Atoms, Pairs, and Pointers

An atom isn't defined in standard Scheme, but it is a useful name to give any data construct that is a single value (symbols, numbers, strings, and boolean values). Pairs and vectors would not be atoms with this definition because they have ways of holding two or more elements. It's important to know that inside Scheme, these single value entities are considered to be equivalent if they look the same. Atoms are drawn with a box around the data they represent. (Sometimes they are drawn without the box, but then they can be easily confused with variable names.)

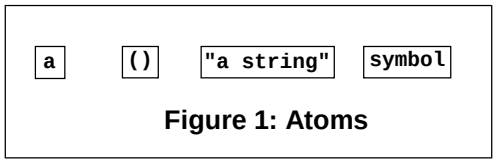


Figure 1: Atoms

A pair is a data type that allows us to point to two different objects (an object is any type of data in Scheme). The constructor for a pair, *cons*, takes two arguments, the first argument can be retrieved from the created pair by the selector *car*, and the second can be retrieved by the selector *cdr*. It's important to notice that the pair does not actually contain the two objects inside itself; the pair only *points* to the different objects, just like a variable name isn't actually the object it represents, but *points* to an object. This subtlety becomes important when we talk about mutators. A pointer is represented by an arrow pointing from the representation to the object being represented. A pair is drawn as two boxes attached to each other with one arrow pointing out of each box to the object it represents. (One problem with this picture is that you can't really tell which side is which, the *car* or the *cdr*.)

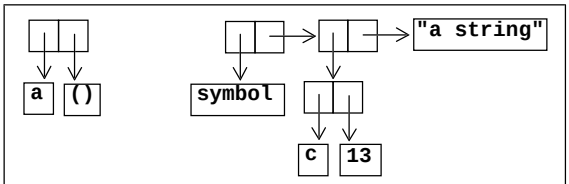


Figure 2: Pairs and atoms

When we draw a name connected to an object, it's similar to to a pair, except we don't have a box, but a name. The name is made with an arrow pointing from it to the object being represented.

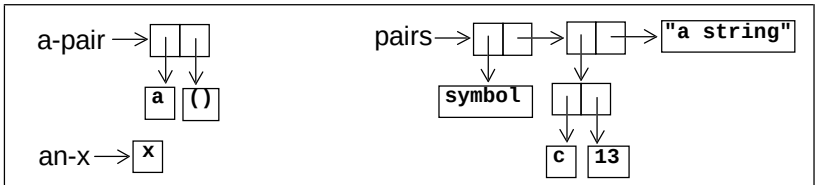


Figure 3: Named pairs and atoms

Mutators

When drawing box-n-pointer diagrams, mutators are those procedures which change pointers in Scheme objects. Since atoms don't have pointers, it should be clear that you can't mutate them. Names and pairs do have pointers, so you can mutate them (later you'll learn about vectors which also contain pointers, so you can mutate them). The standard way of designating a mutator in scheme is by putting an exclamation point after the name.

The mutator for names is *set!*. When you change a pointer, you can do one of two things, erase the old arrow and draw a new arrow to the new object, or put two slashes over the old arrow to signify it has been disconnected (or cutoff) and then draw a new arrow to the new object. I will use the slash method. One thing to remember when connecting pointers to atoms is that there only exists (conceptually) one instance of each unique atom in Scheme, so the correct way to draw pointers a duplicated atom is to only draw the atom once and draw all of the pointers to it. Whenever you create a non-atom, a new object is created, so a new one must be drawn even if it looks identical to one already in the environment.

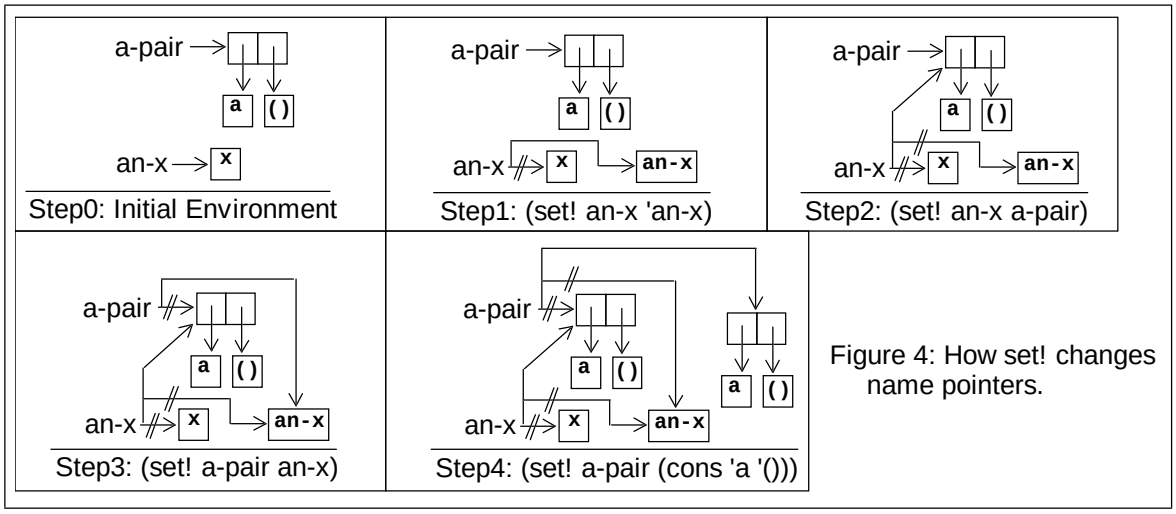


Figure 4: How set! changes name pointers.

To change the pointers associated with pairs, there are the two commands *set-car!* and *set-cdr!*. These change pointers just like *set!* except their argument is a pair and not a name.

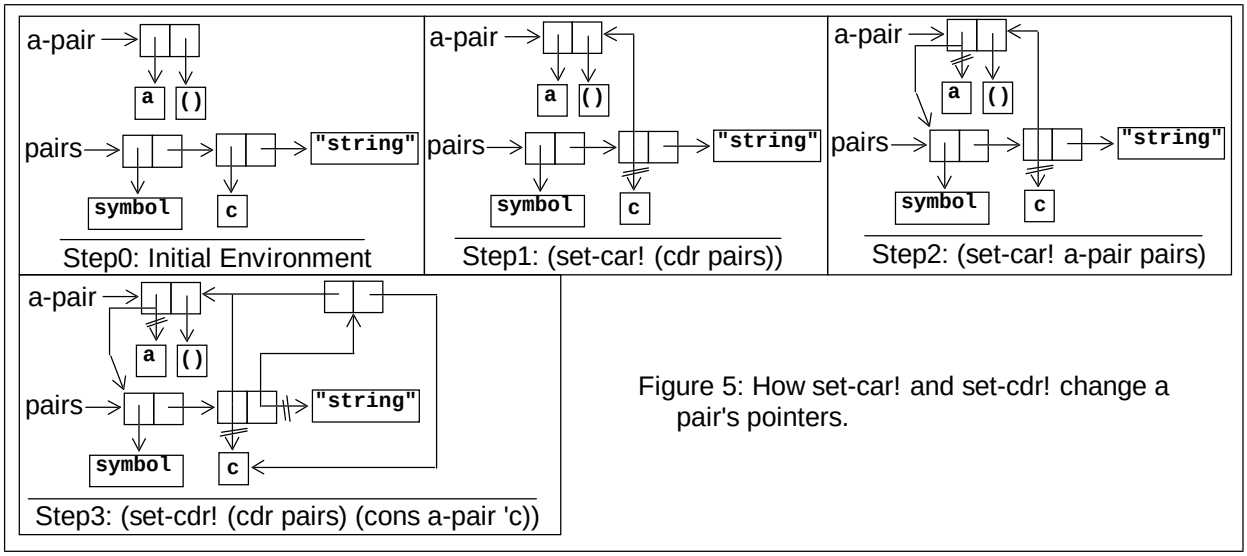


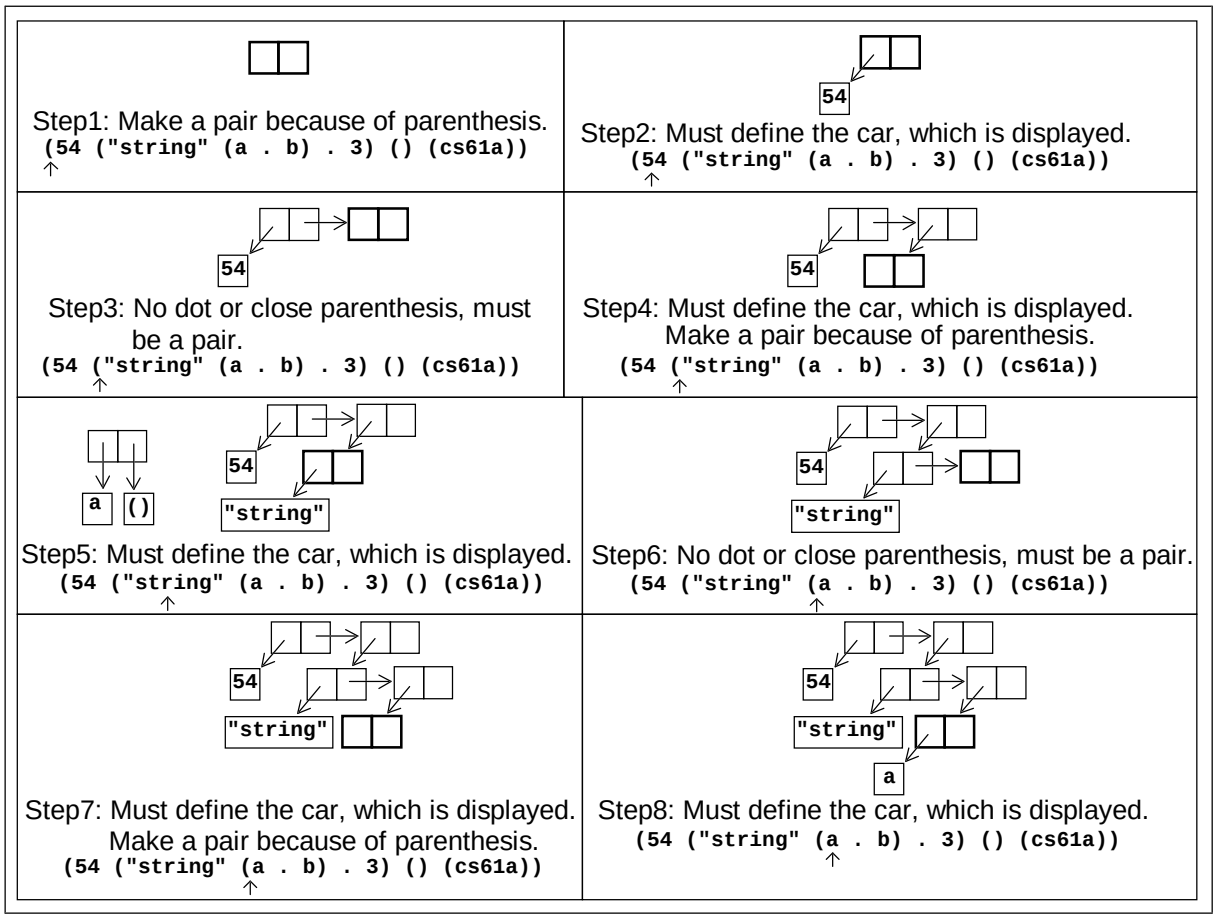
Figure 5: How set-car! and set-cdr! change a pair's pointers.

Interpreting Scheme's Text Representation of Pairs

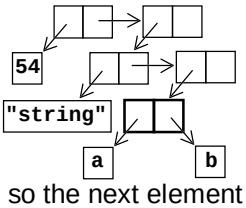
When you display a structure made out of pairs in Scheme, it makes a text representation and prints it on your screen. Although what is displayed doesn't look like pairs, by following a few simple rules, you can make a box-n-pointer diagram out of any text representation. In fact, when you use quote (or "`'`") before one of these text representations, Scheme interprets it and builds the structure in its own memory.

Rule #1: Pairs are surrounded by parenthesis, unless they are pointed to by the cdr of a previous pair.
Rule #2: All objects are displayed with their normal text representation when they are in the car of a pair.
Rule #3: All objects, except pairs or the empty list[1], are displayed after a dot (or period) when they are in the cdr of a pair.
Rule #4: The empty list is omitted from being drawn if it is in the cdr of a pair.

As an example, suppose we were given the following text representation of a bunch of pairs and atoms: `(54 ("string" (a . b) . 3) () (cs61a))`. To show what part of the expression is being evaluated, look at the arrow.

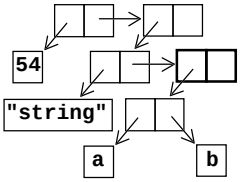


[1] An empty list is displayed as parenthesis without anything inside "`()`" and it is an atom.



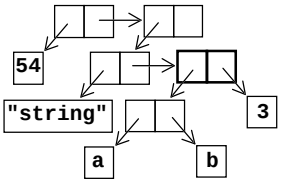
Step9: It's a dot, so the next element is not a pair, and is displayed.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



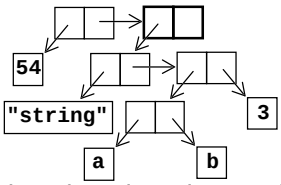
Step10: We filled up the cdr and passed the pair's corresponding close parenthesis.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



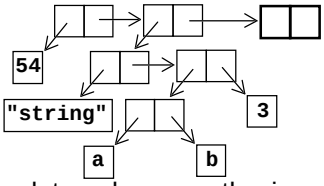
Step11: It's a dot, so the next element is not a pair, and is displayed.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



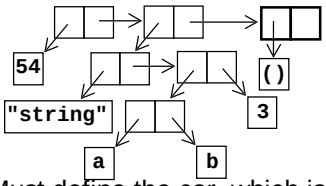
Step12: We filled up the cdr and passed the pair's corresponding close parenthesis.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



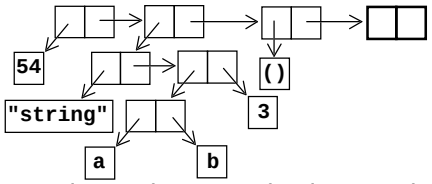
Step13: No dot or close parethesis, must be a pair.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



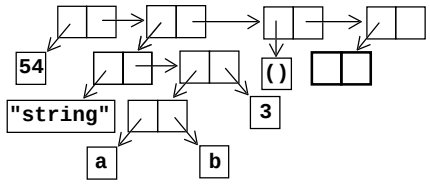
Step14: Must define the car, which is displayed. Be careful, this is the empty list not a pair.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



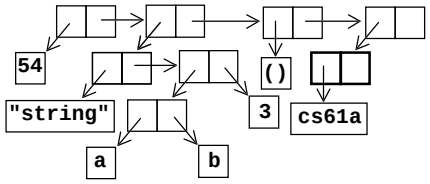
Step15: No dot or close parethesis, must be a pair.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



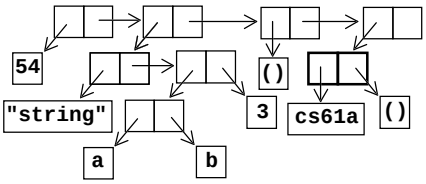
Step16: Must define the car, which is displayed. Make a pair because of parenthesis.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



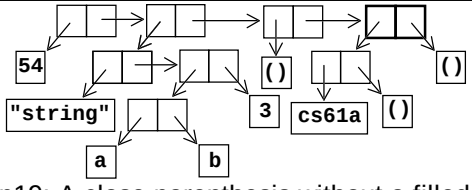
Step17: Must define the car, which is displayed.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



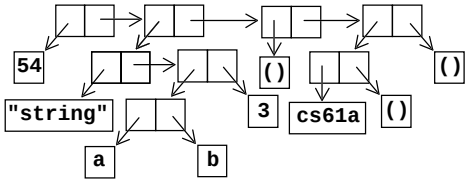
Step18: A close parenthesis without a filled cdr. Must be an empty list. Done with pair.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



Step19: A close parenthesis without a filled cdr. Must be an empty list. Done with pair.

```
(54 ("string" (a . b) . 3) () (cs61a))
```



Step20: We're done!

```
(54 ("string" (a . b) . 3) () (cs61a))
```