# MetaMap Technical Notes

**Alan R. Aronson**

February 2, 1996

This document consists of a detailed description of MetaMap; it is based on the *MetaMap Conversion Specification*, a guide for converting the current Prolog/C implementation of MetaMap to one entirely written in C. Section 1: Introduction gives a general description of the Prolog modules along with an example of MetaMap processing. Section 2: MetaMap Modules describes the modules in more detail.

## 1. Introduction

MetaMap processing is controlled by several Prolog modules shown in Figure 1. The MetaMap

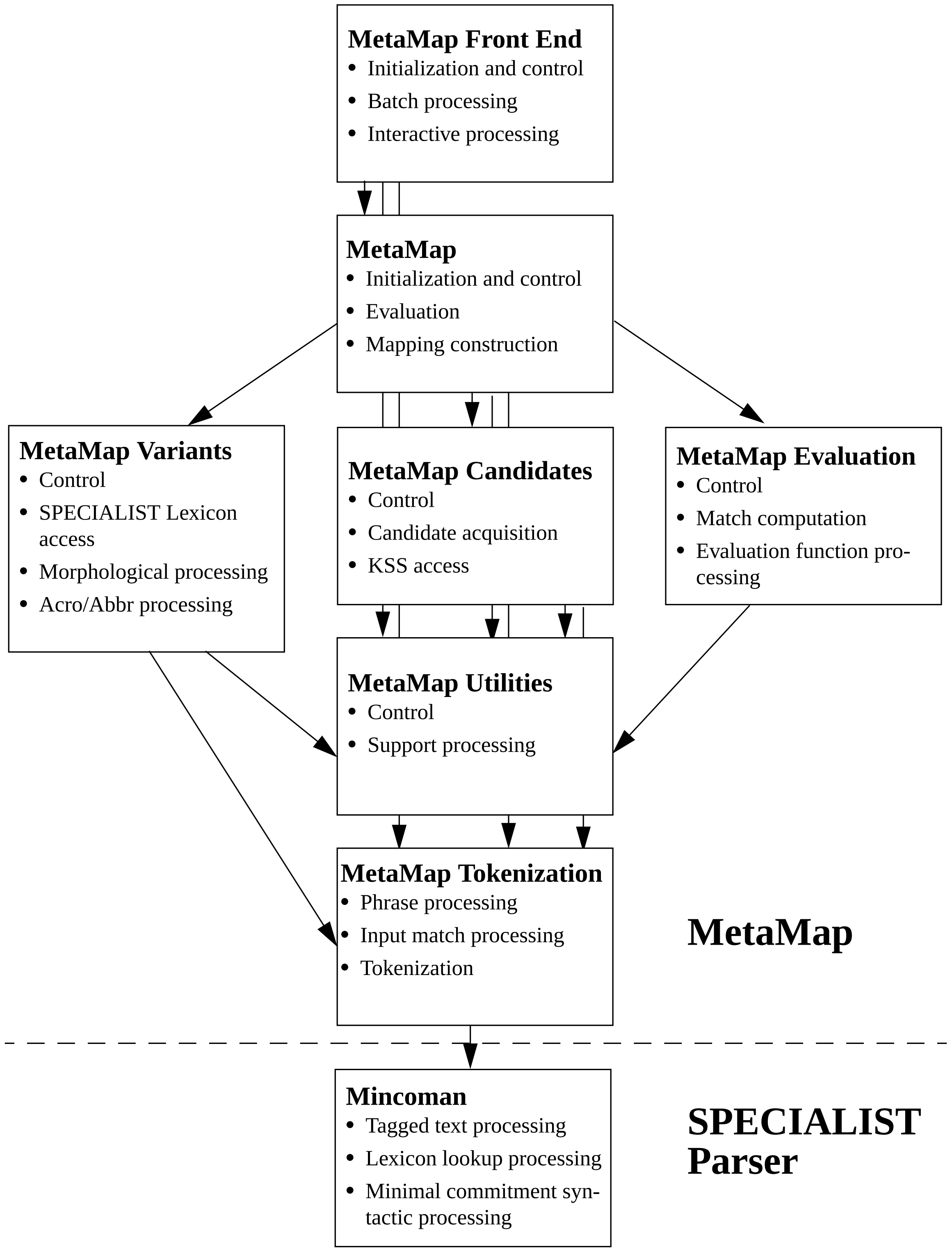

Figure 1. High-level MetaMap Module Dependencies

Front End module performs preliminary tasks such as reading an utterance to be processed. The MetaMap module controls the mapping process which consists of four major subtasks: variant generation, candidate retrieval, candidate evaluation, and mapping construction. These subtasks are performed by the MetaMap Variants, MetaMap Candidates, MetaMap Evaluation and MetaMap modules, respectively. Support for the entire effort is provided by the MetaMap Utilities and MetaMap Tokenization modules. Finally, syntactic processing is provided by one of the SPECIALIST parser modules, the Mincoman module.

Consider the utterance: “[ HSR39.qu.1 ] Anti-gastroesophageal reflux implantation.” The following sections show the major steps in the processing of this utterance.

## 1.1 Preliminary processing

MetaMap processing begins with separating a label from the text of an utterance and reading syntactic tags for the text, if any (see Figure 2). The SPECIALIST parser produces a syntactic analy-

```
Label: HSR39.qu.1

Text: “Anti-gastroesophageal reflux implantation.”

TagList: [ [Anti,pre],
           [-,hy],
           [gastroesophageal,adj],
           [reflux,adj],
           [implantation,noun],
           [.,pd]]
```

Figure 2. An utterance label, text and tagging

sis (Figure 3) which is then augmented with tokens (Figure 4). The final stage in the preliminary

```
[  not_in_lex([inputmatch([Anti])]),
   punc([inputmatch([-])]),
   mod([lexmatch([gastroesophageal reflux]),inputmatch([gastroesoph-
     ageal,reflux])]),
   head([lexmatch([implantation]),inputmatch([implantation])]),
   punc([inputmatch([.])])]
```

Figure 3. The minimal commitment syntactic analysis

```
[  not_in_lex([inputmatch([Anti]),tokens([anti])]),
   punc([inputmatch([-]),tokens([])]),
   mod([lexmatch([gastroesophageal reflux]),inputmatch([gastroe-
     sophageal,reflux]),tokens([gastroesophageal,reflux])]),
   head([lexmatch([implantation]),inputmatch([implanta-
     tion]),tokens([implantation])]),
   punc([inputmatch([.]),tokens([])])]
```

Figure 4. Syntactic analysis augmented with tokens

processing consists of gleaning information from the augmented syntactic analysis, especially finding the correspondence between the syntactic analysis and the words of the utterance. This produces Figure 5.[1] See Section 2.2 for an explanation of this information.

Phrase words: `[anti,gastroesophageal,reflux,implantation]`
Phrase head words: `[implantation]`
Syntax/word mapping (phrase map): `[[1,1],[0,-1],[2,3],[4,4],[0,-1]]`

Figure 5. Phrase word information

## 1.2 Variant generation

The approach taken in computing variants is a canonicalization approach. This simply means that a variant represents not only itself but all of its inflectional and spelling variants.[2] Collapsing inflectional and spelling variants results in significant computational savings. Variant generation begins with finding *meaningful* sequences of words from the input text. These sequences are called *variant generators*. All individual words and all multi-word sequences occurring in the SPECIALIST lexicon are meaningful. The variant generators for the example utterance are given

1. A *filtered* version of the phrase word information omitting prepositions, determiners and other *insignificant* words is not shown since it is the same in this case. Note that the filtered version of the phrase word information is used during the search for a mapping to the Metathesaurus, and the unfiltered version is used to coordinate results to the input text.

2. A spelling variant of a word is just a variant having the same principal part as the word. For example, *haemorrhaged* is a spelling variant of *hemorrhaged*.

in Figure 6. For each of the generators, derivational variants, acronyms/abbreviations and syn-

```
v(anti,[],…
v(gastroesophageal reflux,[noun],…
v(gastroesophageal,[adj],…
v(reflux,[verb,noun,adj],…
v(implantation,[noun],…
```

Figure 6. Variant generators and their parts of speech

onyms are computed producing the variants shown in Figure 7. In preparation for evaluating can-

```
v(anti,[],0,[],…
v(gastroesophageal reflux,[noun],0,[],…
v(gastroesophageal,[adj],0,[],…
v(reflux,[verb,noun,adj],0,[],…
v(implantation,[noun],0,[],…
v(implant,[verb],3,”d”,…
v(implantable,[adj],6,”dd”,…
```

Figure 7. Variants and their parts of speech, variant distance and history

didate Metathesaurus strings, inflectional variants are computed for each of these variants. This final list of variants is shown in Figure 8.

```
v(anti,[],0,[],anti,4)
v(gastroesophageal reflux,[noun],0,[],gastroesophageal reflux,3)
v(gastrooesophageal reflux,[],0,”p”,gastrooesophageal reflux,3)
v(gastroesophageal,[adj],0,[],gastroesophageal,3)
v(gastro-oesophageal,[],0,”p”,gastro esophageal,3)
v(reflux,[verb,noun,adj],0,[],reflux,2)
v(refluxed,[],1,”i”,reflux,2)
v(refluxes,[],1,”i”,reflux,2)
v(refluxing,[],1,”i”,reflux,2)
v(implantation,[noun],0,[],implantation,1)
v(implant,[verb],3,”d”,implant,1)
v(implantable,[adj],6,”dd”,implantable,1)
v(implantations,[],1,”i”,implantation,1)
v(implanted,[],4,”id”,implant,1)
v(implanting,[],4,”id”,implant,1)
v(implants,[],4,”id”,implant,1)
```

Figure 8. All variants

## 1.3 Candidate retrieval

All Metathesaurus strings beginning with one of the variants in Figure 8 are retrieved using the MetaWordIndex module. Some of the 239 such strings for our example are displayed in Figure 9.

```
csc([anti,3,4,diol,1,2,oxide,benz,a,anthracene],anti-3,4-diol 1,2-
     oxide benz(a)anthracene,anti-3,4-diol 1,2-oxide
     benz(a)anthracene)
csc([anti,5,methylchrysene,1,2,dihydrodiol,3,4,epoxide],anti-5-
     methylchrysene-1,2-dihydrodiol-3,4-epoxide,anti-5-methylchry-
     sene-1,2-dihydrodiol-3,4-epoxide)
csc([anti,a,carbohydrate,test],Anti-A-carbohydrate test,Anti-A-car-
     bohydrate test)
csc([anti,a,cho,test],Anti-A-CHO test,Anti-A-carbohydrate test)
csc([anti,abortion,group],Anti-Abortion Group,Anti-Abortion Groups)
csc([gastroesophageal,reflux],GASTROESOPHAGEAL REFLUX,Gastroesoph-
     ageal Reflux)
csc([gastroesophageal,reflux],Gastroesophageal Reflux,Gastroesoph-
     ageal Reflux)
csc([gastro,esophageal,laceration,syndrome],Gastro-esophageal lac-
     eration syndrome,Mallory-Weiss Syndrome)
csc([gastro,esophageal,reflux],Gastro Esophageal Reflux,Gastroe-
     sophageal Reflux)
csc([reflux],Reflux, NOS,Reflux, NOS)
csc([reflux],reflux,Reflux, NOS)
csc([implant],Implant,Implant <1>)
csc([implant],Implant, NOS,Implantation, NOS <1>)
csc([implant],implant,Implants, Artificial)
csc([implantation],Implantation,Implantation, NOS <1>)
csc([implantation],Implantation,Ovum Implantation)
```

Figure 9. Metathesaurus candidates

Each example consists of the string’s tokens, the string itself, and the string’s concept.

## 1.4 Candidate evaluation

Each of the Metathesaurus candidates is evaluated with respect to the text of the phrase. Figure 10

```
ev(-812,Implantation,Implantation, NOS <1>,[implantation],[Thera-
     peutic or Preventive Procedure], [[[4,4],[1,1],0]],yes,no)
ev(-812,Implantation,Ovum Implantation,[implantation],[Organism
     Function], [[[4,4],[1,1],0]],yes,no)
ev(-741,Implant,Implant <1>,[implant],[Medical
     Device],[[[4,4],[1,1],3]],yes,no)
ev(-741,implant,Implants, Artificial,[implant],[Medical
     Device],[[[4,4],[1,1],3]],yes,no)
ev(-729,Implanted,Implanted,[implanted],[Functional Con-
     cept],[[[4,4],[1,1],4]],yes,no)
ev(-729,Implants,Implants <1>,[implants],[Biomedical or Dental
     Material], [[[4,4],[1,1],4]],yes,no)
ev(-729,Implants,Implants <2>,[implants],[Medical
     Device],[[[4,4],[1,1],4]],yes,no)
ev(-694,GASTRO-OESOPHAGEAL REFLUX,Gastroesophageal Reflux,[gas-
     tro,oesophageal,reflux],[Disease or Syndrome],
     [[[2,2],[1,2],0],[[3,3],[3,3],0]],no,no)
ev(-694,GASTROESOPHAGEAL REFLUX,Gastroesophageal Reflux,[gastroe-
     sophageal,reflux],[Disease or Syndrome],
     [[[2,3],[1,2],0]],no,no)
ev(-645,Reflux, NOS,Reflux, NOS,[reflux],[Finding,Sign or Symptom],
     [[[3,3],[1,1],0]],no,no)
```

Figure 10. Candidate evaluations

displays the results. Each result consists of the negation of the score for the candidate, the candidate itself, the candidate's corresponding concept, the candidate's tokens, the concept's semantic types, the mapping from the phrase to the candidate, whether the mapping involves the head of the phrase, and whether or not the match is an overmatch (always no here since the default behavior prohibits overmatches).

## 1.5 Mapping construction

The final step in MetaMap consists of combining candidates to form as complete a mapping as possible. The best mappings for the example, both with final scores of 840, are given in Figure 11.

```
map(-840,[ev(-694,GASTRO-OESOPHAGEAL REFLUX,Gastroesophageal
                 Reflux,[gastro,oesophageal,reflux],[Disease or
                 Syndrome],[[[2,2],[1,2],0],[[3,3],[3,3],0]],no,no),
          ev(-812,Implantation,Implantation, NOS <1>,[implantation],
                 [Therapeutic or Preventive Procedure],
                 [[[4,4],[1,1],0]],yes,no)])
map(-840,[ev(-694,GASTRO-OESOPHAGEAL REFLUX,Gastroesophageal
                 Reflux,[gastro,oesophageal,reflux],[Disease or
                 Syndrome],[[[2,2],[1,2],0],[[3,3],[3,3],0]],no,no),
          ev(-812,Implantation,Ovum Implantation,[implantation],
                 [Organism Function],[[[4,4],[1,1],0]],yes,no)])
```

Figure 11. The best mappings for “Anti-gastroesophageal reflux implantation.”

# 2. MetaMap Modules

This section describes the main tasks for each of the MetaMap modules.

## 2.1 MetaMap Front End Module (`metamap_fe`)

The purpose of `metamap_fe` is to process text interactively or in batch mode, with or without tagging, to the point where `metamap:metamap_phrase/6` can be called. This has already been sufficiently exemplified in Figure 2 and Figure 3.

## 2.2 MetaMap Module (`metamap`)

`metamap` is MetaMap's main source of control. Normal processing consists of calls to the following predicates:

- `metamap_tokenization:add_tokens_to_phrase/2.` The syntactic analysis produced by the Mincoman module often contains multi-word items, e.g., *gastroesophageal reflux*.[1] Uniform treatment of what constitutes a "word" argues for tokenizing multi-word items. The tokenization algorithm used is the same as that of `wordind`. The tokens for *gastroesophageal reflux* are simply *gastroesophageal* and *reflux*. See Figure 4 above for a full example.
- `metamap_tokenization:parse_phrase_word_info/2.` Based on the tokens just computed and in further preparation for computing variants and finding Metathesaurus candidates, `parse_phrase_word_info/2` determines the correspondence between the syntactic components and the tokens of the phrase. This is shown pictorially in Figure 12. The first element of

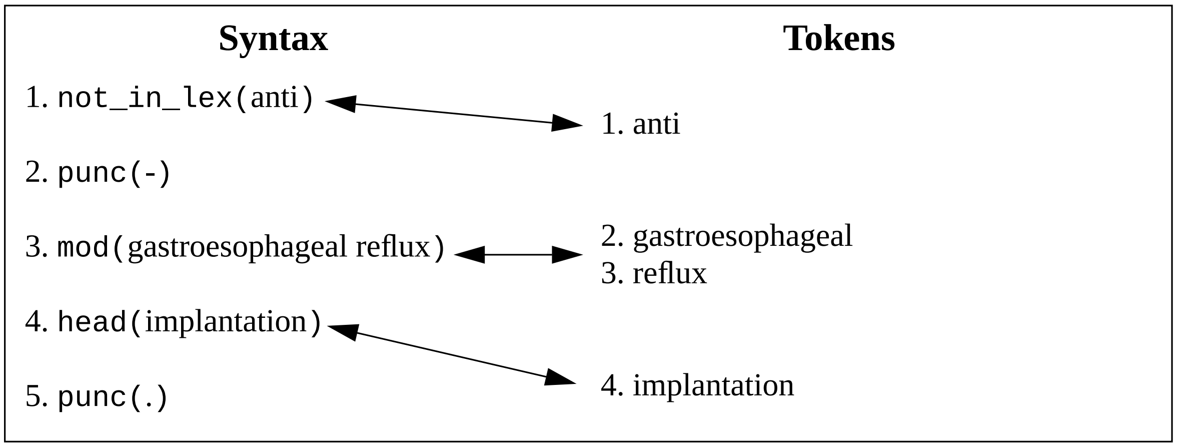

Figure 12. Syntax/Token correspondence

the syntax (simplified for exposition), `not_in_lex(`anti`)` corresponds to the first token, [1,1], denoted by word span beginning and ending at word 1. The second syntactic element, `punc(-)`, corresponds to no tokens, denoted by [0,-1]. The third element corresponds to tokens 2 and 3, [2,3]; the fourth corresponds to word 4, [4,4]; and the last corresponds to nothing, [0,-1]. Concatenating these correspondences produces the *mapping* from syntax to tokens: [[1,1], [0,-1], [2,3], [4,4], [0,-1]]. Besides the mapping, parse_phrase_word_info/2 computes the list of all tokens, [anti, gastroesophageal, reflux, implantation], and a lowercase version (the same in this case[2]); the list of head tokens, [implantation], and a lowercase version (again the same); and

1. The reason for this is that the SPECIALIST lexicon contains multi-word entries.

another analysis as above computed by filtering out the tokens of non-*meaningful* syntactic components, i.e., those with tags prep, det, aux, modal, compl, punc, num, conj and pron. For our example the filtered version is the same as the unfiltered version. The actual phrase word information for the example is shown in Figure 13.

```
pwi(  wdl([anti,gastroesophageal,reflux,implantation],
           [anti,gastroesophageal,reflux,implantation]),
      wdl([implantation],[implantation])
      [[1,1],[0,-1],[2,3],[4,4],[0,-1]])
:
pwi(  wdl([anti,gastroesophageal,reflux,implantation],
           [anti,gastroesophageal,reflux,implantation]),
      wdl([implantation],[implantation]),
      [[1,1],[0,-1],[2,3],[4,4],[0,-1]])
```

Figure 13. Phrase word information

- `metamap:compute_evaluations/3. compute_evaluations/3` is a control predicate which invokes six predicates which successively compute variant generators, variants themselves, Metathesaurus candidates for the variants, and the evaluations of the candidates. The resulting information is stored in several data structures (Prolog terms) defined in Figure 14. A

GVC (Generator/Variants/Candidates): `gvc/3`
```
      gvc(Generator, Variants, Candidates)
```

Variant: `v/6`
```
      v(Word, Categories, VarLevel, History, BasifiedWord, NFR)
```
[canonical algorithm]

Candidate: `csc/3`
```
      csc(CanonicalText, StringText, ConceptText)
```

Evaluation: `ev/8`
```
      ev(NegValue, MetaTerm, MetaConcept, MetaWords, SemTypes,
         MatchMap, InvolvesHead, IsOvermatch)
```

Figure 14. Definitions of GVC, Variant, Candidate and Evaluation

GVC structure consists of a variant generator (which is, itself, a variant), its variants, and the Metathesaurus candidates for the variants. A variant consists of a Word (possibly a multi-word), its syntactic Categories, its variation distance (VarLevel) and History[1] from the corresponding phrase word, its canonical form (BasifiedWord), and the right-to-left count of the word in the phrase (NFR). This count is used to restrict search to the 1- and 2-word indexes for the rightmost two words in the phrase. A candidate consists of a Metathesaurus string (StringText), its

2. Actually, in the current implementation all tokens are lowercased; so the lowercase versions of tokens lists are always redundant.

1. A variant's history is a string of characters (displayed in reverse order) indicating the variation steps taken in order to produce the variant. The characters are p (spelling variant), i (inflection), d (derivational variant), s (synonym), a (acronym/abbreviation), and x (an acronym/abbreviation expansion). A generator's history is the null string.

concept (ConceptText), and the list of tokens in the string (CanonicalText). Note that the use of the word *canonical* here is historical; it is completely *unrelated* to the notion of *canonical form*. An evaluation consists of the negation of the evaluation value (NegValue), the Metathesaurus string (MetaTerm), its concept (MetaConcept), its tokens (MetaWords), its semantic types (SemTypes), the mapping to the phrase (MatchMap), whether the mapping involves the head of the phrase (InvolvesHead) and whether or not the mapping is an overmatch (IsOvermatch). Note that MetaTerm is the same as StringText in the candidate structure `csc/3`, MetaConcept is the same as ConceptText, and MetaWords is the same as CanonicalText. The duplication of names is kept in order to facilitate examination of the code. The six predicates which compute the information in these structures are as follows:

- `metamap_variants:compute_variant_generators_canon/2.` The search for Metathesaurus strings related to the phrase begins by examining the filtered phrase words, i.e., those words of the phrase with a *meaningful* syntactic category. The phrase words for our example are *anti*, *gastroesophageal*, *reflux*, and *implantation*. The variant generators for the phrase consist of each of the individual words and all subsequences of the words which are in the SPECIALIST lexicon. In this case, only one subsequence, *gastroesophageal reflux*, is in the lexicon. It becomes the fifth variant generator for our example. Figure 6 above shows their internal structure.
- `metamap_variants:augment_each_with_variants_canon/1.` The variants for each of the variant generators is computed according to an algorithm described below in Section 2.3. Figure 7 above shows the results. Note that the variants computed at this point are used for searching the Metathesaurus for strings which match the phrase.
- `metamap_variants:gather_variants_canon/1.` For use in determining the variant distance between phrase words and the variants that will be found in Metathesaurus strings, all inflectional variants of the variants computed above are gathered together. Figure 8 above shows all variants for our example.
- `metamap_candidates:add_candidates_canon/3.` The MetaWordIndex module is used to find all Metathesaurus strings containing at least one of the variants computed by `augment_each_with_variants_canon/1`. The algorithm for doing this is discussed in Section 2.4 below, and some of the 239 results for our example are shown in Figure 9 above.
- `metamap_evaluation:evaluate_candidates/10.` Each of the 239 candidates found above is evaluated according to a four-part evaluation metric defined below in Section 2.5. Part of the default evaluation process involves filtering out overmatches and candidates with gaps. For example, all 196 candidates beginning with *anti* (e.g., “Anti Rheumatic Agents” and “anti-liver kidney microsome antibody”) are overmatches and are not completely evaluated. The default behavior can be overridden, but this explains why only thirteen of the 239 candi-

dates (see Figure 15) are fully evaluated. Note that variables occur where semantic types

```
ev(-812,Implantation,Implantation, NOS <1>,[implantation],_701987,
     [[[4,4],[1,1],0]],yes,no)
ev(-812,Implantation,Ovum Implantation,[implantation],_702163,
     [[[4,4],[1,1],0]],yes,no)
ev(-779,Implantations,Implantation, NOS <1>,[implantations],
     _702339,[[[4,4],[1,1],1]],yes,no)
ev(-779,Implantations,Ovum Implantation,[implantations],_702515,
     [[[4,4],[1,1],1]],yes,no)
ev(-741,Implant,Implant <1>,[implant],_701459,
     [[[4,4],[1,1],3]],yes,no)
ev(-741,Implant, NOS,Implantation, NOS <1>,[implant],_701635,
     [[[4,4],[1,1],3]],yes,no)
ev(-741,implant,Implants, Artificial,[implant],_701811,
     [[[4,4],[1,1],3]],yes,no)
ev(-729,Implanted,Implanted,[implanted],_702691,
     [[[4,4],[1,1],4]],yes,no)
ev(-729,Implants,Implants <1>,[implants],_702867,
     [[[4,4],[1,1],4]],yes,no)
ev(-729,Implants,Implants <2>,[implants],_703043,
     [[[4,4],[1,1],4]],yes,no)
ev(-694,GASTRO-OESOPHAGEAL REFLUX,Gastroesophageal Reflux,[gas-
     tro,oesophageal,reflux],_700538,
     [[[2,2],[1,2],0],[[3,3],[3,3],0]],no,no)
ev(-694,GASTROESOPHAGEAL REFLUX,Gastroesophageal Reflux,[gastroe-
     sophageal,reflux],_699985,[[[2,3],[1,2],0]],no,no)
ev(-645,Reflux, NOS,Reflux, NOS,[reflux],_701096,
     [[[3,3],[1,1],0]],no,no)
```

Figure 15. Pre-filtered candidate evaluations

would normally appear since they have not been computed yet.

- `metamap:filter_out_redundant_evaluations/2`. This is the last of the six predicates invoked by `compute_evaluations/3`. It simply filters out evaluations for which there is a higher scoring evaluation involving the same concept (but different strings). For example, the concept "Implantation, NOS" has strings "Implantation", "Implantations" and "Implant" with evaluation scores of 812, 779 and 741, respectively. The second and third evaluations are redundant and thus filtered out. All redundant evaluations are highlighted in Figure 15, and the filtered evaluations (with semantic types) are shown in Figure 10.
- `metamap:construct_best_mappings/5`. MetaMap's last task is to choose combinations of the evaluated candidates in order to produce the best mappings from the phrase to Metathesaurus concepts. In the simplest cases, a single concept maps the entire phrase. In general candidates associated with disjoint parts of the phrase are combined, and an evaluation for the combination is computed just as for individual candidates. The default behavior is to report only the highest scoring mappings.

### 2.3 MetaMap Variants Module (`metamap_variants`)

`metamap_variants` computes variants of phrase words for later use as keys for retrieval of

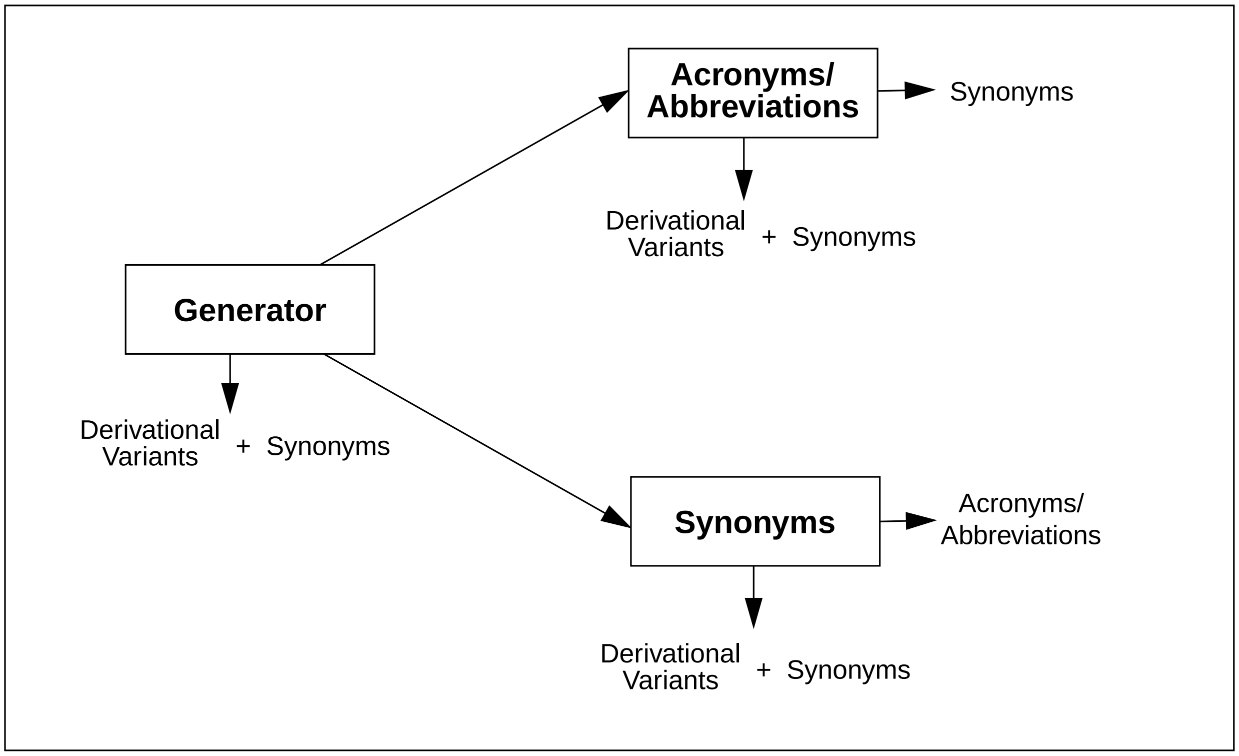
Acronyms/ Abbreviations
Synonyms
Derivational Variants + Synonyms
Generator
Derivational Variants + Synonyms
Synonyms
Acronyms/ Abbreviations
Derivational Variants + Synonyms

Figure 16. Variant generation

Metathesaurus candidates. The algorithm is shown graphically in Figure 16. The algorithm begins by computing variant *generators* for the phrase being processed (see Section 1.2 above). The bulk of the algorithm consists of sequentially computing the following sets of variants. Note that each of the sets has a corresponding designation in Figure 16.

- G—a generator;
- GDs—the derivational variants of G;
- GDSs—the synonyms of GDs;
- GAAs—the acronyms/abbreviations of G;
- GAADs—the derivational variants of GAAs;
- GAADSs—the synonyms of GAADs;
- GSs—the synonyms of G;
- GSDs—the derivational variants of GSs;
- GSDSs—the synonyms of GSDs;
- GAASs—the synonyms of GAAs; and
- GSAAs—the acronyms/abbreviations of GSs.

## 2.4 MetaMap Candidates Module (`metamap_candidates`)

`metamap_candidates` retrieves Metathesaurus strings as candidates for mapping the phrase. The process is conceptually simple. Each variant generated by `metamap_variants` is tokenized into canonical form. The first canonical form is used as a key to the `word_index` module to retrieve all Metathesaurus strings beginning with the form,[1] and the remaining canonical forms (if any) are used to filter out strings which do not contain the entire variant. Each retrieved Metathesaurus string is returned along with its associated concept and an uninverted, lowercased version of itself. If the control option `stop_large_n` is set (the default), `metamap_candidates` does not search single-character variants which begin more than 1,000 Metathesaurus strings or two-character variants which begin more than 500 strings.

## 2.5 MetaMap Evaluation Module (`metamap_evaluation`)

The candidate evaluation process is described in section 5 of *Metamap: Mapping Text to the UMLS® Metathesaurus®*. The information needed to apply the evaluation function consists of

- basic variant information for all variants of the phrase words;
- a *matchmap*, a mapping of phrase words to Metathesaurus string words along with variant distance information;
- an indication of whether the mapping involves the head of the phrase or not; and
- the connected components of the mapping.

The variant information is stored in an AVL tree where keys are the first words of the variants and values are `vinfo/5` terms consisting of the variant's generator, the position of the generator in the phrase, whether the variant involves the head of the phrase, the variant itself, and the tokenized words of the variant (the first of which is the key for finding the variant). The first several AVL entries are shown in Figure 17. The second entry is for the variant *gastro-oesophageal* which is a spelling variant ("p") of the generator *gastroesophageal*. It has variant distance 0, it matches the second word ([2,2]) of the phrase, is the third phrase word counting from the right, it consists of words *gastro* and *oesophageal*, and is indexed under *gastro*.

An example will illustrate the remaining information needed by the evaluation function. The example phrase we have been using consists of the four words [*anti*, *gastroesophageal*, *reflux*, *implantation*] only one of which (*implantation*) is a head word. Consider the Metathesaurus string "GASTRO-OESOPHAGEAL REFLUX" (with concept "Gastroesophageal Reflux"). The words for the string are [*gastro*, *oesophageal*, *reflux*]. Its matchmap is [[[2,2],[1,2],0], [[3,3],[3,3],0]] which means that the second word ([2,2]) *gastroesophageal* of the phrase maps to the first two words ([1,2]) *gastro oesophageal* of the string, and the third word ([3,3]) *reflux* of the phrase maps to the third word ([3,3]) *reflux* of the string. In addition, both of these maps involve a variant distance of 0. The mapping does not involve the head. Finally, the mapping's connected components

1. The canonicalization algorithm assumes that overmatches and concept gaps are not allowed. Thus any Metathesaurus string which can be used to map the phrase *must* begin with one of the variants in the phrase, and the canonicalization versions of indexes used by the `word_index` module only index on the canonical form of the *first* words of the strings.

are [[2], [3]] meaning that two consecutive words *gastroesophageal reflux* of the phrase map to three consecutive words *gasrto oesophageal reflux* of the string.

```
anti→vinfo: v(anti,[],0,[],anti,4)
             [1,1],no
             v(anti,[],0,[],anti,4)
             [anti]
gastro→vinfo:  v(gastroesophageal,[adj],0,[],gastroesophageal,3)
               [2,2],no
               v(gastro-oesophageal,[],0,”p”,gastro-oesophageal,3)
               [gastro,oesophageal]
gastroesophageal→vinfo: v(gastroesophageal,[adj],0,[],gastroesophageal,3)
                        [2,2],no
                        v(gastroesophageal,[adj],0,[],…,3)
                        [gastroesophageal]
                  vinfo:v(gastroesophageal reflux,[noun],0,[],…,3)
                        [2,3],no
                        v(gastroesophageal reflux,[noun],0,[],…,3)
                        [gastroesophageal,reflux]
gastrooesophageal→vinfo:v(gastroesophageal reflux,[noun],0,[],…,3)
                        [2,3],no
                        v(gastrooesophageal reflux,[],0,”p”,…,3)
                        [gastrooesophageal,reflux]
implant→vinfo: v(implantation,[noun],0,[],implantation,1)
               [4,4],yes
               v(implant,[verb],3,”d”,implant,1)
               [implant]
implantable→vinfo:v(implantation,[noun],0,[],implantation,1)
                  [4,4],yes
                  v(implantable,[adj],6,”dd”,implantable,1)
                  [implantable]
implantation→vinfo:  v(implantation,[noun],0,[],implantation,1)
                     [4,4],yes
                     v(implantation,[noun],0,[],implantation,1)
                     [implantation]
implantations→vinfo: v(implantation,[noun],0,[],implantation,1)
                     [4,4],yes
                     v(implantations,[],1,”i”,implantation,1)
                     [implantations]
…
```

Figure 17. Variant information used in evaluation

Details of the computation follow. The evaluation function is a weighted average of several components with different weightings for normal processing and term processing. The components which comprise the function are

- centrality (does the candidate involve the most important part of the phrase, the head?);
- variation (how much to the variants in the candidate differ from the phrase words?);
- coverage (how much of the phrase does the candidate cover?);
- cohesiveness (how smoothly does the candidate cover the phrase?); and

- involvement (an alternative to coverage and cohesiveness answering the question how much of the phrase is involved in the match?).

The evaluation function for normal processing is shown in Figure 18. The function for term pro-

```
Value = integer(1000*[(CenValue + VarValue + 2.0*(CovValue + CohValue))/6.0])
```

Figure 18. Normal Evaluation Function

cessing is given in Figure 19. Note that the *global* evaluation components (coverage, cohesiveness

```
Value = integer(1000*[(CenValue + VarValue + 4.0*InvValue)/6.0])
```

Figure 19. Evaluation Function for Term Processing

and involvement) receive twice the weight as the *local* components (centrality and variation). Also, the evaluation function for term processing is obtained from the normal evaluation function by replacing both the coverage and cohesiveness components with the involvement component. The less demanding involvement component seems to produce better results in browsing situations implied by term processing. The evaluation components are described in the following subsections. Note that the definitions in *Metamap: Mapping Text to the UMLS® Metathesaurus®* suffice for all of the components except involvement. Consequently, the discussion below consists mainly of a detailed example. The example consists of evaluating the Metathesaurus string “GASTRO-OESOPHAGEAL REFLUX” for the phrase *Anti-gastroesophageal reflux implantation.* The basic information needed to compute the evaluation value is contained in Figure 20.

- Phrase: *Anti-gastroesophageal reflux implantation.*
- Phrase words: [anti, gastroesophageal, reflux, implantation]
- Phrase head words: [implantation]
- Metathesaurus string: “GASTRO-OESOPHAGEAL REFLUX”
- Meta words: [gastro, oesophageal, reflux]
- The matchmap: [[[2,2],[1,2],0], [[3,3],[3,3],0]]

Figure 20. Information Needed for Evaluation

### 2.5.1 Centrality (CenValue)

The centrality value is 0 since the two-part mapping from [gastroesophageal] to [gastro, oesophageal] and from [reflux] to [reflux] does not involve the head, [implantation].

### 2.5.2 Variation (VarValue)

Variation scores are computed for each part of the mapping. In this case, the variation distance between [gastroesophageal] and [gastro, oesophageal] is 0 since the latter is a spelling variant of the former. Thus the variation score for the first part of the mapping is 1 (4/(0+4)). The variation score for the second part of the mapping is also 1 since there is no variation at all in the second part ([reflux]). Since the average of 1 and 1 is 1, the final variation value is 1.

### 2.5.3 Coverage (CovValue)

In order to compute the coverage value, we compute the individual values for the phrase and for Meta, and then average them weighting the Meta value twice as heavily as the phrase value. The phrase has 4 words and a phrase span of 2 (since word 2 is the first word involved in the matchmap and word 3 is the last word). Thus the coverage value for the phrase is 2/4 or 1/2. Similarly, there are 3 Meta words, and the Meta span is 3 (since the first and last Meta words involved in the matchmap are 1 and 3, respectively). Thus the coverage value for Meta is 3/3 or 1. The final coverage value is the weighted average (1*(1/2) + 2*(1))/3 or 0.83.

### 2.5.4 Cohesiveness (CohValue)

In order to compute the cohesiveness value, we again compute the individual values for the phrase and for Meta, and then average them in the same way. The phrase has a single connected component [gastroesophageal, reflux] of size 2. Since the phrase has 4 words, the cohesiveness value for the phrase is $2^2/4^2$ or 1/4. Similarly, Meta has a single connected component [gastro, oesophageal, reflux] of size 3. The cohesiveness value for Meta, then, is $3^2/3^2$ or 1. The final cohesiveness value is the weighted average (1*(1/4) + 2*(1))/3 or 0.75.

According to Figure 18, the final value for normal evaluation is integer(1000*[(0 + 1 + 2*(0.83 + 0.75))/6.0]) or 694.

### 2.5.5 Involvement (InvValue)

The involvement value is a rough approximation of the coverage and cohesiveness values. The strict word order implied by the matchmap is no longer followed. The involvement value for the phrase is the proportion of phrase words which *can* map to a Meta word whether or not they do according to the matchmap. For example, given the phrase *Advanced cancer of the lung* with words [advanced, cancer, lung] and the Meta string "Lung Cancer" with words [lung, cancer], the matchmap maps lung to lung, but does not map cancer because of word order. The phrase involvement value here is 2/3 as opposed to the coverage value of 1/3. Similarly, the involvement value for the Meta string is the proportion of words which *can* be mapped to from the phrase. For the current example, the Meta involvement value is 2/2 or 1 rather than 1/2 for coverage. Thus the final involvement value for this example is the weighted average (2/3 + 1)/2 or 0.83.[1]

## 2.6 MetaMap Utilities Module (`metamap_utilities`)

### 2.6.1 MatchMap predicates

- `positions_overlap/2`—determines if two positions overlap. For example, [2,3] and [3,4] do overlap and [2,3] and [4,4] do not.
- `correct_components/2`, `correct_component/2`—corrects terms of the form [m,n] which write_term/2 sometimes garbles on output. For example, `[9,10]` is written as the six

1. Note that the weighting of phrase involvement and Meta involvement is equal rather than the normal 1:2 ratio. Although the involvement value could be computed as implied by the definition, MetaMap actually uses the information contained in the matchmap, finds the extra Meta words relevant to the involvement computation, and combines this information to compute the involvement value.

characters `"^I^J"`. When the erroneous output is read back in, it is a string of four characters rather than a list of two integers. These predicates correct the problem.

### 2.6.2 Text manipulation predicates

- `eliminate_multiple_meaning_designator/2`, `eliminate_multiple_meaning_designator_string/2`—strip <n> from an atom or string. For example, 'Implant <1>' becomes 'Implant'.

### 2.6.3 I/O predicates

- `show_simple_syntax/3`—simply displays the text of the phrase rather than the results of the parser.
- `get_phrase_text/4`—performs the extraction of the `inputmatch` information from a parse for `show_simple_syntax/3`.
- `get_utterance/3`, `get_phrases/2`—reads previously output MetaMap results consisting of one `utterance/2` term, one or more `phrase/2`, `candidates/1` and `mappings/1` terms, and one `'EOU'/1` term from a file.
- `dump_aphrase_mappings/2, print_mappings/3, dump_evaluations/2`, `dump_evaluations_indented/2, print_evaluations/3`, `print_evaluations_indented/3`, `dump_variants_labelled/2`—These predicates display what their names indicate. The difference between the `dump_`... predicates and the `print_`... predicates is that the `print_`... predicates take an additional stream argument.

### 2.6.4 Miscellaneous predicates

- `dump_time/3`—displays timing information associated with the `--dump_timing` option.
- `wgvcs/1`—writes a list of GVCs.
- `wl/1`—writes a list.
- `write_avl_list/1`—writes an AVL list.

## 2.7 MetaMap Tokenization Module (`metamap_tokenization`)

### 2.7.1 Syntactic analysis predicates

- `generate_syntactic_analysis/2`, `generate_syntactic_analysis/3`—produce a syntactic analysis from possibly tagged text. They simply call `qp_token:tokenize_string/2`, `qp_lookup:assembledefns/2`, `generate_varinfo:generate_variant_info/2`, and `mincoman:minimal_commitment_analysis/5`.

### 2.7.2 Phrase processing predicates

- `add_tokens_to_phrase/2`—adds a `tokens/1` feature to each element of a parse. The tokens are computed by applying `tokenize_all_text_more_lc/2` to the `lexmatch/1` feature (or the `inputmatch/1` feature if there is no `lexmatch/1` feature). The example parse before and after calling add_tokens_to_phrase/2 is shown in Figure 3 and Figure 4.
- `parse_phrase_word_info/2`, `parse_phrase_word_info/9`—compute phrase words, phrase head words and a mapping from syntactic components to phrase words. This is shown

informally for the example in Figure 5 and Figure 12 and formally in Figure 13. These predicates use `extract_tokens/3` and `filter_tokens/3` which are similar to `extract_input_match/4` and `filter_input_match/4` (see below).

- `create_word_list/2`—creates a `wdl/2` term with first argument a list of words and second argument the lowercase version of the words.
- `extract_input_matches/5`, `extract_input_match/4`, `filter_input_match/4`—extract filtered (using `filter_input_match/4`) or unfiltered (using `extract_input_match/4`) `inputmatch/1` features from a parse. They produce not only a list of words but also a list of head words from the parse. A filtered extraction is one in which input matches with a syntactic tag of prep, det, aux, modal, compl, punc, num, conj and pron are ignored.
- `linearize_phrase/4`, `linearize_components/2`, `linearize_component/2`—split multi-word phrase items into single-word items in preparation for resolving the parse with mapped Metathesaurus strings. Linearization is needed since mapping does not respect syntactic boundaries. Figure 21 shows the linearization for the example. Note that the original mod

```
Original syntax:
   [not_in_lex([inputmatch([Anti]),tokens([anti])]),
   punc([inputmatch([-]),tokens([])]),
   mod([lexmatch([gastroesophageal reflux]),inputmatch([gastroe-
      sophageal,reflux]),tokens([gastroesophageal,reflux])]),
   head([lexmatch([implantation]),inputmatch([implanta-
      tion]),tokens([implantation])]),
   punc([inputmatch([.]),tokens([])])]
Original phrase map: [[1,1],[0,-1],[2,3],[4,4],[0,-1]]
Linearized syntax:
   [not_in_lex([inputmatch([Anti]),tokens([anti])]),
   punc([inputmatch([-]),tokens([])]),
   mod([lexmatch([gastroesophageal reflux]),inputmatch([gastroe-
      sophageal,reflux]),tokens([gastroesophageal])]),
   mod([lexmatch([gastroesophageal reflux]),inputmatch([gastroe-
      sophageal,reflux]),tokens([reflux])]),
   head([lexmatch([implantation]),inputmatch([implanta-
      tion]),tokens([implantation])]),
   punc([inputmatch([.]),tokens([])])]
Linearized phrase map: [[1],[0],[2],[3],[4],[0]]
```

Figure 21. A linearized phrase

phrase item has been split into two mod phrase items each containing one of the `tokens/1` *gastroesophageal* and *reflux*. All other features have been replicated.

### 2.7.3 Phrase access predicates

- `get_phrase_item_feature/3`, `get_phrase_item_name/2`, `get_phrase_item_subitems/2`, `new_phrase_item/3`, `get_subitems_feature/3`, `get_subitem_name/2`, `get_subitem_value/2`, `set_phrase_item_feature/4`, `set_subitems_feature/4`—manipulate phrase items and their subparts which are defined in Figure 22.

```
PhraseItem ::= <tag>(<SubItemList>)

SubItemList ::= <list> of <SubItem>

SubItem ::= <unary term>  (e.g., inputmatch/1, lexmatch/1, tokens/1, …)

Tag ::= adv | aux | compl | conj | det | error | ing | mod | modal | no_tag | not_in_lex | num |
        pastpart | pre | prep | pron | punc | shapes | verb | <tagger tokenizer tag>
```

Figure 22. Definition of PhraseItem

### 2.7.4 Tokenization predicates

- `parse_multi_word_into_words/2`—parses a single multi-word atom into a list of atoms by calling `nls_text:normalized_syntactic_uninvert_text/2` and `metamap_tokenization:tokenize_text_lc/2`. It parses ‘Gastro-esophageal reflux’ into [gastro, esophageal, reflux], ‘Implantation, NOS <1>’ into [implantation, ‘<1>’], and ‘Implants, Artificial’ into [artificial, implants].
- `tokenize_all_text/2`, `tokenize_text/2`, `tokenize_all_text_lc/2`, `tokenize_text_lc/2`—tokenize text which can be either an atom or a string. Results are lists of atoms or strings depending on the input type. …`_all_`… predicates process lists of text, and …`_lc` predicates produce lowercase results. `tokenize_text/2` tokenizes by breaking at spaces and hyphens and by ignoring colons. The results of applying it to the three examples above are [‘Gastro’, esophageal, reflux], [‘Implantation,’, ‘NOS’, ‘<1>’], and [‘Implants,’, ‘Artificial’].
- `tokenize_all_text_more/2`, `tokenize_text_more/2`, `tokenize_all_text_more_lc/2`, `tokenize_text_more_lc/2`—tokenize text similarly to the `tokenize_text/2` family above except that the tokenization is the wordind algorithm. The results of applying `tokenize_text_more/2` to the three examples above are [‘Gastro’, esophageal, reflux], [‘Implantation’, ‘NOS’, ‘1’], and [‘Implants’, ‘Artificial’].
- `tokenize_all_text_completely/2`, `tokenize_text_completely/2`, `tokenize_all_text_completely_lc/2`, `tokenize_text_completely_lc/2`—also tokenize text similarly to the `tokenize_text/2` family above except that all punctuation is isolated and kept. The results of applying `tokenize_text_completely/2` to the three examples above are [‘Gastro’, ‘-’, ‘esophageal’, ‘reflux’], [‘Implantation’, ‘,’, ‘NOS’, ‘<‘, ‘1’, ‘>’], and [‘Implants’, ‘,’, ‘Artificial’].