计算机科学导论-SICP
第一章 构造过程抽象
1.1小节 程序设计的基本元素

王超

Center for Research and Innovation in Software Engineering (RISE), Southwest University

2022 年 9 月 7 日

第一章讲了什么故事

本章讲述如何构造程序来解决(数字)计算问题

- 可以理解为实现函数，每个函数类似一个数学等式
- 大部分情况下函数的输入和输出是数字，有时函数的输入和输出是另一个函数

本章涉及三个递进的“程序”概念

- 第一个“程序”：表达式，每个表达式对应一次计算
- 第二个“程序”：过程，代表了一系列的表达式
- 第三个“程序”：高阶过程，代表了一些列的过程

涉及的LISP语法

- 表达式和过程的定义
- 分支语句和逻辑运算符
- lambda语句和let语句

语义

- 程序是如何运行的
- 第一个语义模型：代换模型

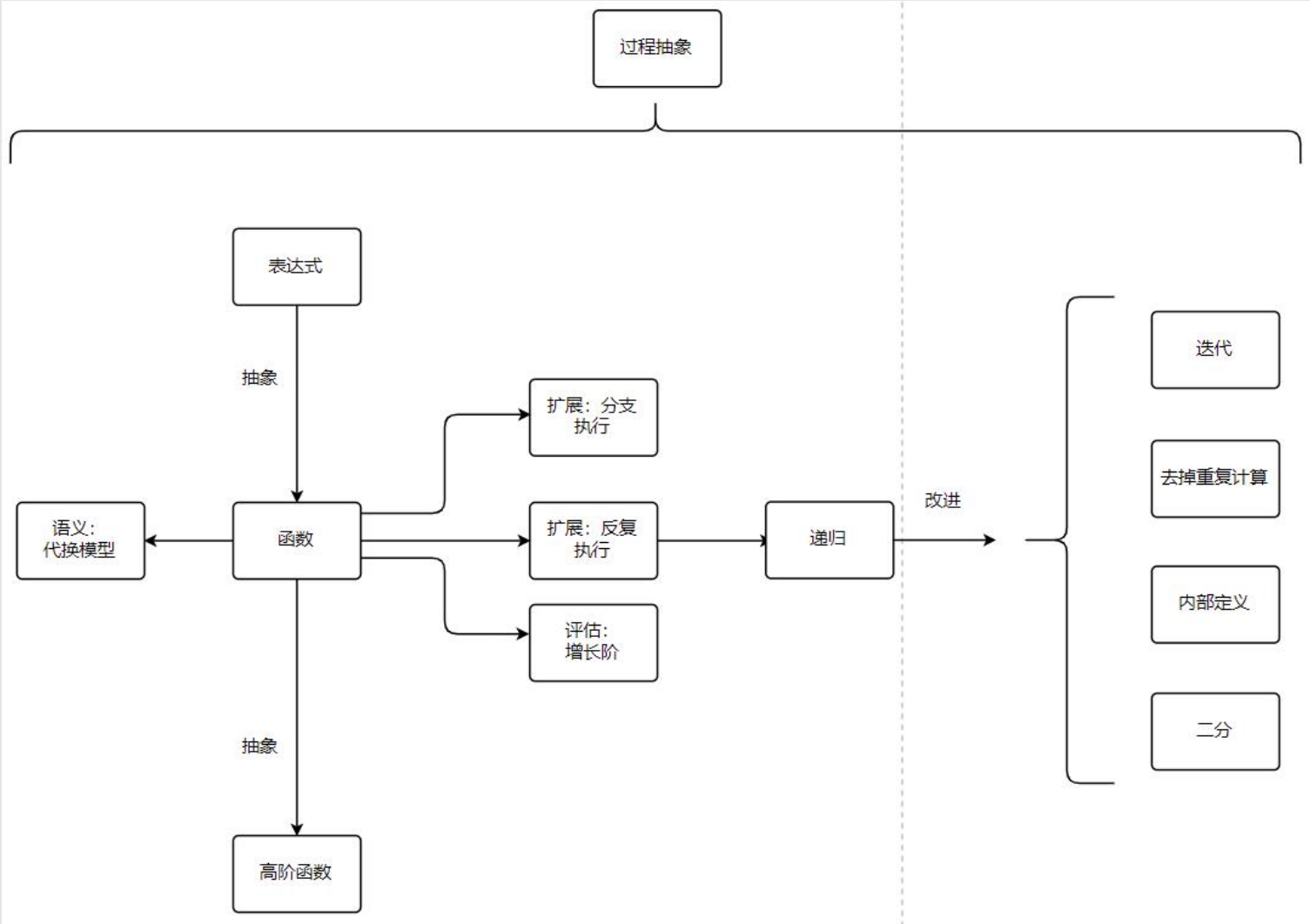
递归

- 当你要定义一个函数，而等式左右两边都有这个函数时，（可能）就发生了名为递归的神奇现象
- 程序设计的基本思想之一
- 也是理论计算机科学的基础之一

时间和空间复杂度(增长阶)

- 评估程序占用的时间和空间资源
- 程序的评估标准之一

第一章知识间的关系


过程抽象
表达式
抽象
扩展：分支
执行
语义：
代换模型
函数
扩展：反复
执行
递归
改进
迭代
去掉重复计算
内部定义
二分
评估：
增长阶
抽象
高阶函数

1.1节讲了什么故事

- 像计算器一样使用Scheme：表达式
- 把本质上完成相似工作的表达式抽象为过程
- 给予过程以不同方式处理不同任务的能力
- 过程是如何工作的：代换模型

1.1节的PPT涉及哪些内容

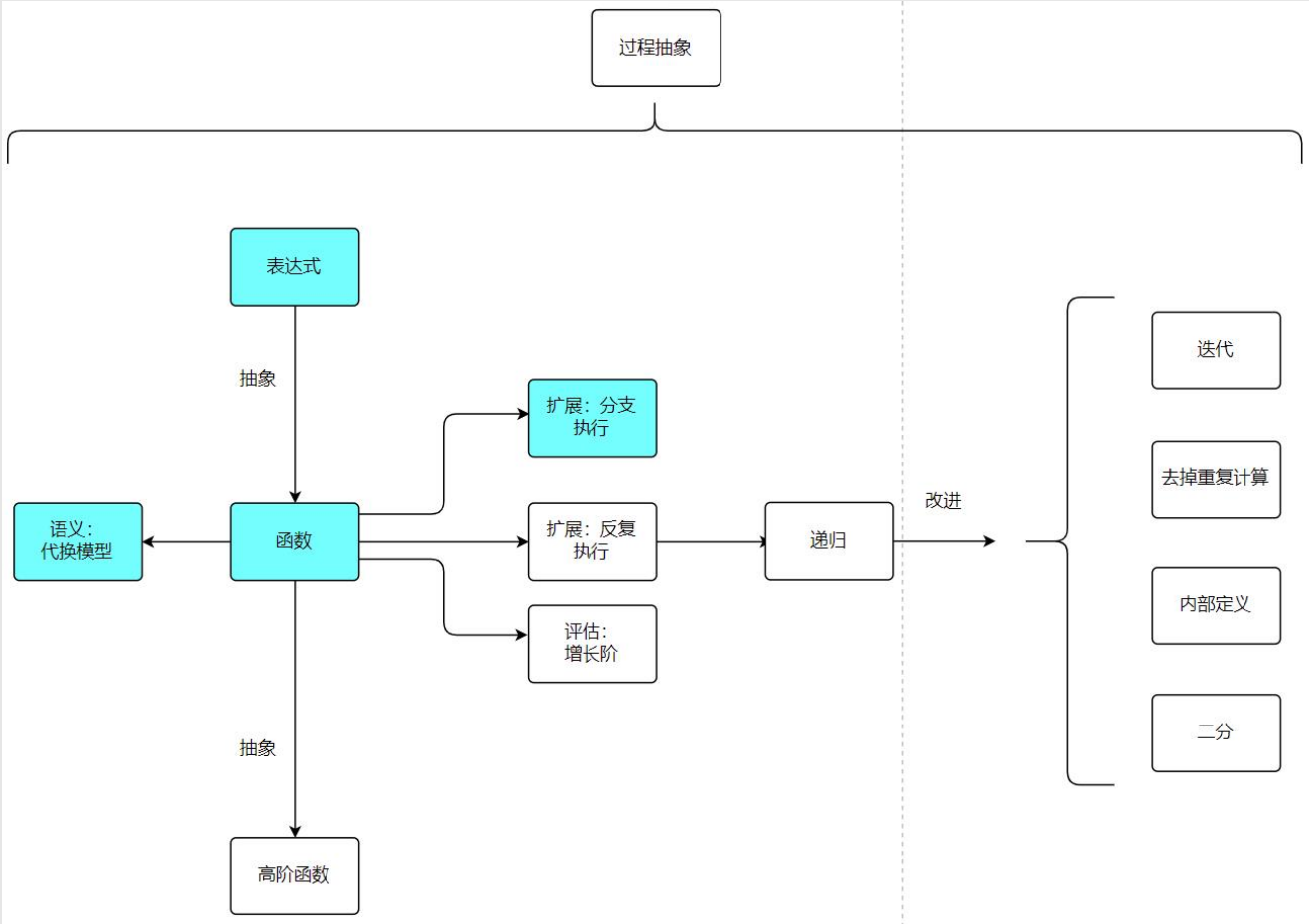
- 表达式与过程的定义
- 条件语句与逻辑运算符
- 代换模型

内容调整

- 1.1.5节中的应用序和正则序暂时不讲，在后续章节涉及时再讲
- 1.1.7节和1.1.8节放到1.2节的PPT中讲

1.1节的内容

蓝色为本次要讲解的内容。



大纲

表达式用来执行数字计算

- 最简单的计算：写出计算步骤，一步一步给出计算结果(数字)
- 数字之间的加减乘除
- 这些运算称为表达式
- 例子: 1，2.5，3+4

前缀表示

- LISP使用前缀表示书写表达式
- 表达式的最外层是一对括号。括号内先写运算符，再写参与运算的数字
- 例子: (+ 3 4)

表达式

定义

- 一个数字是一个表达式：整数或小数均可，正数、负数或零均可
- 一对括号，中间的内容为一个运算符后接若干表达式
- 注意，表达式只有一个数字时无需加括号

表达式的值

- 每个表达式都有值
- 在DrRacket下，输入表达式，点击回车，就可以看到表达式的值
- 此时的Scheme仿佛计算器

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> 1
1
> 2.5
2.5
> -100.8
-100.8
> (+ 3 5)
8
>
```

嵌套的表达式

- 表达式可以嵌套，就像四则运算一样
- 表达式中参与计算的元素，可以是另一个表达式
- Scheme不限制表达式嵌套的层数

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (- (+ (* 4 3) (+ 20 25)) (/ 60 4))
42
>
```

美观打印

- 为了利于理解，建议加入适当的换行
- 在((4*3)+(20+25)) - (60/4)中，3*4和20+25同级，故缩进相同。(4*3)+(20+25)和60/4同级，故缩进相同
- DrRacket会协助“同一级”的元素对齐
- 在输入(* 4 3)之后，使用回车，会自动缩进到当前(+ 20 25)的左括号处

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (- (+ (* 4 3) (+ 20 25)) (/ 60 4))
42
> (- (+ (* 4 3)
        (+ 20 25))
     (/ 60 4))
42
>
```

表达式的求值

- 求值参与最外层运算的各个子表达式
- 把表达式最左边的操作符，应用在求值出的各个值上

表达式无嵌套时

- 数字的值就是自身的数值

表达式有嵌套时

- 需先求值各个“下一级”的子表达式的值
- 当下一级子表达式也有子表达式时，需先求这个子子表达式的值
- . . .

以(- (+ (* 4 3) (+ 20 25)) (/ 60 4))为例

- 首先求值子表达式(+ (* 4 3) (+ 20 25))和(/ 60 4)，再用这两个值做-运算
- 求值(+ (* 4 3) (+ 20 25))：需要求值(* 4 3)和(+ 20 25)，再用这两个值做+运算
  - 求值(* 4 3)：3*4=12
  - 求值(+ 20 25)：20+25=45
  - 所以，(+ (* 4 3) (+ 20 25))的值是12+45=57
- 求值(/60 4)：60/4=15
- 所以，(- (+ (* 4 3) (+ 20 25)) (/ 60 4))的值是57-15=42

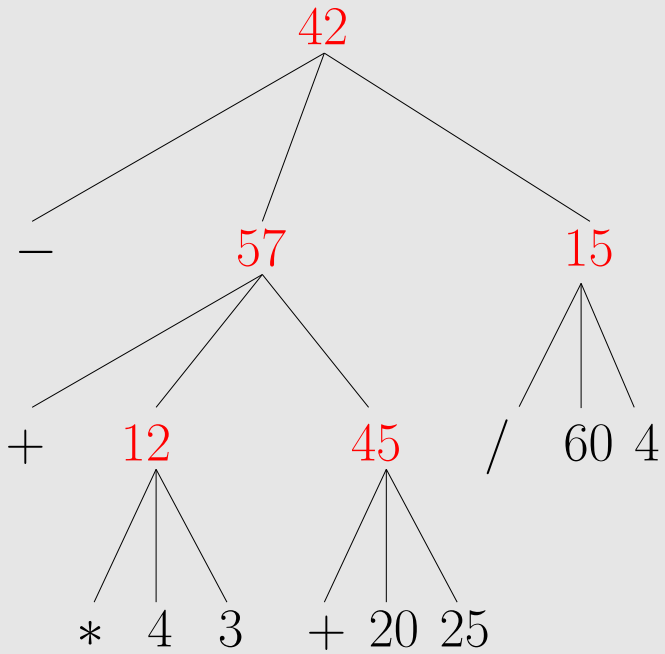
- 这就是小学的四则运算流程啊。为什么不说“先计算最内层括号的数字，再计算次内层的数字，以此类推”
- 原因一：这里我们的叙述方式，其实使用了后面讲到的递归的思想
- 原因二：我们目前叙述的这种求值方法，更容易用计算机实现

递归

- 解决一个问题时，不是从头告诉计算机每一步该如何进行，而是假定当前问题中一些规模较小的部分已经得到解决，在此前提下设计算法
- 求值表达式时，假定一些规模较小的部分已经解决(各个子表达式的值已经求出)，在此前提下如何求值：把表达式最左边的操作符，应用在求值出的各个值上
- 使用递归计算问题对人较为繁琐，因为要在草稿纸上记忆很多信息
- 然而，计算机最适合的就是繁琐的计算，计算机不适合的是理解“真正的含义”
- 1.2节我们正式介绍递归

递归求值例子：(- (+ (* 4 3) (+ 20 25)) (/ 60 4))

- 第一层是待求值的表达式的值
- 第二层是第一层表达式的操作符和值
- 第三层是第二层表达式的操作符和值



表达式的命名

- 格式：(define 表达式名字 表达式内容)
- 将一个名字绑定到一个表达式
- 之后，这个名字可以被当作一个元素使用，会被自动替换为表达式的值

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (define a (- (+ (* 4 3) (+ 20 25)) (/ 60 4)))
> (+ a 100)
142
> (* a a)
1764
>
```

环境

为了求值表达式名字

- Scheme解释器存储名字-值的对
- 通过一种称为环境的机制实现
- 求值表达式名字时，由环境提供对应的值

伏笔：第三章详细叙述环境的定义

大纲

- 表达式
- 过程
- 代换模型
- 条件表达式和谓词
- 结束

过程

表达式缺失了什么

- 计算3的平方：(* 3 3)
- 计算4的平方：(* 4 4)
- 表达式可以进行具体数字的计算，但无法表示“计算平方”这个概念

过程

- 对有着相同“模式”的表达式的抽象
- 一个过程代表一类表达式
- 过程包含参数和过程体
- 过程体是这类表达式的描述，参数说明过程的执行效果等同于这类表达式中的具体哪个
- 过程体是一个带变量(参数)的表达式，参数给定时过程体可以求值
- 过程有输入(参数)，也有输出(求值结果)

可以直观把一个过程理解为一本操作手册

- 手册里有些地方是变量
- 例子：x炒鸡蛋(x是变量)
- 步骤一：锅加热，倒油，放入鸡蛋，加热并搅拌，之后取出
- 步骤二：锅加热，倒油，先放入葱，用铲子拌一下
- 步骤三：把x和鸡蛋放入锅里，加热并搅拌，放盐，之后取出

过程使用：直观理解为把那本手册里的变量替换为给定内容

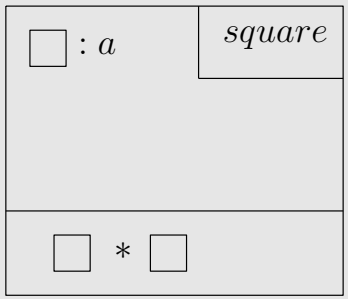
- 例子：以x=西红柿的方式，使用x炒鸡蛋这本手册
- 步骤一：锅加热，倒油，放入鸡蛋，加热并搅拌，之后取出
- 步骤二：锅加热，倒油，先放入葱，用铲子拌一下
- 步骤三：把西红柿和鸡蛋放入锅里，加热并搅拌，放盐，之后取出

x=韭菜，x=黄瓜，...

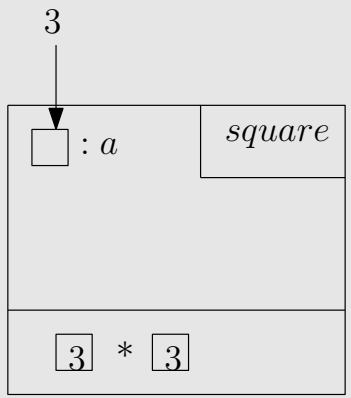
图片化表示

以求平方为例

- 过程定义分为两部分
- 上半半是参数：格子命名为a
- 下半半是过程体：包含格子的表达式，即格子乘以格子



- 过程使用：求3的平方
- 把原有的格子替换为3，过程体的表达式变为(* 3 3)，它的值就是过程使用的值



过程定义

- 格式：(define (过程名 参数列表) 过程体)
- 参数列表：若干变量
- 过程体：一个表达式，允许出现数字、变量和过程使用
- 过程体代表过程所定义的模式

过程使用

- 格式：(过程名 具体值列表)
- 实例化过程所定义的模式，将每个变量替换为对应的值并求值过程体

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (define (square a) (* a a))
> (square 3)
9
>
```

形参和实参

- 形式参数：过程定义中参数列表中的元素
- 例子：定义square时的a
- 实际参数：过程使用时填入变量的具体值
- 例子：使用square时的3

过程作为模块

过程体可以包含过程使用

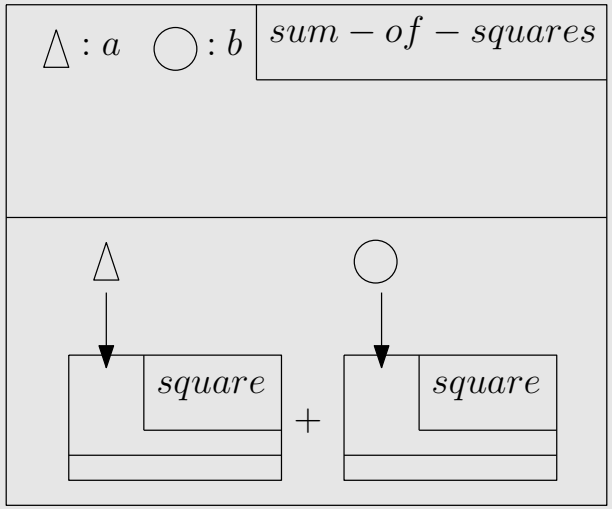
- 先实现简单的计算过程，再使用这些过程来构造更复杂的过程
- 这就是所谓组合的思想

直观理解

- 定义一个过程(f u v)：做两道菜，u炒鸡蛋和v炒鸡蛋
- 步骤一：通过使用x炒鸡蛋的方法，制作u炒鸡蛋
- 步骤二：通过使用x炒鸡蛋的方法，制作v炒鸡蛋

图片例子

- sum-of-squares：计算两个数的平方和




```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp, 带调试; memory limit: 128 MB.
> (define (square x) (* x x))
> (define (sum-of-squares x y)
    (+ (square x) (square y)))
> (sum-of-squares 10 9)
181
>
```

抽象和组合

- 抽象：把满足特定形式的表达式定义为过程
- 组合：使用已有过程定义更加复杂的过程

伏笔

- 过程定义的格式是(define (name x y ...) (...))，而命名表达式的定义是(define name (...))
- 过程定义多了参数列表
- 1.3节我们会讲到，实质上过程也是以(define name (...))的形式定义的
- 语法中目前的形式只是为了使用方便，这是一种语法糖

大纲

- 表达式
- 过程
- 代换模型
- 条件表达式和谓词
- 结束

- 很明显，除非明确了解自己的程序会有什么样的执行后果，否则不可能认为自己掌握了计算机编程
- 没有嵌套的表达式运行结果很容易理解
- 对(包含嵌套)的表达式的求值也不难理解，因为每一部分都是固定的值
- 因为过程体中可能包含过程调用，对过程的求值比对表达式的求值更为复杂
- 困难一：过程体中有变量
- 困难二：求值(f 2)时，f的过程体可能会调用多个函数，甚至它自己(1.2节讨论)

我们马上给出代换模型来解释过程使用的结果

代换模型

如何求过程使用的值

- 求出各个参数的值
- 将过程体定义中的形参替换为对应的值
- 求过程体表达式的值

过程体包含嵌套的过程使用时

- 需要先求值内层的过程使用，再求值当前过程体
- 类似于嵌套表达式，Scheme不限制嵌套的层数

这个求值的过程成为代换模型

例子：求值(sum-of-squares (square (square 3)) 5)的过程

- 把(sum-of-squares (square (square 3)) 5)替换为sum-of-squares的函数体，变量x替换为(square (square 3))，变量y替换为5
- 由于x被替换为(square (square 3))，这不是一个数字，因此要先求出这个过程使用的值
- 为此，把(square (square 3))替换为外层square的过程体，x替换为(square 3)
- 由于x被替换为(square 3)，这不是一个数字，因此要先求出这个过程使用的值
- 求值内层的(square 3)，将x替换为3，求值过程体(* 3 3)，值为9
- 进而，求值(square (square 3))，把x替换为9，求值过程体(* 9 9)，值为81
- 进而，求值(sum-of-squares (square (square 3)) 5)，把x和y分别替换为81和5，值为81*81+5*5=6586

求值(sum-of-squares (square (square 3)) 5)的具体过程

- (sum-of-squares (square (square 3)) 5)
- (+ (square (square (square 3))) (square 5))
- (+ (square (square (* 3 3))) (square 5))
- (+ (square (square 9)) (square 5))
- (+ (square (* 9 9)) (square 5))
- (+ (square 81) (square 5))
- (+ (* 81 81) (* 5 5))
- (+ 6561 25)
- 6586

语法与语义

语法

- 每个语言都有自己的格式
- 语法：什么样的代码符合语言的格式要求
- 语法只关注格式对不对，不关注代码执行结果会不会出错

语义

- 每个编程语言都需要告诉用户，需要假定语言运行在什么样的(抽象)机器上
- 语义给出编程语言的每条语句如何访问这个抽象模型，或者说，每条语句是如何执行的
- 程序员在语言提供的抽象模型上进行设计，只需要关注程序对抽象模型的影响
- 编程语言(编译器)保证编程语言的每条语句行为确实如其语义模型

计算机领域有多种多样的编程语言

- 硬件设计语言：FPGA
- 汇编语言：8086汇编，80386汇编
- 通用编程语言：C语言和C++语言(指针)，以及JAVA语言和python语言(引用)
- 函数式语言：Haskell语言,Ocaml语言
- 应用于特定领域的语言：HTML语言，SQL语言

这些语言位于不同层次，解决不同问题，有着不同的语义模型

- 代换模型是我们本课程中遇到的第一个Scheme的语义模型
- 代换模型只能处理我们目前所学的Scheme语句，它不能处理一些后面章节的Sceheme语句，因此不是整个Scheme的语义模型
- 更复杂的模型在处理当前语句的时候也会得到和代换模型一样的结果，所以我们目前可以安全地使用代换模型解释程序的行为
- 伏笔：随着更多语法元素的引入，语义模型也会扩增
- 代换模型(第一章) $\Rightarrow$ 环境模型(第三章) $\Rightarrow$ 元循环求值器(第四章) $\Rightarrow$ 完整的编译器(第五章)

大纲

- 表达式
- 过程
- 代换模型
- 条件表达式和谓词
- 结束

引入

想象一下你需要制作一辆赛车

- 首先，要跑得快
- 其次，需要学会转弯

已有的过程语句能力太弱

- 过程的执行只能“走同一个流程”，不会“拐弯”
- 例子：如何构造过程计算a和b的最大值
- 例子：如何构造过程，如果x是奇数则执行结果为true，如果是偶数则执行结果为false

本小节引入分支

- 函数的计算过程中可以根据不同的条件，选择不同的流程执行
- cond语句和if语句

本小节引入逻辑运算符

- 用来写出这些条件
- and运算符、or运算符和not运算符

cond语句

- 根据不同的条件成立与否，选择程序的执行分支
- 格式：(cond ($pred_1$ $expr_1$) ... ($pred_k$ $expr_k$) (else $expr$))
- $pred_i$是一个求值结果为true或false的表达式，称为谓词
- cond表达式的求值方法如下
- 首先求值表达式$pred_1$，如果值为true，则cond表达式的值就是$expr_1$的值。求值$expr_1$。此时无需求值其他条件或分支
- 否则，求值表达式$pred_2$，如果值为true，则cond表达式的值就是$expr_2$的值。求值$expr_2$。此时无需求值其他条件或分支
- ...
- 如果表达式$pred_1$到$pred_k$的求值结果都为false，则cond表达式的值就是$expr$的值。求值$expr$

以求绝对值的过程为例

- <是Scheme默认提供的过程，判断数字的小于关系
- >，>=和<=也是Scheme默认提供的过程
- (- x)代表x*(-1)
- 注意cond语句的语法格式，尤其是括号

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (define (abs x)
    (cond ((< x 0) (- x))
          (else x)))
> (abs 10)
10
> (abs -6)
6
> (abs 0)
0
>
```

注意

- cond表达式往往只求值部分条件和分支，而无需求值所有条件和分支
- 这一点和普通语句的求值是不同的
- 例如，如果有4个条件，条件1为假，条件2为真，则cond表达式只求值条件1、条件2和分支2，而不求值条件3、条件4、分支1、分支3和分支4

注意

- else部分不是必需的，但建议写下else部分，作为兜底条件
- 如果cond语句没有else部分，且如果每个条件的求值结果都为false，则此时cond表达式的值为未定义的
- 未定义不是一件可有可无的事情，而是很严重的问题
- 编译器可以任意地实现未定义行为
- 不要让自己的代码包含未定义行为

例子：错误的绝对值过程实现

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (define (abs2 x)
    (cond ((< x 0) (- x))
          ((> x 0) x)))
> (abs2 10)
10
> (abs2 -6)
6
> (abs2 0)
>
```

if语句

- 一种简化的cond语句，用于二选一
- 格式：(if *pred* *consequence* *alternative*)
- if表达式的求值方法如下
- 首先求值表达式*pred*，如果值为true，则if表达式的值就是*consequence*的值。求值*consequence*。此时无需求值*alternative*
- 否则，if表达式的值就是*alternative*的值。求值*alternative*。此时无需求值*consequence*
- 注意，if表达式无需求值所有分支

以求绝对值的过程为例

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs 10)
10
> (abs -6)
6
> (abs 0)
0
>
```

基本谓词

- true和false：Scheme提供，表示逻辑真和逻辑假
- Scheme提供了一些过程来计算真假值：<，<=，=，>，>=
- 在后续章节中，这样的谓词还会扩增

基于这些简单的谓词，可以使用and、or和not来构造复杂的谓词

逻辑运算符

and运算符

- 含义：判断是否所有条件都为真
- 格式：(and $expr_1 \ldots expr_k$)
- 从左到右依次求值表达式$pred_1 \ldots pred_k$。如果都为true，则and表达式的值为true；否则，and表达式的值为false
- 注意，在求值表达式$pred_1 \ldots pred_k$的过程中，如果某个表达式$expr_i$的求值结果为false，则无需接着求值之后的表达式$pred_{i+1} \ldots pred_k$

or运算符

- 含义：判断是否至少一个条件都为真
- 格式：(or $expr_1 \ldots expr_k$)
- 从左到右依次求值表达式$pred_1 \ldots pred_k$。如果至少一个表达式$expr_i$的求值结果为true，则or表达式的值为true；否则，or表达式的值为false
- 类似于and表达式的求值，在求值or表达式时，如果某个$expr_i$的求值结果为true，则无需求值之后的表达式$pred_{i+1} \ldots pred_k$

not运算符

- 含义：判断当前条件是否为假
- 格式：(not *pred*)
- 求值表达式*pred*，如果其值为false，则not表达式的值为true；否则，not表达式的值为false

逻辑运算符可以嵌套

- 判断n是闰年：或者n不是100的倍数且n是4的倍数，或者n是400的倍数
- 判断n非100的倍数：(not (= (remainder n 100) 0))
- 判断n是4的倍数：(= (remainder n 4) 0)
- 判断n是400的倍数：(= (remainder n 400) 0)
- 合在一起：(or (and (not (= (remainder n 100) 0)) (= (remainder n 4) 0)) (= (remainder n 400) 0))
- 基于这个谓词，定义过程isLeapYear?判断给定年份是否是闰年
- remainder是Scheme默认提供的过程，用来计算除法的余数

```
欢迎使用 DrRacket, 版本 8.1 [cs].
语言: sicp，带调试; memory limit: 128 MB.
> (define (isLeapYear? n)
    (or (and (not (= (remainder n 100) 0)) (= (remainder n 4) 0) ) (= (remainder n 400) 0)))
> (isLeapYear? 1985)
#f
> (isLeapYear? 1988)
#t
> (isLeapYear? 1900)
#f
> (isLeapYear? 2000)
#t
>
```

课堂思考题：写出如下过程

- (check1 x)：判断x大于等于0且小于等于100，是的话返回true，否则返回false
- (check2 x)：判断x是一个大于100且除以7的余数为6的数字，是的话返回true，否则返回false

大纲

课堂总结

- “计算”的形式：表达式，进一步抽象为过程
- 求值表达式和过程
- 条件

可以发现，Scheme代码和数学公式有着很大的相似性

Scheme语法

- 表达式
- 使用define定义过程
- 条件语句：cond和if
- 逻辑运算符：and、or和not
- 谓词：>、>=、=、<、<=
- 内置的过程：remainder

本节内容与其他知识的联系

在后续章节中会进一步进展的

- 环境(第三章)，代换模型(第三章)
- 过程定义(1.3节)

1.1节导引的其他课程

- 无

预告

- 计算机提供给程序员的数字计算操作只有加减乘除
- 毫无疑问，计算机可以实现诸如计算π、$\sqrt{2}$、$\sqrt[3]{5}$、$sin(x)$、$\int_a^b f(x)$等任务
- 在1.2节我们会叙述，如何使用1.1节中的内容，计算$\sqrt{2}$、$\sqrt[3]{5}$和$sin(x)$
- 在1.3节我们会叙述，如何使用1.1节中的内容以及高阶过程，计算π和$\int_a^b f(x)$

1.1节练习题

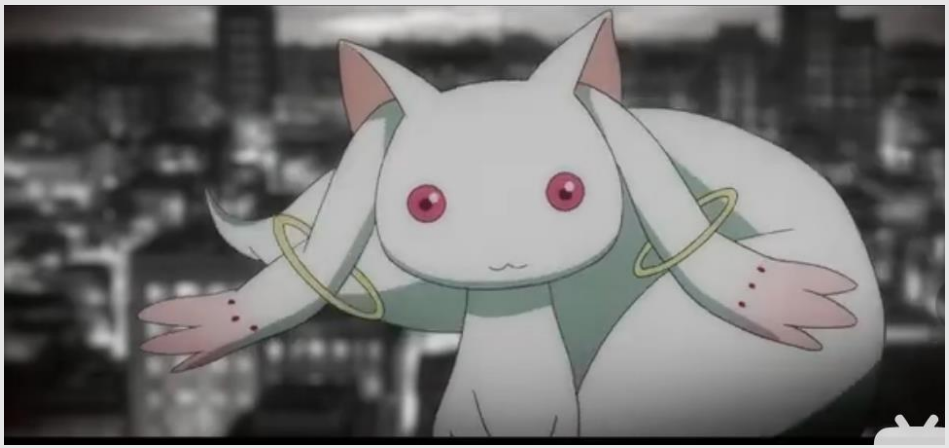
练习级

- 教材练习1.3

挑战级

- 无

- 要动手练习才能学会编程
- 1.1的代码是最简单的
- 有问题及时提问，以及和老师交流



下课