语义定义

语义定义：目标是赋予每个合法的程序一个明确的意义

- 精确定义程序执行的效果（行为，结果……）
- 要求编译程序（等）生成这种效果
- 写程序的人可以依赖于这种效果，据此写自己的程序

语义定义的基本方法是基于语言的语法结构：

- 定义语言中各种基本元素的意义
- 定义语言中各种语法结构的意义

  基于相应结构的成分的意义，定义整个结构的意义

一些具体情况：

- 表达式的意义是其怎样求值，求出什么值
- 语句的意义是其执行时的产生的效果

语义定义

Algol 60 修订报告

- 用 **BNF** 定义语言的语法，
- 采用自然语言形式的说明和实例的方式，非形式地定义语言的语义

Backus 在讨论 **Algol 60** 时说：

> 现在我们已经有办法严格地定义语言的语法了，希望今后不久就能严格地定义语言的语义

很遗憾，至今的语言手册仍一直沿用上述方式。因为：

- 语义远比语法复杂，至今计算机工作者还没有开发出一种完美、功能强大、易理解、易使用，适用于定义一切语言中一切结构的语义描述方式
- 但是，对于程序语义的研究已经得到许多成果。有关程序语言语义的研究是计算机科学的一个重要研究领域
- 后面会介绍一些简单情况

静态语义：类型的理论

类型是程序设计语言的一个重要问题

关于语言中的类型的各方面具体问题将在后面详细讨论

程序语言需要有清晰定义的类型系统，以确定程序里

- 各表达式的类型（typing）。注意，表达式可能嵌套
- 各语句是否类型良好的（well-typed），或称为良类型的
- 处理程序中与类型有关的各种问题：类型等价、类型相容、类型转换（自动转换规则，手工转换规则）等

有关类型的研究形成了计算机科学中一个重要的研究领域：类型理论

类型理论基础的专著：**Pierce, Benjamin C., Types and Programming Languages, MIT Press, 2002.** 中译本（电子工业出版社）译的不好

类型理论采用形式化的严格方法，研究与类型有关的理论问题。本课程不准备深入讨论，但简单介绍类型理论的一些基本想法

类型环境和类型断言

类型判断的根据：

- 每个文字量都有自己的确定类型
- 变量、函数等的声明提出了相关的类型要求
- 对类型合适的参数，运算符、函数将给出特定类型的结果（注意：重载的运算符也有运算结果和运算对象之间的关系）

类型检查需要知道程序里所有变量的类型信息。当前所有可见变量的类型信息的全体构成了当前的类型环境

一个变量的类型信息用变量名和类型的对表示，如 **x** 具有类型 **T**：

$$x : T$$

类型环境就是这种对的序列，如：

$$x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n$$

下面用 Γ （可能加下标）表示类型环境

类型推导

我们用形式化的记法表示有关类型的断言。类型断言有两种形式：

$\Gamma \vdash e : T$　在特定类型环境 Γ 下，我们确定了表达式 *e* 的结果类型是 *T*（表达式：常量、变量、有结构成分的复杂表达式）

$\Gamma \vdash c : \mathsf{ok}$　在环境 Γ 下命令（语句）c 是类型良好的（基本语句如赋值，各种复合语句和更复杂的程序结构）

为了作出程序中各种结构是否类型良好的判断，确定有类型结构（表达式）的具体类型，要定义一套做类型推断的规则（类型规则），说明如何

- 得到程序中基本表达式的类型
- 基于表达式的部分的类型推出整个表达式的类型
- 然后基于表达式的类型，推断基本语句是否类型良好；基于基本语句的良类型性得到复合语句的良类型性

不能推导出类型的表达式是类型错误的表达式，对语句也一样

类型规则

下面用一个简单语言介绍类型规则的一些定义。假定语言里有

- **int** 和 **double** 类型
- 变量声明，赋值语句和结构语句

类型规则包括（例子）：

- 基本规则（1）

$$\vdash 0 : \mathsf{int}$$
$$\cdots\cdots$$
$$\vdash 0.0 : \mathsf{double}$$
$$\cdots\cdots$$
$$\Gamma, x : T \vdash x : T$$

- 基本规则（2）

$$\frac{\Gamma \vdash e : T \quad y \text{ is not in } e}{\Gamma, y : T_1 \vdash e : T}$$

类型规则

表达式的类型规则（一组）：

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

$\cdots\cdots$

语句的类型规则（一组）：

$$\frac{\Gamma \vdash x : \mathsf{int} \quad \Gamma \vdash e : \mathsf{int}}{\Gamma \vdash x := e; : \mathsf{ok}}$$

$$\frac{\Gamma \vdash b : \mathsf{bool} \quad \Gamma \vdash s_1 : \mathsf{ok} \quad \Gamma \vdash s_2 : \mathsf{ok}}{\Gamma \vdash \mathsf{if}\ (b)\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 : \mathsf{ok}}$$

$\cdots\cdots$

变量声明：

$$\frac{\Gamma \vdash T\ x; : \mathsf{ok} \quad \Gamma, x : T \vdash c : \mathsf{ok}}{\Gamma \vdash T\ x; c : \mathsf{ok}}$$

其他规则可以类似写出

语义基础：环境和状态

考虑程序的（动态）语义（程序执行的语义）

- 下面希望用清晰严格的方式描述程序执行中各种动作的效果
- 对程序动态行为的理解和严格描述依赖于环境的概念

程序执行过程中存在着一些实体（变量、子程序等）

- 可能存在与这些实体相关的信息（例如，变量的值，函数的定义）
- 程序里动作的执行可能改变与一些实体相关的信息，如改变变量的值
- 环境记录与各种有效实体相关的信息
- 因此，动作的执行可能改变环境

讨论语义时关心的是程序的动态环境（运行中的环境）

编译（和类型检查）工作中维护的符号表是支持语言处理的“静态环境”

最简单的环境模型：从合法名字集合到“值”的映射（是一个有限函数）

环境和状态

程序执行的动作可能依赖于环境，或导致环境的变化：

- 需要某变量的值时，从当时环境里取出它的值
- 一些操作修改环境中变量的值，如赋值、输入等
- 进入或退出作用域（如进入/退出子程序或内部嵌套的复合语句），可能改变当前有定义的变量集合（改变环境的定义域）
- 程序开始运行前，需要建立一个初始全局环境，其中包含着所有具有全局定义的名字及与之相关的默认信息

下面把环境建模为变量名到其取值的简单的有限函数

- 如果研究的是更复杂的语言，这种简单形式就可能不够了，需要考虑更复杂的环境模型
- 例如包含指针和动态存储分配/释放，或面向对象的语言（OO 语言），就需要为更复杂的环境模型

在程序运行中，当时环境中所有可用变量的取值情况构成了运行时的状态

环境和状态

例：设当时环境中可见的变量是 x, y, z，取值情况是

x 取值 3，y 取值 5，z 取值 0

这个状态就是有限函数：

$$\{x \mapsto 3, y \mapsto 5, z \mapsto 0\}$$

随着程序的执行，特别是赋值语句的执行，环境的状态就可能改变

经过给 x 赋值 4，给 y 赋值 1 的两个操作，状态将变成：

$$\{x \mapsto 4, y \mapsto 1, z \mapsto 0\}$$

理解一个（或一段）程序的意义，就是理解它在有关环境下的意义，以及它的执行对环境的作用

环境和状态

严格些说，一个环境是一个“部分函数”：

$$\sigma \ : \ Name \rightarrow Val \cup \{\perp\}$$

一个环境　名字的集合　“值”集合　无定义

为描述状态变化，在环境上定义了一种覆盖操作：

$$\sigma \oplus \sigma'(x) = \begin{cases} \sigma'(x) & \text{if } x \in \text{dom } \sigma' \\ \sigma(x) & \text{otherwise} \end{cases}$$

例：

$$\{x \mapsto 1, y \mapsto 2, z \mapsto 3\} \oplus \{x \mapsto 0\}$$
$$= \{x \mapsto 0, y \mapsto 2, z \mapsto 3\}$$

环境和表达式

对于常量（如整数），任何环境都给出它们的字面值

环境确定一组变量的值，就确定了基于这些变量构造的表达式的意义

假定　$\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$

我们有：$\sigma(x) = 1 \qquad \sigma(y) = 2 \qquad \sigma(z) = 3$

对于一般表达式：

$$\sigma(e_1 + e_2) = \sigma(e_1) + \sigma(e_2)$$
$$\sigma(e_1 * e_2) = \sigma(e_1) \times \sigma(e_2)$$
$$\ldots\ldots$$

由于环境可以确定表达式的意义（确定表达式的值）。我们也可以把环境看成从合法表达式到值集合的映射，　$\sigma \ : \ Exp \rightarrow Val \cup \{\perp\}$

有些表达式的值无定义，例如其中出现了在环境取值为无定义的变量，或者求值中出现无定义运算结果（如除数为 0）等

环境和语句等

我们把语句的意义定义为它们对环境的作用。为定义语义需要：

- 定义基本语句的意义
- 定义各种结构的意义如何由它们的组成部分得到

有了这些定义，就可以确定由这些语言结构构成的复杂程序的意义

一个程序的执行导致环境的变化，问题是如何严格定义这种变化

人们提出了多种不同的形式化的语义定义方式（技术）：

- 操作语义：把程序运行看成执行了一系列改变环境的操作，用一组描述环境变化的规则定义程序中各种结构的意义
- 公理语义：用谓词描述状态，用公理和推理规则描述各种基本操作和组合结构的语义
- 指称语义：把程序的意义映射到某种的数学结构，这种结构能清晰地表达程序对环境的作用和环境的变化
- 代数语义：考虑程序之间的等价关系，间接定义程序的意义

操作语义

操作语义的想法很自然，有许多与之相关的朴素想法：

- “要弄清这段程序的意思，去看它的编译结果，看它执行时做什么”
- “一步一步弄清楚它做什么，你就知道这个程序的意思了”

但这种做法并不是定义。人们希望在某种抽象层面上把握程序的语义

操作语义的开创性工作是 **John McCarthy** 用 **Lisp** 语言本身定义了 **Lisp** 语言的执行过程（语义）

人们提出了抽象机的概念，并提出用抽象机定义程序语言的语义

定义性解释器（**Definitional Interpreter**），用语言解释器给出语言的定义

Plotkin 提出了结构化操作语义（**Structural Operational Semantics, SOS**）

SOS被广泛用于程序和软件的理论研究，用于严格描述各种结构的语义

有些 **Web** 标准语言的规范里用 **SOS** 定义语言的语义

操作语义：格局和规则

首先需要定义格局（configuration）。这里定义两种格局：

- 终止格局，就是一个环境，表示代码运行完成时的状态
- 非终止格局： p, σ 表示程序运行中的一个“状态”

一段程序代码（待执行代码段）和一个环境

结构化操作语义（SOS）用一组格局变迁规则定义语言的语义。可能包括一些无前提的规则和一些有前提的规则

赋值语句的语义规则（无前提规则）：

$$x := e, \sigma \rightsquigarrow \sigma \oplus \{x \mapsto \sigma(e)\}$$

复合语句的语义规则（有前提规则）：

$$\frac{p_1, \sigma \rightsquigarrow \sigma'}{p_1; p_2, \sigma \rightsquigarrow p_2, \sigma'}$$

操作语义：续

if 条件语句的语义规则（两条）：

$$\frac{\sigma(b) = \mathsf{true} \quad p_1, \sigma \rightsquigarrow \sigma'}{\texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2, \sigma \rightsquigarrow \sigma'}$$

$$\frac{\sigma(b) = \mathsf{false} \quad p_2, \sigma \rightsquigarrow \sigma'}{\texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2, \sigma \rightsquigarrow \sigma'}$$

while 循环语句的语义规则（两条）：

$$\frac{\sigma(b) = \mathsf{true} \quad p, \sigma \rightsquigarrow \sigma'}{\texttt{while } b \texttt{ do } p, \sigma \rightsquigarrow \texttt{while } b \texttt{ do } p, \sigma'}$$

$$\frac{\sigma(b) = \mathsf{false}}{\texttt{while } b \texttt{ do } p, \sigma \rightsquigarrow \sigma}$$

上面这组规则严格地定义了一个理想的小语言的语义

注意这个小语言的特点。它与实际的程序语言还是有很大差距

公理语义

R. Floyd 在**1967**年考虑如何说明一个程序完成了所需工作时，提出用逻辑公式描述程序环境的状态的思想。这种描述环境的逻辑公式称为断言

逻辑公式：$a_1: \ x = 1 \wedge y = 2 \wedge z = 3$

描述了状态：$\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$

说环境满足该公式，记为：$\sigma \models a_1$

实际上，这个状态还满足另外的许多逻辑公式，如：

$a_2: \ x = 1$

$a_3: \ x > 0 \wedge y = 2 \wedge z \geq 2$

反过来看，一个逻辑公式可以被许多状态满足

因此可以认为，一个公式描述了一个状态集合。例如，上面第二个公式，描述了所有的其中 ***x*** 值是 **1** 的环境

公理语义：断言

把断言写在程序里某个特定位置，就是想提出一个要求：

- 程序执行到这个位置，当时的环境状态必须满足断言
- 如果满足，程序就正常向下执行（就像没有这个断言）
- 如果不满足，程序就应该（非正常地）终止

实例：

- **C** 语言标准库提供了断言宏机制，可以在程序里的写

```
assert(x > 0);
```

- **Stroustrup** 在《**The C++ Programming Language**》里特别讨论了如何定义断言类的问题
- 有些语言本身提供了断言机制，如 **Eiffel**。见《**Design by Contract**》，中译本：邮电出版社

断言的概念在实际程序开发中起着越来越重要的作用

公理语义：前条件和后条件

如何借助于逻辑公式描述一段程序的意义？

Hoare 提出的方法是用一对公式，放在相应程序段之前和之后，分别称为该程序段的前条件（**precondition**）和后条件（**postcondition**）

$$pre \;\; P \;\; post$$

意思是：如果在程序 **P** 执行之前条件 ***pre*** 成立（当时的环境满足 ***pre***），那么在 **P** 执行终止时的环境中，条件 ***post*** 一定成立

我们可以用一对公式 **(*pre*, *post*)** 描述一段程序的语义，也可以把这样的公式对 **(*pre*, *post*)** 看作是对一段程序的语义要求，看作程序的规范

前后条件的两种基本用途：

- 作为评判程序是否正确的标准
- 作为待开发的程序的规范，研究如何从这种规范得到所需的程序

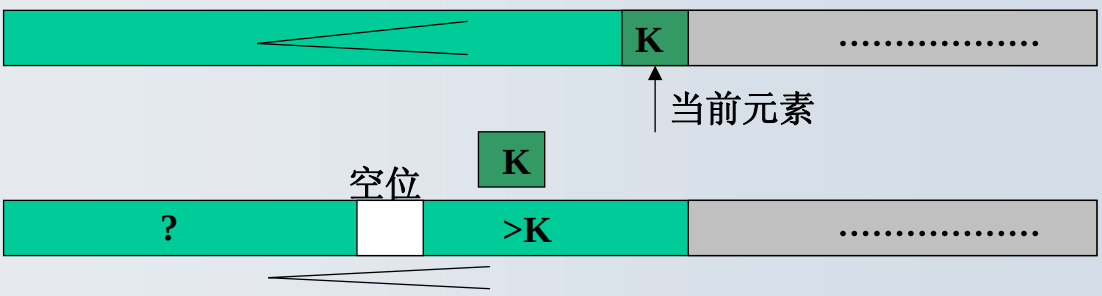
公理语义：不变式

不变式（不变量，**invariant**）的概念在许多领域中有重要地位

- 程序理论中的不变式，指写在程序里特定位置的逻辑公式，要求程序执执行中每次达到某个（某些）情况时，这个公式都必须成立
- 其实，不变式并不特殊，因为写在程序里的断言，也就是位于代码中相应特定位置的不变式，“情况”就是执行达到这个位置

程序理论里最重要的不变式之一是循环不变式（**loop invariant**），要求一个循环的每次迭代开始之前成立，在描述和推导程序的意义时有特殊价值

直接插入排序算法两重循环的不变式（图示）：



公理语义：程序的意义

Floyd 最早提出用断言描述程序的意义，通过逻辑推导证明程序的正确性。他称自己提出的方法为“断言法”。基本方法：

- 在流程图程序里的每条边上放一条断言
- 设法证明从一个动作框的入边上的断言出发，执行相应动作产生的效果能保证该动作框的出边上的断言成立
- 对分支控制框也有特殊的规则（请自己考虑）
- 如果对每个动作框都能证明上述事实成立，那么标注在流程图中各条边上的断言就形成了该程序的一个（一套）语义描述
- 特别的，标在程序入口和出口上的断言表示了这段程序的整体效果

Hoare 总结了 **Floyd** 的工作，针对结构化的程序控制结构提出了一套逻辑推理规则，这就是有名的 **Hoare** 逻辑

公理语义：Hoare逻辑

Hoare 逻辑是对有关程序意义的逻辑描述的整理和系统化

Hoare 逻辑中的逻辑公式形式为（称为 **Hoare** 三元组）：

$$\{p\}\ S\ \{q\}$$

其中 **p** 和 **q** 是谓词逻辑公式，**S** 是一段程序

直观意义：如果在 **S** 执行之前公式 **p** 成立，在 **S** 执行终止时 **q** 就成立

Hoare 的最重要贡献是提出了一套推理规则，通过这些规则，可以把证明一个 **Hoare** 公式（程序 **S** 相对于 **p** 和 **q** 的正确性）的问题归结为证明一组普通一阶谓词逻辑公式的问题

Dijkstra 在此基础上提出了最弱前条件（**weakest precondition**）的概念，借助于这一概念，可以

- 把程序的正确性证明工作进一步规范化
- 用于做程序的推导（从规范出发推导程序）

公理语义：Hoare逻辑

公理：

[SKIP] $\{p\}\ \texttt{skip}\ \{p\}$

[ASSIGN] $\{p[e/x]\}\ x := e\ \{p\}$

[COMP] $$\frac{\{p\}\ S_1\ \{q\} \quad \{q\}\ S_2\ \{r\}}{\{p\}\ S_1; S_2\ \{r\}}$$

[COND] $$\frac{\{p \wedge b\}\ S_1\ \{q\} \quad \{p \wedge \neg b\}\ S_2\ \{q\}}{\{p\}\ \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{q\}}$$

[LOOP] $$\frac{\{I \wedge b\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ b\ \texttt{do}\ S\ \{I \wedge \neg b\}}$$

[IMPLY] $$\frac{p \Rightarrow p' \quad \{p'\}\ S\ \{q'\} \quad q' \Rightarrow q}{\{p\}\ S\ \{q\}}$$

规则 **[LOOP]** 里的 ***I*** 是循环不变式

公理语义：Hoare逻辑

可以用 **Hoare** 逻辑证明程序的正确性，也就是说，证明三元组中的程序语句“符合”前后条件的要求。其意义是：

若在前条件满足的情况下执行语句且语句执行终止，那么后条件满足

- 这个证明并不保证语句终止，如果语句的执行不终止，什么后条件都可以证明。因此，这样证明的正确性称为“部分正确性”
- 程序终止性需要另外证明，主要是需要证明其中循环的执行必然终止
- 如果同时证明程序的部分正确性和终止性，这一程序就是“完全正确的”

程序正确性证明中的一个关键点是为各个循环提供适当的不变式

就像做数学归纳法证明中需要合适的归纳假设，过强或过弱都不行

人们已证明，循环不变式不可能自动生成（无完全的算法，但有许多研究）

公理语义为我们提供了许多有助于理解程序行为的概念，为设计程序时思考其行为提供了重要的依据和线索。很重要

指称语义

指称语义学（**denotational semantics**）有坚实的数学基础，它的基本想法由 **C. Strachey** 提出，**D. Scott** 完成了它的基础理论并因此获图灵奖

指称语义的基本思想是定义一个语义函数（指称函数），把程序的意义映射到某种意义清晰的数学对象（就像用中文解释英文）

作为被指的数学对象可以根据需要选择

对简单的程序语言，一种自然的选择是把程序的语义定义为从环境到环境的函数（作为指称）。定义语义就是指定每个程序对应的函数

环境的集合： $\Sigma \qquad \sigma \in \Sigma$

语义映射： $[\![.]\!]$

表达式的语义： $[\![e]\!] \in \Sigma \to \mathbb{Z}$ 假设是整型表达式

命令的语义： $[\![S]\!] \in \Sigma \to \Sigma$

指称语义

表达式的语义定义：

$$[\![n]\!](\sigma) \widehat{=} n \quad n \in \mathbb{Z}$$
$$[\![x]\!](\sigma) \widehat{=} \sigma(x)$$
$$[\![e_1 + e_2]\!](\sigma) \widehat{=} [\![e_1]\!](\sigma) + [\![e_2]\!](\sigma)$$
$$\ldots$$

语句的语义定义：

$$[\![x := e]\!](\sigma) \widehat{=} \sigma \oplus \{x \mapsto [\![e]\!](\sigma)\}$$
$$[\![S_1; S_2]\!] \widehat{=} [\![S_1]\!] \circ [\![S_2]\!]$$
$$[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!](\sigma) \widehat{=} \begin{cases} [\![S_1]\!](\sigma) & \text{if } [\![b]\!](\sigma) = \textsf{true} \\ [\![S_2]\!](\sigma) & \text{if } [\![b]\!](\sigma) = \textsf{false} \end{cases}$$
$$\ldots$$

语义定义问题

有兴趣的同学可以读：

《程序设计语言——原理和实践》第13章

《程序设计语言的形式语义》，G. Winskel，中译本：机械工业出版社

信息学院现在有“形式语义学”课程

程序设计语言的语义模型仍然是一个研究问题：

- 存在许多复杂的结构，如何做出简单明晰的严格语义定义
- 不断出现的新语言特征，如何把它们的语义说清楚，说准确
- 如何使严格的程序理论能够成为软件开发中的有力武器

环境	状态	断言	前条件和后条件
循环不变式	部分正确性	终止性	完全正确性
数据不变式	对象不变式（类不变式）		