献[8]给出的方法的第二步确保每个谓词及其取值所代表的控制条件只有一个儿子节点，因此若某谓词的某个取值有多个儿子，则创建一个区域顶点，其父亲节点是该谓词（且边的标记为该取值），而该区域顶点的儿子节点则是原先那些儿子节点。

图2.8中的区域顶点r可以说根据这一点而创建的，但是如果按照文献[8]给出的方法，则应该分别为控制条件$\langle C, l3\rangle$和$\langle D, l4\rangle$创建区域顶点r_1, r_2，前者以C为父亲节点，而后者以D为父亲节点，并且都有到E和F的边。但是文献[9]的方法则将这两个区域顶点合并成了一个区域顶点。

根据文献[8]给出的插入区域顶点的方法，我们可得到程序`PowerCalculator.java`的方法`main()`的（带区域顶点）的控制依赖图，如图2.9。

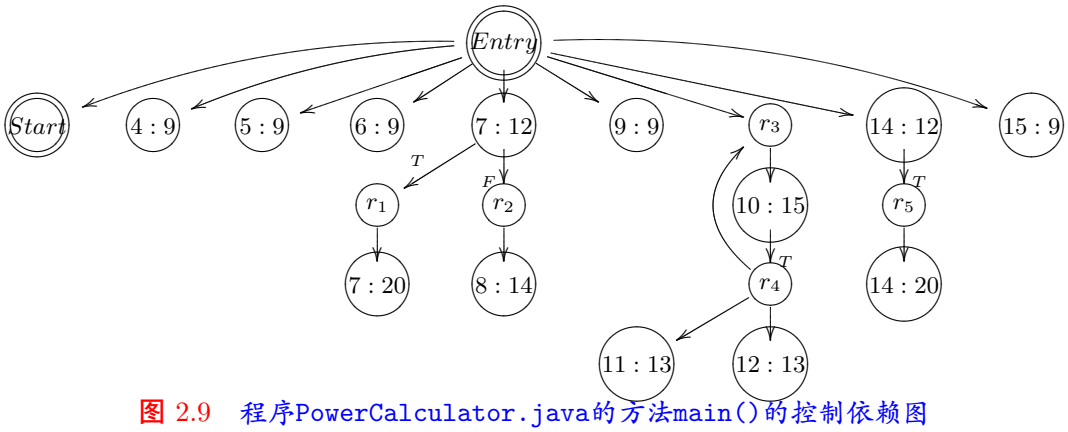


图 2.9 程序PowerCalculator.java的方法main()的控制依赖图

实际上，对于结构化程序，文献[10]指出，其控制依赖图的区域顶点可分为**Entry**, “if-clause”, “else-clause”, “while header”, “while body”, “while exit”, “summary”以及“label”几种，其中“if-clause”和“else-clause”分别对应`if`语句的条件表达式为真和为假的情况，例如图2.9中的r_1和r_2，而“while header”对应循环语句导入，例如图2.9中的r_3，而“while body”概括循环语句的循环体，例如图2.9中的r_4，而“while exit”, “label”用于处理带`continue`, `break`的循环语句以及标号。“summary”类的区域则用于纯粹概括相同的控制条件，如图2.8中的区域顶点r等。文献[10]还给出了根据结构化程序的语法结构（即以抽象语法树为输入）如何生成控制依赖图。

2.4.3 数据流基本概念

程序的一个可执行点称为变量x的**定值**(definition)（**点**），如果变量x在这个点获得值。变量的有些定值点是明确的，例如使用赋值对该变量进行赋值，或使用输入语句输入该变量的值等，但有些并不明确，例如将该变量作为实际参数传递给某个子程序，该子程序是否对该变量有副作用很难确定，又例如对于指针或引用变量，由于别名的存在而难以确定某个可执行点到底对哪些变量进行了定值。为说明数据流相关基本概念，这一节只涉及到赋值语句这种对变量的明确定值的情况，不考虑指针或引用变量，也假定所有的子程序对于实际参数都没有副作用。

称变量x的一个定值（点）p**到达**(reach)程序可执行点q，如果在该程序的控制流图中存在从p到q的有向路径且在这个路径中不存在x的其他定值（点），这条路径也称为变量x的**干净定值路径**(definition-clear path)。注意，这样的路径可能在控制流图中存在，但不一定是可执行的路径（即

不一定存在输入使得程序恰好执行这条路径），因此也有的文献要求p在**Start**到q的有向路径上才成为定值p到达q。

图2.10给出了一个简单的控制流图（省略了节点**Start**和**End**）。该控制流图只有五条语句（每条语句对应一个可执行点），分为四个基本块。该控制流图中只有两个变量I和J，表2.11给出了它们的定值点以及每个定值点所到达的可执行点。

表 2.11 图2.10控制流图中的定值点与到达点

变量	定值点	到达点	到达的基本块
x	1	2, 3	$B2$
	3	1, 4, 5	$B1, B3, B4$
y	2	1, 3, 4	$B1, B2, B3$
	4	5	$B4$
	5	**End**	

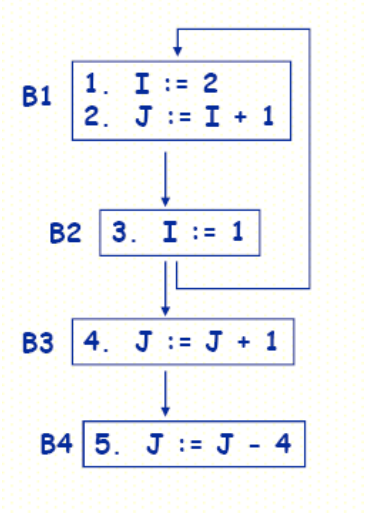


图 2.10 一个用于说明数据流基本概念的简单控制流图

通常我们更关心变量的一个定值点所能到达的基本块。说变量的定值点到达某个基本块是指该定值点可到达该基本块的入口点。表2.11也给出了每个定值点能到达的基本块。

对于每个基本块B，将能到达该基本块（入口点）的所有定值点构成的集合记为$In[B]$（或更明确地，$RchIn[B]$，因为在数据流分析中$In[B]$还可能用于表示在基本块B入口点能访问的其他信息），而将能到达该基本块出口点的所有定值点记为$Out[B]$（或更明确地，$RchOut[B]$）。为了计算所有基本块的这两个集合，可先收集一个基本块B的局部信息$Gen[B]$和$Kill[B]$，其中：

$$
\begin{aligned}
Gen[B] &= \{\text{在}B\text{中且能到达}B\text{出口的定值点}\} \\
Kill[d] &= \{d' \mid \text{定值点}d'\text{与定值点}d\text{所定值的变量相同且 } d' \neq d\} \\
Kill[B] &= \bigcup_{d \in B} Kill[d]
\end{aligned}
$$

实际上，$Gen[B]$就是在基本块B中创建（或说生成）的定值点集，而$Kill[B]$就是那些被基本块B中定值点可能“杀死”的定值点。我们说一个定值点可能“杀死”另一个定值点，如果这两个定值点对相同的变量进行定值。基本块B中的某个定值点可能既属于$Gen[B]$又属于$Kill[B]$，例如，设基本块B由如下两条语句构成：

$$d1 : I = J + 1;$$
$$d2 : I = I + K;$$

则$Gen[B] = \{d2\}$，因为$d1$不能到达B的出口，而$d1 \in Kill[B]$，因为$d1 \in Kill[d2]$，且$d2 \in Kill[B]$，因为$d2 \in Kill[d1]$。

使用如下数据流迭代方程可求解每个基本块B的$In[B]$和$Out[B]$：

$$In[B] = \bigcup_{P\text{是}B\text{的控制流图前驱节点}} Out[P]$$
$$Out[B] = Gen[B] \cup (In[B] - Kill[B])$$

其中，对所有基本块B，$In[B]$的初值为空集$\varnothing$，而$Out[B]$的初值为$Gen[B]$，然后使用以上方程根据上一次迭代得到的每个基本块B的$In[B]$和$Out[B]$值计算当次每个基本块B的$In[B]$和$Out[B]$，直到每个基本块B的$In[B]$和$Out[B]$值都不再发生变化。

表 2.12 图2.10控制流图中基本块的定值到达分析

	局部信息		初始值		迭代1		迭代2	
基本块	$Gen[-]$	$Kill[-]$	$In[-]$	$Out[-]$	$In[-]$	$Out[-]$	$In[-]$	$Out[-]$
$B1$	$\{1,2\}$	$\{3,4,5\}$	$\varnothing$	$\{1,2\}$	$\{3\}$	$\{1,2\}$	$\{2,3\}$	$\{1,2\}$
$B2$	$\{3\}$	$\{1\}$	$\varnothing$	$\{3\}$	$\{1,2\}$	$\{2,3\}$	$\{1,2\}$	$\{2,3\}$
$B3$	$\{4\}$	$\{2,5\}$	$\varnothing$	$\{4\}$	$\{2,3\}$	$\{3,4\}$	$\{2,3\}$	$\{3,4\}$
$B4$	$\{5\}$	$\{2,4\}$	$\varnothing$	$\{5\}$	$\{3,4\}$	$\{3,5\}$	$\{3,4\}$	$\{3,5\}$

表2.12给出了图2.10控制流图中每个基本块B的$In[B]$和$Out[B]$的计算，第三次迭代不再改变任何基本块的$In[-]$和$Out[-]$值，也即与第二次迭代的计算结果相同（因此表中没有给出第三次迭代结果），从而计算结束。在该控制流图中，基本块$B1$的前驱是$B2$，而$B2$的前驱是$B1$，$B3$的前驱是$B2$，$B4$的前驱是$B3$。在迭代1中计算$In[B3]$时，使用的已经是在迭代1中计算得到的$Out[B2]$值，而不是$Out[B2]$的初值，因此迭代1中$In[B3]$的结果是$\{2,3\}$，而非$\{3\}$。也即每次迭代的计算可利用当此迭代已经计算好的基本块的信息，这样可加速计算的收敛，从而提高计算的效率。

表2.12表明到达基本块$B1$的入口点（即执行语句1之前的点）的定值点是语句2和语句3给出的定值点，也即语句2和语句3对其变量的赋值将会影响基本块$B1$中的语句，而达到基本块$B1$的出口点（即执行语句2之后的点）的定值点是语句1和语句2给出的定值点，即语句1和语句2对其变量的赋值将会影响到基本块$B1$的后继节点中的语句。其他基本块的$In[-]$和$Out[-]$也可类似理解。

2.4.4 数据依赖

变量的一个使用（点）(use)是指程序的一个可执行点，在这个执行点对变量的值进行了一次读

取。变量的使用点又可细分为**计算使用（点）**(computation use, c-use)和**谓词使用（点）**(predicate use, p-use)。计算使用点是指对变量值的读取直接影响了（表达式）的计算结果或程序的输出结果，而谓词使用点是指变量值的读取直接影响了程序控制流的转向，从而间接影响程序的计算结果。

变量v的一个**定值-使用对**(definition-use pair)是二元组(D,U)，其中D是v的一个定值点，而U是v的一个使用点，且D可到达U，也即存在控制流图中的一条从D到P的有向路径，且在这条路径中D没有被“杀死”（即在这个路径中没有v的其他定值点），实际上也就是说，从D到U有一条v的干净定值路径(definition-clear path)。

基于前面的定值到达分析很容易确定每个基本块所包含的定值-使用对。对于基本块B，经过定值达到分析之后我们可得到达到B的入口的所有定值点集$In[B]$，然后对于基本块B中某个变量v的使用点U，若基本块B的入口点到U没有v的定值点，则对$In[B]$中每个对v的定值点D都生成一个定值-使用对(D,U)。

注意，这里关心的是基本块之间的定值-使用对，即在某个基本块的定值点D是否能到达另一个基本块的使用点U，而没有给出基本块内的定值-使用对。例如，设基本块B由如下两条语句构成：

$$d1 \ : I = J + 1;$$
$$d2 \ : I = I + K;$$

变量I的定值点$d1$和使用点$d2$有一个基本块内的定值-使用对$(d1,d2)$。使用$In[B]$中的信息并不能得到这样的定值-使用对，因为定值点$d1$不一定属于$In[B]$。如果要得到基本块内的定植-使用对，我们可将分析的粒度定位每个可执行点，而不是基本块，以可执行点为粒度的定值到达分析与以基本块为粒度的定值到达分析可使用类似的方法。

程序中每个变量的每个定值-使用对都给出了程序可执行点之间的一个数据依赖关系，且这种数据依赖关系被称为**流依赖**(flow dependence)。除了流依赖之外，还有输出依赖(output dependence)、反依赖(anti-dependence)等数据依赖关系。程序的可执行点p输出依赖q，如果存在p到q的变量v的干净定值路径，且p和q都是变量v的定值（点）。程序的可执行点p反依赖q，如果p是变量v的使用（点），而q是变量v的定值（点），并存在p到q的有向路径，且该路径除q之外没有v的其他定值点。在实际应用中，通常流依赖是最重要的数据依赖关系，因此在程序依赖图中往往只给出表示流依赖的边。

下面我们仍使用程序`PowerCalculator.java`的方法`main()`作为例子进一步说明定值-使用对的计算，以及程序依赖图中的流依赖的表示。图2.11给出了该方法的控制流图以及基本块、可执行点和语句信息。

表2.13给出程序`PowerCalculator.java`的方法`main()`中的变量定值和使用。我们使用“变量名:n@可执行点”的形式给出变量的定值点和使用点，其中n表示该变量名在相应可执行点的第n次出现，因此该形式使用名为“变量名”的变量在“可执行点”的第n次出现来标识变量的定值点和使用点。

表 2.13 程序`PowerCalculator.java`的方法`main()`的变量定值和使用

变量	定值	使用	变量	定值	使用
x	4	11	power	6, 7, 8, 1@12	10, 2@12
y	5	7:12, 7:20, 8, 14	z	9, 1@11, 1@14	2@11, 2@14, 15

基本块	可执行点	语句
1	4:9	int x = readAInt("Please Input x:");
	5:9	int y = readAInt("Please Input y:");
	6:9	int power;
	7:12	if (y < 0)
2	7:20	power = -y;
3	8:14	power = y;
4	9:9	double z = 1;
5	10:15	while (power != 0)
6	11:13	z = z * x;
	12:13	power = power - 1;
7	14:12	if (y < 0)
8	14:20	z = 1/z;
9	15:9	System.out.println("x^y = " + z);

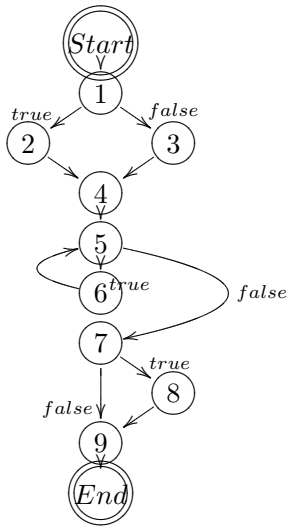


图 2.11 程序PowerCalculator.java的方法main()的基本信息

我们前面使用“行号:起始列”的形式标识可执行点，在给出变量定值点和使用点时，我们通常省略其中可执行点的起始列，因为这时有可执行点行的信息已经足够区分不同的定值点和使用点。其次当变量在该执行点只有一次出现时，我们省略定值点和使用点中的“:n”。而表2.13进一步省略了定值点和使用点中的变量名，因为该表的第一列已经给出了变量名。因此表中power的定值点“1@12”表示变量power在第12行的第1次出现，而使用点“2@12”表示它在第12行的第2次出现等。

表 2.14 程序PowerCalculator.java的方法main()的定值到达分析初值

可执行点	前驱节点	$Gen[-]$	$Kill[-]$	$In[-]$	$Out[-]$
4:9	∅	{x@4}	∅	∅	{x@4}
5:9	{4:9}	{y@5}	∅	∅	{y@5}
6:9	{5:9}	{power@6}	{power@7, power@8, power:1@12}	∅	{power@6}
7:12	{6:9}	∅	∅	∅	∅
7:20	{7:12}	{power@7}	{power@6, power@8, power:1@12}	∅	{power@7}
8:14	{7:12}	{power@8}	{power@6, power@7, power:1@12}	∅	{power@8}
9:9	{7:20,8:14}	{z@9}	{z:1@11, z:1@14}	∅	{z@9}
10:15	{9:9,12:13}	∅	∅	∅	∅
11:13	{10:15}	{z:1@11}	{z@9, z:1@14}	∅	{z:1@11}
12:13	{11:13}	{power:1@12}	{power@6, power@7, power@8}	∅	{power:1@12}
14:12	{10:15}	∅	∅	∅	∅
14:20	{14:12}	{z:1@14}	{z@9, z:1@11}	∅	{z:1@14}
15:9	{14:12,14:20}	∅	∅	∅	∅

我们以语句为控制流图中的节点来进行定值到达分析。实际上，由于上述程序的每个基本块内都没有同一变量的两次定值或两次使用，因此基于基本块的分析和基本语句（可执行点）的分析最后得到的定值-使用对完全相同。因为前面控制依赖是以语句（可执行点）为单位进行分析（实际上，控制依赖也完全可以以基本块为单位进行分析，结果类似），这里对数据依赖的分析也以可执行点

为基本单位。

表2.14给出程序PowerCalculator.java的方法main()的定值到达分析初值，其中最后两列给出了对应每个可执行点的$In[-]$和$Out[-]$集的初值。

表 2.15 程序PowerCalculator.java的方法main()的定值到达分析第一次迭代

执行点	$In[-]$	$Out[-]$
4:9	$\varnothing$	{x@4}
5:9	{x@4}	{x@4, y@5}$(=A)$
6:9	A	{power@6} $\cup A$
7:12	{power@6} $\cup A$	{power@6} $\cup A$
7:20	{power@6} $\cup A$	{power@7} $\cup A$
8:14	{power@6} $\cup A$	{power@8} $\cup A$
9:9	{power@7, power@8}$(=B) \cup A$	{z@9} $\cup B \cup A$
10:15	{power:1@12, z@9} $\cup B \cup A$	{power:1@12, z@9} $\cup B \cup A$
11:13	{power:1@12, z@9} $\cup B \cup A$	{z:1@11, power:1@12} $\cup B \cup A$
12:13	{z:1@11, power:1@12} $\cup B \cup A$	{z:1@11, power:1@12} $\cup A$
14:12	{power:1@12, z@9} $\cup B \cup A$	{power:1@12, z@9} $\cup B \cup A$
14:20	{power:1@12, z@9} $\cup B \cup A$	{z:1@14, power:1@12} $\cup B \cup A$
15:9	{z@9, z:1@14, power:1@12} $\cup B \cup A$	{z@9, z:1@14, power:1@12} $\cup B \cup A$

表2.15给出程序PowerCalculator.java的方法main()的定值到达分析的第一次迭代计算结果，而表2.16给出了第二次迭代计算结果，与上一次迭代计算结果的唯一差别是变量z在第11行的定值传播到了循环条件判断处（可执行点10:15），从而进一步传播到了循环体中的语句。不难看到，第二次迭代计算结果就是最终的定值到达分析结果。在计算时为了简洁起见我们令集合$A = \{$x@4, y@5$\}$，而$B = \{$power@7, power@8$\}$。

表 2.16 程序PowerCalculator.java的方法main()的定值到达分析第二次迭代

执行点	$In[-]$	$Out[-]$
4:9	$\varnothing$	{x@4}
5:9	{x@4}	{x@4, y@5}$(=A)$
6:9	A	{power@6} $\cup A$
7:12	{power@6} $\cup A$	{power@6} $\cup A$
7:20	{power@6} $\cup A$	{power@7} $\cup A$
8:14	{power@6} $\cup A$	{power@8} $\cup A$
9:9	{power@7, power@8}$(=B) \cup A$	{z@9} $\cup B \cup A$
10:15	{z:1@11, power:1@12, z@9} $\cup B \cup A$	{z:1@11, power:1@12, z@9} $\cup B \cup A$
11:13	{z:1@11, power:1@12, z@9} $\cup B \cup A$	{z:1@11, power:1@12} $\cup B \cup A$
12:13	{z:1@11, power:1@12} $\cup B \cup A$	{z:1@11, power:1@12} $\cup A$
14:12	{z:1@11, power:1@12, z@9} $\cup B \cup A$	{z:1@11, power:1@12, z@9} $\cup B \cup A$
14:20	{z:1@11, power:1@12, z@9} $\cup B \cup A$	{z:1@14, power:1@12} $\cup B \cup A$
15:9	{z:1@11, z@9, z:1@14, power:1@12} $\cup B \cup A$	{z:1@11, z@9, z:1@14, power:1@12} $\cup B \cup A$

根据定值到达分析的结果，我们可得到程序PowerCalculator.java的方法main()的所有定值-使用对，如表2.17所示，表2.17按照变量将定值-使用对进行了分类。最后，在程序控制依赖图的基础上，加入表示数据依赖的边我们则可得到程序程序PowerCalculator.java的方法main()的程序依赖图，如图2.12所示。

表 2.17 程序PowerCalculator.java的方法main()的定值-使用对

变量	定值-使用对
x	$\langle 4, 11\rangle$
y	$\langle 5, 7:12\rangle, \langle 5, 7:20\rangle, \langle 5, 8\rangle, \langle 5, 11\rangle$
power	$\langle 7, 10\rangle, \langle 8, 10\rangle, \langle 1@12, 10\rangle, \langle 7, 2@12\rangle, \langle 8, 2@12\rangle, \langle 1@12, 2@12\rangle$
z	$\langle 9, 2@11\rangle, \langle 1@11, 2@11\rangle, \langle 9, 2@14\rangle, \langle 1@11, 2@14\rangle, \langle 9, 15\rangle, \langle 1@11, 15\rangle, \langle 1@14, 15\rangle$

在程序依赖图中，数据依赖使用虚线给出的有向边表示，例如从可执行点4:9到可执行点11:13的（虚线）有向边表示可执行点4:9到可执行点11:13的数据依赖关系（具体来说是流依赖关系），相关的变量是x，因此这个边使用x标记，其他变量所产生的流依赖关系也如图2.12中的虚线有向边及其标记。

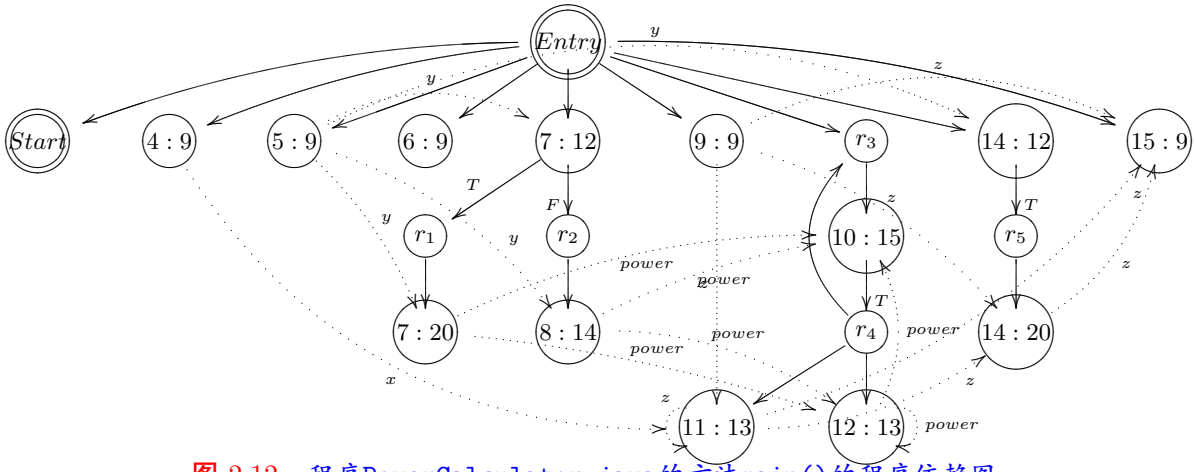


图 2.12 程序PowerCalculator.java的方法main()的程序依赖图

2.5 系统依赖图

上一节给出的一个方法内可执行点之间的控制依赖和数据依赖，但在面向对象软件中更关注的是方法、类、编译单元乃至程序包之间的依赖关系。我们把表示这些层次依赖关系的图统称为系统依赖图。方法、类、编译单元和程序包之间的依赖关系在不同的文献中可能有不同定义，这里基于我们的理解将这些依赖关系予以明确定义。

2.5.1 参与依赖关系的程序实体

一个Java软件由多个程序包构成，每个程序包又有多个编译单元，每个编译单元中可声明多个类，类有初始化块、字段、方法和内部类，方法内部还可能声明局部类和使用匿名类。在这些程序元

素中，程序包、编译单元、类和方法都可看做是程序中的一片区域，称为作用域(scope)，在这个区域中，对其他程序元素的使用构成了程序元素之间的依赖关系。

因此，我们认为程序包、编译单元、类和方法之间都存在依赖关系，而且我们可允许考虑不同层次之间的依赖关系，例如一个程序包是否依赖于一个方法等，这些更有助于对程序的理解。

2.5.2 程序元素之间的使用关系

依赖关系基于程序元素之间的使用关系，因此这里先考虑程序元素之间可能的使用。这里所谓的使用，是指在程序源代码的某个地方对（程序元素的）名字的引用。基于我们对Java程序语法的理解，有：

1. 在编译单元导入声明语句中会导入某个类，这是这个编译单元对这个类的使用；
2. 在声明类时，会声明类继承(extends)一个类或实现(implements)一个接口，这是待声明的类对所继承的类或实现的接口的使用；
3. 在声明类的字段时，会声明字段的类型，这是这个类的字段对这个类型的使用；字段声明有初始化表达式，这个表达式中可能使用其他字段、变量或调用方法；
4. 在类的初始化块中，声明变量时会用到类型，在语句中会使用字段、变量或调用方法；
5. 在声明类的方法时，会声明返回类型、参数类型或抛出异常，这是这个类的方法对返回类型、参数类型或抛出异常（这些类型的使用；在类的方法体中声明变量会用到类型，在语句中会使用字段、变量或调用方法。

在程序元素之间的这些使用中，类型、字段和变量的使用都容易确定所使用的类型、字段或变量是在程序中哪里声明（定义）的类型、字段或变量，但对于方法的使用（调用）则情况比较复杂。在Java程序中，方法调用语句的基本形式是，

表达式.方法名(参数列表)

我们设这个方法调用语句在类A的初始化块或类A的方法M中，而其中的表达式可能是一个类名（后面调用的是静态方法），也可能结果是一个对象O，我们设这个类名是B，或这个声明对象O的类型是B。程序编译时确定这个方法调用到底是调用哪个方法（即绑定这个语句中的方法名），是在类型B或其祖先类中匹配具有相同方法名，以及相同参数（类型）的方法，假定匹配到的方法为N，方法N声明在类C中（C可能就是B，但也可能是B的祖先类）。因此这个方法调用可概括为：

类型A（的方法M）通过类型B（的对象O）（静态直接）调用类型C声明的方法N

这里说“直接”，是因为这个调用语句直接出现在类型A中，而不是类型A的方法M调用方法P，方法P又调用方法N这种间接方式。说“静态”调用N，是因为我们说在编译时匹配到的方法为N。实际上，对象O声明时的类型是B，但在运行时可能引用的是类型B的后代类D的对象，从而在运行时调用的可能是类型D中重定义N的方法N'，这种情况，我们称为类型A（的方法M）多态直接调用方法N'。

我们下面可能省略“通过……”及“静态直接”（即不特别说明都是指静态直接调用），而直接说**类型A调用类型C 声明的方法N**，或**方法M调用类型C声明的方法N**，或更简单地说**类型A调用方法N**，及**方法M调用方法N**。

在确定这些“使用”，特别是方法调用后，我们分程序元素讨论它们之间的依赖关系。依赖关系是有向关系，下面基于源节点分类讨论。

2.5.3 方法对程序元素的依赖

这一节明确方法对其他程序元素之间的依赖关系，也即方法是源节点，而其他程序元素是目标节点。我们定义：

1. 说方法M依赖方法N，如果方法M（静态直接）调用方法N，记为$M \Longrightarrow_{\text{call}} N$，或直接简记为$M \Longrightarrow N$；

2. 说方法定义M依赖类型（接口或类）T，记为$M \Longrightarrow T$，如果下列情况之一存在：

2.1 方法M的基调使用了类型T，记为$M \Longrightarrow_{\text{sig}} T$，即下面三种情况之一：

2.1.1 方法M的返回类型是T；

2.1.2 方法M的某个参数类型是T；或

2.1.3 方法M抛出类型为T的异常。

2.2 方法M调用了类型T声明的方法，记为$M \Longrightarrow_{\text{call}} T$；

2.3 方法M的方法体使用了类型T，记为$M \Longrightarrow_{\text{body}} T$，即下面二种情况之一：

2.3.1 方法M使用了类型T声明的字段；

2.3.2 方法M使用了类型T声明局部变量。

3. 说方法M依赖编译单元C，记为$M \Longrightarrow C$，如果存在在C声明的类型T，使得方法M依赖类型T。编译单元C声明的类型包括顶层类型、内部类或局部类，但不包括枚举类型和匿名类。

4. 说方法M依赖程序包P，记为$M \Longrightarrow P$，如果存在属于P的编译单元C，使得方法定义M依赖C。

不难看到，根据上述定义，若设类型T声明了方法N，而且类型T在编译单元C中声明，编译单元C属于程序包P，则对任意方法M：$M \Longrightarrow N$蕴含$M \Longrightarrow T$、$M \Longrightarrow T$蕴含$M \Longrightarrow C$，而$M \Longrightarrow C$蕴含$M \Longrightarrow P$。

2.5.4 类型对程序元素的依赖

这一节定义的都是以类型T为源节点的依赖关系。这里说的类型T是指在某个编译单元或类类型中定义的类类型或接口类型（包括内部类和局部类，但不包括枚举类型和匿名类）。

1. 说类型T依赖方法M，记为$T \Longrightarrow M$，如果下列情况之一存在：

1.1 类型T的初始化块有静态直接调用方法M的语句；

1.2 类型T的某个字段声明初始化表达式中静态直接调用方法M；

1.3 类型T声明的某个方法N，N静态直接调用方法M。

2. 说类型T依赖类型S，记为$T \Longrightarrow S$，如果下列情况之一存在：

2.1 类型T使用了类型S的字段，记为$T \Longrightarrow_{\text{field}} S$，即下列情况之一存在：

2.1.1 类型T的初始化块中使用了类型S声明的字段；

2.1.2 类型T的某个字段声明初始化表达式中使用了类型S声明的字段；

2.1.3 类型T的某个方法方法体中使用了类型S声明的字段；

2.2 类型T使用了（调用了）类型S的方法，记为$T \Longrightarrow_{\text{call}} S$，即下列情况之一存在：

2.2.1 类型T的初始化块中调用了类型S声明的方法；

2.2.2 类型T的某个字段声明初始化表达式中调用了类型S声明的方法；

2.2.3 类型T的某个方法调用了类型S声明的方法；

2.3 类型T有类型S的成员，即类型T声明了某个类型为S的字段，记为$T \Longrightarrow_{\text{member}} S$；

2.4 类型T有对类型S的其他使用方式，记为$T \Longrightarrow_{\text{other}} S$，即有下列情况之一存在：

2.4.1 类型T的初始化块中声明了类型为S的局部变量；

2.4.2 类型T声明的某个方法的基调使用了类型S，即这个方法的返回类型是S，某个参数的类型是S，或者抛出类型为S的异常，即存在类型T声明的方法M，有$M \Longrightarrow_{\text{sig}} S$；

2.4.3 类型T声明的某个方法的方法体声明了类型为S的局部变量。

3. 说类型T依赖编译单元C，记为$T \Longrightarrow C$，如果存在在C声明的类型S，使得类型T依赖类型S。编译单元C中声明的类型包括顶层类型、内部类或局部类，但不包括枚举类型和匿名类。

4. 说类型T依赖程序包P，记为$T \Longrightarrow P$，如果存在属于P的编译单元C，使得类型T依赖C。

设类型T声明了方法M，类型S声明了方法N，而且类型S在编译单元C中声明，编译单元C属于程序包P，则根据上述定义有：

(1) $M \Longrightarrow N$蕴含$T \Longrightarrow N$，注意$M \Longrightarrow N$也蕴含$M \Longrightarrow S$；

(2) $T \Longrightarrow N$蕴含$T \Longrightarrow S$，$T \Longrightarrow S$蕴含$T \Longrightarrow C$，$T \Longrightarrow C$蕴含$T \Longrightarrow P$。

(3) $M \Longrightarrow S$蕴含$T \Longrightarrow S$，而且$M \Longrightarrow_{\text{sig}} S$蕴含$T \Longrightarrow_{\text{other}} S$。

进一步有：$T \Longrightarrow_{\text{call}} S$当且仅当，存在类型$S$声明的某个$N$，使得$T \Longrightarrow N$。

2.5.5 编译单元对程序元素的依赖

这一节定义的都是以编译单元C为源节点的依赖关系。

1. 说编译单元C依赖方法M，记为$C \Longrightarrow M$，如果存在编译单元C声明的类型T，$T \Longrightarrow M$。编译单元C声明的类型包括顶层类型、内部类或局部类，但不包括枚举类型和匿名类。

2. 说编译单元C依赖类型定义S，记为$C \Longrightarrow S$，如果存在编译单元C声明的类型T，$T \Longrightarrow S$。

3. 说编译单元C依赖编译单元B，记为$C \Longrightarrow B$，如果存在编译单元C声明的类型T，并存在编译单元B声明的类型S，$T \Longrightarrow S$。

4. 说编译单元C依赖程序包P，记为$C \Longrightarrow P$，如果存在属于程序包P的编译单元B，$C \Longrightarrow B$。

2.5.6 程序包对程序元素的依赖

这一节定义的都是以程序包P为源节点的依赖关系。

1. 说程序包P依赖方法M，记为$P \Longrightarrow M$，如果存在属于程序包P的编译单元C，$C \Longrightarrow M$。

2. 说程序包P依赖类型S，记为$P \Longrightarrow S$，如果存在属于程序包P的编译单元C，$C \Longrightarrow S$。

3. 说程序包P依赖编译单元B，记为$P \Longrightarrow B$，如果存在属于程序包P的编译单元C，$C \Longrightarrow B$。

4. 说程序包P依赖程序包Q，记为$P \Longrightarrow Q$，如果存在属于程序包P的编译单元C，$C \Longrightarrow Q$。

2.5.7 程序元素间依赖关系分类

在上面定义的依赖关系中，需要注意几点：

(1) 编译单元的导入类型声明没有用于定义依赖关系。

(2) 声明类时的继承(extends)和实现(implements)声明没有用于定义依赖关系，因此如果类A扩展类B（即类A是类B的子类），我们没有称类A依赖类B，因为在程序分析中，继承关系通常需要单独处理。

(3) 内部类、局部类参与依赖关系，但是与他们所在的类或方法间没有必然的依赖关系，也即当一个内部类B声明在类A内部时不意味着类A依赖类B，局部类B声明在方法M内部是也不意味着方法M依赖类M。

从上面的定义可以看到，程序包、编译单元对其他程序元素的依赖关系都是基于方法、类之间的依赖关系定义，因此最基本的依赖关系是方法对方法、方法对类型、类型对方法和类型对类型之间的依赖关系。

上面定义了种类繁多的依赖关系，但在程序理解中，我们可能只关心其中的一部分依赖关系：

(1) 我们将基于$M \Longrightarrow N$、$M \Longrightarrow_{\text{call}} T$、$T \Longrightarrow M$和$T \Longrightarrow_{\text{call}} S$四种基本依赖关系所导出的程序包、编译单元、类型、方法之间的依赖关系称为**基于方法调用的依赖关系**，这里M, N是方法，而T, S是类型。

(2) 我们将基于$T \Longrightarrow_{\text{member}} S$所导出的程序包、编译单元、类型之间的依赖关系称为基于**基于组成成员的依赖关系**，这里T, S是类型。基于组成成员的依赖关系只能在类型以上层次定义。

(3) 我们将基于$M \Longrightarrow N$、$M \Longrightarrow_{\text{call}} T$、$T \Longrightarrow M$、$T \Longrightarrow_{\text{call}} S$和$T \Longrightarrow_{\text{member}} S$五种基本依赖关系所导出的程序包、编译单元、类型、方法之间的依赖关系称为**强依赖关系**。

我们在生成一个Java软件系统的依赖图时应该支持只生成基于方法调用的依赖关系、或只生成基于组成成员的依赖关系，或者只生成强依赖关系。注意到，在基于方法调用的依赖关系图中，如果所有节点都是方法，则就是这些方法之间的静态调用关系图。

第三章　程序分析基础理论

这一章的内容来自于Michael I. Schwartzbach的讲义“Lecture Notes on Static Analysis”。

3.1　引言

程序静态分析(static analysis)的目的在于回答有关程序性质的问题，这些性质可能针对整个程序，例如: (1) 这个程序终止吗? (2) 在运行时，程序占用的堆（内存）会多大? (3) 程序可能的输出会是什么? 也有可能针对程序源代码中具体的点，例如:

(1) 变量x是否总是具有相同的值?

(2) 变量x的值会在后面被读取吗?

(3) 指针p可能为空值(null)吗?

(4) 指针p可能指向哪些变量?

(5) 变量x在被读取之前是否已经初始化?

(6) 整数变量x的值是否总是正的?

(7) 整数变量x的值的上下界(lower and upper bound on the value)是多少?

(8) 在程序的哪一点，变量x被赋予了当前的值?

(9) 指针p和q指向是堆中不相交结构(disjoint structures)吗?

非形式化地，1953年给出的Rice定理给出了上述关于程序行为的问题都是不可判定的(undecidable)的一个一般结论。这不难理解，因为，例如存在一个通用的分析器可判定程序中的一个变量是否取某个常量值，那么就可利用这个分析器来判定停机问题：

$$\texttt{x} = 17;\ \ \texttt{if (TM(j))}\ \texttt{x} = 18;$$

对于上述程序片段，变量x取常量值当且仅当第j个图灵机停机，因此如果该分析能判定该程序中变量x取常量值，那么它能判定（任意的）图灵机是否停机。

虽然这个结果令人沮丧，但是在实际上我们并不关注性质的可判定性，而是试图解决一些像使得程序运行更快或发现程序缺陷的实际问题。解决的方案都是要给出一些近似(approximative)答案，但又足够精确使得能满足应用。

通常这种近似都是保守的(conservative)，意味着错误都基于我们所需要的应用而倾向于一边(meaning that all errors lean to the same side, which is determined by our intended application)。例如，对于判定变量是否具有常量值而言，如果我们期望的应用是执行常量传播，那么应该只在变量真的具有常量值时才回答“是”，而在变量可能或不可能具有常量值时都回答“否”。当然，平凡的解决方案是都回答“否”，因此我们面临的工程挑战是在获得合理的性能前提下尽可能回答“是”。

考虑另外一个问题：指针p会指向哪些变量？如果我们期望的应用是使用变量x替换*p以减少解引用操作，那么分析可能应该只在p确定必然(certainly must)指向x时才回答"&x"，否则当p不指向x或不能确定时都必须回答"?"。但如果我们期望的应用(intended application)是确定*p（占用内存）的最大尺寸，那么分析必须返回一个尽可能大的集合{&x, &y, &z, ···}，以确保能包含所有可能的目标。

通常，所有的优化应用都需要保守的近似。如果给出错误的信息，那么优化可能是不合理(unsound)的且可能改变程序的语义。相反，如果只给出平凡的信息，那么优化却无从做起。

近似答案斗鱼发现程序中的缺陷也很有帮助，通常这可看做是程序验证的弱化形式。例如，考虑C语言含有指针的程序，可能有空指针解引用(dereferences)、悬挂指针(dangling pointers)、内存泄漏(leaking memory)以及非期望别名(unintended aliases)。标准的基于类型检测的编译技术通过很难发现这类指针错误，而即使只有关于空值和指针指向目标的近似答案也可发现有关指针的很多错误。

3.2 例子语言

这里给出一个称为TIP的例子语言，它含有绩效的语法，但是所包含的结构已经足以使得静态分析有趣和富有挑战性。

(1) 表达式。基本的表达式都指称整数值：

$$
\begin{array}{lll}
E & \longrightarrow (intconst) & \text{// 整数常量} \\
& \longrightarrow id & \text{// 标识符} \\
& \longrightarrow E + E \mid E - E \mid E * E \mid E/E \mid E > E \mid E == E & \text{// 运算符} \\
& \longrightarrow (E) & \text{// 园括号} \\
& \longrightarrow \texttt{input} & \text{// 输入值}
\end{array}
$$

这里input作为表达式含义是从输入流(input stream)读入一个整数。比较运算符(>和==)的结果以0表示假，以1表示真。指针表达式将在后面引入。

(2) 语句。简单语句的语法如下：

$$
\begin{array}{lll}
S & \longrightarrow (id) = E; & \text{// 赋值} \\
& \longrightarrow \texttt{output}\ E & \text{// 输出} \\
& \longrightarrow SS & \text{// 顺序结构} \\
& \longrightarrow \texttt{if}\ (E)\ \{\ S\ \} & \text{// 分支结构} \\
& \longrightarrow \texttt{if}\ (E)\ \{\ S\ \}\ \texttt{else}\ \{\ S\ \} & \text{// 分支结构} \\
& \longrightarrow \texttt{while}\ (E)\ \{\ S\ \} & \text{// 循环结构} \\
& \longrightarrow \texttt{var}\ id_1, \cdots, id_n & \text{// 变量声明}
\end{array}
$$

在条件表达时中，0值被认为是假，而其他值都认为是真。output将一个表达式的整数值输出到输出流。语句var 声明一组没有初始化的变量。

(3) 函数。函数有任意数量的参数，并且返回一个单一值(single value)：

$$F \longrightarrow id(id, \cdots, id)\ \{\texttt{var}\ id, \cdots, id; S\texttt{return}\ E;\}$$

函数调用也是一种表达式：

$$E \longrightarrow id(E, \cdots, E)$$

(4) 指针。最后，为允许使用动态内存，引入堆上面的指针：

$$\begin{aligned} E &\longrightarrow \&id && \text{// 指针引用} \\ &\longrightarrow \texttt{malloc} && \text{// 分配内存} \\ &\longrightarrow {}^*E && \text{// 指针解引用} \\ &\longrightarrow \texttt{null} && \text{// 空指针} \end{aligned}$$

这里第一个表达式让指针指向一个变量（从而产生一个指针），第二个表达式在堆上分配一段内存（从而也产生一个指针）。第三个表达式访问指针所指向的值（即指针解引用）。下面形式的赋值语句将值存入到指针所指向的内存：

$$S \longrightarrow {}^*id = E;$$

注意，这里指针是和整数不同的值，因此不允许指针运算。表达式`malloc`也只能分配一个单位的内存，不过即使有如此限制也能展示指针分析的挑战性。这个语言也允许函数指针，为此引入新的函数调用形式：

$$E \longrightarrow (E)(E, \cdots, E)$$

函数指针给出了对象或高阶函数的简单模型。

(5) 程序。最后程序就是一组函数：

$$P \longrightarrow F \cdots F$$

最后一个函数是起始运行的主函数，它的实际参数来自输入流，而它返回的值会添加到输出流。我们假定程序所声明的所有标识符(identifiers)都是唯一的。

3.3 类型分析

上面给出的程序设计语言是无类型的(untyped)，但是许多操作被设计成只用于某些参数，特别地，下面的一些约束是合理的：

(1) 算术操作和比较操作只用于整数；

(2) 只有整数才被用于输入和输出；

(3) 控制结构中的条件表达式的值只会是整数；

(4) 只有函数会被调用；

(5) 操作 *只用于指针。

我们假定违反上述规则会导致运行错误，因此，对于给定的程序，我们希望知道它们的运行是否会满足上述要求。根据Rice定理，这些问题本身是不可判定的，但是我们可求助于类型推断(typability)这个保守近似求解方案。一个程序被称为**可类型化的**(typable)，如果它满足一系列能从给定程序的语法树推导出来的类型约束(A program is typable if it satisfies a collection of type constraints that is systematically derived from the syntax tree of the given program)。可类型化程序意味着它满足上面的规则，但反之则不为真（即满足上面的规则不一定意味着是可类型化的）。因此，我们的类型化检测器是保守的，可能拒绝某些实际上在运行时不会违反上述规则的程序。

3.3.1 类型

首先给出类型的定义，以描述（上面程序语言中）可能的值：

$$
\begin{aligned}
\tau &\longrightarrow \texttt{int} && \text{// 整数类型} \\
&\longrightarrow \&\tau && \text{// 指针类型} \\
&\longrightarrow (\tau, \cdots, \tau)\texttt{->}\tau && \text{// 函数指针类型}
\end{aligned}
$$

上面的类型项分别描述整数、指针和函数指针。上面的语言将规范地生成有限类型(finite type)，但是对于递归函数和数据结构我们需要**正则类型**(regular types)，被定义为在上述构造上的正则树(regular tree)。注意，一个可能无穷的树称为是正则的，如果它只包含有限多个不同的子树。

3.3.2 类型约束

给定一个程序，我们生成一个约束系统(constraint system)，并且定义程序是可类型化的，如果这些约束是可求解的。在我们的例子中，只考虑带变量正则类型项上的等式约束(equality constraints over regular type terms with variables)，这种类型的约束系统可使用合一算法(unification algorithm)高效地求解。

对每个标识符id，引入一个类型变量$[\![id]\!]$，每个表达式E也引入一个类型变量$[\![E]\!]$，这里，E代表在语法树的一个具体节点，而不是它所对应的整个语法，这使得我们的术语有点模糊，但是比那种遵循严谨学术正确途径的方法更为简单(simpler than a pedantically correct approach)。约束可以系统化地针对上面语言的每个构造进行定义：

$$
\begin{aligned}
intconst: &\quad [\![intconst]\!] = \texttt{int} \\
E_1 \text{ op } E_2: &\quad [\![E_1]\!] = [\![E_2]\!] = [\![E_1 \text{ op } E_2]\!] = \texttt{int} \\
E_1 == E_2: &\quad [\![E_1]\!] = [\![E_2]\!] \wedge [\![E_1 == E_2]\!] = \texttt{int} \\
\texttt{input}: &\quad [\![\texttt{input}]\!] = \texttt{int} \\
id = E_2: &\quad [\![id]\!] = [\![E]\!] \\
\texttt{input } E: &\quad [\![E]\!] = \texttt{int} \\
\texttt{if } (E)\ S: &\quad [\![E]\!] = \texttt{int} \\
\texttt{if } (E)\ S_1 \texttt{ else } S_2: &\quad [\![E]\!] = \texttt{int} \\
\texttt{while } (E)\ S: &\quad [\![E]\!] = \texttt{int}
\end{aligned}
$$

$$
\begin{aligned}
id(id_1,\cdots,id_n)\{\cdots \texttt{return}\ E;\ \}: &\quad [\![id]\!] = ([\![id_1]\!],\cdots,[\![id_n]\!])\texttt{->}[\![E]\!] \\
id(E_1,\cdots,E_n): &\quad [\![id]\!] = ([\![E_1]\!],\cdots,[\![E_n]\!])\texttt{->}[\![id(E_1,\cdots,E_n)]\!] \\
(E)(E_1,\cdots,E_n): &\quad [\![E]\!] = ([\![E_1]\!],\cdots,[\![E_n]\!])\texttt{->}[\![(E)(E_1,\cdots,E_n)]\!] \\
\&id: &\quad [\![\&id]\!] = \&[\![id]\!] \\
\texttt{malloc}: &\quad [\![\texttt{malloc}]\!] = \&\alpha \\
\texttt{null}: &\quad [\![\texttt{null}]\!] = \&\alpha \\
{}^*E: &\quad [\![E]\!] = \&[\![{}^*E]\!] \\
{}^*id = E: &\quad [\![id]\!] = \&[\![E]\!]
\end{aligned}
$$

上面α的每处出现代表一个新的类型变量(a fresh type variable)。注意，（表达式中的）变量使用和（语句中的）声明不产生任何约束，而园括号表达式不存在于抽象语法树中。

基于上面的定义，这样每个给定的程序都产生在带变量的类型项上的一组约束等式(a given program gives rise to a collection of equality constraints on type terms with variables)。这组约束等式的**解**是对每个类型变量赋予一个类型，使得每个等式约束都能满足。**约束求解算法的正确性可保证如果解存在则特定的运行错误不会出现**(The correctness claim for this algorithm is that the existence of a solution implies that the specified runtime errors cannot occur during execution)。

3.3.3 约束求解

如果解存在，则它们可以使用针对正则项(regular terms)的合一算法(unification algorithm)在几乎线性时间内求解，因为约束本身也可以在线性时间内提取，因此整个类型分析过程非常高效。

针对下面故意复杂化的求阶乘程序：

```
foo ( p, x ) {
    var f, q;
    if ( *p == 0 ) { f = 1; }
    else {
        q = malloc;
        *q = ( *p) − 1;
        f = ( *p) ∗ ((x)(q, x));
    }
    return f;
}
```

```
main () {
    var n;
    n = input;
    return foo (&n, foo);
}
```

产生的等式约束系统（方程）如下：

$$
\begin{aligned}
[\![\texttt{foo}]\!] &= ([\![\texttt{p}]\!],[\![\texttt{x}]\!])\texttt{->}[\![\texttt{f}]\!] \\
[\![{}^*\texttt{p}]\!] &= \texttt{int} \\
[\![1]\!] &= \texttt{int}
\end{aligned}
\qquad
\begin{aligned}
[\![{}^*\texttt{p}]\!] &= \texttt{int} \\
[\![\texttt{f}]\!] &= [\![1]\!] \\
[\![0]\!] &= \texttt{int}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{p}]\!] = \&[\![{}^*\texttt{p}]\!] && [\![\texttt{q}]\!] = [\![\texttt{malloc}]\!] \\
&[\![\texttt{malloc}]\!] = \&\alpha && [\![q]\!] = \&[\![({}^*\texttt{p}) - 1]\!] \\
&[\![\texttt{q}]\!] = \&[\![{}^*\texttt{q}]\!] && [\![{}^*\texttt{p}]\!] = \texttt{int} \\
&[\![\texttt{f}]\!] = [\![({}^*\texttt{p}) * ((\texttt{x})(\texttt{q},\texttt{x}))]\!] && [\![({}^*\texttt{p}) * ((\texttt{x})(\texttt{q},\texttt{x}))]\!] = \texttt{int} \\
&[\![(\texttt{x})(\texttt{q},\texttt{x})]\!] = \texttt{int} && [\![\texttt{x}]\!] = ([\![\texttt{q}]\!],[\![\texttt{x}]\!])\texttt{->}[\![(\texttt{x})(\texttt{q},\texttt{x})]\!] \\
&[\![\texttt{input}]\!] = \texttt{int} && [\![\texttt{main}]\!] = ()\texttt{->}[\![\texttt{foo}(\&\texttt{n},\texttt{foo})]\!] \\
&[\![\texttt{n}]\!] = [\![\texttt{input}]\!] && [\![\&\texttt{n}]\!] = \&[\![\texttt{n}]\!] \\
&[\![\texttt{foo}]\!] = ([\![\&\texttt{n}]\!],[\![\texttt{foo}]\!])\texttt{->}[\![\texttt{foo}(\&\texttt{n},\texttt{foo})]\!] && [\![{}^*\texttt{p}]\!] = [\![0]\!]
\end{aligned}
$$

这些约束有一个解，其中大多数变量都被赋予整数类型，除了：

$$
\begin{aligned}
[\![\texttt{p}]\!] &= \&\texttt{int} & [\![\texttt{q}]\!] &= \&\texttt{int} \\
[\![\texttt{malloc}]\!] &= \&\texttt{int} & [\![\texttt{x}]\!] &= \phi \\
[\![\texttt{foo}]\!] &= \phi & [\![\&\texttt{n}]\!] &= \&\texttt{int} \\
[\![\texttt{main}]\!] &= ()\texttt{->}\texttt{int}
\end{aligned}
$$

其中ϕ是对应如下无穷展开的正则类型：

$$\phi = (\&\texttt{int}, \phi)\texttt{->}\texttt{int}$$

或者说，ϕ是一个递归类型，是满足上述递归方程的解（不动点）。因为解是存在的，因此上面的程序是类型正确的。递归类型对于数据结构也是需要的，例如下面的程序：

```
var p;
p = malloc;
*p = p;
```

这个程序产生下面的约束方程：

$$
\begin{aligned}
[\![\texttt{p}]\!] &= \&\alpha \\
[\![\texttt{p}]\!] &= \&[\![\texttt{p}]\!]
\end{aligned}
$$

上述约束方程有解$[\![\texttt{p}]\!] = \psi$，这里$\psi = \&\psi$。也有些约束方程有无穷多个解，例如下面的函数：

```
poly(x){
    return *x;
}
```

有类型$\&\alpha\texttt{->}\alpha$（对任意的类型$\alpha$），这对应这个函数所展现的多态行为(polymorphic behavior)。

3.3.4 疏漏与局限

上面的类型分析无疑是近似的，这意味着有些程序会被不公平的拒绝（即认为是不可类型化的）。一个简单的例子是：

```
bar(g,x) {
  var r;
  if (x==0) r=g; else r=bar(2,0);
  return r+1;
}

main() {
  return bar(null,1);
}
```

这个程序不会导致运行错误（注意，在所给出的程序语言中，变量的类型是动态的），但是会产生等价于下面的类型约束方程：

$$\texttt{int} = [\![\texttt{r}]\!] = [\![\texttt{g}]\!] = \&\alpha$$

这显然是不可解的。可能可以使用更强的多态类型分析去接受上面的程序，但是仍然有许多其他例子是不可类型化的。

上述类型分析存在的另外一个问题是忽略了一些运行错误，例如空指针`null`的解引用，未初始化变量的读取，被0整除，以及如下面程序所给出的更微妙的栈元素溢出(escaping stack cell)问题：

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p=baz(); *p=1;
  return *p;
}
```

其中的问题是`*p`指向的栈元素是从函数`baz`溢出(escape)来的（也即，它指向的内存是函数的局部变量所占用的内存，而局部变量的内存是分配在调用栈中，在函数调用完毕之后会被释放）。后面可以看到，这里给出的这些问题可使用更进一步的静态分析处理。

3.4 格论

将要研究的静态分析技术基于数学上面的格论，这里对格论做简单介绍。

3.4.1 格

偏序是一个数学结构$L = (S, \sqsubseteq)$，这里S是集合，而$\sqsubseteq$是一个序关系且满足：

$$\begin{aligned}
\text{自反性：}&\ \forall x \in S\ :\ x \sqsubseteq x \\
\text{传递性：}&\ \forall x, y, z \in S\ :\ x \sqsubseteq y \wedge y \sqsubseteq z\ \Rightarrow\ x \sqsubseteq z \\
\text{反对称性：}&\ \forall x, y \in S\ :\ x \sqsubseteq y \wedge y \sqsubseteq x\ \Rightarrow\ x = y
\end{aligned}$$

设$X \subseteq S$，说$y \in S$是X的**上界**(upper bound)，记为$X \sqsubseteq y$，如果有$\forall x \in X : x \sqsubseteq y$。相似地，说$y \in S$是$X$的**下界**(lower bound)，记为$y \sqsubseteq X$，如果$\forall x \in X : y \sqsubseteq x$。一个**最小上界**(least upper bound)，记为$\sqcup X$，定义为：

$$X \sqsubseteq \sqcup X \ \ \wedge \ \ \forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

对偶地，一个**最大上界**(greatest lower bound)，记为$\sqcap X$，定义为：

$$\sqcap X \sqsubseteq X \ \ \wedge \ \ \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

一个**格**(lattice)是一个偏序$(S, \sqsubseteq)$，而且对任意$X \subseteq S$，$\sqcup X$和$\sqcap$都存在。注意格总有唯一的最大元$\top = \sqcup S$和唯一的最小元$\bot = \sqcap S$。这里主要针对有限格，对于有限格，只需要求$\bot$和$\top$存在，以及对任意两个元素x, y有它们的最小上界$x \sqcup y$和最大下界$x \sqcap y$存在即可。实际上，有$\bot = \sqcup \varnothing$以及$\top = \sqcap \varnothing$（注意所有的元素都是空集的上界或者下界），而任意两个元素有最大下界和最小上界，也就很容易得到任意有限个元素也有最大下界和最小上界。

有限格可使用哈斯图(Hasse diagram)表示。对任意有限集合A，可定义格$(2^A, \subseteq)$，其中$\bot = \varnothing, \top = A, x \sqcup y = x \cup y$以及$x \sqcap y = x \cap y$。有限格的**高度**(height)定义为从$\bot$到$\top$的最长路径的长度。

3.4.2 不动点

格$(L, \sqsubseteq)$上的函数$f : L \to L$称为单调的(monotone)，如果$\forall x, y \in S \ : \ x \sqsubseteq y \ \Rightarrow f(x) \sqsubseteq f(y)$。注意这个性质不意味着$f$是递增的(increasing)（即$\forall x \in S \ : \ x \subseteq f(x)$），例如所有常函数都是单调的。如果将$\sqcup$和$\sqcap$看做函数，它们对其两个参数都是单调的。注意到，两个单调函数的复合仍是单调的。

我们需要的核心结果(central result)是不动点理论。对于有有限高度的格L，每个单调函数f有唯一的最小不动点：

$$fix(f) \ = \bigsqcup_{i \geq 0} f^i(\bot)$$

满足$f(fix(f)) = fix(f)$。计算不动点的时间复杂度取决于三个因素：

(1) 格的高度，这给出了一个界k；

(2) 计算函数f的代价；

(3) 检测（两个函数值）相等性的代价。

3.4.3 封闭性质

如果$L_1, L_2, \cdots, L_n$是有限高度的格，那么它们的笛卡尔积也是：

$$L_1 \times L_2 \times \cdots \times L_n \ = \ \{(x_1, x_2, \cdots, x_n) \mid x_i \in L_i\}$$

其中序$\sqsubseteq$按点定义(defined pointwise)，且$\sqcup$和$\sqcap$也可按点计算(computed pointwise)，而且$height(L_1 \times \cdots \times L_n) = height(L_1) + \cdots + height(L_n)$。也可以有和操作：

$$L_1 + L_2 + \cdots + L_n = \{(i, x_i) \mid x_i \in L_i \backslash \{\bot, \top\}\} \cup \{\bot, \top\}$$

这里$\bot$和$\top$是额外引入的最小元和最大元，而$(i, x) \sqsubseteq (j, y)$当且仅当$i = j$且$x \sqsubseteq y$。注意到$height(L_1 + \cdots + L_n) = \max\{height(L_i)\}$。

如果L是有限高度的格，则$lift(L) = L \cup \{\bot\}$也是格，而且$height(lift(L)) = height(L) + 1$。如果$A$是有限集，则$flat(A)$是高度为2的格（$flat(A)$引入两个元素$\bot$和$\top$，所有$A$的元素大于$\bot$而小于$\top$，$A$中的元素之间不可比）。最后，如果$A$是有限集，$L$是有限高度的格，则它们之间的函数是一个映射格(map lattice)，也具有有限高度：

$$A \mapsto L = \{[a_1 \mapsto x_1, \cdots, a_n \mapsto x_n] \mid x_i \in L\}$$

两个映射（函数）的序定义为$f \sqsubseteq g$当且仅当$\forall a_i : f(a_i) \sqsubseteq g(a_i)$，注意到$height(A \mapsto L) = |A| \cdot height(L)$。

3.4.4 等式与不等式

不动点可用于求解等式方程。设L是有限高度的格，一个方程组(equation system)具有如下形式：

$$\begin{aligned} x_1 &= F_1(x_1, \cdots, x_n) \\ x_2 &= F_2(x_1, \cdots, x_n) \\ &\vdots \\ x_n &= F_n(x_1, \cdots, x_n) \end{aligned}$$

这里x_i是变量，而$F_i : L^n \to L$是一组单调函数。每个这样的方程组都有最小解，它可通过函数$F : L^n \to L^n$的最小不动点得到：

$$F(x_1, \cdots, x_n) = (F_1(x_1, \cdots, x_n), \cdots, F_n(x_1, \cdots, x_n))$$

也可类似地求解如下形式的不等式组：

$$\begin{aligned} x_1 &\sqsubseteq F_1(x_1, \cdots, x_n) \\ x_2 &\sqsubseteq F_2(x_1, \cdots, x_n) \\ &\vdots \\ x_n &\sqsubseteq F_n(x_1, \cdots, x_n) \end{aligned}$$

因为可看到$x \sqsubseteq y$等价于$x = x \sqcap y$，因此求解可通过重写上述方程如下：

$$\begin{aligned} x_1 &= x_1 \sqcap F_1(x_1, \cdots, x_n) \\ x_2 &= x_2 \sqcap F_2(x_1, \cdots, x_n) \\ &\vdots \\ x_n &= x_2 \sqcap F_n(x_1, \cdots, x_n) \end{aligned}$$

3.5 控制流图

类型分析从程序的语法树开始，并且将定义在变量上的约束赋予语法树节点。这种分析是流不敏感的(flow insensitive)，其含义是如何交换语句$S_1\ S_2$的顺序为$S_2\ S_1$，分析的结果也不会改变。流敏感(flow sensitive)的分析使用控制流图，这是程序源代码的另外一种表示。

从现在开始，我们只考虑TIP语言的一个子集，仅包含单个函数体，而且没有指针。一个控制流图是一个有向图，其节点对应程序点(program points)，而边表示可能的控制流向。一个控制流图总有单个的入口，使用$entry$表示，也有单个出口，使用$exit$表示。

如果v是CFG的一个节点，使用$pred(v)$表示它的前驱节点集，而使用$succ(v)$表示它的后继节点集。一个程序的控制流图可归纳地进行构造。

3.6 数据流分析

经典的数据流分析，也称为单调框架(monotone framework)，始于一个CFG和一个有有限高度的格，这个格可能对所有程序都是固定的，也可能是给定不同程序有不同的格。

对CFG的每个节点v，都赋予一个取值返回是L中元素的变量$[\![v]\!]$。对程序设计语言的每个构造，都要定义一个数据约束(dataflow constraint)，将对应节点的变量值与其他节点（通常是它的邻居）关联起来。跟在类型推导中一样，下面将模糊地使用$[\![S]\!]$代替$[\![v]\!]$，如果S是与v相关联的语法，具体的含义可从上下文区分。

对整个CFG，我们可以系统地提取一组在变量上的约束，如果所有的约束恰好是等式或不等式，且右边是单调的，那么我们可使用不动点算法计算它的唯一最小解。数据流约束称为合理(sound)的，如果所有的解都对应关于程序的正确性质。分析是保守的，因为解多多少少会有些不精确，但是计算最小解会给出最高程度的精确性。

3.6.1 不动点算法

如果控制流图有节点$V = \{v_1, v_2, \cdots, v_n\}$，这样我们工作在格$L^n$上。假定节点$v_i$生成的数据流方程是$[\![v_i]\!] = F_i([\![v_i]\!], \cdots, [\![v_n]\!])$，我们可构造联合函数$F : L^n \to L^n$：

$$F(x_1, \cdots, x_n) = (F_1(x_1, \cdots, x_n), \cdots, F_n(x_1, \cdots, x_n))$$

朴素算法可如下进行（计算不动点）：

$$
\begin{array}{l}
x = (\bot, \cdots, \bot); \\
\texttt{do}\ \{ \\
\quad t = x; x = F(x); \\
\}\ \texttt{while}\ (x \neq t);
\end{array}
$$

一个更好的算法称为混沌迭代(chaotic iteration)：

$$
\begin{array}{l}
x_1 = \bot; \cdots x_n = \bot; \\
\texttt{do}\ \{ \\
\quad t_1 = x_1; \cdots t_n = x_n; \\
\quad x_1 = F_1(x_1, \cdots, x_n); \\
\quad \vdots \\
\quad x_n = F_n(x_1, \cdots, x_n); \\
\}\ \texttt{while}\ (x_1 \neq t_1 \vee \cdots \vee x_n \neq t_n);
\end{array}
$$

这个算法之所以比上面算法稍好一点，应该取决于(1) 向量在程序设计语言中怎样表示，向量的操作通常比单个量的操作所需代价大；(2) 在`while`的条件判断中，逐个判断可利用短路计算从而加快算法的执行速度（但本质上海市要取决于向量的表示方法，和上一个算法中向量运算的实现方法）。

上面两个算法都会浪费时间，因为在每次迭代都计算了所有节点，即使我们可能知道有些节点不可能被改变。为得到更好的算法，我们需要进一步研究单个约束的结构。在一般的情况下，每个变量$[\![v_i]\!]$依赖所有其他变量，但通常，F_i的实际实例只会读取少数其他变量，我们使用下面的映射表示这种依赖信息：

$$dep : V \to 2^V$$

对每个节点v给出变量$[\![v]\!]$以非平凡方式出现在它们右边的那些变量构成的集合，也即$dep(v)$是那些信息的计算依赖v的信息的节点。基于这个信息，可给出一个工作列表算法(work-list algorithm)：

```
x_1 = ⊥; ··· x_n = ⊥;
q = [v_1, ··· , v_n]
while (q ≠ []) {
    assume q = [v_i, ···]                  // 也即假定q的头一节点是v_i
    y = F_i(x_1, ··· , x_n);
    q = q.tail();                          // 也即去掉原来q的头一节点
    if (y ≠ x_i) {                         // x_i有了改变
        for (v ∈ dep(v_i)) q.append(v)     // v的计算依赖v_i，所以下次要重新计算v;
        x_i = y;
    };
};
```

虽然最坏情况下的时间复杂度没有变，但是在实际中上述算法节省许多时间。更进一步的优化也是可能的，例如可通过dep映射导出的强连通分量分别计算，也可将队列变为有限队列使得我们可以针对一些特定的数据流问题探索领域特定的知识。

3.6.2 例子：活跃性

称一个变量在一个程序点是**活跃的**(live)，如果它当前的值可能在程序剩余部分的执行中会被读取(A variable is live at a program point if its current value may be read during the remaining execution of the program)。虽然很显然是不可判定的问题，但是这个性质可以使用静态分析近似。

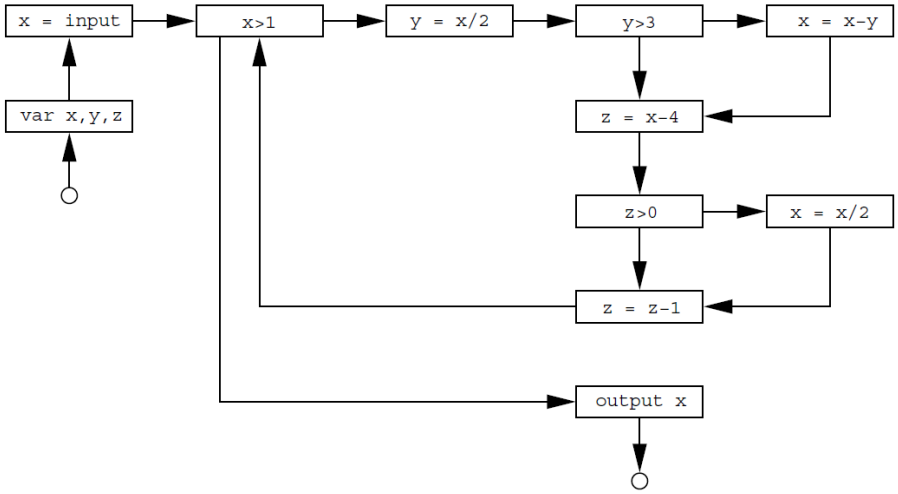
我们使用一个幂集格，其中的元素是出现在给定程序中的变量。对于下面的例子程序：

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

所需要的格是

$$L = (2^{\{x,y,z\}}, \subseteq)$$

而对应的控制流图是：



对每个CFG节点v引入一个约束变量$[\![v]\!]$表示在该节点之前的程序点活跃的变量子集。分析是保守的，因为所计算的集合可能过大。使用下面的辅助定义：

$$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$

对于出口节点，约束方程是：

$$[\![exit]\!] = \{\}$$

对于条件（包括分支语句和循环语句）以及输出语句，约束方程是：

$$[\![v]\!] = JOIN(v) \cup vars(E)$$

这里$vars(E)$是表达式E中出现的所有变量的集合。对于赋值语句，约束方程是：

$$[\![v]\!] \;=\; (JOIN(v) \setminus \{id\}) \cup vars(E)$$

对于变量声明，约束方程是：

$$[\![v]\!] \;=\; JOIN(v) \setminus \{id_1, \cdots, id_n\}$$

最后，对于所有其他节点，约束方程是：

$$[\![v]\!] \;=\; JOIN(v)$$

很显然，这些约束具有单调的右部。具体来说，对于给定的程序，约束方程中的标识符（即$id, id_1, \cdots, id_n$）等都是固定的，表达式中的变量集（即$vars(E)$）是固定的，语句的后继语句集也是固定的，也就是说，一个语句v对应的约束方程所导出的函数$F_{[\![v]\!]} : L^n \to L$的表达式中只有$JOIN(v)$中用于求并集的$[\![w]\!]$是这个函数的变量，其他都是常量，因此函数$F_{[\![v]\!]}$的单调性只取决于$v$的后继语句集$w_1, w_2, \cdots, w_k$所对应的活跃变量集$[\![w_1]\!], \cdots, [\![w_k]\!]$的值，即只要满足如下条件则$F_{[\![v]\!]}$是单调的：

$$[\![w_1]\!] \subseteq [\![w_1]\!]' \wedge \cdots \wedge [\![w_k]\!] \subseteq [\![w_k]\!]' \qquad \text{蕴涵} \qquad JOIN(v) \subseteq JOIN(v)'$$

这里$[\![w_i]\!]$是在迭代求解最小不动点时某一次迭代时语句w_i的活跃变量集，$JOIN(v)$是对一次迭代语句v的这些后继语句的活跃变量集求并的结果，而$[\![w_i]\!]$是另外某次迭代时语句w_i的活跃变量集，$JOIN(v)'$是相应的求并结果。也就是说，由于求并操作对于集合子集这个序是单调的，因此函数$F_{[\![v]\!]}$就是单调的。

直观上来说，一个变量是活跃的，如果它在当前节点被读取，或者在某个未来节点被读取除非它在当前被赋值（被写）。而上述或者是单调的含义则是，如果在迭代求解某个变量已经确定是某个程序点的活跃变量，那么在后续的迭代中它不会被删除，总保持是这个程序点的活跃变量。

上面的例子程序产生如下的约束方程组：

$$\begin{aligned}
[\![\texttt{var x, y, z}]\!] &= [\![\texttt{x = input}]\!] \setminus \{\texttt{x},\texttt{y},\texttt{z}\} \\
[\![\texttt{x = input}]\!] &= [\![\texttt{x} > 1]\!] \setminus \{\texttt{x}\} \\
[\![\texttt{x} > 1]\!] &= ([\![\texttt{y} = \texttt{x}/2]\!] \cup [\![\texttt{output x}]\!]) \cup \{\texttt{x}\} \\
[\![\texttt{y} = \texttt{x}/2]\!] &= ([\![\texttt{y} > 3]\!] \setminus \{\texttt{y}\}) \cup \{\texttt{x}\} \\
[\![\texttt{y} > 3]\!] &= ([\![\texttt{x} = \texttt{x} - \texttt{y}]\!] \cup [\![\texttt{z} = \texttt{x} - 4]\!]) \cup \{\texttt{y}\} \\
[\![\texttt{x} = \texttt{x} - \texttt{y}]\!] &= ([\![\texttt{z} = \texttt{x} - 4]\!] \setminus \{\texttt{x}\}) \cup \{\texttt{x},\texttt{y}\} \\
[\![\texttt{z} = \texttt{x} - 4]\!] &= ([\![\texttt{z} > 0]\!] \setminus \{\texttt{z}\}) \cup \{\texttt{x}\} \\
[\![\texttt{z} > 0]\!] &= ([\![\texttt{x} = \texttt{x}/2]\!] \cup [\![\texttt{z} = \texttt{z} - 1]\!]) \cup \{\texttt{z}\} \\
[\![\texttt{x} = \texttt{x}/2]\!] &= ([\![\texttt{z} = \texttt{z} - 1]\!] \setminus \{\texttt{x}\}) \cup \{\texttt{x}\} \\
[\![\texttt{z} = \texttt{z} - 1]\!] &= ([\![\texttt{x} > 1]\!] \setminus \{\texttt{z}\}) \cup \{\texttt{z}\} \\
[\![\texttt{output x}]\!] &= [\![exit]\!] \cup \{\texttt{x}\} \\
[\![exit]\!] &= \{\}
\end{aligned}$$

在这个约束方程组的方程中，每个使用⟦ ⟧括起来的语句都代表一个变量，其他的集合都是常量。例如，第二个约束方程的右部$[\![\mathtt{x} > 1]\!] \setminus \{\mathtt{x}\}$是单调的，因为随着集合$[\![\mathtt{x} > 1]\!]$的不断增大，整个表达式的结果集合也不断增大。

这个约束方程组的最小解是：

$$
\begin{aligned}
[\![entry]\!] &= \{\} & [\![\mathtt{var\ x,\ y,\ z}]\!] &= \{\} \\
[\![\mathtt{x = input}]\!] &= \{\} & [\![\mathtt{x} > 1]\!] &= \{\mathtt{x}\} \\
[\![\mathtt{y} = \mathtt{x}/2]\!] &= \{\mathtt{x}\} & [\![\mathtt{y} > 3]\!] &= \{\mathtt{x}, \mathtt{y}\} \\
[\![\mathtt{x} = \mathtt{x} - \mathtt{y}]\!] &= \{\mathtt{x}, \mathtt{y}\} & [\![\mathtt{z} = \mathtt{x} - 4]\!] &= \{\mathtt{x}\} \\
[\![\mathtt{z} > 0]\!] &= \{\mathtt{x}, \mathtt{z}\} & [\![\mathtt{x} = \mathtt{x}/2]\!] &= \{\mathtt{x}, \mathtt{z}\} \\
[\![\mathtt{z} = \mathtt{z} - 1]\!] &= \{\mathtt{x}, \mathtt{z}\} & [\![\mathtt{output\ x}]\!] &= \{\mathtt{x}\} \\
[\![exit]\!] &= \{\}
\end{aligned}
$$

例如，在语句$\mathtt{x} = \mathtt{x} - \mathtt{y}$之前的那个程序点的$x, y, z$这三个变量的值，实际上只有$x$和$y$的值在这个语句，以及执行这个语句之后的地方用到（实际上，在后面用到的变量z的值都不等于这个程序点z的值，而是由语句$\mathtt{z} = \mathtt{x} - 4$所赋予的值）。

基于上述活跃变量集的信息，聪明的编译器可推断出实际上y和z的值从来没有同时活跃，而且赋值$\mathtt{z} = \mathtt{z} - 1$所赋予的变量$z$的值从来就没有被用到过，因此上述程序可以安全地被优化为：

```
var x,yz;
x = input;
while (x>1) {
  yz = x/2;
  if (yz>3) x = x-yz;
  yz = x-4;
  if (yz>0) x = x/2;
}
output x;
```

这节省了一个赋值，而且也可以更好地分配寄存器。

可以估计上述分析的最坏时间复杂度。如果程序有n个CFG节点和k个变量，那么格的高度是$k \cdot n$，这给出了迭代求解的迭代次数的上限。每个格的元素可使用长度为k的二进制向量表示，每次迭代可以要执行$O(n)$次并（原文是交）、差和相等操作，每次操作都要花费$O(kn)$的时间，因此整个时间复杂度是$O(k^2n^2)$。

3.6.3 例子：可用表达式

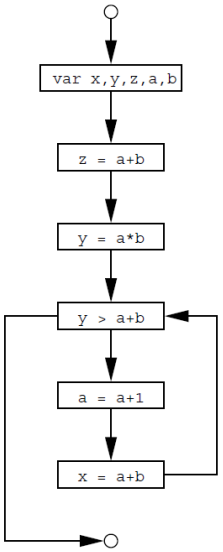
在某个程序点，程序中的一个非平凡表达式称为可用的(available)，如果它的当前值在早先的执行中已经被计算过。所有程序点的可用表达式可使用数据流分析近似，所使用的格是在程序中出现的所有非平凡表达式，序采用子集关系的逆。对于具体的程序：

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```

有四个不同的非平凡表达式，因此我们的格是：

$$L = (2^{\{\texttt{a+b},\texttt{a*b},\texttt{y>a+b},\texttt{a+1}\}}, \supseteq)$$

格的最大元是空集，它对应平凡信息（即没有任何可用表达式）。上述程序的控制流图如下：



对每个CFG节点v引入取值范围为L的约束变量$[\![v]\!]$，其期望含义是包含在这个节点之后的程序点肯定是可用的表达式集合。例如表达式$\texttt{a} + \texttt{b}$在循环条件（之前）处是可用的，但在循环体中最后一次赋值（之前）时不可用。分析是保守的，因为计算的集合可能过小。数据流约束按照如下方式定义，这回我们定义：

$$JOIN(v) = \bigcap_{w \in pred(v)} [\![w]\!]$$

对于入口节点有：

$$[\![entry]\!] = \{\}$$

如果v包含一个条件表达式E，或语句`output` E，则约束方程如下：

$$[\![v]\!] = JOIN(v) \cup exps(E)$$

如果v包含形如$id = E$的赋值，则约束方程如下：

$$[\![v]\!] = (JOIN(v) \cup exps(E)) \downarrow id$$

对于所有其他类型的节点，约束方程为：

$$[\![v]\!] = JOIN(v)$$

这里函数$\downarrow id$移除其中包含所有引用变量id的表达式，而$exps$函数定义为：

$$\begin{aligned} exps(intconst) &= \varnothing \\ exps(id) &= \varnothing \\ exps(\texttt{input}) &= \varnothing \\ exps(E_1 \ \texttt{op} \ E_2) &= \{E_1 \ \texttt{op} \ E_2\} \cup exps(E_1) \cup exps(E_2) \end{aligned}$$

这里op是任意的二元运算符。上述约束的直观含义是，一个表达式在v是可用的，如果它在所有前导节点都是可用的，或者在v被计算，除非它的值被某个（对其所引用的变量进行赋值的）赋值语句所破坏。

同样上述方程的右部构成的函数是单调的，因为本质上是使用集合交，而这里的序是子集关系的逆序，直观上来看，一个表达式在某个程序点一旦确定不是可用表达式，那么它会一直保持不是该程序点的可用表达式，因此上述约束方程是单调的。

对于上面的程序，我们将产生如下的约束方程组：

$$\begin{aligned} [\![entry]\!] &= \{\} \\ [\![\texttt{var x, y, z, a, b;}]\!] &= [\![entry]\!] \\ [\![\texttt{z = a + b}]\!] &= exps(\texttt{a + b}) \downarrow \texttt{z} \\ [\![\texttt{y = a * b}]\!] &= ([\![\texttt{z = a + b}]\!] \cup exps(\texttt{a * b})) \downarrow \texttt{y} \\ [\![\texttt{y > a + b}]\!] &= ([\![\texttt{y = a * b}]\!] \cap [\![\texttt{x = a + b}]\!]) \cup exps(\texttt{y > a + b})) \\ [\![\texttt{a = a * 1}]\!] &= ([\![\texttt{y > a + b}]\!] \cup exps(\texttt{a + 1})) \downarrow \texttt{a} \\ [\![\texttt{x = a + b}]\!] &= ([\![\texttt{a = a + 1}]\!] \cup exps(\texttt{a + b})) \downarrow \texttt{x} \\ [\![exit]\!] &= [\![\texttt{y > a + b}]\!] \end{aligned}$$

使用不动点算法进行求解，将得到下面的最小解：

$$\begin{aligned} [\![entry]\!] &= \{\} \\ [\![\texttt{z = a + b}]\!] &= \{\texttt{a + b}\} \\ [\![\texttt{y > a + b}]\!] &= \{\texttt{a + b}, \texttt{y > a + b}\} \\ [\![\texttt{x = a + b}]\!] &= \{\texttt{a + b}\} \end{aligned} \qquad \begin{aligned} [\![\texttt{var x, y, z, a, b;}]\!] &= \{\} \\ [\![\texttt{y = a * b}]\!] &= \{\texttt{a + b}, \texttt{a * b}\} \\ [\![\texttt{a = a * 1}]\!] &= \{\} \\ [\![exit]\!] &= \{\texttt{a + b}\} \end{aligned}$$

注意到一个表达式在某个节点v之前是否可用可通过计算$JOIN(v)$得到，基于这一点，优化的编译器可在保持程序语义的情况下，将上面的程序转换为下面稍微高效一点的程序：

```
var x,y,z,a,b,aplusb;
aplusb = a+b;
z = aplusb;
y = a*b;
while (y > aplusb) {
  a = a+1;
  aplusb = a+b;
  x = aplusb;
}
```

基于类似的分析，我们也可得到上面分析的最坏情况时间复杂度为$O(k^2n^2)$，这里n是CFG节点数，而k是非平凡表达式数。

3.6.4 例子：忙碌表达式

一个表达式被称为忙碌的(very busy)，如果在它的值改变之前肯定会被重新计算。为逼近该性质，我们需要与可用表达式分析中相同的格和辅助函数。对每个CFG节点v，变量$[\![v]\!]$表示在该节点之前的程序点的忙碌表达式集。定义：

$$JOIN(v) = \bigcap_{w \in succ(v)} [\![w]\!]$$

出口节点的约束方程是：

$$[\![exit]\!] = \{\}$$

对于条件表达式和输出语句有：

$$[\![v]\!] = JOIN(v) \cup exps(E)$$

对于赋值语句有：

$$[\![v]\!] = (JOIN(v) \downarrow id) \cup exps(E)$$

对所有其他节点有：

$$[\![v]\!] = JOIN(v)$$

直观上来看，一个表达式是忙碌的，如果它在当前节点被计算，或者它在所有未来的运行中都会被计算，除非赋值改变它的值。对于下面的例子程序：

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x>0) {
  output a*b-x;
  x = x-1;
}
output a*b;
```

忙碌表达式分析可发现在循环中的`a * b`是忙碌表达式。编译器可执行代码提升(code hosting)，将这个计算移到更早的，该表达式是忙碌表达式的程序点。对上述例子，可得到一个更高效的版本：

```
var x,a,b,atimesb;
x = input;
a = x-1;
b = x-2;
atimesb = a*b;
while (x>0) {
  output atimesb-x;
  x = x-1;
}
output atimesb;
```

3.6.5 例子：到达定值

一个程序点的到达定值(reaching definitions)是那些定义了变量（在这个程序点）的当前值的赋值（语句）(The reaching definitions for a given program point are those assignments that may have defined the current values of variables)。对于这个分析，我们需要程序中所有赋值（或实际上所有CFG节点）的幂集构成的格。对于下面的例子程序：

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

需要的格是：

$$L = (2^{\{\texttt{x=input},\texttt{y=x/2},\texttt{x=x-y},\texttt{z=x-4},\texttt{x=x/2},\texttt{z=z-1}\}}, \subseteq)$$

对CFG的每个节点v，变量$[\![v]\!]$表示在这个节点之后的那个程序点定义了变量值的赋值集合(the set of assignments that may define values of variables at the program point after the node)。定义：

$$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

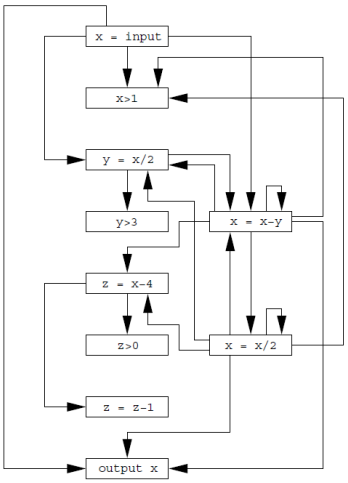
对于赋值，约束方程是：

$$[\![v]\!] = (JOIN(v) \downarrow id) \cup \{v\}$$

对于所有其他类型的节点简单地都有：

$$[\![v]\!] = JOIN(v)$$

这回函数$\downarrow id$去掉对标识符id的所有赋值语句。上述分析可用于构造定值-使用图(def-use graph)，它与CFG类似，但是边从定值指向可能的使用，例如对于上面的例子程序，它的定值-使用图是：



定值-使用图是对程序的进一步抽象，是像死代码删除和代码移动等代码优化的基础。可以证明，定值-使用图总是控制流图CFG的传递闭包的子图(the def-use graph is always a subgraph of the transitive closure of the CFG)。

3.6.6 前向、后向、可能和必然

前面给出的四种传统分析可从各种角度分类，它们是一般的单调框架的例子，但是它们的约束方程有自己特有的结构。

一个**前向分析**(forward analysis)是这样的分析，对每个程序点，计算的是这个程序点过去的行为信息(A forward analysis is one that for each program point computes information about the past behavior)。前向分析的例子是可用表达式分析和到达定值分析，它们的特点是约束方程的右部只依赖于CFG节点的前导节点(predecessors)，因此分析从入口节点(entry node)开始，并且在CFG中向前移动。

一个**后向分析**(backwards analysis)是这样的分析，对每个程序点，计算的是这个程序点未来的行为信息(A backwards analysis is one that for each program point computes information about the future behavior)。后向分析的例子是活跃变量和忙碌表达式分析，它们的特点是约束方程的右部只依赖于CFG节点的后继节点(successors)。因此分析从出口节点(exit node)开始，并且在CFG中向后移动。

一个**可能分析**(may analysis)是这样的分析，它所描述的信息可能为真，因此计算的是上近似(A may analysis is one that describes information that may possibly be true and, thus computes an upper approximation)。可能分析的例子是活跃变量和到达定值分析，它们的特点是约束方程的右部是使用集合并(union)操作去合并信息。

一个**必然分析**(must analysis)是这样的分析，它所描述的信息必然为真，因此计算的是下近似(A may analysis is one that describes information that may possibly be true and, thus computes a lower approximation)。必然分析的例子是可用表达式和忙碌表达式分析，它们的特点是约束方程的右部是使用集合交(intersection)操作去合并信息。

因此上面的例子给出了四种可能的组合，正如下表所示：

	前向(Forwards)	**后向**(Backwards)
可能(May)	到达定值(Reaching Definitions)	活跃变量(Liveness)
必然(Must)	可用表达式(Available Expressions)	忙碌表达式(Very Busy Expressions)

这样的分类很自然，但是熟知这样的分类可提供构造新的分析的灵感。

3.6.7 例子：变量初始化分析

让我们尝试定义一个分析，它保证一个变量在被读取之前必定被初始化，这可通过为每个程序点计算肯定被初始化的变量集而求解，因此我们需要的格是给定程序中所出现的所有变量构成的集合的幂集格。变量的初始化性质是属于过去的信息，因此我们需要前向分析，同时我们需要定义的信息意味着它是一个必然分析。这些意味着我们的约束方程主要使用前导节点和集合交操作。

对于入口节点我们有：

$$[\![entry]\!] = \{\}$$

对于赋值语句我们有：

$$[\![v]\!] = \bigcap_{w \in pred(v)} [\![w]\!] \cup \{id\}$$

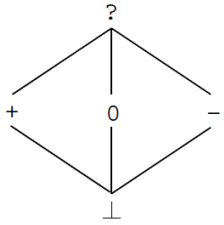
对于所有其他的节点有：

$$[\![v]\!] = \bigcap_{w \in pred(v)} [\![w]\!]$$

现在编译器可以对每个变量的使用检查它是否包含在已经初始化的变量集合（即上述分析的结果）中。

3.6.8 例子：符号分析

现在我们想确定每个表达式的符号$(+, 0, -)$。前面使用的格都是某个集合的幂集格，但是对于符号分析，我们从下面关于符号的迷你格开始：



这里?表示变量的符号值不是常量，而⊥表示变量的值不知道。我们需要的整个格是映射格：

$$Vars \mapsto Sign$$

这里$Vars$是给定程序中所出现的所有变量集合。对每个CFG节点v，赋予变量$[\![v]\!]$表示一个符号表，这个符号表给出在这个节点之前的所有变量的符号值。

对于变量声明，约束方程是：

$$[\![v]\!] = JOIN(v)[id_1 \mapsto ?, \cdots, id_n \mapsto ?]$$

对于赋值，约束方程是：

$$[\![v]\!] = JOIN(v)[id \mapsto eval(JOIN(v), E)]$$

对所有其他节点，约束方程是：

$$[\![v]\!] = JOIN(v)$$

这里：

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

而$eval$执行表达式的抽象求值：

$$\begin{aligned} eval(\sigma, id) &= \sigma(id) \\ eval(\sigma, intconst) &= = sign(intconst) \\ eval(\sigma, E_1 \ \mathtt{op} \ E_2) &= \overline{\mathtt{op}}(eval(\sigma, E_1), eval(\sigma, E_2)) \end{aligned}$$

这里σ是当前环境，函数$sign$给出一个整数敞亮的符号，$\overline{\mathtt{op}}$是对应给定运算符的抽象计算，由下面的表定义：

+	⊥	0	−	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	−	+	?
−	⊥	−	−	?	?
+	⊥	+	?	+	?
?	⊥	?	?	?	?

−	⊥	0	−	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	−	?
−	⊥	−	?	−	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

*	⊥	0	−	+	?
⊥	⊥	0	⊥	⊥	⊥
0	0	0	0	0	0
−	⊥	0	+	−	?
+	⊥	0	−	+	?
?	⊥	0	?	?	?

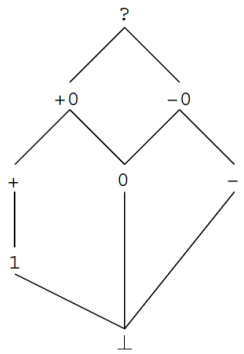
/	⊥	0	−	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	?	0	0	?
−	⊥	?	?	?	?
+	⊥	?	?	?	?
?	⊥	?	?	?	?

>	⊥	0	−	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	0	?
−	⊥	0	?	0	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

==	⊥	0	−	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	+	0	0	?
−	⊥	0	?	0	?
+	⊥	0	0	?	?
?	⊥	?	?	?	?

虽然这时方程右部的单调性不那么显然，但是$\sqcup$运算符以及映射修改是单调的，因此方程右部的单调性仅取决于上面抽象运算表对于符号格是单调的，注意对于$n \times n$的表，可在时间复杂度$O(n^3)$内自动检查其单调性。

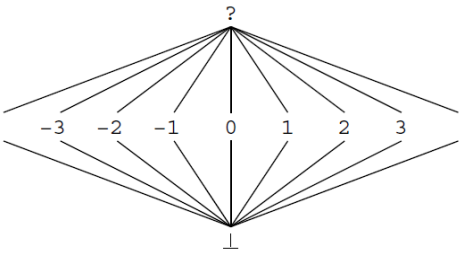
上面的分析可能丢失某些信息，例如对于表达式$(2 > 0) == 1$将被标记为?，这表明上述分析不必要地过粗，而且$+/+$也会导致?而不是$+$，因为，例如$1/2$会约简为0（去掉小数）。为了更精确处理这些情况，可以将上面的符号格扩展元素1（常量1），$+0$表示正或0，而-0表示负或0，以跟踪更准确的信息，从而使用下面的格：



相应地，运算的抽象需要使用8×8的表格。符号分析的结果在理论上可用于通过只接受除数的符号是$+, -$或1而消除被0除的错误，但是这可能不公平地拒绝太多程序从而不实用。

3.6.9 常量传播

一个类似的分析是常量传播，这里我们对程序的每个点想决定变量的值是否是常量值。这种分析的结构与符号分析类似，只是使用下面的格：



而运算的抽象则使用如下的方式，例如对于加法有：

$$\lambda n \lambda m.\texttt{if}\ (n \neq ? \wedge m \neq ?)\ \{n+m\}\ \texttt{else}\ \{?\}$$

基于常量传播分析，一个优化编译器能将下面的程序：

```
var x,y,z;
x = 27;
y = input;
z = 2*x+y;
if (x < 0) { y = z-3; } else { y = 12; }
output y;
```

转换为：

```
var x,y,z;
x = 27;
y = input;
z = 54+y;
if (0) { y = z-3; } else { y = 12; }
output y;
```

在经过到达定制和消除死代码之后，最终得到：

```
var y;
y = input;
output 12;
```

3.6.10 拓宽和收窄

区间分析为每个整数变量计算一个上界和下界以确定它可能取的值。用于描述单个变量的格有如下形式：

$$Interval = lift(\{[l,h] \mid l,h \in N \wedge l \leq h\})$$

这里

$$N = \{-\infty, \cdots, -2, -1, 0, 1, 2, \cdots, \infty\}$$

是整数的集合并带有两个无穷的端点，而区间的序定义为：

$$[l_1,h_1] \sqsubseteq [l_2,h_2] \iff l_2 \leq l_1 \wedge h_1 \leq h_2$$

注意到区间格的高度不是有限的。我们最终使用的格如下：

$$L = Vars \mapsto Interval$$

对于入口节点，我们使用总是返回$\top$运算的常函数：

$$[\![entry]\!] = \lambda x.[-\infty, \infty]$$

对于赋值，约束方程是：

$$[\![v]\!] = JOIN(v)[id \mapsto eval(JOIN(v), E)]$$

对所有其他节点，我们有：对所有其他节点，约束方程是：

$$[\![v]\!] = JOIN(v)$$

这里：

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

而$eval$执行表达式的抽象求值：

$$\begin{aligned} eval(\sigma, id) &= \sigma(id) \\ eval(\sigma, intconst) &= [intconst, intconst] \\ eval(\sigma, E_1 \ \mathtt{op} \ E_2) &= \overline{\mathtt{op}}(eval(\sigma, E_1), eval(\sigma, E_2)) \end{aligned}$$

这里抽象运算$\overline{\mathtt{op}}$都定义为：

$$\overline{\mathtt{op}}([l_1, h_1], [l_2, h_2]) = [\min_{x \in [l_1, h_1], y \in [l_2, h_2]} x \ \mathtt{op} \ y, \quad \max_{x \in [l_1, h_1], y \in [l_2, h_2]} x \ \mathtt{op} \ y]$$

例如$\overline{+}([1, 10], [-5, 7]) = [1 - 5, 10 + 7] = [-4, 17]$。

所使用的格仍然有无穷的高度，所以不能立即使用前面的单调框架，因为不动点算法可能不终止，也即对于格L^n，逼近序列：

$$F^i(\bot, \cdots, \bot)$$

可能不收敛。为此我们使用称为拓宽(widening)的技术，引入一个函数$w : L^n \to L^n$使得序列：

$$(w \circ F)^i(\bot, \cdots, \bot)$$

可收敛到一个不动点，且比所有的$F^i(\bot, \cdots, \bot)$都大，因此只表示关于程序的合理信息。直观上来看，函数w粗化计算的信息以确保可以终止。对于区间分析，w按点定义约简至单个区域(w is defined pointwise down to single intervals)。相对来说，我们固定一个包含元素$-\infty$和∞的有限子集$B \subset N$。一般来说，B必须包含出现在程序中所有的常量，但是其他的启发式规则也可使用。对于单个区间，定义：

$$w([l, h]) = [\max\{i \in B \mid i \leq l\}, \min\{i \in B \mid h \leq i\}]$$

实际上，w的直观含义是为区间$[l, h]$寻找一个在由B确定的允许范围内的最佳区间。不难看到w是单调的，从而对于单调框架F，$w \circ F$也是单调的，而且由于$w(Interval)$是有限高度格（因为B是有限集），所以$(w \circ F)^i(\bot, \cdot, \bot)$会收敛，从而得到一个不动点。

使用一个称为收窄(narrowing)的技术可进一步改善上述结果。定义：

$$fix = \bigsqcup F^i(\bot, \cdots, \bot) \qquad fixw = \bigsqcup (w \circ F)^i(\bot, \cdots, \bot)$$

这样我们有$fix \sqsubseteq fixw$，但是我们也有$fix \sqsubseteq F(fixw) \subseteq fixw$，这意味着对$fixw$应用$F$可优化我们的结果并且保持产生合理的信息，这种技术就称为**收窄**(narrowing)，它实际上可应用多次。

作为例子，考虑在下面的程序：

```
y = 0; x = 8;
while (input) {
    x = 7;
    x = x + 1;
    y = y + 1;
}
```

如果不使用拓宽技术，那么对于循环之后的程序点，分析可能产生下列发散的序列：

$$\begin{aligned}
&[x \mapsto \bot,\ y \mapsto \bot] \\
&[x \mapsto [8,8],\ y \mapsto [0,1]] \\
&[x \mapsto [8,8],\ y \mapsto [0,2]] \\
&[x \mapsto [8,8],\ y \mapsto [0,3]] \\
&\vdots
\end{aligned}$$

如果我们基于集合$B = \{-\infty, 0, 1, 7, \infty\}$，$B$中的值（除两个无穷端点外）来自出现在程序中常量，这样可得到一个收敛的序列：

$$\begin{aligned}
&[x \mapsto \bot,\ y \mapsto \bot] \\
&[x \mapsto [7,\infty],\ y \mapsto [0,1]] \\
&[x \mapsto [7,\infty],\ y \mapsto [0,7]] \\
&[x \mapsto [7,\infty],\ y \mapsto [0,\infty]]
\end{aligned}$$

这里当然x的结果不尽人意，但是幸运的是，只使用一次收窄技术我们可将结果改善为：

$$[x \mapsto [8,8],\ y \mapsto [0,\infty]]$$

这实际上就是我们期望能得到的最好结果，相应地，继续使用收窄技术不会再有效果。注意下降序列：

$$fixw \sqsupseteq F(fixw) \sqsupseteq F^2(fixw) \sqsupseteq F^3(fixw) \cdots$$

不能保证是收敛的，因此需要启发式规则去确定到底要用几次收窄技术。

3.6.11 条件与断言

上面的分析都忽略了条件的值，而将`if`和`while`语句看做在这两个分支之间不确定性地选择(nondeterministic choice)，这样可能没有包含一些可以用在静态分析中的信息。考虑下面的例子程序：

```
x = input;
y = 0;
z = 0;
while (x > 0) {
  z = z+x;
  if (17 > y) { y = y+1; }
  x = x-1;
}
```

前面给出的带拓宽技术的区间分析将断定，在`while`循环之后，变量`x`的区间是$[-\infty,\infty]$，`y`的区间是$[0,\infty]$，`z`的区间是$[-\infty,\infty]$，但是如果结合程序中的条件看，这个结果过于悲观了。

为了利用已知的信息，将前面的程序语言扩展两个人工语句：`assert`(E)和`refute`(E)，这里E是语言中的表达式。这样在区间分析中，对于这两个新语句的约束条件通过基于表达式E必须分别为真和假而优化变量的区间。条件表达的含义可以通过下面的程序转换而体现：

```
x = input;
y = 0;
z = 0;
while (x > 0) {
  assert(x > 0);
  z = z+x;
  if (17 > y) { assert(17 > y); y = y+1; }
  x = x-1;
}
refute(x > 0);
```

对于有语句`assert`或`refute`的节点v，其约束方程可平凡地如下：

$$[\![v]\!] = JOIN(v)$$

但这样不会得到额外的问题求解精度。实际上，需要对特定的静态分析有所洞察，对这些语句定义非平凡而合理的约束。

对于区间分析，针对一般的表达式，例如$E_1 > E_2$或$E_1 == E_2$提取信息可能会太复杂（但这是一个研究课题），我们也可考虑两种简单的表达式形式$id > E$或$E > id$，前者对于`assert`语句可如下处理：

$$[\![v]\!] \;=\; JOIN(v)[id \mapsto gt(JOIN(v)(id), eval(JOIN(v), E))]$$

这里

$$gt([l_1,h_1],[l_2,h_2]) = [l_1,h_1] \cap [l_2,\infty]$$

对于`refute`，以及对偶的条件可类似的处理，而其他条件则给出平凡的，但是合理的恒等约束(identity constraint)。

基于这些优化，上面例子的区间分析将断定，在`while`循环之后，变量`x`的区间是$[-\infty,0]$，`y`的区间是$[0,17]$，而`z`的区间是$[0,\infty]$。

3.7 过程间分析

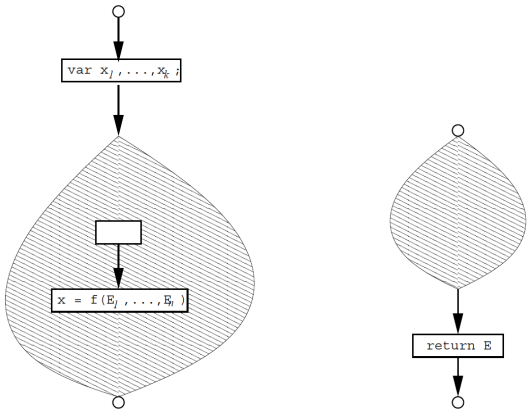
到目前为止，我们只分析单个函数的函数体，这称为过程内分析(intra-procedural)。当考虑包含函数调用的整个程序时，称为过程间分析(inter-procedural)。过程间分析的一个替代技术是独立分析每个函数，并对每个函数调用的结果做最悲观的假设(maximally pessimistic assumptions)。

3.7.1 程序的流图

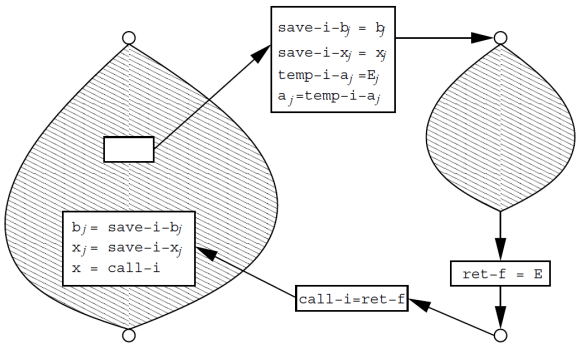
现在我们考虑上面TIP语言的一个包含函数，但是仍然忽略指针的子集。这时整个程序的CFG也很容易得到，因为它对应一个容易系统得到的简单程序的CFG(since it corresponds to the CFG for a simple program that can be systematically obtained)。首先我们所有的函数体单独构造CFG，然后剩下的事情就是将它们粘合(glue)在一起以正确地反映函数调用，这由于我们假定所有声明的标识符都是唯一的而变得更为简单。

我们从引入一些隐士变量(shadow variable)开始，对每个函数`f`，引入变量`ret-f`对应其返回值，对每个调用点引入变量`call-i`，这里`i`是唯一的索引，这个变量代表函数调用所计算的值。对在函数调用者(the calling function)的局部变量和形式参数x，引入变量`save-i-x`以保存这些变量的值，最后对每个在被调用函数(the called function)的形式参数`x`在每个调用点都引入临时变量`temp-i-x`。

我简单起见，我们假定所有的函数调用都在赋值语句中：$\texttt{x} = \texttt{f}(E_1, \cdots, E_n)$，考虑下面的函数调用者和被调用函数的CFG：



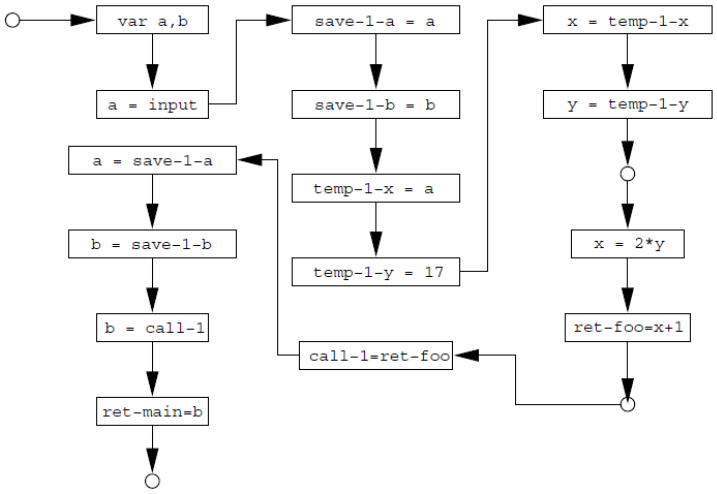
如果被调用函数的形式参数是$\texttt{a}_1, \cdots, \texttt{a}_n$，而函数调用者的形式参数是$\texttt{b}_1, \cdots, \texttt{b}_m$，则上述函数调用得到的图如下：



上图给出了函数调用时变量值的流向。作为例子，考虑下面的程序：

```
foo(x,y) {
  x = 2*y;
  return x+1;
}

main() {
  var a,b;
  a = input;
  b = foo(a,17);
  return b;
}
```

最终得到的CFG如下图：



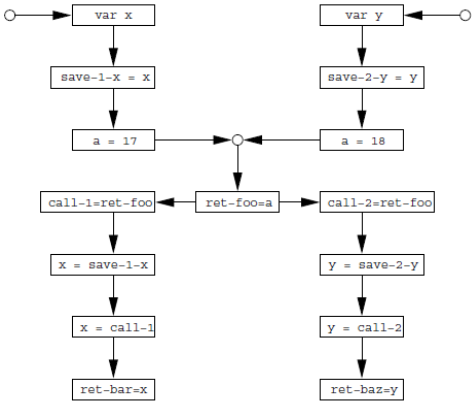
基于这个CFG，就可以使用标准的单调框架进行分析。注意，上面的构造意味着函数参数从左至右计算。在下面的例子中，临时变量将只在需要的时候使用。

3.7.2 Polyvariance

上面给出的过程间分析称为monovariant，因为每个函数体在所有调用点只表示一次。一个polyvariant分析将对函数调用执行上下文依赖的分析，作为例子，考虑下面的程序：

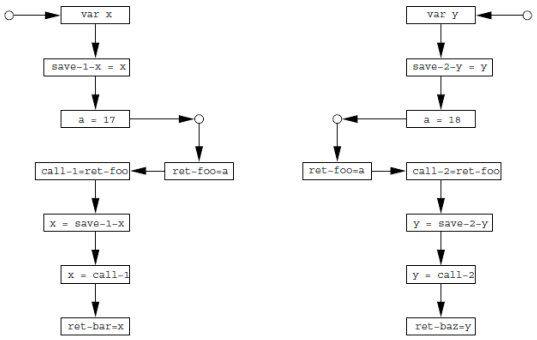
```
foo(a) {
  return a;
}

bar() {
  var x;
  x = foo(17);
  return x;
}

baz() {
  var y;
  y = foo(18);
  return y;
}
```

上面的monovariant分析将得到下面的CFG：



如果我们基于此CFG做常量传播分析，那么函数bar和baz的返回值都将认为不是常量，这里的问题来自对函数foo的调用在CFG中进行了合并。

可通过创建被调用函数体的CFG的多份拷贝而变成polyvariant分析，有很多策略去决定要创建几份拷贝。最简单的策略是对每个调用点都创建一份拷贝，得到下面的CFG：



这可解决上面的常量传播分析存在的问题。但是如果foo的调用被包装在更深一层次，则还是没有解决问题，类似地，递归函数调用也不会从这个技术上得到好处。最好的途径是针对特定的分析使用特定的启发式规则，当然最重要的是要确保只产生有限份拷贝。

3.7.3 例子：Tree Shaking

一个过程间分析的例子是tree shaking，这里我们希望能识别哪些从不会被调用的函数，从而可以安全地将它们从程序中删除。这在程序与一个大的函数库进行一起编译的时候特别有用。这个分析用于monovariant过程间分析得到的CFG上，并采用前面类似的单调框架，所使用的格是程序中出现的函数名构成的集合的幂集格。对任意的CFG节点v，引入约束变量$[\![v]\!]$表示在这个节点的将来可能要调用的函数构成的集合。使用记号$entry(id)$表示名为id的函数的入口节点，对于赋值、条件和输出语句，约束方程是：

$$[\![v]\!] = \bigcup_{w \in succ(v)} [\![w]\!] \cup funcs(E) \cup \bigcup_{f \in funcs(E)} [\![entry(f)]\!]$$

而对于所有其他节点是：

$$[\![v]\!] = \bigcup_{w \in succ(v)} [\![w]\!]$$

这里$funcs$的定义为：

$$\begin{aligned} funcs(id) &= funcs(intconst) = funcs(\texttt{input}) = \varnothing \\ funcs(E_1\ \texttt{op}\ E_2) &= funcs(E_1) \cup funcs(E_2) \\ funcs(id(E_1, \cdots, E_n)) &= \{id\} \cup funcs(E_1) \cup \cdots \cup funcs(E_n) \end{aligned}$$

这些约束可以看做是单调的，所有那些不在$[\![entry(\texttt{main})]\!]$中的函数可确保是不会被调用的（从而可以删除）。

3.8 控制流分析

对于只有一阶函数的语言而言，过程间分析是相当直接的，但如果引入高阶函数、对象或函数指针，则控制流和数据流立即会缠绕在一起。控制流分析的任务是对于这种语言保守地给出它的控制流图。

3.8.1 λ演算的闭包分析

控制流分析可使用经典的λ演算来说明：

$$E \longrightarrow \lambda id.E \mid id \mid E\ E$$

后面我们将这里的技术推广到整个TIP语言。为简单起见，我们假定用于作为λ界的变量都是不同的。为一个这种演算中的项构造CFG，我们需要计算每个表达式E的闭包集合，在这里闭包是形如λid的符号，它识别一个具体的λ抽象。这个问题称为**闭包分析**(closure analysis)，可使用单调框架进行求解。但是因为还没有CFG，因此分析是针对语法树。

我们使用的格是出现在给定的项中的所有闭包集的幂集格，对每个语法节点v引入一个约束变量$[\![v]\!]$表示结果闭包集，对于λ抽象$\lambda id.E$有约束：

$$\{\lambda id\} \subseteq [\![\lambda id.E]\!]$$

对于应用$E_1\ E_2$，对每个闭包$\lambda id.E$有如下条件约束：

$$\lambda id \in [\![E_1]\!] \Rightarrow [\![E_2]\!] \subseteq [\![id]\!] \wedge [\![E]\!] \subseteq [\![E_1\ E_2]\!]$$

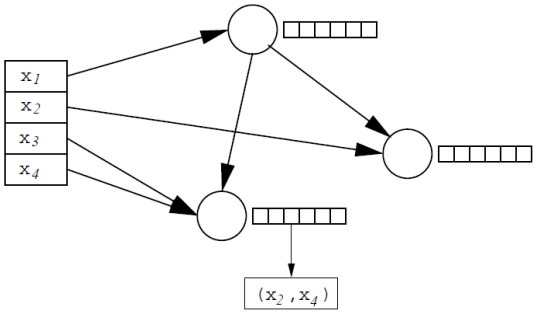
注意这是一个非流敏感分析。

3.8.2 Cubic算法

闭包分析中的约束是能给在立方时间类求解的一类问题的实例，许多问题都可归结于此类，所以这里对这个算法做更详细讨论。我们有一个符号(token)集$\{t_1, \cdots, t_k$，以及一组变量$x_1, \cdots, x_n$，它的值是符号子集。算法的任务是读取一系列形如$\{t\} \subseteq x$或$t \in X \Rightarrow y \subseteq z$这样的约束，然后求最小解。

算法基于一个简单的数据结构，每个变量映射到一个无环有向图(directed acyclic graph, DAG)的节点，每个节点附加一个属于$\{0,1\}^k$的位向量(bit-vector)，初始化都是0。每一位附加一个

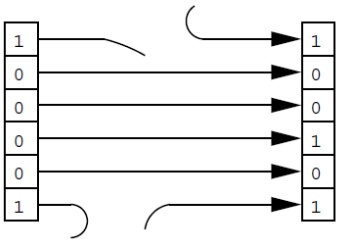
变量对列表，它用于建模条件化约束。DAG中的边反映自己约束。位向量在任何时候都直接表示最小解。一个图的例子如下：



每次条件一个约束。形如$\{t\} \subseteq x$的约束需要查找x附加的节点，并将相应的位设置为1。如果变量对列表非空，则对每个对(y, z)在对应y和z的节点之间增加一条边。形如$t \in X \Rightarrow y \subseteq z$的约束，首先查找$x$附加的节点中对应$t$的位是否为1，如果是则在对应$y$和$z$的节点之间增加一条边，否则将变量对$(y, z)$添加到这个位所附加的变量对列表。

如果新增加的边形成了一个环（回路），则在环上的所有节点需要合并成一个节点，这意味着它们的位向量做并，而它们的变量对列表做连接。从变量到节点的映射也做相应的修改。在任何情况下，重建所有的子集关系，都需要将新的集合位的值传播到图中的所有边。

为分析这个算法，我们假定符号和约束的复杂度都是$O(n)$，这对我们所分析程序来说是显然的，因为这时变量的个数，符号的个数和约束的个数都与程序长度成线性比。对于DAG节点的合并可以在最多$O(n)$的时间复杂度完成，每次合并最多涉及$O(n)$个节点，而且位向量的并最多在$O(n^2)$内完成，因此总的复杂度是$O(n^3)$。新边的加入最多$O(n^2)$次，对每个$\{t\} \subseteq x$约束，常量集合最多在$O(n^2)$的时间复杂度内包含进来。最后为了限制位在边上的传播代价，我们设想对应位的每个变量对通过一个析的位先联系在一起，当源位被设置为1，那么沿着位线的值的传播被打破：



由于最多有n^3条位线，因此传播的总的复杂度为$O(n^3)$。因此总的算法复杂度是$O(n^3)$。由于这事实上是一个比较低的复杂度上界，因此被称为立方时间瓶颈(cubic time bottleneck)。

使用上述算法求解的约束是更一般的集合约束的简单情形。更一般的集合约束允许有限项之间的更丰富的约束，这种约束（方程组）也可求解，但时间复杂度将是$O(2^{2^n})$。

3.8.3 函数指针的控制流图

考虑允许函数指针的TIP语句，对于下面函数调用（需要通过计算表达式E来确定所调用的函数）(computed function call)：

$$E \longrightarrow (E)(E_1, \cdots, E_n)$$

从语法上无法看出哪个函数将被调用。一个粗略但合理CFG可能通过假定任何有正确数目的参数的函数都可能被调用而得到，但是通过控制流分析可以做得更好。注意函数调用$id(E_1,\cdots,E_n)$也看做是上述形式调用的一种，即看做$(id)(E_1,\cdots,E_n)$。

分析所使用的格是对每个函数名id对应的符号$\&id$所构成的集合的幂集格。对每个语法树节点v，引入约束变量$[\![v]\!]$表示v可能需要计算的函数或函数指针构成的集合。对于函数名id有约束：

$$\{\&id\} \subseteq [\![id]\!]$$

对于赋值$id = E$有约束：

$$[\![E]\!] \subseteq [\![id]\!]$$

最后对于通过函数指针求值而进行的函数调用(computed function calls)我们有：对每个有参数$a_1,\cdots,a_n$和返回表达式E'的f有约束：

$$\&f \in [\![E]\!] \Rightarrow [\![E_i]\!] \subseteq [\![a_i]\!] \wedge [\![E']\!] \subseteq [\![(E)(E-1,\cdots,E_n)]\!]$$

针对类型化的程序可做更精确的分析，即只对哪些具有正确类型的函数f生成上面类似的约束。

基于上面的信息，我们可与以前一样构造空值流图，但是边将基于控制流分析给出调用点和所有可能调用的目标函数。考虑下面的例子程序：

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = (f)(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

将产生下面的约束：

$$\{\&\texttt{inc}\} \subseteq [\![\texttt{inc}]\!] \qquad \{\&\texttt{dec}\} \subseteq [\![\texttt{dec}]\!]$$

$$\{\&\texttt{ide}\} \subseteq [\![\texttt{ide}]\!] \qquad [\![\texttt{ide}]\!] \subseteq [\![\texttt{f}]\!]$$

$$[\![(\texttt{f})(\texttt{n})]\!] \subseteq [\![\texttt{r}]\!] \qquad \&\texttt{inc} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{i}]\!] \wedge [\![\texttt{i}+1]\!] \subseteq [\![(\texttt{f})(\texttt{n})]\!]$$

$$\&\texttt{dec} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{j}]\!] \wedge [\![\texttt{j}+1]\!] \subseteq [\![(\texttt{f})(\texttt{n})]\!] \qquad \&\texttt{ide} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{k}]\!] \wedge [\![\texttt{k}]\!] \subseteq [\![(\texttt{f})(\texttt{n})]\!]$$

$$[\![\texttt{input}]\!] \subseteq [\![\texttt{x}]\!] \qquad [\![\texttt{foo(x,inc)}]\!] \subseteq [\![\texttt{y}]\!]$$

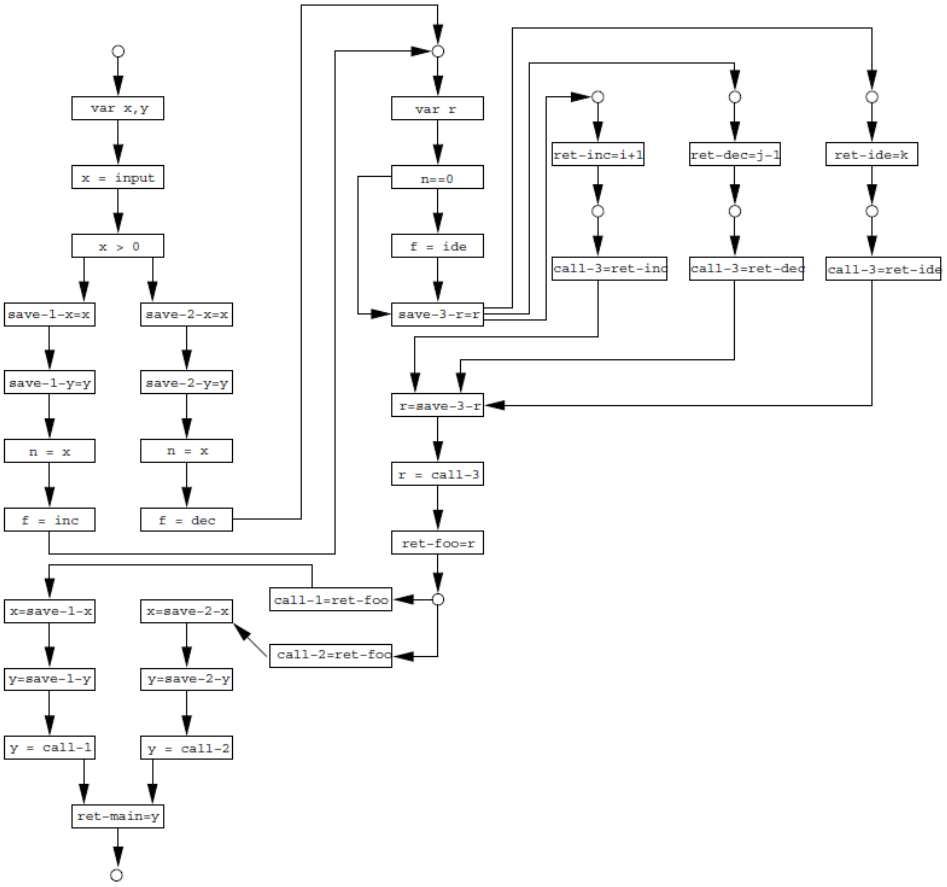
$$[\![\texttt{foo(x,dec)}]\!] \subseteq [\![\texttt{y}]\!] \qquad \{\&\texttt{foo}\} \subseteq [\![\texttt{foo}]\!]$$

$$\&\texttt{foo} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \wedge [\![\texttt{inc}]\!] \subseteq [\![\texttt{f}]\!] \wedge [\![\texttt{r}]\!] \subseteq [\![\texttt{foo(x,inc)}]\!]$$
$$\&\texttt{foo} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \wedge [\![\texttt{dec}]\!] \subseteq [\![\texttt{f}]\!] \wedge [\![\texttt{r}]\!] \subseteq [\![\texttt{foo(x,dec)}]\!]$$

上述约束（方程组）最小解的非空值如下：

$$[\![\texttt{inc}]\!] = \{\&\texttt{inc}\} \qquad [\![\texttt{dec}]\!] = \{\&\texttt{dec}\}$$
$$[\![\texttt{ide}]\!] = \{\&\texttt{ide}\} \qquad [\![\texttt{foo}]\!] = \{\&\texttt{foo}\}$$
$$[\![\texttt{f}]\!] = \{\&\texttt{inc}, \&\texttt{dec}, \&\texttt{ide}\}$$

基于上述分析，我们可得到程序的monovariant版本的过程间CFG如下：



3.9 指针分析

3.9.1 指向分析

需要获取的最重要信息是指针可能指向的目标集，当然在执行时指针的指向有无穷多的可能，所以我们必须选择有限的表示。规范的选择是对每个名为id的变量引入一个目标$\&id$，而且为每个不同的内存分配点（程序执行一个malloc操作的地方也引入一个目标malloc-i，这里i是一个唯一的索引。我们使用$Targets$表示给定程序中所有可能的指针指向目标构成的集合。

指向分析也针对语法树，因为指向可能在控制流分析之前或同时进行。指向分析的最终结果是一个函数pt，对每个指针变量p，给出它可能指向的目标集合$pt(p)$。我们必须做保守的分析，因此这些集合通常会过大。基于指向信息，许多其他事实可以被近似，例如两个指针变量p和q是否可能互为别名，那么安全的回答可通过检查$pt(p) \cap pt(q)$是否为空集得到。

3.9.2 Andersen 算法

指向分析的方法与控制流分析的方法非常相似，对每个名为id的变量引入一个集合变量$[\![id]\!]$，取值范围为给定程序中可能的指针指向目标。指向分析简单程序已经被规范化，从而所有队指针的操作是下面六种情况之一：

$$\begin{array}{ll} 1)\ id = \texttt{malloc} & 2)\ id_1 = \&id_2 \\ 3)\ id_1 = id_2 & 4)\ id_1 = {}^*id_2 \\ 5)\ {}^*id_1 = id_2 & 6)\ id = \texttt{null} \end{array}$$

对每个这样的指针操作，我们生成下面的约束：

$$\begin{aligned} id = \texttt{malloc} &: \{\texttt{malloc-i}\} \subseteq [\![id]\!] \\ id_1 = \&id_2 &: \{\&id_2\} \subseteq [\![id_1]\!] \\ id_1 = id_2 &: [\![id_2]\!] \subseteq [\![id_1]\!] \\ id_1 = {}^*id_2 &: \&id \in [\![id_2]\!] \Rightarrow [\![id]\!] \subseteq [\![id_1]\!] \\ {}^*id_1 = id_2 &: \&id \in [\![id_1]\!] \Rightarrow [\![id_2]\!] \subseteq [\![id]\!] \end{aligned}$$

最后两个约束是针对每个名为id的变量，但实际上我们只需考虑哪些在给定程序中真正获取了它们地址的变量。空值null的赋值被忽略，因为它对应约束$\varnothing \subseteq [\![id]\!]$。因为上面的约束与立方算法的要求匹配，因此上述约束可以在$O(n^3)$的时间复杂度内求解。最后，指向函数可定义为：

$$pt(p) = [\![p]\!]$$

考虑下面的例子程序：

```
var p, q, x, y, z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

Andersen算法生成下面的约束：

$$\begin{aligned} \texttt{malloc-1} &\subseteq [\![\texttt{p}]\!] & [\![\texttt{y}]\!] &\subseteq [\![\texttt{x}]\!] \\ [\![\texttt{z}]\!] &\subseteq [\![\texttt{x}]\!] & \&\texttt{y} \in [\![\texttt{p}]\!] &\Rightarrow [\![\texttt{z}]\!] \subseteq [\![\texttt{y}]\!] \\ \&\texttt{z} \in [\![\texttt{p}]\!] &\Rightarrow [\![\texttt{z}]\!] \subseteq [\![\texttt{z}]\!] & [\![\texttt{q}]\!] &\subseteq [\![\texttt{p}]\!] \\ \{\&\texttt{y}\} &\subseteq [\![\texttt{q}]\!] & \&\texttt{y} \in [\![\texttt{p}]\!] &\Rightarrow [\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!] \\ \&\texttt{z} \in [\![\texttt{p}]\!] &\Rightarrow [\![\texttt{z}]\!] \subseteq [\![\texttt{x}]\!] & \{\&\texttt{z}\} &\subseteq [\![\texttt{p}]\!] \end{aligned}$$

最小解中的非空值是：

$$pt(\texttt{p}) = [\![\texttt{p}]\!] = \{\texttt{malloc-1}, \&\texttt{y}, \&\texttt{z}\} \qquad pt(\texttt{q}) = [\![\texttt{q}]\!] = \{\&\texttt{y}\}$$

这已经给出了非常精确的结果。注意，虽然这个算法是非流敏感的，但约束的方向意味着数据流在某种精确程度上得到了建模。

3.9.3 Steensgaard算法

一个常用的比较粗一点的替代分析可将赋值看做是双向的。我们使用的集合由malloc-i，以及对每个名为id的变量有id和 $^*id$。同样我们假设程序是经过规范化的，但这是对于符号生成等价型约束：

$$\begin{aligned} id = \texttt{malloc} &: {}^*id \sim \texttt{malloc-i} \\ id_1 = \&id_2 &: {}^*id_1 \sim id_2 \\ id_1 = id_2 &: id_1 \subseteq id_2 \\ id_1 = {}^*id_2 &: id_1 \subseteq {}^*id_2 \\ {}^*id_1 = id_2 &: {}^*id_1 \subseteq id_2 \end{aligned}$$

生成的约束到处一个在符号上的等价关系，它可以在线性时间内容计算，最后的指向函数定义为：

$$pt(p) = \{\&id \mid {}^*p \sim id\} \cup \{\texttt{malloc-i} \mid {}^*p \sim \texttt{malloc-i}\}$$

同样我们也可约束至那些$\&id$出现在程序中的变量。如果我们只考虑类型化的程序，那么可进一步删除那些类型不匹配的目标。对于前面的例子，Steensgaard算法生成如下约束：

$$\begin{aligned} {}^*\texttt{p} &\sim \texttt{malloc-1} & \texttt{p} &\sim \texttt{q} \\ \texttt{x} &\sim \texttt{y} & {}^*\texttt{q} &\sim \texttt{y} \\ \texttt{x} &\sim \texttt{z} & \texttt{x} &\sim {}^*\texttt{p} \\ {}^*\texttt{p} &\sim \texttt{z} & {}^*\texttt{p} &\sim \texttt{z} \end{aligned}$$

从而得到：

$$pt(\texttt{p}) = pt(\texttt{q}) = \{\texttt{malloc-1}, \&\texttt{x}, \&\texttt{y}, \&\texttt{z}\}$$

与Andersen算法对比，这显著降低了精度。如果只限于那些提取了地址的变量，我们得到：

$$pt(\texttt{p}) = pt(\texttt{q}) = \{\texttt{malloc-1}, \&\texttt{y}, \&\texttt{z}\}$$

这对于p来说与Anersen算法有相同的精度，但对于q来说仍比较差。

3.9.4 过程间指向分析

如果函数指针和其他指针是不同的，那么可以通过先按照前面所说的方法构建过程间CFG，然后再使用Andersen算法或Steensgaard 算法而完成过程间指向分析。但是，如果函数指针也可能有间接引用，我们可能需要同时指向控制流分析和指向分析。例如对于函数调用：

$$(\,^{***}\mathtt{x})(1,2,3)$$

为说明合并的算法，我们假设所有的函数调用具有如下形式：

$$id_1 = (id_2)(a_1, \cdots, a_n)$$

这里id_i和a_i都是变量。类似的，所有返回表达式都假设只是单个的变量。Andersen算法已经与控制流分析相似，因此它可通过增加合适的约束而简单地扩展。对一个函数的引用产生约束：

$$\{\&f\} \subseteq [\![f]\!]$$

通过函数指针调用函数产生如下约束：

$$\&f \in [\![id_2]\!] \Rightarrow [\![a_1]\!] \subseteq [\![x_1]\!] \wedge \cdots \wedge [\![a_n]\!] \subseteq [\![x_n]\!] \wedge [\![id]\!] \subseteq [\![id_1]\!]$$

这里函数f的声明为：

$$f(x_1, \cdots, x_n)\{\ldots \mathtt{return}\ id;\}$$

加入以上约束仍然维持控制流的分析。对于Steensgaard算法，将扩展下面的约束：

$$a_1 \sim x_1 \wedge \cdots \wedge a_n \sim x_n \wedge id \sim id_1$$

这将会丢失一些精度，因为每个n个参数的函数都被考虑为函数调用的目标。

3.9.5 空指针分析

现在可定义一个分析去检测空值解引用(`null` dereferences)。具体来说，我们希望确保 *`p`只有在`p`已经初始化且不指向`null`时才会执行。跟前面一样，我们假定程序是规范化的，因此所有的指针操作都是下面六种形式之一：

1) $id = \mathtt{malloc}$
2) $id_1 = \&id_2$
3) $id_1 = id_2$
4) $id_1 = {}^{*}id_2$
5) ${}^{*}id_1 = id_2$
6) $id = \mathtt{null}$

我们使用的基本格称为$Null$：

这里IN表示已经初始化，而NN表示非空值(not null)，然后形成了映射格：

$$Vars \mapsto Null$$

这里$Vars$是声明中程序中的变量集合。对每个控制流图节点v引入约束变量$[\![v]\!]$给出在这个节点对应的程序点每个变量的状态。对于变量声明有：

$$[\![v]\!] = [id_1 \mapsto \texttt{?}, \cdots, id_n \mapsto \texttt{?}]$$

对各种指针操作，我们有下面的约束：

$$\begin{aligned}
id = \texttt{malloc} &: [\![v]\!] = JOIN(v)[id \mapsto \texttt{NN}] \\
id_1 = \&id_2 &: [\![v]\!] = JOIN(v)[id_1 \mapsto \texttt{NN}] \\
id_1 = id_2 &: [\![v]\!] = JOIN(v)[id_1 \mapsto JOIN(v)(id_2)] \\
id_1 = {}^*id_2 &: [\![v]\!] = right(JOIN(v), id_1, id_2) \\
{}^*id_1 = id_2 &: [\![v]\!] = left(JOIN(v), id_1, id_2) \\
id = \texttt{null} &: [\![v]\!] = JOIN(v)[id \mapsto \texttt{IN}]
\end{aligned}$$

而对于所有其他的节点则有

$$[\![v]\!] = JOIN(v)$$

这里定义：

$$\begin{aligned}
JOIN(v) &= \bigsqcup_{w \in pred(v)} [\![w]\!] \\
right(\sigma, x, y) &= \sigma[x \mapsto \sigma(y) \sqcup \bigcup_{\&p \in pt(y)} \sigma(p)] \\
left(\sigma, x, y) &= \sigma_{\&p \in pt(x)}[p \mapsto \sigma(p) \sqcup \sigma(y)]
\end{aligned}$$

注意分配的位置总是映射为$\perp$，这表明我们不跟踪堆的大小和堆的内容之间的连接关系。通过分析，在程序点v处${}^*p$的计算是安全的，如果$[\![v]\!](p) = \texttt{NN}$。上述分析的准确性显然依赖于指向分析的质量。考虑下面由缺陷的例子程序：

```
var p, q, r, n;
p = malloc;
q = &p
n = null
*q = n
*p = r
```

Andersen算法将计算出下面的指向集合：

$$\begin{aligned}
pt(\texttt{p}) &= \{\texttt{malloc-1}\} & pt(\texttt{q}) &= \{\&\texttt{p}\} \\
pt(\texttt{r}) &= \{\} & pt(\texttt{n}) &= \{\}
\end{aligned}$$

基于这些信息，空指针分析将产生下面的约束：

$$
\begin{aligned}
[\![\texttt{var p, q, r, n}]\!] &= [\texttt{p} \mapsto ?, \texttt{q} \mapsto ?, \texttt{r} \mapsto ?, \texttt{n} \mapsto ?] \\
[\![\texttt{p} = \texttt{malloc}]\!] &= [\![\texttt{var p, q, r, n}]\!][\texttt{p} \mapsto \texttt{NN}] \\
[\![\texttt{q} = \&\texttt{p}]\!] &= [\![\texttt{p} = \texttt{malloc}]\!][\texttt{q} \mapsto \texttt{NN}] \\
[\![\texttt{n} = \texttt{null}]\!] &= [\![\texttt{q} = \&\texttt{p}]\!][\texttt{n} \mapsto \texttt{IN}] \\
[\![\ ^*\texttt{q} = \texttt{n}]\!] &= [\![\texttt{n} = \texttt{null}]\!][\texttt{p} \mapsto [\![\texttt{n} = \texttt{null}]\!](\texttt{p}) \sqcup [\![\texttt{n} = \texttt{null}]\!](\texttt{n})] \\
[\![\ ^*\texttt{p} = \texttt{r}]\!] &= [\![\ ^*\texttt{q} = \texttt{n}]\!]
\end{aligned}
$$

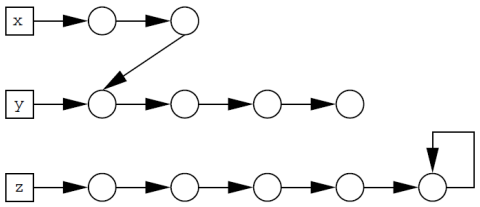
而求解得到的最小解是：

$$
\begin{aligned}
[\![\texttt{var p, q, r, n}]\!] &= [\texttt{p} \mapsto ?, \texttt{q} \mapsto ?, \texttt{r} \mapsto ?, \texttt{n} \mapsto ?] \\
[\![\texttt{p} = \texttt{malloc}]\!] &= [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto ?, \texttt{r} \mapsto ?, \texttt{n} \mapsto ?] \\
[\![\texttt{q} = \&\texttt{p}]\!] &= [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto \texttt{NN}, \texttt{r} \mapsto ?, \texttt{n} \mapsto ?] \\
[\![\texttt{n} = \texttt{null}]\!] &= [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto \texttt{NN}, \texttt{r} \mapsto ?, \texttt{n} \mapsto \texttt{IN}] \\
[\![\ ^*\texttt{q} = \texttt{n}]\!] &= [\texttt{p} \mapsto \texttt{IN}, \texttt{q} \mapsto \texttt{NN}, \texttt{r} \mapsto ?, \texttt{n} \mapsto \texttt{IN}] \\
[\![\ ^*\texttt{p} = \texttt{r}]\!] &= [\texttt{p} \mapsto \texttt{IN}, \texttt{q} \mapsto \texttt{NN}, \texttt{r} \mapsto ?, \texttt{n} \mapsto \texttt{IN}]
\end{aligned}
$$

基于上述信息，编译器可静态检测出在执行 $^*\texttt{p} = \texttt{r}$时，变量p可能包含null，而变量r可能没有被初始化。

3.9.6 形状分析

上面将堆存储看做是没有组织的一种结构，而且只回答基于栈的变量的问题。堆可使用形状分析(shape analysis)做更详细的分析。即使TIP语言中的malloc只分配一个堆单元，仍可能产生有趣的堆结构。一个非平凡堆结构的例子如下：



这里x, y和z是程序变量。我们试图去回答包含在程序变量中的结构之间的互斥性(disjointness)，在上面的例子中，x和y不互斥，而y和z互斥。

堆的形状分析需要称为形状图(shape graphs)的元素构成的格。形状图是有向图，其节点是给定程序的指针指向目标。形状图之间的序由它的边集之间的子集关系确定。因此$\bot$是没有任何边的图，而$\top$是全连接图（有向完全图）。指针指向目标看做是在程序执行期间所有可能创建的堆单元的抽象，而它们之间的边意味着存储可能包含代表源和目标的两个节点之间的一个引用。正式地来说，我们的格是：

$$2^{Targets \times Targets}$$

序是通常的子集包含关系。对任意的CFG节点v，引入一个约束变量$[\![v]\!]$表示一个描述在这个节点对应程序点之后的所有可能存储的形状图。对于那些对应各种指针操作的节点，有如下约束：

$$\begin{aligned}
id = \texttt{malloc} &: [\![v]\!] = JOIN(v) \downarrow id \cup \{(\&id, \texttt{malloc-i})\} \\
id_1 = \&id_2 &: [\![v]\!] = JOIN(v) \downarrow id_1 \cup \{(\&id_1, \&id_2)\} \\
id_1 = id_2 &: [\![v]\!] = assign(JOIN(v), id_1, id_2) \\
id_1 = {}^*id_2 &: [\![v]\!] = right(JOIN(v), id_1, id_2) \\
{}^*id_1 = id_2 &: [\![v]\!] = left(JOIN(v), id_1, id_2) \\
id = \texttt{null} &: [\![v]\!] = JOIN(v) \downarrow id
\end{aligned}$$

而对其他节点都有约束：

$$[\![v]\!] = JOIN(v)$$

这里定义：

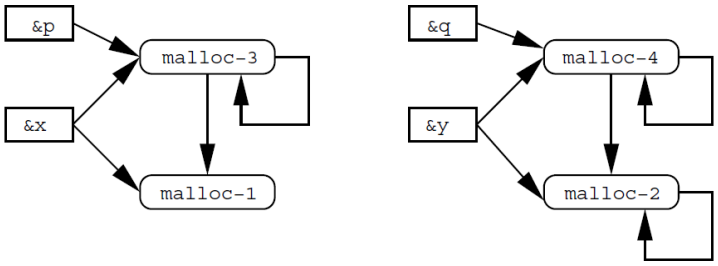
$$\begin{aligned}
JOIN(v) &= \bigcup_{w \in pred(v)} [\![w]\!] \\
\sigma \downarrow x &= \{(s,t) \in \sigma \mid s \neq \&x\} \\
assign(\sigma, x, y) &= \sigma \downarrow x \cup \bigcup_{(\&y,t) \in \sigma} \{(\&x, t)\} \\
right(\sigma, x, y) &= \sigma \downarrow x \cup \bigcup_{(\&y,s),(s,t) \in \sigma} \{(\&x, t)\}
\end{aligned}$$

$$left(\sigma, x, y) = \begin{cases}
\sigma & \{s \mid (\&x, s) \in \sigma\} = \varnothing \\
\bigcup_{(\&x,s) \in \sigma} \sigma \downarrow s & \{s \mid (\&x, s) \in \sigma\} \neq \varnothing wedge \{t \mid (\&y, t) \in \sigma\} = \varnothing \\
\bigcup_{(\&x,s),(\&y,t) \in \sigma} \sigma \downarrow s \cup \{(s,t)\} & otherwise
\end{cases}$$

考虑下面的程序：

```
var x,y,n,p,q;
x = malloc; y = malloc;
*x = null; *y = y;
n = input;
while (n>0) {
  p = malloc; q = malloc;
  *p = x; *q = y;
  x = p; y = q;
  n = n-1;
}
```

在程序中的循环之后，分析将产生如下的形状图：



根据这个分析结果，我们可以安全地断言x和y总是互斥的。

注意形状分析也为每个程序点v计算了一个流敏感的指针指向图，定义为：

$$pt(p) = \{t \mid (\&p, t) \in [\![v]\!]\}$$

形状分析比Andersen算法得到的指向关系更精确，但显然也需要更高的代价去执行。作为例子，考虑下面的程序：

$$\texttt{x} = \&\texttt{y}$$
$$\texttt{x} = \&\texttt{z}$$

在这些语句之后的程序点，Andersen算法将预言$pt(\texttt{x}) = \{\&y, \&z\}$，但形状分析将得到$pt(\texttt{x}) = \{\&z\}$。这种流敏感的指向信息可用于驱动空指针分析(null pointer analysis)，但是一个初始的非流敏感的指向分析在为使用了函数指针的程序构造CFG时仍是需要的。反之，如果我们有另外的指向分析，那么它可能可通过限制在上面的函数$left$和$right$中的目标而得到更精确的形状分析。

3.9.7 更好的形状分析

上面的形状分析允许我们断言x和y总是互斥的，但是上面给出的形状图不能回答一些其他有趣的问题。例如，无法断言malloc-2总是包含一个自循环(self-loop)，而malloc-4永远不会出现在回路(cycles)中。为了得到这些信息，我们需要更细化的格。作为例子，如果我们想让维护图中的回路性质(cyclicity)，我们使用的格需要是：

$$2^{Targets \times Targets} \times 2^{Targets \times Targets}$$

这里对于其中的元素(X, Y)，X表示可能的边，而$Y \subseteq X$表示那些可能在图中作为回路一部分的边（因此Y用于记住堆的部分历史）。

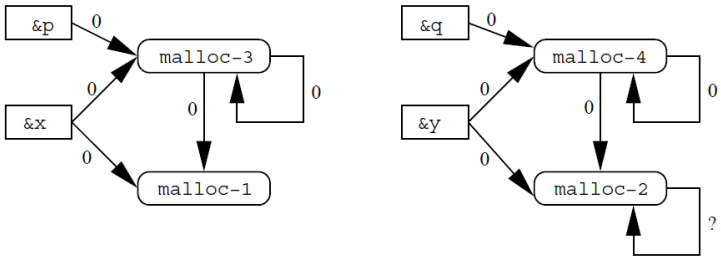
基于这更细化的格，我们需要对约束方程也做相应的修改。当然，我们可以平凡地将第二分量（即前面的Y）包含所有的边，但为了得到更有用结果，我们需要做得更好。赋值$id = \texttt{malloc}$永远不会创建一个回路，因此相应的约束可如下修改（这里假定$JOIN(v) = (X, Y)$）：

$$id = \texttt{malloc} \ : \ [\![v]\!] = (X \downarrow id \cup \{(\&id, \texttt{malloc-i})\}, Y \downarrow id)$$

赋值$id_1 = \&id_2$当当前的形状图不包括从$\&id_2$到$\&id_1$的路径时也不会创建回路（因为形状图中的可达性是保守的(conservative)）。因此相应的约束方程修改为：

$$id_1 = \&id_2 \ : \ [\![v]\!] = (X \downarrow id_1 \cup \{(\&id_1, \&id_2)\}, Y \downarrow id_1 \cup reach(X \downarrow id_1, \&id_2, \&id_1))$$

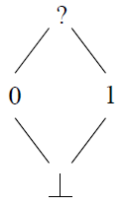
这里$reach(\sigma, s, t)$返回$\{(t, s)\}$如果s可通过零条过多条在σ中的边到达t，否则返回$\varnothing$。其他的约束也可做相似的修改。基于更详细的分析，针对前面的例子程序，我们可计算得到下面的形状图：



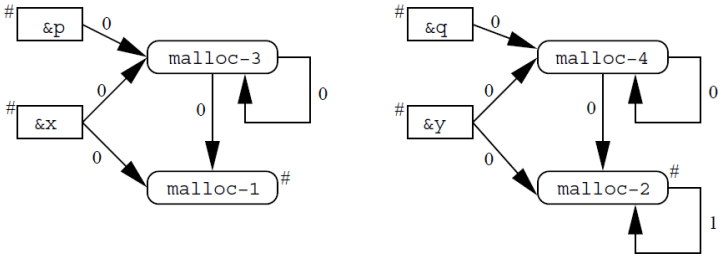
其中边上的标记0意味着这条边肯定不会出现回路中，而?表示相应的答案是不知道，从而现在可以断定`malloc-4`不会出现在回路上。对于`malloc-2`，它可能出现在回路上，但是目前还没有足够的信息表明会形成一个自环(self-loop)。为得到这个信息，需要将格进一步细化去跟踪一条边是否确定作为一个回路的一部分，以及去跟踪对于给定的目标是否多与一个节点曾被分配。这需要使用如下形式的格：

$$2^{Targets \times Targets} \times 3^{Targets \times Targets} \times 2^{Targets}$$

这里使用3^A表示格$A \mapsto 3Val$，这里$3Val$是下面的3-值逻辑格：



这里0表示肯定为假，1表示肯定为真，而?表示未知。对于一个格元素(X, Y, Z)，X表示形状图的边，Y表示关于边的回路性（可能或必然）知识，而Z表示那些只曾经分配了一个实例的指针指向目标。这样可定义更复杂的约束去维护这些信息，从而分析可得到下面的信息（其中#表示一个被唯一分配的目标(a target is uniquely allocated)）：



这里最后我们可断定只有一个`malloc-2`节点，而且它有一个自环。

上面的分析也可使用一个称为参数化形状分析(parametric shape analysis)的框架去完成。这里目标被一系列一元置入谓词去刻画(the targets are characterized by a number of unary instrumentation predicates)，这些谓词可提供分析所需的必要信息，例如：

(1) 这个节点是否有两个或多个指向的指针(incoming pointers)?

(2) 这个节点是否可从变量`x`可达?

(3) 这个节点是否在一个回路上?

使用怎样的谓词依赖我们所需要寻找的答案。简单的形状图中每个指针指向目标有一个单一节点，但在参数化框架中，对谓词的每个可能的3-值解释都有一个拷贝，因此一个形状图的节点对应：

$$3^{Targets} \times 3^{Targets} \times \cdots \times 3^{Targets}$$

一个每个谓词的一个拷贝。形状图自己描述节点之间的连通性，而3-值逻辑描述边的确定存在、确定不存在以及可能存在，因此最后使用的格是：

$$3^{(3^{Targets} \times 3^{Targets} \times \cdots \times 3^{Targets})^2}$$

或简写为（如果有k个谓词）：

$$3^{((3^{Targets})^k)^2}$$

为完成参数化形状分析，必须给出约束。如果包含多个谓词，那么需要很大的负担去维持它们的精确性，这通常即使对于上面简单的例子，其约束方程也会很复杂。但是这个技术很有威力，可能可用于验证一个对红黑搜索树求红黑不变式的过程。

3.9.8 逃逸分析

前面给出了一个有逃逸栈单元(escaping stack cell)错误的程序：

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p=baz(); *p=1;
  return *p;
}
```

这个错误不能通过类型系统检查。通过执行简单的形状分析，我们可执行一个逃逸分析(escape analysis)去发现这类错误，只需为返回表达式检查哪些不能到达参数或函数中定义的变量的可能的指针指向目标，因为其他的指针指向目标已经在调用栈的早先的框架中。

3.10 结论

上面给出了执行程序静态分析所需的基本工具。实际应用基本上可归结为上面的技术，但当然有各种变化和扩展。

有两个主要的论题在上面没有提到。分析的质量(quality)只能相对于所期望的应用而进行度量，通常很难去形式化地比较同类分析，因此通常去执行一些实验取建立所提出分析的精确性和效率。分析的正确性(correctness)通常所分析程序设计语言的形式语义。对分析正确性的形式化定义通常是非常困难的，但还是需要提供对正确性的令人信服的论证。

参考文献

[1] Kenneth C. Louden and Kenneth A. Lambert. *Programming Languages: Principles and Practices*. Course Technology, Cengage Learning, 3rd edition, 2012.

[2] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 10th edition, 2012.

[3] Tenrence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[4] Tenrence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. The Pragmatic Bookshelf, 2010. 中译本: 李袁奎译. 编程语言实现模式. 华中科技大学出版社, 2012.

[5] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[6] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–1296, 1998.

[7] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems*, 19(6):1031–1052, 1997.

[8] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[9] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Compact representations for control dependence. In Bernard N. Fischer, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 337–351, White Plains, New York, USA, June 20-22 1990. ACM.

[10] Mary Jean Harrold and Gregg Rothermel. Syntax-directed construction of program dependence graphs. Technical report, Department of Computer and Information Science, The Ohio State University, 19??