A First Step Towards Automated Knowledge Integration

Shan Wang
Department of Computer Sciences
University of Texas at Austin

May 2, 2006

1 Problem Description

Knowledge integration is the process of incorporating new information into a body of existing knowledge (Murray and Porter, 1989). This process is a cycle of knowledge acquisition, formulation, recognition, elaboration, and adaptation. Knowledge acquisition is the act of gathering facts. These facts must be formulated into the language of the underlying representation of some knowledge base. Recognition is a preliminary identification of what piece of existing knowledge is most closely related to the new information. Elaboration makes inferences that are based on combining the new information with the existing knowledge. Adaptation involves moderating inconsistencies between the knowledge base and the new information. The process aims to create knowledge bases used in reasoning.

Let's look at each of these steps in the context of an example. A student in a biology class acquires knowledge through reading a chapter about muscles in a biology text book. He must then interpret syntactic and semantic constructs in each sentence perhaps to learn that each muscle fiber is composed of both thick and thin myofilaments. Here, he has to reformulate the knowledge to understand the content. He recognizes from previous learning that every muscle is composed of muscle fiber. Thus the information about myofilaments is an extension of what he already knew about muscle fibers. In this scenario, there were no conflicts caused by the new knowledge, and there was no need to adapt the existing model. However, in the general case, new information may reveal gaps in existing knowledge or even refute existing knowledge.

This process can be repeated on a set of facts. As more knowledge is integrated into a system, it can make further inferences and align pieces of information which it previously had no basis to work with. Automating the process aims to create knowledge bases correctly and efficiently.

1.1 The Importance of Knowledge Integration

Currently, domain specific knowledge base construction requires domain experts to relay information to knowledge engineers who perform all of the steps of knowledge integration manually. Therefore, the knowledge engineer acquires knowledge from domain experts, formulates it,

sometimes incorrectly due to misunderstandings in communication, recognizes familiar pieces, and elaborates and adapts old knowledge. In the process, errors may be introduced that are discovered in testing. Worse still, the errors may not be discovered until the system is put to use.

The Halo Project, designed to evaluate the state-of-the-art in applied Knowledge Representation and Reasoning systems (Friedland et al., 2004), depended on knowledge engineers to encode knowledge from a domain, roughly 70 pages of a college-level chemistry textbook. The aim of the project was to subject systems to an AP Chemistry exam focused on stoichiometry and equilibrium reactions given only a limited time for knowledge base construction and testing. Not only is this process slow, in the Halo Pilot project, it proved to be a recurring cause of failure. The project reported several errors due to incorrect knowledge engineering such as asserting that only cations can be Lewis acids and assuming questions requiring computing an ionic equation must pertain to chemicals in solution (Barker et al., 2004).

It would be ideal to move the burden of knowledge integration off of the knowledge engineers and into the hands of the domain experts. However, representation languages that are expressive enough to capture complex concepts also require a steep learning curve. If domain experts were to be the sole encoders, similar problems may arise where domain experts misrepresent concepts due to unfamiliarity with the representation language.

Knowledge integration is an iterative, algorithmic process. A system that carries out this process is given a small knowledge base and some pieces of new knowledge. The system then tries to align the new information to the old in order to find the pieces of consequence. Using the result of the alignment as a guide, the system then integrates the two sets of information. If the knowledge integration process could be automated, knowledge base construction could be limited to defining a small domain-specific knowledge base with which to seed the knowledge integration process. As an algorithmic process, it no longer has to depend on communication between a domain expert and a knowledge engineer in order to acquire its data. Instead, it can gather knowledge from textbooks to minimize errors in misunderstanding between two people. In addition, the system does not need to be trained in the representation language as a domain expert encoding information must be. Therefore, knowledge integration is the ideal between the previous two methods of knowledge base construction: it has the directness of allowing a domain expert to encode knowledge while maintaining the expertise of a knowledge engineer's familiarity with the underlying representation mechanisms.

1.2 Why Is It Hard?

In knowledge acquisition and formulation, the goal is to have a machine algorithmically read text and interpret facts. This involves the difficult topic of natural language processing. One sentence can have multiple interpretations, yielding more than one potential representation in the underlying knowledge base.

Consider the sentence "The girl touches the boy with the flowers." This sentence's meaning is ambiguous. It is not clear, even to us, whether "with the flowers" is meant to describe the instrument of the touch action or as a descriptive characteristic of the boy receiving the action.

There are also less obviously ambiguous sentences such as “Water the flower with the brown leaves.” To a naive natural language processing system, it is again unclear whether “with the brown leaves” is meant to describe an instrument or a descriptive characteristic. People use their model of the world to rule out the possibility of using leaves to water a flower and thus perceive the sentence to have precisely one meaning. A sophisticated natural language system must be integrated with a reasoning system in order to prune out such syntactically possible but semantically incorrect interpretations.

Noun phrase interpretation, understanding the correct relationship between two nouns which make up the name of a concept, is another problem in natural language processing. For example, “olive oil” and “peanut oil” share the characteristics that they describe an oil made of the product named in the first noun. In contrast, “baby oil” specifies an oil used on babies, not one derived from them. In addition, anaphora resolution, determining the correct antecedent for each pronoun proves to be very difficult. Consider the set of sentences in Figure 1. Though people can determine that “She” and “her” cannot share an antecedent through contextual knowledge that a person cannot beat himself, there are still two valid interpretations of the phrase.

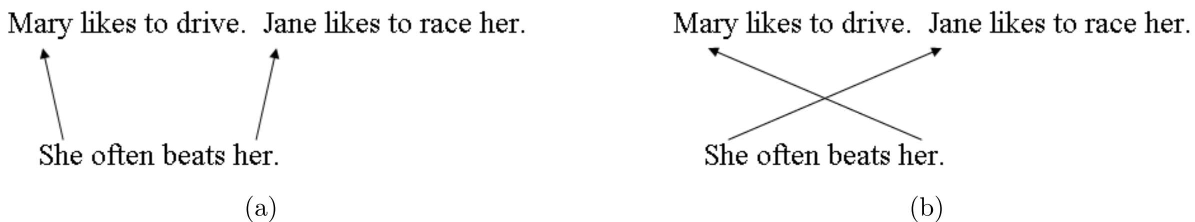


Figure 1: One interpretation, (a), chooses Mary to be the antecedent of “She” and Jane to be the antecedent of “her”. However, the reverse, (b), where Jane is the antecedent of “She” and Mary is the antecedent of “her” is equally plausible semantically.

These are obstacles that must be overcome by a natural language processing system. Either contextual clues must be used to deduce the most appropriate representation, all representations must be maintained for later evaluation, or the system must be constructed to be tolerant under incorrect interpretations.

Recognition potentially involves a search through the underlying knowledge base for the pieces of information most relevant to the new knowledge. Some past systems have taken the extreme approach of simply adding knowledge without a search for consequences or relevant old knowledge. This is unsatisfactory as no learning is taking place under the introduction of new knowledge. If the knowledge base is allowed to grow in this manner indefinitely, it simply becomes a mass of axioms, some redundant, some contradictory, which cannot support rigorous reasoning. The other extreme is to find all consequences and inconsistencies resulting from the introduction of new knowledge. However, this computationally infeasible. Finally, some systems, such as KI, have taken a middle road. It uses *views* to determine the relevant concepts among which one is heuristically selected. A view defines a subset of the knowledge base defined by several concepts that interact in some significant way (Murray and Porter, 1989).

Also, vocabulary often differs between the original knowledge base and the new information. For example, an initial knowledge base might use "Animal" as the label of a concept but in later learning, a natural language processor, which does not query the existing knowledge base, chooses to label the same concept "Creature". This complicates recognition as a system not only has to search for relevant components given only their names and their graph structures, but also make educated guesses about whether two apparently different components refer to the same concept.

At the foundation of knowledge integration, there must lie a store of rules for inference. A person spends his formative years learning concepts with which to interpret the knowledge he gains for the rest of his life. For example, he learns any object that acts as a container has an inside and an opening through which things may be moved into and out of the container. When he later learns that a bottle is a type of container, he immediately knows certain characteristics of how the bottle can interact with other materials.

To create such a foundation of inference rules, the English language must be condensed to a small, finite set of concepts. Not only must this set be small, it is crucial to select the correct core set of concepts so the expressiveness of the language is not lost. The English language is constantly evolving and growing. However, a foundation of inference rules should be fairly stable in order to be useful. A foundation that changes even yearly requires too much maintenance to provide a basis for knowledge integration. Any system that depends on this basis would have to evolve just as quickly to maintain compatibility. No algorithm exists for selecting concepts so building such a store of generic information requires educated guessing and rigorous tests.

2 Our System for Knowledge Integration

2.1 Assumptions

We are working under the assumption that the system is applied to learning from textbooks or other such manuals. Information given to the system continuously builds on previous knowledge and is restricted to a given domain. In addition, knowledge given is meant to be interpreted as general axioms and not in regard to one particular instance or individual. Knowledge is always assumed to be correct in its original representation in the acquisition phase.

We chose to focus on axioms about taxonomy and meronomy. These axioms include ones that describe the superclass/subclass relationships between concepts as well as the material or parts that a concept is composed of. These axioms are ideal for an initial system as they are fairly straightforward to represent. They restrict how to represent information, and they hold up well under introduction of new information. When the system deals with axioms such as purpose, there can potentially be multiple representation for a single idea. An increase in the expressiveness of the representation language increases the number of ways to represent equivalent ideas, as illustrated in Figure 2.

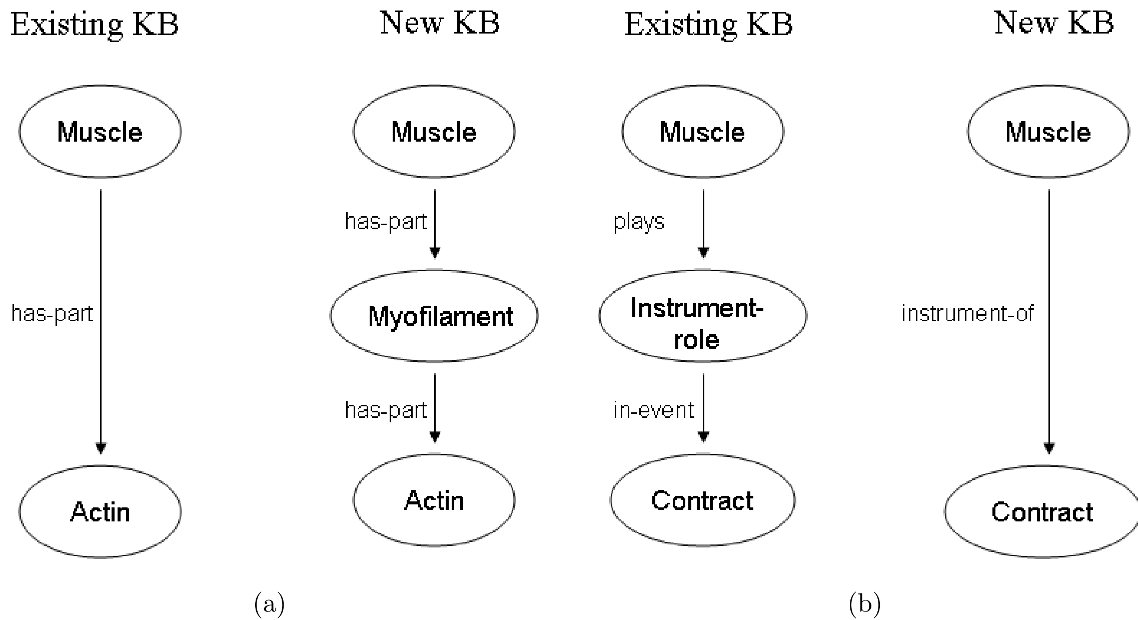


Figure 2: In (a), a graph which directly states that Muscles have a part named Actin is not greatly different from a representation containing additional information. In (b), equivalent knowledge takes on largely different forms. There relationships between the concepts in the two graphs are of completely different types. In addition, the Existing KB uses an extra concept, "Instrument-Role", to represent the same knowledge as compared to the New KB.

2.2 Experimental Setup

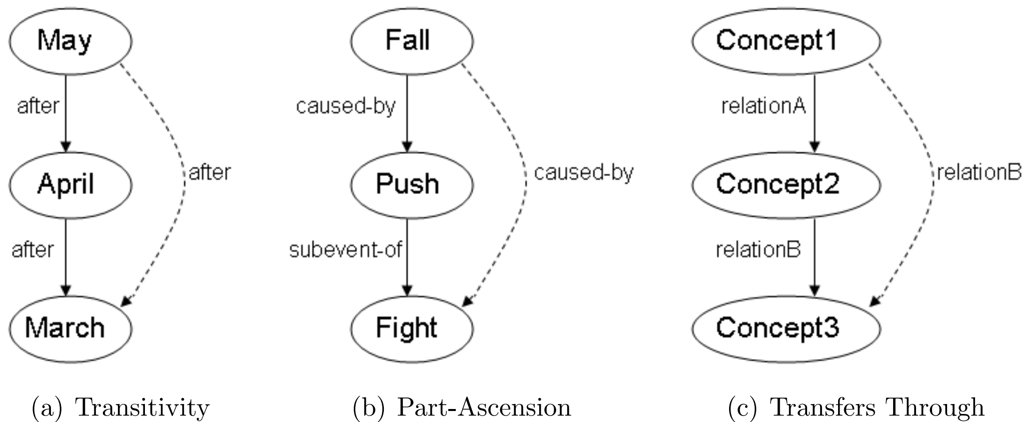
Our system uses the KM inferencing engine, along with the Component Library (Barker, Porter, and Clark, 2001) as the basic store of axioms for inference. The most basic representation of knowledge in this system is a triple, represented as `(Head-Concept Relation Tail-Concept)`. The Component Library is a set of generic concepts and a set of constraints or axioms about their properties. They represent a fundamental set of rules about the world which can be instantiated within specific domains. These axioms form a model of the world that drives the system's inferencing.

The KM inferencing engine is a frame system where each concept is defined as a class. The concepts can then be organized into a superclass/subclass hierarchy where multiple inheritance is allowed. Axioms can be asserted but inferencing can only be performed on an instance of a class. Therefore, knowledge in an axiom is not realized until that axiom's affected concepts are instantiated. Unnamed instances are referred to as Skolems whose standard representation is `_<ClassName>N` where `N` is a number. No two Skolems ever end in the same number in order to preserve uniqueness.

Our system also makes use of the semantic matcher developed by Peter Yeh (Yeh, Porter, and Barker, 2005). The matcher applies a predefined set of transformations against a given knowledge structure in order to try to achieve a maximal alignment with some target knowledge structure and returns a score reflecting the degree of the match. There is one basic type of transformation,

transfers through, which includes the special cases of transitivity and part-ascension (Yeh, Porter, and Barker, 2003).

Transfers through describe a transformation where a certain relationship is able to transfer through another. For example, enables transfers through causes. Transitivity is the special case where a relation transfers through itself. Part ascension is the special case there a relation transfers from a part to the whole. The semantic matcher also checks for subsumption through the superclass/subclass hierarchy. Figure 3 illustrates examples of each of the special cases as well as the general rule.



(a) Transitivity (b) Part-Ascension (c) Transfers Through

Figure 3: After is an example of a transitive relation (a). Since May is after April which is in turn after March, May must be after march. Caused-by is a relation which transfers by part-ascension through subevent-of. Therefore, since Push is a subevent of a Fight (b), and a Fall is caused by that Push, in essence, the Fall is caused by the Fight. In the general case shown in (c), there are special combinations of relations where one can transfer through another in sound reasoning, such as relationB transferring through relationA. As it is shown, part-ascension seems to occur in reverse of transfers through. However, since every relation automatically has an inverse in the system, we need only invert each relation to yield a similar graph.

Our project implements the recognition, elaboration, and adaptation pieces of the cycle of knowledge integration. We bypassed the difficult issues of knowledge acquisition and formulation (which involve natural language processing) by hand simulating the results of a parser applied to a textbook. The output of parsing is a set of triples, referred to as the source. All new knowledge is represented through instantiated Skolems instead of as KM axioms. Each Skolem must have an accompanying declaration of which class it belongs to. Also, every class involved must have a hierarchy declaration that is connected with an element of the Component Library. The class hierarchy can be overly conservative, placing elements further up in the hierarchy than their true position.

An initial knowledge base containing domain-specific information is required to seed the knowledge integration process. This file will be referred to as the target. The information in this file provides concepts that allow the recognition phase to operate. Without some seed concepts in the

system, it is difficult to align wholly new structures with those in the Component Library. Even when alignment is possible, the result may be suboptimal as the system must make conservative assumptions in order to maximize correctness. The hierarchy in this file may only be an extension of that of the Component Library. If a new concept is introduced as an intermediate superclass, it must be related to one already existing in the Component Library. Our system requires a fully defined hierarchy in order to operate. See Appendix A, B, and C for examples of knowledge base files.

Our chosen domain for experimentation purposes was muscle tissues. Our seed knowledge base was constructed from four introductory paragraphs about muscle tissue in a biology textbook. Our hand simulated parser results were from encoding a page of more detailed information about muscles presented later in the text.

2.3 Our Algorithm

The knowledge acquisition and formulation phases are not implemented by our system. The output of a hand simulated parser in the form of a set of triples, `source`, is given to our system as input along with the seed knowledge base, `target`.

1. Recognition.
   (a) Merge the class hierarchies of `source` and `target`. Concept names are allowed to differ but must at least be related as synonyms in the WordNet hierarchy. New hierarchy information is immediately asserted into the KM inferencing engine.
   (b) Use semantic matcher to recognize the overlapping sections of `source` and `target`.
   (c) Create global lookup table of matching concepts.
2. Elaboration.
   (a) For each unmatched triple from `source` check if either the head or tail concept of a triple has an entry in the lookup table. If so, assert an axiomatization of the triple into the knowledge base. If not, save the triple for future iterations of integration.
3. Adaptation.
   (a) When asserting knowledge, detect errors and contradictions. Report the error to allow user intervention to resolve the problem.

2.4 Recognition

To allow for more robust recognition by accounting for vocabulary differences between the initial and the new knowledge, we have added a name-matcher. Given two component names, the system first checks if the two names are identical. Failing this, it uses the semantic matcher's subsumption testing to determine if there is a superclass of one concept which is an identical match to another concept. Finally, it uses new utilities that interface with WordNet to check if they are synonyms of each other. WordNet arranges the English language in a tree-like structure. We are mostly

concerned with the hypernym, hyponym, and synonym categories for each word. Hypernyms and hyponyms define relationships similar to superclass and subclass, respectively, in the tree while synonyms identify siblings. If the system fails to find a match at the synonym level, it checks if one is a hypernym of another to a default depth of two levels away in the hierarchy. Because the granularity of WordNet is much finer than that of most knowledge bases, it is reasonable to expect hypernyms and hyponyms of a word to be used in substitution. Depth is limited to two as the higher the depth, the less likely that two words are being used to refer to the same concept and the more likely that one should be a subclass of the other. The vocabulary matching system ties into the match score calculation using the following formula:

Given:
`triple1 = (head1 relation1 tail1)`
`triple2 = (head2 relation2 tail2)`
d_{head} = taxonomic distance between head1 and head2
d_{tail} = taxonomic distance between tail1 and tail2

$$Score = \frac{1}{2}(\frac{1}{d_{tail}} + \frac{1}{d_{head}}).$$

Let us use the following example in Figure 4, `target`, and Figure 5, `source` to trace through each step of the algorithm. There are key points to note about these two pieces of knowledge. First, the two knowledge bases have slightly different vocabularies. The `target` calls a concept Muscular-Tissue whereas the `source` calls the same concept Muscle. Second, the `source` calls a concept Fiber where the `target` refers to it as Muscle-Fiber. In addition, Fiber is an instance (designated by the asterisks) of a Cell whereas Muscle-Fiber is a subclass of a Cell. These are obstacles that must be overcome by a system trying to align the two representations.

The `target` would be represented as follows:

```
(Muscular-Tissue has (superclasses (Tissue)))
(Skeletal-Muscle has (superclasses (Muscular-Tissue)))
(Myofibril has (superclasses (Entity)))
(every Muscular-Tissue has
  (has-part ((a Aggregate with
             (element-type (Muscle-Fiber))
             (number-of-elements ((a Entity)))
             (element ((a Muscle-Fiber)))))))
(Muscle-Fiber has (superclasses (Cell)))
(every Muscle-Fiber has
  (has-part ((a Aggregate with
             (element-type (Myofibril))
             (element ((a Myofibril)))))))
```

The `source` would be represented as follows:

```
(Skeletal-Muscle superclasses Muscle)
```

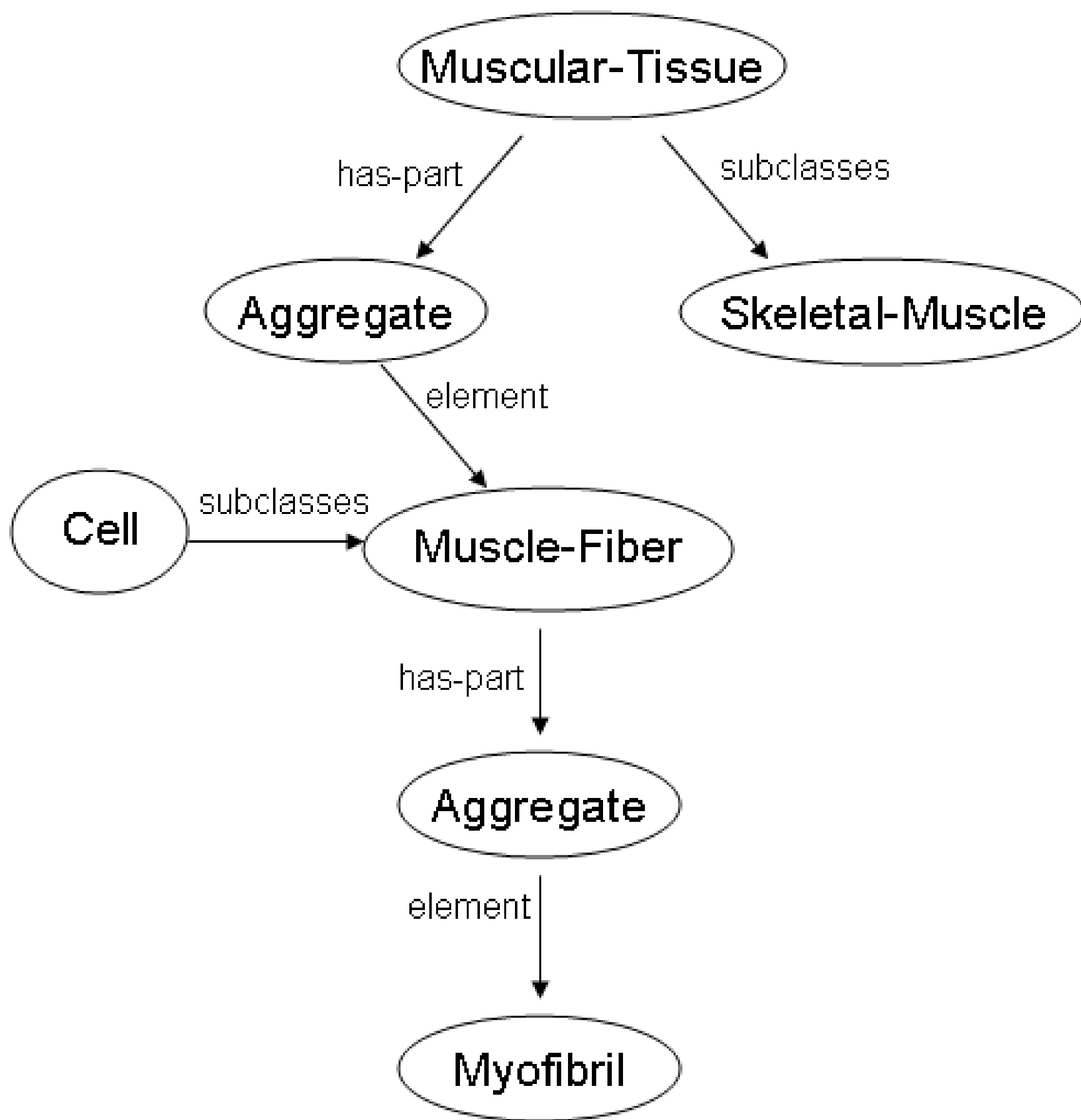


Figure 4: Graph of old knowledge base: `target`.

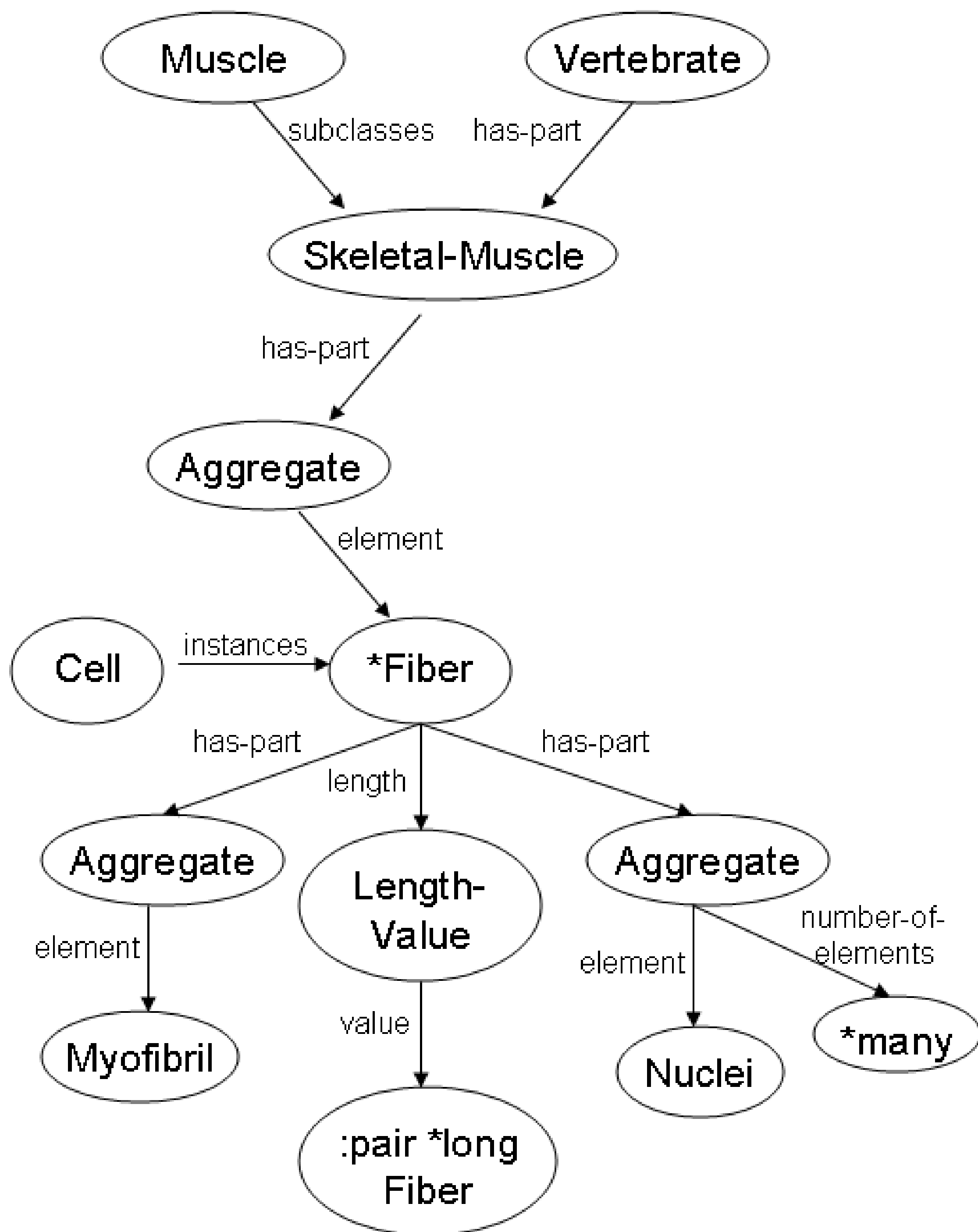


Figure 5: Graph of new knowledge: `source`.

```
(Myofibril superclasses Entity)
(Nuclei superclasses Living-Entity)
(Vertebrate superclasses Animal)
(_Skeletal-Muscle01 instance-of Skeletal-Muscle)
(_Vertebrate02 instance-of Vertebrate)
(_Skeletal-Muscle01 is-part-of Vertebrate02)
(_Aggregate03 instance-of Aggregate)
(_Skeletal-Muscle01 has-part _Aggregate03)
(_Fiber04 instance-of Cell)
(_Aggregate03 element _Fiber04)
(_Aggregate05 instance-of Aggregate)
(_Length-Value06 instance-of Length-Value)
(_Aggregate07 instance-of Aggregate)
(_Fiber04 has-part _Aggregate05)
(_Fiber04 length _Length-Value06)
(_Fiber04 has-part _Aggregate07)
(_Length-Value06 value (:pair *long Fiber))
(_Myofibril08 instance-of Myofibril)
(_Nuclei09 instance-of Nuclei)
(_Aggregate05 element _Myofibril08)
(_Aggregate07 element _Nuclei09)
(*many instance-of Quantity-Constant)
(_Aggregate07 number-of-elements *many)
```

When the system first tries to merge the class hierarchies of the `source` and `target` knowledge bases, the name matcher discovers that Muscular-Tissue and Muscle are in fact synonyms. It installs the two concepts as synonyms of each other in the knowledge base. This serves as a clue for future knowledge integration. The merged hierarchy is necessary for the semantic matcher to perform properly. Transformation rules in the semantic matcher are defined based on concepts in the Component Library. For example, the transitivity of the `has-part` relation only applies to a concept whose superclass hierarchy includes `Entity`. The new knowledge sometimes introduces concepts which do not yet exist in the reasoning engine and must be related to the Component Library so all applicable transformations are tried in the alignment process.

At the end of recognition, a hash table such as the following is created:

```
((_Aggregate03 -> (_Aggregate12 . 3/4))
 (_Aggregate05 -> (_Aggregate16 . 3/4))
 (_Fiber04 -> (Muscle-Fiber . 3/4))
 (_Myofibril08 -> (Myofibril . 1))
 (_Skeletal-Muscle01 -> (Skeletal-Muscle . 1)))
```

The number represents the fitness of the match. Since subsumption of the concept `Muscle-Fiber` with its superclass `Cell` was required to achieve the match between `_Fiber04` and `Muscle-Fiber`,

these concepts received an imperfect score of 3/4. The aggregate concepts matched as a result of _Fiber04 and Muscle-Fiber and thus their scores were affected as well.

Aggregates are treated specially by the system. Whereas all other concepts are matched to classes, aggregates are matched to a particular instance. An aggregate is a way to represent a collection of items in the reasoning engine. As a result, aggregates have the special property that knowledge learned about a particular instance almost never applies to the general concept of an aggregate.

2.5 Elaboration and Adaptation

After all matching knowledge is identified in the recognition phase, the elaboration and adaptation phases focus on processing the unmatched information. The head and tail concepts of every remaining triple is checked against the hash table for its matching class concept. If one is found, the triple is axiomatized and asserted into the knowledge base. If neither the head nor tail concepts yield a match in the hash table, it is saved for future iterations of knowledge integration.

For our example, these are the triples which would be integrated into the system at this point. These triples represent knowledge that is learned by the system.

```
(_Fiber04 length _Length-Value06)
(_Fiber04 has-part _Aggregate07)
(_Aggregate07 element _Nuclei09)
(_Aggregate07 number-of-elements *many)
(*many instance-of Quantity-Constant)
(_Vertebrate02 instance-of Vertebrate)
(_Skeletal-Muscle01 is-part-of Vertebrate02)
```

However, the triples that are remaining are of more interest.

```
(_Length-Value06 instance-of Length-Value)
(_Length-Value06 value (:pair *long Fiber))
```

The triples that remain are related to knowledge that was integrated. However, they cannot be integrated in the current iteration for one of two reasons.

1. The system can only extend its knowledge by one level at a time (with the exception of aggregates as discussed below). This is due to the fact that the system considers triples independently of one another. Since either the head or the tail concept of the triple must have an entry in the hash table, we extend the graph of knowledge by adding one concept to its peripheral per iteration of integration. If no transformations are required, the number of iterations required to fully integrate is: the maximum depth of concepts that outlay the overlapping subgraphs of source and target. If transformations are required, the number of iterations depends on the particular transformation used in semantic matching.

2. A triple cannot be integrated if it is composed of concepts which have no overlapping information with the `target` except possibly for its class hierarchy. Semantic matching does not consider the class hierarchy when aligning two sets of knowledge. Therefore, a triple that fails due to this second reason can only be integrated once more knowledge is acquired that results in an overlap with `target`.

Aggregates are again treated as a special case by our system. Since aggregate knowledge cannot be asserted piecewise to maintain its correctness, our system will assert all information regarding aggregates at once. With our example, the system will consider the following triples as a group:

```
(_Fiber04 has-part _Aggregate07)
(_Aggregate07 element _Nuclei09)
(_Aggregate07 number-of-elements *many)
(*many instance-of Quantity-Constant)
```

It will then be able to assert the axiom

```
(every Muscle-Fiber
  (has-part ((a Aggregate with (element-type (Nuclei))
                               (element ((a Nuclei)))
                               (number-of-elements ((a Quantity-Constant)))))))
```

As shown by this example, the system always generalizes every concept to its class in order to create an axiom. It does so either through the use of the hash table created from the recognition phase or through `instance-of` triples given in the `source` file.

In this example, a second iteration with the complete `source` and an updated `target` will be able to integrate all triples which failed due to the first reason. An updated `target` is the original `target` file with the newly learned axioms included. This second iteration would completely integrate the example `target` and `source`. In practice, the system should be allowed to iterate over the `source` information until it produces a list of triples with only the relation `instance-of` or the same list of unmatched triples is produced twice in a row. At this point, any triples left unmatched fail solely due to lack of information.

Though this second iteration fully integrates the two pieces of knowledge, there will be one erroneous assertion:

```
(every Length-Value has
  (value ((:pair *long Fiber))))
```

The introduction of this assertion stems from the fact that this piece of knowledge violates one of our assumptions: that every triple can be treated independently. This limitation is discussed further in the Discussion section.

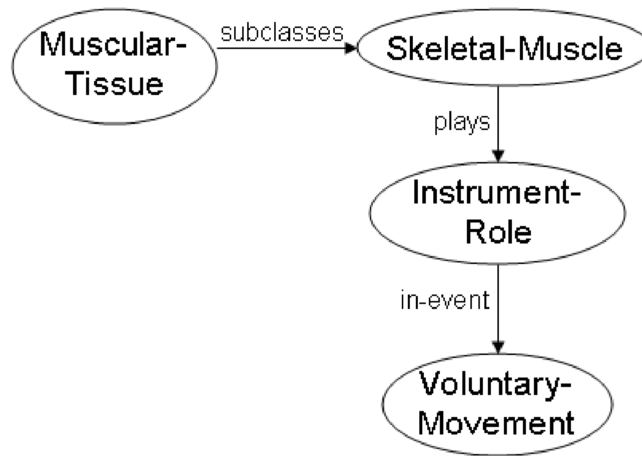


Figure 6: Testcase 1 `target`

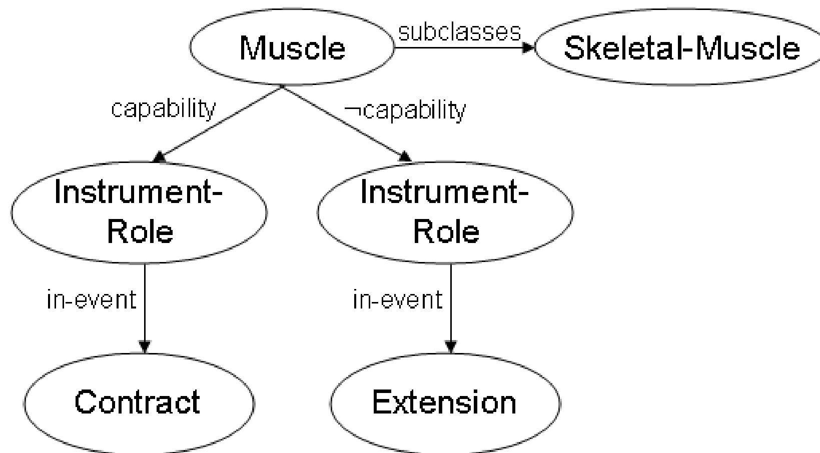


Figure 7: Testcase 1 `source`

3 Results

In development, we used two additional testcases to gauge the effectiveness of our system. The first is detailed by Figures 6 and 7. This testcase results in no knowledge integration. The triples fail because the similarities in the knowledge depend on the class hierarchy. As viewed by the semantic matcher, there is no common information between the graphs. Consequently, no matches can be made. In addition, these triples are largely devoid of taxonomy and meronomy information. Our system currently lacks the ability to recognize similarities between such knowledge bases even though the semantic matcher includes transformations beyond that involving simply meronomy information. Even if matches were possible, the system lacks the appropriate algorithms to axiomatize information involving non-meronomy relations.

Our second testcase is detailed by Figures 8 and 9. This testcase requires three iterations of the system to be integrated. However, a mistake occurs in the process of integrations. Since the

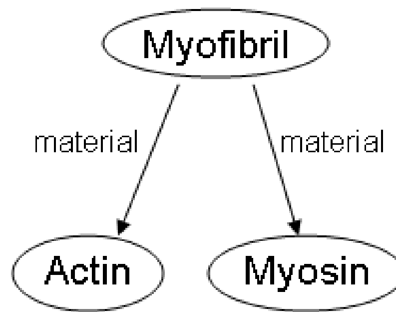


Figure 8: Testcase 2 `target`.

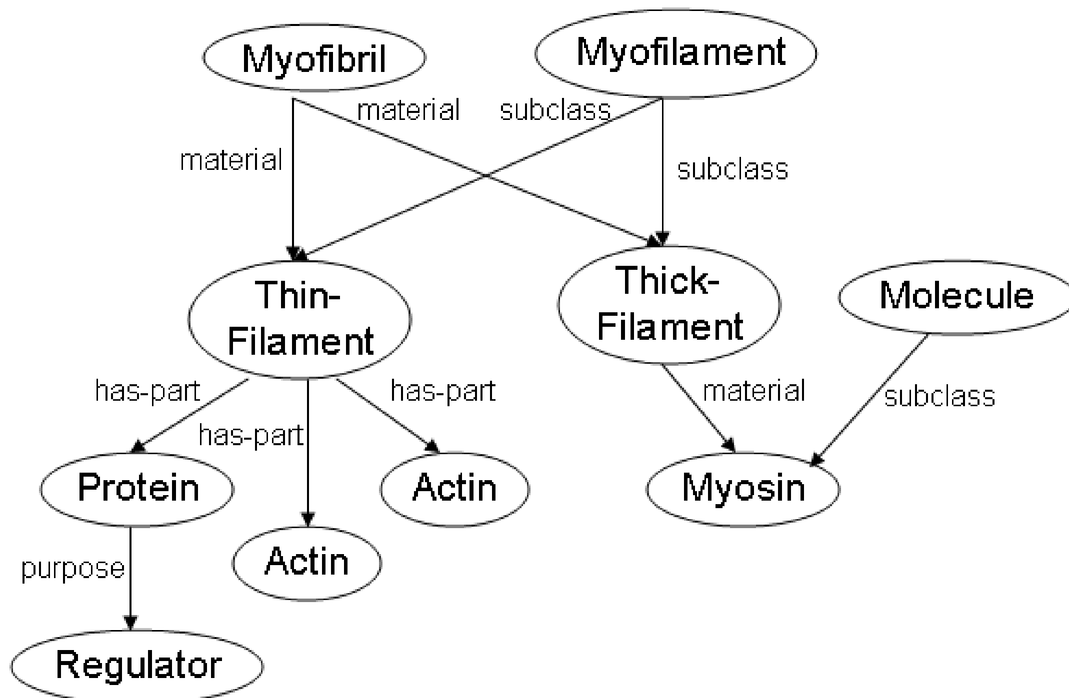


Figure 9: Testcase 2 `source`.

two `Actin` concepts have nothing to distinguish them, the system regards the second `Actin` as a redundant match and therefore does not try to assert it as an axiom. Similar to the erroneous assertion for our example in the previous section, this mistake is due to our assumption that triples can be treated independently. However, in this case, the mistaken assertion results in an absence of information.

4 Discussion & Future Work

Our system was developed with meronomy and taxonomy relations in mind. Therefore, relations of these types are handled the best. Any information which can be integrated as a result of learning that occurs through these relations is only a side benefit. More rules for recognizing

equivalent representations must be added to further the abilities of the system. Along with these transformation rules, rules for integrating information matched under such transformations must also be added. To return to Figure 2, once the system can recognize that there are two equivalent ways to represent `instrument`, it needs formulae for integrating any information represented as relation arcs between `Muscle` and `Contract`.

The amount of new information supplied to the system at each integration can affect the system's correctness. It is better to give the system more information than it needs, allowing some new information to go unintegrated, than to supply too little. As a rule of thumb, if information is being extracted from a textbook, supplying the system with an entire section or chapter of information is sufficient. Giving the system only one triple of new information at a time would likely yield mistakes.

Our system also currently fails to properly perform explanation based learning. This type of learning is a method for extracting general rules from explaining individual examples and generalizing the explanation to form an axiom to apply in future, similar situations (Russell and Norvig, 2003).

Consider a knowledge base that contains the concept of a Bear which, through inheritance, is known to have a part called Paw. If the following triples are presented as new knowledge:

```
(_Bear01 has-part _Paw02)
(_Paw03 color *white)
(_Bear01 location *North-Pole)
```

`(_Bear01 has-part _Paw02)` would yield a match in the knowledge base. From here, our system would assert that every Paw has the color white. It would also assert that every Bear has the location of North-Pole. This is based on our assumption that every piece of knowledge we are given is a general axiom presented to us in the form of an instantiated fact. However, this assumption is not always valid. An appropriate interpretation would consider the interaction between the triples, that some triples serve as explanatory support for the others. With explanation based learning, the triples are first considered as a group to yield the interpretation that "Every bear located at the North Pole has white paws." In addition, each concept may be generalized to form a general axiom. If bear is generalized to animal, North Pole to a cold place, and paws to limbs, the knowledge base gains the axiom "Every animal located in a cold place has white limbs."

However, this process requires that multiple triples are considered together. It is also difficult to surmise how many triples are truly consequential. Consider adding a triple representing "A bear likes honey" to the above set. To act correctly, the system would have to conclude that whether or not a bear likes honey has no relevance to the color of his paws.

An axiom acquired through explanation based learning is much more general than the original example presented to the system. However, a system capable of this type of learning must have sufficient background knowledge to produce the explanation to begin with. Therefore, the agent does not learn any factually new information (Russell and Norvig, 2003).

The assumptions made by our system place very high demands on the natural language processor. In order for recognition, elaboration, and adaptation to occur, the natural language processor must guarantee correct interpretations of text in the parsed output it generates. In addition, it must have an understanding of the Component Library as our system requires that class hierarchies be fully related to the Component Library.

Currently, the system only allows Aggregate triples to be axiomatized as a group. A simple extension that would greatly reduce errors in axiomatization is including more exceptions. To find the container of an Aggregate, the system currently queries the `is-part-of` relation. This can be extended to many other concepts in the Component Library by installing rules for the concepts that must be handled as a group along with the relation the system must use to find the concept's container. We have currently identified that all instances of `Property-Value` and `Role` should be axiomatized as a group of triples instead of individually. As previously demonstrated, the following triples are erroneously axiomatized.

```
(_Fiber04 length _Length-Value06)
(_Length-Value06 instance-of Length-Value)
(_Length-Value06 value (:pair *long Fiber))
```

By adding the ability to consider triples dealing with `Property-Value` in groups, the system would be able to axiomatize the triples correctly as:

```
(every Muscle-Fiber has
  (length ((a Length-Value with
            (value ((:pair *long Fiber)))))))
```

Adding synonym annotation would enable the system to apply information about past vocabulary matches to future integrations. For example, it would be desirable for the system to record that `Muscle` and `Muscular-Tissue` were used to represent the same concept. In later integrations in the same domain, this synonym connection could be reinforced or used to be able to infer larger matches. In a separate integration process within a different domain, the system may be able to apply this knowledge when it encounters the concept `Muscular-Tissue`.

The current system cannot integrate based solely on hierarchical matches since the semantic matcher does not use superclass/subclass information when identifying matching concepts. If there is sufficient overlap between the hierarchical structures of two knowledge bases, some integration should occur.

5 Summary

The aim of automated knowledge integration is to create knowledge bases quickly and correctly. We have defined the steps involved in an automated knowledge integration system as knowledge acquisition, knowledge formulation, knowledge recognition, and elaboration and adaptation. In knowledge acquisition and knowledge formulation, the largest difficulty is executing sophisticated natural language processing. To perform knowledge recognition, there may be a large search as

well as a need to correlate different vocabularies. Overall, an inferencing model of the world is needed to support each step. We have outlined an algorithm for executing the recognition and elaboration and adaptation steps of knowledge integrations and implemented a prototype system focused on meronomy and taxonomy relations. Our system's performance has been analyzed with success against a set of examples in the domain of muscles in biology.

References

Ken Barker, Bruce Porter, and Peter Clark. A Library of Generic Concepts for Composing Knowledge Bases. In *K-CAP*, 2001.

Ken Barker, Vinay K. Chaudhri, Shaw Yi Chaw, Peter E. Clark, James Fan, David Israel, Sunil Mishra, Bruce Porter, Pedro Romero, Dan Tecuci, and Peter Yeh. A Question-Answering System for AP Chemistry: Assessing KR&R Technologies. In *Ninth International Conference on the Principles of Knowledge Representation and Reasoning*, 2004.

Noah S. Friedland, Paul G. Allen, Micahel Witbrock, Gavin Matthews, Nancy Salay, Pierluigi Miraglia, Jurgen Angele, Steffen Staab, David Israel, Vinay Chaudhri, Bruce Porter, Ken Barker, and Peter Clark. Towards a Quantitative, Platform-Independent Analysis of Knowledge Systems. In *Ninth International Conference on the Principles of Knowledge Representation and Reasoning*, 2004.

Kenneth S. Murray and Bruce W. Porter. Controlling Search for the Consequences for New Information During Knowledge Integration. In *Sixth International Workshop on Machine Learning*, pages 290–295, 1989.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

Peter Z. Yeh, Bruce Porter, and Ken Barker. Using Transformations to Improve Semantic Matching. In *Second International Conference on Knowledge Capture*, 2003.

Peter Z. Yeh, Bruce Porter, and Ken Barker. Matching Utterances to Rich Knowledge Structures to Acquire a Model of the Speaker's Goal. In *Proceedings of Third International Conference on Knowledge Capture*, 2005.

A Testcase 1

A.1 source

```
(Skeletal-Muscle superclasses Muscle)
(Myofibril superclasses Entity)
(Nuclei superclasses Living-Entity)
(_Skeletal-Muscle01 instance-of Skeletal-Muscle)
(_Vertebrate02 instance-of Vertebrate)
(_Skeletal-Muscle01 is-part-of _Vertebrate02)
(_Aggregate03 instance-of Aggregate)
(_Skeletal-Muscle01 has-part _Aggregate03)
(_Fiber04 instance-of Cell)
(_Aggregate03 element _Fiber04)
(_Aggregate05 instance-of Aggregate)
(_Length-Value06 instance-of Length-Value)
(_Aggregate07 instance-of Aggregate)
(_Fiber04 has-part _Aggregate05)
(_Fiber04 length _Length-Value06)
(_Fiber04 has-part _Aggregate07)
(_Length-Value06 value (:pair *long Fiber))
(_Myofibril08 instance-of Myofibril)
(_Nuclei09 instance-of Nuclei)
(_Aggregate05 element _Myofibril08)
(_Aggregate07 number-of-elements *many)
(_Aggregate07 element _Nuclei09)
```

A.2 target

```
(Muscular-Tissue has
  (superclasses (Tissue)))

(Skeletal-Muscle has
  (superclasses (Muscular-Tissue)))

(Myofibril has
  (superclasses (Entity)))

(every Muscular-Tissue has
  (has-part ((a Aggregate with
               (element-type (Muscle-Fiber))
               (number-of-elements ((a Entity)))
               (element ((a Muscle-Fiber)))))))

(Muscle-Fiber has
  (superclasses (Cell)))
```

```
(every Muscle-Fiber has
  (has-part ((a Aggregate with
               (element-type (Myofibril))
               (number-of-elements ((a Entity)))
               (element ((a Myofibril)))))))
```

B Testcase 2

B.1 source

```
(Myofibril superclasses Tangible-Entity)
(Myofilament superclasses Tangible-Entity)
(Thin-Filament superclasses Myofilament)
(Thick-Filament superclasses Myofilament)
(Myosin superclasses Tangible-Entity)
(Myosin superclasses Molecule)
(Actin superclasses Tangible-Entity)
(Protein superclasses Substance)
(Regulator superclasses Role)
(_Myofibril01 instance-of Myofibril)
(_Myofilament02 instance-of Myofilament)
(_Thin-Filament03 instance-of Thin-Filament)
(_Thick-Filament04 instance-of Thick-Filament)
(_Myosin05 instance-of Myosin)
(_Actin06 instance-of Actin)
(_Actin07 instance-of Actin)
(_Protein08 instance-of Protein)
(_Regulator09 instance-of Regulator)
(_Myofibril01 material _Thin-Filament03)
(_Myofibril01 material _Thick-Filament04)
(_Thin-Filament03 has-part _Actin06)
(_Thin-Filament03 has-part _Actin07)
(_Thin-Filament03 has-part _Protein08)
(_Protein08 purpose _Regulator09)
(_Thick-Filament04 material _Myosin05)
```

B.2 target

```
(Myofibril has
  (superclasses (Living-Entity)))
(every Myofibril has
  (material ((a Actin) (a Myosin))))
```

C Testcase 3

C.1 source

```
(Muscle superclasses Tangible-Entity)
(Skeletal-Muscle superclasses Muscle)
(_Muscle01 instance-of Muscle)
(_Instrument-Role02 instance-of Instrument-Role)
(_Instrument-Role03 instance-of Instrument-Role)
(_Muscle01 capability _Instrument-Role02)
(_Contract04 instance-of Contract)
(_Instrument-Role02 in-event _Contract04)
(_Extension05 instance-of Extension)
(not (_Muscle01 capability _Instrument-Role03))
(_Instrument-Role03 in-event _Extension05)
```

C.2 target

```
(Muscular-Tissue has
  (superclasses (Tissue)))

(Skeletal-Muscle has
  (superclasses (Muscular-Tissue)))

(every Skeletal-Muscle has
  (plays ((a Instrument-Role with
            (in-event ((a Voluntary-Movement)))))))

(Voluntary-Movement has
  (superclasses (Move Intentional)))
```

D Source Code

```
(in-package :km)

;;---------------------------------------------------------------------
;; GLOBAL PARAMETER for the hash table of source-to-target concept
;; matches.
;; To do a lookup, use (gethash <key> *match-table*); if nil, no entry
;; Use (setf (gethash <key> *match-table*) <new-value>) to update table
(defvar *match-table* (make-hash-table :test #'equal))
(defvar *equivalent-relations-table* (make-hash-table :test #'equal))

;;------------------------------------------------------------------
;; DESC: Given a base-kb-file and a triples-file, performs
;;       recognition and elaboration & adaptation in order to
;;       integrate the two knowledge bases
;; INPUT: base-kb-file = file in *.km format (referred to
;;                       as old knowledge or target)
;;        triples-file = list of triples (referred to as
;;                       new knowledge or source)
;; OUTPUT: triples from triples-file which were not axiomatized
;;         (integrated)
;;------------------------------------------------------------------
(defun integrate (base-kb-file triples-file)
  (let (matches triples concepts matched-triples
        target source-heirarchy target-heirarchy
        more-unmatched unmatched-source aggregates)
    (setf concepts (remove-duplicates
                     (get-concepts (file-to-list base-kb-file) nil)))
    (setf triples (file-to-list triples-file))
    (load-kb base-kb-file)
    (setf target-heirarchy (get-superclasses concepts))
    (setf source-heirarchy (merge-heirarchy target-heirarchy
                                            (get-triple-relation triples
                                                                 '|superclasses|
                                                                 '())))
    (assert-heirarchy (car source-heirarchy))
    (setf triples (do-replacements triples (cdr source-heirarchy)))
    (setf triples (remove-triple-relation triples '|superclasses| '()))
    (new-situation)   ; to make Inertial-Fluent slots visible to gather-graph
    (dolist (concept concepts)
      (setf target
      (append (gather-graph (first (km `(|a| ,concept))) nil 5)
              target)))
    (setf matches (semantic-matcher (convert-to-peters-form triples)
```

```
                                    (convert-to-peters-form target)))
    (pprint (list "Matches: " matches))
    (setf matched-triples (build-table matches))
    (pprint "Evaluate Matches:")
    (setf aggregates
      (evaluate-matches (cadr matched-triples) (car matched-triples)))
    (pprint "Evaluate All Unmatched:")
    (setf unmatched-source
      (remove-triples
       (remove-triples aggregates
        (remove-triple-relation (evaluate-flatten (car matched-triples))
                                '|instance-of|
                                '()))
       triples))
    (setf more-unmatched
      (evaluate-all-unmatched (make-a-list
                                (get-triple-relation unmatched-source
                                                     '|instance-of|
                                                     '())
                                '())
                              (remove-triple-relation unmatched-source
                                                      '|instance-of|
                                                      '()))))
    (setf unmatched-source (append more-unmatched
                                   (get-triple-relation unmatched-source
                                   '|instance-of|
                                   '())))
    (pprint "Evalute Aggregates")
    (process-unmatched-aggregates unmatched-source
                                  (make-a-list
                                    (get-triple-relation unmatched-source
                                                         '|instance-of|
                                                         '())
                                    '()))
    ))

;;------------------------------------------------------------------
;; DESC: Determines whether two triples can be matched together by
;;       querying WordNet to see if words in each triple are
;;       related. Depth indicates how far we seek in the hypernym
;;       tree of WordNet for each word.
;; INPUT: triple1 = a triple
;;        triple2 = a triple
;;        depth = a non-negative integer, preferably small (ie [0, 3])
;;            where a triple = (<concept> <relation> <concept>)
```

```
;;                  <concept> = <symbol> | (<symbol> . <id>)
;;                  <relation> = <symbol>
;; OUTPUT: NIL is returned if the two triples cannot be matched
;;         A list of the form
;;             ((<spec> . <spec>) . <score>)
;;         where <spec> = first | second
;;         <score> = indicating the degree of match based on
;;         the hypernym distance is returned otherwise.
;; NOTE: It is important to realize that a negative score does not denote
;;       a negative score! A negative score means that the triples matched
;;       inversely. The absolute value of the score will be used to
;;       calculate the final score of the match.
;;-----------------------------------------------------------------
(defun name-matcher (triple1 triple2 &optional (depth 1) (match-type? 'weak))
  (let (alignment dist-frame dist-value)
    (wn:init)
    (setf alignment (line-up-relation triple1 triple2))
    (if (not (null alignment))
        (progn
          (setf triple1 (caar alignment))
          (setf triple2 (cadar alignment))))
    (cond
     ((null alignment)
      nil)
     ((equal triple1 triple2)
      (cons '(first . first) 1))
     ((null (setf dist-frame (name-match-frame (triple-head triple1)
                                               (triple-head triple2)
                                               depth
                                               match-type?)))
      nil)
     ((null (setf dist-value (name-match-frame (triple-tail triple1)
                                               (triple-tail triple2)
                                               depth
                                               match-type?)))
      nil)
     (t
      (cons '((car dist-frame) . (car dist-value))
      (* (cdr alignment)
         (/ (+ (/ 1 (+ (cdr dist-frame) 1))
                  (/ 1 (+ (cdr dist-value) 1))) 2)))))))

;;-----------------------------------------------------------------
;; DESC: Checks to see if one triple uses a subslot relation of
;;       the other, or if one triple is an inverse of the other,
```

```
;;        and if so, returns a list of aligned triples
;; INPUT: triple1 = a triple
;;        triple2 = a triple
;;            where a triple = (<concept> <relation> <concept>)
;;                  <concept> = <symbol> | (<symbol> . <id>)
;;                  <relation> = <symbol>
;; OUTPUT: NIL if the relation cannot be lined up
;;         A list of the form
;;             ((<triple> <triple>) . <relation-match?>)
;;         is returned otherwise, where
;;         relation-match? = 1 if the relations or subslots match
;;                         = -1 if the inverse relation or subslots match
;; NOTE: Modelled after function subsume-p in semantic-matcher.lisp
;;-----------------------------------------------------------------
(defun line-up-relation (triple1 triple2)
  (let ((rel1 (triple-relation triple1))
        (rel2 (triple-relation triple2))
        rel1-inv rel2-inv score)
    ;; Throw away the not if there is one.
    (if (consp rel1) (setf rel1 (second rel1)))
    (if (consp rel2) (setf rel2 (second rel2)))
    (setf rel1-inv (invert-slot rel1))
    (setf rel2-inv (invert-slot rel2))
    ;; Line up the relation, and also check for subslots.
    (cond
     ;; Equal or a subslot
     ((or (eql rel1 rel2)
          (member rel1 (get-all-subslots rel2) :test #'eql)
          (member rel2 (get-all-subslots rel1) :test #'eql))
      (cons (cons (list (triple-head triple1) rel1 (triple-tail triple1))
                  (list (list (triple-head triple2) rel2 (triple-tail triple2))))
            1))
     ;; Inverse is equal or a subslot
     ((or (eql rel1-inv rel2)
          (member rel1-inv (get-all-subslots rel2) :test #'eql)
          (member rel2 (get-all-subslots rel1-inv) :test #'eql))
      (cons (cons (list (triple-tail triple1) rel1-inv (triple-head triple1))
                  (list (list (triple-head triple2) rel2 (triple-tail triple2))))
            -1))
     ;; Special cases
     ((setf score (or (equiv-relation rel1 rel2)
                      (equiv-relation rel1-inv rel2-inv)))
      (cons (cons (list (triple-head triple1) rel1 (triple-tail triple1))
                  (list (list (triple-head triple2) rel2 (triple-tail triple2))))
            score))
```

```
     ;; Inverse special case
     ((setf score (or (equiv-relation rel1-inv rel2)
                      (equiv-relation rel1 rel2-inv)))
      (cons (cons (list (triple-tail triple1) rel1 (triple-head triple1))
                  (list (list (triple-head triple2) rel2 (triple-tail triple2))))
            (* -1 score)))
     (t
      nil))))

(defvar *equivalent-relations* '(
         (|has-part| |material| 0.75)
         ))

;;-------------------------------------------------------------------
;; INPUT: rel1 = the name of a relation
;;        rel2 = the name of a relation
;; OUTPUT: A score between 0 and 1 if the two relations are equivalent;
;;         NIL otherwise
;;-------------------------------------------------------------------
(defun equiv-relation (rel1 rel2 &optional (equivalencies
                                             *equivalent-relations-table*))
  (let ((hash-rel1 (gethash rel1 equivalencies))
        (hash-rel2 (gethash rel2 equivalencies)))
    (if (= 0 (hash-table-count equivalencies)) (build-equiv-relations))
    (cond
     ((and (consp hash-rel1) (eql rel2 (car hash-rel1)))
      (cadr hash-rel1))
     ((and (consp hash-rel2) (eql rel1 (car hash-rel2)))
      (cadr hash-rel2))
     (t
      NIL))))

(defun build-equiv-relations ()
  (dolist (line *equivalent-relations*)
    (setf (gethash (car line) *equivalent-relations-table*) (cdr line))))

;;-------------------------------------------------------------------
;; DESC: Determines if there is any possible way that two concepts
;;       could match, either by subsumption or being semantically
;;       related.
;; INPUT: concept1 = a concept
;;        concept2 = a concept
;;        depth = a small non-negative integer
;;        match-type? (optional) = the type of the match (i.e. strong
;; OUTPUT: NIL if the two concepts cannot match
```

```
;;          A pair (<spec> . <score>) is returned otherwise, where
;;          <spec> = first | second
;;          and score indicates the "distance" between the
;;          concepts (either taxonomic distance or distance in
;;          WordNet's hypernym tree, depending on how they match).
;;-----------------------------------------------------------------
(defun name-match-frame (concept1 concept2 depth match-type?)
  (let ((word1 concept1) (word2 concept2) result)
    (if (consp concept1) (setf word1 (car concept1)))
    (if (consp concept2) (setf word2 (car concept2)))
    (setf word1 (symbol-name (reformat-word word1)))
    (setf word2 (symbol-name (reformat-word word2)))
    (cond
     ((setf result (subsume-frame-p concept1 concept2 match-type?))
      result)
     ((setf result (name-match-frame-p word1 word2 depth match-type?))
      result)
     (t
      NIL))))

;;-----------------------------------------------------------------
;; DESC: Determines if the two concepts can be matched together
;;       by queyring WordNet to see if one word is related to the
;;       other.
;; INPUT: concept1 = a concept
;;        concept2 = a concept
;;        depth = a non-negative integer
;;        match-type? = strong | weak
;;        where a concept = <symbol> | (<symbol> . <id>)
;; OUTPUT: NIL if the two names cannot be matched
;;         A pair (<spec> . <score>) is returned otherwise, where
;;         <spec> = first | second
;;         and score indicates the distance in the hypernym
;;         tree between the two words.
;;         (0 if the two words are the same or synonymous,
;;          1 if one is a direct hypernym of the other, etc.)
;;-----------------------------------------------------------------
(defun name-match-frame-p (word1 word2 depth &optional (match-type? 'weak))
  (let (distance1 distance2)
    (cond
     ((equal word1 word2)
      (cons 'first 0))
     ((check-synsets word1 word2)
      (cons 'first 0))
     ((eql match-type? 'strong)
```

```
     (if (< 0 (setf distance2 (check-hypernym word2 word1 depth)))
   (cons 'second distance2)))
    ((eql match-type? 'weak)
     (if (or (< 0 (setf distance1 (check-hypernym word1 word2 depth)))
             (< 0 (setf distance2 (check-hypernym word2 word1 depth))))
             (if (< 0 distance1)
                 (cons 'first distance1)
                 (cons 'second distance2))))
    (t
     nil))))

;;-----------------------------------------------------------------
;; DESC: Determines if one concept subsumes the other, in this
;;       case one concept is the superclass of another
;; INPUT: concept1 = a concept
;;        concept2 = a concept
;;        match-type? (optional) = the type of the match (i.e. strong
;; OUTPUT: NIL if the two concepts do not subsume each other
;;         A pair (<spec> . <score>) is returned otherwise, where
;;         <spec> = first | second
;;         and score indicates the taxonomic distance between the
;;         concepts.
;;         (0 if the two words are the same class,
;;          1 if one is a direct superclass of the other, etc.)
;; NOTE: Modelled after function subsume-p in semantic-matcher.lisp
;;-----------------------------------------------------------------
(defun subsume-frame-p (concept1 concept2 &optional (match-type? 'weak))
  (let (instance1 instance2 distance1 distance2)
    (if (is-instance-p concept1)
        (setf instance1 concept1 concept1 (immediate-classes (first concept1)))
        (setf concept1 (first concept1)))
    (if (is-instance-p concept2)
        (setf instance2 concept2 concept2 (immediate-classes (first concept2)))
        (setf concept2 (first concept2)))
    ;; Determine if the two triples subsume each other.
    (cond
     ;; Determine if the two things compared are both instances.
     ((and instance1 instance2)
      (if (equal instance1 instance2)
          (cons 'first 0)))
     ;; The matching is done in the context of auto-classification/
     ;; intensional subsumption. Thus, inorder for there to be a
     ;; match t1, the subsumee, has to be more specific than t2
     ;; (i.e. t1 is generalized by t2).
     ((eql match-type? 'strong)
```

```
      (if (setf distance2 (tax-dist concept2 concept1))
          (cons 'second distance2)))
     ;; The matching is done in the context of some like
     ;; Literal Similarity.
     ((eql match-type? 'weak)
      (if (or (setf distance1 (tax-dist concept1 concept2))
              (setf distance2 (tax-dist concept2 concept1)))
          (cons (if distance1 'first 'second)
                (or distance1 distance2))))
     (t
      NIL))))

;;-----------------------------------------------------------------
;; DESC: Determines whether or not the two slots can be matched
;; INPUT: word1 = a symbol for a slot
;;        word2 = a symbol for a slot
;; OUTPUT: NIL if the two slot names do not match
;;         t if they do
;; NOTE: Line-up-relation does this and more, so name-match-slot
;;       is not being used right now
;;-----------------------------------------------------------------
(defun name-match-slot (word1 word2)
  (equal word1 word2))

;;-----------------------------------------------------------------
;; DESC: Formats an entire triple so that the concept names
;;       can be searched for in WordNet
;; INPUT: triple = (<concept> <relation> <concept>)
;;        where <concept> = <word> | (<word> . <id>)
;;              <relation> = <word>
;; OUTPUT: A triple whose form matches that of the input,
;;         with the word portions of the triple formatted to be
;;         usable by WordNet
;; NOTE: Needs more work
;;-----------------------------------------------------------------
(defun reformat-triple (triple)
  (let ((t-head (triple-head triple)) (t-tail (triple-tail triple))
  (t-rel  (triple-relation triple)) w-head w-tail)
    ;; Extract the word from the concept and format
    (setf w-head (reformat-word (if (consp t-head) (car t-head) t-head)))
    (setf w-tail (reformat-word (if (consp t-tail) (car t-tail) t-tail)))
    ;; Recreate the concepts
    (setf t-head (if (consp t-head) (cons w-head (cdr t-head)) w-head))
    (setf t-tail (if (consp t-tail) (cons w-tail (cdr t-tail)) w-tail))
    ;; Rejoin concepts into a triple
```

```
    (list t-head t-rel t-tail)))

;;-----------------------------------------------------------------
;; DESC: Formats a word to be usable by Wordnet
;; INPUT: word = any symbol
;; OUTPUT: A symbol whose name is formatted to be usable by WordNet
;;-----------------------------------------------------------------
(defun reformat-word (word)
  (let ((word-string (symbol-name word)))
    (setf word-string (excl:replace-regexp word-string "-" "_"))
    (setf word-string (string-downcase word-string))
    (intern word-string)))

;;-----------------------------------------------------------------
;; DESC: Checks if two words are in the synsets of each other by
;;       gathering the synset offsets of both and seeing if there
;;       is a common element
;; INPUT: word1 = any string
;;        word2 = any string
;; OUTPUT: T if there is a common element, NIL if there isn't
;;-----------------------------------------------------------------
(defun check-synsets (word1 word2)
  (match-syns (get-synset-offsets word1 4) (get-synset-offsets word2 4)))

;;-----------------------------------------------------------------
;; DESC: Helper to check-synsets
;;       Gathers all offsets for the given word for up to <pos>
;;       parts of speech in the order of "n", "v", "a", "r".  For
;;       example, 3 for <pos> would yield gathering offsets for
;;       "n", "v", and "a" parts of speech for <word>.
;; INPUT: word = any string
;;        pos = an integer between 0 and 4 inclusive
;; OUTPUT: A list of integers representing offset numbers used
;;         within WordNet.
;;-----------------------------------------------------------------
(defun get-synset-offsets (word pos)
  (if (= pos 0)
      '()
      (progn
        (setf syns (wn:synset-offsets word pos))
        (if (consp syns)
            (append syns (get-synset-offsets word (decf pos)))
            (if (null syns)
                (get-synset-offsets word (decf pos))
                (cons syns (get-synset-offsets word (decf pos))))))))
```

```
;;-----------------------------------------------------------------
;; DESC: Helper to check-synsets
;;       Checks to see if any element of list1 matches any element
;;       of list2
;; INPUT: list1 = list of integers
;;        list2 = list of integers
;; OUTPUT: T if match is found, NIL otherwise
;;-----------------------------------------------------------------
(defun match-syns (list1 list2)
  (if (null list2)
      NIL
      (if (match list1 (car list2))
          T
          (match-syns list1 (cdr list2)))))

;;-----------------------------------------------------------------
;; DESC: Checks if word2 is in the set of the hypernyms of word 1
;; INPUT: word1 = any string
;;        word2 = any string
;;        depth = how far in the hypernym tree to search
;;                2 represents search one level above
;; OUTPUT: The level at which a hypernym was matched (i.e. 1 means
;;         a hypernym match occured 1 level away in the tree)
;;-----------------------------------------------------------------
(defun check-hypernym (word1 word2 depth)
  (match-hyper (gather-hypernyms word1 depth) word2 0))

;;-----------------------------------------------------------------
;; DESC: Helper to check-hypernym
;;       Checks to see which of the list of lists word matches in
;;       corresponding to the depth away that the match occured
;; INPUT: list = list of lists of strings
;;        word = string
;;        iter = used for recursion; always start at 0
;; OUTPUT: An integer representing where the match occurred.  0
;;         represents a match failed.  1 represents a match one
;;         level away in the hypernym tree.
;;-----------------------------------------------------------------
(defun match-hyper (list word iter)
  (if (null list)
      0
      (if (match (car list) word)
          iter
          (match-hyper (cdr list) word (incf iter)))))
```

```
;;-----------------------------------------------------------------
;; DESC: Matches to see if <word> is contained in <list>
;; INPUT: list = list of strings
;;        word = string
;; OUTPUT: T if a match is found, NIL otherwise
;;-----------------------------------------------------------------
(defun match (list word)
  (if (null list)
      nil
      (if (equal word (car list))
          t
          (match (cdr list) word))))

;;-----------------------------------------------------------------
;; DESC: Helper to check-hypernym
;;       Gathers all hypernyms for the given word to the provided
;;       depth from the word.
;; INPUT: word = any string
;;        depth = an integer representing how many levels to search
;; OUTPUT: A list of lists of strings.  The first list are
;;         hypernyms 0 levels away.  The second list are hypernyms
;;         1 level away.  Etc.
;;-----------------------------------------------------------------
(defun gather-hypernyms (word depth)
  (do
    ((iter -1 (incf iter))
     (word-list (list word) (gather-hypernyms-help word-list))
     (result nil (cons word-list result)))
    ((= iter depth) (reverse result))))

;;-----------------------------------------------------------------
;; DESC: Helper to check-hypernym
;;       Finds the hypernyms of all the words in the given list,
;;       extracting only the words from the triples returned by
;;       WordNet
;; INPUT: word-list = list of strings
;; OUTPUT: List of strings containing all relevant hypernyms
;;-----------------------------------------------------------------
(defun gather-hypernyms-help (word-list)
  (if (null word-list)
      '()
      (append (mapcar #'(lambda (trip) (car trip))
                      (wn:hypernyms (car word-list)))
              (gather-hypernyms-help (cdr word-list)))))
```

```
;;-------------------------------------------------------------------
;; DESC: Helper funtion which asserts a list of relations.  Meant
;;       to assert "superclasses" relations.  Will not modify
;;       triples.
;; INPUT: triples = list of relations
;; OUTPUT: --
;;-------------------------------------------------------------------
(defun assert-heirarchy (triples)
  (if (null triples)
      nil
      (progn
        (km `(|assert| (:|triple| ,(first (car triples))
                                  ,(second (car triples))
                                  ,(third (car triples)))))
        (assert-heirarchy (cdr triples)))))

;;-------------------------------------------------------------------
;; DESC: Helper function which asserts a single triple.  Meant to
;;       assert triples which have matches in the *match-table*.
;;       If the triple contains an Aggregate, queries KM for
;;       head = (the is-part-of of <Aggregate>) and asserts
;;       (every head has (has-part ((a Aggregate with...)))).  Else
;;       asserts axiom as is.
;; INPUT: triple = (head relation tail) where head and tail are
;;        class names (unless either is an Aggregate)
;; OUTPUT: --
;;-------------------------------------------------------------------
(defun assert-triple-axiom (triple)
  (cond
   ((not (null (search "Aggregate" (symbol-name (first triple)))))
    (let (head body)
      (setf body (first (km `(|the| |instance-of| |of| ,(first triple)))))
      (setf head (first (km `(|the| |is-part-of| |of| ,(first triple)))))
      (setf head (first (km `(|the| |instance-of| |of| ,head))))
      (km `(|every| ,head |has|
             (|has-part| ((|a| ,body |with|
                             (,(second triple) ((|a| ,(third triple))))))))
      ))
   ((not (listp (third triple)))
    (km `(|every| ,(first triple) |has|
           (,(second triple) ((|a| ,(third triple)))))))
   (t
    (km `(|every| ,(first triple) |has|
           (,(second triple) (,(third triple)))))))
```

```
;;-----------------------------------------------------------------
;; DESC: Asserts triples dealing with Aggregates as a group where
;;       the Aggregate didn't have a matching concept in the target
;; INPUT: aggregate = skolem name of Aggregate being asserted
;;        triples = list of all triples relating to aggregate
;; OUTPUT: --
;;-----------------------------------------------------------------
(defun assert-aggregates (aggregate triples)
  (let (head relation body-list)
    (setf head (find-tail aggregate triples))
    (setf relation (second head))
    (setf triples (remove-triples (list head) triples))
    (setf head (first head))
    (setf body-list (make-agg-body-list aggregate triples))
    (km `(|every| ,head |has|
           (,relation (,body-list))))))

;;-----------------------------------------------------------------
;; DESC: Helper function to find the triple where agg appears as
;;       the tail concept
;; INPUT: agg = skolem name of Aggregate
;;        triples = list of triples where each is (head relation tail)
;; OUTPUT: triple where tail = agg
;;-----------------------------------------------------------------
(defun find-tail (agg triples)
  (if (equal agg (third (car triples)))
      (car triples)
      (find-tail agg (cdr triples))))

;;-----------------------------------------------------------------
;; DESC: Helper function to assert-aggregates which builds the
;;       property list of an aggregate.
;; INPUT: agg = skolem name of relevant aggregate
;;        triples = list of triples dealing with agg
;; OUTPUT: (a Aggregate with (<list of axiomatizations of triples>))
;;-----------------------------------------------------------------
(defun make-agg-body-list (agg triples)
  (let (result)
    (dolist (trip triples)
      (cond
       ((equal agg (first trip))
        (if (equal (second trip) '|element|)
            (progn
              (push `(|element| ,(list `(|a| ,(third trip)))) result)
```

```
              (push `(|element-type| ,(list (third trip))) result))
              (push `(,(second trip) ,(list `(|a| ,(third trip)))) result)))
       (t
         (if (equal (second trip) '|element-of|)
             (progn
              (push `(|element| ,(list `(|a| ,(first trip)))) result)
              (push `(|element-type| ,(list (first trip))) result))
              (push `(,(invert-slot (second trip)) ,(list `(|a| ,(first trip))))
                    result)))))
    (cons '|a| (cons '|Aggregate| (cons '|with| result)))))

;;-----------------------------------------------------------------
;; DESC: Helper function which removes a list of triples from
;;       another list
;; INPUT: remove = list of items to be removed
;;        list = list from which they are to be removed from
;; OUTPUT: list with any element in remove removed
;;-----------------------------------------------------------------
(defun remove-triples (remove list)
  (let (result found-p)
    (dolist (item list)
      (setf found-p NIL)
      (dolist (rem remove)
        (if (or (equal rem item)
                (equal (list (third rem)
                             (invert-slot (second rem))
                             (first rem))
            item))
        (setf found-p t)))
      (if (null found-p)
          (push item result)))
    result))

;;-----------------------------------------------------------------
;; DESC: Helper function to gather the hierarchy of target
;; INPUT: list = list of concept names to gather for
;; OUTPUT: list of triples defining the superclasses of each
;;         concept
;;-----------------------------------------------------------------
(defun get-superclasses (list)
  (let (result)
    (dolist (concept list)
      (push (list concept '|superclasses|
                  (first (km `(|the| |superclasses| |of| ,concept))))
            result))
```

```
   result))

;;-------------------------------------------------------------------
;; DESC: Merges two lists of class hierarchies, allowing for synonym
;;       matches to occur as defined by WordNet.
;; INPUT: target = list of superclasses triples
;;        source = list of superclasses triples
;; OUTPUT: A cons'ed list of the hierarchy that needs to be asserted
;;         and a list of class-to-class replacements that were
;;         determined (so source will be recognized by the semantic
;;         matcher)
;;-------------------------------------------------------------------
(defun merge-heirarchy (target source)
  (let (heirarchy replacements matched-p score)
    (dolist (striple source)
      (dolist (ttriple target)
        (setf score (cdr (name-matcher (format-instance-triple striple)
                                       (format-instance-triple ttriple) 0)))
        (if (and (numberp score) (= 1 score))
            (cond
              ((and (equal (first ttriple) (first striple))
                    (not (equal (third ttriple) (third striple))))
                    (progn
                      (push (cons (third striple) (third ttriple)) replacements)
                      (setf matched-p t)))
              ((and (not (equal (first ttriple) (first striple)))
                    (equal (third ttriple) (third striple)))
                    (progn
                      (push (cons (first striple) (first ttriple)) replacements)
                    (setf matched-p t)))
              (t
               (setf matched-p t)))))
      (if (null matched-p)
          (push striple heirarchy)
    (setf matched-p nil)))
    (cons heirarchy replacements)))

;;-------------------------------------------------------------------
;; DESC: Helper function to merge-hierarchy to allow triples to
;;       be formatted as the name-matcher expects them
;; INPUT: triple = (head relation tail)
;; OUTPUT: ((head) relation (tail))
;;-------------------------------------------------------------------
(defun format-instance-triple (triple)
  (list (list (first triple)) (second triple) (list (third triple))))
```

```
;;-----------------------------------------------------------------
;; DESC: Helper function to allow source to be recognized by
;;       the semantic matcher by performing all replacements on
;;       triples
;; INPUT: triples = list of triples
;;        replacements = cons cell of class-to-class mappings
;; OUTPUT: triples but with any class defined in replacements
;;         substituted with its pair
;;-----------------------------------------------------------------
(defun do-replacements (triples replacements)
  (if (null replacements)
      triples
      (let (result head relation tail)
  (dolist (rep replacements)
    (dolist (triple triples)
      (setf head (symbol-name (first triple)))
      (setf relation (second triple))
      (if (not (listp (third triple)))
          (setf tail (symbol-name (third triple)))
          (setf tail (third triple)))
      (if (equal head (car rep))
          (setf head (symbol-name (cdr rep)))
          (setf head (intern (excl:replace-regexp
                               head
                               (concatenate 'string
                                            "_"
                                            (symbol-name (car rep)))
                               (symbol-name (cdr rep))))))
      (if (not (listp tail))
          (if (equal tail (car rep))
              (setf tail (symbol-name (cdr rep)))
              (setf tail (intern (excl:replace-regexp
                                  tail
                                 (concatenate 'string
                                              "_"
                                              (symbol-name (car rep)) )
                                 (symbol-name (cdr rep)))))))
      (push (cons head (cons relation (list tail)))
      result))
    (km `(|assert| (:|triple| ,(cdr rep) |synonym| ,(car rep)))))
  result)))

;;-----------------------------------------------------------------
;; DESC: Helper function to read in file as a list
```

```
;; INPUT: file = string representing file name and/or location
;; OUTPUT: list of contents of file (removes standard lisp comments)
;;------------------------------------------------------------------
(defun file-to-list (file)
  (with-open-file (infile file)
    (do ((result nil (cons next result))
         (next (my-case-sensitive-read infile nil 'eof)
         (my-case-sensitive-read infile nil 'eof)))
  ((equal next 'eof) (reverse result)))))
;;------------------------------------------------------------------
;; DESC: Extracts every concept defined in a KB description file
;; INPUT: list = list form of standard *.km file
;;        res = null list to hold result
;; OUTPUT: list of concepts defined with possible duplicates
;;------------------------------------------------------------------
(defun get-concepts (list res)
  (if (null list)
      res
      (if (consp (car list))
          (if (equal (caar list) '|every|)
              (get-concepts (cdr list) (cons (cadar list) res))
              (get-concepts (cdr list) (cons (caar list) res)))
          (get-concepts (cdr list) res))))
;;------------------------------------------------------------------
;; DESC: Helper function to replace all skolems with their
;;       corresponding class concept name from *match-table* if
;;       one exists.
;; INPUT: triples = list of triples of form (head relation tail)
;; OUTPUT: If head has entry in *match-table* (class . score) then
;;         (class relation tail). (Same for tail)
;;         If not, then (head relation tail).
;;------------------------------------------------------------------
(defun replace-skolem (triples)
  (let (result head tail)
    (dolist (trip triples)
      (setf head (first trip))
      (setf tail (third trip))
      (if (not (equal head (lookup-triple head)))
          (setf head (caar (lookup-triple head))))
      (if (not (equal tail (lookup-triple tail)))
          (setf tail (caar (lookup-triple tail))))
      (push (list head (second trip) tail) result))
    result))

;;------------------------------------------------------------------
```

```
;; DESC: Given a list of triples that haven't been axiomatized
;;       searches for aggregates and asserts all triples relation
;;       to a given aggregate at once
;; INPUT: triples = list of unmatched triples
;;        a-list = association list mapping from instance to class
;;                 name
;; OUTPUT: All triples which are still unprocessed
;;-----------------------------------------------------------------
(defun process-unmatched-aggregates (triples a-list)
  (let (aggregates agg-triples result instances)
    (setf instances (get-triple-relation triples '|instance-of| '()))
    (setf triples (remove-triple-relation triples '|instance-of| '()))
    (setf aggregates (remove-duplicates (get-all-aggregates triples)))
    (dolist (agg aggregates)
      (setf agg-triples (find-all-aggregates triples '() agg))
      (assert-aggregates agg (substitute-class (replace-skolem agg-triples)
                                               a-list))
      (setf result (append agg-triples result)))
    (append (remove-triples result triples) instances)))

;;-----------------------------------------------------------------
;; DESC: Given a list of triples, returns all that involve any
;;       aggregate.
;; INPUT: triples = list of triples of form (head relation tail)
;; OUTPUT: List of triples where either head or tail was an aggregate
;;-----------------------------------------------------------------
(defun get-all-aggregates (triples)
  (let (result)
    (setf triples (flatten triples))
    (dolist (trip triples)
      (if (and (not (equal '|Aggregate| trip))
               (search "Aggregate" (symbol-name trip)))
          (push trip result)))
    result))

;;-----------------------------------------------------------------
;; DESC: Given a list of triples and a specific aggregate to search
;;       for, find all triples that involve it.
;; INPUT: triples = list of triples
;;        res = null list to hold result
;;        agg = skolem name of aggregate to search for
;; OUTPUT: list of triples which involve agg
;;-----------------------------------------------------------------
(defun find-all-aggregates (triples res agg)
  (if (null triples)
```

```
      res
      (if (or (equal agg (first (car triples)))
              (equal agg (third (car triples))))
          (find-all-aggregates (cdr triples) (cons (car triples) res) agg)
          (find-all-aggregates (cdr triples) res agg))))

;;------------------------------------------------------------------
;; DESC: Reads a given stream with cas sensitivity turned on
;; INPUT: stream (optional) = the stream from which to read
;;        eof-err-p (optional) = end of file error
;; OUTPUT: line of read text
;;------------------------------------------------------------------
(defun my-case-sensitive-read (&optional stream (eof-err-p t) eof-val rec-p)
  (let ((old-readtable-case (readtable-case *readtable*)))
    (handler-case
     (prog2
       (setf (readtable-case *readtable*) :preserve)
       (read stream eof-err-p eof-val rec-p)
       (setf (readtable-case *readtable*) old-readtable-case))
       (error (error)                    ; make sure readtable-case gets reset!
       (setf (readtable-case *readtable*) old-readtable-case)
       'km-syntax-error))))


;;------------------------------------------------------------------
;; DESC: Determines whether or not this concept is one which maybe
;;       should not have assertions made about in in the following
;;       "evaluate" functions
;; INPUT: concept = a value from lookup-triple
;; OUTPUT: T if
;;         NIL otherwise
;;------------------------------------------------------------------
(defun evaluate-postpone-p (concept)
  (if (listp concept)
      nil
      (not (null (search "Aggregate" (symbol-name concept))))))

(defun evaluate-flatten (list)
  (evaluate-flatten-rest list '()))

(defun evaluate-flatten-rest (list result)
  (if (null list)
      result
      (evaluate-flatten-rest (cdr list) (append (car list) result))))
```

```
;;-----------------------------------------------------------------
;; DESC: Given sets of matched triples, asserts matches into the KB
;;       when the source of the match is more descriptive than the
;;       target
;; INPUT: target-list = a list of sets of triples from the target
;;        source-list  = a list of sets of triples from the source
;;            The two lists should be parallel, ie the same length and
;;                the nth set in base-list matches the nth set in new-list
;;            Each list is of the form:
;;                ((<triple> ...) ...)
;; OUTPUT: A list of triples which still need to be asserted into the new KB
;;-----------------------------------------------------------------
(defun evaluate-matches (target-list source-list)
  (let ((mapping-a-list (make-a-list (get-triple-relation
                                       (evaluate-flatten source-list)
                                       '|instance-of|
                                       '())
                                     '()))))
    (do
      ((s-list  source-list (cdr s-list))
       (t-list  target-list (cdr t-list))
       (results NIL         (append (evaluate-match (car s-list)
                                                    (car t-list)
                                                    mapping-a-list)
                                    results)))
      ((or (null s-list) (null t-list))
   results))))

;;-----------------------------------------------------------------
;; DESC: Given a set of matched triples, asserts the match into the KB
;;       when the source of the match is more descriptive than the
;;       target
;; INPUT: source-set = a set of triples from the source
;;        target-set  = a set of triples from the target
;;        mapping-a-list = an association list mapping skolems to classes
;;                         in the source
;;            A set is of the form:
;;                (<triple> ...)
;; OUTPUT: A list of triples which still need to be asserted into the new KB
;;-----------------------------------------------------------------
(defun evaluate-match  (source-set target-set mapping-a-list)
  (if (> (length source-set) (length target-set))
      (evaluate-matched-triples (remove-triple-relation source-set
                                                         '|instance-of|
                                                         '())
```

```
                                mapping-a-list
                                '())))

;;-----------------------------------------------------------------
;; DESC: Given a set of matched triples, asserts those triples for
;;       which it is able to do so right away, and returns the rest
;; INPUT: source-set = the rest of a set of triples from the source
;;        mapping-a-list = an association list mapping skolems to
;;                         classes in the source
;;        results = A list of triples so far which still need to be
;;                  asserted into the KB
;; OUTPUT: A list of all triples from the match
;;         which still need to be asserted into the new KB
;;-----------------------------------------------------------------
(defun evaluate-matched-triples (source-set mapping-a-list results)
  (if (null source-set)
      results
     (if (evaluate-assert-triple mapping-a-list (car source-set))
         (evaluate-matched-triples (cdr source-set) mapping-a-list results)
         (evaluate-matched-triples (cdr source-set)
                                   mapping-a-list
                                   (cons (car source-set) results)))))

;;-----------------------------------------------------------------
;; DESC: Given a list of triples from the source that do not match
;;       anything in the target, asserts those triples which are
;;       related to concepts in the target
;; INPUT: mapping-a-list = Association list mapping skolems
;;                         in the source to their classes in the source
;;        unmatched-triples = a list of triples from the target KB
;; OUTPUT: A list of triples which were not asserted into the new KB
;;-----------------------------------------------------------------
(defun evaluate-all-unmatched (mapping-a-list unmatched-triples)
  (let (still-unmatched)
    (dolist (triple unmatched-triples)
      (if (not (evaluate-assert-triple mapping-a-list triple))
          (setf still-unmatched (cons triple still-unmatched))))
    still-unmatched))

;;--------------------------------------------------------------------
;; DESC: Asserts the triple if it is safe to do so
;; INPUT: mapping-a-list = an association list mapping skolems to
;;                         classes in the source
;;        triple = a triple from the source
;; OUTPUT: T if the triple did get asserted;
```

```
;;          NIL otherwise
;;---------------------------------------------------------------------
(defun evaluate-assert-triple (mapping-a-list triple)
  (let* ((head (first triple))
   (rel (second triple))
   (tail (third triple))
   (head-tgt (lookup-triple head))
   (tail-tgt (lookup-triple tail))
   (head-src (assoc head mapping-a-list))
   (tail-src (assoc tail mapping-a-list))
   mapped-triple postpone)
    (if (and (null tail-src) (listp tail))
        (setf tail-src (cons '() tail)))
    (if (and (null head-src) (listp head))
        (setf head-src (cons '() head)))
    (setf postpone T)
    (cond
     ;; Both head and tail match in the target
     ((and (not (equal head-tgt head))
           (not (equal tail-tgt tail)))
      (setf mapped-triple (list (caar head-tgt) rel (caar tail-tgt))))
     ;; The head matches in the target
     ((not (equal head-tgt head))
      (progn
        (setf mapped-triple (list (caar head-tgt) rel (cdr tail-src)))
        (setf postpone (evaluate-postpone-p (third mapped-triple)))))
     ;; The tail matches in the target
     ((not (equal tail-tgt tail))
      (progn
        (setf mapped-triple (list (cdr head-src) rel (caar tail-tgt)))
        (setf postpone (evaluate-postpone-p (first mapped-triple))))))
    (pprint (list "Decided to assert"
                  triple
                  "as"
                  mapped-triple
                  "?"
                  (not postpone)
                  ";"))
     ;; Don't bother mapping if there is no match in the target,
     ;; it won't be asserted
    (if (not postpone)
        (progn
          (assert-triple-axiom mapped-triple)
          T))))
```

```
;;------------------------------------------------------------------
;; DESC: Given a list of matches (output from semantic matcher) in
;;       Peter's form, fills *match-table* with entries of the form
;;       <source_skolem> --hashes to--> (<target_class> . <score>)
;; INPUT: List of matches in Peter's form (reference cpl-conversion.lisp
;;        for explanation of Peter's form).
;; OUTPUT: A list of 2 lists.  The first is the set of triples matched
;;         in the source returned to Pete's form.  The second is the
;;         set of triples matched in the target returned to Pete's form.
;;         The lists are the same length and the entries correspond.
;;------------------------------------------------------------------
(defun build-table (matches)
  (let (matched-source matched-target triple-form
        instance-s instance-t source target score)
    (setf triple-form (convert-mappings-to-cpl-format matches
                                                       *node-to-km-mappings*
                                                       *km-to-node-mappings*))
    (setf matches (reverse matches))
    (dolist (match matches)
      (cond
       ((path-p (get-mapping-source match))
        (setf instance-s (get-triple-relation
                          (convert-to-petes-form (get-mapping-source match))
                          '|instance-of| '())))
       (t
        (setf instance-s (get-triple-relation
                          (convert-to-petes-form (list
                                                  (get-mapping-source match)))
                          '|instance-of| '()))))
      (cond
       ((path-p (get-mapping-target match))
        (setf instance-t (get-triple-relation
                          (convert-to-petes-form (get-mapping-target match))
                          '|instance-of| '())))
       (t
        (setf instance-t (get-triple-relation
                          (convert-to-petes-form (list
                                                  (get-mapping-target match)))
                          '|instance-of| '()))))
      (setf source (get-mapping-source (car triple-form)))
      (if (not (path-p source))
          (setf source (list source)))
      (setf target (get-mapping-target (car triple-form)))
      (if (not (path-p target))
          (setf target (list target)))
```

```
      (setf score (get-mapping-score (car triple-form)))
      (setf triple-form (cdr triple-form))
      (build-match source
       (substitute-class target
                         (make-a-list instance-t '()))
                         score)
      (push (append instance-s source) matched-source)
      (push (append instance-t target) matched-target))
    (cons matched-source (list matched-target))))

;;----------------------------------------------------------------------
;; DESC: Turns one set of source-to-target matches (in Pete's form)
;;       into a hash table entry.  If it already exists, adds entry
;;       into list sorted by score.  Otherwise, creates the entry.
;; INPUT: source = matched triples from source without triples with
;;                 instance-of relation
;;        target = matched triples where frame and value are class
;;                 names, not skolems (no instance-of relations)
;; OUTPUT: --
;;         As side effect, *match-table* has an added entry.
;;----------------------------------------------------------------------
(defun build-match (source target score)
  (let (head tail)
    (setf head (gethash (get-first-frame source) *match-table*))
    (setf tail (gethash (get-last-value source) *match-table*))
    (if (null head)
        (setf (gethash (get-first-frame source) *match-table*)
              (list (cons (get-first-frame target) score)))
        (setf (gethash (get-first-frame source) *match-table*)
              (insert-by-score (get-first-frame target) score head '())))
    (if (null tail)
        (setf (gethash (get-last-value source) *match-table*)
              (list (cons (get-last-value target) score)))
        (setf (gethash (get-last-value source) *match-table*)
              (insert-by-score (get-last-value target) score tail '())))))

;;----------------------------------------------------------------------
;; DESC: Out of a list of triples, returns the frame part of the very
;;       first one in the list.
;; INPUT: triples = list of triples, (i.e. ((h1 s1 v1) (h2 s2 v2)...))
;; OUTPUT: h1
;;----------------------------------------------------------------------
(defun get-first-frame (triples)
  (if (listp (car triples))
      (caar triples)
```

```
      (car triples)))

;;------------------------------------------------------------------
;; DESC: Out of a list of triples, returns the value part of the very
;        last one in the list.
;; INPUT: triples = list of triples, (i.e. ((h1 s1 v1)...(hn sn vn)))
;; OUTPUT: vn
;;------------------------------------------------------------------
(defun get-last-value (triples)
  (if (and (listp (car triples)) (null (cdr triples)))
      (car (cddar triples))
      (get-last-value (cdr triples))))

;;------------------------------------------------------------------
;; DESC: Adds an entry into a list of entries so that they are sorted
;;       on the basis of score.  Used to help create hash table entry
;;       when one already exists.
;; INPUT: datum = a item a key is to be hashed to
;;        score = the match score related to that datum
;;        entries = list of preexisting things the key hashed to
;;        res = empty list (will have result)
;; OUTPUT: datum in a hash table that is a list of (item . score)
;;         sorted in descending order of score
;;------------------------------------------------------------------
(defun insert-by-score (datum score entries res)
  (cond
  ((null entries)
   (reverse (cons (cons datum score) res)))
  ((equal datum (caar entries))
   (append (reverse res) entries))
  ((> score (cdar entries))
   (append (reverse (cons (cons datum score) res))
           (remove-datum datum entries nil)))
  (t
   (insert-by-score datum score
                    (cdr entries)
                    (cons (car entries) res)))))

(defun remove-datum (datum list res)
  (if (null list)
      (reverse res)
      (if (equal datum (caar list))
          (append (reverse res) (cdr list))
          (remove-datum datum (cdr list) (cons (car list) res)))))
```

```
;;-------------------------------------------------------------------
;; DESC: Takes a list of triples and executes the list of substitutions
;;       on the frame and value of every triple.
;; INPUT: target = triples from the target knowledge base in Pete's form
;;        substitutions = association list of instances and their classes
;; OUTPUT: List with frame and values replaced according to the
;;         substitutions
;;-------------------------------------------------------------------
(defun substitute-class (target substitutions)
  (let (result head-datum tail-datum)
    (dolist (triple target)
      (setf head-datum (cdr (assoc (car triple) substitutions :test #'equal)))
      (if (null head-datum)
          (setf head-datum (car triple)))
      (setf tail-datum (cdr (assoc (caddr triple) substitutions :test #'equal)))
      (if (null tail-datum)
          (setf tail-datum (caddr triple)))
      (push (list head-datum (cadr triple) tail-datum) result))
    (reverse result)))

;;-------------------------------------------------------------------
;; DESC: Removes any triple of the form (x <relation> y)
;; INPUT: list = list of triples to process
;;        relation = a slot relation
;;        res = empty list (to return result)
;; OUTPUT: Original list without triples with <relation>
;;-------------------------------------------------------------------
(defun remove-triple-relation (list relation res)
  (if (null list)
      (reverse res)
      (if (equal (cadar list) relation)
          (remove-triple-relation (cdr list) relation res)
          (remove-triple-relation (cdr list) relation (cons (car list) res)))))

;;-------------------------------------------------------------------
;; DESC: Returns a list of only triples of the form (x <relation> y)
;; INPUT: list = list of triples to process
;;        relation = a slot relation
;;        res = empty list (to return result)
;; OUTPUT: All triples with <relation> from the original list.
;;-------------------------------------------------------------------
(defun get-triple-relation (list relation res)
  (if (null list)
      (reverse res)
      (if (equal (cadar list) relation)
```

```
          (get-triple-relation (cdr list) relation (cons (car list) res))
          (get-triple-relation (cdr list) relation res))))

;;------------------------------------------------------------------
;; DESC: Turns a list of triples into an association list.  Meant
;;       to be used on a list of only instance-of triples to achieve
;;       association list matching each skolem name to the class to
;;       which it belongs.  Aggregates are not included in the list as
;;       they are specially handled by the system later.
;; INPUT: list = list of triples of form ((h1 s1 v1) (h2 s2 v2)...)
;;        res = empty list (to return result)
;; OUTPUT: Association list of ((h1 . v1) (h2 . v2)...)
;;------------------------------------------------------------------
(defun make-a-list (list res)
  (if (null list)
      res
      (if (equal (car (cddar list)) '|Aggregate|)
          (make-a-list (cdr list) (acons (caar list) (caar list) res))
          (make-a-list (cdr list) (acons (caar list) (car (cddar list)) res)))))

;;------------------------------------------------------------------
;; DESC: Convenience function that looks up a key in the match table.
;; INPUT: key = skolem name from source file
;; OUTPUT: List of (<class> . <score>) entries where <class> is a
;;         class in the target KB.  Returns key is no entry in hash
;;         table for key.
;;------------------------------------------------------------------
(defun lookup-triple (key)
  (if (null (gethash key *match-table*))
      key
      (gethash key *match-table*)))

(defun convert-mappings-to-cpl-format (mappings
                                       node-to-km-mappings
                                       km-to-node-mappings)
  (let (mapping-source mapping-target mapping-score
        cpl-mapping-source cpl-mapping-target result)
    (dolist (mapping mappings)
      (setf mapping-source (get-mapping-source mapping))
      (setf mapping-target (get-mapping-target mapping))
      (setf mapping-score  (get-mapping-score  mapping))
      (cond
       ((path-p mapping-source)
        (setf cpl-mapping-source
              (mapcar #'(lambda (x)
```

```
                    (convert-triple-to-petes-form x node-to-km-mappings
                                                    km-to-node-mappings))
                  mapping-source)))
       (t
        (setf cpl-mapping-source
              (convert-triple-to-petes-form mapping-source
                                            node-to-km-mappings
                                            km-to-node-mappings))))
      (cond
       ((path-p mapping-target)
        (setf cpl-mapping-target
              (mapcar #'(lambda (x)
                    (convert-triple-to-petes-form x node-to-km-mappings
                                                  km-to-node-mappings))
                  mapping-target)))
       (t
        (setf cpl-mapping-target
              (convert-triple-to-petes-form mapping-target
                                            node-to-km-mappings
                                            km-to-node-mappings))))
      (setf result (cons (cons (list cpl-mapping-source
                                     cpl-mapping-target)
                               mapping-score)
                         result)))
    result))

;;;;------------------------------------------------------------------
;;;;------------------------------------------------------------------
;;;; FILE: cpl-conversion.lisp
;;;;
;;;; Code to convert between the format used by the matcher and CPL.
;;;;
;;;; FUNCTION INVENTORY:
;;;; - (defun convert-to-peters-form (petes-triples
;;;;                                  &optional node-to-km-mappings)
;;;; - (defun convert-to-petes-form  (peters-triples
;;;;                                  &optional node-to-km-mappings)
;;;; - (defun convert-triple-to-peters-form (triple node-to-km-mappings
;;;;                                         km-to-node-mappings)
;;;; - (defun convert-triple-to-petes-form  (triple node-to-km-mappings
;;;;                                         km-to-node-mappings)
;;;; - (defun get-instance-of-triples    (triples node-to-km-mappings
;;;;                                      km-to-node-mappings)
;;;; - (defun create-instance-of-triples (triples node-to-km-mappings)
;;;; - (defun create-and-hash-new-node-instance (km-instance
```

```
;;;;                                                node-to-km-mappings
;;;;                                                km-to-node-mappings)
;;;;------------------------------------------------------------------------
;;;;------------------------------------------------------------------------

;;;------------------------------------------------------------------------
;;;------------------------------------------------------------------------
;;; Functions to convert between the different formats used by CPL and
;;; the matcher.
;;;------------------------------------------------------------------------
;;;------------------------------------------------------------------------

;;------------------------------------------------------------------------
;; DESC: Given a list of triples in Pete's CPL format, convert them to
;;   a list of triples in the format used by the matcher.
;; INPUT: petes-triples = a list of triples in Pete's CPL format.
;;    node-to-km-mappings (optional) = hash table with the mappings
;;    from a graph node to km concept
;;    km-to-node-mappings (optional) = hash table with the mappings
;;    from a km concept to a graph node.
;; OUTPUT: A list of triples in Peter's format.
;;------------------------------------------------------------------------
(defun convert-to-peters-form (petes-triples
                               &optional (node-to-km-mappings
                                          *node-to-km-mappings*)
                                         (km-to-node-mappings
                                          *km-to-node-mappings*))
  (let (triples)
    ;; Get all instance-of triples in petes-triples and hash them.
    (get-instance-of-triples petes-triples node-to-km-mappings
                             km-to-node-mappings)
    ;; For each triple in petes-triples convert them to the format used
    ;; by the matcher.
    (dolist (triple petes-triples)
      (cond
       ((eql (triple-relation triple) '|instance-of|))
       (t
        (setf triples (cons (convert-triple-to-peters-form triple
                                                           node-to-km-mappings
                                                           km-to-node-mappings)
                            triples)))))
    triples))

;;------------------------------------------------------------------------
;; DESC: Given a list of triples in the format used by the Matcher,
```

```
;;   convert them to the format used in CPL.
;; INPUT: peters-triples = a list of triples in the Matcher format.
;;    node-to-km-mappings (optional) = hash table with the mappings
;;    from a graph node to km concept
;;    km-to-node-mappings (optional) = hash table with the mappings
;;    from a km concept to a graph node.
;; OUTPUT: A list of triples in Pete's format.
;;----------------------------------------------------------------------
(defun convert-to-petes-form (peters-triples
                              &optional (node-to-km-mappings
                                         *node-to-km-mappings*)
                                        (km-to-node-mappings
                                         *km-to-node-mappings*))
  (let (triples pete-triple)
    ;; Convert each triple to Pete's form.
    (dolist (triple peters-triples)
      (setf pete-triple (convert-triple-to-petes-form triple
                                                      node-to-km-mappings
                                                      km-to-node-mappings))
      (if pete-triple (setf triples (cons pete-triple triples))))
    ;; Make sure to create the instance-of triples and add them to the
    ;; final output.
    (append (create-instance-of-triples peters-triples node-to-km-mappings)
            triples)))

;;;---------------------------------------------------------------------
;;;---------------------------------------------------------------------
;;; Fucntions to convert an single triple between the CPL and
;;; Matcher format.
;;;---------------------------------------------------------------------
;;;---------------------------------------------------------------------

;;----------------------------------------------------------------------
;; DESC: Given a triple in Pete's form convert it to Peter's form.
;; INPUT: triple = a triple in Pete's (i.e. CPL) form.
;;    node-to-km-mappings = hash with the mappings from a graph
;;    node instance to a KM instance.
;;    km-to-node-mappings = hash with the mappings from a KM
;;    instance to a graph node instance.
;; OUTPUT: The corresponding triple in Peter's format.
;;----------------------------------------------------------------------
(defun convert-triple-to-peters-form (triple
                                      node-to-km-mappings
                                      km-to-node-mappings)
  (let ((head (or (gethash (triple-head triple) km-to-node-mappings)
```

```
                 (create-and-hash-new-node-instance (triple-head triple)
                                                    node-to-km-mappings
                                                    km-to-node-mappings)))
        (rel  (triple-relation triple))
        (tail (if (atom (triple-tail triple))
                 (or (gethash (triple-tail triple) km-to-node-mappings)
                     (create-and-hash-new-node-instance (triple-tail triple)
                                                        node-to-km-mappings
                                                        km-to-node-mappings))
                 (triple-tail triple))))
    (list head rel tail)))

;;----------------------------------------------------------------------
;; DESC: Given a triple in peters form convert it to Pete's form.
;; INPUT: triple = a triple in Peter's form.
;;    node-to-km-mappings = hash with the mappings from a graph
;;    node instance to a KM instance.
;;    km-to-node-mappings = hash with the mappings from a KM
;;    instance to a graph node instance.
;; OUTPUT: The corresponding triple in Pete's format.
;;----------------------------------------------------------------------
(defun convert-triple-to-petes-form (triple
                                     node-to-km-mappings
                                     km-to-node-mappings)
  (let ((head (triple-head triple))
        (relation (triple-relation triple))
        (tail (triple-tail triple))
        km-head km-tail)
    ;; See if the head is bounded.
    (cond
      ((is-instance-p head)
       (setf km-head (first head)))
      (t
       (setf km-head (gethash head node-to-km-mappings))
      ))
    ;; Do the same thing for the tail.
    (cond
      ((is-instance-p tail)
       (setf km-tail (first tail)))
      ((is-value-p tail)
       (setf km-tail tail))
      (t
       (setf km-tail (gethash tail node-to-km-mappings))
      ))
    (if (and km-head km-tail) (list km-head relation km-tail) nil)))
```

```
;;;-----------------------------------------------------------------------
;;;-----------------------------------------------------------------------
;;; Functions for extracting, storing, and creating new instances.
;;;-----------------------------------------------------------------------
;;;-----------------------------------------------------------------------

;;------------------------------------------------------------------------
;; DESC: Given a list of triples in CPL (i.e. Pete's) format:
;;   1) extract all the instance-of triples, 2) create a
;;   graph node corresponding to the KM instance, and 3)
;;    hash the mapping between the graph node instance and
;;   the KM instance.
;; INPUT: triples = a list of triples in CPL format.
;;    node-to-km-mappings = hash to store the mappings from a
;;    graph node instance to a KM instance.
;;    km-to-node-mappings = hash to store the mappings from a
;;    KM instance to a graph node instance.
;; OUTPUT: This function does not return any output, but as a side
;;     effect behavior, it will 1) create a corresponding graph
;;     node instance for each KM instance encountered and 2)
;;     store the mappings between the graph and KM instances.
;;------------------------------------------------------------------------
(defun get-instance-of-triples (triples
                                node-to-km-mappings
                                km-to-node-mappings)
  (let (unique-id new-node)
    (dolist (triple triples)
      (if (eql (triple-relation triple) '|instance-of|)
          (progn
            (setf unique-id (string (gensym "Node")))
            (setf new-node (cons (triple-tail triple) unique-id))
            (setf (gethash new-node node-to-km-mappings) (triple-head triple))
            (setf (gethash (triple-head triple) km-to-node-mappings)
                  new-node)))
)))

;;------------------------------------------------------------------------
;; DESC: Given graph (i.e. a list of triples in Peter's format), create
;;   an instance-of triple for each unique node in the graph. We
;;   need this function to make the output compliant with the
;;   format expected by CPL.
;; INPUT: triples = a list of triples in Peter's format.
;;    node-to-km-mappings = hash with the mappings from a graph
;;    node instance to a KM instance.
```

```
;; OUTPUT: A list of instance-of triples -- one for each unique node
;;      in the input graph.
;;------------------------------------------------------------------------
(defun create-instance-of-triples (triples node-to-km-mappings)
  (let (nodes instance instance-triples)
    ;; Get all the nodes in the graph.
    (setf nodes (remove-duplicates
                  (apply #'append (mapcar #'(lambda (x)
                                              (list (triple-head x)
                                                    (triple-tail x)))
                                          triples))
                  :test #'equal))
    ;; For each node, look up its correspondings KM instance.
    (dolist (node nodes)
      (setf instance (gethash node node-to-km-mappings))
      (if instance
          (setf instance-triples (cons (list instance
                                             '|instance-of|
                                             (first node))
                                       instance-triples))))
    instance-triples))

;;------------------------------------------------------------------------
;; DESC: This function is called by convert-triple-to-peters-form.
;;    Its purpose is to create a new graph node corresponding to
;;    the given KM instance. This function will also hash the
;;    the mappings between the newly create node instance and
;;    the existing KM instance.
;; INPUT: km-instance = a KM instance -- e.g. |_Dog12|
;;     node-to-km-mappings = hash to store the mappings from a
;;     graph node instance to a KM instance.
;;     km-to-node-mappings = hash to store the mappings from a
;;     KM instance to a graph node instance.
;; OUTPUT: The newly created graph node instance that corresponds to
;;      the given KM instance.
;;------------------------------------------------------------------------
(defun create-and-hash-new-node-instance (km-instance
                                          node-to-km-mappings
                                          km-to-node-mappings)
  (let (new-node)
    (cond
      ;; There is no need to hash a named instance.
      ((named-instancep km-instance)
       (list km-instance))
      (t
```

```
(setf new-node (cons (singleton-class (immediate-classes km-instance))
                     (string (gensym "Node"))))
(setf (gethash new-node node-to-km-mappings) km-instance)
(setf (gethash km-instance km-to-node-mappings) new-node)
new-node))))
```