|  |  |
| --- | --- |
| TEA PRIMITIVE | SEMANTICS |
| U: | |  |  | | --- | --- | | NAME | Uniqueify | | PURPOSE | Compute the unique projection of things (effectively, their **modal sequence** statistic) | | SYNTAX  & SEMANTICS | |  | | --- | | u: | | Return AI reduced to only its UNIQUE WORDS, their order representing their relative frequency in AI, most frequent first, otherwise preserving order of first occurrence. | | u:STR | | Same as u:, but operating on the given string STR | | u!: | | Return AI reduced to only its UNIQUE CHARACTERS, their order representing their relative frequency in AI, most frequent first, otherwise preserving order of first occurrence. | | u!:STR | | Same as u!:, but operating on the given string STR | | u\*: vNAME | | Same as u:, but operating on the string stored in the vault with the name vNAME | | u\*!: vNAME | | Same as u!:, but operating on the string stored in the vault with the name vNAME | |  | | NOTES | Uniquefying things is critical and primitive in TEA. Outputs of the U-command space can help in making critical decisions about things just by looking at their ordered base elements. For example, it is a common fact that in the English language, the letter “e” occurs the most in most across words. Now, such interesting analysis can be done primitively in TEA using the U-commands---especially because they not only tell us what the unique elements in a string are, but also which ones are most common, which ones rarest. Very useful foundational statistical analysis for free, so many classes of string processing problems become simpler to reason about in TEA.  A basic example of how to use U: is the problem of determining the winner at an election given the votes data. In this example TEA program, we assume a vote for candidate represented by letter A, appears as just the letter “A” in the list of votes, for candidate B, by “B”, and so on. Thus, a list of votes from a polling station, for 3 candidates W, C & A can be represented by the string “AWCCAWAWAAAAACCWACCCCWCACCA”. The following TEA program then, would help automatically rank the candidates by their votes, with the winner appearing first in the result.  I!:{AWCCAWAWAAAAACCWACCCCWCACCA} | u!:  Should return exactly "ACW" and not "CAW" because although A & C each have 11 votes and W has only 5, A occurs before C in the input. Essentially computes the modal sequence statistic of the input. A proof demonstration of this TEA program can be obtained using the “-d” DEBUG flag to the TEA interpreter TTTT when being invoked with the TEA program as shown in the screenshot below:  **EXPERIMENTS|< 13:56:15 $>\* tttt -c "I!:{AWCCAWAWAAAAACCWACCCCWCACCA} | u!:"**  **ACW**  **EXPERIMENTS|< 13:56:21 $>\* tttt -c "I!:{AWCCAWAWAAAAACCWACCCCWCACCA} | u!:" -d**  **No explicit INPUT found, using STDIN!**  **INPUT:**  **None**  **CODE:**  **I!:{AWCCAWAWAAAAACCWACCCCWCACCA} | u!:**  **---------[ IN TEA RUNTIME ]**  **#2 of ['I!:{AWCCAWAWAAAAACCWACCCCWCACCA} ', ' u!:']**  **##2 of ['I!:{AWCCAWAWAAAAACCWACCCCWCACCA} ', 'u!:']**  **CLEAN TEA CODE TO PROCESS:**  **I!:{AWCCAWAWAAAAACCWACCCCWCACCA}**  **u!:**  **---<< EXTRACTED TEA LABEL BLOCKS:**  **{}**  **Executing Instruction#0 (out of 2)**  **Processing Instruction: I!:{AWCCAWAWAAAAACCWACCCCWCACCA}**  **PRIOR MEMORY STATE: (=, VAULTS:{})**  **RESULTANT MEMORY STATE: (=AWCCAWAWAAAAACCWACCCCWCACCA, VAULTS:{})**  **Executing Instruction#1 (out of 2)**  **Processing Instruction: u!:**  **PRIOR MEMORY STATE: (=AWCCAWAWAAAAACCWACCCCWCACCA, VAULTS:{})**  **--[util]-| Computing Unique Character Projection for [AWCCAWAWAAAAACCWACCCCWCACCA]**  **--[util]-| Unique Char Tally [[('A', 11), ('C', 11), ('W', 5)]]**  **RESULTANT MEMORY STATE: (=ACW, VAULTS:{})**  **ACW**  Note that the example TEA program, because of its use of the special BASH character “!” [6], requires some clever treatment of the TEA source code when being used on the command-line. Thus, the actual command-line invocation of this example uses the modified syntax:  tttt -c "I!"":{AWCCAWAWAAAAACCWACCCCWCACCA}|u!":  Otherwise would raise Bash-runtime errors such as:  -bash: ": unrecognized history modifier  The work-around is to first disable history modifier functionality by running command “set +H”, then the code can be entered normally as:  tttt -c "I!:{AWCCAWAWAAAAACCWACCCCWCACCA} | u!:"  Otherwise, another great example is the following, this time demonstrating how u: can serve the useful purpose of computing a modal sequence statistic for a list of words.  #compute and return modal sequence statistic of the provided words list  v:vI:{CAT DOG CAT FISH CAT COW COW FISH COW DUCK} | u\*:vI  That program should return "CAT COW FISH DOG DUCK", which helps us tell that the word “CAT” occurs most frequent, or at least earliest in the provided list, while “DUCK” occurs least number of times, or last in the original input.  In essence, both u: and u!: compute the important, modal sequence statistic of the provided input. For those unfamiliar with the statistic or why it is important, consult Lutalo’s mathematical research paper[12] on Transformatics for details. |  |  |  | |