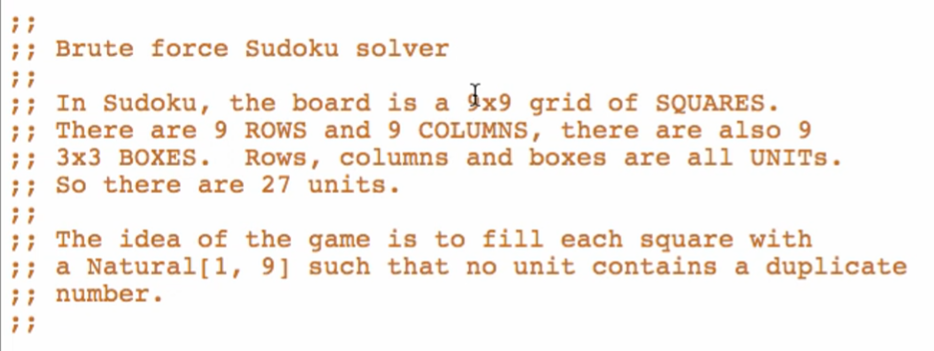
New require:

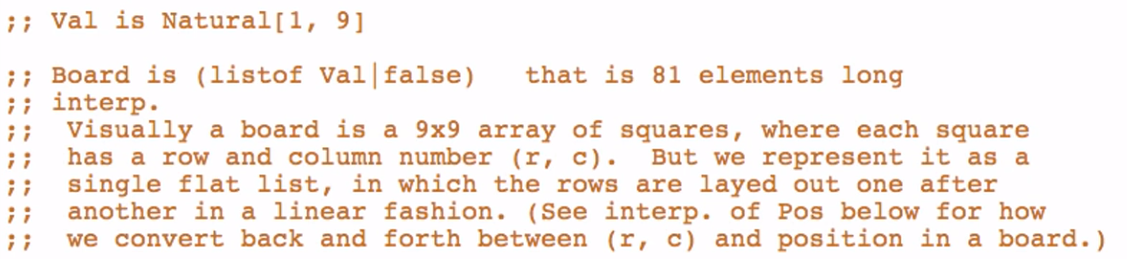


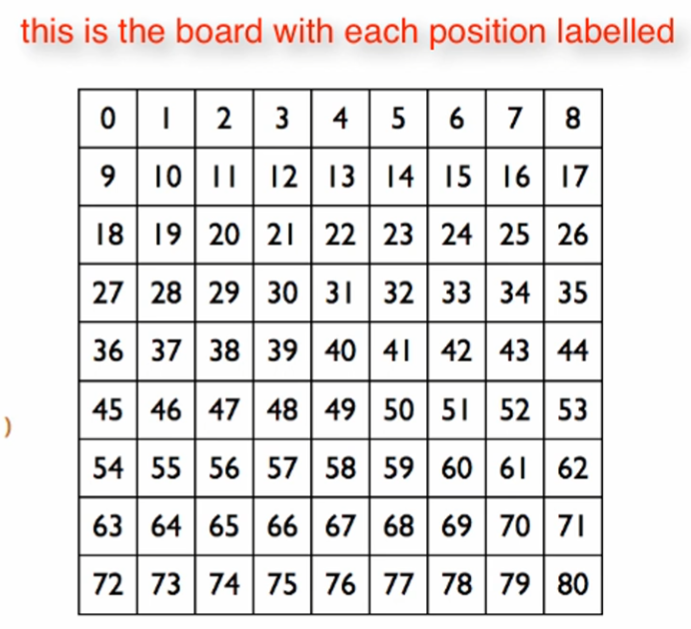
Textual form of the domain analysis:

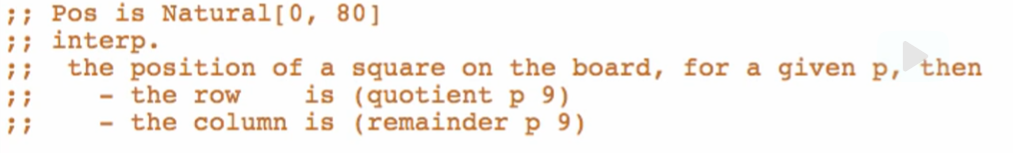


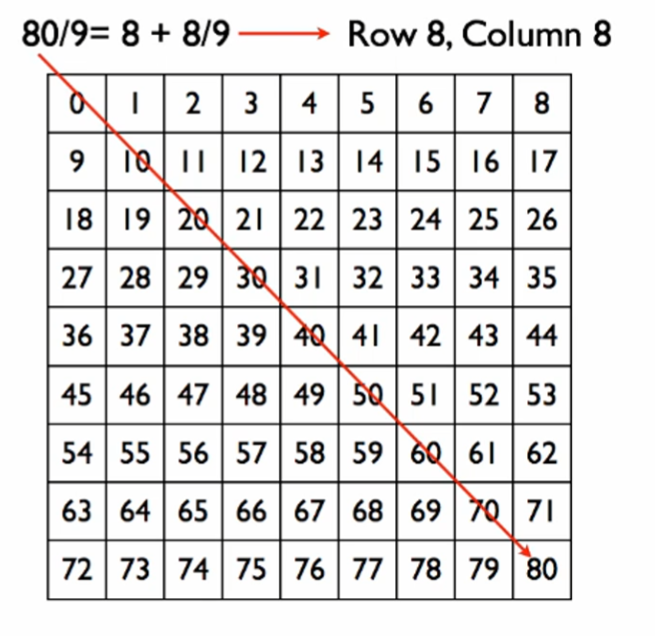
**Data Definitions**

Type comments





How to compute for the position?

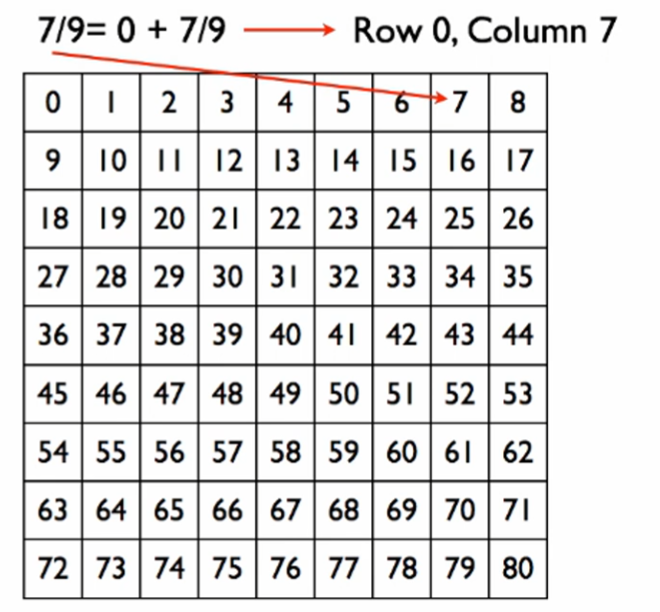


At position 80. Divide the position by 9. 80/9 = 8 + 8/9

The whole number = Row = 8

The remainder = Column = 8

Another example

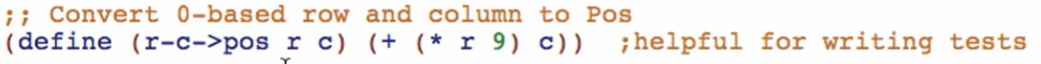


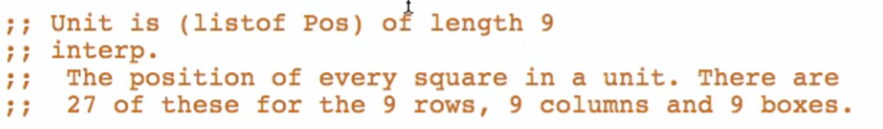
At position 7. Divide 7 by 9. 7/9 = 0 + 7/9

Row = 0

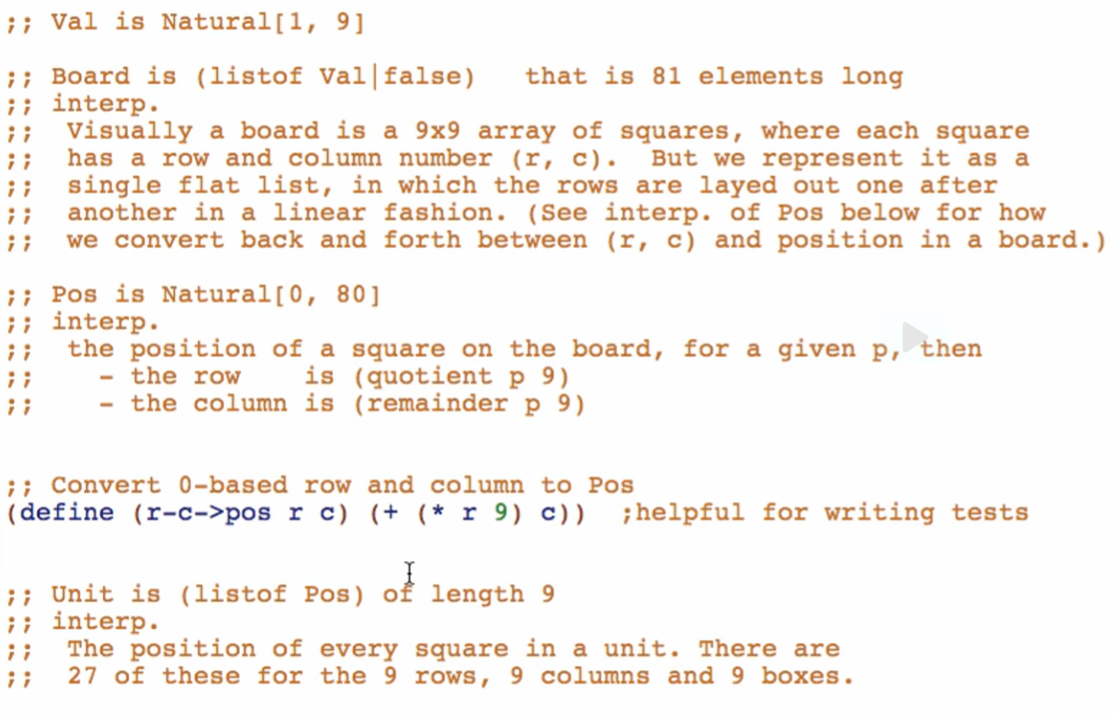
Column = 7

By these results, we can come up with a formula to convert a row and a column to a position





Overview of type definition

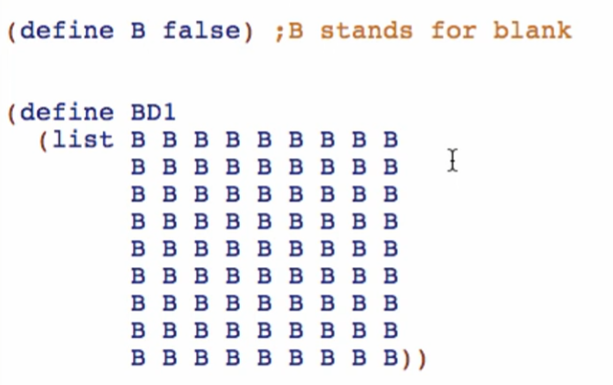


**Examples**

Val:

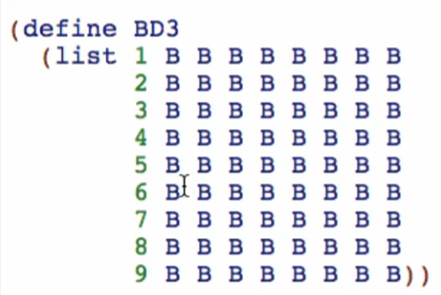


Board:

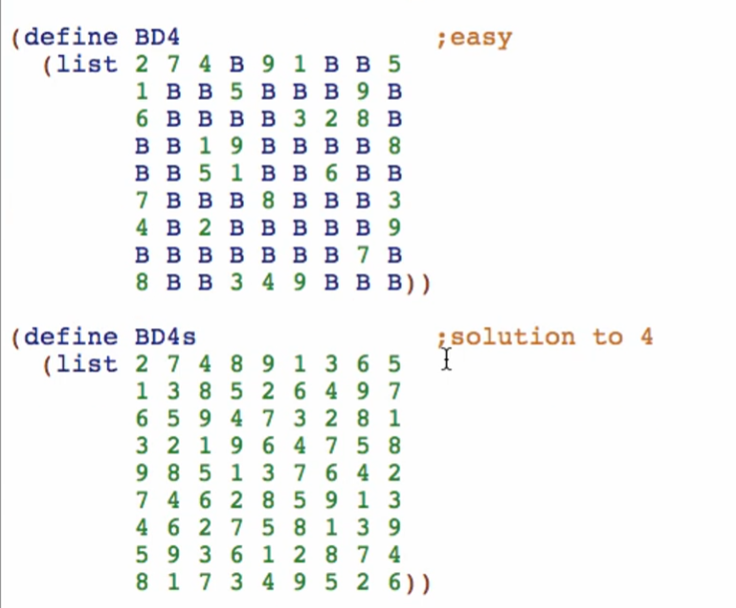


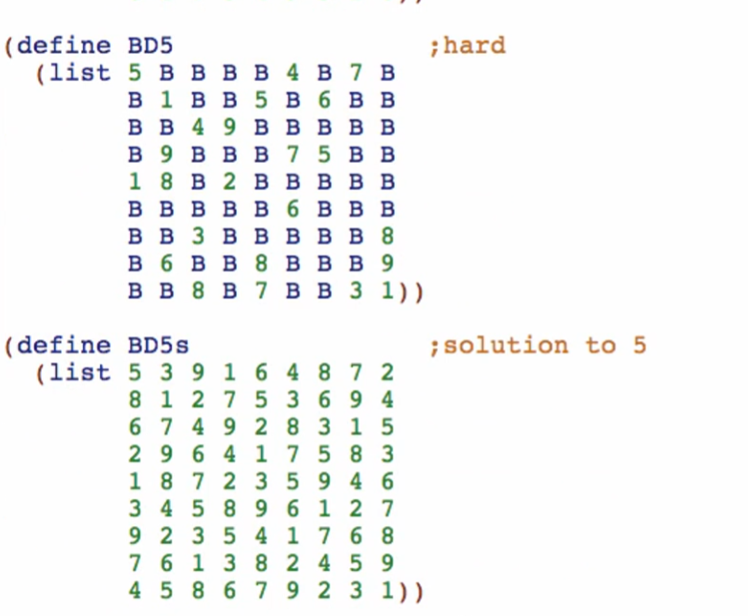
B is defined so we will have a simpler definition for Board. Not populating it with so many “false”





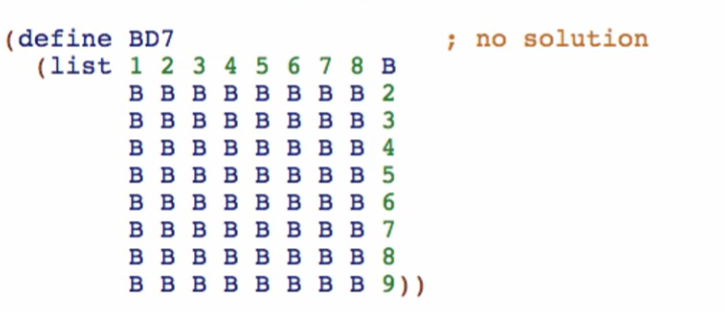
Real-life example of Board and their solutions





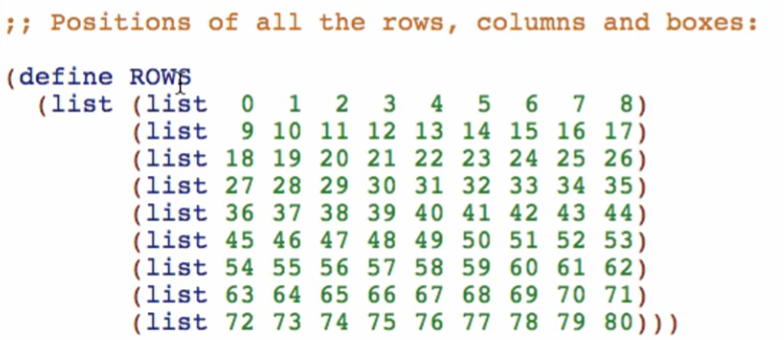


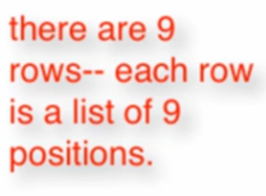
Board with no solution (failed)

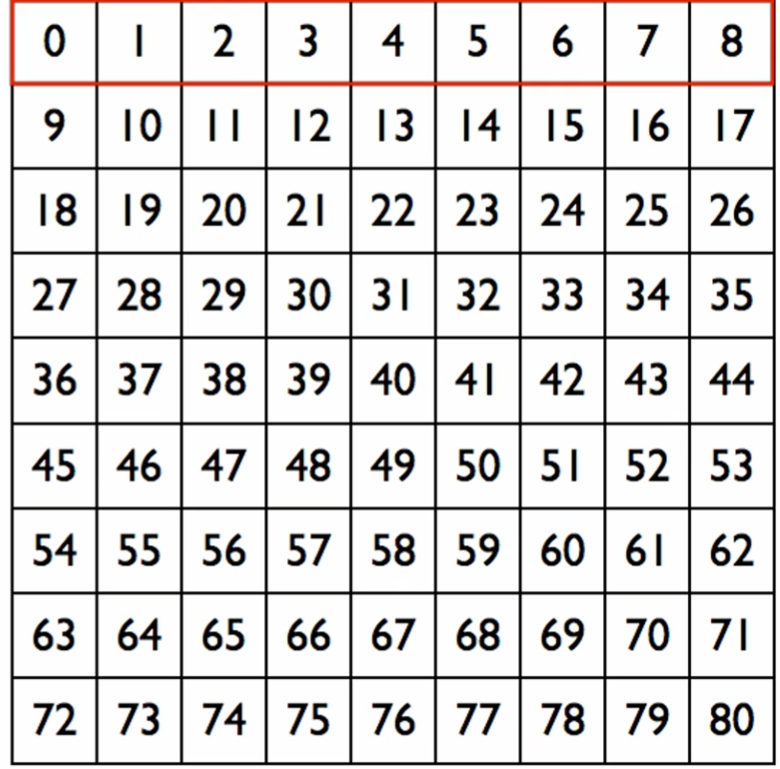


Can’t put the missing value because it will violate the rules

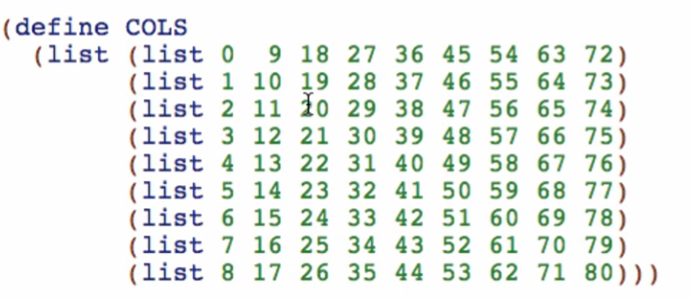
Examples of positions

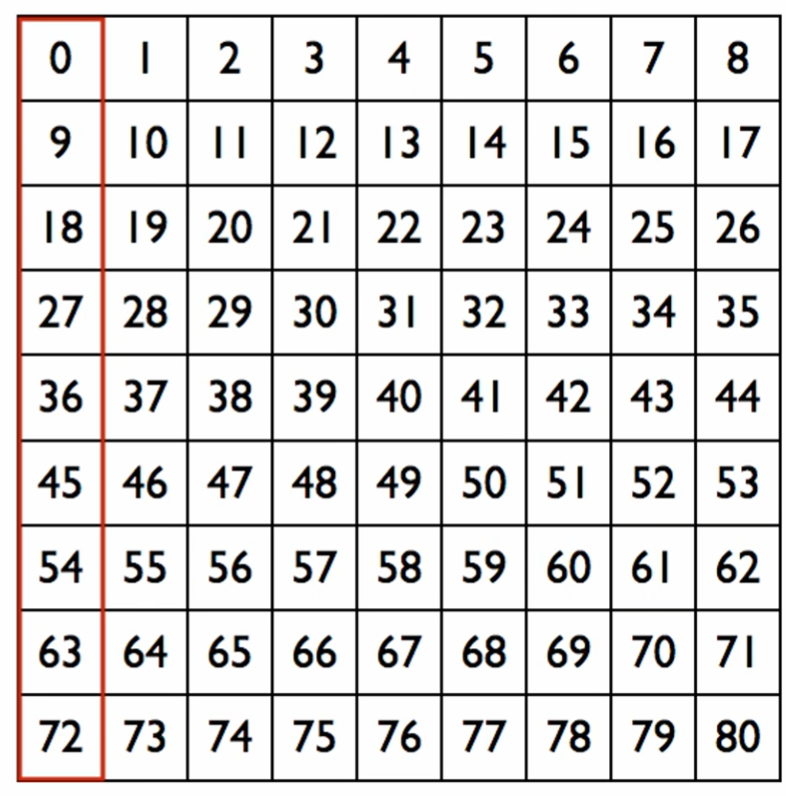




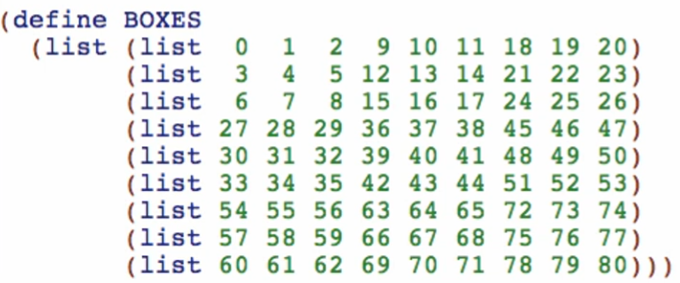


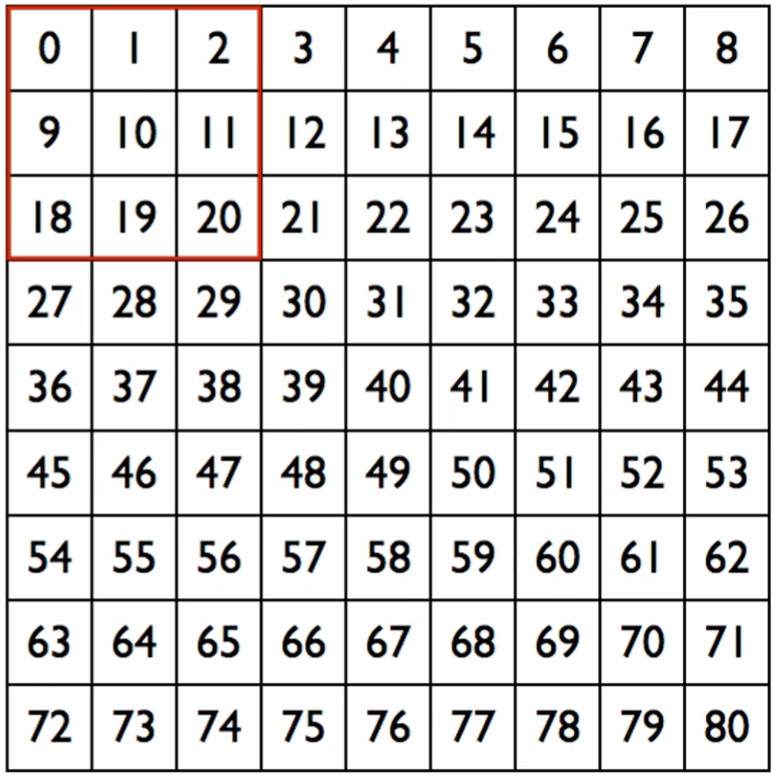
Examples for columns





Examples for boxes



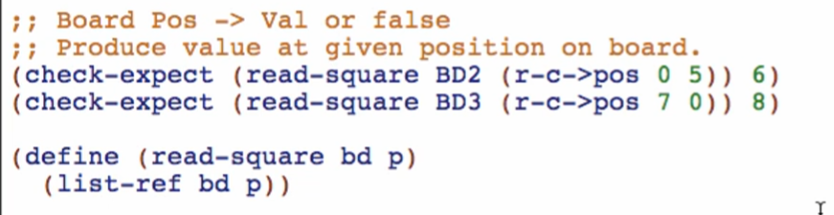


Examples for Units

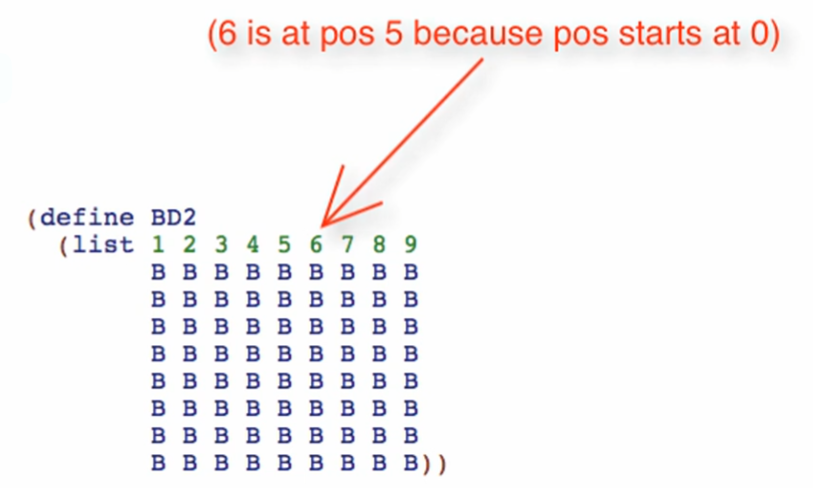


**Function Definitions**

**read-square Function**



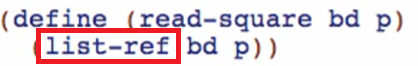
1ST check-expect:



2nd check-expect



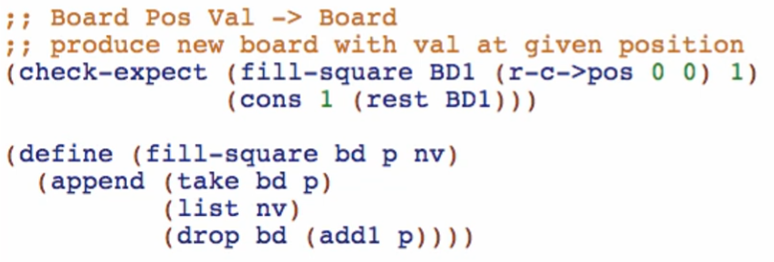
Body:



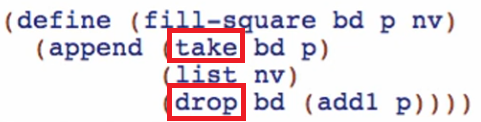
list-ref

* Returns the value of the board in a given position
* This is like array[index] = value

fill-square Function

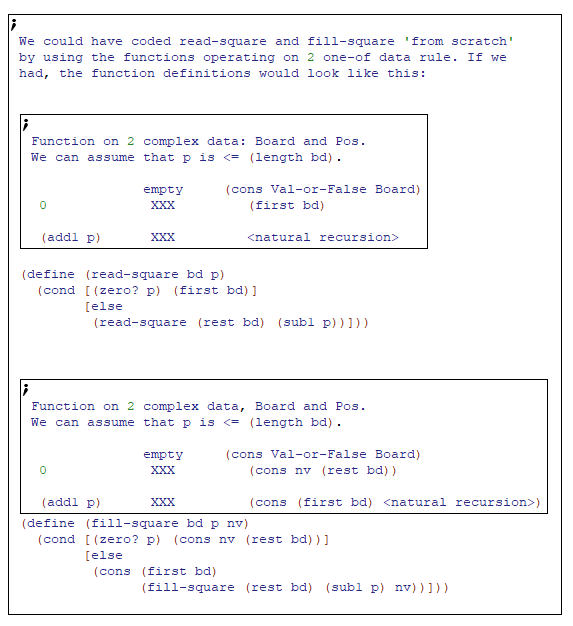


New primitives:



Notice that these are functions that consumes **Two One-of Types**

* If we don’t have the new primitives list-ref, take, and drop, we would’ve done the functions this way:



* But we already have it in our new language so it made our lives and coding simpler

Note: we use 0-based indexing