**Extended notes on Search (and introductory notes on feature vectors)**

Revision: Z::1.5j

Anchor … Terminal

Range (1 : 1) # range expression for the anchor

Range( : z) # range expression for terminal

Range( : ) # range expression for any location between anchor and terminal

Range( ti : tj ) # range expression for any location between ith token and jth token (i>1 and j>i+1)

a=1

z=span

consider: “p q … r s … [t u] ... v (w x) y”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Token** | **Range** | **Token Sequence** | **Position Tuple**  **(int, uint[], uint[])** | **min-position** | **max-position** |
| p | (1 : 1) | Anchor/T1 | (1, null, null) | 1 | 1 |
| q | (2 : 2) | T2 | (2, null, null) | 2 | 2 |
| r | (t2 : ) | T3 | (0, [2], null) | 1+T2.position() |  |
| s | (t3 : ) | T4 | (-1, [3], null) | 1+T3.position() | 1+T3.position() |
| t | (t4 : ) | T5 | (0, [4], [6]) | 1+T4.position() |  |
| u | (t4 : ) | T6 | (0, [4], [5]) | 1+T4.position() |  |
| v | (t5|t6 : ) | T7 | (0, [5,6], null) | 1+MAX(T6.position(), T6.position()) |  |
| (w x) | (t7 : ) | T8 | (-1, [7], null) | 1+T7.position() | 1+T7.position() |
| y | (t8 : ) | Terminal/T9 | (-1, [8], null) | 1+T8.position() | 1+T8.position() |

How does the Range Tuple work?

X = (int, \*, \*)

Y = (\*, uint[], \*)

Z = (\*, \*, uint[])

|  |  |  |  |
| --- | --- | --- | --- |
| **X** | **Y** | **Z** | **Meaning** |
| X > 0 | null | null | position = X (absolute position for token within the span) |
| X = 0 | Y.Length >= 1 | null | maxPosition = span;  minPosition = 0;  foreach y in Y:  if y > minPosition:  minPosition = y |
| X < 0 | Y.Length >= 1 | null | minPosition = 0;  foreach y in Y:  if y > minPosition:  minPosition = y  maxPosition = y |
| X = 0 | Y.Length >= 1 | Z.Length >= 1 | maxPosition = span;  minPosition = 0;  foreach y in Y such that y in not in already consumed by a member of Z:  if y > minPosition:  minPosition = y |
| X < 0 | Y.Length >= 1 | Z.Length >= 1 | minPosition = 0;  foreach y in Y such that y in not in already consumed by a member of Z:  if y > minPosition:  minPosition = y  maxPosition = y |

Another method: If we mark each matching token with a bit (and exclude token-matching on already matched tokens):

|  |  |  |
| --- | --- | --- |
| **X** | **Y** | **Meaning** |
| X > 0 | null | position = X (absolute position for token within the span) |
| X = 0 | Y.Length >= 1 | maxPosition = span;  minPosition = 0;  foreach y in Y:  if y > minPosition:  minPosition = y |
| X < 0 | Y.Length >= 1 | minPosition = 0;  foreach y in Y:  if y > minPosition:  minPosition = y  maxPosition = y |

And our earlier table above is simplified:

consider: “p q … r s … [t u] ... v (w x) [y z]”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Token** | **Range** | **Token Sequence** | **Position Tuple**  **(int, uint[])** | **min-position** | **max-position** |
| p | (1 : 1) | Anchor/T1 | (1, null) | 1 | 1 |
| q | (2 : 2) | T2 | (2, null) | 2 | 2 |
| r | (t2 : ) | T3 | (0, [2]) | 1+T2.position() |  |
| s | (t3 : ) | T4 | (-1, [3]) | 1+T3.position() | 1+T3.position() |
| t | (t4 : ) | T5 | (0, [4]) | 1+T4.position() |  |
| u | (t4 : ) | T6 | (0, [4]) | 1+T4.position() |  |
| v | (t5|t6 : ) | T7 | (0, [5,6]) | 1+MAX(T5.position(), T6.position()) |  |
| (w x) | (t7 : ) | T8 | (-1, [7]) | 1+T7.position() | 1+T7.position() |
| y | (t7|t10 : ) | T9 | (-1, [8]) | TBD | TBD |
| z | (t7:t8 : ) | Terminal/T10 | (-1, [8]) | TBD | TBD |

Consider the same list unquoted: p q r s t u v (w x) y z

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Token** | **Token Sequence** | **Range Tuple**  **(int, uint[])** | **min-position** | **max-position** |
| p | T1 | (0, null) | 1 | span |
| q | T2 | (0, null) | 1 | span |
| r | T3 | (0, null) | 1 | span |
| s | T4 | (0, null) | 1 | span |
| t | T5 | (0, null) | 1 | span |
| u | T6 | (0, null) | 1 | span |
| v | T7 | (0, null) | 1 | span |
| (w x) | T8 | (0, null) | 1 | span |
| y | T9 | (0, null) | 1 | span |
| z | T10 | (0, null) | 1 | span |

Tokens, represented by individual letters in this document, are ordinarily words in the lexicon to be searched. However, they can instead be a variety of feature of the lexical word:

* Lemmas
* Person/Number (1st, 2nd, or 3rd person / singular or plural)
* Part of Speech
* Etc…

These feature items can be utilized instead of the word itself and they can be ANDed and ORed. Consider a token as follows:

“couldst”

It could also be found using lemma and part-of-speech:

#could&/vmd2/

Or a more relaxed search could be executed as:

could#modern

**Bitwise Part-of-Speech and person number is also possible using these tables:**

WordClass (12 bits)

Person/Number (4 bits)

|  |  |  |
| --- | --- | --- |
| **Description** |  | **Left-Most Nibble** |
| Person bits |  | 0x3--- (0b--11) |
| Number bits |  | 0xC--- (0b11--) |
| Indefinite |  | 0x0--- (0b--00) |
| 1st Person |  | 0x1--- (0b--01) |
| 2nd Person |  | 0x2--- (0b--10) |
| 3rd Person |  | 0x3--- (0b--11) |
| Singular |  | 0x4--- (0b01--) |
| Plural |  | 0x8--- (0b10--) |
| WH\* |  | 0xC--- (0b00--) |

|  |  |
| --- | --- |
| **Description** | **Right 3 Nibbles** |
| NounOrPronoun | 0x-03- |
| Noun | 0x-01- |
| Noun: unknown gender | 0x-010 |
| Proper Noun | 0x-03- |
| Pronoun | 0x-02- |
| Pronoun: Neuter | 0x-021 |
| Pronoun: Masculine | 0x-022 |
| Pronoun: Non-feminine**\*** | 0x-023 |
| Pronoun: Feminine | 0x-024 |
| Pronoun/Noun: Genitive | 0x-0-8 |
| Pronoun: Nominative | 0x-06- |
| Pronoun: Objective | 0x-0A- |
| Pronoun: Reflexive | 0x-0E- |
| Pronoun: no case/gender | 0x-020 |
| Verb | 0x-1-- |
| to | 0x-200 |
| Preposition | 0x-400 |
| Interjection | 0x-800 |
| Adjective | 0x-A00 |
| Numeric | 0x-B00 |
| Conjunction | 0x-C0- |
| Determiner | 0x-D0- |
| Particle | 0x-E00 |
| Adverb | 0x-F00 |

Therefore, while terse, the previous search could be precisely executed as:

could#modern&/61--/

**Simplify:** “p q … r s … [t u] ... v (w x) [y z]” Since we search ordered tokens sequentially, we don’t care about min(position) … only adjacency.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Token** | **Token Sequence** | **Anchor/Adjacency**  **(-1/1)** | **Subgroup**  **(unordered)** | **Function Call** | **Results:** |
| p | Anchor/T1 | -1 | 0 | P = find(&cursor, p, 1, &span) | ++cursor, --span |
| q | T2 | 1 | 0 | Q = P & find(&cursor, q, 1, &span) | ++cursor, --span |
| r | T3 | 0 | 0 | R = Q & find(&cursor, r, span, &span) | cursor+=pos, span-=pos |
| s | T4 | 1 | 0 | S = R & find(&cursor, s, 1, &span) | ++cursor, --span |
| t | T5 | 0 | (1) | TU = S & find(&cursor, [t u], span, &span) | cursor+=pos, span-=pos |
| u | T6 | 0 | (1) |  | cursor+=pos, span-=pos |
| v | T7 | 0 | 0 | V = TU & find(&cursor, v, span, &span) |  |
| (w x) | T8 | 1 | 0 | WX = V & find(&cursor, (w x), span, &span) |  |
| y | T9 | 0 | (2) | YZ = WX & find(&cursor, [y z], span, &span) |  |
| z | Terminal/T10 | 0 | (2) |  |  |

FOUND=Z

Consider the same list unquoted: p q r s t u v (w x) y z

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Token** | **Token Sequence** | **Anchor/Adjacency** | **Subgroup**  **(unordered)** | **Function Call** | **Results:** |
| p | n/a | 0 | 1 | P = find(&cursor, p, span, &span) | n/a |
| q | n/a | 0 | 1 | Q = find(&cursor, q, span, &span) | n/a |
| r | n/a | 0 | 1 | R = find(&cursor, r, span, &span) | n/a |
| s | n/a | 0 | 1 | S = find(&cursor, S, span, &span) | n/a |
| t | n/a | 0 | 1 | T = find(&cursor, t, span, &span) | n/a |
| u | n/a | 0 | 1 | U = find(&cursor, u, span, &span) | n/a |
| v | n/a | 0 | 1 | V = find(&cursor, p, span, &span) | n/a |
| (w x) | n/a | 0 | 1 | WX = find(&cursor, (w x), span, &span) | n/a |
| y | n/a | 0 | 1 | Y = find(&cursor, y span, &span) | n/a |
| z | n/a | 0 | 1 | Z = find(&cursor, z, span, &span) | n/a |

FOUND = P & Q & R & S & T U & V & WX & Y & Z

**NOTES:**

* … [t u] is equivalent to [t u] without the ellipses prefix. But the ellipses postfix [t u] ... is significant.
* Quelle spec version 1.0.1.5i eliminated the (w x) syntax, replacing it at the feature level as w|x. Supporting a single [unified] or-syntax greatly reduces the complexity of Quelle search algebra. Currently, (w x) is be silently mapped to w|x by the standard Quelle driver.