计算机问题求解 – 论题1-7

- 不同的程序设计方法

2015年11月5日

检查

Since this information is unavailable to the compiler, the machine code it generates for the expression $a[\,j\,]$ just takes the address of a (that is, the first cell of the array), adds j to it, and retrieves the value stored in that address. (In fact, the same expression can also be written in C as $*(a + j\,)$, which more closely reflects its implementation.) In contrast, when the PL/I compiler encounters the corresponding expression, it will also generate code to check that the index is indeed within the legal bounds.

问题0

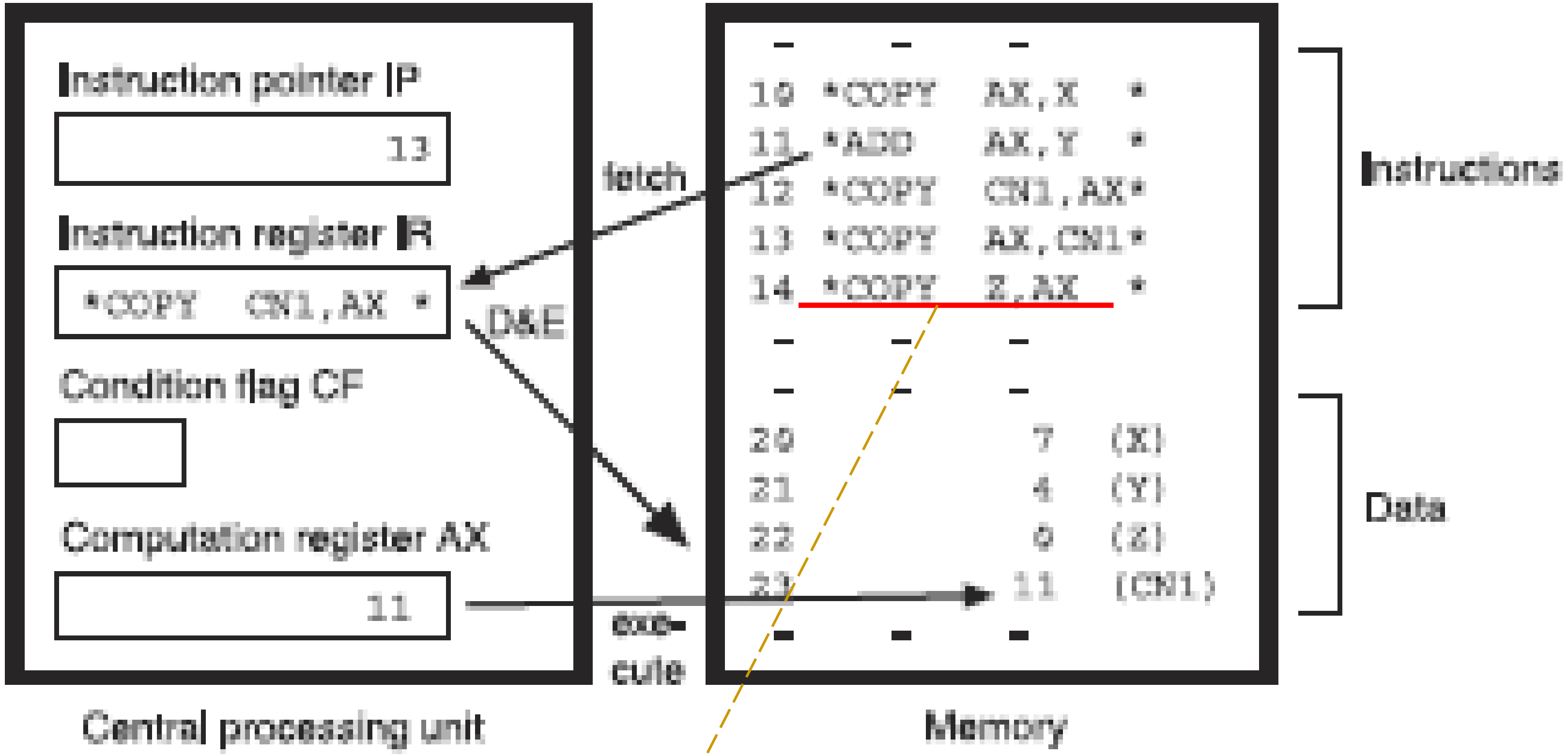
- 计算机真正能够“识别”和“执行”的程序是什么？
  - 机器代码，hard wired primitive instructions and it combination instructions

- 我们能否编写这样的程序？

问题1

- 我们常说“高级程序设计”，有没有与之对应的“低级”程序设计?

- 如果有，两者是什么区别?
  - 抽象 VS 具体
  - 巨大的“细节”鸿沟



每个汇编语言的语句（大致）对应于一条机器能够“理解”并“执行”的机器语言的“指令”。

问题2:

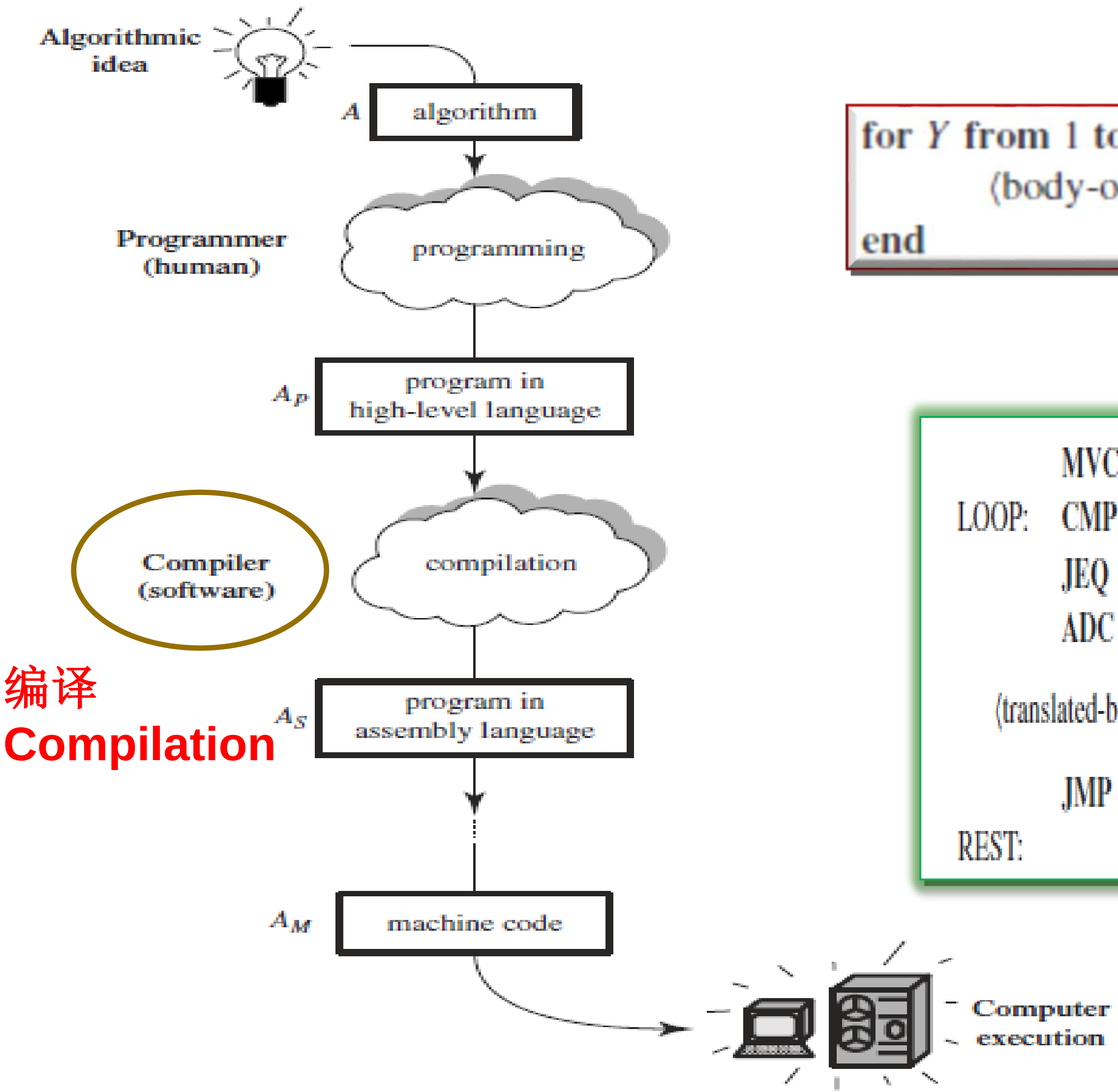
程序设计语言的“implementation(实现)”是什么意思？

问题2

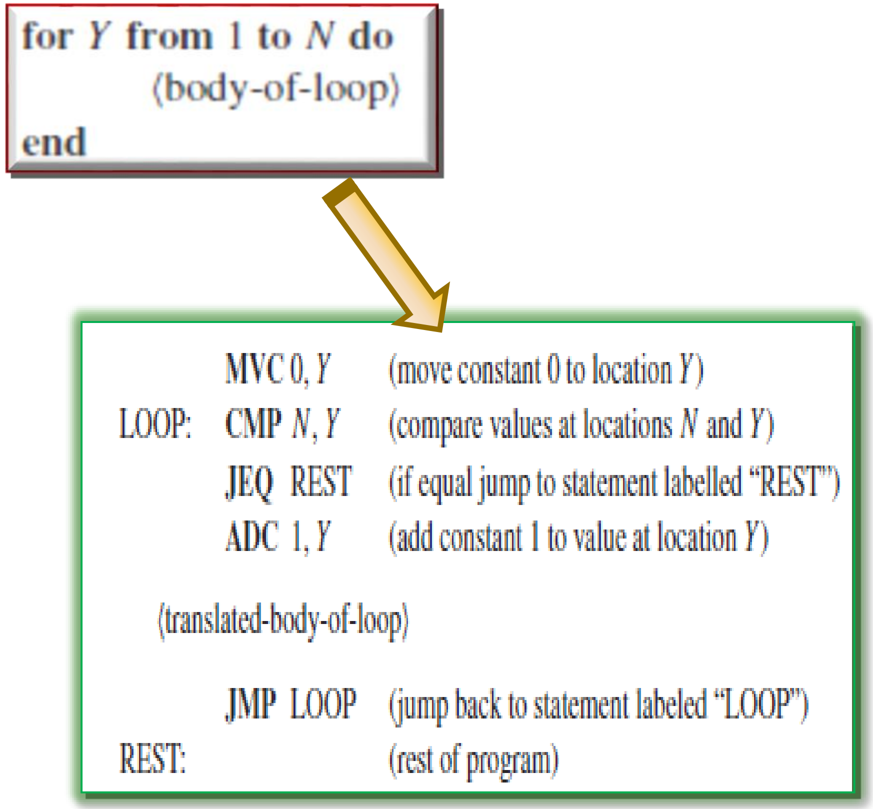
任意给出一个用该语言书写的合法程序，将其转换成可以在某台机器上可以运行的机器语言程序

问题3：

我们通常用什么手段来“实现”一个程序设计语言？



编译
Compilation

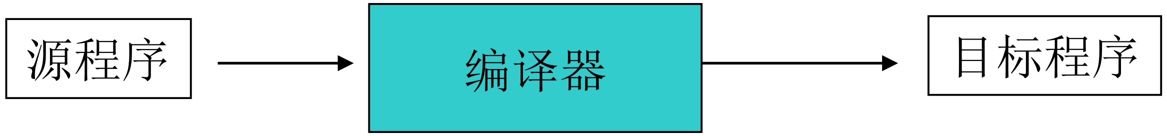


```
for Y from 1 to N do
        (body-of-loop)
end
```

```
        MVC 0, Y     (move constant 0 to location Y)
LOOP:   CMP N, Y     (compare values at locations N and Y)
        JEQ  REST    (if equal jump to statement labelled "REST")
        ADC  1, Y    (add constant 1 to value at location Y)

   (translated-body-of-loop)

        JMP  LOOP    (jump back to statement labeled "LOOP")
REST:                (rest of program)
```

什么是编译器

- 一个编译器就是一个程序
- Input：以某一种语言（源语言）编写的程序，
- Output：与input等价的、用另一种语言（目标语言）编写的程序。

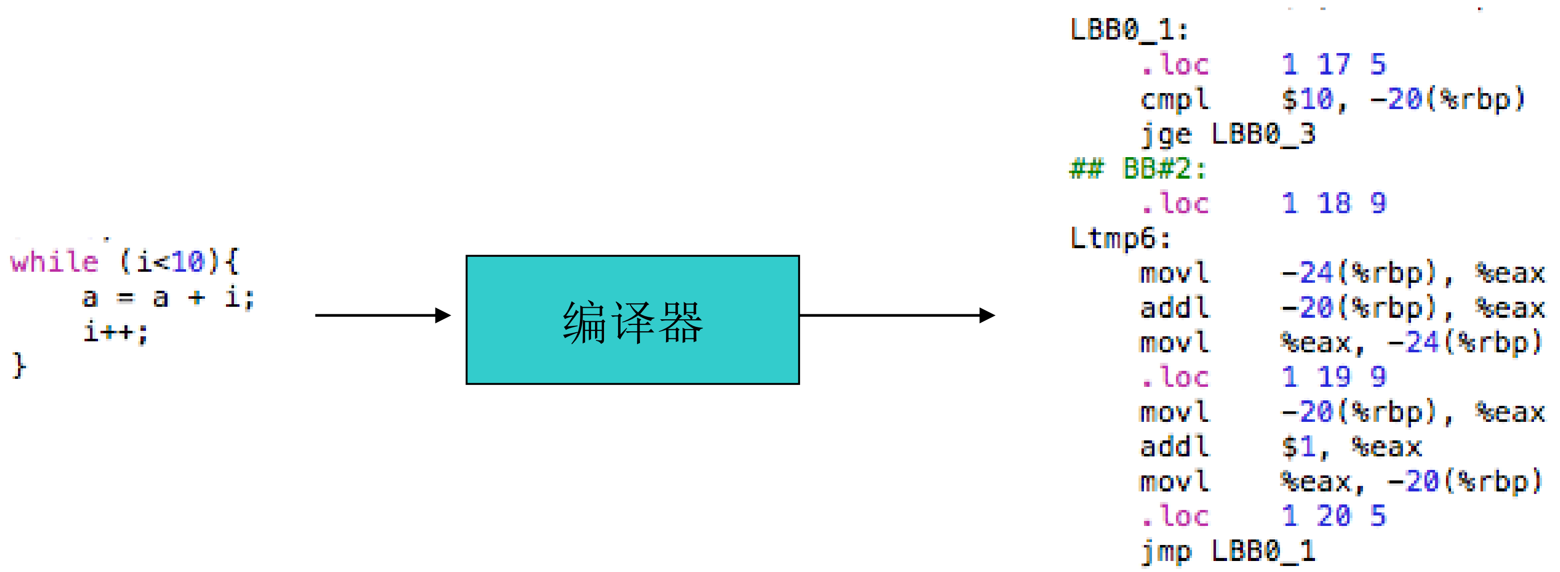


- 狭义： 程序设计语言 → 机器代码
- 广义：程序变换 C++ → C →汇编

  Pascal → C

示例

- 程序的运行过程
  - 源程序
  - 汇编代码
  - 机器代码



```
while (i<10){
     a = a + i;
     i++;
}
```

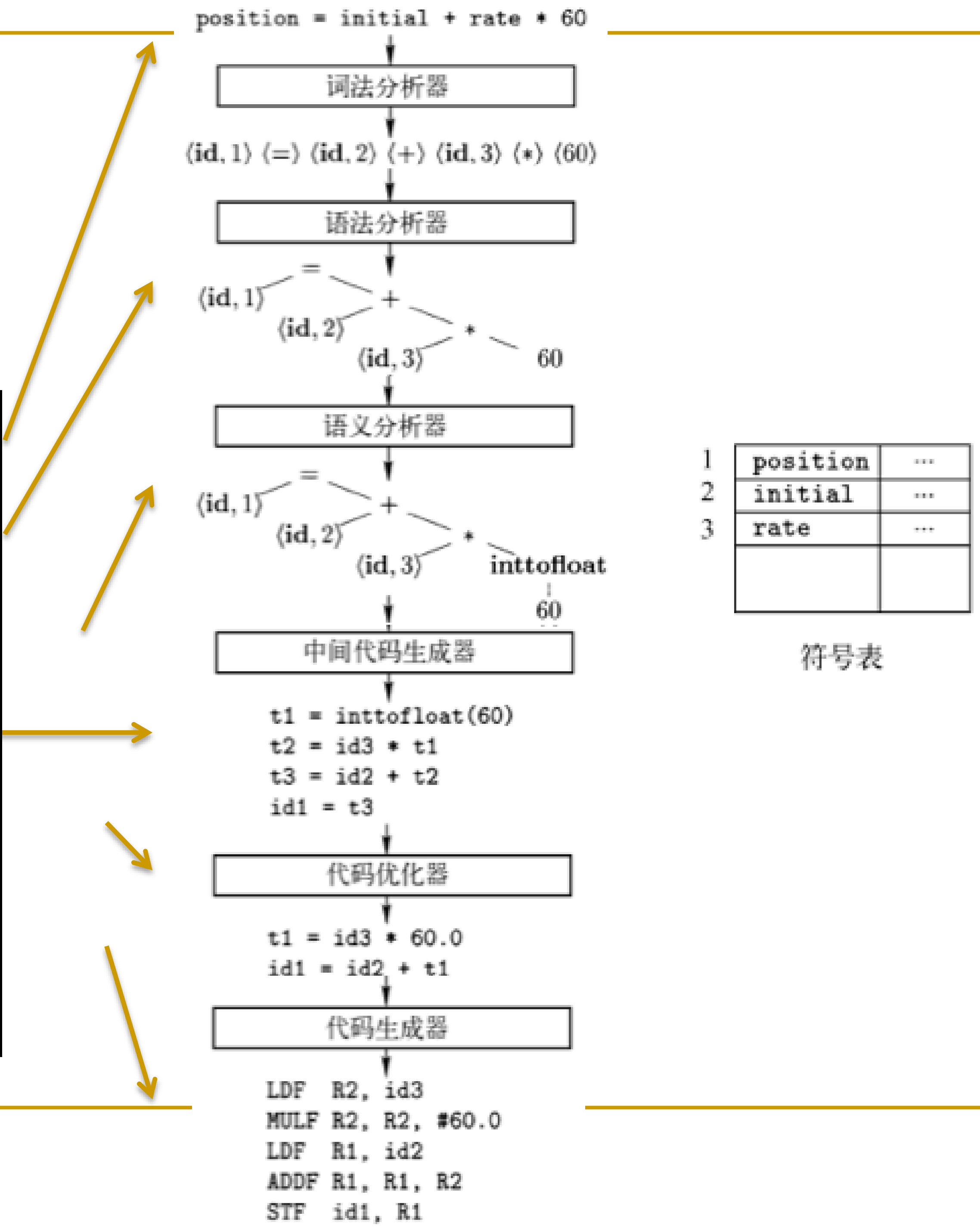
```
LBB0_1:
     .loc      1 17 5
     cmpl      $10, -20(%rbp)
     jge LBB0_3
## BB#2:
     .loc      1 18 9
Ltmp6:
     movl      -24(%rbp), %eax
     addl      -20(%rbp), %eax
     movl      %eax, -24(%rbp)
     .loc      1 19 9
     movl      -20(%rbp), %eax
     addl      $1, %eax
     movl      %eax, -20(%rbp)
     .loc      1 20 5
     jmp LBB0_1
```

编译器工作流程

Void Compiler(){

[redacted]

}



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

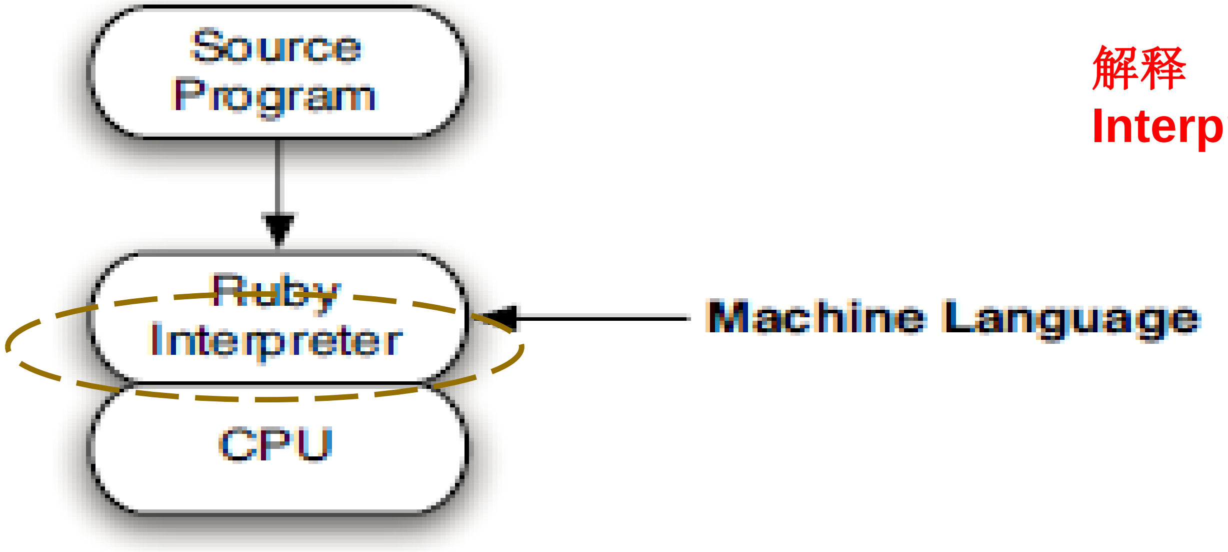
```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

1	position	···
2	initial	···
3	rate	···

问题4：

你有没有想到过这样的问题：C语言的编译程序用什么语言来写？可以就用C语言写吗？

解释
Interpreting



- it is usually easier to write a “quick-and-dirty,” but reasonably useful, interpreter, than it is to write a reasonable compiler;
- interpreter-driven execution yields a more traceable account of what is happening, especially when working interactively with the computer through a terminal with a display screen.

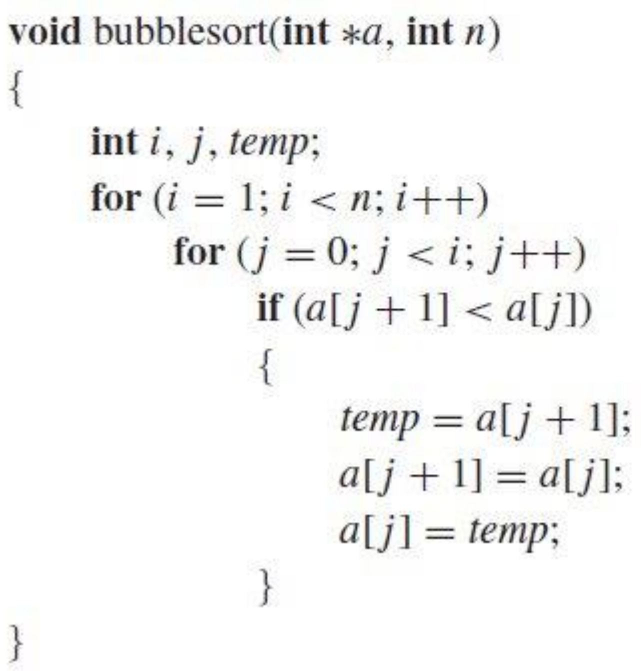
问题5：

从程序执行的角度来看，编译与解释两种方式最突出的差别是什么？

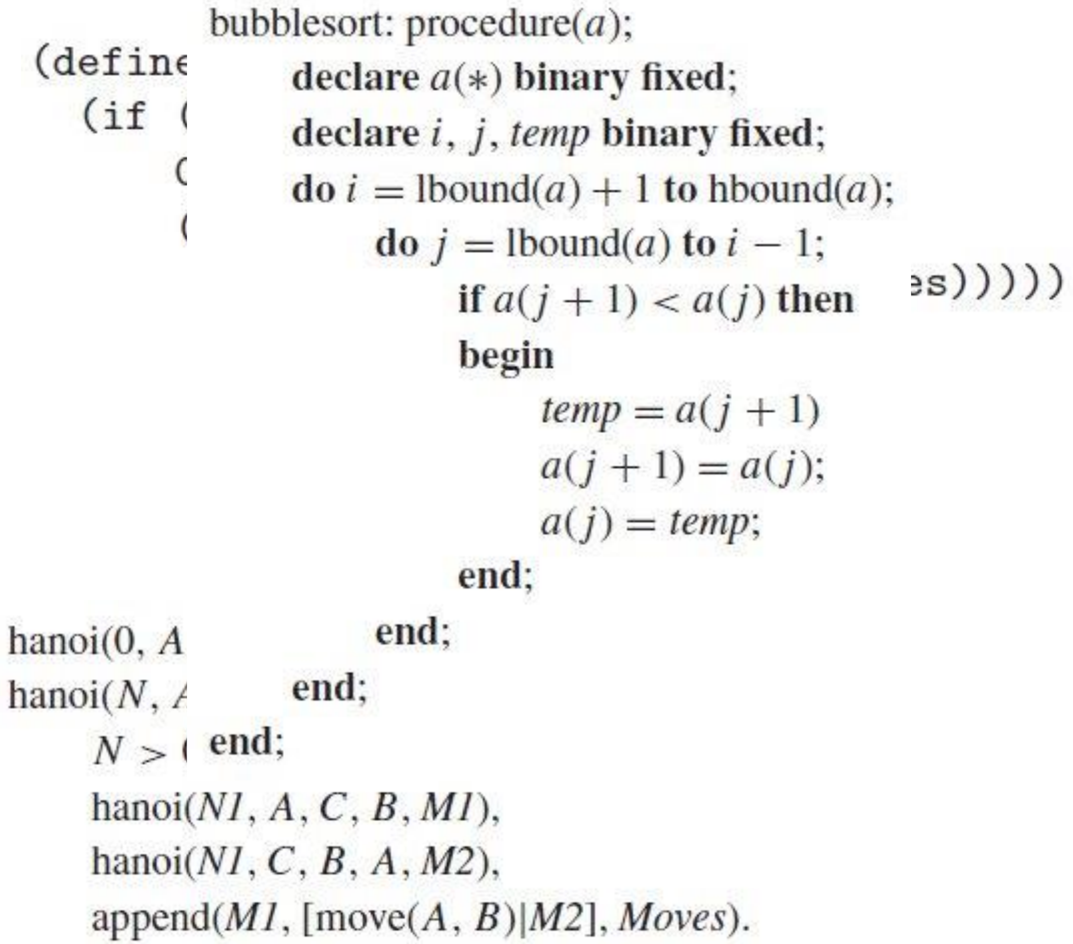
问题6：为什么没有算法的“世界语”？

- 同一个算法，因机器不同而采用全然不同的设计
  - 并行计算机==》并发程序设计
- 同一个算法，因目标不同而采用全然不同的设
  - As the power of computers is harnessed in more and more application areas, programmers encounter more and more types of abstractions
  - Each application area has its own set of concepts that need to be incorporated into computer programs. This can be done in many ways, one of which is the creation of a variety of special-purpose programming languages, which embody the concepts of specific areas.

语言不同，最根本的不同在“风格”的不同！



```
void bubblesort(int *a, int n)
{
    int i, j, temp;
    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++)
            if (a[j + 1] < a[j])
            {
                temp = a[j + 1];
                a[j + 1] = a[j];
                a[j] = temp;
            }
}
```



```
(define
  (if (
      (
      (
                                         es)))))
```

```
bubblesort: procedure(a);
    declare a(*) binary fixed;
    declare i, j, temp binary fixed;
    do i = lbound(a) + 1 to hbound(a);
        do j = lbound(a) to i - 1;
            if a(j + 1) < a(j) then
            begin
                temp = a(j + 1)
                a(j + 1) = a(j);
                a(j) = temp;
            end;
        end;
    end;
end;
```

```
hanoi(0, A
hanoi(N, A
    N > (
    hanoi(N1, A, C, B, M1),
    hanoi(N1, C, B, A, M2),
    append(M1, [move(A, B)|M2], Moves).
```

问题7:

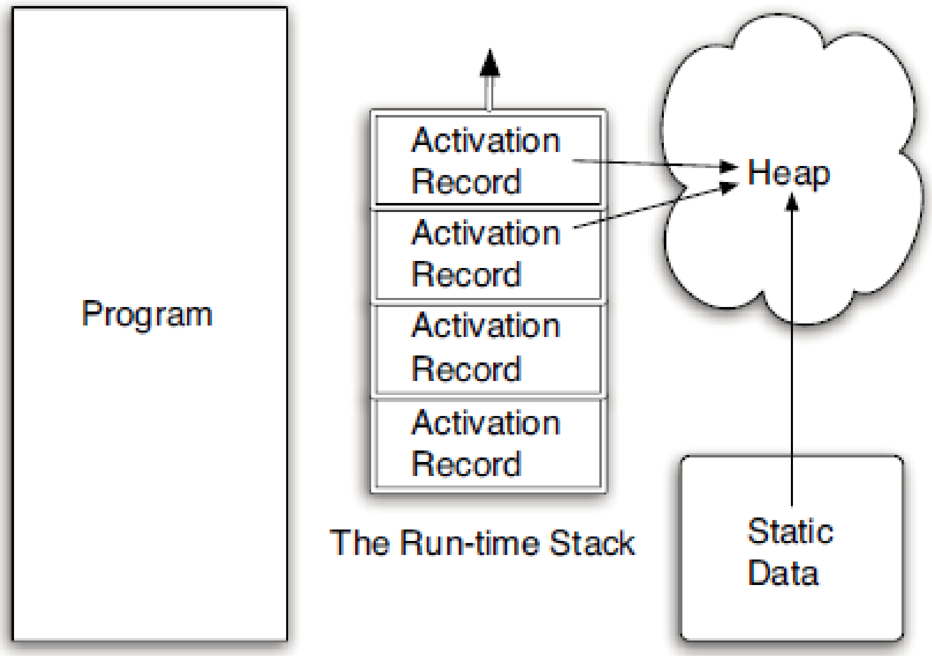
什么是语言的风格？不同风格的语言给我们带来什么不同？

A programming paradigm is a way of thinking about the computer, around which other abstractions are built.

问题7:

什么是“programming paradigm”?

机器的抽象-命令式编程

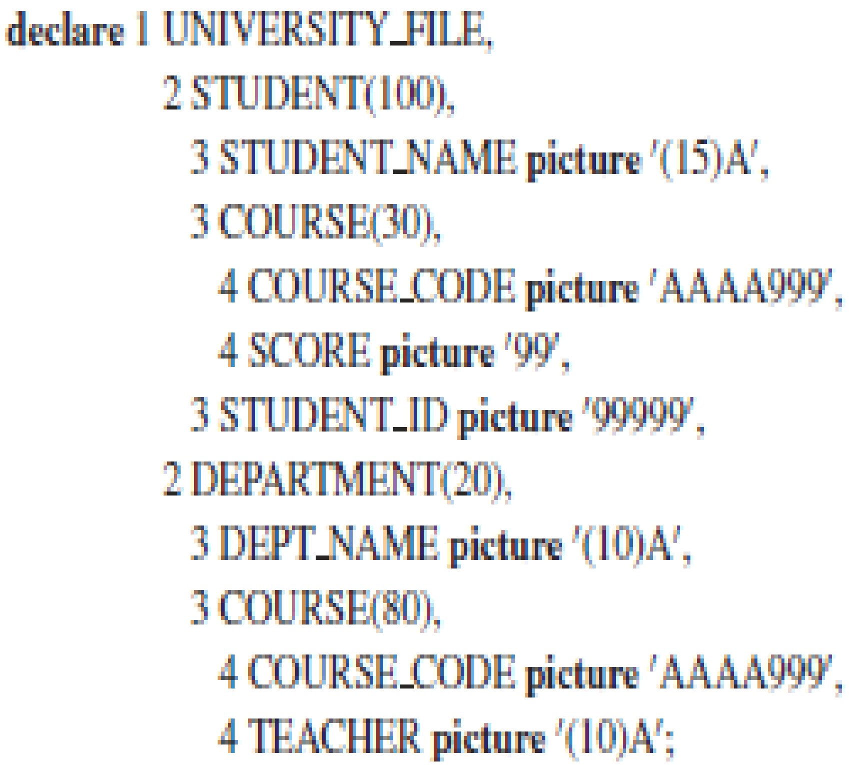


Conceptual View of the Imperative Model

von Neuman 架构：存储程序，顺序执行

We think of the computer as a collection of memory cells, organized into many types of data structures, such as arrays, lists, and stacks. Programs in this approach are concerned with building, traversing, and modifying these data structures, by reading and modifying the values stored in memory.

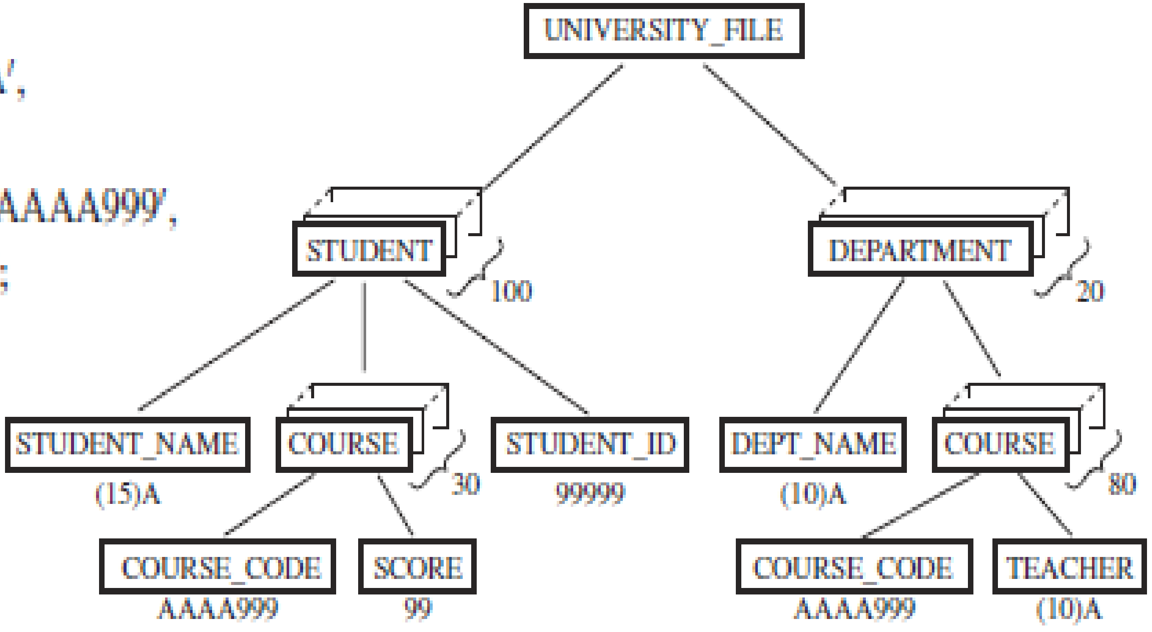
复杂的数据结构定义设施



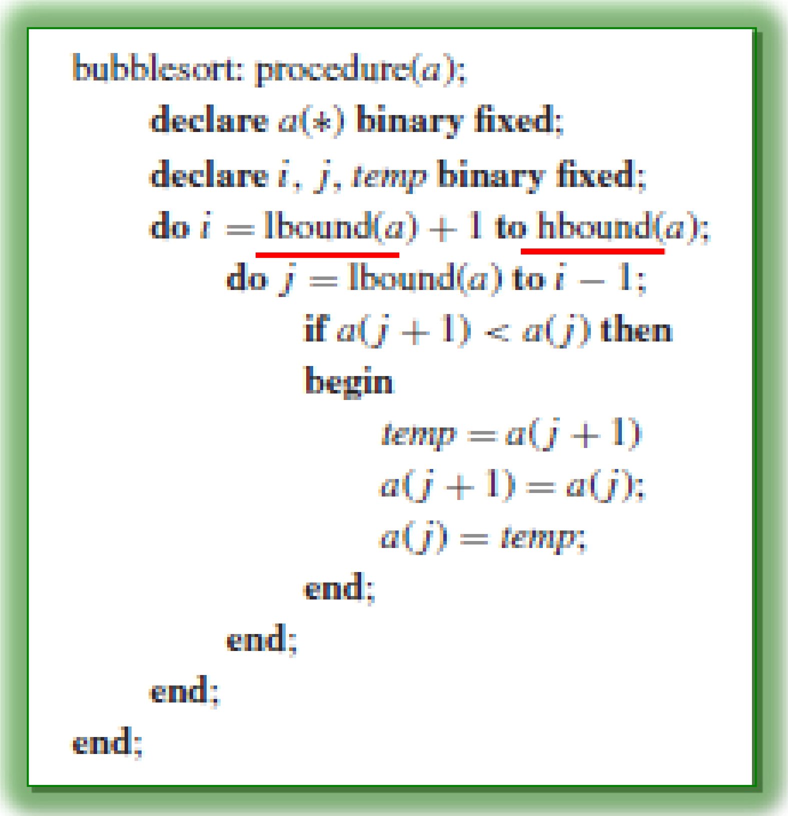
```
declare 1 UNIVERSITY_FILE,
          2 STUDENT(100),
            3 STUDENT_NAME picture '(15)A',
            3 COURSE(30),
              4 COURSE_CODE picture 'AAAA999',
              4 SCORE picture '99',
            3 STUDENT_ID picture '99999',
          2 DEPARTMENT(20),
            3 DEPT_NAME picture '(10)A',
            3 COURSE(80),
              4 COURSE_CODE picture 'AAAA999',
              4 TEACHER picture '(10)A';
```

为程序员考虑“过于”周到使得机器实现比较困难。

PL/1的例子

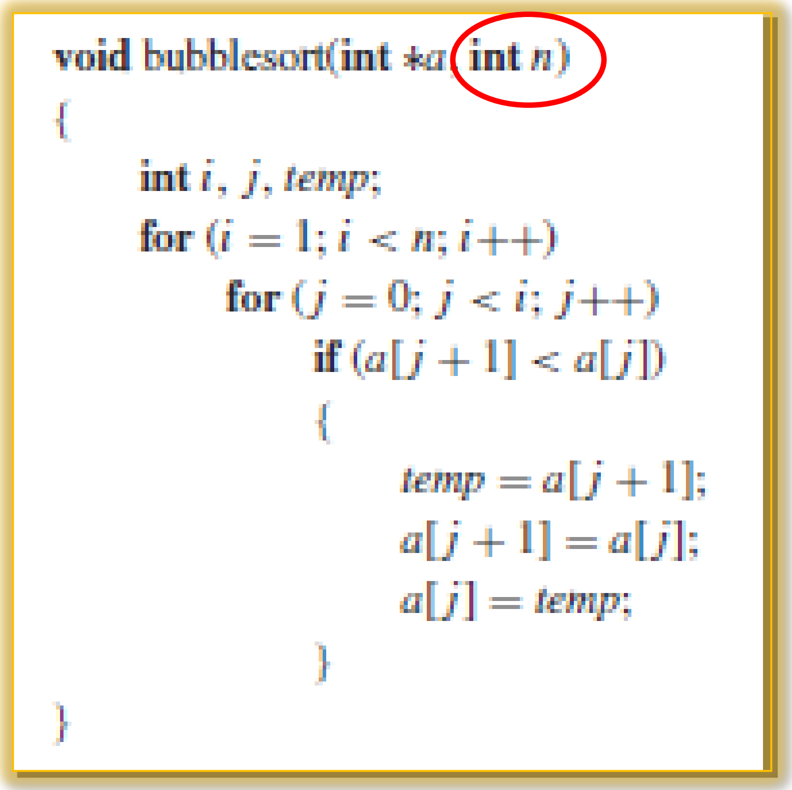


C语言的灵活性及其代价



```
bubblesort: procedure(a);
    declare a(*) binary fixed;
    declare i, j, temp binary fixed;
    do i = lbound(a) + 1 to hbound(a);
        do j = lbound(a) to i − 1;
            if a(j + 1) < a(j) then
            begin
                temp = a(j + 1)
                a(j + 1) = a(j);
                a(j) = temp;
            end;
        end;
    end;
end;
```

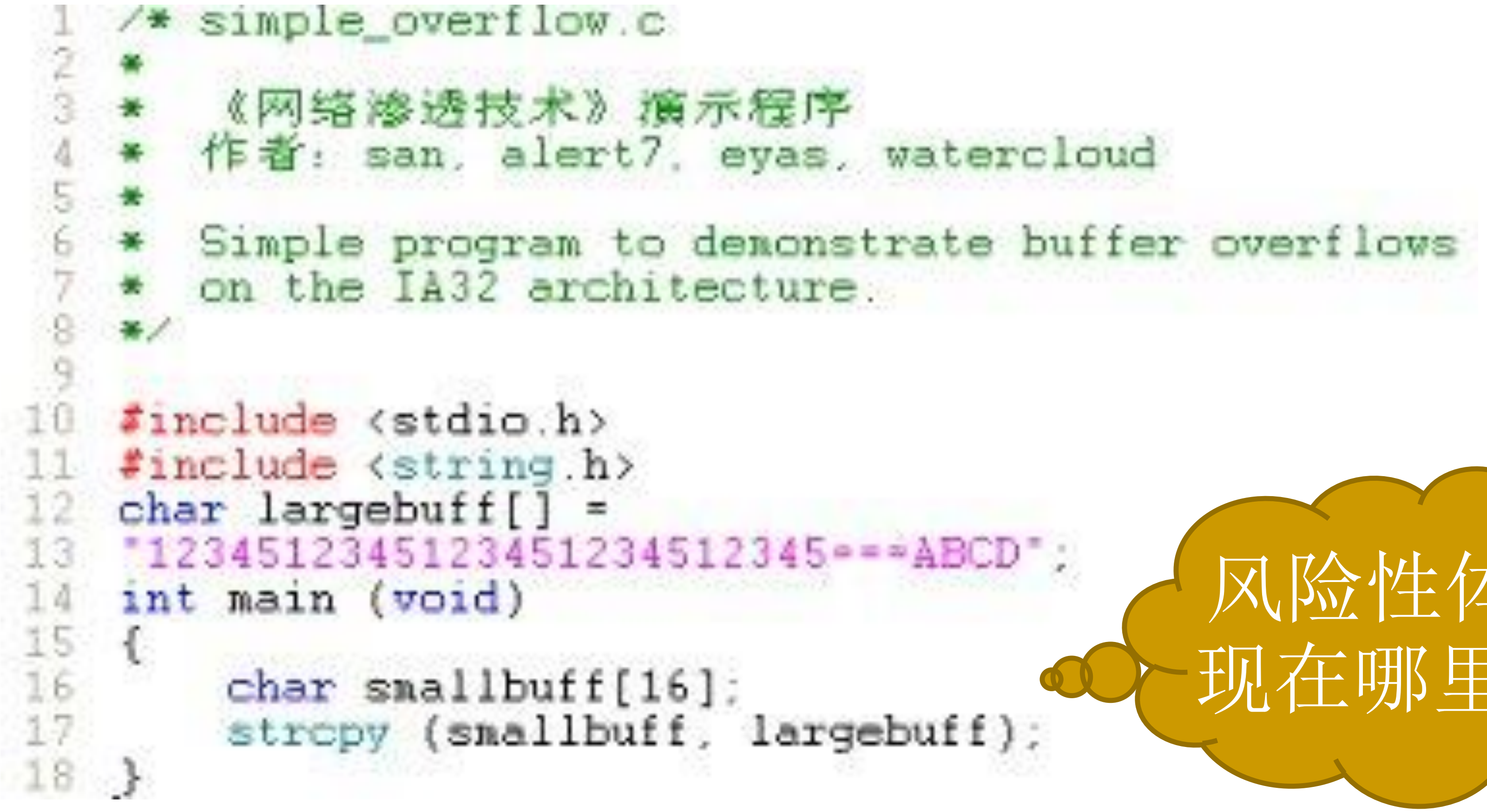
PL/1的安全设施



```
void bubblesort(int *a, int n)
{
    int i, j, temp;
    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++)
            if (a[j + 1] < a[j])
            {
                temp = a[j + 1];
                a[j + 1] = a[j];
                a[j] = temp;
            }
}
```

在C中由程序员控制

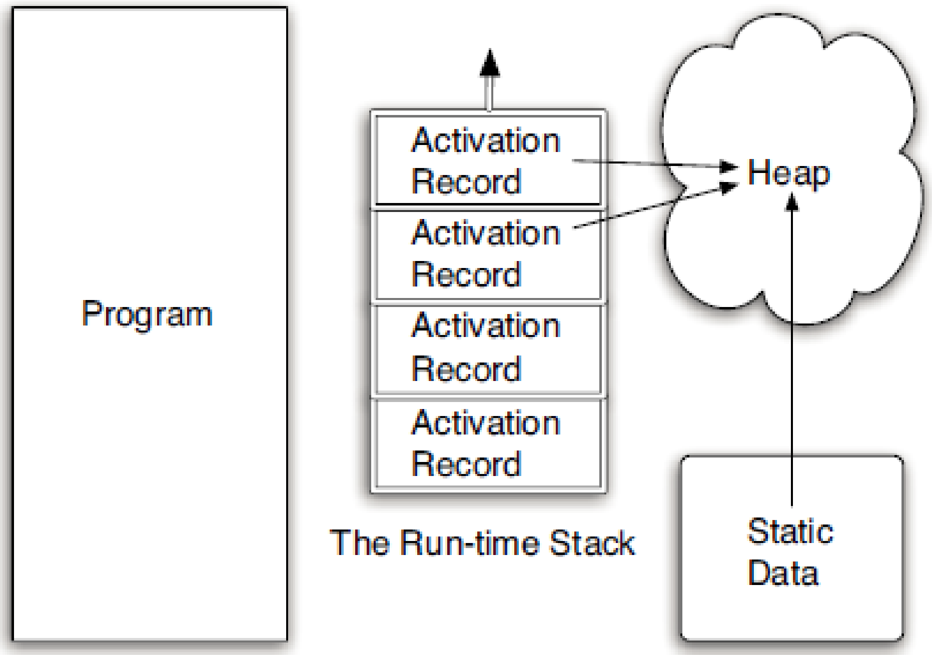
高效的C代码中隐藏的“风险”



```
/* simple_overflow.c
 *
 *   《网络渗透技术》演示程序
 *  作者：san，alert7，eyas，watercloud
 *
 *  Simple program to demonstrate buffer overflows
 *  on the IA32 architecture.
 */

#include <stdio.h>
#include <string.h>
char largebuff[] =
"1234512345123451234512345===ABCD";
int main (void)
{
    char smallbuff[16];
    strcpy (smallbuff, largebuff);
}
```

风险性体现在哪里？

机器的抽象-命令式编程



Conceptual View of the Imperative Model

问题8：既然命令式程序以修改数据为己任，那么内存里的数据，一定是只会被“我自己”修改吗？

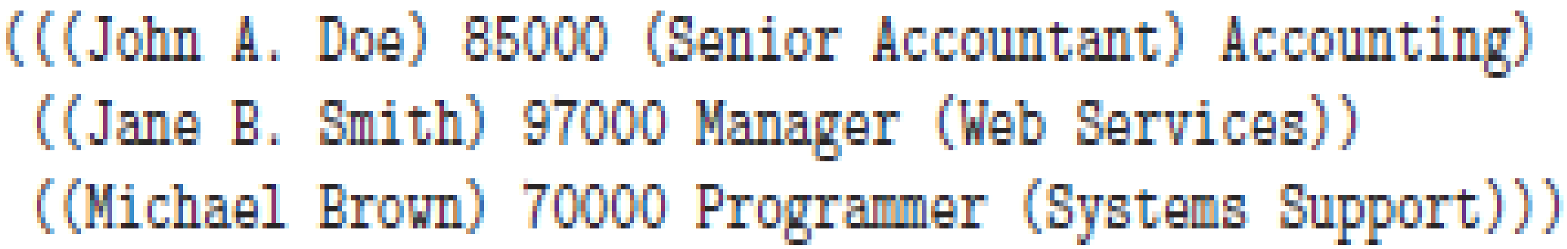
While this paradigm is close to the real architecture of the computer, it is quite far from its mathematical origins

问题9:

这是什么意思?

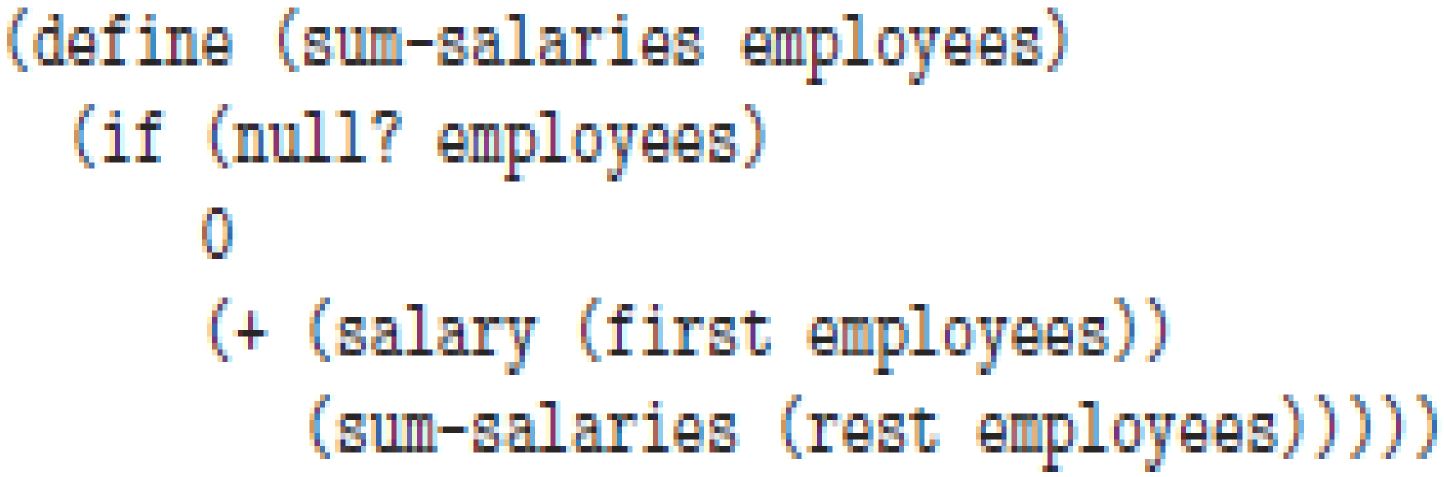
算法即函数-LISP语言

数据的形式



```
(((John A. Doe) 85000 (Senior Accountant) Accounting)
 ((Jane B. Smith) 97000 Manager (Web Services))
 ((Michael Brown) 70000 Programmer (Systems Support)))
```

程序的形式

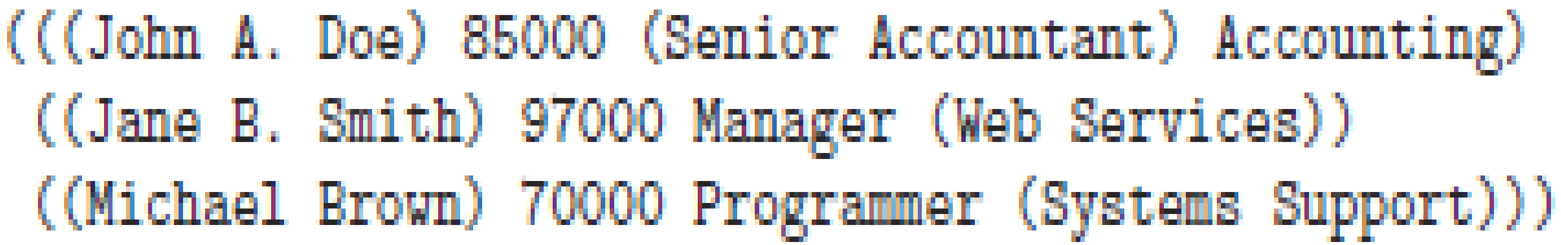


```
(define (sum-salaries employees)
  (if (null? employees)
      0
      (+ (salary (first employees))
         (sum-salaries (rest employees)))))
```

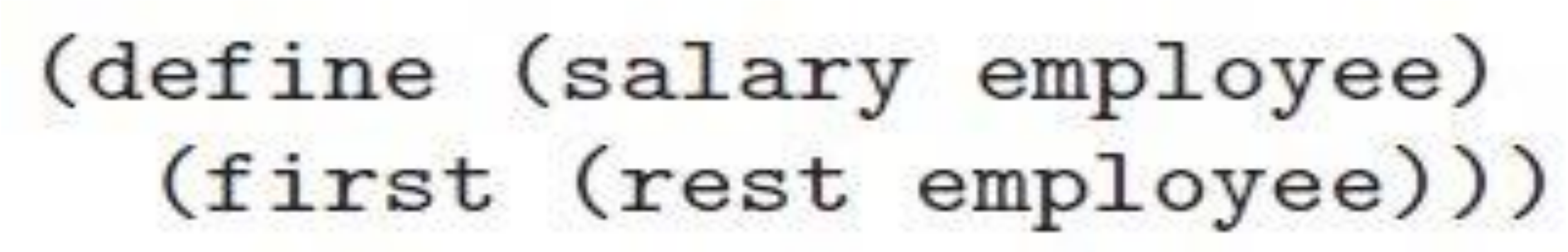
数据的形式和“程序”（函数）是一样的 - list

如何定义salary函数？

数据

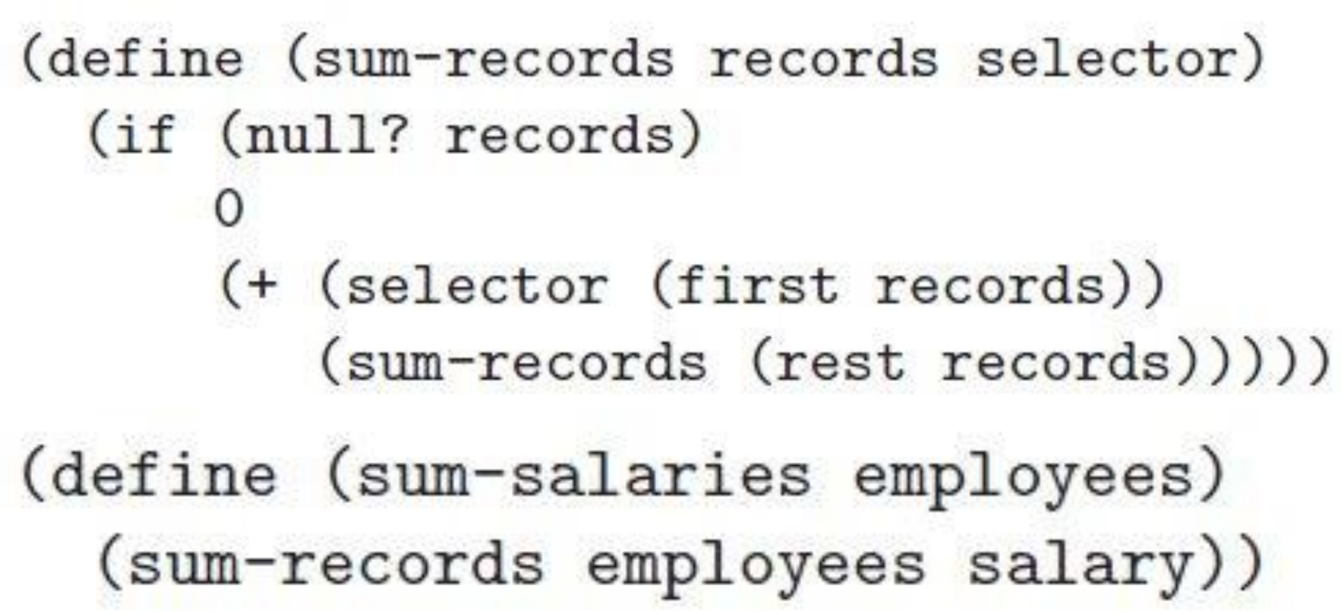


```
(((John A. Doe) 85000 (Senior Accountant) Accounting)
 ((Jane B. Smith) 97000 Manager (Web Services))
 ((Michael Brown) 70000 Programmer (Systems Support)))
```



```
(define (salary employee)
  (first (rest employee)))
```

观察以下定义

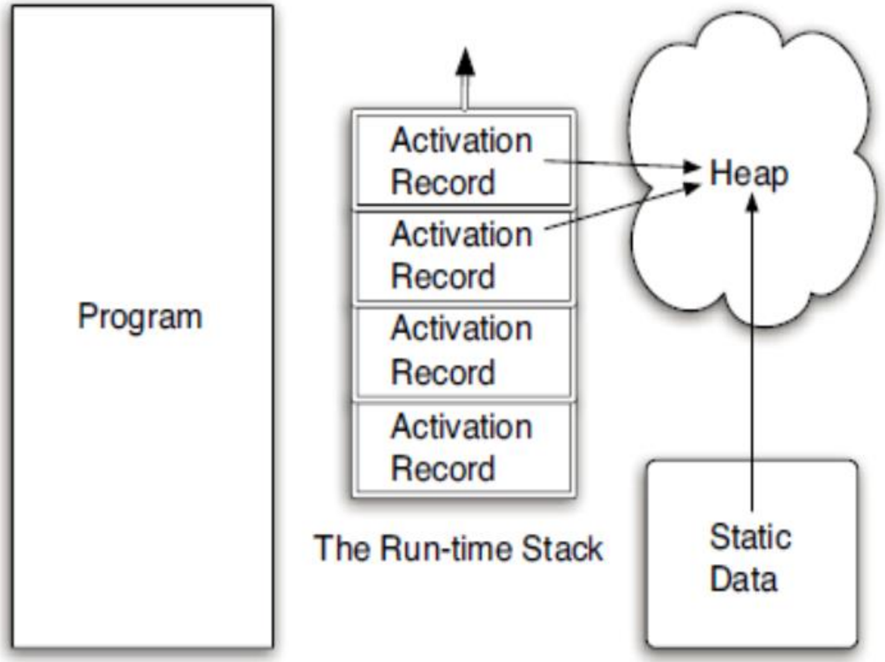


```
(define (sum-records records selector)
  (if (null? records)
      0
      (+ (selector (first records))
         (sum-records (rest records)))))

(define (sum-salaries employees)
  (sum-records employees salary))
```

这也可以看作函数sum-salaries的递归定义。

Unlike the formulation of this algorithm in Imperative language, this definition is recursive. And, in fact, recursion is the central and most natural control structure in LISP.

类似的架构，不同的视点



Functional programming: 有什么不同？

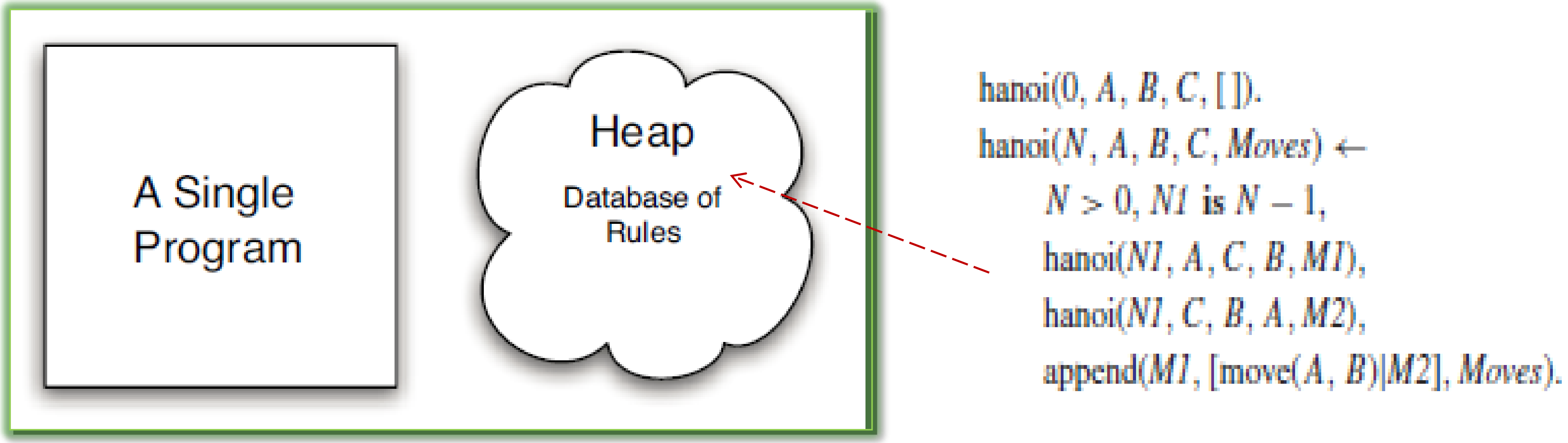
问题10：

什么是“garbage collector”？

为什么它对于functional programming很重要？

程序和数据的界限消失了（程序只是函数调用）；
运行栈成为核心，而数据区不是程序员考虑的事了。

完全不同的模型



```
hanoi(0, A, B, C, []).
hanoi(N, A, B, C, Moves) ←
    N > 0, N1 is N − 1,
    hanoi(N1, A, C, B, M1),
    hanoi(N1, C, B, A, M2),
    append(M1, [move(A, B)|M2], Moves).
```

Conceptual View of the Logic Model of Computation

在logic programming中，
“推理”和“计算”是统一的。

Prolog是通过一个“虚拟机”（推理引擎）来实现的。
它不仅要决定如何做，还要决定作什么。也就是说，程序员不仅要考虑程序的行为，还得考虑interpreter的行为。

一个OO的场景

顾客、收银、队列：

需求：顾客根据购物篮中物品数量，选择排哪个队伍(多->短队；少->快速通道队伍).如何建立一个软件系统，模拟超市顾客的check out？

备注：顾客由超市的统计数据由模拟软件产生；

一个有趣的想法：

- 将计算机理解为现实世界的直接模拟：
  - The object-oriented paradigm views the computer’s memory as being composed of many **objects**, corresponding to the data structures of the imperative view.
- 顾客对象；队列对象；收银对象
  - 新顾客对象：向队列对象询问队列长度，并加入合适队列
  - 队列对象：接受顾客的排队要求，询问队列第一位物品数量，通知收银对象
  - 收银对象：根据队列的通知，处理收银业务。业务完成后，通知队列对象
  - 队列对象：接受收银对象的通知，移除队列第一位顾客

```
public class Linkable
{
    private Object _item;
    private Linkable _next;

    public Linkable(Object x)
    {
        _item = x;
        _next = null;
    }
    public Object item()
    {
        return _item;
    }

    public Linkable next()
    {
        return _next;
    }

    public void set_next(Linkable next)
    {
        _next = next;
    }
}
```

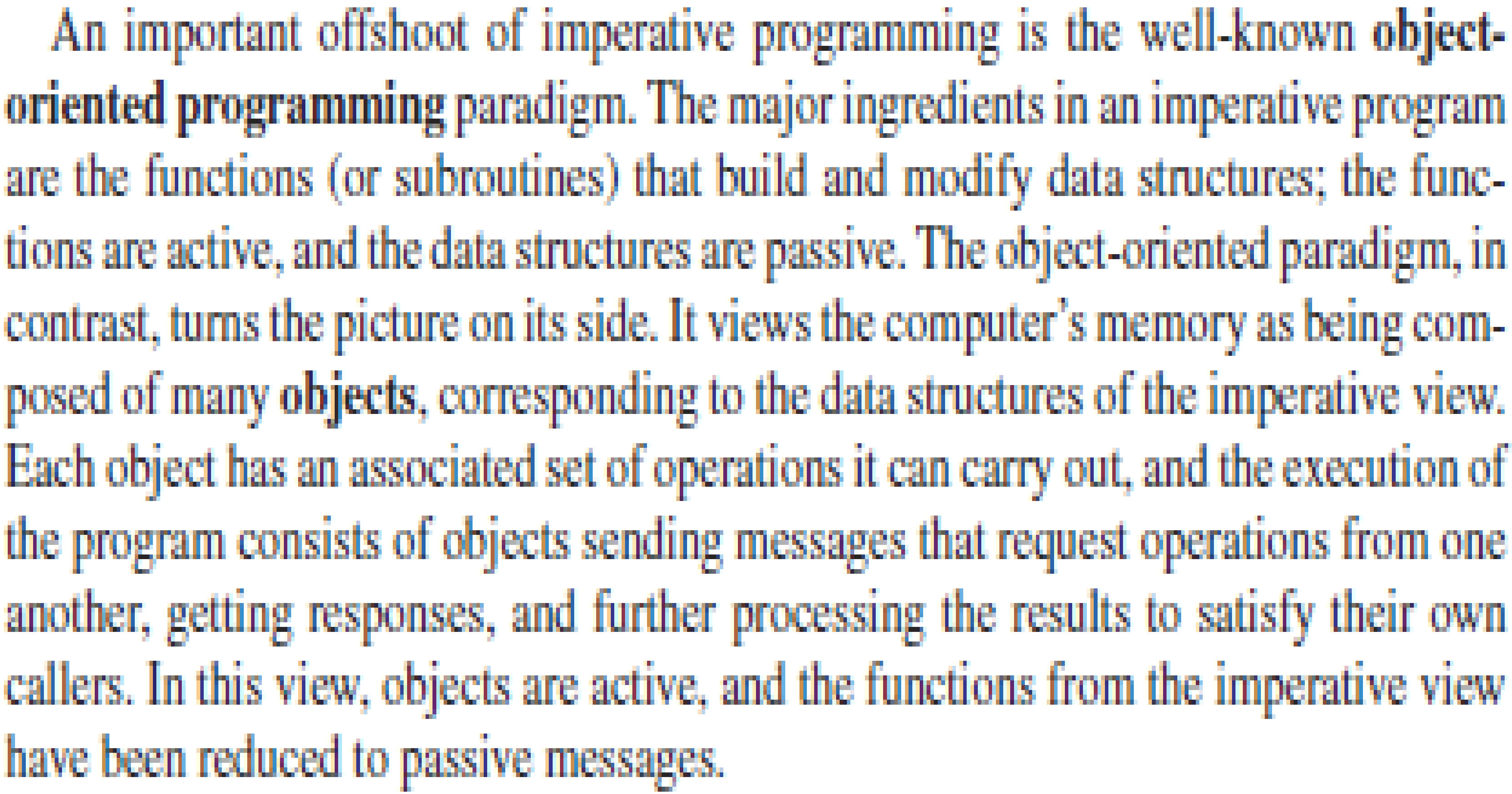
```
public interface Queue
{
    boolean empty();
    Object front();
    void add(Object x);
    void remove();
}
```

```
public class LinkedQueue implements Queue
{
    private Linkable _front = null, _back = nul

    public boolean empty()
    {
        return _front == null;
    }

    public Object front()
    {
        return _front.item();
    }

    public void add(Object x)
    {
        Linkable new_back = new Linkable(x);
        if (_front == null)
        {
            _front = new_back;
            _back = new_back;
        }
        else
        {
            _back.set_next(new_back);
            _back = new_back;
        }
    }

    public void remove()
    {
        _front = _front.next();
    }
}
```

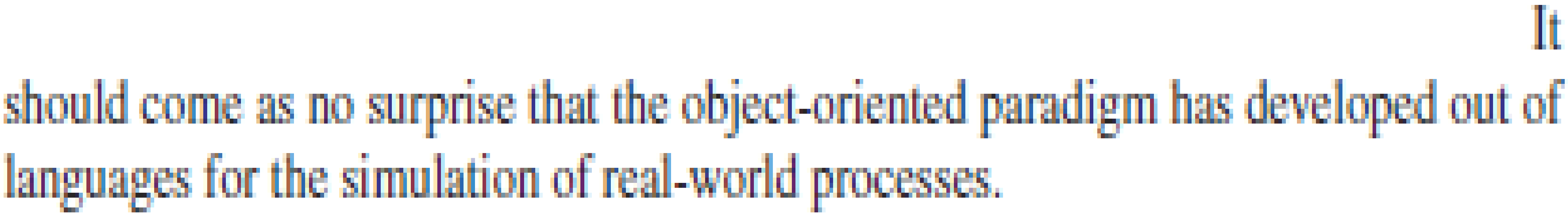
OO中的抽象

Such a class describes *what* a queue can do, but not *how* it does it. What is nice about this distinction is that the “what” is exactly the information that other classes need in order to use queues; they don’t really need the “how.”

从imperative 到 object-oriented



An important offshoot of imperative programming is the well-known **object-oriented programming** paradigm. The major ingredients in an imperative program are the functions (or subroutines) that build and modify data structures; the functions are active, and the data structures are passive. The object-oriented paradigm, in contrast, turns the picture on its side. It views the computer's memory as being composed of many **objects**, corresponding to the data structures of the imperative view. Each object has an associated set of operations it can carry out, and the execution of the program consists of objects sending messages that request operations from one another, getting responses, and further processing the results to satisfy their own callers. In this view, objects are active, and the functions from the imperative view have been reduced to passive messages.

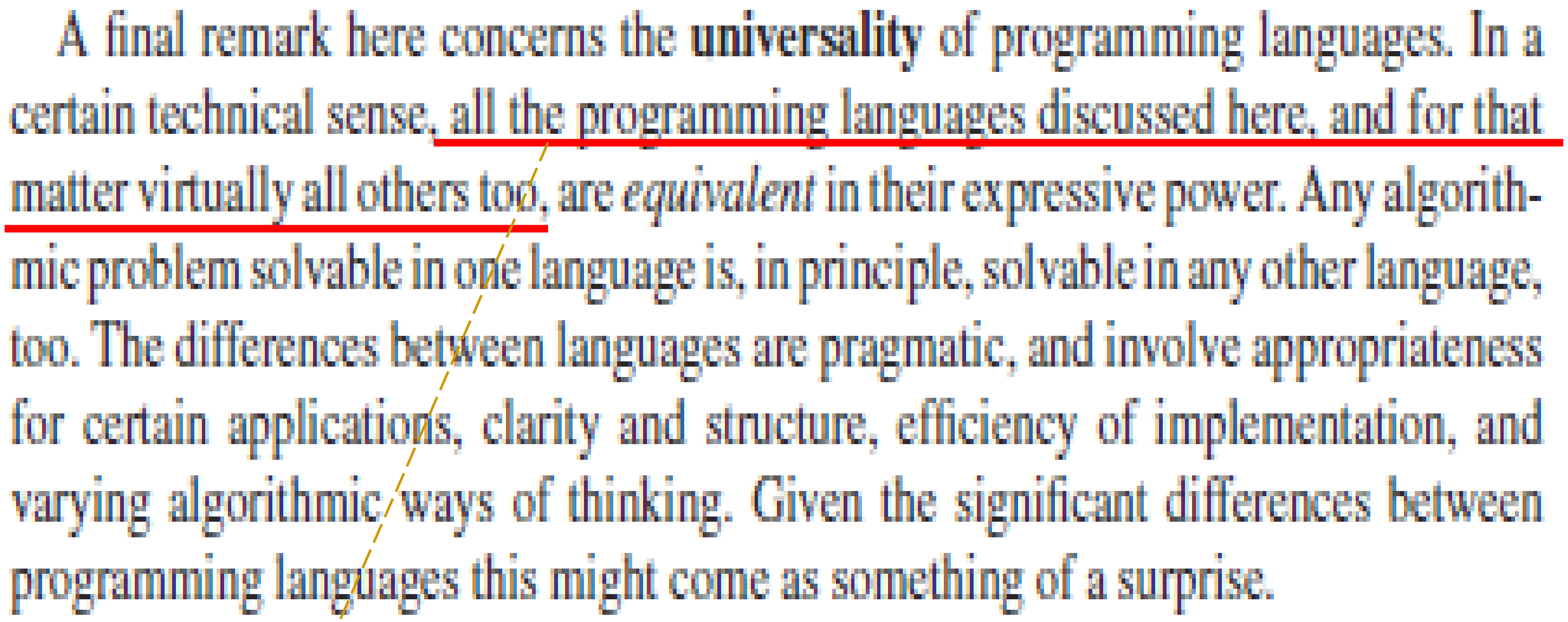


It should come as no surprise that the object-oriented paradigm has developed out of languages for the simulation of real-world processes.

问题11：

你注意到这句话吗？它是否能反映面向对象方法为什么会流行，并推动了传统的imperative方法的发展？

A Final Remark



A final remark here concerns the **universality** of programming languages. In a certain technical sense, all the programming languages discussed here, and for that matter virtually all others too, are *equivalent* in their expressive power. Any algorithmic problem solvable in one language is, in principle, solvable in any other language, too. The differences between languages are pragmatic, and involve appropriateness for certain applications, clarity and structure, efficiency of implementation, and varying algorithmic ways of thinking. Given the significant differences between programming languages this might come as something of a surprise.

So-called “Turing complete” language