KNOW v2

Ben Goertzel
February 26, 2003

Notes

- Section 2 is a modification of a section of an earlier document written by Pei Wang, Cate Hartley and Charlie Derr
- The basic concept of KNOW was created by Pei Wang, although KNOW v2 contains a lot of non-Pei ideas

1. Introduction

This brief document gives an overview of the KNOW v2 (version 2) knowledge representation language.

This language is a descendant of the KNOW v1 language developed for Webmind in 1999-2000. A small corpus of knowledge was encoded using KNOW v1 in 2000. As it turns out, the KNOW v2 syntax is very different from KNOW v1 syntax, but, the differences pertain to advanced features, and the vast majority of the knowledge entered in KNOW v1 is also valid KNOW v2. From here on we will refer to KNOW v2 simply as "KNOW", using the version numbers only when that distinction really needs to be made.

KNOW is intended for two use cases:

1. Formal-language conversation between humans and Novamente, in the context of the ShapeWorld UI and other future interfaces
2. Humans entering knowledge in files, a la "expert system encoding"

Generally speaking, KNOW may be considered analogous to CycL, the knowledge representation language used in the Cyc AI project. However, KNOW is customized for Novamente, which means that in detail it's quite different from CycL. Note also that CycL is designed only for the first use case mentioned above, not the second.

Although KNOW is designed for correspondence with Novamente's internal knowledge representation, it does not correspond exactly to Novamente's internal knowledge representation. One major difference has to do with the use of variables. Novamente

does not use variables internally, but KNOW does use variables, as they are much more intuitive to most humans that Novamente's internal combinator representation.

2. The Grammar

The grammar is written with the following notations:

- "A := B C D" means that "A" consists of a sequence "B C D".
- "A := B C [D]" means that "A" consists of a sequence "B C" or "B C D", that is, D is optional.
- "A := B | C" means that "A" consists of "B" or "C".
- "A := {B}$^{+}$" means that "A" consists of a (non-empty) sequence of "B"s (with an arbitrary length).
- "A = B /<n/>" means that "A" consists of B followed by "<n>" (for instance A = B /<.5/> in the grammar means that in reality A is "B <.5>"). The /'s denote that the < and > are to be taken literally.

Here comes the grammar:

<text> ::= {<sentence>}$^{+}$

<sentence> ::=

<plainTerm> <truthValue>

| <firstOrderRelation> {<firstOrderArgument>}$^{+}$ [<truthValue>] [<punctuation>]

| <logicalOperator> {<Argument>}$^{+}$ [<truthValue>][<punctuation>]

| <predicateName> {<Argument>}$^{+}$ [<truthValue>][<punctuation>]

| <schemaName> {<Argument>}$^{+}$?

| <higherOrderRelation> {<higherOrderArgument>}$^{+}$ [<truthValue>][<punctuation>]

| Context <Argument> <higherOrderArgument> [<truthValue>][<punctuation>]

| SatisfyingSet <sentence> [<punctuation>]

| OutputValue <schemaName> {<Argument>}$^{+}$ [<punctuation>]

<punctuation> ::= . | ? | !

<truthValue> = [/< <strength> >/ | /< <strength, weight_of_evidence> >/]

<strength>: real number in [0, 1]
<weightOfEvidence>: real number in [0, 1]

<Argument> = <firstOrderArgument> | <higherOrderArgument>

<firstOrderArgument> = <term> | <variable>

<higherOrderArgument> = <sentence> | <variable>

<term> = (<plainTerm> | <pragmaticTerm> | <determinedTerm>) [<truthValue>]

<plainTerm> = string enclosed in quotes, or, string without whitespace that is neither a pragmaticTerm nor a varName

<pragmaticTerm> = one of the terms: this, that, that1, that2, now, here, there [not enclosed in quotes]

<determinedTerm> = <determiner> <plainTerm>

<determiner > = one of the terms: *this* [others may be added later, perhaps]

<predicateName> = <plainTerm> | <varName> | <determinedTerm>

<schemaName> = <plainTerm> | <varName> | <determinedTerm>

<variable> = <universalVariable> | <existentialVariable>

<universalVariable> = <plainUniversalVariable> | <determinedUniversalVariable>

<plainUniversalVariable> = string starting with _ : e.g _r, _x, etc.

<determinedUniversalVariable> = <determiner> <plainUniversalVariable>

<existentialVariable> = <existentialVariable> [<dependency>]

<existentialVariable> = string starting with !_ : e.g. !_r, _x, etc.

<dependency> ::= {<variable>}$^{+}$

The scope of a variable name is assumed to be the text. That is, if a certain variable name (say _x) occurs in the different sentences in the same text, it is assumed to have a common meaning in all the sentences. On the other hand, if the variable name _x occurs in different sentences in different texts, this is not

assumed; different texts have no semantic overlap except insofar as they may be decided to overlap within Novamente after they're loaded in. Each text is loaded into Novamente as a whole.

Next there, is a list of specific logicalOperator, firstOrderRelation and higherOrderRelation objects that we know are going to be useful. This list may be expanded over time. These have different arities, and formally speaking, the arities are parts of the formal grammar of KNOW, even though the above grammar is not written out that way.

In the following, I am using the notation <1> to denote 1-ary, <2> to denote binary, <n> to denote n-ary, etc.

<logicalOperator> ::= AND <n> | OR <n> | NOT <1> | XOR <n>| Ordered-AND <n>

<firstOrderRelation> = Inheritance <2>
| Similarity <n>
| Member <2>
| Subset <2>
| ExtensionalSimilarity <n>

|IntensionalInheritance <2>

|IntensionalSimilarity <n>
| SymmetricAssociation <n>

| AsymmetricAssociation <2>

| PartOf <2>

<higherOrderRelation> = Implication <2>
| Equivalence <n>
| ExtensionalImplication <2>
| ExtensionalEquivalence <n>

| IntensionalImplication <2>
| IntensionalEquivalence <n>
| Hypothetical <1>

Next, for sake of user sanity, KNOW should also support shorthand names. Suggested shorthand names are as follows:

Full Name	Shorthand
Implication	Imp
Equivalence	Equiv
ExtensionalImplication	ExtImp
ExtensionalEquivalence	ExtEquiv
IntensionalImplication	IntImp
IntensionalEquivalence	IntEquiv
Hypothetical	Hyp
AssymetricAssociation	AsymAss
SymmetricAssociation	SymAss
Inheritance	Inh
Similarity	Sim
ExtensionalSimilarity	ExtSim
IntensionalSimilarity	IntSim
IntensionalInheritance	IntInh
OutputValue	OutVal
SatisfyingSet	SatSet

Finally, there are two grouping methods that I have found useful: parentheses and indentation. I suppose KNOW should support both.

An example KNOW sentence expressed using both forms of grouping is

Parentheses:

Imp (Inh _x cat) (likes _x fish)

Indentation:

Imp

Inh _x cat

likes _x fish

3. Examples, and Mappings into Novamente Structures

In this section I will run through the various parts of the above-given grammar, giving examples of each part. I will illustrate the mapping of KNOW expressions into Novamente structures in the context of these examples.

The most natural way to review the grammar is to go through the sentence types one by one.

<firstOrderRelation> {<firstOrderArgument>}$^+$ [<truthValue>] [<punctuation>]

First, a note about punctuation. If no punctuation mark is given, the assumption is that the sentence is declarative, i.e. that the mark "." is implicitly intended.

Examples here would be

Inheritance cat animal

Similarity Ben Cassio Thiago Senna <.8>

Similarity Ben Cassio Izabela <.8, .9>

Member Ben "Novamente Team"

Shorthand examples are:

Inh cat animal

IntSim Ben Ken <.5> ?

A more interesting example is

Inheritance this red !

(This contains the ! for emphasis, and it also contains "this", which is a pragmaticTerm.)

Some question examples are:

Inheritance this blue?

Subset square circle?

The mapping of declarative (.) first-order relations into Novamente is obvious and simple.

In the case of binary arguments, the sentence becomes a single Novamente link. The arguments of the sentence are the source and target of the link. These arguments are ConceptNodes, and WordNodes or PhraseNodes are used for the strings used to label the concepts. If these WordNodes and PhraseNodes already exist, they are simply used; otherwise new ones are created. On the other hand, the question of whether new ConceptNodes are created is a subtle one.

Consider the case where the relevant Word/PhraseNode already exists, and there is an existing ConceptNode. Say there is an existing ConceptNode for "square", and the user has entered a KNOW text regarding "square." Do we want to simply assume that the user's text refers to the existing ConceptNode for "square"? Not necessarily. What if there is more than one ConceptNode linking to the WordNode for "square" – which one should be chosen?

For starters we can use a simple and temporary heuristic:

- If the relevant Word/PhraseNode already exists, we take the ConceptNode with the highest-strength AsymmetricAssociativeLink to it, and assume it is the right one.
- If there are no ConceptNodes linked to that Word/PhraseNode with AsymmetricAssociativeLinks, we simply create a new ConceptNode

This heuristic is totally inadequate because it does not support ambiguity. It will have to be replaced fairly quickly with a schema embodying a more sophisticated heuristic for sense disambiguation. In this more general approach, the system will have to create a new ConceptNode for each term in the KNOW text, and then there will be MindAgents and/or schemata that figure out how whether this new ConceptNode should be fused with an existing one or not.

So, for instance,

Inheritance cat animal

is mapped into (using Sasha notation)

```
ConceptNode: C_cat, C_animal

WordNode: W_cat#cat, W_animal#animal

InheritanceLink C_cat C_animal

AsymmetricAssociativeLink W_animal C_animal

AsymmetricAssociativeLink W_cat C_cat
```

On the other hand, the determiner *this* exists in KNOW syntax to allow the user to specify that he intends a particular entity rather than a general category.

So, for example,

Subset (this square) (this circle) ?

asks whether some particular square being referred to is a subset of some particular circle being referred to, whereas

Subset square circle ?

asks whether the general concept "square" is a subset of the general concept "circle." In the former case, the correct mapping into Novamente structures involves creating a new ConceptNode for (this square) and a new ConceptNode for (this circle). So one has a mapping like

SubsetLink

ConceptNode: square_1

ConceptNode: circle_1

InheritanceLink

square_1

ConceptNode: square

InheritanceLink

circle_1
ConceptNode: circle

It is up to Novamente cognition to figure out how the node we've called square_1 relates to previous instances of squares that the system has been told about.

We've been talking about declarations. But an exclamatory first-order relational sentence is entered into the system basically the same way, the only difference being that the system is supposed to note that the user has emphasized the sentence. So, for instance, if the system is sold

Inh cat animal !

we may have

```
EvaluationLink

      Emphasis
```

```
ListLink

      UserNode: U

      InheritanceLink C_cat C_animal
```

where *Emphasis* is a PredicateNode that we are singling out as the semantic mapping of the ! mark.

An interrogative sentence such as

Inh cat animal ?

is handled only a little differently. One enters the link into the system as if it were a declaration, and then enters also

```
UserNode: U

EvaluationLink

      Asked

      ListLink

            U

            OutputValueLink

                  TruthValue

                  InheritanceLink C_cat C_animal
```

This denotes that the user asked for the answer to the question: "What is the truth value of the link (InheritanceLink C_cat C_animal)?" Of course, it denotes this in a very primitive way: It is up to the system to figure out how to appropriately respond to *Asked* relations that are mapped into its mind via KNOW interpretation.

So far we have given only examples of isolated firstOrderRelation sentences, but these sentences may also occur embedded within other sentences. We'll see examples of that later.

<logicalOperator> {<Argument>}$^{+}$ [<truthValue>][<punctuation>]

Next, the logical operator based sentences.

These may be used with first-order terms as arguments, e.g.

AND cat ugly ?

AND square (NOT blue)

They may also be used with sentences as arguments, e.g.

Imp

Inh _X cat

AND (Inh _X fluffy) (eats _X mouse)

<predicateName> {<Argument>}$^{+}$ [<truthValue>][<punctuation>]

Predicate sentences are going to be very common in KNOW; for instance

```
above square triangle

inside circle square <.8>
```

We have already seen other examples of predicate sentences embedded within other KNOW sentences.

The first above example gets internally mapped into:

```
EvaluationLink

      PredicateNode: above_1

      ListLink

            ConceptNode: square_1

            ConceptNode: triangle_1

AsymmetricAssociativeLink

      WordNode: #"square"

      square_1

AsymmetricAssociativeLink

      WordNode: #"triangle"

      triangle_1
```

```
AsymmetricAssociativeLink

       WordNode: #above

       above_1
```

<higherOrderRelation> {<higherOrderArgument>}$^{+}$ [<truthValue>][<punctuation>]

For example, the transitivity of the "inside" relationship could be taught to the system via the higher-order KNOW sentence

Implication

AND

inside _x _y

inside _y _z

inside _x _z

An example with more complex use of variables would be "every square on the screen now is contained in a circle

Implication

Inheritance (this _x) square

AND

Inheritance _y(_x) circle

Contains *y(*_x) _x

In this case, the variable _y contains a dependency list, indicating that it depends on _x [each _x may lead to a different _y]. The interaction between variable dependency lists and the *this* predicate requires some subtlety. Basically, if a variable depends on a variable that occurs with the *this* determiner, then it must also be interpreted as a specific entity (a new node) upon being entered into Novamente. Also, if a variable occurs in a text once with a *this* determiner, then within that text it always must occur with a *this* determiner.

Context <Argument> <higherOrderArgument> [<truthValue>][<punctuation>]

The Context sentence corresponds to ContextLink in Novamente. For instance the KNOW utterance

```
Context skiing (Inh Ben incompetent)
```

becomes internally

```
ContextLink

      ConceptNode: skiing

      InheritanceLink

            UserNode: Ben

            ConceptNode: incompetent

AsymmetricAssociativeLink

      WordNode #"Ben"

      ConceptNode: Ben

AsymmetricAssociativeLink

      WordNode #"incompetent"

      ConceptNode: incompetent

AsymmetricAssociativeLink

      WordNode #"skiing"

      ConceptNode: skiing
```

SatisfyingSet <sentence> [<punctuation>]

SatisfyingSet is a construct used in Novamente to, essentially, do the same thing as quantifiers without explicitly using quantifiers. Having both SatSet and quantifiers in KNOW is redundant, but this redundancy may conceivably be found useful.

For instance, consider the sentence "Every boy loves some girl."

In terms of quantifiers, this is

Imp

Inh _b boy

AND

Inh _g(b) girl

loves _b _g(_b)

Without quantifiers, using SatSet, it's

Imp
Inh _b boy
Subset
SatSet (loves _b)
girl

Note that (loves _b) is interpreted as a predicate, using the currying convention from functional programming. This is what allows SatSet to work without existential variables and dependency lists.

OutputValue <schemaName> {<Argument>}$^{+}$ [<punctuation>]

This kind of sentence is only going to be used only within other sentences, or

For instance,

Sim "George W Bush" (OutputValue son_of "George H.W. Bush")

is another way of saying

Member "George W Bush" (SatisfyingSet (ExecutionLink son_of "George H.W. Bush" _x))

The mapping into internal Novamente structures is simple; the OutputValue relation becomes an OutputValueLink, and the schemaName becomes a SchemaNode linked to a WordNode containing the schemaName string (with an AsymAss link pointing from the WordNode). If there already exist SchemaNode(s) linked to the appropriate WordNode, then we have a familiar problem of disambiguation, which for starters can be hacked as in the other cases mentioned above.

<schemaName> {<Argument>} ?

This is a special shorthand used for asking questions,

For instance

OutputValue + 2 3 ?

asks the system to add 2 and 3.

But we may rephrase this simply as

+ 2 3 ?

Similarly to ask what's the neighbor of the square on the screen, we can ask

neighbor (this square) ?

<plainTerm> <truthValue>

This type of sentence is used to assign "node probabilities", e.g

aardvark <.01>

It's most useful in the context of Context sentences e.g.

Context zoo (aardvark <.01>)