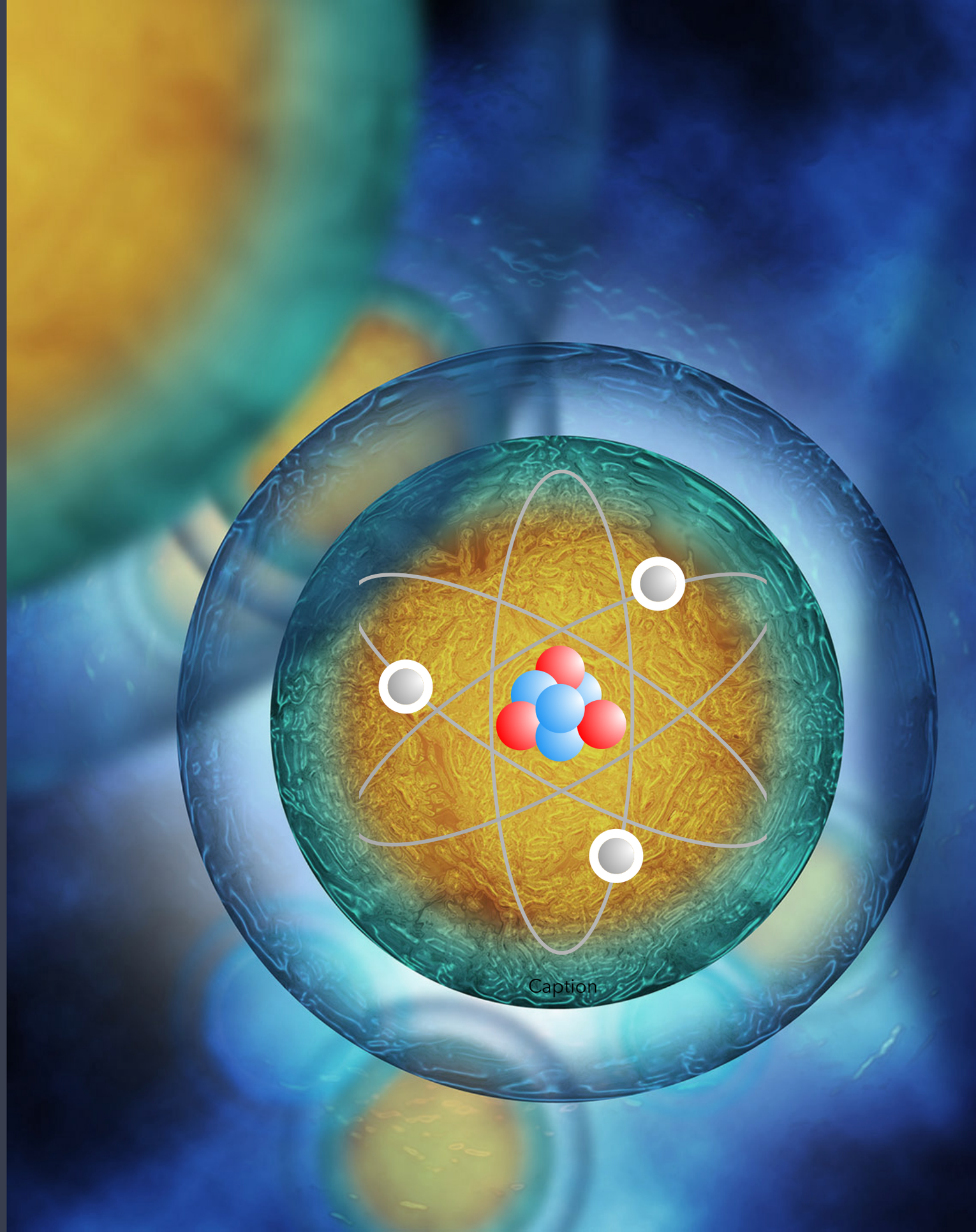
Paul Tarvydas

PROGRAMMING SIMPLICITY

Memory Layout Cells and Atoms

Sector Lisp and Lisp 1.5



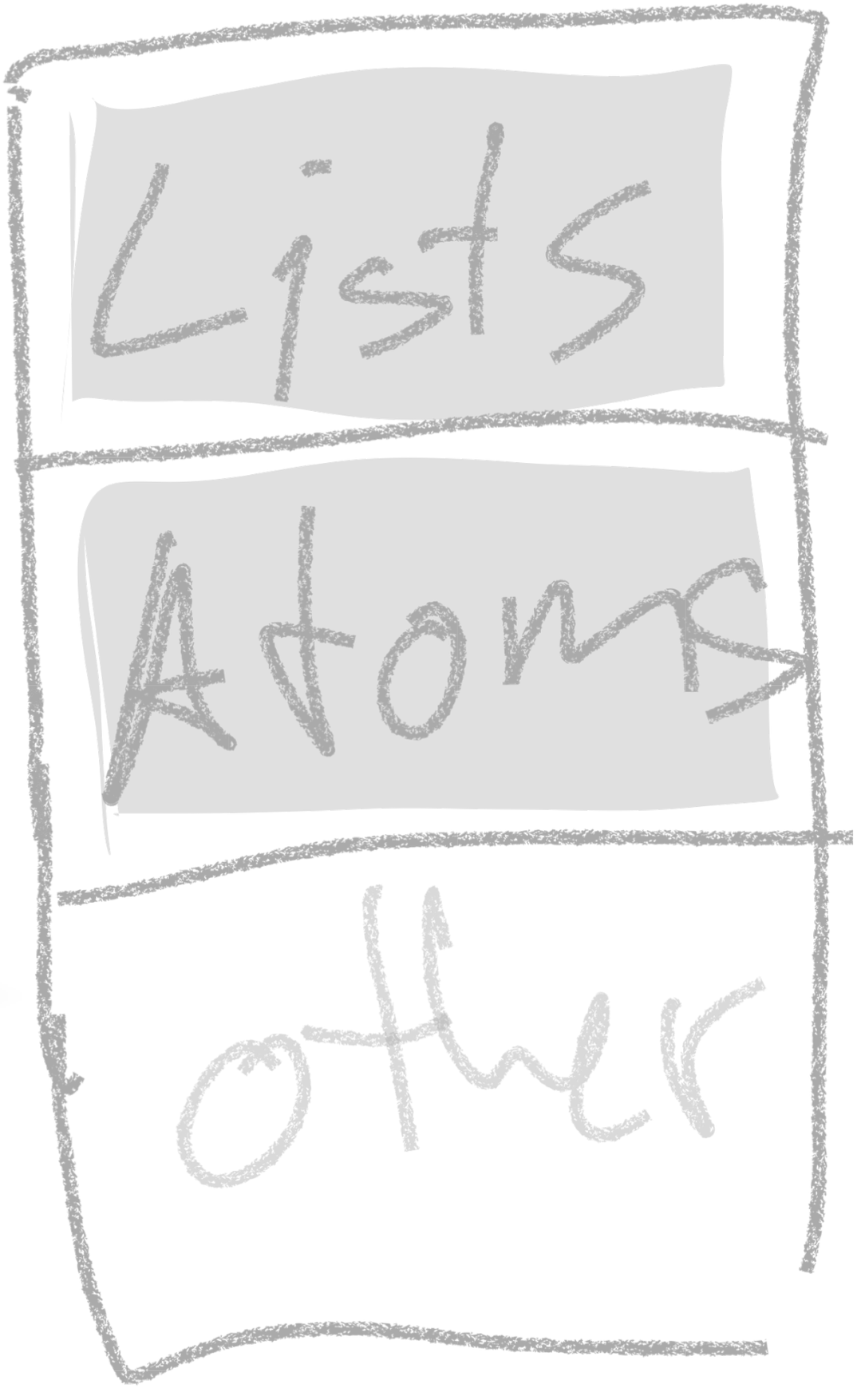
Cover art uses https://upload.wikimedia.org/wikipedia/commons/archive/1/15/20210331051741!Atomic_structure_of_Lithium-7.svg Under the CC license https://en.wikipedia.org/wiki/Creative_Commons

Cover art uses Apple's stock image for a Visual Textbook, "The Study of Cell Biology

Memory Layout

Partition memory into 2 main chunks

Plus some space for left-overs



BENEFITS OF PARTITIONING MEMORY INTO 2 SPACES

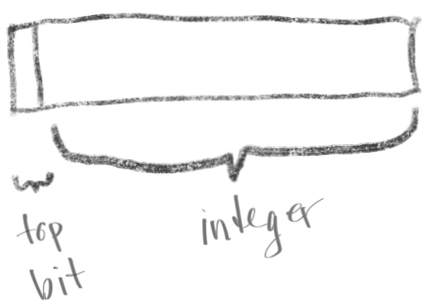
The main benefit is that it takes only 1 bit to tag each of the memory spaces.

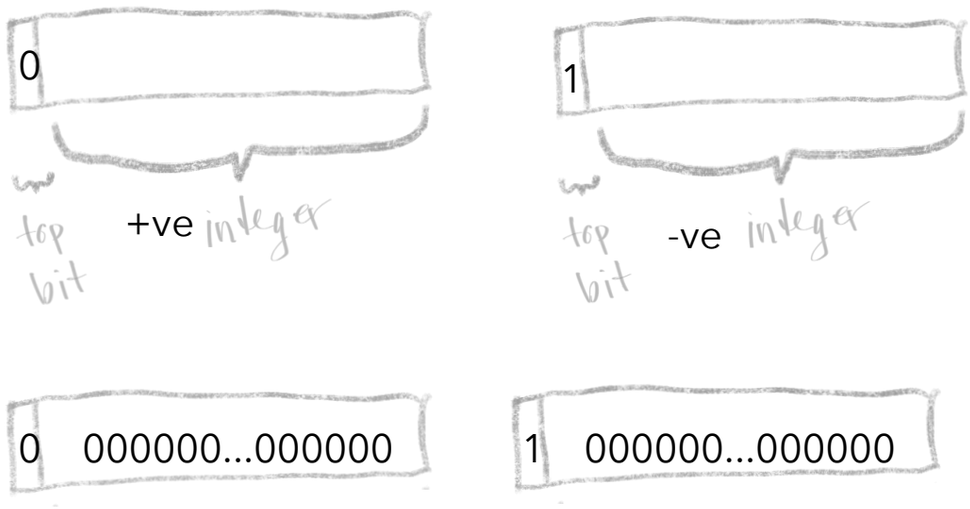
In this model, memory is treated as an array.

Indexing the array is done with *integers*.

CPUs incorporate efficient implementations of integers.

The top bit, the sign + or -, is used to flag the integers as being positive or negative.





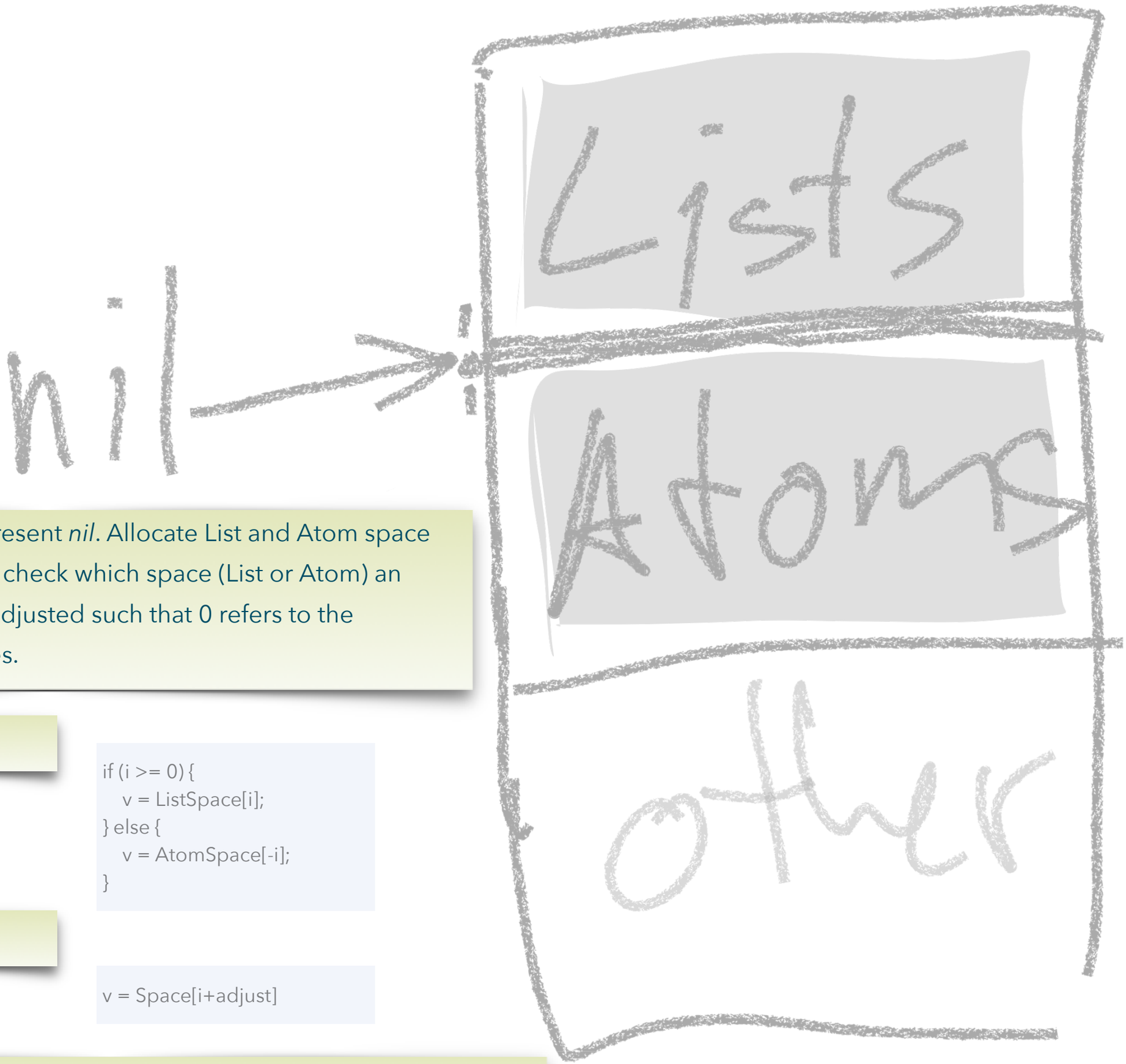
CPUs make it very efficient to test whether an integer is exactly zero (0).

We can use these minor details to create very efficient implementations of the memory array split into 2 types. We can set aside index zero for some special purpose, if we wish.

Let's say that we want all positive indices (except 0) to mean "list", and, all negative indices to mean "atom", and, let's use 0 to mean the special, frequently used, value *NIL*.

*Aside: at the lowest level, CPUs treat RAM as arrays of cells. Assembler programmers use the word "*pointer*" to mean an index into the RAM array. McCarthy, when he invented Lisp 1.5, used a similar trick to what has been described above. He simply treated all* pointers *as positive and negative indices and divided all of memory up into 2 spaces - cells and atoms - and reserved 0 as a special value (nil).*

Nil



Optimize by using index 0 to represent *nil*. Allocate List and Atom space contiguously to avoid needing to check which space (List or Atom) an index refers to. Indices must be adjusted such that 0 refers to the boundary between the two spaces.

Instead of writing…

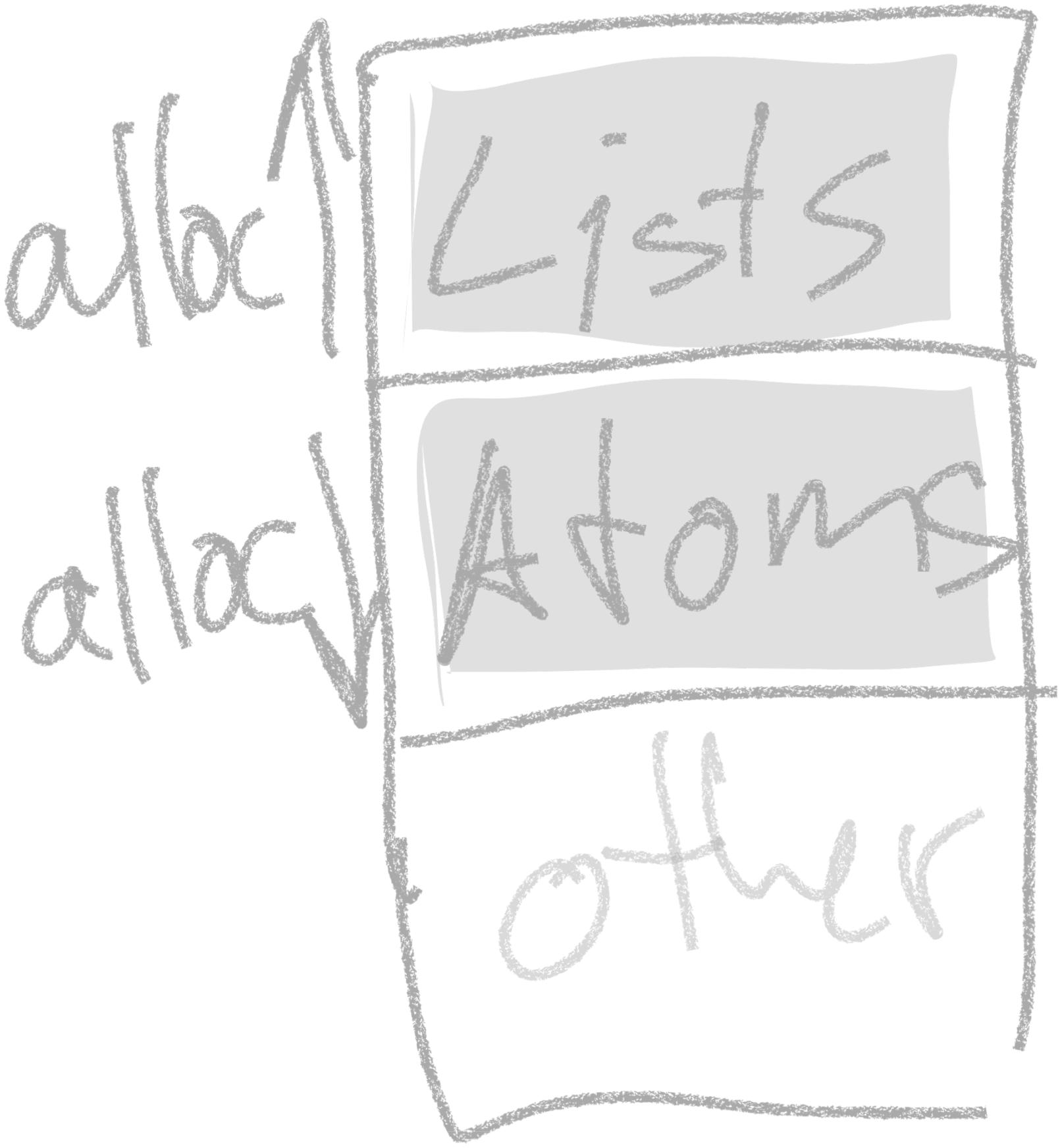
```
if (i >= 0) {
   v = ListSpace[i];
} else {
   v = AtomSpace[-i];
}
```

We can write…

```
v = Space[i+adjust]
```

Which boils down to less code and faster speed.

Allocating List and Atom Cells



Stacks, No Heap

True functional programming notation treats all memory in a stack-like manner.

When everything is a stack, you don't need Random Access heaps.

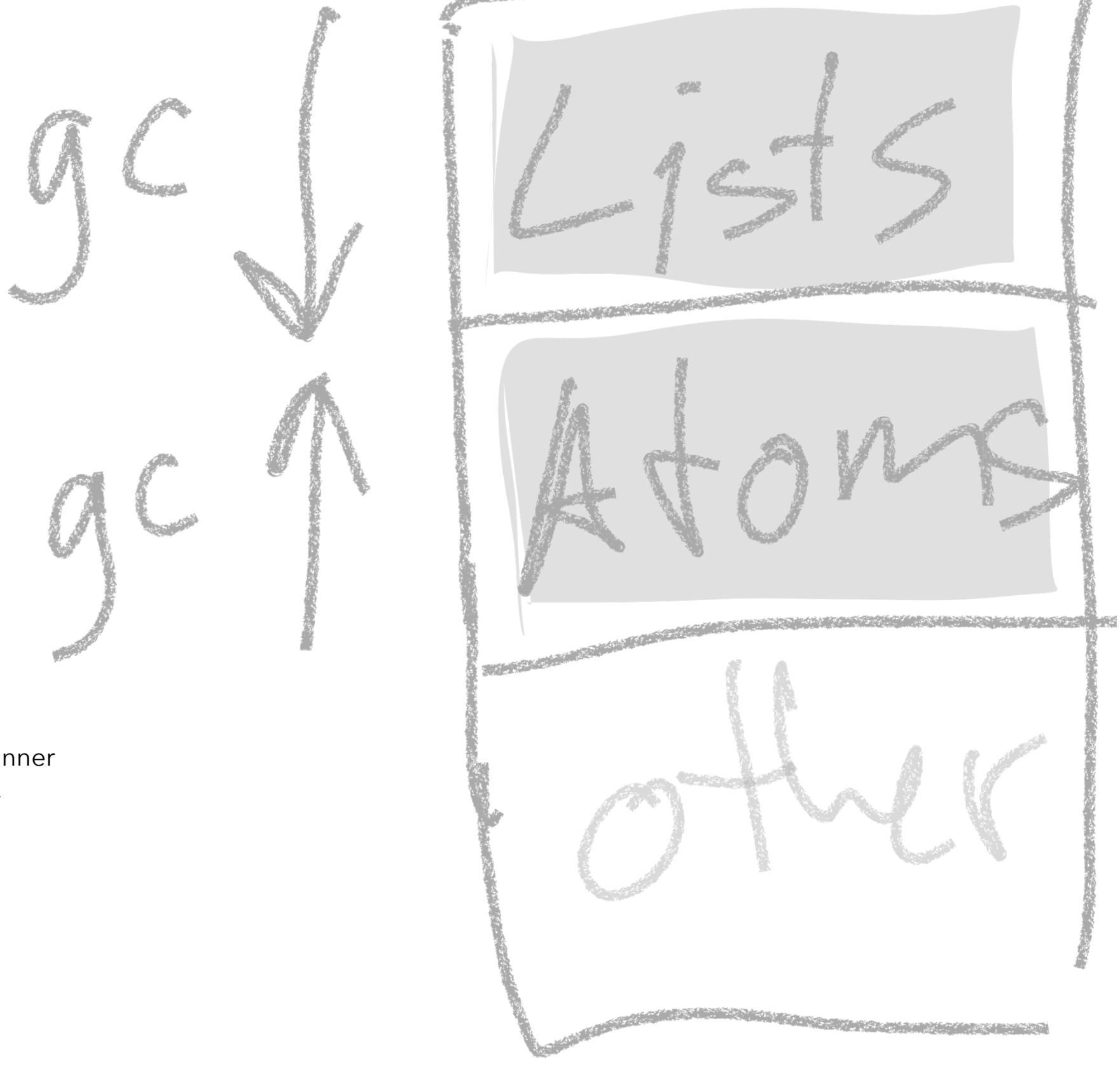
When you squint the right way at program *functions* you can see that the functions' parameters are on stacks. If a function creates temporary values, those values are placed on the stack, too.

A function's stack is wiped out when the function 'returns'. This operation is directly supported by the CPU hardware, and, is very efficient. Any temporaries created by the function are wiped out on 'return'. Parameters are wiped out, too, on 'return'. Only the return value remains.

In the past, the return value was always made to fit inside the CPUs' *registers*. When we started creating languages with larger, "non-scalar", return values, the return value couldn't always be put into registers, so various tricks were developed for holding the return values. These tricks are carefully managed by compilers.

Reclaiming List and Atom Cells

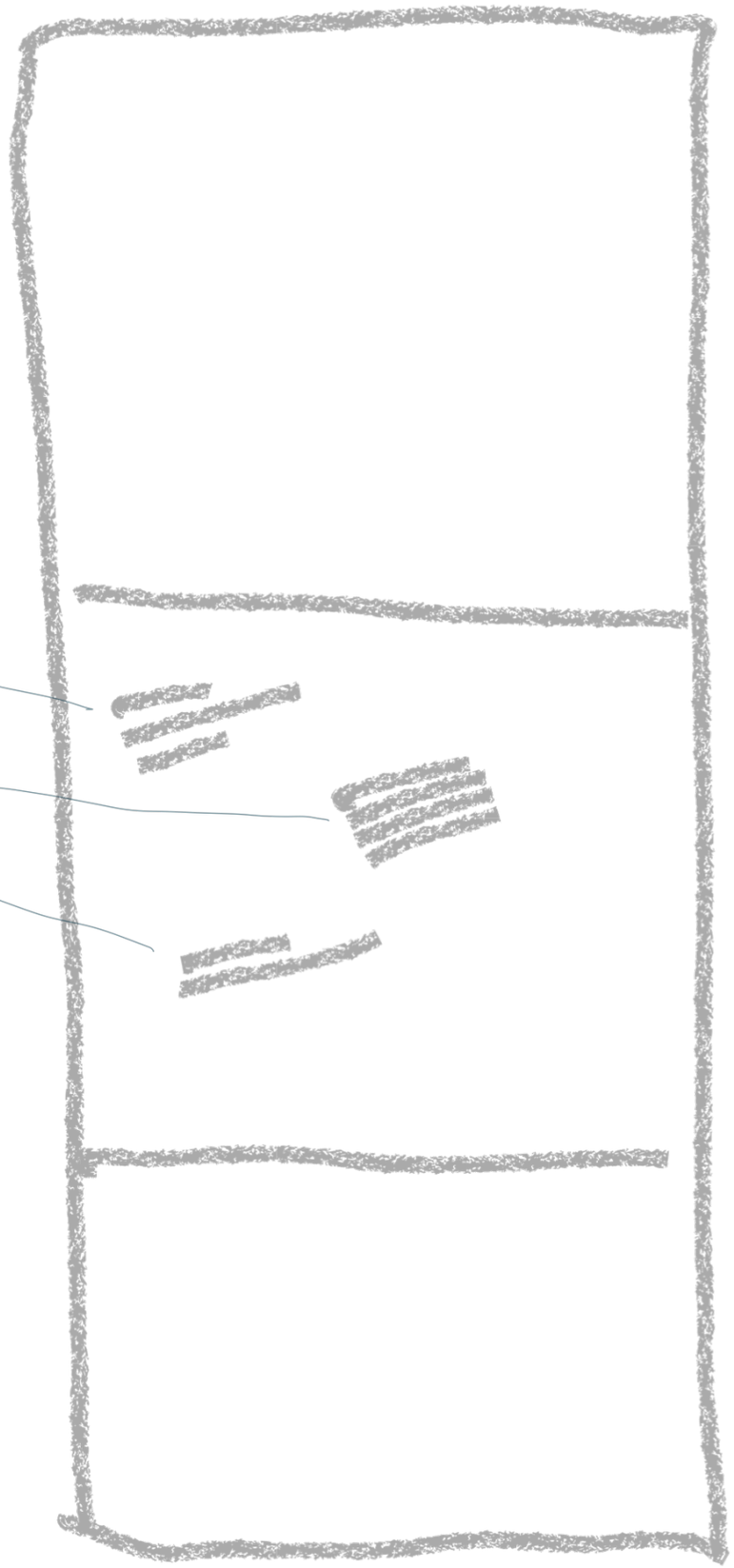
GC - Garbage Collection



Allocation in a strict stack-like manner (no mutation, no heap) allows for significantly simpler GC code.

Atom Dictionary

abc	
Hello	
Goodbye	



Text atoms are never duplicated.

On input, text atoms are hashed into a dictionary. In Lisp, text code is read in by “the reader” subroutine.

Actually, McCarthy used linear search instead of hashing, but we can do better now.

N.B. in this super-simple representation, numbers are not special and are simply “text atoms”. E.g. 42 is an atom with a 2-character name.

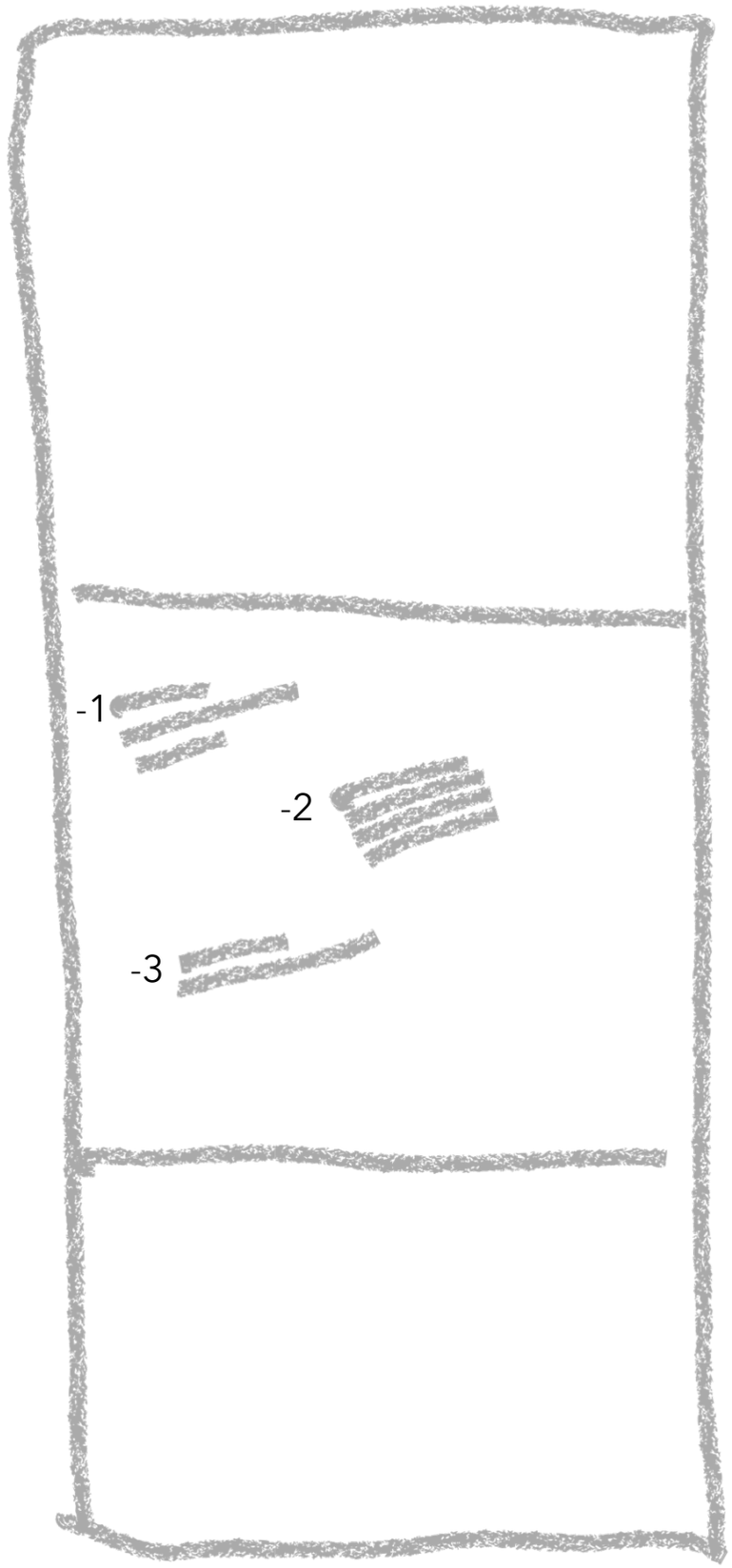
Atom Indices

abc	-1
Hello	-2
Goodbye	-3

Pointers are just indices.

We use -ve indices to represent Atoms.

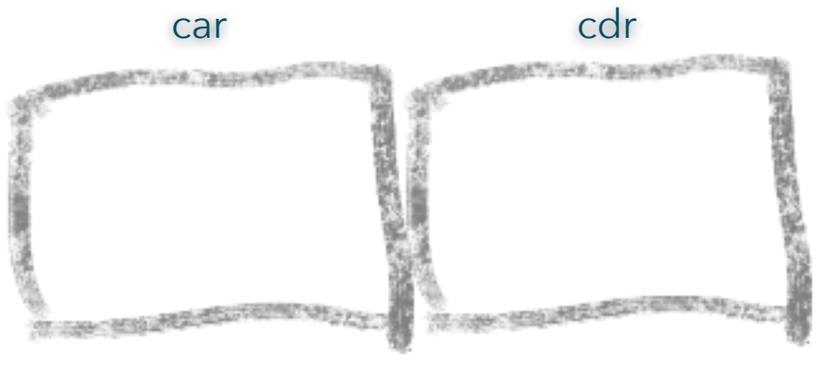
So, an index points to an atom if the value is < 0.



Cons cells

(Lists)

Cons Cell



printed as “(1 . 42)”

printed as “(1)”

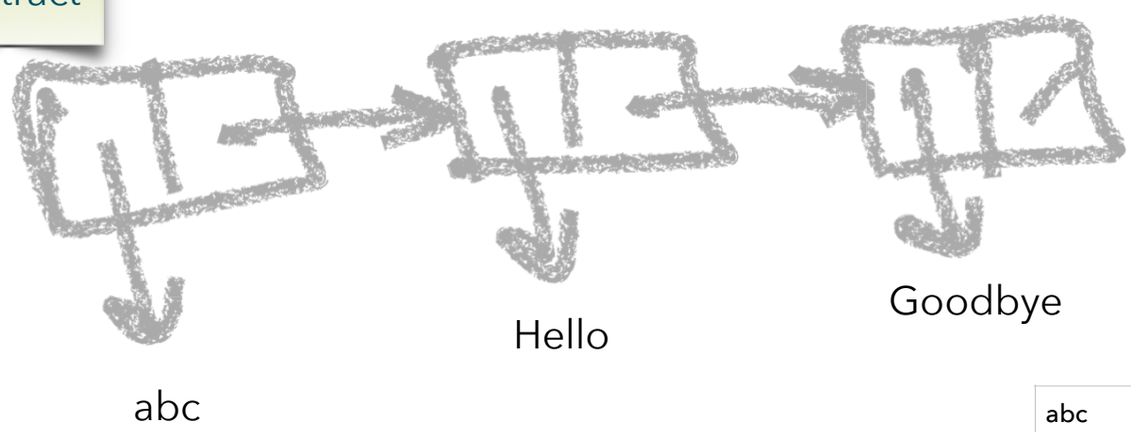
shorthand for “(1 . nil)”

Lists

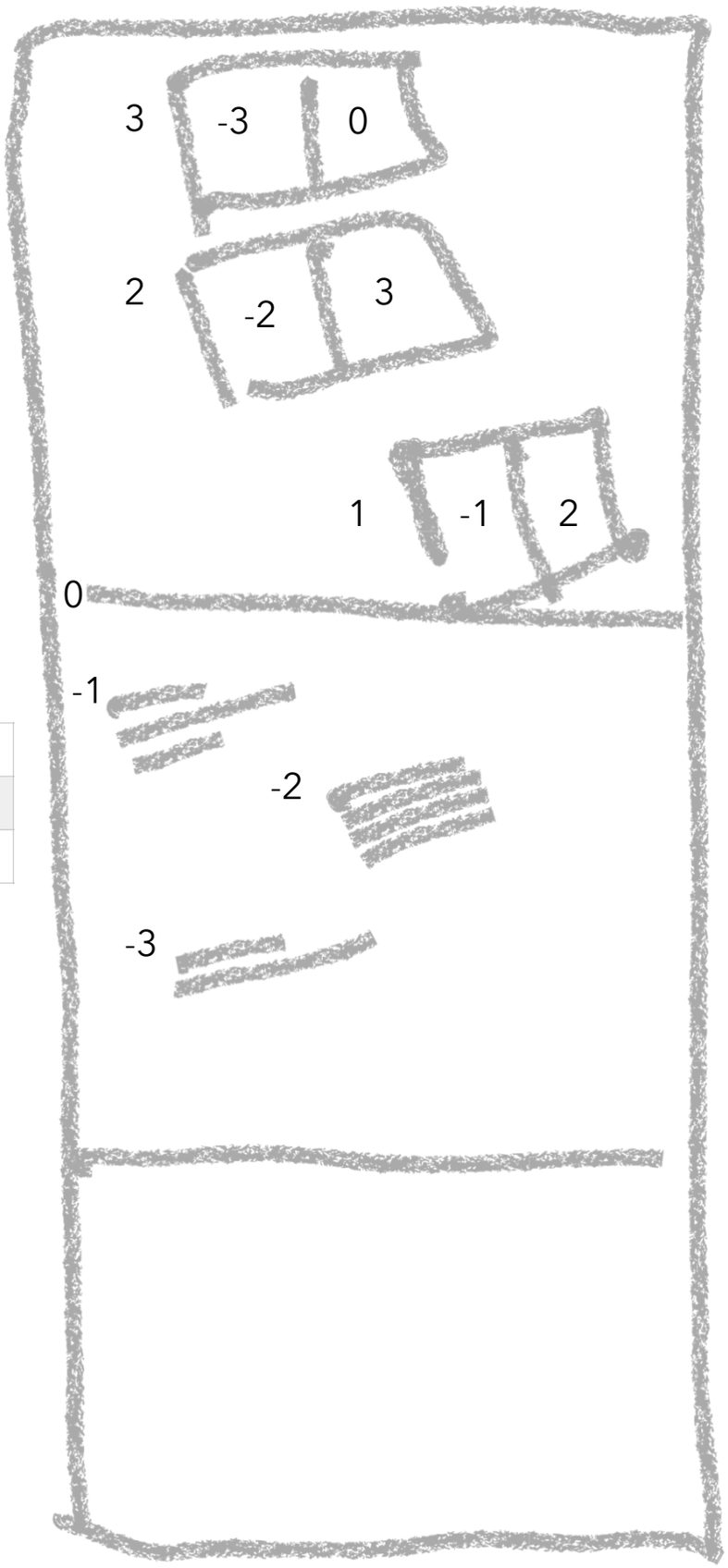
printed

(abc Hello Goodbye)

abstract



abc	-1
Hello	-2
Goodbye	-3



We use +ve indices to represent Atoms.

We reserve index 0 for a special use. CPU hardware makes it very efficient to use and to test for 0.

So, an index points to a List if the value is > 0.

And, if the value is = 0, we treat it as a special value.