|  |  |
| --- | --- |
| TEA PRIMITIVE | SEMANTICS |
| R: | |  |  | | --- | --- | | NAME | Replace | | PURPOSE | Replace things with other things | | SYNTAX  & SEMANTICS | |  | | --- | | r: | | Replace all visible characters in AI with the EMPTY STRING and any whitespace except the NEW LINE character with the FULL-STOP character. | | r:REGEX:SUBSTR | | Return AI with the first section matching REGEX replaced by SUBSTR | | r!: | | The Inverse of r:; replaces all visible characters in AI with the SINGLE WHITESPACE, and all whitespace other than the NEW LINE character in AI with the FULL-STOP character. | | r!:REGEX:SUBSTR | | Return AI with ALL sections matching REGEX replaced by SUBSTR | | r\*: | | INERT | | r\*: vNAME:REGEX:SUBSTR | | Same as r:REGEX:SUBSTR,but operating on the string stored in the vault with the name vNAME instead of AI. With only vNAME, like r:, but operating on strings in the named vaults instead of AI | | r\*!: vNAME:REGEX:SUBSTR | | Same as r!:REGEX:SUBSTR,but operating on the string stored in the vault with the name vNAME instead of AI. | |  | | NOTES | String substitution as a core operation in most text processing, finds its main mechanics implemented using the R-command space in TEA. However, it should be noted that r: primitives aren’t the only kind that can perform string substitution in TEA. Some kinds of text replacement operations are possible using other TEA primitives such as g: that replaces whitespace with empty space, but also does some automatic replacements on punctuation with the g! variant.  Much can be accomplished with mere text substitution operations. The example below is one solution to compressing messages meant for SMS so as to keep the messages short, still meaningful, and thus save on SMS/data charges.  r:[Ww]h:w | r:[iI][nN][gG]:in | r:and:n  r:to:2 | r:for:4 | r:how:hw | r:ed:d  r:[ ]\*are[ ]: r | r:why:y | r:ou:u | r:[ ]be[ ]: b  Concerning controlling what to replace in a string, note that r:REGEX:STR replaces only the FIRST occurrence of REGEX in what is being processed with STR, but r!: replaces ALL occurrences of the pattern in what is being processed. We see this by the following two examples:  i!:I like this|r:[aeiou]:\_:  returns “I l\_:ke this”, while  i!:I like this|r!:[aeiou]:\_:  returns “I l\_:k\_: th\_:s”. Basically, the R!: form performs multiple substitutions, while R: only replaces the first occurrence of the target pattern.  Note that the **r:** and **r!:** primitives offer distinct, but related cryptographic utilities for using whitespace as information in machine-readable cryptograms. Essentially, they offer a strange but useful means to read “words between lines” so to say; systematically converting usual writing using visible text to a kind of invisible but readable writing system – writing using whitespace, an important primitive capability for some families of cryptography.  Also, note that r: and r!: work very differently, though somewhat similarly. The first eliminates visible characters and recognizes spaces and new lines, while r!: recognizes visible characters and new lines, but simplifies reading white-space. Assuming we have the following simple text as the input:  Myself should tell  You O my Lord.  I trust You Know Me.  We can then appreciate the “Braillish” projections of it with r: and r!: as seen below:  i!:{Myself should tell  You O my Lord.  I trust You Know Me.} | r:  Should return  ..  ...  ....  While  i!:{Myself should tell  You O my Lord.  I trust You Know Me.} | r!:  Returns what we see in the test screenshot below  **CLITTTT|< 13:42:43 $>\* cat tests/test\_r2.tea | tttt**  **. .**  **. . .**  **. . . .**  Clearly, we see that these two primitives serve a role very hard to replace with anything else! |  |  | |