Lambda Calculus Pairs

In the following la stands for greek lambda.

1. So far we are limited to functions of one variable: our syntax does not allow anything else. We have been stacking function calls to implement multiple arguments. Curry proved that m-ary functions can be computed by m-deep stacked functions and vice-versa, and thus we shall soon introduce multi-argument functions into our syntax. Meanwhile it may help to impose our intentions on the interpretation of functions. Recall:

def make-pair = la first. la second. la func. ((func first) second)

Does this mean make-pair is a 3-argument function? Not really. Here's a real 3-arg function, which returns the "middle" (second) argument.

def middle = la first. la second. la third. (second)

What's the difference? Not the names (func not same as third). In fact nothing differs but *how we want to think about it!*

Consider applying make-pair and middle to arguments A, B, C. First consider middle.

(((la first. la second. la third. (second) A)B)C) == >
((la second. la third. (second) B)C)== >
(la third. (B) C)== >

Now at this point we do NOT think "oh yeah, middle has created a function that requires another argument" (though that's true). No, we think
"middle is still looking for its third argument". We think of it as a 3-ary function not completely invoked. But…

With a pair, analogously, after sucking up two arguments we have

(la func. (func first ) second)

And this IS interesting! If first and second have some sort of semantics like first-element and rest-of-list, or then-value and else-value, then it's like at this point we have a little 'data structure' or 'function structure' that's ready and waiting to have a function applied to it. That's a useful gizmo. So usually a pair has its first two arguments 'frozen in', 'sucked up' 'already bound', and is now sitting there with its tongue hanging out saying "what do I do with these?" Answer: apply yourself to some function!

Attention: Sometimes our descriptive language obscures what's going on. For instance: if a pair like the last example "is applied to" a function, Then on evaluation the function is "applied to" the two pair componentsl So when we apply the pairs to their relevant operations,, it sounds backwards but upon evaluation the operations are applied to the components of the pairs.