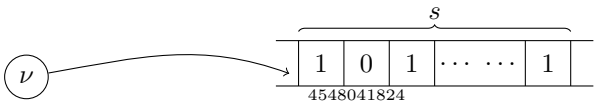
NAMES AND VALUES

ERIC MARTIN

We talk of 0, 1, 2, 3.14 as values, but 0, 1, 2, 3.14 are also names of values; we cannot refer to values without naming them. In the computing world, a value is "stored" in memory as bits. How a sequence of bits relates to a value is determined by a particular coding and decoding scheme. For instance, the scheme that represents nonnegative integers in base 2 over 8 bits can let us "store" the value (named) 9 as 00001001 somewhere in memory. Neither a name nor a sequence of bits is a value; to refer to the name of a value and to the storage of a value somewhere in memory, let us talk about *v-name* and *v-storage*, respectively, without investigating further what the value itself actually is...

A v-name can have many *occurrences*: in the expression `X = 2+2`, there are two occurrences of the v-name `2`. As much as "value" is an abstract notion, occurrences of v-names and v-storages can be concretely located in the text of a program and in computer memory, respectively.

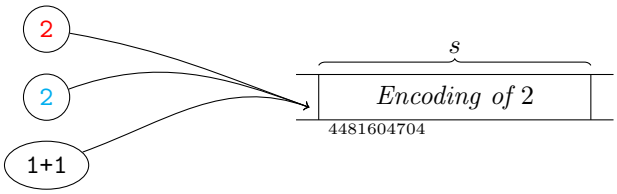
An occurrence of a v-name is always *referring* to a v-storage that determines some value according to an implicit coding and decoding scheme. The following diagram is fundamental; it indicates that an occurrence ν of a v-name is always, necessarily, associated with, referencing, a v-storage s, that is, a sequence of bits stored at a particular memory address; a particular coding and decoding scheme is implicit and determines the value that is named via ν and stored as s.



There are two kinds of v-names, *constant expressions* and *variable expressions*, with as particular cases, *constants* and *variables*, respectively.

For instance, `2` and `1+1` are two constant expressions, the first of which is a constant. In the following code, the two occurrences of `2` and the unique occurrence of `1+1` are seen to refer to the same v-storage, with the expectation that it "stores" the value 2 thanks to an appropriate coding and decoding scheme.

```
>>> id(2), id(2), id(1+1)
(4481604704, 4481604704, 4481604704)
```

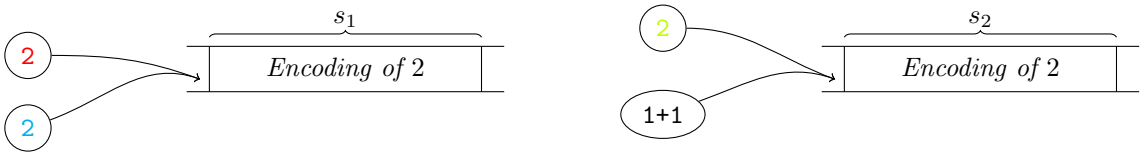


Date: Session 1, 2016.

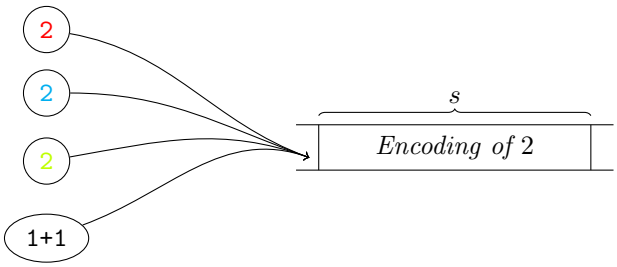
The code

```
>>> 2 is 2, 2 is 1+1
(True, True)
```

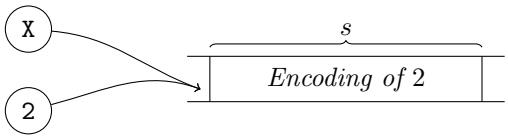
allows us to draw the diagrams



and suspect that s_1 and s_2 are actually identical, so that the picture could actually be:



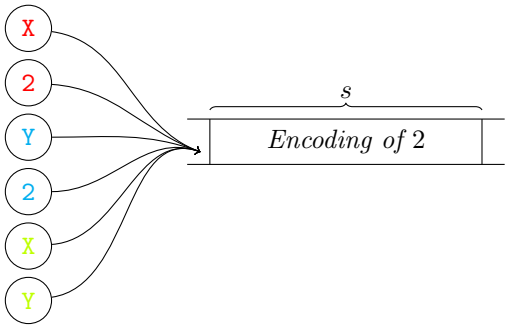
Whereas the value named by a constant expression is fully determined by the syntactic rules and semantics of the language, the value named by a variable is arbitrary. With the assignment operator `=`, we let a variable refer to a v-storage. Thanks to the code `X = 2`, we are somehow letting `X` be a v-name for the value 2, but more precisely, and fundamentally, we are letting `X` be another v-name for the value (named) 2 by letting `X` reference the same v-storage as the occurrence of the v-name `2` to the right of the assignment operator does, and this for at least as long as we do not change the value of `X`:



So from the code

```
>>> X = 2; Y = 2
>>> X is Y
True
```

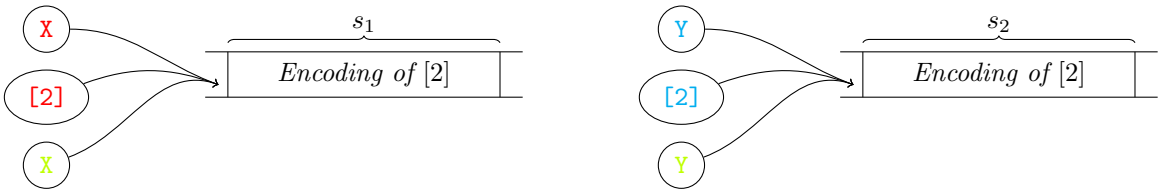
we are justified in drawing the diagram



whereas from the code

```
>>> X = [2]; Y = [2]
>>> X is Y
False
```

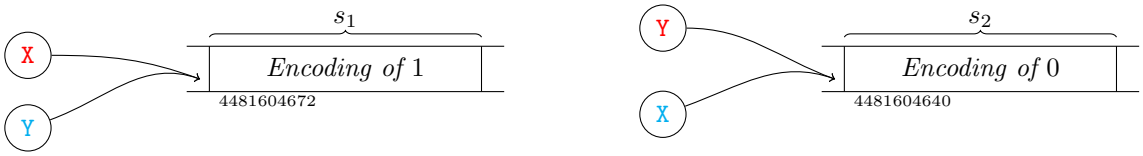
we are justified in drawing the diagram



The code

```
>>> X = 1; Y = 0; id(X), id(Y)
(4481604672, 4481604640)
>>> X = 0; Y = 1; id(X), id(Y)
(4481604640, 4481604672)
```

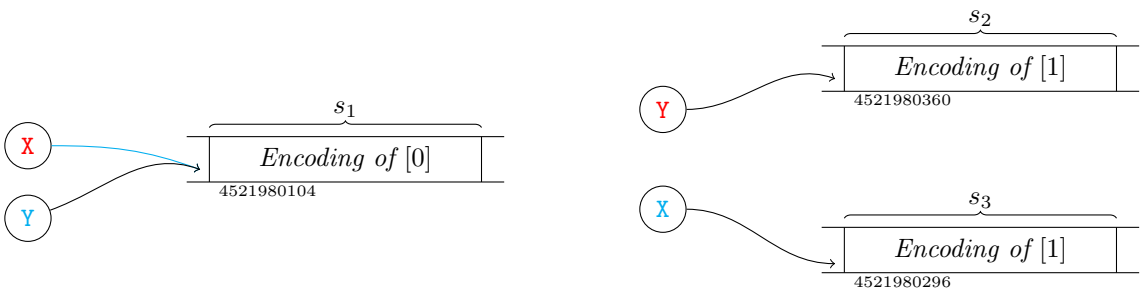
leads to the diagram:



The code

```
>>> X = [0]; Y = [1]; id(X), id(Y)
(4521980104, 4521980360)
>>> X = [1]; Y = [0]; id(X), id(Y)
(4521980296, 4521980104)
```

leads to the diagram:



The following code involves 5 occurrences of names for the same value and 3 v-storages.

```
>>> id([0, 1]); id([0, 1]); id([0, 1]); id([0, 1]); id([0, 1])
4511633480
4486339464
4486339464
4486339464
4511631496
```

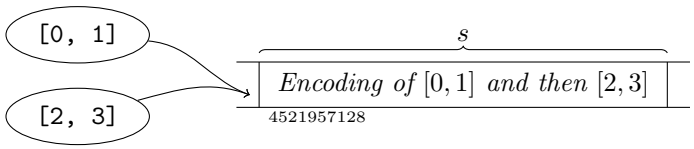
In reference to the four occurrences of `[0, 1]`, either 2 or 3 v-storages are involved with the following code.

```
>>> id([0, 1]) is id([0, 1])
False
>>> id([0, 1]), id([0, 1])
(4521980232, 4521980232)
```

With the following code

```
>>> id([0, 1]), id([2, 3])
(4521957128, 4521957128)
```

we see one value "take the place" of another one in memory:



There are more occurrences of v-names than in our code:

```
>>> import sys
>>> sys.getrefcount(0), sys.getrefcount(1), sys.getrefcount(2), sys.getrefcount(3)
(1459, 1670, 457, 172)
>>> sys.getrefcount(4), sys.getrefcount(5), sys.getrefcount(6), sys.getrefcount(7)
(243, 104, 89, 51)
>>> sys.getrefcount(8), sys.getrefcount(9), sys.getrefcount(254), sys.getrefcount(255)
(187, 60, 4, 18)
>>> sys.getrefcount(256), sys.getrefcount(257), sys.getrefcount(258), sys.getrefcount(259)
(59, 2, 2, 2)
```

The following code provides another illustration of the transition between integers smaller than 257 and integers at least equal to 257.

```
>>> X = 255
>>> X is 255
True
>>> X = 256
>>> X is 256
True
>>> X = 257
>>> X is 257
False
>>> X = 258
>>> X is 258
False
```

But the following code does not.

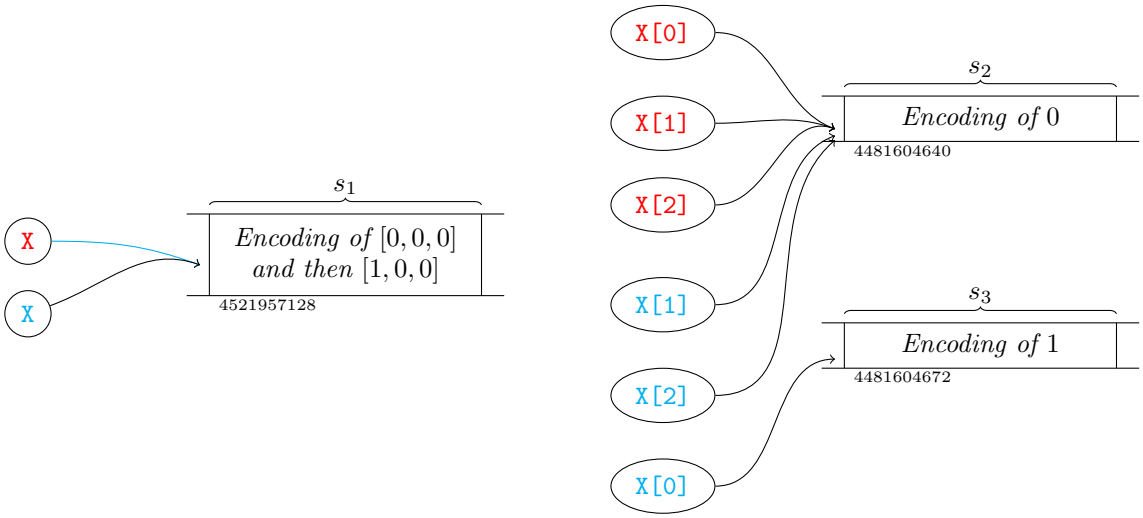
```
>>> X = 255; X is 255
True
>>> X = 256; X is 256
True
>>> X = 257; X is 257
True
>>> X = 258; X is 258
True
```

In the following code, each of the 4 occurrences of `[0, 1]` refers to a different v-storage of the value $[0, 1]$, and each of those v-storages is minimally referred to.

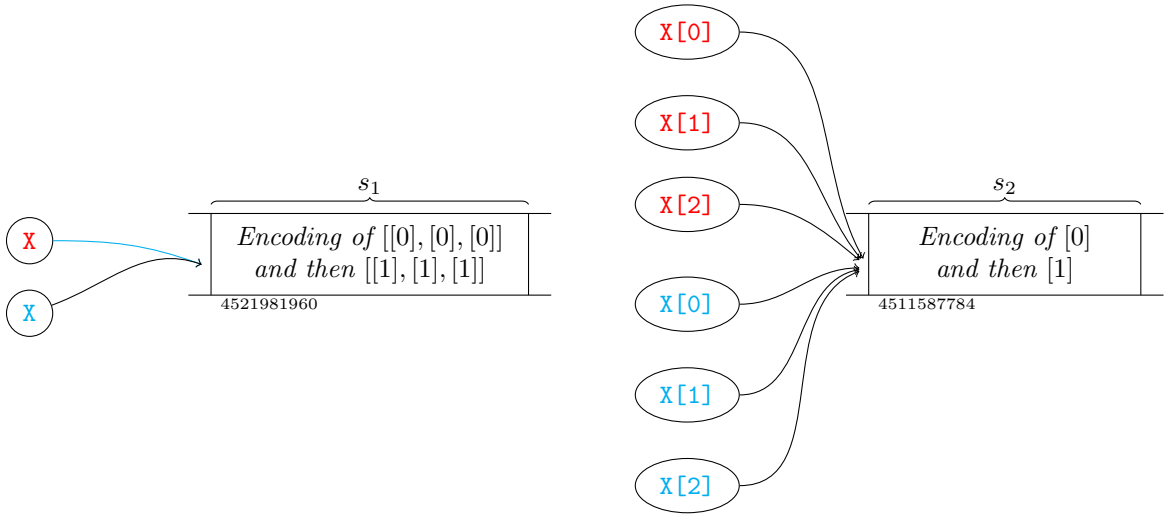
```
>>> X = [0, 1]; Y = [0, 1]; Z = [0, 1]
>>> X is Y, Y is Z, X is Z
(False, False, False)
>>> sys.getrefcount([0, 1])
1
>>> sys.getrefcount(X), sys.getrefcount(Y), sys.getrefcount(Z)
(2, 2, 2)
```

The following three pieces of code deal with a list of integers and lists of lists of integers.

```
>>> X = [0] * 3; X
[0, 0, 0]
>>> id(X), id(X[0]), id(X[1]), id(X[2])
(4521957128, 4481604640, 4481604640, 4481604640)
>>> X[0] = 1; X
[1, 0, 0]
>>> id(X), id(X[0]), id(X[1]), id(X[2])
(4521957128, 4481604672, 4481604640, 4481604640)
```




```
>>> X = [[0]] * 3; X
[[0], [0], [0]]
>>> id(X), id(X[0]), id(X[1]), id(X[2])
(4521981960, 4511587784, 4511587784, 4511587784)
>>> X[0][0] = 1; X
[[1], [1], [1]]
>>> id(X), id(X[0]), id(X[1]), id(X[2])
(4521981960, 4511587784, 4511587784, 4511587784)
```



```
>>> X = [[0], [0], [0]]; X
[[0], [0], [0]]
>>> id(X), id(X[0]), id(X[1]), id(X[2])
(4521980424, 4521980104, 4521981832, 4521980680)
>>> X[0][0] = 1; X
[[1], [0], [0]]
>>> id(X), id(X[0]), id(X[1]), id(X[2])
(4521980424, 4521980104, 4521981832, 4521980680)
```