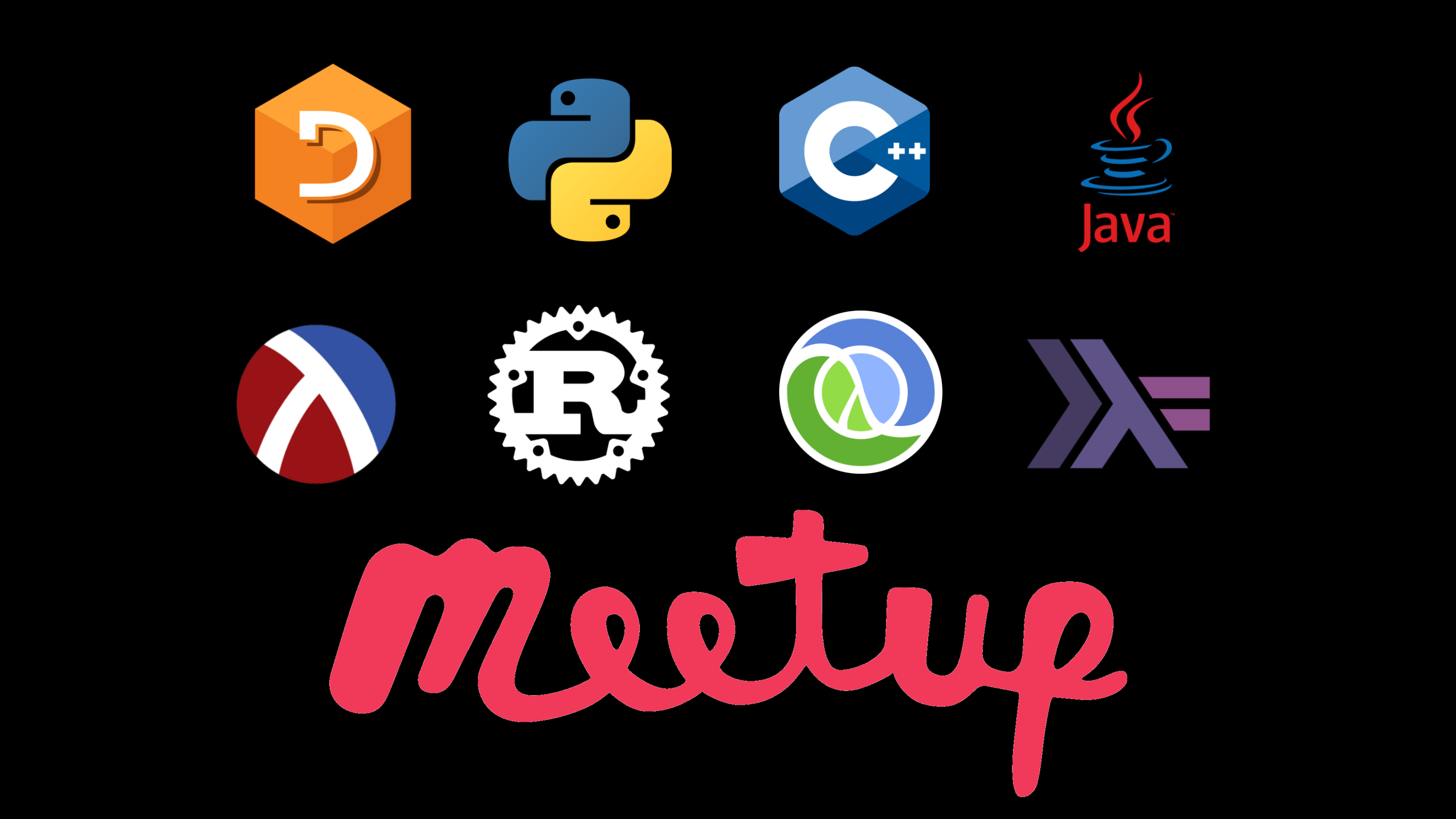
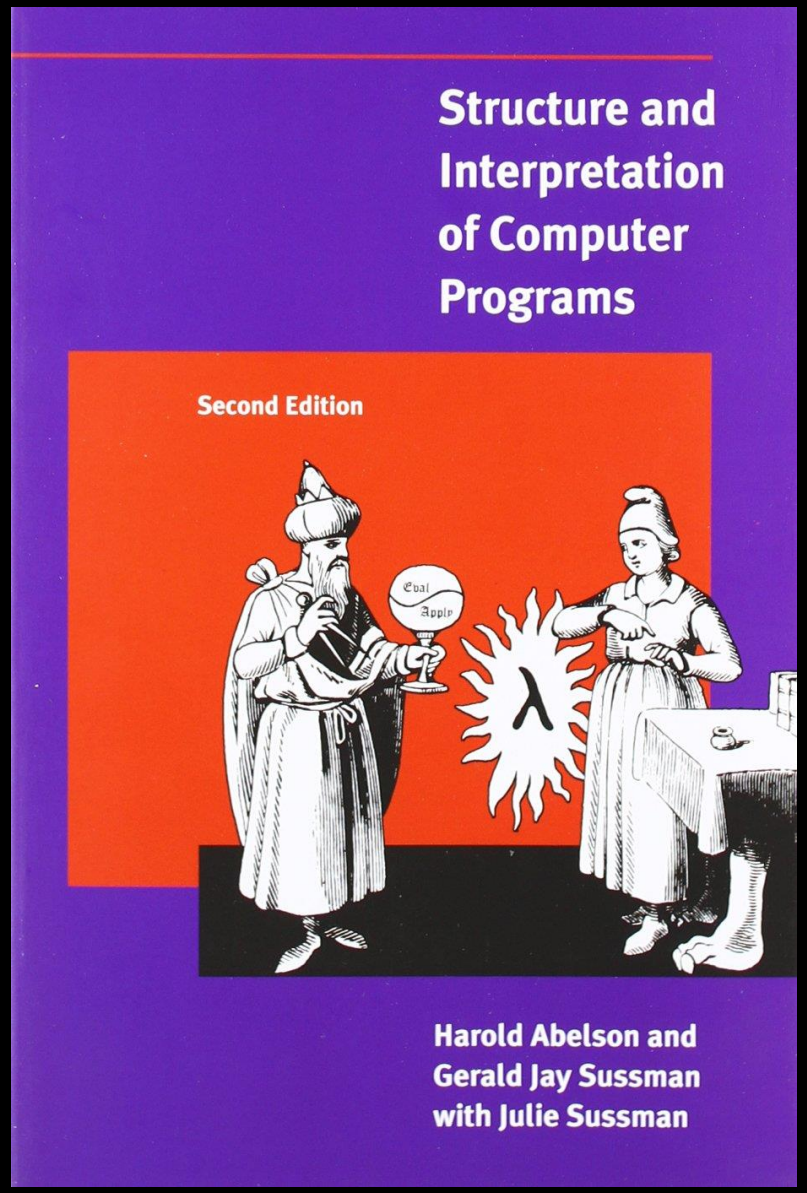

Java
meetup



Structure and Interpretation of Computer Programs

Chaper 5.3

Before we start …

Friendly Environment Policy

Berlin Code of Conduct


DISCORD

Programming Languages Virtual Meetup

1 Tweet

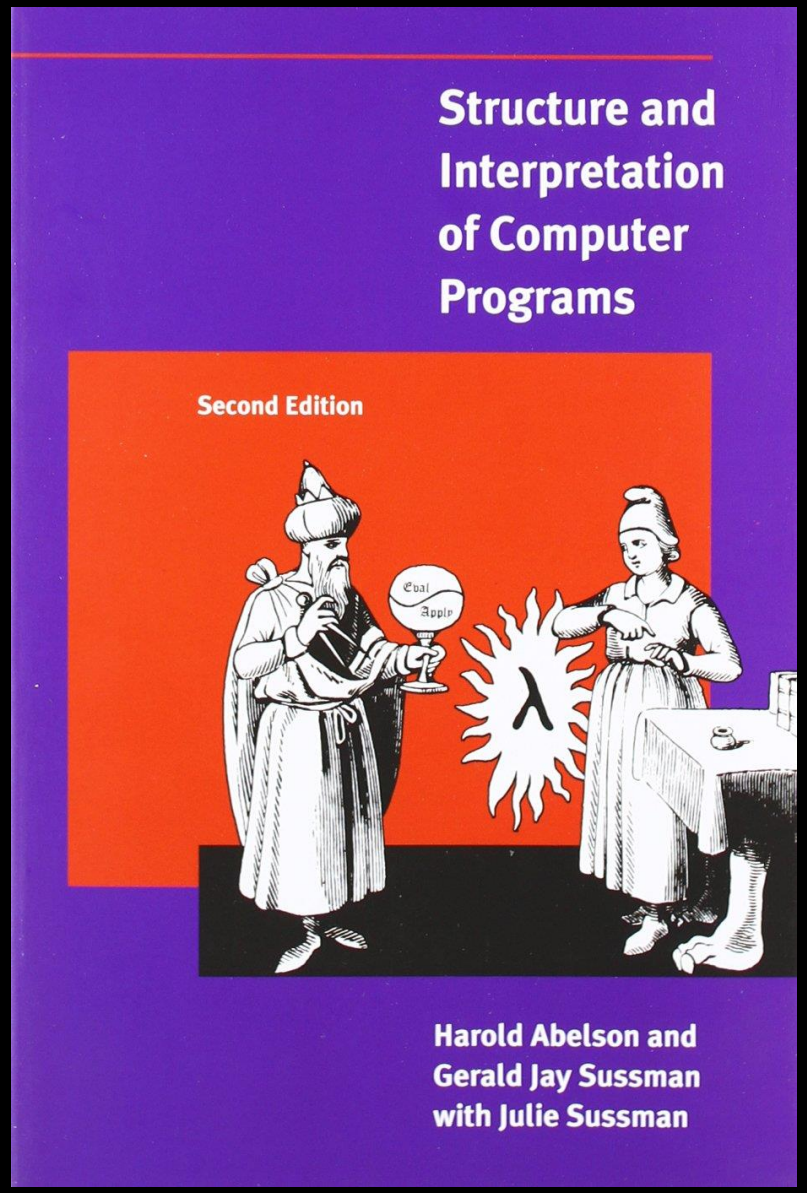
Following

Programming Languages Virtual Meetup

@PLvirtualmeetup

Official Twitter account of the Programming Languages Virtual Meetup. The meetup group is currently working through SICP: web.mit.edu/alexmv/6.037/s....

Toronto, CA meetup.com/Programming-La... Joined March 2020



Structure and Interpretation of Computer Programs

Chaper 5.3

In Section 5.4, we will show how to implement a Scheme evaluator as a register machine. In order to simplify the discussion, we will assume that our register machines can be equipped with a *list-structured memory*, in which the basic operations for manipulating list-structured data are primitive. Postulating the existence of such a memory is a useful abstraction when one is focusing on the mechanisms of control in a Scheme interpreter, but this does not reflect a realistic view of the actual primitive data operations of contemporary computers. To obtain a more complete picture of how a Lisp system operates, we must investigate how list structure can be represented in a way that is compatible with conventional computer memories.

Lisp systems thus provide an *automatic storage allocation* facility to support the illusion of an infinite memory. When a data object is no longer needed, the memory allocated to it is automatically recycled and used to construct new data objects. There are various techniques for providing such automatic storage allocation. The method we shall discuss in this section is called *garbage collection*.

Representing Lisp data

We can use vectors to implement the basic pair structures required for a list-structured memory. Let us imagine that computer memory is divided into two vectors: `the-cars` and `the-cdrs`. We will represent list structure as follows: A pointer to a pair is an index into the two vectors. The `car` of the pair is the entry in `the-cars` with the designated index, and the `cdr` of the pair is the entry in `the-cdrs` with the designated index.

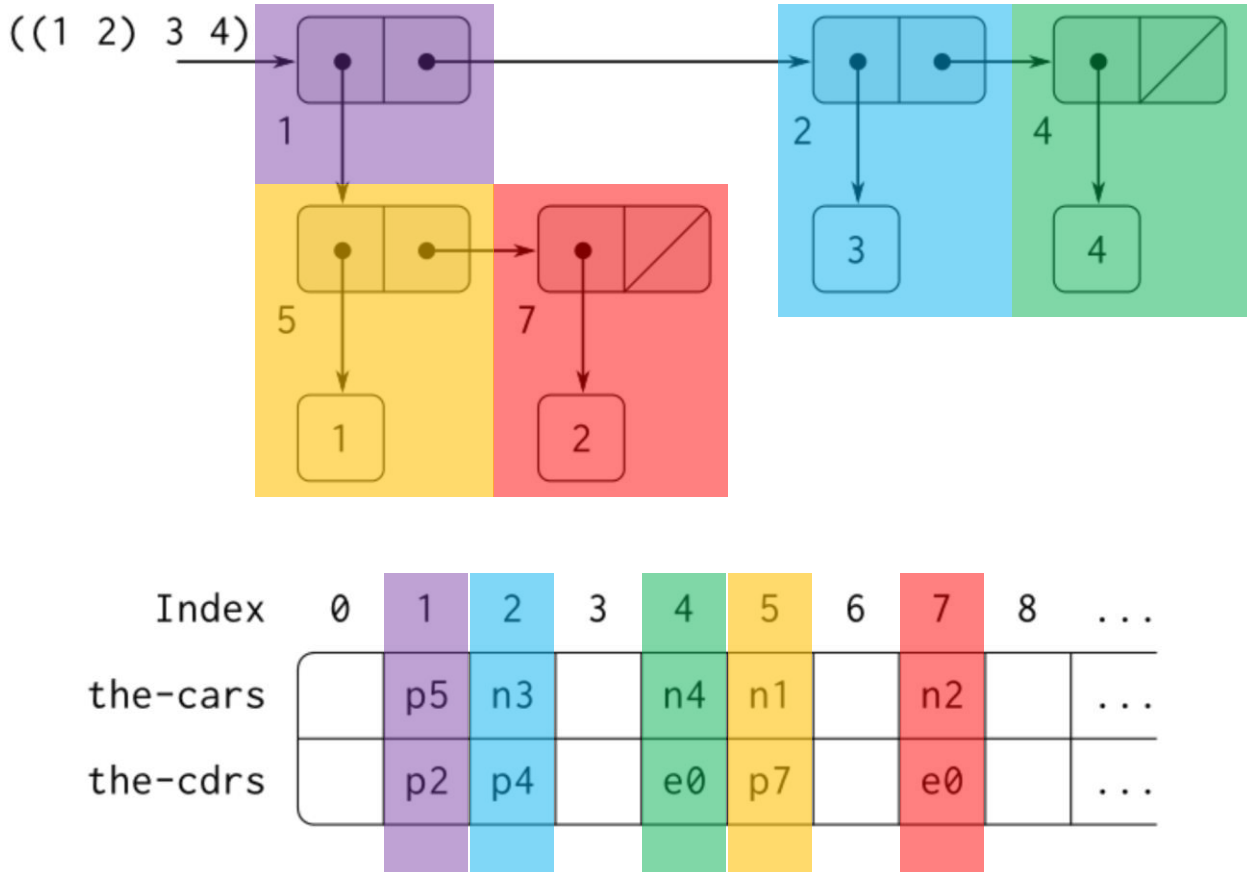
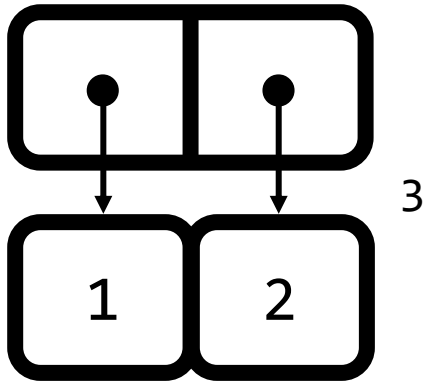


Figure 5.14: Box-and-pointer and memory-vector representations of the list `((1 2) 3 4)`.

Exercise 5.20: Draw the box-and-pointer representation and the memory-vector representation (as in Figure 5.14) of the list structure produced by

```
(define x (cons 1 2))
(define y (list x x))
```

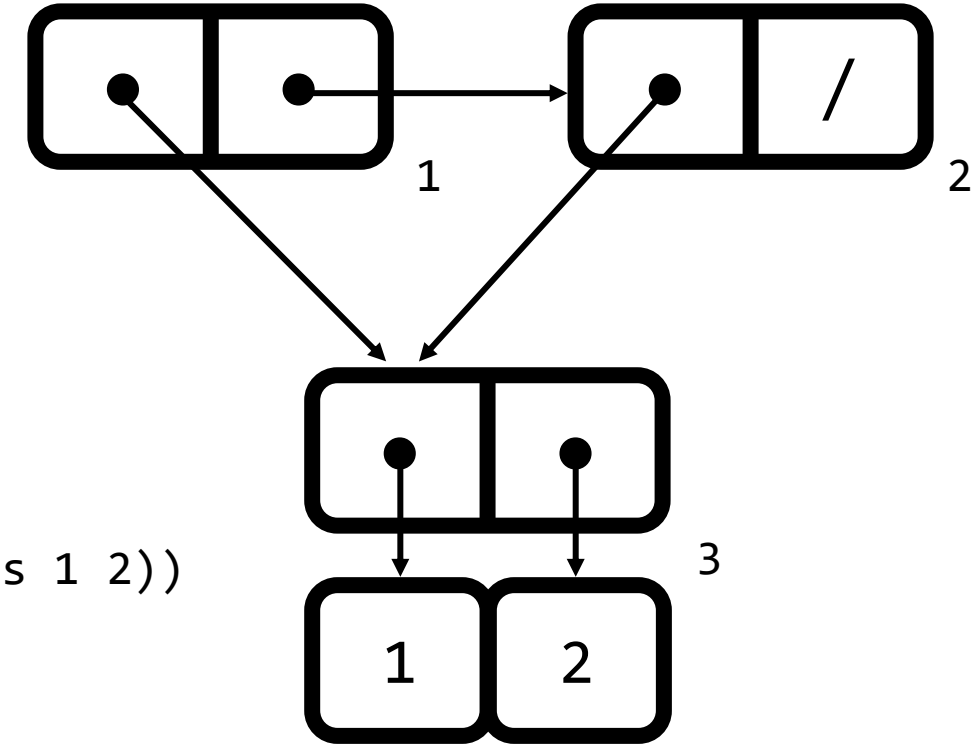
```
(define x (cons 1 2))
```



3

index	0	1	2	3	4
the-cars				n1	
the-cdrs				n2	

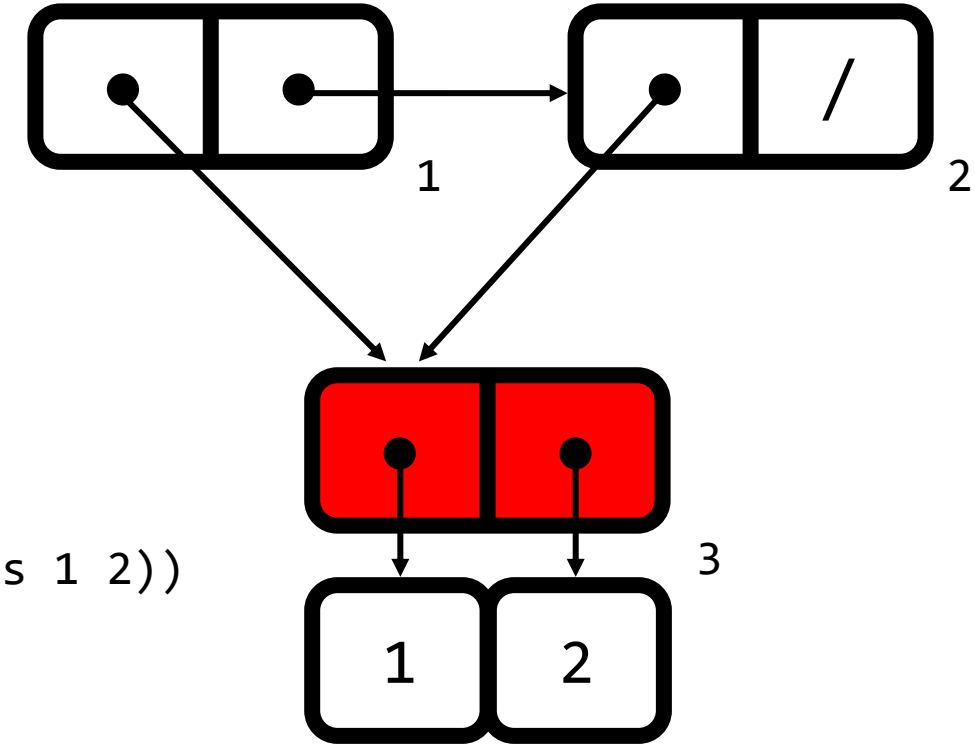
(define y (list x x))



(define x (cons 1 2))

index	0	1	2	3	4
the-cars		p3	p3	n1	
the-cdrs		p2	e0	n2	

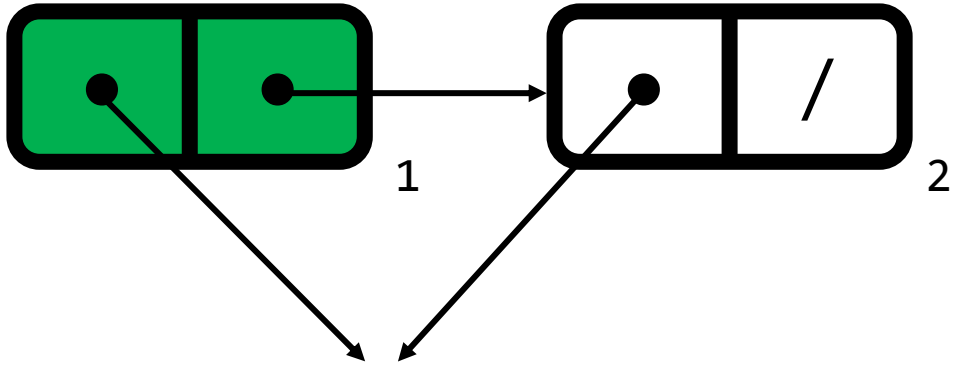
(define y (list x x))



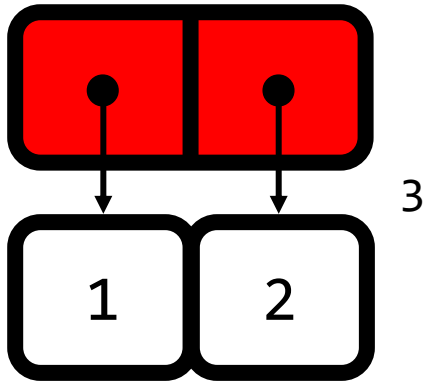
(define x (cons 1 2))

index	0	1	2	3	4
the-cars		p3	p3	n1	
the-cdrs		p2	e0	n2	

(define y (list x x))

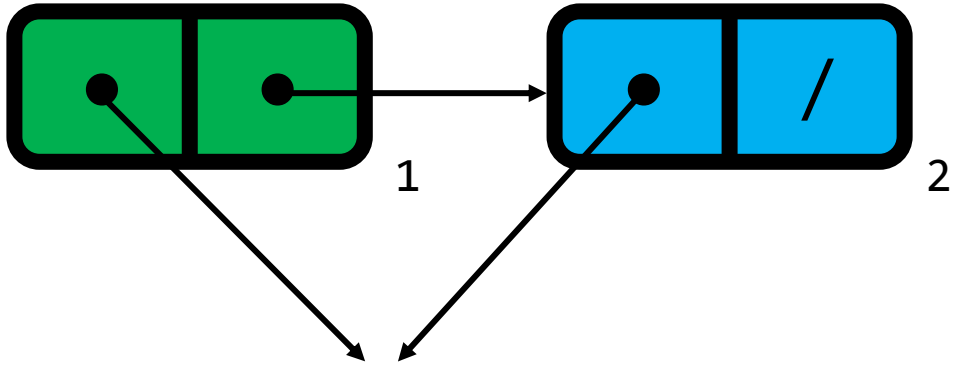


(define x (cons 1 2))

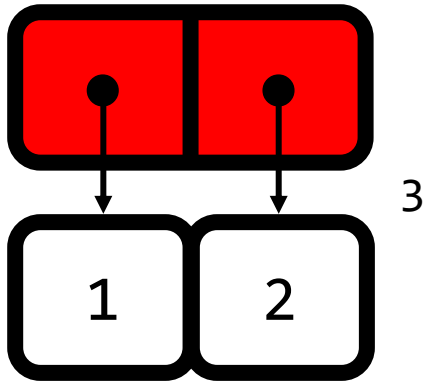


index	0	1	2	3	4
the-cars		p3	p3	n1	
the-cdrs		p2	e0	n2	

(define y (list x x))



(define x (cons 1 2))



index	0	1	2	3	4
the-cars		p3	p3	n1	
the-cdrs		p2	e0	n2	

Exercise 5.21: Implement register machines for the following procedures. Assume that the list-structure memory operations are available as machine primitives.

a. Recursive `count-leaves`:

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+ (count-leaves (car tree))
                 (count-leaves (cdr tree))))))
```

http://community.schemewiki.org/?sicp-ex-5.21 meteorgan

```
(define count-leaves-machine
  (make-machine
   '(continue val tree var) ; added explicit registers
   (list (list '+ +) (list 'null? null?)
         (list 'pair? pair?) (list 'car car) (list 'cdr cdr))
   '(
     (assign continue (label count-leaves-done))
     (assign val (const 0))
     tree-loop
     (test (op null?) (reg tree))
     (branch (label null-tree))
     (test (op pair?) (reg tree))
     (branch (label left-tree))
     (assign val (const 1))
     (goto (reg continue))
     left-tree
     (save tree)
     (save continue)
     (assign continue (label right-tree))
     (assign tree (op car) (reg tree))
     (goto (label tree-loop))
     right-tree
     (restore continue)
     (restore tree)
     (save continue)
     (save val)
     (assign continue (label after-tree))
     (assign tree (op cdr) (reg tree))
     (goto (label tree-loop))
     after-tree
     (assign var (reg val))
     (restore val)
     (restore continue)
     (assign val (op +) (reg var) (reg val))
     (goto (reg continue))
     null-tree
     (assign val (const 0))
     (goto (reg continue))
     count-leaves-done)))
```

http://community.schemewiki.org/?sicp-ex-5.21 meteorgan

```
(define count-leaves-machine
  (make-machine
   '(continue val tree var) ; added explicit registers
   (list (list '+ +) (list 'null? null?)
         (list 'pair? pair?) (list 'car car) (list 'cdr cdr))
   '(
     (assign continue (label count-leaves-done))
     (assign val (const 0))
    tree-loop
     (test (op null?) (reg tree))
     (branch (label null-tree))
     (test (op pair?) (reg tree))
     (branch (label left-tree))
     (assign val (const 1))
    (goto (reg continue))
    left-tree
     (save tree)
     (save continue)
     (assign continue (label right-tree))
     (assign tree (op car) (reg tree))
    (goto (label tree-loop))
    right-tree
     (restore continue)
     (restore tree)
     (save continue)
     (save val)
     (assign continue (label after-tree))
     (assign tree (op cdr) (reg tree))
    (goto (label tree-loop))
    after-tree
     (assign var (reg val))
     (restore val)
     (restore continue)
     (assign val (op +) (reg var) (reg val))
    (goto (reg continue))
    null-tree
     (assign val (const 0))
    (goto (reg continue))
    count-leaves-done)))
```

With a real computer we will eventually run out of free space in which to construct new pairs.[13]

[13]This may not be true eventually, because memories may get large enough so that it would be impossible to run out of free memory in the lifetime of the computer. For example, there are about $3 \cdot 10^{13}$ microseconds in a year, so if we were to cons once per microsecond we would need about 10^{15} cells of memory to build a machine that could operate for 30 years without running out of memory. That much memory seems absurdly large by today's standards, but it is not physically impossible. On the other hand, processors are getting faster and a future computer may have large numbers of processors operating in parallel on a single memory, so it may be possible to use up memory much faster than we have postulated.

There are many ways to perform garbage collection. The method we shall examine here is called *stop-and-copy*. The basic idea is to divide memory into two halves: "working memory" and "free memory." When `cons` constructs pairs, it allocates these in working memory. When working memory is full, we perform garbage collection by locating all the useful pairs in working memory and copying these into consecutive locations in free memory. (The useful pairs are located by tracing all the `car` and `cdr` pointers, starting with the machine registers.) Since we do not copy the garbage, there will presumably be additional free memory that we can use to allocate new pairs. In addition, nothing in the working memory is needed, since all the useful pairs in it have been copied. Thus, if we interchange the roles of working memory and free memory, we can continue processing; new pairs will be allocated in the new working memory (which was the old free memory). When this is full, we can copy the useful pairs into the new free memory (which was the old working memory).[15]

[15]This idea was invented and first implemented by Minsky, as part of the implementation of Lisp for the PDP-1 at the MIT Research Laboratory of Electronics. It was further developed by Fenichel and Yochelson (1969) for use in the Lisp implementation for the Multics time-sharing system. Later, Baker (1978) developed a "real-time" version of the method, which does not require the computation to stop during garbage collection. Baker's idea was extended by Hewitt, Lieberman, and Moon (see Lieberman and Hewitt 1983) to take advantage of the fact that some structure is more volatile and other structure is more permanent.

An alternative commonly used garbage-collection technique is the *mark-sweep* method. This consists of tracing all the structure accessible from the machine registers and marking each pair we reach. We then scan all of memory, and any location that is unmarked is "swept up" as garbage and made available for reuse. A full discussion of the mark-sweep method can be found in Allen 1978.

The Minsky-Fenichel-Yochelson algorithm is the dominant algorithm in use for large-memory systems because it examines only the useful part of memory. This is in contrast to mark-sweep, in which the sweep phase must check all of memory. A second advantage of stop-and-copy is that it is a *compacting* garbage collector. That is, at the end of the garbage-collection phase the useful data will have been moved to consecutive memory locations, with all garbage pairs compressed out. This can be an extremely important performance consideration in machines with virtual memory, in which accesses to widely separated memory addresses may require extra paging operations.

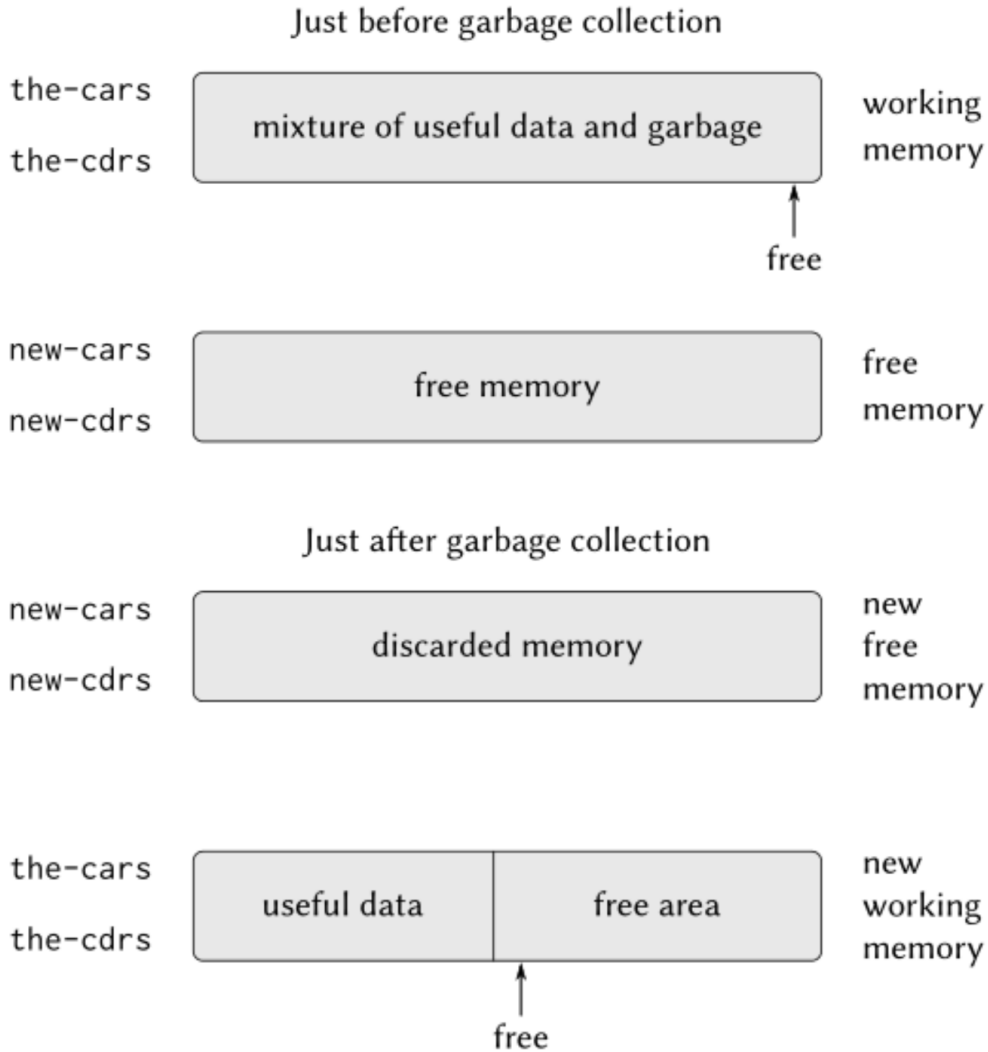


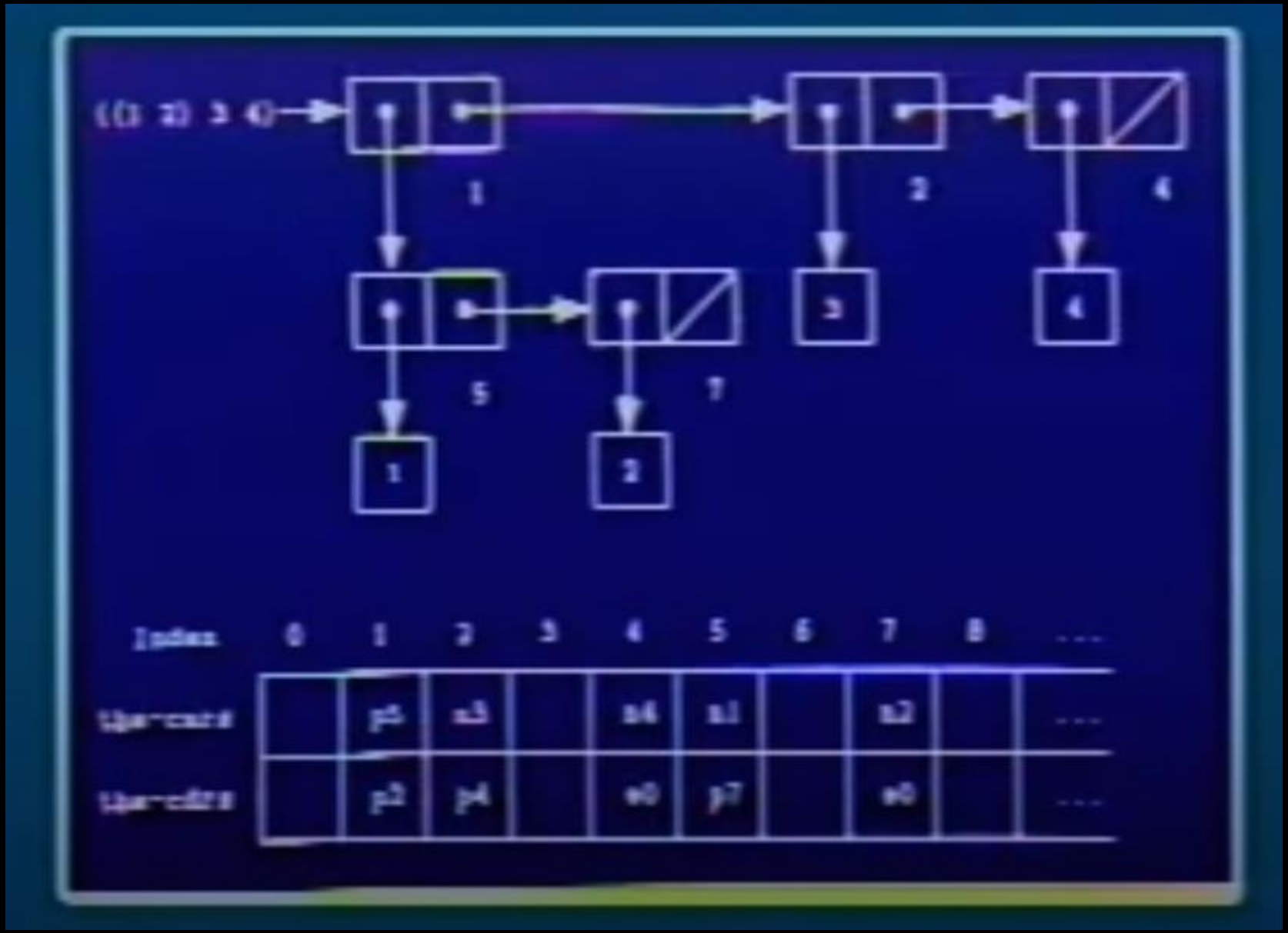
Figure 5.15: Reconfiguration of memory by the garbage-collection process.

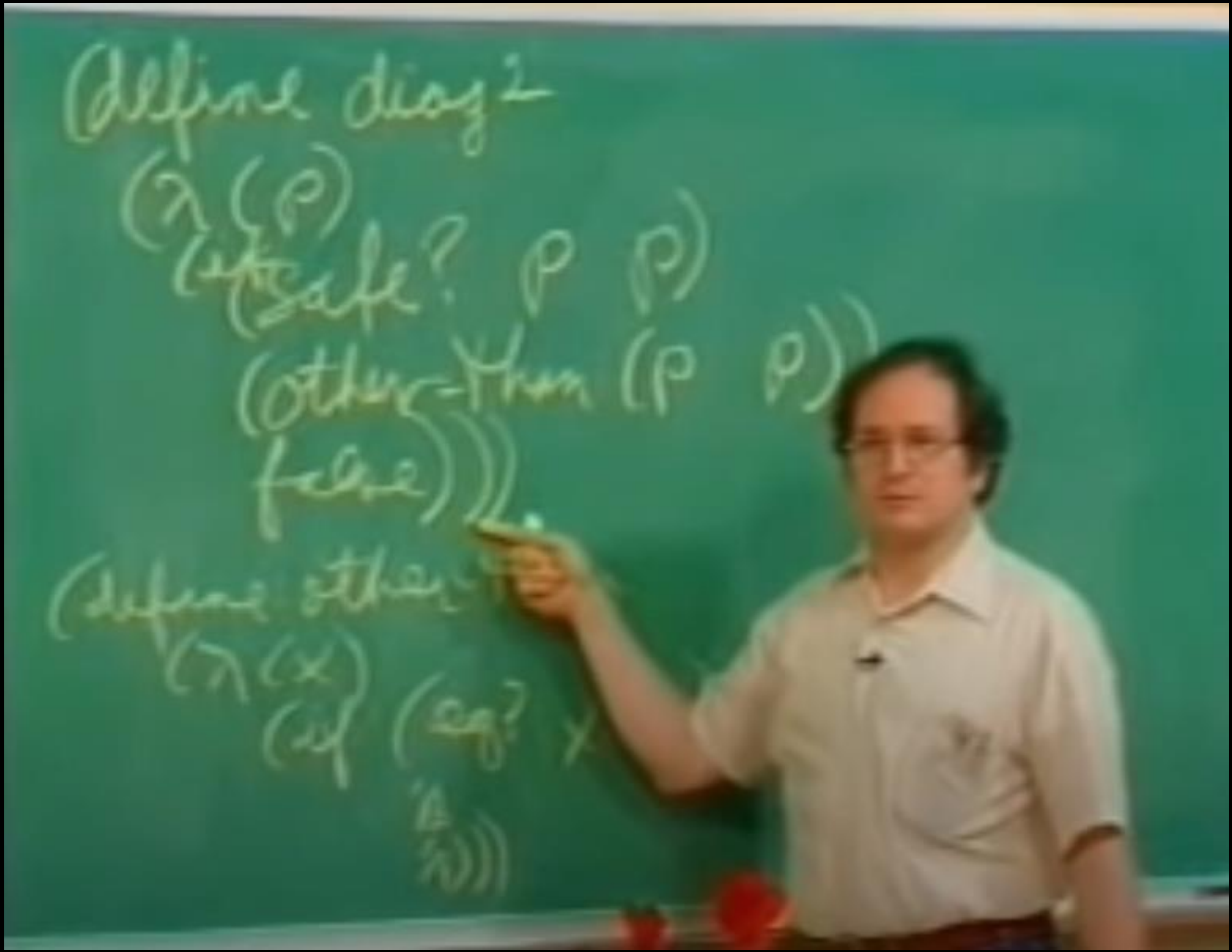
[17] The term *broken heart* was coined by David Cressey, who wrote a garbage collector for MDL, a dialect of Lisp developed at MIT during the early 1970s.

Structure & Interpretation of Computer Programs

Harold Abelson

Gerald Jay Sussman

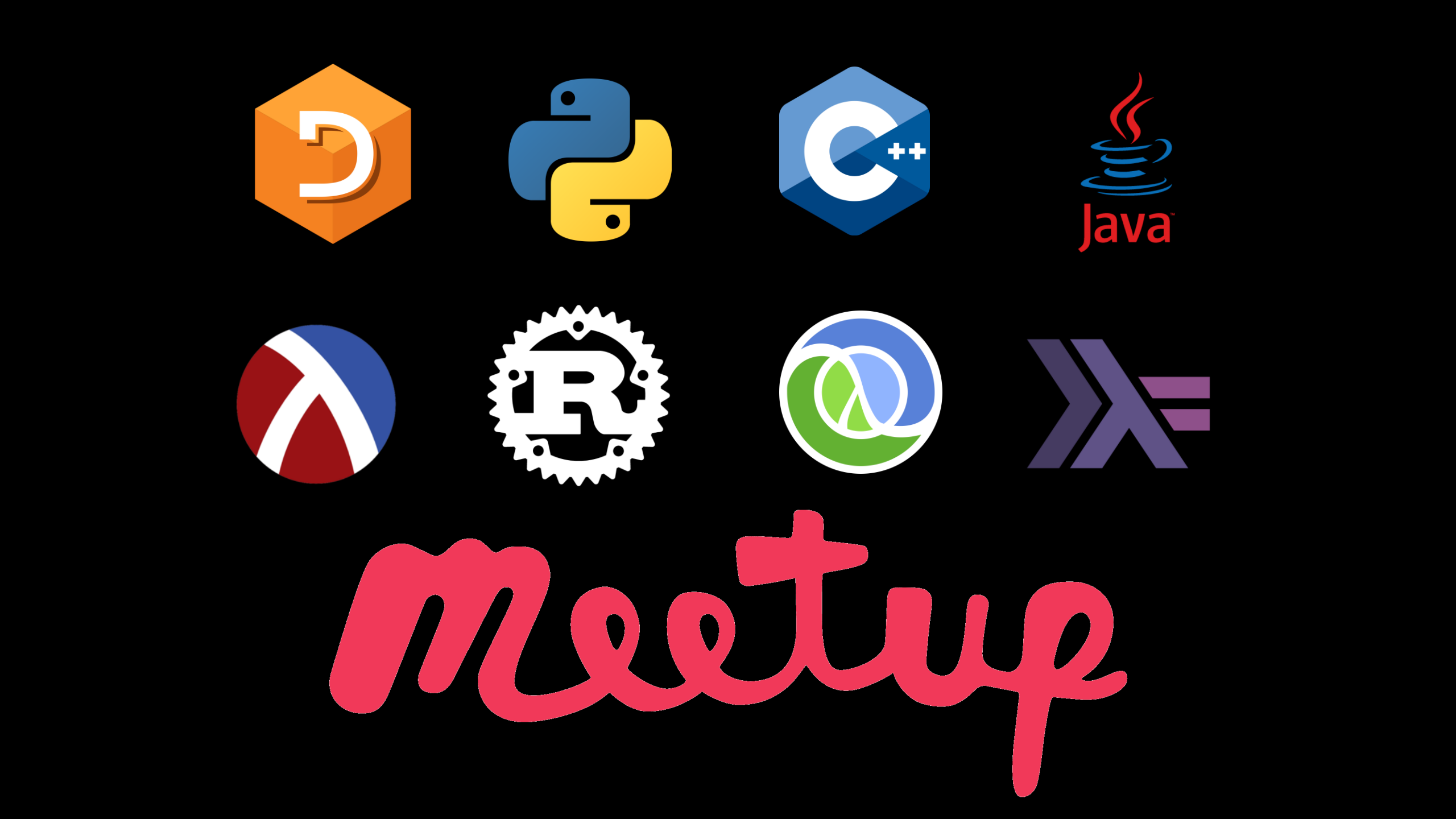

1
2
4
5
7
3
4
1
3
Index
0
1
2
3
4
5
6
7
8


(define diag2
(λ (p)
(safe? p p)
(other-than (p p))
false)))
(define other-than
(λ (x)
(if (eq? x
'A
'B)))

Math Nerd Gamer 9 years ago

56:54 "Is this the last question?"
That 20 second stare into the camera, with the final answer "Apparently so," is the nerdiest thing I have ever seen on this channel.

I love it.


Java
meetup