

# 问题求解课讲义

AUGPath

2023 年 2 月 26 日



# 目录

<b>第一章 Python 基础回顾</b>	<b>5</b>
1.1 问题的提出: 通过程序, 可以求解的问题 . . . . .	5
1.2 程序设计的语法 (grammar) 与语义 (schematics) . . . . .	5
1.3 Python 代码执行可视化 (visualization): 一个网站 . . . . .	6
1.4 可以执行任何程序的最小指令集 . . . . .	6
1.4.1 Python 中的整个世界 . . . . .	8
1.4.2 函数: 整合相似过程 . . . . .	12
1.5 一些小例子 . . . . .	13
<b>第二章 简单数理逻辑</b>	<b>19</b>
2.1 问题的提出: 为什么要研究简单的数理逻辑 . . . . .	19
2.2 开始行动: 符号化逻辑 . . . . .	23
2.2.1 命题, 真值表和指派 . . . . .	23
2.2.2 命题逻辑的化简 . . . . .	29
2.2.3 化简的目标: 范式 . . . . .	30
2.3 命题推演的真象: 命题的推演 . . . . .	33
2.4 表达更丰富的内容: 谓词 部分与整体的关系 . . . . .	36
2.5 尝试解释谓词公式 . . . . .	39
2.6 常见的证明方法 . . . . .	41
2.7 重新审视算法的正确性: 从基本结构开始 . . . . .	44
2.8 继续理解子过程和递归 . . . . .	48



# 第一章 Python 基础回顾

## 1.1 问题的提出：通过程序，可以求解的问题

上学期我们学习了 Python 程序设计。可能我们会对学程序设计合理性产生质疑。看上去，计算机是一个笨重的工具，在时间或空间限制下可以有效地执行许多重复的任务。到现在为止，计算机还不能分析问题并提出解决方案。另一方面，人类在分析和解决问题方面非常出色，但很容易对重复的任务感到厌倦。

人类通过利用他们的分析和解决问题的技能，可以为可计算的问题提出算法（有限的指令，与有限的输入一起工作，产生输出）。然后，计算机可以遵循这些指令并产生一个答案。

你可以将你生活中的各种多余的任务自动化。一点点的 Python 可以给你的生活带来神奇的效果。甚至提高你的生产力。

## 1.2 程序设计的语法 (grammar) 与语义 (schematics)

我们在学习英语的时候很重要的一部分是语法：也就是什么样的语言是可以被接受 (acceptable) 的。比如下面这个英文句子是没有语法错误的：

`Fuhai Zhu said that this test is only a small test, so don't panic.`

但是这句话不同人有不同的理解方式，这就是这句话的语义。

- 我们可以推断朱富海老师在安抚准备参加小测试学生们的情绪
- 但是南京大学数学系的同学知道这是一个有名的梗：上学期教高等代数的朱富海老师把线 upper 中考叫做“小测验”，并在同学询问考试范围的时候微笑的答道：“从小学学的都考。”

简而言之（不严谨），现在我们有一个由一堆字符串和推导规则组成的形式系统 (formal system)，语法决定了这个形式系统能生存什么样的字符串，而至于这些字符串有什么样的含义则是语义的范畴。语法类似材料，语义类似与材料组成的各种建筑物，我们可以通过语法研究语义层面的推导，同时也可以从语义层面捕获语法中内涵的结构，其实语法和语义是相互区别又紧密联系，即从范畴论的角度看语法和语义是伴随的（其实不同的人做数学证明可以有不同的风格：偏语法和偏语义，不过大部分数学家更喜欢语义风格的证明，可能因为更直观，更容易被人脑接受 ~）

作者: 知乎用户链接: <https://www.zhihu.com/question/31347357/answer/892133941>

来源: 知乎著作权归作者所有. 商业转载请联系作者获得授权, 非商业转载请注明出处.

## 1.3 Python 代码执行可视化 (visualization): 一个网站

### Tools 工具

这是Python 代码执行可视化的机器, 我们可以用一个小程序来测试之.

```
1 for i in range(10):  
2     print(i**2)
```

点击 Visualize Execution 就可以了, 你可以点击 Next 来继续模拟执行下一步.

在这里, 你可能会看到很多新奇的名词: 什么是 Global Frame, Object? 暂时先不用管. 不过你确实可以看到点击 Next 的时候 Print output 一栏一步一步的模拟了你的代码.

可以知道, 代码按照行数执行, 一次执行一行, 每一次执行计算机内部结构的状态 (右侧的面板). 下面我们化繁为简, 来看一看一个系统 (数学意义上) 能够完成任何人类完成的操作需要的最小可能的操作是什么.

## 1.4 可以执行任何程序的最小指令集

像数学的公理体系那样, 我们自然希望得到一个最小的指令集合, 并且我们可以用来写出任何的程序. 我们不妨从日常生活中找一点灵感吧.

### Example 例子:

(等红绿灯) 观察红绿灯, 如果 (if) 是绿灯, 那就通过这个路口; 否则 (else) 继续等待. (遵纪守法的好公民)

(做作业) 明确今天的作业范围, 从**第一题**开始写, 写完题目**或者 (or)** 一题目没有思路之后做**下一道题**, **直到 (until)** 做完所有的问题.

(排序成绩单) 获得班上同学的所有**成绩单**, 拿一张新的白纸打好**表格**, **每一次 (for each time)** 从成绩单中选取最大的分数, 把那一行**抄写到**新的白纸上. 之后把原来那张纸上的内容划去. **一直重复下去**, **直到**原来的成绩单上没有任何可以被划去的内容.

我们需要找一些东西来具象化我们脑子中的“红绿灯的状态 (state)”, “现在在做作业的题目编号”, 这些内容, 因此我们就希望把这些抽象出来. 因此我们有了变量 (variable) 的概念, 也就是值存在的空间.

把上面的三个内容转化成伪代码 (不唯一) 就是:

```
----- GO THROUGH CONJUNCTION -----  
if traffic light's color is green:  
    go pass by  
else
```

```

wait

----- DO HOMEWORK -----
range = [a..b]
working on problem = a
while working on problem <= b:
    finished this problem or can't work out
    working on problem = working on problem+1

----- SORT EXAM SCORE -----
list = get the source table
result = empty(for now)
while list is not empty
    k= get the max element of list
    write k to the next line of result
show result

```

其实要是能够构造出任何程序的原材料并不复杂. 无非就是变量的赋值 (assign), 判断 (judgement), 跳转 (jump), 终止 (terminate). 也就是, 如果你能声称有一套系统可以自动化的解决这四个内容, 那么这个系统就具有机械化地做任何人类做的事情. 换句话说, 你可以用这个工具创造整个世界.

其实最早这个想法是在计算机诞生之前人们孜孜以求的问题. Alan Turing 在 1936 年就提出了这样的设想. 他就是由只一条 (无限长) 的纸带和一根笔 (可以改纸带的内容, 并且查看纸带的内容并据此做判断), 并且有一个程序 (墙上的表格), 指示下一步要往哪转移. 只要能够移动读写头, 写纸带的某一个格子, 读纸带的某一个格子, 跳转, 以及终止, 这个机器就和我们人类的计算能力等价.

这是最早的图灵机的原型.

#### Example 例子:

运行上面图片的程序, 左右按照我们的左右进行 (规定 ABC 右移一格是 ABC).

(1) 现在机器的状态是 A(头部的字母), 看到的是 1(放大镜的字母)

(2) 于是把当前的格子改为 1, 纸带向右移动一格, 然后停机.

假设当前纸带的放大镜看到的是 0, 再运行一次:

状态:A 纸带状态: 0 1 1 1 0 1 1 0 0

(1) 现在机器的状态是 A(头部的字母), 看到的是 0(放大镜的字母), 执行第一行第一列的指令 1(改为 1) → (向右移动一格)B(状态改为 B) .

状态:B 纸带状态: 0 1 1 1 1 1 1 0 0

(2) 现在机器的状态是 B(头部的字母), 看到的是 1(放大镜的字母), 执行第二行第二列的指令 1(改为 1) → (向右移动一格)B(状态改为 B) .

状态:B 纸带状态: 0 1 1 1 1 1 1 0 0

(3) 现在机器的状态是 B(头部的字母), 看到的是 1(放大镜的字母), 执行第二行第二列的指令 1(改为 1)  $\rightarrow$  (向右移动一格)B(状态改为 B) .

状态:B 纸带状态: 0 1 1 1 1 1 1 0 0

(4) 现在机器的状态是 B(头部的字母), 看到的是 0(放大镜的字母), 执行第二行第一列的指令 0(改为 0)  $\rightarrow$  (向右移动一格)C(状态改为 C) .

状态:C 纸带状态: 0 1 1 1 1 1 1 0 0

(5) 现在机器的状态是 C(头部的字母), 看到的是 0(放大镜的字母), 执行第三行第一列的指令 1(改为 1)  $\leftarrow$  (向左移动一格)C(状态改为 C) .

状态:C 纸带状态: 0 1 1 1 1 1 1 0 1

(6) 现在机器的状态是 C(头部的字母), 看到的是 0(放大镜的字母), 执行第三行第一列的指令 1(改为 1)  $\leftarrow$  (向左移动一格)C(状态改为 C) .

状态:C 纸带状态: 0 1 1 1 1 1 1 1 1

(7) 现在机器的状态是 C(头部的字母), 看到的是 1(放大镜的字母), 执行第三行第二列的指令 1(改为 1)  $\leftarrow$  (向左移动一格)A(状态改为 A) .

状态:A 纸带状态: 0 1 1 1 1 1 1 1 1

(8) 现在机器的状态是 A(头部的字母), 看到的是 1(放大镜的字母), 执行第一行第二列的指令 1(改为 1)  $\rightarrow$  (向右移动一格)†(停机)

状态:A 纸带状态: 0 1 1 1 1 1 1 1 1

下面我们来看一看为什么说可以用 Python 创造整个世界. 在这之前, 我们先来了解一下如何获取帮助和得到通俗易懂的教程.

请认真阅读并实践 (无论是在脑子还是在交互器 (interactive shell) 里面) 文档 <https://docs.python.org/zh-cn/3/tutorial/introduction.html> 的内容. 可以让你了解更多易于理解的东西. 如果文章中有描述 C 和 Pascal 的句子, 忽略它就可以.

#### Idea 启示

摒弃“我一定要一次把所有的东西都弄懂”的理念, 因为这些知识十分的巨大. 所以先了解一部分作为正确的立足点, 然后慢慢扩大就行了. 这需要很长时间的积累, 千万不要急. (着急也没用, 只会加大精神内耗.)

### 1.4.1 Python 中的整个世界

下面的内容其实不用单独记忆, 只要明确有哪些语句, 这些语句造成的效果是什么就行了. 如果看到有任何的问题, 可以去搜一搜词典.

下面会有两个术语 (term), 分别是表达式 (expression) 和过程 (procedure), 表达式可以暂且认为是形如  $x+12$ ,  $[2]*3$  这样的可以进行计算的内容, 过程就是一系列执行的过程, 不一定要能得到值. 我们用一个例子感受一下.



```

def InsertionSort(A):
    for j in range(1, len(A)):          #Proc
        key = A[j]      #A[j],key are expr #Proc  #|
        i = j - 1      #j-1,i are an expr #v      #|
        while (i >=0) and (A[i] > key):          #|
            #    <-expr->    <----expr----->    #|
            #    <-----expr----->            #|
            A[i+1] = A[i] #A[i]is expr    #Proc    #|
            i = i - 1      #i-1 is expr    #v      #|
        A[i+1] =    key                                #v
    <-expr-> <-expr->

```

其实上面的 Proc 表示过程, 然后右边的是一个字符画, 表示 ↓, <-expr-> 其实表示的意思是 example 这一段是表达式.

expr

重要的是, 把示例代码放到上面提到的可视化网站里面看一看就会很清楚, 很多概念都是不用记忆的.

## 变量的定义与赋值

**定义 1.4.1.** (变量的赋值) 变量名 = 变量的值

语义:

下面我们给出注解:

- 在不加修饰的情况下, 变量的名称只在当前的缩进块内有效
- 命名是用来指代对象的. 这就是为什么有时候可视化工具里面 Frames 后面有一个箭头指着 Objects.
- 如果用一个变量 = 另一个变量, 大多数情况是现计算出来右手边表达式的值之后给左手的变量. 有时候一些文章里面写作  $lhs \leftarrow rhs$ .

```
b=114514
```

```
a=b+1 # 执行完本句之后 a=114515
```

```
b = b+1 # 执行完本句之后 b=114515, a=114515 不变
```

```
a = b+1 # 执行完本句之后 a=114515, b=114516
```

- 在 Python 中, 变量的值的类型可以是任意的. 因为 Python 声明变量的时候没有说明类型.

```
a='Fuhai Zhu teached Advanced Algebra' # a 现在是字符串
```

```
a=1 # a 现在是整数
```

```
a=None # None 是一个关键字, 表示什么都没有.
```

- 如果没有定义就使用了一个变量, 通常就会有如下的报错:

```
print(a+1)
~

Traceback (most recent call last):  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
(命名错误: 名称'n' 没有定义)
```

什么是 Traceback? stdin 又是什么? 后面可能会注意到.

可能经常会常用的变量类型: 数字、字符串、列表. 这时候可以参看官方文档 <https://docs.python.org/zh-cn/3/tutorial/introduction.html> 来继续.

### 控制语句: 判断与循环

定义 1.4.2. (条件判断) 可以使用 if 语句进行条件判断, 一般的, 有如下的形式:

```
if 表达式 1:
    过程 1
elif 表达式 2: # 可以有零个或者多个 elif, 但是 else 后面不能有 elif
    过程 2
else:
    过程 r
```

语义: 它通过逐个计算表达式, 直到发现一个表达式为真, 并且执行使表达式为真的这个过程 (完成后不执行或计算 if 语句的其他部分的判断表达式). 如果所有表达式都为 false, 如果存在 else 下方语句块的过程.

下面我们同样给出注记和例子.

- 什么是真? 什么是假? 我们会在后面探讨. 首先可以认为非 0 数字和 True 是真, 0 和 False 和 None 是假.

```
if "AK":
    print("AK") # 会输出 AK, 这是怎么判断的?(后续会回答)
```

- 可以用逻辑运算符 and(且) or(或) not(非) 进行逻辑表达, 比如

```
zgw = 0
kertz = 1
ak = 1
cmo = 1
if kertz and ak and cmo :
    print(`Zixuan Yuan got full mark in CMO`)
elif zgw and ak and cmo:
    print(`zgw got full mark in CMO`)
else:
```

```

    print(`zgw is such a noob'`)
# 会输出 Kertz got full mark in CMO, 由于已经找到了一个表达式的值为真的
# 表达式, 所以执行完 print(`Zixuan Yuan got full mark in CMO'`) 之后
就
# 会跳转到这个语句块的尾部了. 不会执行 print(`zgw is such a noob'`).
# (为自己菜爆的数学基础做了一个掩盖 (大雾))

```

- 如果结构不完整, 或者在 else 之后还有 elif, 那么就触发形如这样的错误:

例子 1.py-----

```

if True:
    print("Err")

```

----

File "main.py", line 3

```

    print("Err")

```

```

~ IndentationError: expected an indented block

```

(缩进 (indent) 错误: 我预期有一个带着缩进的语句块, 但是没有)

例子 2.py-----

```

if False:

```

```

    print(1)

```

```

else:

```

```

    print(2)

```

```

elif True:

```

```

    print(3)

```

---

File "main.py", line 5

```

    elif True:

```

```

~ SyntaxError: invalid syntax

```

(语法错误: 无效的语法)

**定义 1.4.3.** (while 循环) 可以使用 if 语句进行条件判断, 一般的, 有如下的形式:

**while** 表达式:

    过程 1

else: # 可以有, 也可以没有

    过程 2

语义: 这样反复测试表达式, 如果为真, 则执行**过程 1**; 如果表达式为假 (这可能是第一次测试), 则执行 else 子句的**过程 2**(如果存在的话), 然后循环终止. 在**过程 1**中执行的 **break** 语句会终止循环, 且不执行 else 子句的**过程 2**. 在**过程 1**中执行的 continue 语句跳过**过程 1**的 continue 语句之后的其余部分, 然后立刻回到测试表达式语句.

有了循环 (loop), 我们就可以解读这个东西:

```

def InsertionSort(A):

```

```

j=1
while(j<len(A)):
    key = A[j]
    i = j - 1
    while (i >=0) and (A[i] > key):
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
    j=j+1
return A
InsertionSort([1,1,4,5,1,4])

```

这个做的事情就和排序成绩类似.

### 程序的终止

**定义 1.4.4.** Python 程序的终止可能包含有如下的情况:

- (1) 执行到了最后一条语句, 且没有下一条语句可以执行;
- (2) 程序有没有被处理的异常;
- (3) 通过语句 `exit(0)` 退出.

因此, 我们就得到了最小的可以 (理论上) 执行任何与人类计算能力等价的模型  
 这些内容看上去十分的平凡, 但是通过一些过程的复合, 我们就能看到更多的魔力.

### 1.4.2 函数: 整合相似过程

我们可以把相似的过程写在一起, 为了简洁和可维护.

下面, 可以阅读 <https://docs.python.org/zh-cn/3/tutorial/controlflow.html#defining-functions> 的 4.7, 4.8.1-4.8.6 节的内容, 把所有代码是怎么执行的放在 `pythontutor` 里面模拟着看一遍. 文字可以不用看, 但是代码一定要执行一遍.

### 递归 (recursion) 过程和栈帧 (stack frame)

观察下面的代码, 可能难以想象是怎么执行的:

```

def fib(n):
    if(n==1):
        return 1
    if(n==2):
        return 1
    else:
        return fib(n-1) + \
            fib(n-2)

```

```
fib(5)
```

像这样用自己调用自己的函数调用通常叫做递归。一个关于递归的有趣定义是：

**定义 1.4.5.** 递归的定义：如果你没有理解什么是递归，那么参见递归。

事实上，我们可以把它放在 `pythontutor` 里面执行一下，发现如下的规则：

- 原来的程序就像是一张纸，上面标注着当前执行到的行数；
- 每次函数调用的时候，就会在一张新的纸片上抄下来调用的内容，并且代换传进来的参数；
- 把这个内容放在原来纸片上面，然后从第一行开始执行；
- 执行完的纸片扔掉。

看上去就像是：

- 你在晚自习上看课外书（执行原来的函数）
- 老师来了，让你写作业（函数调用）
- 你把作业叠放在课外书上，开始做作业（执行函数）
- 做完作业之后你把作业扔了继续看课外书（回到原来的函数）

像羽毛球球桶那样，只能从一个方向插入，弹出的内容的东西叫做“**栈 (stack)**”，由于这些内容通常都是一些数据，由此我们用术语**数据结构 (data structure)**来描述。能被取出来的那个元素是**栈顶 (top of the stack)**，在这个可视化工具里面用蓝色标示出来了。

`Traceback` 就是出错之后，Python 顺着栈一层一层找的结果。`Trace` 是跟踪，`back` 是返回，意思可能就是说堆栈的**回溯 (traceback)**。

## 1.5 一些小例子

下面我们来看一些基础的例子，来体会上一节中的一些思考：

### 片段 1. 热身练习

```
1 def getpercent(chinese, math, english):
2     return (chinese+math+english)/(150+150+150)
3
4 print(getpercent(100, 120, 135))
```

#### Bonus 思考题

如何判断一个程序的行为 (behavior)? 为了刻画程序的行为，我们能不能像第一节里面提到的那样，用一个“模型 (model)”来描述它？

宏观来说，程序就是状态机。程序的执行就是状态的迁移 (transform)。到底有哪些状

态呢?

## 片段 2. 计算钱数之和

```
1 money = [10, 14, 13, 10]
2 int total = 0
3 for i in money:
4     total = total + i
5
6 print(money)
```

这里的 `for i in ...` 只是 `for u in range(len(money)), i=money[u]` 的简称罢了. 对我们的实际程序没有太多的障碍. 还是逃脱不了循环的大框架. 我们在前面说的最小指令集还是起作用的.

## 片段 3. 求一个数是不是素数

```
1 def isprime(x):
2     flag = 1
3     d=2
4     while d<x:
5         if x%d == 0:
6             flag = 0
7             d = d+1
8     if flag:
9         return 1
10    else:
11        return 0
```

在这个程序片段中, `flag` 代表了什么? 是不是有些像现实世界中的信息传递方式? 因为我们无法跨越循环次序改变程序的行为, 我们只好使用变量 `flag` 来记录, 并且让程序的控制流通过 `flag` 来进行判断与执行.

### Bonus 思考题

素数 (prime) 有什么数学性质? 我们是用什么性质来判定性质的? 在我们下达的指令里面, 形式上和数学表达式相似吗?

事实上, 数学上的定义是  $\forall x, \exists p, s.t. p|x$ . 我们的程序看上去并不是像数学那样简洁. 其实, 程序设计家族也有其他成员可以写起来比较优美.

### Passage 文章

## 程序设计家族的其他成员: 不只有下达命令

南京大学 李樾等 节选自《程序分析》教科书

在 IP 中, 指令一个一个给出, 用条件、循环等来控制逻辑 (指令执行的顺序), 同时这些逻辑通过程序变量不断修改程序状态, 最终计算出结果. 我觉得, 尽管 IP 现在都是高级语言了, 但是本质上并没有脱离那种“类似汇编的, 通过读取、写入等指令操作内存数据”的

编程方式（我后面会提及，这是源于图灵机以及后续冯诺依曼体系结构一脉的历史选择）。国内高等教育中接触的绝大多数编程语言都是 IP 的，比如 Java、C、C++ 等。

在 FP 中，逻辑（用函数来表达）可以像数据一样抽象起来，复杂的逻辑（高阶函数）可以通过操纵（传递、调用、返回）简单的逻辑（低阶函数）和数据来表达，没有了时序与状态，隐藏了计算的很多细节。不同的逻辑因为没有被时序和状态耦合在一起，程序本身模块化更强，也更利于不同逻辑被并行的处理，同时避免因并行或并发处理可能带来的程序故障隐患，这也说明了为什么 FP 语言如 Haskell 在金融等领域（高并发且需要避免程序并发错误）受到瞩目。

```

1 (defun quadratic-roots-2 (A B C)
2 (cond ((= A 0) (string "Not a quadratic equation."))
3 (t
4 (let ((D (- (* B B) (* 4 A C))))
5 (cond ((= D 0) (concatenate 'string "x = " (write-to-string (/ (+ (- B) (sqrt D)) (*
6 2 A))))))
7 (t
8 (values (concatenate 'string "x1 = " (write-to-string (/ (+ (- B) (sqrt D)) (* 2 A)
9 (concatenate 'string "x2 = " (write-to-string (/ (- (- B) (sqrt D)) (* 2 A)
10 ))))))))

```

LP 抽象的能力就更强了 (用逻辑来表达)，计算细节干脆不见了。把你想表达的逻辑直观表达出来就好了：如“第三代火影的徒弟”且不是“女性”且“其徒弟也是火影” $\Rightarrow$ “自来也”。“嗯，学会”与或非“，编程都不怕。如今，在数据驱动计算日益增加的背景下，LP 中的声明式语言（Declarative programming language，如 Datalog）作为代表开始崭露头角，在诸多专家领域开拓应用市场。我们这本小书也准备用一章节来教大家如何使用 Datalog 语言编写程序分析器。

```

1 ancestor(A, B) :-
2   parent(A, C),
3   ancestor(C, B).
4

```

#### 片段 4. Perfect 数

```

1 def is_perfect(n):
2     sum=0
3     for i in range(1,n):
4         if n%i==0:
5             sum=sum+i
6     return sum==n

```

其中 range 现在可以认为是生成  $[1, n)$  的列表。并且每一次循环就取列表的下一个元素。比如 for i in range(1,5) 每次循环 i 的值会是 1 2 3 4。

在这个实例中，逻辑关系体现的如何？

#### 片段 5. 求 3 和 5 的因数个数

```

1 def multiples_of_3_and_5(n):

```

```

2 sum=0
3 for i in range(1, n):
4     if i%3==0 or i%5==0:
5         sum=sum+i
6 return sum

```

可以注意我们的逻辑在 Python 里面是如何表达的？还有哪些逻辑表达关系？事实上，这也是后续离散数学部分要学的命题逻辑—我们需要对于以前的逻辑有一个比较确切的定义。

### Bonus 思考题

命题“若  $p$  则  $q$ ”的否定是什么？

普通的高中毕业生基本是无法回答这个问题的。因为课本的知识完全没有提及类似的问题。这就导致我们的高中数学看上去更像是民科学习的数学。同时轻松地毁掉了高中与大学的衔接过程。

### 片段 6. 一定范围内勾股数的个数

```

1 def integer_right_triangles(p):
2     # a^2+b^2=c^2    a+b+c=p    c is the longest
3     count=0
4     for a in range(1,p):
5         for b in range(a,p-a):
6             c=p-a-b
7             if a+b+c==p and a**2+b**2==c**2:
8                 count+=1
9                 print(a,b,c)
10    return count

```

这个例子对于数学的关系好像更加清晰了。比如直角三角形数对有兴致  $a^2 + b^2 = c^2$ 。于是剩下的就比较像自然的“数数”一样了。

### 片段 7: 递归的力量

```

1 def fib(n):
2     if(n==1):
3         return 1
4     if(n==2):
5         return 1
6     else:
7         return fib(n-1) + \
8             fib(n-2)
9
10 fib(5)

```

这个内容也在前方的例子中有提及，这样我们自然的就得出了“栈”的定义。这也是我们这里接触的第一个数据结构—栈。

### Idea 启示



计算机高级程序可以由较为低级的程序解释。这种程序一般而言更加机械，但是更不利于我们的问题的解答。这就需要一层一层的抽象叠加起来。

如果自己曾经动手写过一点代码的话，我们就会发现把代码调试对是一件很不容易的事情。下面我们给出一些小提示：

(1) **阅读程序的报错信息**。我们发现很多同学会对于红色的 Syntax Error 如临大敌，见到就跑。下面，我来举一个例子来说明为什么这是对的：

#### Dialogue 对话

A: 我将会按照一定的规则给出三个数字，我想让你找出这个规则是什么。但是你能够获取信息的途径是：你自己再列举三个数字。我会告诉你这列数字是不是符合我的规则。然后你们就可以说出来你们认为的规则是什么。

B: 好的，明白了。

A: 我说出来的三个数字是 2, 4, 8.

B: 我猜测 16,32,64.

A: 符合我的规则。

B: 那我想规则是  $2^n$ 。

A: 其实并不是这样。

为什么会出现这样的情况？这就是因为没有知道什么东西是“错的”。请观看真理元素的《你能解决这一问题吗》视频，思考一下为什么错误也是很重要的。这位 UP 在 B 站上的官方中文翻译视频链接是<https://www.bilibili.com/video/BV1Hx41157jV>。

(2) **程序出现难以预料的行为时，在脑子里面模拟执行一遍程序**。告诉自己“程序就是状态机”。看一看逻辑设计的是不是出错了。

(3) **善于使用调试器**。观察程序在哪一个地方与你预期的执行不相符。这时候，往往就意味着可以提问了。

#### Passage 文章

### 与其说是学会提问，倒不如说是学会不提问

南京大学 蒋炎岩

中国科学技术大学 余子豪

节选自《PA 实验手册》

很多同学不多不少都会抱有这样的观点：

我向大佬请教，大佬告诉我答案，我就学习了。

但你是否想过，将来你进入公司，你的领导让你尝试一个技术方案；或者是将来你进入学校的课题组，你的导师让你探索一个新课题。你可能会觉得：到时候身边肯定有厉害的同事，或者有师兄师姐来带我。但实际情况是，同事也要完成他的 KPI，师兄师姐也要做他们自己的课题，没有人愿意被你一天到晚追着询问，总有一天没有大佬告诉你答案，你将要如何完成任务？

如果你觉得自己搞不定, 你很可能缺少独立解决问题的能力.

但幸运的是, 这种能力是可以训练出来的. 你身边的大佬之所以成为了大佬, 是因为他们比你更早地锻炼出独立解决问题的能力: 当你还在向他们请教一个很傻的问题的时候, 他们早就解决过无数个奇葩问题了. 事实上, 你的能力是跟你独立解决问题的投入成正比的, 大佬告诉你答案, 展示的是大佬的能力, 并不是你的能力. 所以, 要锻炼出独立解决问题的能力, 更重要的是端正自己的心态: 你来参加学习, 你就应该尽自己最大努力独立解决遇到的所有问题.

很多问题都可以通过查资料解答. 其中, 有一个很好的途径就是先看一看官方文档. 通常官方文档都有非常详细的解释.

## 第二章 简单数理逻辑

“有两种推广. 一种没有什么价值, 另一种则是有价值的. 没什么思想、仅凭唬人的  
专人术语做推广是很容易做到的. 颇为困难的是从若干好的素材中提取出一份精致  
凝练的精品.”  
— Gorge Polya

### 2.1 问题的提出: 为什么要研究简单的数理逻辑

#### Dialogue 对话

A: 问一个这一章比较简单<sup>a</sup>的问题: 如果我有一个命题叫做“若  $p$  则  $q$ ”, 那么这个命题  
的否定是什么?

B: 我研究这个干什么? 闲着找事情吗?

A: 想一想你学习的极限理论. 如果我希望对于“一个数列的极限存在”这个事情做否  
定, 你会怎样否定?

B: 数列极限的定义是如果一个数列的极限是  $A$ , 那么就是说  $\forall n > N, \exists |\epsilon| \geq 0, \text{s.t. } |a_n - A| < \epsilon$ . 要是否定还是真的不是一件容易的事情啊...

A: 这就是我们学习命题逻辑的原因. 以后我们会遇见成百上千的命题等待我们的操  
作, 如何从中找到逻辑就至关重要.

<sup>a</sup>这里的简单并不是不用预习与听课就能自动掌握. 这里的简单指的是相对于数理逻辑这一门科目而言, 这章内容显得十分微不足道且基础.

事实上, 很多同学是在学习数学中体会到逻辑的. 但是, 我们当中的很多人会发现: 数学仅  
仅是为了对付高考这样的考试. 那么希望在这一节以及以后的生活中, 慢慢体会数学带给我们的  
潜移默化的影响.

这一部分我们建议参看教科书《Reading, Writing, and Proving A Closer Look at Mathe-  
matics》的第一章. 我们会在这一章主要概括一下它的主要意思. 由于是为了本科生写的数学课  
本, 所以句子十分的容易懂. 就当做一个小练习吧.

**问题 2.1.1.** 结合自己的数学学习经历, 阅读《Reading, Writing, and Proving A Closer Look at  
Mathematics》的第一章, 然后与下文进行比对. 看一看自己的英语理解能力如何. 不设置时间  
限制, 因为我们需要做的是尽可能的联系自己过去的数学学习经历, 然后去体会这段文本.

我们在生活中经常看到这样的对话:

### Dialogue 对话

学那么多的数学有什么用？买菜又用不到这样的数学，学这些还有用吗？

再后来，我们发现所有的数学问题都可以通过一种“程序化”的手段来解决。比如，我们在上一学期学习的 Gram-Schmidt 正交化矩阵向量的基、解其次线性方程组、求导数等等这样的操作，都有一系列的明确的步骤。

那么数学仅仅是局限于此吗？我们来看一看那些伟大的数学家的思想是什么样的：

伟大的数学家、教育学家 George Polya 专门出了一本书叫做“如何解题”。于是，学习数学一个很重要的目的可能就是教会我们：

- (1) 如何解决一个问题；
- (2) 为什么这样做是对的；
- (3) 这个方法什么时候是对的。

### Example 例子：

我们是如何解含有未知数的等式（通常称为方程），其中一个比较重要的方法是消去律。

对于实数构成的方程，消去律大多数都是成立的（只要等式两端不除以 0），但是对于含有未知矩阵的方程，这样的方法很多时候就不灵了。

在我们遇到一个难以解答的数学问题的时候，还是回过头来看看 George Polya 为我们总结的 How to solve it 的一个 list 吧。

在解答完这些问题之后，我们往往会感到满足。很多时候这也是我们去学习数学的一个很重要的原因。

### Dialogue 对话

A: 可我一点感觉开心也没有啊！

B: 可能是把做题看得太重了。高考的“把题目作对”的观念在大学里面就应该淡化掉了。

A: 此话怎讲？

B: 来看一看朱富海老师的文章就知道了。

### Passage 文章

## 高中数学与大学数学

南京大学 朱富海 节选自数林广记微信公众号

美国大学的数学研究者们对于学生包括中学生的培养的确非常有热情，比如一些名校的博士生在暑假期间常常有打工的机会，主要任务是指导一些高中生尝试做科研。2011 年，MIT 的 Pavel Etingof 教授与另外六位作者合作出版了一本书，题目是 Introduction to Representation Theory。

这本书的内容包括代数、有限群、quiver（箭图）表示论，以及范畴论和有限维代数结构理论，其中的大部分内容在国内高校数学院系的本科甚至研究生课程中都讲不到。在

Etingof 的主页可以找到这本书的 PDF 文档. 他在前言中说, 这本书是他在 2004 年给其他六位合作者的授课讲稿, 而这六位听众当时都是高中生! 其中的 Tiankai Liu 应该是华人, 在 2001, 2002, 2004 年三次代表美国队参加国际数学奥林匹克都获得金牌. 还有一位合作者是来自 South Eugene 高中的 Dmitry Vaintrob, 他在 2006 年获得面向高中生的 Siemens 竞赛的第一名, 论文题目是 The string topology BV algebra, Hochschild cohomology and the Goldman bracket on surfaces, 论文已经涉及到很深的数学理论, 在 Dmitry Vaintrob 的主页上也能找到.

再看看我们在做什么? 曾经看过一道竞赛训练题, 其本质是把八位数 19101112 (华罗庚先生的生日) 分解质因数. 很容易找到因数 8, 然后就一筹莫展了. 后来借助网络工具才直到  $19101112 = 8 \times 1163 \times 2053$ . 看到结果有点傻眼了: 有谁能只用纸笔得到这个分解? 后来发现自己孤陋寡闻了, 有学生说这种分解质因数早就背过! 细细一想真的极为恐怖: 他们为什么要背这个? 他们又背了多少类似的东西?

想想挺有意思: 杰出的数学家们用他们的智慧和汗水去探索和展现数学之美, 而我们花费了大量时间和脑细胞记忆一些很容易遗忘的意义不大的知识点, 轻轻松松地毁掉数学之美的同时顺便浇灭了学生们的求知欲.

### Dialogue 对话

(... 对话仍在继续...)

B: 所以嘛, 我们只要把高考带来的陋习去除掉就行了. 也就是所谓的“去高考化”.

A: 听起来确实很有希望. 我们终于不用再整天因为分数担惊受怕了.

B: 是的, 但是看起来我们都是在这份讲稿里面存在的人物. 希望我们的存在能够对现实世界的你有一定的帮助吧.

同样, 顺着南京大学的问题求解课程, 我们同样找到了一本很有趣的书: 《Mathematics: A Discrete Introduction, Second Edition. Edward R. Scheinerman》. 这本书里面详细讲述了我们为什么要学习数学, 以及数学学习带来的享受.

**问题 2.1.2.** 和上面的问题一样, 带入自己之前的数学学习经历, 然后认真体会这本书写的内容. 只需要阅读第章的前五个小节就好了.

### Dialogue 对话

A: 为什么不让我读第六个小节?

B: 这是因为嘛... 第六小节就是我们下一节课的内容了.

A: 太好了, 我要预习!

B: 这时候终于知道了高中老师说的“预习”的重要性了吧!

A: 确实, 这样一来确实切身感受到了预习的重要性. 这样做可以帮助我了解我的理解哪里出了问题, 于是就可以更加准确地向老师发问, 而不必纠缠于那些可有可无的奇怪问题了.

我们先来看存在于大学数学课本的很多重要的元素和栏目.

条件 A	条件 B	可能吗?
真	真	可能
真	假	<b>不可能!</b>
假	假	可能
假	真	可能

表 2.1: 若条件 A, 则条件 B 的若干种情形

**定义 (definition).**

定义的结构通常形如: X 是一个具有性质 Y 的东西. 其实, 很多情形下, 我们对于定义的理解是很需要时间的, 一般地, 我们需要关注:

**Idea 启示**

对于数学定义, 我们需要关注如下的几个问题:

- 这个定义是怎么来的? 有什么背景?
- 这个定义说的是什么?
- 我们能用更好的方法或不同的角度定义吗?

**命题 (proposition).**

命题是我们关于数学对象的一些陈述性的性质. 那些陈述性的性质, 如果命题是真的, 我们就称为真命题 (有些难以得到的也称作“定理”); 不知道是不是真的命题一般来称为猜想; 错误的命题通常就称为“错误”.

数学中的命题和物理里面的命题有什么不同点? 比如, 我们说 Galileo 的速度变换公式在低速的情形下是成立的, 当速度接近光速  $c$  的时候, 这个公式就不灵了. 换言之, 我们只是使用了一个近似的表达结果. 但是, 数学的逻辑世界中这样的事情是不会发生的. 我们如果认为一个“命题”是真的, 那么在给定公理体系下, 无论什么情形下, 都是对的.

通常描述一个命题的时候, 我们使用的是若  $p$  则  $q$  的形式来完成表述. 那么什么叫做“若...则...”呢? 其实它的意思是对于任何一个真的  $p$ ,  $q$  一定是真的. 具体的关系可以参看表格2.1.

**Bonus 思考题**

若  $p$  则  $q$  还有哪些等价的表示形式? 英语里面有哪些表达方式? 是不是比中文更加自然了?

**问题 2.1.3.** 用“当且仅当”写出的类似表2.1的表格.

关于逻辑连接词, 我们会在后面专门讨论. 下面再来看几个名词. 从上述参考资料中直接摘录了部分结果.

- 结果 (result): A modest, generic word for a theorem. There is an air of humility in calling your theorem merely a "result." Both important and unimportant theorems can be called results.

- 事实 (fact): A very minor theorem. The statement " $6 + 3 = 9$ " is a fact.
- 命题 (proposition): A minor theorem. A proposition is more important or more general than a fact but not as prestigious as a theorem.
- 引理 (lemma): A theorem whose main purpose is to help prove another, more important theorem. Some theorems have complicated proofs. Often one can break the job of proving a complicated theorem down into smaller parts. The lemmas are the parts, or tools, used to build the more complicated proof.
- 推论 (corollary): A result with a short proof whose main step is the use of another, previously proved theorem.

### 证明 (proof).

这个用的就比较多。我们曾经学过很多的方法, 比如反证法, 数学归纳法等。

#### Bonus 思考题

为什么这些东西是对的? 例如, 我们为什么不能用数学归纳法证明含有  $\mathbb{R}$  为变量的命题? 可以使用数学归纳法证明关于无穷的证明吗?

## 2.2 开始行动: 符号化逻辑

### Dialogue 对话

A: 为什么采用符号化的方法? 用自然的语言不是更方便吗?

B: 这个视频可能是给出了一个答案, 里面提出了一些历史。

(数学有一个致命的缺陷 <https://www.bilibili.com/video/BV1464y1k7Ya/> )

A: 我们还要符号化更多的东西吗?

B: 当然! 后面我们的抽象层次还会进一步加深。只有在前面的抽象领域打好坚实的基础, 才可以学得动下面的内容!

A: 举个例子?

B: 现在让你去给刚学完加减乘除的小学生讲数学分析, 能在一个下午让他写出来很好的证明吗?

A: 这当然不行。可能是没有受到一些理论的熏陶, 训练时间不是很足。

B: 确实是这样的。在下面的命题逻辑和谓词逻辑中, 我们会像程序设计语言那样, 关注命题逻辑和谓词逻辑的语法和语义。

### 2.2.1 命题, 真值表和指派

从高中开始, 我们似乎就开始处理各种各样的命题。下面我们加一层抽象, 这样, 我们就可以把一些繁琐的内容交给计算机完成, 并且在探索的过程中对“什么是有效的推理”有一个更深的理解。



符号	名称	英文读法	中文读法	L <sup>A</sup> T <sub>E</sub> X
$\neg$	negation(否定)	not	非	<code>\lnot</code>
$\wedge$	conjunction(合取)	and	与	<code>\land</code>
$\vee$	disjunction(析取)	or	或	<code>\lor</code>
$\rightarrow$	conditional	implies(if then)	蕴含 (如果, 那么)	<code>\to</code>
$\leftrightarrow$	biconditional	if and only if	当且仅当	<code>\leftrightarrow</code>

**定义 2.2.1** (命题 (proposition)). **命题**是可以判定真假的陈述句 (不可既真又假). 其中, 称**真** (true) 与假 (false) 称为命题的**真值** (truth value) .

相仿地, 我们试图像第一章探求的“最小的指令集合”一样, 问一问: 我们表达的逻辑, 有没有一些基础的组成部分?

#### Example 例子:

- (高等代数)  $V$  可分解为  $A$  的特征子空间的直和, 当且仅当  $A$  可以对角化.
- (高等数学) 如果一个数列的极限是  $A$ , 那么就是说  $\forall n > N, \exists |\epsilon| \geq 0, \text{s.t.} |a_n - A| < \epsilon$ .
- (解析几何) 两直线的方向向量  $s_1 = (l_1, m_1, n_1), s_2 = (l_2, m_2, n_2)$  垂直的充要条件是  $l_1 l_2 + m_1 m_2 + n_1 n_2 = 0$ ; 平行的充要条件是  $\frac{l_1}{l_2} = \frac{m_1}{m_2} = \frac{n_1}{n_2}$ .

**定义 2.2.2** (命题逻辑的语法). 命题逻辑的语言有且仅有如下的内容构成:

- 任意多的命题符号
- 5 个逻辑连接词 (见表2.2.1)
- 左括号, 右括号

#### Dialogue 对话

A: 为什么没有常见的“任意”, “存在”之类的量词?

B: 因为这样就复杂了. 包括结果判定的正确性和推理的难度上. 我们下一章会详细讲这个.

#### Bonus 思考题

为什么叫合取, 为什么叫析取?

其实命名的关键在于描述中的“和”和“析”. 我们可以查询古汉语字典来获取他们的意思. 并且从中找到一些合理性.

下面, 不妨用 Python 语言为例, 来看一看这些内容是如何在语言中有所设计的. 需要注意的是, 上表格中的后两个记号并不是新的. 只是我们经常用他们, 于是就变成了一个独立的记号.

如果  $p$  那么  $q$  的意思是: 如果  $p$  是对的, 那么  $q$  一定是对的, 如果  $p$  是错的, 那么  $q$  的真假性不确定. 因此, 我们可以把  $p \rightarrow q$  表示为  $\neg p \vee q$ —意味着要么  $p$  不成立, 要么当  $p$  成立的时候  $q$  是对的—命题只有对和错, 所以我们就很干净的进行了一次分类讨论.



当且仅当的表示就是“若  $p$  则  $q$ , 且若  $q$  则  $p$ ”. 本质上还是命题符号之间的“非”, “或”, “与”之间的连接.

#### Example 例子:

命题的真假在 Python 中可以用布尔表达式 (boolean expression) 的真 (true) 和假 (false) 表示. 比如今有变量  $a=True, b=False, c=True$ , 那么

```
1 a=True, b=False, c=True
2 var = a and b or not c
3 print(var)
4
```

打印的真值就是就是  $a \wedge b \vee \neg c$  的真值.

既然我们规定了符号, 自然要研究一下他们的运算律和运算关系. 什么是运算律?

#### Example 例子:

我们从小就开始听到运算律的相关内容了. 那么什么是运算律? 某国外网站的解释如下.

The order of operations is a rule that tells the correct sequence of steps for evaluating a math expression. We can remember the order using PEMDAS: Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right).

但是我们还听说过“交换律 (commutative law)”, “结合律 (associative law)”这样的名词. 这些内容反应了我们可以如何书写, 如何计算一个表达式.

#### Idea 启示

除了这些, 对于命题, 我们需要了解这四个内容:

- 一个可以判定“真/假”的“东西”-命题
  - 魏恒峰是南京大学的教师.
- 简单命题组成更复杂的命题-连接词
  - 如果这门课是魏恒峰老师教的, 那么他一定是很受欢迎的.
- 深入命题的内部-谓词与变元
  - 在“今天下雨了.”和“昨天下雨了”之间建立逻辑联系.
- 体现“普遍性”与“存在性”-量词
  - 任何一个学生计算机科学的学生都值得学习魏恒峰老师的离散数学课.

现在, 我们就可以探讨更多的语义相关的内容了. 来看一看能对这个公式产生怎样的解释.

**定义 2.2.3** (命题符号的运算规则). 一般地, 命题记号遵循如下的运算规则:

$p$	$\neg p$
T	F
F	T

表 2.2: “非”的真值表

- 最外层的括号可以省略
- 优先级:  $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- 结合性: 右结合. 例如  $(\alpha \wedge \beta \wedge \gamma)$  表示  $\alpha \wedge (\beta \wedge \gamma)$ ,  $\alpha \rightarrow \beta \rightarrow \gamma$  表示  $\alpha \rightarrow (\beta \rightarrow \gamma)$

那么我们说的命题的真假是怎么界定的呢? 通常情况下, 我们需要分类讨论每一个需要讨论的命题的真假, 最后看一看根据公式表达的真假性就行了. 所以, 我们很多时候希望“假定”这些命题的真假, 来考察最终结论的真假, 并且希望从中找到一点规律. 这样做其实有一个更专业的名字叫“真值指派”. 这样我们就可以对于“这句话永远是对的”有一个更加深刻的定义.

#### Dialogue 对话

A: 我现在知道这些条件是, 对于明天高等数学课程的一些事情.

B: 什么事情?

A: 如果明天老师讲完了《空间立体几何》, 那么他就会做一个小测试. 同时如果我如果学的非常差的话, 并且旁边还没有大佬捞我的话, 我的平时分就非常惨淡.

B: 那我来分析一下, 假设老师没有讲完, 那么平时分暂时还不会受到影响; 如果老师讲完了, 同时我学得不差, 平时分也不会受到影响. 如果老师讲完了, 我学得非常差, 但是有人捞, 那么平时分也不会受到影响. 但是这是违反学术诚信的, 并不能做. 所以, 只要自己学得比较好才能得到很好的平时分.

像上面的例子, 我们通常会对命题的一些内容的真假进行预先假定, 即: 对于命题进行真值指派.

**定义 2.2.4** (真值指派 ( $v$ )). 令  $S$  为一个命题符号的集合.  $S$  上的一个真值指派  $v$  是一个从  $S$  到真假值的映射

$$v : S \rightarrow \{T, F\}.$$

具体的, 我们可以“指派”命题符号中的各个变量的值, 然后映射到真或假两种情况.

我们可以借助这个想法为我们上面定义的为我们上面的逻辑连接词做一个精确的数学定义. 叫做“真值表”.

**定义 2.2.5** (真值表). 表征逻辑事件输入和输出之间全部可能状态的表格. 列出命题公式真假值的表. 通常以 1 表示真, 0 表示假.

比如, 我们有“非”的真值表 (表2.2)、“和”的真值表 (表2.3)、“或”真值表 (表2.4)、“若, 则”真值表 ((表2.5) 以及“当且仅当”的真值表 (表2.6)

**问题 2.2.1.** 人类学者埃贝尔考察一个有着许多古怪社会现象的群岛, 他到访的第一个小岛上的居民分为两类, 而且每人必属其中的一类:

$p$	$q$	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

表 2.3: “和”的真值表

$p$	$q$	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

表 2.4: “或”的真值表

$p$	$q$	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

表 2.5: “如果, 那么”的真值表

$p$	$q$	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

表 2.6: “当且仅当”的真值表

- Knight: 这类人永远说真话
- Knave: 这类人永远说假话

在岛上埃贝尔遇到一行三人，且称他们为 A, B, C。埃贝尔问 A: “你是 knight 还是 knave?” A 回答了，但埃贝尔没听清；于是埃贝尔就问 B: “他 (A) 说的是什么?” B 告诉埃贝尔 A 说自己是 knave。

此时，C 插话说：“别相信他 (B)，他说谎!”

我们的问题是：C 是 knight 还是 knave?

事实上，我们在小学很可能通过列举的方法完成求解。但是现在我们可以用真值表列举。甚至可以把公式写出来进行推演!

### Bonus 思考题

等等，什么叫推演？有哪些推演规律？这些推演规律是不是可以用公式表示？这些都是下一节要介绍的内容。

比如一个命题的否定的否定还是原命题本身一样，我们可以定义一些“公式”。比如  $\neg p$  与  $p$  等价。首先我们定义一下什么叫“满足”，“蕴含”或者“等价”。

**定义 2.2.6** (满足 (Satisfy)). 如果  $\bar{v}(\alpha) = T$ ，则称真值指派  $v$  满足公式  $\alpha$ 。

很多时候一些逻辑表达式看上去就是废话。比如“如果我后天知道了考试的成绩，那我明天就知道了”。数学上面对这类问题有一个定义叫做“重言蕴含”。

**定义 2.2.7** (重言蕴含 (Tautologically Implies)). 设  $\Sigma$  为一个公式集。

$\Sigma$  重言蕴含公式  $\alpha$ ，记为  $\Sigma \models \alpha$ ，

如果每个满足  $\Sigma$  中所有公式的真值指派都满足  $\alpha$ 。

**定义 2.2.8** (重言式/永真式 (Tautology)). 如果将等价词两侧的子公式各自看作表达式，则这两个逻辑表达式对于相关逻辑变量的任意赋值有相同的逻辑值。(或者：如果  $\emptyset \models \alpha$ ，则称  $\alpha$  为重言式，记为  $\models \alpha$ 。)

反之，就是永远都不能成立的矛盾的形式。

**定义 2.2.9** (矛盾式/永假式 (Contradiction)). 若公式  $\alpha$  在所有真值指派下均为假，则称  $\alpha$  为矛盾式。

**定义 2.2.10** (重言等价 (Tautologically Equivalent)). 如果  $\alpha \models \beta$  且  $\beta \models \alpha$ ，则称  $\alpha$  与  $\beta$  重言等价，记为  $\alpha \equiv \beta$ 。

对于命题公式而言，如果对于相同的变量输入，外部观测的结果的真假相同，我们就称为命题的等价。

**定义 2.2.11** (命题的等价). 设  $G, H$  是两个命题公式， $P_1, \dots, P_n$  是出现在命题  $G, H$  中的所有变元，如果对于  $P_1, P_2, \dots, P_n$  的  $2^n$  组不同的解释， $G, H$  的真值结果都相同，那么称  $G$  和  $H$  是等价的，记作  $G \Leftrightarrow H$ 。

根据上面的定义，我们可以证明：

**定理 2.2.1.**  $G \Leftrightarrow H$  的充分必要条件是  $G \leftrightarrow H$ 。

### 2.2.2 命题逻辑的化简

在上面我们发现了有很多的“废话”，但是，当看上去是“废话”的东西堆多堆复杂的时候，那么它就不是显然的。这就需要一些推理规律来帮助我们联通看上去毫不相干的逻辑符号。

经过我们的探讨，我们就希望把一些最基本的规律写出来：

**命题 2.2.1.** (逻辑的运算律 (1)) 如果  $A, B, C$  是命题，那么以下内容是永真式：

- 交换律：

$$(A \wedge B) \leftrightarrow (B \wedge A)$$

$$(A \vee B) \leftrightarrow (B \vee A)$$

- 结合律：

$$((A \wedge B) \wedge C) \leftrightarrow (A \wedge (B \wedge C))$$

$$((A \vee B) \vee C) \leftrightarrow (A \vee (B \vee C))$$

- 分配律：

$$(A \wedge (B \vee C)) \leftrightarrow ((A \wedge B) \vee (A \wedge C))$$

$$(A \vee (B \wedge C)) \leftrightarrow ((A \vee B) \wedge (A \vee C))$$

- De Morgan 律：

$$\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$$

$$\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$$

- 双重否定律：

$$\neg\neg A \leftrightarrow A$$

- 排中律：

$$A \vee (\neg A)$$

- 矛盾律：

$$\neg(A \wedge \neg A)$$

- 逆否命题：

$$(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)$$

这些为什么有用呢？考虑有一天你在求解一个数学问题，其中你想把一个命题否定掉，比如

**问题 2.2.2.** 如果  $P, Q, R$  是命题，请否定  $P \rightarrow Q \wedge R$ 。

其中一个很重要的手段就是通过上面的这些重言式的替换。就像在学习三角函数的时候使用三角恒等式替换一样。

下面我们来看一个比较有趣的逻辑代数推演的例子：

**问题 2.2.3.** 我们已经知道 Bill, Jim 和 Sam 分别来自 Boston, Chicago 和 Detroit. 以下每句话半句对, 半句错:

- Bill 来自 Boston( $p_1$ ), Jim 来自 Chicago( $p_2$ ).
- Sam 来自 Boston( $p_3$ ), Bill 来自 Chicago( $p_4$ ).
- Jim 来自 Boston( $p_5$ ), Bill 来自 Detroit( $p_6$ ).

能确定每个人究竟谁来自何处吗?

**解答:** 我们可以将上述条件用以下逻辑表达式来表示:

$$((p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)) \wedge ((p_3 \wedge \neg p_4) \vee (\neg p_3 \wedge p_4)) \wedge ((p_5 \wedge \neg p_6) \vee (\neg p_5 \wedge p_6))$$

先看前两个括号 (上述式子红色的部分), 以连接两个式子中间的  $\wedge$  展开 (下式红色符号), 我们有

$$\begin{aligned} & ((p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)) \wedge ((p_3 \wedge \neg p_4) \vee (\neg p_3 \wedge p_4)) \\ &= (p_1 \wedge \neg p_2 \wedge p_3 \wedge \neg p_4) \vee (p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge p_4) \vee (\neg p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4) \vee (\neg p_1 \wedge p_2 \wedge \neg p_3 \wedge p_4) \end{aligned}$$

根据已知条件,  $p_1 \wedge p_4, p_2 \wedge p_4, p_1 \wedge p_3$  均为假的, 所以上述式子是

$$(\neg p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4)$$

与后面的  $((p_5 \wedge \neg p_6) \vee (\neg p_5 \wedge p_6))$  进行  $\wedge$  操作, 也就是有

$$\begin{aligned} & (\neg p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4) ((p_5 \wedge \neg p_6) \vee (\neg p_5 \wedge p_6)) \\ &= (\neg p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4 \wedge p_5 \wedge \neg p_6) \vee (\neg p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge p_6) \\ &= (\neg p_1 \wedge p_2 \wedge p_3 \wedge \neg p_4 \wedge \neg p_5 \wedge p_6) \end{aligned}$$

所以我们知道:  $p_2, p_3, p_6$  是对的.

#

事实上, 我们能这样做是因为有带入定理帮助我们.

**定理 2.2.2** (带入定理). 运用永真式代替命题的变元, 得到的命题结果与原命题等价.

### Bonus 思考题

缺失的证明: 为什么没有了定理的证明?

一个很重要的问题是我们数学的基石是缺失的. 其实, 人类在认识世界的时候开始也是缺乏基础的, 仅仅凭借直觉来建立一些体系. 直到直觉无法完全覆盖的时候, 人类才开始探索有没有什么基础的支撑点来支持这一系列理论.

### 2.2.3 化简的目标: 范式

范式的意思是“规范的形式”. 那么, 这些内容化简到最后有没有一个目标呢? 其实是有的. 任何一个命题都可以写成“合取范式 (CNF)”或者“析取范式 (DNF)”的形式. 下面给出形如这样的式子的定义:

**定义 2.2.12** (合取范式 (Conjunctive Normal Form)). 我们称公式  $\alpha$  是**合取范式**, 如果它形如

$$\alpha = \beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_k,$$

其中, 每个  $\beta_i$  都形如

$$\beta_i = \beta_{i1} \vee \beta_{i2} \vee \cdots \vee \beta_{in},$$

并且  $\beta_{ij}$  或是一个命题符号, 或者命题符号的否定.

**定义 2.2.13** (析取范式 (Disjunctive Normal Form)). 我们称公式  $\alpha$  是**析取范式**, 如果它形如

$$\alpha = \beta_1 \vee \beta_2 \vee \cdots \vee \beta_k,$$

其中, 每个  $\beta_i$  都形如

$$\beta_i = \beta_{i1} \wedge \beta_{i2} \wedge \cdots \wedge \beta_{in},$$

并且  $\beta_{ij}$  或是一个命题符号, 或者命题符号的否定.

**定理 2.2.3.** 每一个命题都有等价的合取范式和析取范式的形式. (Given any proposition, there exists a proposition in disjunctive normal form which is equivalent to that proposition.)

证明. (Copied from <https://planetmath.org/everypropositionisequivalenttoapropositionindnf>) Any two propositions are equivalent if and only if they determine the same truth function. Therefore, if one can exhibit a mapping which assigns to a given truth function a proposition in disjunctive normal form such that the truth function  $f$  of this proposition is  $f$ , the theorem follows immediately.

Let  $n$  denote the number of arguments  $f$  takes. Define

$$V(f) = \{X \in \{T, F\}^n, f(X) = T\}$$

For every  $X \in \{T, F\}^n$ , define  $L_i(X) = \{T, F\}^n \rightarrow \{T, F\}$  as follows:

$$L_i(X)(Y) = \begin{cases} Y_i, & X_i = T; \\ \neg Y_i, & X_i = F. \end{cases}$$

Then, we claim that

$$f(Y) = \bigwedge_{x \in V(f)} \bigvee_{i=1}^n L_i(X)(Y)$$

On the one hand, suppose that  $f(Y) = T$  for a certain  $Y \in \{T, F\}^n$ . By definition of  $V(f)$ , we have  $Y \in V(f)$ . By definition of  $L_i$ , we have

$$L_i(Y)(Y) = \begin{cases} Y_i, & Y_i = T \\ \neg Y_i, & Y_i = F \end{cases}$$

In either case,  $L_i(Y)(Y) = T$ , since a conjunction equals  $T$  if each term of the conjunction equals  $T$ , it follows that  $\bigvee_{i=1}^n L_i(Y)(Y) = T$ . Finally, since a disjunction equals  $T$  if and only

if there exists a term which equals  $T$ , it follows the right hand side equals  $T$  when the left-hand side equals  $T$ .

On the one hand, suppose that  $V(Y) + F$  for a certain  $Y \in \{T, F\}$ . Let  $X$  be any element of  $V(f)$ . Since  $Y \notin V(f)$ , there must exist an index  $i$  such that  $X_i \neq Y_i$ . For this choice of  $i$ ,  $Y_i = \neg X_i$ , Then we have

$$L_i(X)(Y) = \begin{cases} \neg X_i, X_i = T \\ \neg \neg X_i, X_i = F \end{cases}$$

In either case,  $L_i(X)(Y) = F$ , Since a conjunction equals  $F$  if and only if there exists a term which evaluates to  $F$ , it follows that  $\bigvee_{i=1}^n = F$  for all  $X \in V(f)$ . Since a disjunction equals if and only if each term of the conjunction equals  $F$ , it follows that the right hand side equals  $F$  when the left-hand side equals  $F$ .

□

上面的只是给了我们一个正确性证明, 但是并没有告诉我们如何把一个式子化为合取范式或者析取范式. 一般的, 我们有如下的方法:

- (1) 用  $\neg, \wedge, \vee$  代替  $\rightarrow, \leftrightarrow$ ;
- (2) 用双重否定律, 消去律去掉多余的否定连接词, 运用 De Morgan 律将否定连接词内移.
- (3) 利用分配率, 结合律, 幂等律整理得到.

实际上, 在上面的三个人从哪里来的例子中, 我们就用到了这样的想法.

### Bonus 思考题

为什么合取范式 (外面  $\wedge$  里面  $\vee$ ) 和析取范式 (外面  $\vee$  里面  $\wedge$ ) 这么重要?

下面的这个回答来源于 StackExchange 上面的回答 [1], 简要概括如下:

逻辑公式中的变量 (输入) 可以以复杂的方式混在一起. 如果公式采用 CNF 或 DNF, 变量就更加分离, 从而更容易看出表达式何时成立. 比如, 要检查 CNF 是否成立, 只需逐个检查每个子句有一个是假的整个都是假的. DNF 也类似: 逐个检查子句, 并在找到一个为真的子句时停止, 整个句子都是真的.

很多时候真值表会带来很多的麻烦: 意味着穷举和非常麻烦的事情. 有了 CNF 与 DNF 以后, 我们不必从真值表开始枚举, 可以通过操作表达式来形式地构造标准形式. 在某些情况下可能可以更方便一些.

下面我们来看一看合取范式与析取范式的一些性质.

我们发现, 在合取范式中, 只要有一个条件是假的, 那么整个命题都是假的. 在析取范式中, 只要有一个条件是真的, 那么这个命题是真的. 特别地, 如果每个命题变元或其否定恰好只有一个出现过并且只出现了一次, 那么我们说这个合取范式为极小项 (析取范式为极大项). 比如我们有两个变量, 那么就可以有如下的四个逻辑表达式:

$$P \wedge Q \quad \neg P \wedge Q \quad P \wedge \neg Q \quad \neg P \wedge \neg Q$$

这样就保证了我们除了枚举输入的命题之外, 我们还可以枚举极小项, 极大项, 从两方面来观察命题的真假. 比如, 还是上面的两个命题符号的情形, 对于合取范式和析取范式, 分别有表2.2.3 以及表2.2.3的情形.



P Q	$\neg P \wedge \neg Q$	$\neg P \wedge Q$	$P \wedge \neg Q$	$P \wedge Q$
0 0	1	0	0	0
0 1	0	1	0	0
1 0	0	0	1	0
1 1	0	0	0	1

表 2.7: 合取范式枚举命题符号是否取反以及指派为真假的情形 (极小项)

P Q	$\neg P \wedge \neg Q$	$\neg P \wedge Q$	$P \wedge \neg Q$	$P \wedge Q$
0 0	0	1	1	1
0 1	1	0	1	1
1 0	1	1	0	1
1 1	1	1	1	0

表 2.8: 合取范式枚举命题符号是否取反以及指派为真假的情形 (极大项)

**Bonus 思考题**

列表有什么好处？我们可以发现什么规律？

我们发现每一个赋值对应一个取值为  $\frac{真}{假}$  的极  $\frac{小}{大}$  项，每个极  $\frac{小}{大}$  项的  $\frac{真}{假}$  赋值是唯一的。并且没有等价的两个极  $\frac{小}{大}$  项，任何两个不同的极  $\frac{小}{大}$  项  $\frac{合}{析}$  取必为真，极  $\frac{小}{大}$  的否定是极  $\frac{大}{小}$ ，所有的极  $\frac{小}{大}$  项的  $\frac{析}{合}$  取为永假公式。

**Dialogue 对话**

A: 这个和二进制表示好像。我们能不能有一种方法来用一个整数来表示命题？

B: 确实，但是需要考虑一下命题变元的记号问题。

如果我们把合取范式和析取范式里面的内容都变为极小项/极大项的时候，我们就可以得到更标准，更易于处理的内容。通常称为主合（析）取范式。

## 2.3 命题推演的真象：命题的推演

我们发现，蕴含词的逻辑永真式和我们经常所做的推理有很好的联系。具体地，如果  $\alpha \rightarrow \beta$ ，那么如果  $\alpha$  成立，可以推导出  $\beta$  为真。

同样的，如果假设  $\alpha$  成立，只要经过合理的推导可以推导出  $\beta$  为真，那么这个规则就是成立的。

我们在推理时多半用“因为... 所以...”，那么前面的什么概念能起关键作用呢？

一般而言，逻辑推理有一系列的假设前提  $A_1, A_2, \dots, A_n$ ，有一个结论  $B$ ，什么是正确的推理呢？形式化地，对前提与结论所涉及的逻辑变量任意赋值，如果任意的  $A_i$  均为这真，它们的合取式的值当然也为真，那么  $B$  必为真。

形式化的，我们就说：这样的推理是正确的。

**定义 2.3.1.** 设  $G_1, G_2, \dots, G_n, H$  是命题公式, 对任意指派  $I$ , 如果  $G_1 \wedge G_2 \wedge \dots \wedge G_n$  为真,  $H$  也为真, 或者  $G_1 \wedge G_2 \wedge \dots \wedge G_n$  为真,  $H$  为假, 那么称  $H$  是  $G_1, G_2, \dots, G_n$  的逻辑结果. 或者  $G_1, G_2, \dots, G_n$  共同蕴含  $H$ , 记做  $G_1 \wedge G_2 \wedge \dots \wedge G_n \Rightarrow H$ .

**定理 2.3.1.** 公式  $H$  是前提集合  $\{G_1, G_2, \dots, G_n\}$  的逻辑结果当且仅当  $G_1 \wedge G_2 \wedge \dots \wedge G_n \rightarrow H$  为永真的公式.

需要注意的是,  $\Rightarrow$  是一个关系, 这是一个不可计算的公式.

**定义 2.3.2.** 设  $G_1, G_2, \dots, G_n, H$  是命题公式, 如果  $H$  是  $G_1, G_2, \dots, G_n$  的逻辑结果, 那么称  $G_1, G_2, \dots, G_n \Rightarrow H$  是有效的或正确的. 否则称为无效的.

称  $G_1, G_2, \dots, G_n$  为一组前提,  $H$  为结论, 如果记  $\Gamma = \{G_1, G_2, \dots, G_n\}$ , 上述关系可以记作  $\Gamma \Rightarrow H$ .

### Dialogue 对话

A: 如果我有一个有效的推理, 那么结论应该是正确的吧?

B: 那可不一定. 举个例子:  $1+1=3, 2+2=4 \Rightarrow 114+514=191$  虽然形式是有效的, 但是结论确实是不对的.

A: 确实, 很像是  $\rightarrow$  记号的拓展. 如果前提条件为假, 那么就不关心结论的正确性了.

所以这就解释了我们为什么经常使用的因为... 所以... 是合理的了. 比如  $(p \wedge (p \wedge q)) \rightarrow q$  是永真的, 那么就可以有三段论这样的合理的推理过程了.

### Example 例子:

一个三段论的示例如下:

关爱学生, 好好讲课, 有真才实学的老师会被学生欢迎

朱富海老师关爱学生, 好好讲课, 有真才实学

朱富海老师受学生欢迎.

我们经常对于知识研究一些规律. 比如在逻辑公式很长的时候, 逐个枚举是十分低效率的. 比如, 如果有 200 个命题符号的逻辑公式, 用计算机枚举则需要  $2^{200}$  种情况. 用 Python 的强大的计算器可以知道, 需要核验的情况是一个 61 位数.

### Example 例子:

我们可以使用 `ipython` 中输入 `len(str(2**200))` 知道它的位数. Python 自带高精度计算, 因此如果直接输入 `2**200`, 它就会直接把数字算出来!

既然要推理, 我们就要有推理的定律和推理的推理的规则. 正如前面说的程序语言的语法和语义类似.

经过总结归纳, 我们有如下的推理定律:

**定理 2.3.2** (常见的推理定律). 设  $G, H, I, J$  是任意的命题公式, 那么我们有:

(1) 简化规则:  $I_1 : G \wedge H \Rightarrow G, I_2 : G \wedge H \Rightarrow H$ .

(2) 添加规则:  $I_3 : G \Rightarrow G \vee H, I_4 : H \Rightarrow G \wedge H$ .

- (3)  $I_5 : G, H \Rightarrow G \wedge H$ .
- (4) 选言三段论:  $I_6 : \neg G, G \vee H \Rightarrow H$ .
- (5) 分离规则:  $I_7 : G, G \rightarrow H \Rightarrow H$ .
- (6) 否定结论:  $I_8 : \neg H, G \rightarrow H \Rightarrow \neg G$ .
- (7) 连续推理:  $I_9 : G \rightarrow H, H \rightarrow I \Rightarrow G \rightarrow I$ .
- (8) 两难推理:  $I_{10} : G \vee H, G \rightarrow I, H \rightarrow I \Rightarrow I$

一个学习的很好的方法是举一些例子. 我们可以用上面的符号翻译成具体的例子. 不用太过于拘束.

#### Idea 启示

遇见新概念的时候, 可以应该举一些例子辅助理解, 尽可能弄懂来龙去脉.

以最后一个例子为例, 有:

#### Example 例子:

8 现在发动机坏了并且没油了. 如果发动机坏了, 车子就动不了了; 如果没油了, 车子就动不了了; 我们可以得到: 车子动不了了.

#### Bonus 思考题

为什么叫他们为基础的规则? 是不是因为他们具有某种完备性?

推理规则可以使用真值表验证. 除此之外, 我们还需要推理定律: 也就是需要完成什么样的变换. 我们有三条规则.

(1) Premise 规则: 前提引用. 如果推理的时候引入了前提集合中的一个前提, 那么称为使用了 P 规则;

(2) Transformation 规则: 逻辑结果引用的规则. 在推理过程中, 如果引入推理过程中某个产生的中间结果, 那么称使用了 T 规则;

(3) Conclusion premise 规则: 附加前提规则. 在推理过程中, 如果逻辑结果为一个含有蕴含  $\rightarrow$  的式子, 那么把它的前提条件作为假设引入, 就是用了 CP 规则. 所有引入的假设最终必须被“释放” (discharged) 所谓释放, 其实就是在使用假设  $\alpha$  作为假设的前提下推出了  $\beta$ , 最后要写成  $\alpha \rightarrow \beta$  的形式. 这就是假设的释放.

我们来推导一下  $I_6$ .

证明. (1) 直接引入前提集合的前提, 有  $\neg G$ ;

(2) 直接引入前提集合的前提, 有  $G \wedge H$ ;

(3) 使用 (1),(2) 的中间结果, 有  $\neg G \wedge (G \vee H)$ ;

(4) 使用 (3) 的中间结果, 使用分配律, 有  $(\neg G \wedge G) \vee (\neg G \wedge H)$ ;

(5) 使用 (4) 的中间结果, 有  $\neg G \wedge H$ ;

(6) 使用 (5) 的中间结果, 有  $H$ . □

对于  $I_9$ , 我们可以先把  $G$  是对的引入集合, 然后最后推理得到  $I$  的时候, 为了说明它的正确性, 就需要使用 CP 规则.

**Bonus 思考题**

P 规则和 T 规则能足够做所有的内容的推理吗?

在上面我们做的研究中, 我们会发现很多东西相当的机械化. 于是, 在有机械化的内容出现的时候, 计算机就有用了. 所以, 我们希望找到一种方法, 使得计算机可以自动地推理我们的命题. 在 1965 年, J.A. Robinson 发现了消解原理: 它的原理是规则  $I_7$ . 具体的有:

**定义 2.3.3** (消解与消解式). 设  $G$  和  $H$  是两个析取式, 如果  $G$  中有变元  $P$ ,  $H$  中有变元  $\neg P$ , 则从  $G$  和  $H$  中分别消去  $P$  和  $\neg P$ , 并且将余下的部分析取, 构成新的表达式  $W$ , 这个过程称为消解.  $W$  为  $G, H$  的消解式.

我们经过消解的方法, 立即可以得到一个很有“自动化”意思的定理:

**定理 2.3.3.** 如果  $W$  是  $G, H$  的消解式, 则  $G \wedge H \Rightarrow W$ .

## 2.4 表达更丰富的内容: 谓词 部分与整体的关系

我们发现命题逻辑无法表达部分与整体的关系. 例如:

**Example 例子:**

- (1) 张三是个法外狂徒;
- (2) 李四与张三是好朋友;
- (3) 李四也是一个法外狂徒;
- (4) 王五站在张三与李四中间; (害怕极了)
- (5) 王五长得比张三与李四都高;
- (6) 所有的法外狂徒终将绳之以法;
- (7) 存在法外狂徒改过自新.

我们发现以前学过的东西仅仅能应对很基本的内容. 比如任意, 存在无法表示. 下面我们形式化的先来说说什么是什么任意和存在. 这些是对于一类类似于变元的内容的性质的总体描述. 因此, 我们首先定义什么是谓词

**定义 2.4.1** (谓词 (predicate)). 在不能被拆分为更小的命题中 (通常叫做原子命题 (atomic expression)), 可以独立存在的客体 (subject) (通常是句子中的主语, 宾语) 称为个体词 (individual), 描述客体的性质或客体之间的关系的部分称为谓词.

很多时候, 我们会泛指一些很笼统的概念, 比如“人”. 也有时候我们会举一些明确的个体, 比如“蒋炎岩老师”. 这时候我们就需要区分“个体变量”(通常是泛指) 和“个体常量”(通常是特指).

**定义 2.4.2** (个体常量, 个体变量, 个体域, 全总个体域). 对于具体明确的个体而言, 称为个体常量 (individual constant), 一般用形如  $a, b, a_1, b_1$  这样的小写字母表示. 个体变量 (individual variable) 取值范围称作个体域 (或论域, individual field), 通常用字母  $D$  表示. 我们通常把宇宙间所有个体聚集在一起的个体域称为全总论域 (universal individual field).

**定义 2.4.3** ( $n$  元命题函数). 设  $P(x_1, x_2, \dots, x_n)$  是  $n$  元函数 (function), 其中每一个变量可以取其可以取到的对应的值. 如果  $P$  的值域是  $\{0, 1\}$ , 那么说  $P(x_1, x_2, \dots, x_n)$  是  $n$  元命题函数或  $n$  元谓词 ( $n$ -ary propositional function).

这样子, 我们就可以更加方便的定义量词 (qualifier) 了.

**定义 2.4.4** (全称量词与特称量词). 表达全部数量关系的词语诸如“一切”, “所有”, “每一个”等称为全称量词 (universal quantifier), 记作  $\forall$ . 如所有的  $x$  记作  $\forall x$ .

表达部分数量关系的词语诸如“存在”, “有一个”, “有一些”称为存在量词 (existential quantifier). 记作  $\exists$ . 如有一些  $x$  记作  $\exists x$ .

这里面,  $x$  被称为作用变量 (function variable), 一般将量词加在对应的谓词之前, 如  $\forall x.F(x)$  或  $\exists x.F(x)$ . 此时  $F(x)$  被称为全称量词和存在量词的作用域 (又成为辖域, scope).

上面的内容一个很重要的地方在于引入了变元. 例如, 在函数符号和谓词符号中, 变元的引入使得我们要讨论的事情变得更加多了.

### Bonus 思考题

你哪里还见到过由常数引入到变量? 其实, 我们在初中的时候, 先研究了二次方程的解, 再把  $x$  当做一个变元, 研究所有  $x$  构成的二元组  $(x, f(x))$  的一些性质. 有兴趣学习高等代数的同学会发现在引入  $\lambda$ -矩阵 的时候也用了这样的知识.

这下子, 我们就可以定义谓词逻辑的构成了.

**定义 2.4.5** (谓词逻辑的构成). 谓词逻辑由且仅有以下的几部分构成:

- (1) 逻辑联词:  $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- (2) 量词符号:  $\forall$  (forall; 全称量词),  $\exists$  exists; 存在量词)
- (3) 变元符号:  $x, y, z, \dots$
- (4) 左右括号:  $(, )$
- (5) 常数符号: 零个或多个常数符号  $a, b, c, \dots$ , 表达特殊的个体
- (6) 函数符号:  $n$ -元函数符号  $f, g, h, \dots$  ( $n \in \mathbb{N}^+$ ), 表达个体上的运算
- (7) 谓词符号:  $n$ -元谓词符号  $P, Q, R, \dots$  ( $n \in \mathbb{N}$ ), 表达个体的性质与关系

和普通公式一样, 谓词公式也是由一项一项构成的.

**定义 2.4.6** (项 (Term)). 1. 每个变元  $x, y, z, \dots$  都是一个项;

2. 每个常数符号都是一个项;

3. 如果  $t_1, t_2, \dots, t_n$  是项, 且  $f$  为一个  $n$ -元函数符号, 则  $f(t_1, t_2, \dots, t_n)$  也是项;

4. 除此之外, 别无其它.

**定义 2.4.7** (公式 (Formula)). 谓词逻辑的公式定义如下:

- 如果  $t_1, \dots, t_n$  是项, 且  $P$  是一个  $n$  元谓词符号, 则  $P(t_1, \dots, t_n)$  为公式, 称为原子公式;

- 如果  $\alpha$  与  $\beta$  都是公式, 则  $(\neg\alpha)$  与  $(\alpha * \beta)$  都是公式;
- 如果  $\alpha$  是公式, 则  $\forall x. \alpha$  与  $\exists x. \alpha$  也是公式;
- 除此之外, 别无其它.

我们来举一些例子:

**Example 例子:**

- 0 不是任何自然数的后继

$$\forall x. \neg(Sx = 0)$$

- 两个自然数相等当且仅当它们的后继相等

$$\forall x. \forall y. (x = y \leftrightarrow Sx = Sy)$$

- $x$  是素数 ( $x > 1$  且  $x$  没有除自身和 1 之外的因子)

$$\text{Prime}(x) : S0 < x \wedge \forall y. \forall z. (y < x \wedge z < x) \rightarrow \neg(y \times z = x)$$

- 哥德巴赫猜想 (任一大于 2 的偶数, 都可表示成两个素数之和)

$$\begin{aligned} \forall x. (SS0 < 2 \wedge (\exists y. 2 \times y = x)) \rightarrow \\ (\exists x_1. \exists x_2. \text{Prime}(x_1) \wedge \text{Prime}(x_2) \wedge x_1 + x_2 = x) \end{aligned}$$

有了坐标系, 辩证法进入了数学. 有了坐标系, 运动进入了数学.

—Engels

这里的“运动”, “坐标系”只是变量引入数学的结果. 自然, 这让我们探索的内容更加的丰富, 同时也更容易犯错了. 比如, 我们很多时候喜欢对变量做替换. 但是, 下面的代换显然是错的:

**Example 例子:**

“那曲市” := “中国最大的城市”

“那曲市” 有三个汉字

“中国最大的城市” 有三个汉字

因为“中国最大的城市”这个字符串有 7 个汉字构成. 这就警告我们要在变量代换的时候要格外小心. 那么, 变量有哪些类型呢?

在公式  $\forall x. P(x, y) \vee \exists y. R(x, y)$  中, 根据量词作用域的定义,  $\forall x$  的作用域是  $P(x, y)$ , 因此  $x$  受到了  $\forall x$  的约束, 从而把这个变元称作**约束变元**. 然而这个的  $y$  没有受到  $\forall x$  的约束, 所以称为**自由变元**.

**定义 2.4.8** (约束变元与自由变元). 给定公式  $\forall x. G$  和  $\exists x. G$ ,  $G$  为  $\forall x$  和  $\exists x$  的作用域, 那么  $G$  中的  $x$  出现的都是约束出现 (bounded occurrence), 称变元  $x$  则是约束变元 (bound variable),

$G$  中不同于  $x$  的其他变元出现的则是自由出现 (free occurrence), 称这些变元为自由变元 (free variable)

### Dialogue 对话

A: 这就像 C 里面判断一个变量是不是被定义过了一样! 如果被定义过了, 自然不能乱搞, 如果没有定义过, 改个和定义过的名字不一样的, 应该也是等价的。

B: 对, 名字是次要的, 真正的表达的东西才是核心啊!

可能由于重名的原因, 我们发现同样一个字母既可以表示自由变元, 又可以表示约束变元, 这样就很不利于我们问题的分析. 为此, 我们给出变元的代换规则, 可以让每一个符号各司其职。

**定理 2.4.1** (变元的代换规则). 对变元可以做如下的变量替换, 推理是有效的:

(1) 约束变元改名. 将量词作用域内与作用变元相同的约束变元用新的个体变元替换. 新的变元要与作用域的其他变量名符号不同。

(2) 自由变元带入. 将公式中的某个自由变元的每一处用新的个体变元或个体常量带入. 并且新变元不允许在公式中以任何约束形式出现。

上面的两条关系在带入之前和带入之后都没有改变原有命题之间的约束关系. 但是使用 (2) 之后, 公式的含义就发生了变化, 但是推理是成立的—比如选择常量带入进去, 那么命题就退化成了一个特化的情况了。

没有自由变量的公式有着很好的性质—他的真值表就可以确定了. 我们把这样的内容叫做闭式。

**定义 2.4.9** (闭式). 设  $G$  是任意的一个公式, 如果  $G$  中没有自由变元, 那么  $G$  被称为封闭的公式, 简称闭式。

**定理 2.4.2.** 闭式一定是命题。

在命题逻辑中, 我们运用了真值指派的方式来了解了这个命题的真值情况, 那么这里, 我们能不能也用类似的方法“解释”一个命题逻辑的含义呢?

## 2.5 尝试解释谓词公式

谓词公式的解释远远没有我们想象的那么简单! 在不同的数学结构中, 同一个谓词公式很可能有不同的结果!

### Example 例子:

考虑

$$\alpha : \forall x. (x \times x \neq 1 + 1),$$

$\alpha$  在数学结构  $\mathcal{U} = \mathbb{Q}$  中是真的, 但是在数学结构  $\mathcal{U} = \mathbb{R}$  中是假的。

从上面的例子, 我们可以看出来, 要想解释清楚一个谓词逻辑的公式, 我们需要考虑对象所处的数学空间来下结论. 也就是一个表达式的语义大致取决于:



- 对量词论域 (universe) 的解释, 限定个体范围;
- 对常数符号、函数符号、谓词符号的解释;
- 对自由变元的解释 (赋值函数  $s$ ).

也就是这种“解释”将公式映射到一个数学结构  $\mathcal{U}$  上, 这决定了该公式的语义.

什么是解释呢? 更正式一点, 我们认为一个公式的解释由下列四部分组成:

**定义 2.5.1.** 谓词逻辑中的谓词公式  $G$  的一个解释 (interpretation)  $I$  由以下的四部分构成:

- (1) 非空的个体域  $D$ ;
- (2)  $G$  中的每个常量符号, 指定  $D$  中的某个特定的元素.
- (3)  $G$  中的每个  $n$  元函数符号, 指定的是每一个元随意选择得到的所有可能的结果的集合对应到  $D$  的一个函数
- (4)  $G$  中的每个  $n$  元谓词符号, 指定每一个元随意选择得到的所有可能的结果的集合到  $\{0, 1\}$  的某个特定的谓词.

因此, 我们给出在一个特定的数学结构上“满足”的定义.

**定义 2.5.2**  $((\mathcal{U}, s) \models \alpha)$ .  $\mathcal{U}$  与  $s$  满足公式  $\alpha$ :

$$(\mathcal{U}, s) \models \alpha$$

- 将  $\alpha$  中的常数符号、函数符号、谓词符号按照结构  $\mathcal{U}$  进行解释,
- 将量词的论域限制在集合  $|\mathcal{U}|$  上,
- 将自由变元  $x$  解释为  $s(x)$ ,
- 这样就将公式  $\alpha$  翻译成了某个数学领域中的命题,
- 然后, 使用数学领域知识我们知道该命题成立

同样的, 我们也可以说明语义蕴含在谓词逻辑中的定义:

**定义 2.5.3** (语义蕴含 (Logically Imply)). 令  $\Sigma$  为一个公式集,  $\alpha$  为一个公式.

$\Sigma$  **语义蕴含**  $\alpha$ , 记为  $\Sigma \models \alpha$ , 如果每个满足  $\Sigma$  中所有公式的结构  $\mathcal{U}$  与赋值  $s$  都满足  $\alpha$ . 记作

$$\{\forall x. P(x)\} \models P(y)$$

当然, 我们当然希望为公式能够在等价的意义下能够完成推理. 所以我们需要首先研究什么样的公式在怎样的情形下是等价的.

**定义 2.5.4** (谓词公式等价的定律).



## 2.6 常见的证明方法

### 著名的推理规则：三段论

上一次我们说到了因为  $(p \wedge (p \rightarrow q)) \rightarrow q$  是永真式, 因此我们有了  $(p \wedge (p \rightarrow q)) \Rightarrow q$ . 三段论的内容被叫做 Modus Ponens.

Modus Ponens 来源于古拉丁文, 翻译成英语就是 method of putting by placing. 也就是说, 在推理的过程中, 如果  $p$  是真的,  $p \rightarrow q$ , 那么  $q$  是可以被接受的.

为什么呢? 因为如果有含蕴含词的永真式  $(A_1 \wedge A_2 \wedge \cdots \wedge A_n) \rightarrow B$ , 利用推理规则, 我们就能从前提条件  $A_1 \wedge A_2 \wedge \cdots \wedge A_n$  推出  $B$ , 以及  $A_1 \rightarrow (A_2 \rightarrow \cdots \rightarrow (A_n \rightarrow B))$

#### Bonus 思考题

蕴含记号支持结合律吗? 支持交换律吗?

### 当且仅当的证明

在中学, 我们学习了充分条件, 必要条件, 以及四种命题.

#### Dialogue 对话

A: 现在我知道如何否定命题“若  $p$  则  $q$ ”了!

B: 很好! 注意一下四种命题. 为什么命题和它的逆否命题真假性是一致的?

先对充要条件做描述: 充要条件是  $(P \rightarrow Q) \wedge (Q \rightarrow P) \rightarrow (P \leftrightarrow Q)$

比如来看下面的问题:

**问题 2.6.1.** 求证:  $a^3 = b^3$  当且仅当  $a = b$ .

充分条件很容易说明. 但是必要条件如何说明呢? 我们高中知道命题和逆否命题真假一致, 能从命题逻辑的角度说明吗?

其实很容易.  $(P \rightarrow Q) \leftrightarrow (\neg Q \rightarrow \neg P)$  是永真式就行了. 我们就有更加公理化的方法完成了证明.

**问题 2.6.2.** 为什么证明  $a^2 = b^2$  当且仅当  $a = b$  时不行的?

我们发现结论是析取式子 (或), 那么我们应该如何证明? 我们考察永真式  $(P \wedge Q) \leftrightarrow (\neg P \wedge Q); (P \wedge Q), \neg P \Rightarrow Q$ . 为之构造一个合理的语言描述, 然后翻译成符号语言看上去是很好的一种方法.

### 反证法

Hardy 说证明  $\sqrt{2}$  是无理数是最漂亮的证明之一. 其中用了反证法. 不妨自己写出一个证明.

**问题 2.6.3.** 证明  $\sqrt{2}$  是无理数.

**Dialogue 对话**

A: 我感觉好难...

B: 那是因为高中没有学过数论! 去了解一些基本的整数的性质吧! 很多时候感觉困难只是因为前置知识没有做好. 补充上去就行了.

反证法之所以能成立, 是因为  $\neg A \Rightarrow C$ ,  $\neg A \Rightarrow \neg C$ , 必然有  $C \wedge \neg C \Rightarrow \text{False}$ . 根据上述的内容,  $(p \rightarrow q) \wedge (\neg q) \rightarrow \neg p$ , 所以化简得到  $\neg\neg A = A$ .

**Bonus 思考题**

我们应该如何否定含有量词的命题? 留到下一章节.

**基于数学对象结构的证明方法: 数学归纳法**

在分牌的时候, 我们可以使用这个小魔术:

- (1) 将一副纸牌按照红黑相间的模式排好;
- (2) 按照传统方式洗一次牌, 分牌时两叠牌显出的两张颜色互异;
- (3) 你能做到将牌放在背后 (自己看不见), 却能保证每次抽出两张牌, 始终是一红一黑吗?

这个过程的证明可以使用数学归纳法描述. 回忆一下数学归纳法的常见过程. (1) 归纳奠基; (2) 归纳假设; (3) 结论.

我们现在进行一个灵魂拷问:

**Bonus 思考题**

为什么数学归纳法在逻辑上是正确的?

事实上, 其合理性来自证明对象结构的定义.

**定义 2.6.1** (Peano 公理). 自然数的 Peano 公理有如下几条:

1. 0 是自然数;
2. 如果  $n$  是自然数, 则它的后继  $S_n$  也是自然数;
3. 0 不是任何自然数的后继;
4. 两个自然数相等当且仅当它们的后继相等;

对于自然数的归纳正确性, 我们还需要良序原理 (well-ordering principle). 可能需要很多的背景知识, 在这里仅仅给出基本的定义.

**定理 2.6.1** (第一数学归纳法 (The First Mathematical Induction)). 设  $P(n)$  是关于自然数的一个性质. 如果

1.  $P(0)$  成立;
2. 对任意自然数  $n$ , 如果  $P(n)$  成立, 则  $P(n+1)$  成立.

那么,  $P(n)$  对所有自然数  $n$  都成立.

**定义 2.6.2** (良序原理 (The Well-Ordering Principle)). 自然数集的任意非空子集都有一个最小元.

除了第一类数学归纳法, 其实还有更加强大的第二类数学归纳法. 并且两者之间是等价的.

**定理 2.6.2** (第二数学归纳法 (The Second Mathematical Induction)). 设  $Q(n)$  是关于自然数的一个性质. 如果

1.  $Q(0)$  成立;
2. 对任意自然数  $n$ , 如果  $Q(0), Q(1), \dots, Q(n)$  都成立, 则  $Q(n+1)$  成立.

那么,  $Q(n)$  对所有自然数  $n$  都成立.

他们的等价的推导, 用到了后面的量词相关的内容, 所以现在先暂时跳过. 不过, 第二类数学归纳法确实更加强大大一点. 比如可以证明任何一个整数可以分解成若干个素数的乘积.

#### Dialogue 对话

A: 数学归纳法这么好, 感觉什么整数的问题都能证明啊!

B: 其实没有你想象的那么简单. 《具体数学》上有关于它的论述, 但是我一时间找不到了.

最后, 我们用一个例子来结束这一节. 这是一个很烧脑的问题, 据说是由 Terry Tao 放在他的个人主页上的一道题.

**问题 2.6.4.** Of the 1000 islanders, it turns out that **100 of them have blue eyes** and **900 of them have brown eyes**, although the islanders are not initially aware of these statistics (each of them can of course only see 999 of the 1000 tribespeople).

One day, a **blue-eyed foreigner** visits to the island and wins the complete trust of the tribe.

One evening, he addresses the entire tribe to thank them for their hospitality.

However, not knowing the customs, the foreigner makes the mistake of mentioning eye color in his address, remarking “**how unusual it is to see another blue-eyed person like myself in this region of the world**”.

**What effect, if anything, does this *faux pas* (失礼) have on the tribe?**

答案比较令人出乎意料:

**定理 2.6.3.** Suppose that the tribe had  $n > 0$  blue-eyed people.

Then  $n$  days after the traveller's address, all  $n$  blue-eyed people commit suicide.

证明. 考虑采用数学归纳法:

- 基础步骤:  $n = 1$ .

这个**唯一的蓝眼人**的内心独白: “你直接念我身份证吧”

- 归纳假设: 有  $n$  个蓝眼人时, 前  $n - 1$  天无人自杀, 第  $n$  天集体自杀.
- 归纳步骤: 考虑恰有  $n + 1$  个蓝眼人的情况.  
每个蓝眼人都如此推理: 我看到了  $n$  个蓝眼人, 他们应该在第  $n$  天集体自杀.  
但是, 每个蓝眼人都在等其它  $n$  个蓝眼人自杀, 因此, 第  $n$  天无人自杀.  
每个蓝眼人继续推理: 一定不止  $n$  个蓝眼人, 但是我看到的其余人都不是蓝眼.  
所以, “小丑竟是我自己”.

□

仔细考察这个例子, 考虑  $n = 1, n = 2$  的简单情况, 出现了类似“我知道你知道我知道...”的思维递归情形. 数学归纳法帮我们挑选了其中的相邻两层, 让我们更加清楚的直面问题.

#### Idea 启示

考虑递归问题的时候, 在非边界条件的情形下, 只用考虑连续的两层就可以了. 关注状态变化.

#### Bonus 思考题

刚刚的魔术, 为什么不是关于自然数的, 也能用数学归纳法证明?

### 一个特别的证明: 鸽巢原理

**定理 2.6.4.** 如果  $m$  只对象放入  $n$  个容器, 其中  $m > n$ , 则至少有一个容器中对象个数大于 1.

#### Bonus 思考题

为什么鸽巢原理叫做“原理”而不是“公理”, “定理”?

## 2.7 重新审视算法的正确性: 从基本结构开始

**问题 2.7.1.** 在本节开始之前, 不妨先阅读 David Harel 写的书本《Algorithmics: The spirits of computing》, 之后与本节的内容进行对比.

我们在之前知道, 计算机解题的正确性在于正确算法的实现. 那么算法的正确性来源于什么呢? 不同于人类指令的有歧义和模糊, 机器语言应该做到的是有明确的“基本指令”, 与“控制结构”.

(对于初学者而言,) 精确的定义毫无意义.

—蒋炎岩

无论如何, 先忘掉这些抽象的名词吧. 基本指令就可以想象成一行代码是如何进行的, 它是如何操作了计算机的内部的. 控制结构是一些描述那个当前的代码执行如何执行下一条指令的一系列规约. 也就是描述了基本算法是什么, 以及如何被实施的.

生活中很多的事情都可以用一些控制结构完成. 我们来举几个例子:

## 顺序结构

### Example 例子:

吃一只蟹黄包的算法.

如何确保我们的过程无误? 其实我们可以在执行完之后进行监视. 每一个 guard 需要看一看前面的内容是不是已经完成了即可.

如果我们不打算只吃一只, 那么我们如何控制数量呢?

## 循环

一个很有趣的问题是: 如何确定循环的过程是正确的? 我们可以发现, 虽然循环的变量在变, 但是我们总是希望维系一个命题的值是真的. 由此我们引入“循环不变式”.

**定义 2.7.1** (循环不变式). 循环不变量是一个逻辑表达式, 在循环执行过程终止之前它应该始终为真.

例如, 我们使用累乘的方法去循环的时候, 循环不变式可以是“存放中间结果的量的值  $= a \times$  循环变量当前的值”.

类似于数学归纳法, 我们可以利用循环不变式证明循环算法证明的一个基本的方法:

**定理 2.7.1** (证明循环不变式的基本方法). (1) Initialization. It is true prior to the first iteration of the loop.

(2) Maintenance. If it is true before an iteration of the loop, it remains true before next iteration.

(3) Termination. The loop terminates, and when it terminates, the invariant – usually along with the reason that the loop terminated – gives us a useful property that helps show that the algorithm is correct.

下面我们为上述的两种吃蟹黄汤包的策略选定一个循环不变式. 对于策略 I, 我们有在循环终止之前, 我还是没有饱的; 对于策略 II, 我们的循环不变式是计数器的值 = 存放临时变量的值 + 1

我们发现不变量对于问题求解有很大的价值. 比如下面的这个问题:

**问题 2.7.2.** 在世界杯足球赛的第二阶段 (小组赛结束后) 进行单循环比赛, 每场必须分出胜负, 赢者进下一轮, 输者回家. 16 个队参加, 赛多少场决出冠军?

答案是 15 场. 因为每一次循环满足不变量: 这一次的值 = 上一次的值 - 1. 由此就可以归纳了.

与之类似的问题,

**问题 2.7.3.** A rectangular chocolate bar is divided into squares by horizontal and vertical grooves, in the usual way. It is to be cut into individual squares. A cut is made by choosing a piece and cutting along one of its groove. (Thus each cut splits one piece into two pieces) How many cuts are needed to completely cut the chocolate into all its squares?

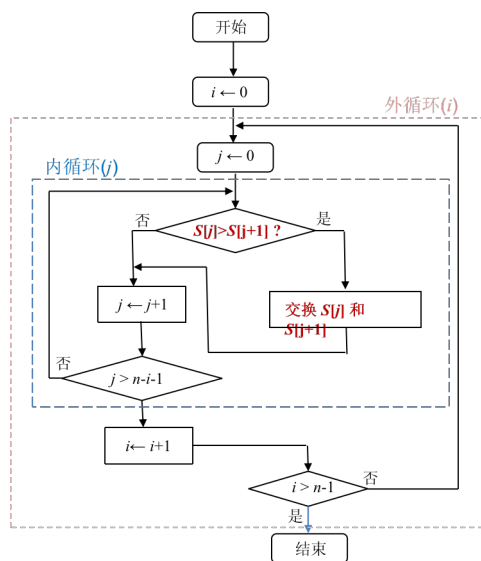


图 2.1: 冒泡排序的一个实现方法 (流程图)

解答的方法也是和前面的完全一样, 他们是遵循同一个“差 1”的不变量的. 这就是不变量对于问题求解的价值.

### 算法的条件分支

基本上, 算法用于决策的判断条件是一个命题逻辑, 根据其真假可以判别. 这时候我们在前面的一些内容就派上用场了.

上面我们已经对每一个部分所有的语义简单进行了了解, 下面, 我们把它们组合, 嵌套起来, 看一看他们能够解决什么样的问题.

### 组合起来: 列举与嵌套 (nesting)

从一个很常见的问题开始: 冒泡排序.

**问题 2.7.4.** 用上面的语句 (可以画流程图) 及其列举与嵌套, 完成排序问题:

- (1) 输入: 长度为  $n$  的自然数序列  $S$  (假设其中没有重复元素).
- (2) 输出: 序列  $S'$ , 其元素与  $S$  完全一样, 但已从小到大排好序.

一个可能的设计方法如图2.1所示. 我们可以模拟执行它.

我们可以把内循环同样抽象成一个问题, 将一个 (子) 序列中的最大元素放到该子序列位置地址最大的地方 (可以认为最后方). 那么对于它, 我们应该如何定义循环不变量?

每次  $i$  循环开始前, 序列  $S$  中地址最高的  $i$  个位置 (除  $i = 0$  外, 即  $S[n-i]$  到  $S[n-1]$ ) 包含  $S$  中前  $i$  个最大的元素, 且已从小到大排好序; 序列中其它位置上的元素即  $S$  中其它  $n-i$  个元素.

下面按照如下的三个方法归纳:

- 当  $i = 0$ , 显然;

- 如果当  $i = k > 0$  时上述命题成立, 即  $k$  循环开始前,  $S$  中最大的  $k$  个元素已被置换到  $S[n-k]$  到  $S[n-1]$ ; 在  $k$  循环中, 指针  $j$  从位置 0 扫描到位置  $n-k-1$ , 并将此范围内最大元素置换到  $S[n-k-1]$ . 则当  $i = k+1$  时,  $S$  中最大的  $k+1$  个元素被排好序, 放置在  $S[n-(k+1)]$  到  $S[n-1]$ ;
- 当  $i = n$ , 算法终止, 即  $n$  循环 (未执行) 前,  $n$  个  $S$  中的元素排好序, 并放置在  $S[0]$  到  $S[n-1]$  的位置上.

### Bonus 思考题

在第一章里面我么说 TRM 里面有一条跳转指令, 为什么现在我们没有详细介绍 goto 指令, 并且绝大多数语言 (除了 C,C++) 都没有了 goto 呢?

### Passage 文章

#### Go To Statement Considered Harmful(Cropped)

E.W. Dijkstra

from Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

.....

The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely

complicated affair to define all those points of progress where, say,  $n$  equals the number of persons in the room minus one!

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

.....

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluosness of the go to statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

## 2.8 继续理解子过程和递归

### 子过程

在程序变的更加复杂的时候, 程序就会变得像面条一样难以驾驭. 这时候, 如果工程文件十分的大, 那么我们很可能会在这种情况下迷失. 比如写的一个 C 文件, 如果把所有的函数展开, 得到的一个难以阅读的文件. 可以用下面的命令得到一份.

```
1 gcc your_file_name.c -S
```

之后文件夹下面就有一些.s 文件了. 这是把 C 语言展开为了最基本的指令序列的结果. 可见, 当问题规模变得复杂的时候, 我们必须让复用重复的部分更加容易.

比如, 我们要对一个句子里面含有 “money” 的词语计数, 我们可以有如下的算法.

#### Bonus 思考题

所以, 用子过程有什么好处?

事实上, 定义了子过程, 我们很多时候可以自己调用自己. 这对我们人类编写程序的时候是大有裨益的.

### 递归

在第一章我们用小纸条和老师以及作业本说明了什么是递归. 在这一节我们提供更多的例子.

**问题 2.8.1** (Hanoi 塔问题).



## 参考文献

- [1] MJD (<https://math.stackexchange.com/users/25554/mjd>). For cnf and dnf why do we look at the interpretations that make the formula false and true respectively? Mathematics Stack Exchange. URL:<https://math.stackexchange.com/q/4227598> (version: 2021-08-18).