LIVING IN CYN:
MATING AIML AND CYC TOGETHER WITH PROGRAM N

Kino Coursey
Daxtron Laboratories, Inc.
http:\\www.daxtron.com
January, 2004

This is a brief note on some of Daxtron Labs' experiments on linking the Artificial Intelligence Markup Language (AIML) interpreter called **Program N** (by Gary Debuque) to **OpenCyc** (a freeware version of the **Cyc** program created by Cycorp). The modified interpreter is called **CyN** (pronounced "sin"). The CyN interpreter reuses the basic resources of Program N, modifies its graphmaster pattern match algorithm, adds a few basic tags to allow translation of English to CycL (Cyc's internal language) and control of the formatting of OpenCyc's responses.

All work was done on a Windows™ XP laptop. The modifications made should have high portability to most Internet-enabled AIML interpreters. No modifications were made to OpenCyc. No modification should be required to swap full Cyc for OpenCyc.

The goals of the system are to:

- provide a pattern based parser for use with OpenCyc suitable for chatbots / virtual human experimentation and basic knowledge entry of logical relationships
- be an interpreter that allowed an easy way to interface AIML to OpenCyc
- create a set of example AIML categories to answer questions using OpenCyc
- provide a tool for experimentation with both systems
- provide a natural language front-end ahead of the current OpenCyc schedule
- be a tool capable of creating a "personality" for OpenCyc systems
- be a scaleable tool (AIML sets have 40,000 language patterns and full Cyc has 1.6 million facts)
- be a tool that can be integrated with other systems
- be open; based on Program N it is free to download, modify and reuse

For example, given (boss John Steve) "the boss of John is Steve" and the pre-existing OpenCyc ontology (version 0.7.1) the system could answer:

"Who is John boss"	
"Who does John know"	(people are acquainted with their bosses)
"Who is the boss of John"	
"Who is the superior of John"	(boss defines a hierarchy relationship)
"Who influences John"	(boss defines an influences relationship)
"Who does Steve boss"	
"Who does Steve influence"	

… and similar questions. CyN provides a way to access those answers via a conversational front end. As a side benefit it can also provide answers through a MSAgent character using text-to-speech.

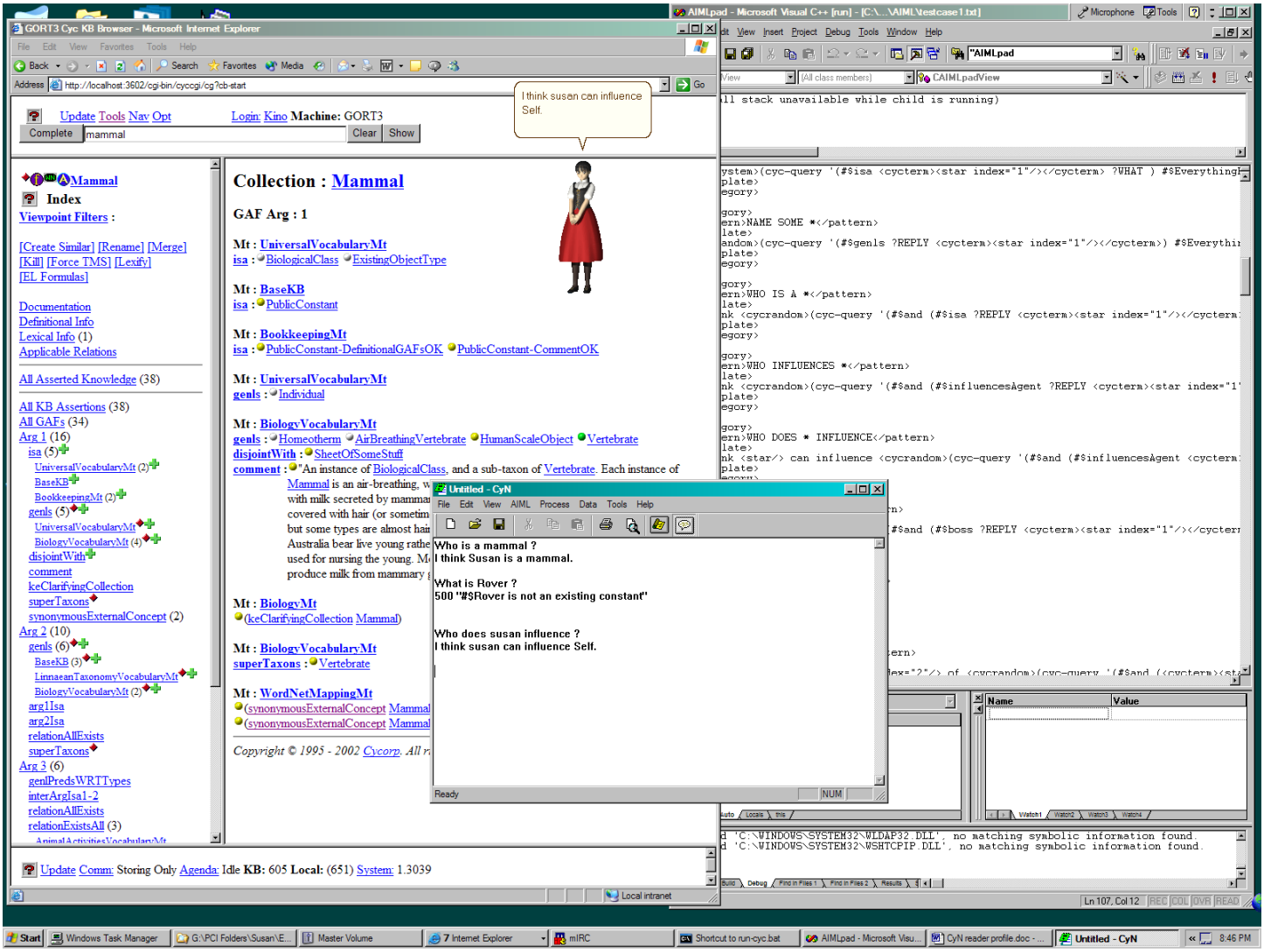
WARNING: This paper contains graphic depiction of promiscuous AIML <tag> development, and forced inter-model mixing. Please delete this paper if you find such behavior or its depiction offensive.

1. AIML, OPENCYC, CYCL API AND ENGLISH

CyN's Design Principles

One of the first questions that interested programmers want to know about OpenCyc is what kind of natural language interface is there and can they talk with it. As of January 2004 the natural language interfaces have not been released. So the primary goal was to provide such an interface that was easy to program, and as a side effect allowed for immediate rewarding feedback from learning CycL.

The core design principle was to provide maximum flexibility and transparency with minimum effort. AIML was selected due to its large user community, user interest, and easy to learn syntax. Into this were added the basic ability to communicate with OpenCyc and use its responses in replies or control the flow of execution. Another choice was as an experimental system to make the system complete first before focusing on efficiency. This does not mean it is inefficient, just that given a choice, simplicity and completeness had higher design value.



Screenshot of Daxtron's CyN Development System

What is AIML? … and the History of Pattern Matching

AIML stands for Artificial Intelligence Markup Language. It is an XML-based language created by Dr. Richard Wallace and primarily used to create conversational Chatbot applications. The reference character ALICE won the 2000 and 2001 Lobener prizes, and is used as a starting point for new developers. The core feature of the language is the easy ability to create conversational agents through the process of pattern matching. ALICE has 40,000 patterns that either trigger a direct response or cause a reinterpretation to a simpler form.

One of the most famous early pattern based natural language programs is ELIZA by Weizenbaum. ELIZA utilized several hundred patterns, and tried to mirror the personality of the conversational partner. After some analysis of conversations Dr. Wallace noticed that conversational inputs followed Zipf's law and that upwards of 95% of inputs could be

accounted for in common chat by several thousand patterns. Thus, ALICE's performance owes to the fact that for the most part in general conversation people tend to say only a few thousand things in different ways. AIML is designed to make the specification of these common conversational patterns fairly easy. It also provides for easy specification of surface variations in both input and output, which is useful for virtual human personality scripting.

```
<category>
<pattern>WHAT ARE YOU DOING</pattern>
<template>
       Just waiting for a mystery to solve.
</template>
</category>


<category>
<pattern>WHAT ARE YOU UP TO</pattern>
<template>
       <srai>WHAT ARE YOU DOING</srai>
</template>
</category>
```

Example AIML Categories

A factor from the early history of AI was the combination of semantic processing with pattern matching. ELIZA as part of its processing could execute arbitrary LISP programs of any complexity to answer a query. SIR (Semantic Information Retrieval) by Bertram Raphael in 1964 combined a pattern matching front end with a simple relational logic inference engine, to answer questions about the relation between objects. Both were created before a split between pattern processing and symbolic inference occurred. At the time the field was small enough that all methods were unified. CyN explores the possibility of reintegrating the two methods of large scale pattern matching (embodied in AIML) with large scale inference (embodied in OpenCyc) after a 40 year separation.

One of the best places to find out information about AIML is http://www.alicebot.org/articles/

What is OpenCyc?

The Cycorp Cyc project has released their "free to use version", OpenCyc. Currently a Linux and Windows XP binary executable, at 40-60 MB, OpenCyc claims to be a resource for commonsense reasoning. The system supports a full logic engine capable of reasoning with contradictions by defining local non-contradictory (i.e. locally consistent) subdomains. Tutorial, documentation and download are available at the site http://www.opencyc.org

OpenCyc is:

- a collection of facts that are a subset extracted from full Cyc
- an inference engine to reason using those facts and others acquired
- a browser interface for the facts and inference engine
- a lisp interpreter with a TCP/IP interface
- an API designed to access the inference engine using the TCP/IP interface

Cyc and OpenCyc support associating facts together in inherited clusters called microtheories. Different microtheories can contain contradictory, fictional or contra-factual statements, but all facts in a given chain of microtheories must be consistent. Microtheories allow explicit representation of the context of logical statements, providing focus to the inference engine. This design allows the system to reason about multiple scenario's with contradictions without breaking. ("Captain Kirk watch out!")

Tutorial materials are available at: http://www.opencyc.org/doc

The larger system Cyc, on which OpenCyc is based, has been in development for 19 years, and contains 1.6 million manually-entered facts and rules that interrelate 118,000 concepts.

What is the CycL API?

The CycL API defines a series of functions and modules that allow external applications to connect and uses the Cyc level knowledge base. The interface is via TCP/IP, and different services are provided on different ports. The default port for external applications is 3601. The port provides access both to Cyc level functions and access to the **SubL** LISP interpreter, the language Cyc is written in. You can access this port directly using telnet or via your program.

There are some basic functions and facts you need to know about the API:

```
(fi-complete "string")
```

returns a list of CycL symbols that might match "string"

```
(cyc-find-or-create cycsymbol)
```

returns the CycL symbol if it finds it, or if it does not find it, it creates a new one

```
(cyc-assert '(cycformula) #$MicrotheoryMt)
```

adds (cycformula) to the facts in the speficied MicrotheoryMt

```
(cyc-unassert '(cycformula) #$MicrotheoryMt)
```

removes (cycformula) from the facts in the specified MicrotheoryMt

```
(cyc-ask '(formula) #$MicrotheoryMt)
```

asks cyc to find the answers to the the formula in a certain Microtheory. Note that Cyc will use its inferencing abilities to find responses to the answers to the formula.

All Cyc symbols are prefaced by "#$". So, "dog" is "#$Dog" or "#$Doggie-TheWord".

The best current online reference sources detailing external connections to OpenCyc:

http://www.opencyc.org/doc/opencycapi

http://www.cyc.com/cycdoc/ref/subl-reference.html

English Terms and CycL Symbols

Cyc deals with symbols that are associated to one another by a web of relations. The relations and symbols denoting objects or concepts have names similar to the English or other language words. Words are combined in camel format. In camel format the phrase "United States Person" becomes "UnitedStatesPerson", which designated a concept that corresponds to "American". More on the relationship between words and symbols are explained in the section on the <cycterm> tag.

Also if note is the fact that the Cyc KB has a finer distinction of some relations than English. For instance:

- "A Collie is a Dog" = (#$genls #$Collie #$Dog) since Collie is a collection and Dog is one generalization of what a Collie is.
- "Rover is a Collie" = (#$isa #$Rover #$Collie) since Rover is an individual. Cyc will infer that Rover is a #$Dog, #$Mammal, #$Thing, etc.

2. THE "TAG" REFERENCE

Additional features are required in both AIML and OpenCyc to communicate effectively. In trying to keep with the minimalist tradition of AIML, the smallest set of new tags that would offer the most functionality was sought. The code to implement the new functionality was done in C++ in Program N. CyN thus is a superset of AIML as implemented by Program N. Most of the new tags correspond to existing AIML tags.

Several new Cyc-centric tags were created:

<cycterm>	translates an English word/phrase into a Cyc symbol
<cycsystem>	executes a CycL statement and returns the result
<cycrandom>	executes a CycL query and returns one response at random
<cycassert>	simple way to assert a CycL statement
<cycretract>	simple way to retract a CycL statement
<cyccondition>	controls the flow execution in a category template
<guard>	processes a template only if the CycL expression is true

These tags directly wrapper SubL/CycL expressions to allow for maximum flexibility with minimum overhead.

The <cycterm> tag

One core problem between the external world of English and the internal world of Cyc symbols is translating between the two. Cyc symbols are designed to be unambiguous. Each symbol is designed to have one meaning while a natural language word can have many. A word can be represented by the symbol "#$Word-TheWord". This symbol representing the "word" can then be associated with the many internal symbols. For instance "#$Dog" can be associated with "#$Dog-TheWord" , "#$Doggie-TheWord", "#$Puppy-TheWord" etc. Another example is "American" can be translate to the unique internal concept "#$UnitedStatesPerson".

<cycterm> is responsible for converting the external English term into the internal Cyc symbol by querying the OpenCyc KB.

```
found=lookupCycTerm("(fi-ask '(#$denotation #$%s-TheWord ?TEXT ?TYPE ?CYCOBJECT) #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$denotationRelatedTo #$%s-TheWord ?TEXT ?TYPE ?CYCOBJECT)
#$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$nameString ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$initialismString ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$abbreviationString-PN ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$preferredNameString ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$countryName-LongForm ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$countryName-ShortForm ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$acronymString ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$scientificName ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$termStrings ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$termStrings-GuessedFromName ?CYCOBJECT  \"%s\")
#$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$prettyName ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$nicknames ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$preferredTermStrings ?CYCOBJECT  \"%s\") #$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$preferredGenUnit ?CYCOBJECT ?POS #$%s-TheWord )
#$EverythingPSC)",text);

if (!found) found=lookupCycTerm("(fi-ask '(#$and (#$wordStrings ?WORD \"%s\") (#$or (#$denotation ?WORD ?TEXT
?TYPE ?CYCOBJECT) (#$denotationRelatedTo ?WORD ?TEXT ?TYPE ?CYCOBJECT) )) #$EverythingPSC)",ltext);

//Followed by asking Cyc to guess at the word using (fi-complete \"%s\")

// and if that fails returns a string of  using #$\"%s\"
```

Basic English word to Cyc term lookup process

This tends to find at least one plausible term. Often there are more. When there are multiples the system chooses to use the first. A better method would be to use an English to CycL parser. Additional KB resources also exist. See: http://www.cyc.com/cycdoc/ref/nl.html

We are looking at ways of providing phrase parsing to improve the functionality of <cycterm>.

One possible optimization is to cache the results.

The <cycsystem> tag

Cyc system does basically what the <system> tag does but is specifically modified to use the Cyc API port. Once the string is processed on the AIML side it is passed to the Cyc API port, and the response is returned. It does have one modification, in that the LISP terms used to signal either true or false are converted to "TRUE" or "FALSE". In this way the AIML interpreter does not need to parse LISP directly.

In the following AIML example:

```
<category>
<pattern>ARE YOU A *</pattern>
<template>
<srai>CYCREPLY <cycsystem>(cyc-query '(#$isa #$Self <cycterm><star
index="1"/></cycterm>) #$EverythingPSC) </cycsystem></srai>
</template>
</category>

<category>
<pattern>CYCREPLY *</pattern>
<template>
<star/>
</template>
</category>

<category>
<pattern>CYCREPLY 500 * IS NOT AN EXISTING CONSTANT</pattern>
<template>
I am sorry. Cyc does not have a Constant called <star/>.
</template>
</category>

<category>
<pattern>CYCREPLY TRUE</pattern>
<template>
```

```
<random>
<li>Yes. </li>
<li>Affarmitive. </li>
<li>Right. </li>
</random>
<random>
<li>I beleive that.</li>
<li>I agree.</li>
<li>I think so.</li>
</random>
</template>
</category>

<category>
<pattern>CYCREPLY FALSE</pattern>
<template>
<random>
<li>No. </li>
<li>Negative. </li>
<li>Nope. </li>
</random>
<random>
<li>I can't prove that.</li>
<li>I can't say that.</li>
<li>Not that I know of.</li>
</random>

</template>
</category>
```

If given "Are you a person?", the pattern would match with "*"= "person". Working from the inside out the system will translate `(cyc-query '(#$isa #$Self <cycterm><star index="1"/></cycterm>) #$EverythingPSC )` into `(cyc-query '(#$isa #$Self <cycterm>person</cycterm>) #$EverythingPSC )` and then into `(cyc-query '(#$isa #$ #$Person) #$EverythingPSC )`. This is passed to <cycsystem> for execution, which will return TRUE. <srai> will then process CYCREPLY TRUE, match another pattern and return "Right. I believe that."

The <cycrandom> tag

<Cycrandom> combines both the features of the regular AIML <random>/ tag combination with the <cycsystem> tag function. First, it executes a formula on OpenCyc. It then collects all the responses that bind to the ?REPLY variable and converts them into a set of <li> tags. It then passes these to a copy of the existing <random> tag processor. The result is that it returns one of the items that matches ?REPLY.

For example:

```
<category>
<pattern>WHO IS AN *</pattern>
<template>
      I think <cycrandom>(cyc-query '(#$and (#$isa ?REPLY <cycterm><star
      index="1"/></cycterm>) (#$isa ?REPLY #$Person) )#$EverythingPSC)
      </cycrandom> is a <star/>.
</template>
</category>
```

Would match "Who is an american" and might respond, "Susan is a American", where "American" is translated to "#$UnitedStatesPerson" by the <cycterm> tag.

The use of <cycrandom> opens the door for storing and using responses on the Cyc side. The responses returned can be in response to very complex queries, and take in to account items like the expected consequences of making a particular response, or collect them from the microtheories active at the time.

The <cycretract> and <cycassert> tags

<cycassert> will accept a CycL formula and assert it to the CycKB.

<cycretract> will first find those terms that match the expression, then un-assert it from the CycKB.

```
<category>
<pattern>FORGET * IS A *</pattern>
<template>
<cycretract>(#$isa <cycterm><star index="1"/></cycterm> <cycterm><star
index="2"/></cycterm> ) #$AimlContextMt</cycretract>
</template>
</category>


<category>
<pattern>MAKE * A *</pattern>
<template>
```

```
<cycsystem>(fi-find-or-create "<star index="1"/>")</cycsystem>
<cycassert>(#$isa <cycterm><star index="1"/></cycterm> <cycterm><star
index="2"/></cycterm>) #$AimlContextMt</cycassert>
</template>
</category>


<category>
<pattern>WHAT IS *</pattern>
<template>
<cycsystem>(fi-ask '(#$isa <cycterm><star index="1"/></cycterm> ?REPLY)
#$AimlContextMt)</cycsystem>
</template>
</category>
```

The <cyccondition> tag

The <cyccondition> is a template side tag. It allows branching to occur based on the outcome of a CycL query. The <cyccondition> has two forms: the single condition and list-conditional. Multiple <cyccondition> can be listed inside a single <template>. The CycL query can be contained either inside the query attribute of the <cyccondition> or enclosed tags. The query is run through the same process as <cycsystem> and true or false is returned, with the results controlling execution.

The format can be (case 1):

```
<cyccondition query="cycL query"> AIML Statements </cyccondition>
```

—OR (case 2)—

```
<cyccondition>
      <li query="cycL query1"> AIML Statements </li>
       :
       :
      <li query="cycL queryN"> AIML Statements </li>
      <li>Default AIML statements </li>
</cyccondition>
```

—OR (case 3)—

```
<cyccondition query="cycL query1"> AIML Statements </cyccondition>
<cyccondition query="cycL query2"> AIML Statements </cyccondition>
      :
      :
<cyccondition query="cycL queryN"> AIML Statements </cyccondition>
```

In the first case the AIML statements only execute if the query is true.

In the second case only the first one with a true query will be executed.

In the last case each one whose query matches will be executed.

As tested below <cycconditions> can be nested:

```
<category>
<pattern>TESTY *</pattern>
<template>
Test Y =
       <cyccondition query="(fi-ask '(#$genls <cycterm><star/></cycterm>
       #$Mammal) #$AimlContextMt)"> Its a Mammal.</cyccondition>

       <cyccondition query="(fi-ask '(#$genls <cycterm><star/></cycterm>
       #$PhysicalDevice) #$AimlContextMt)"> Its a Physical Device
       Thingy.</cyccondition>

       <cyccondition query="(fi-ask '(#$genls <cycterm><star/></cycterm>
       #$Thing) #$AimlContextMt)"> Its a Thingy.</cyccondition>
</template>
</category>

<category>
<pattern>TESTZ *</pattern>
<template>
Test Z =
       <cyccondition query="(fi-ask '(#$genls <cycterm><star/></cycterm>
       #$Thing) #$AimlContextMt)"> Its a Thingy.
       <cyccondition query="(fi-ask '(#$genls <cycterm><star/></cycterm>
       #$PhysicalDevice) #$AimlContextMt)"> Its a Physical Device
       Thingy.</cyccondition>
       </cyccondition>

       <cyccondition query="(fi-ask '(#$genls <cycterm><star/></cycterm>
       #$Mammal) #$AimlContextMt)"> Its a Mammal.</cyccondition>
</template>
</category>
```

Dialog using above categories:

```
testy dog
Test Y =  Its a Mammal. Its a Thingy.
```

```
testy car
Test Y =  Its a Physical Device Thingy. Its a Thingy.


testz dog
Test Z =  Its a Thingy. Its a Mammal.


testz car
Test Z =  Its a Thingy. Its a Physical Device Thingy.


testz plant
Test Z =  Its a Thingy.
```

The <guard> tag

A way to have a distributed version of the <cyccondition> tag is via the <guard> tag. The <guard> tag is usually the first tag of a template, and can specify that a CycL condition to be tested. If that condition is not met, the system backtracks to try other patterns that can match both the <pattern> and <guard> OR matches the <pattern> and has no <guard>. This allows CycL conditions to be part of the category matching process.

Before ELIZA and PARRY there was SIR (Semantic Information Retrieval). SIR used both pattern matching AND a hand written logical inference engine. As part of its pattern matching was the ability to query the inference engine to determine if a pattern element belonged to a certain class *before* performing the action. This idea over time came to be expanded into the class of semantic parsers. The <guard> provides a means to implement this functionality in CyN.

The <guard> is implemented in the <template> because in Program N to access the <star> info that is often of interest, you have to be in the <template>. In addition, the graphmaster algorithm (used by many AIML interpreters) expects words as test conditions, not arbitrary evaluation strings. Since the <guard> is a condition for execution of the template, technically it should appear outside the <template> tag. Such a change may be considered in a future release.

Example categories using <guard>:

```
<category>
<pattern>ID A *</pattern>
<template>
<guard>(fi-ask  '(#$genls
<cycterm><star/></cycterm> #$Mammal)
#$AimlContextMt)</guard>
Its a Mammal.
</template>
</category>

<category>
<pattern>ID A *</pattern>
<template>
<guard>(fi-ask '(#$genls
<cycterm><star/></cycterm>
#$PhysicalDevice)
#$AimlContextMt)</guard>
Its a Physical device(tm).
</template>
</category>

<category>
<pattern>ID A *</pattern>
<template>
<guard>(fi-ask '(#$genls
<cycterm><star/></cycterm> #$Thing)
#$AimlContextMt)</guard>
Its Something.
</template>
</category>

<category>
<pattern>ID A *</pattern>
<template>
You know, I don't know what <star/>
really is.
</template>
</category>
```

Test dialog using above categories

```
ID a Dog
Its a Mammal.

ID a Car
Its a Physical device(tm).

ID a American
Its a Mammal.

ID a country
Its Something.

ID a Cat
You know, I don't know what CAT really
is.
```

3. OPENCYC AS A PREDICATE STORE

A deeper coupling can exist between OpenCyc and AIML. AIML during a conversation can store and use variables based on the dialog. These variables could be reported and retrieved from OpenCyc. In AIML variables are called “predicates”, which is a word also used by first order predicate logic and Cyc.

I modified the Program N predicate/variable storage mechanism to send changes to an OpenCyc microtheory called AimlContextMt. A snapshot is included in the Appendixes. The variable "myvar" would be stored using (#$aimlPred #$aimlvar-myvar "Myvar's value").

Queries from the AIML interpreter to OpenCyc can be either result in a direct lookup or the use of inference. This allows the values of the AIML side predicates to be inferred when possible from OpenCyc.

Another aspect is to make the <topic> tag variable controlled from the OpenCyc side. This again allows complex rules to be created on the OpenCyc side to vary the response or processing selected. For instance, depending on the relationship with or likes of the user, the system can select categories that produce more different output.

4. SAMPLE AIML CATEGORIES

Let's say we wanted to ask Cyc "Who is a female?"

In CycL/SubL this could be answered by:

```
(cyc-query '(#$and (#$isa ?WHO #$FemaleAnimal)(#$isa ?WHO #$Person))
#$EverythingPSC)
```

…and Cyc will return a list of all the symbols that match both being a female animal and a person. #$EverythingPSC is a microtheory connected to everything in system. So in the context of everything in knowledge base, find something that is a female animal and a person. In my case this returns:

```
200 (((?WHO . #$Femie)) ((?WHO . #$Susan)) ((?WHO . #$Self)))
```

We can write a generic category using the above tags

```
<category>

<pattern>WHO IS A *</pattern>

<template>

I think <cycrandom>(cyc-query '(#$and (#$isa ?REPLY <cycterm><star
index="1"/></cycterm>) (#$isa ?REPLY #$Person) )#$EverythingPSC )</cycrandom>
is a <star/>.

</template>

</category>
```

We replaced the ?WHO with ?REPLY. Any value bound to the ?REPLY variable will be selected at random by the <cycrandom> tag. We let the system take its best guess at the object class being requested via the "cycterm><star index="1"/></cycterm>".

This category will then produce the following:

```
User: who is a female

Cyn : I think Self is a female.

User: who is a american

Cyn : I think Susan is a american.
```

Variations in syntax can easily be handled with additional <srai> based categories.

5. SAMPLE DIALOG

The following was done with early tests of the system. Its purpose was to test the interface, not to provide an example of a finished system.

Is a human a mammal ?

Affarmitive. I agree.

is a dog a vegetable ?

Nope. I can't prove that.

is kino alive ?

Affarmitive. I agree.

is a lion a animal ?

Affarmitive. I think so.

is a lion a product ?

Nope. I can't prove that.

is a lion a thing ?

Affarmitive. I beleive that.

is a dog a plant ?

Nope. I can't prove that.

Name some plant

FloweringPlant

is a dog a person

Negative. I can't say that.

who is a cyclist

I think CycAdministrator is a cyclist.

Is CycAdministrator a cyclist

Right. I agree.

Is CycAdministrator a person ?

Affarmitive. I agree.

Who is a student?

I think Self is a student. *<-- one of the things in the system profile*

Are you a studen *?* *<-- note: fi-complete fixed the typo and guessed right*

Right. I think so.

guess studen

I think studen means #$Student to Cyc

What do you want ?

I want Susan to like Self.

What do you want ?

I want Self to like Self.

[Since the cyc explainer is not avaiable yet through the API I asked OpenCyc through the web browser and got the following justification]

Mt : DAX-SelfAwareMt

EL Query :

(desires Self ?WHAT)

Answer Bindings :

?WHAT ? (likes-Generic Self Self)

Justifications :

(implies

(isa ?ANYONE Person)

(desires Self

(likes-Generic ?ANYONE Self))) in DAX-SelfAwareMt

:ISA (isa Self Person) in DAX-SelfAwareMt

Hence the system wants to like itself.

Make Rover a Dog

200 #$Rover

200 T

Make Skippy a Dog

200 #$Skippy

200 T

Is Skippy a Dog ?

Affarmitive. I agree.

Is Rover a Dog ?

Right. I agree.

Is Skippy a Person ?

No. I can't prove that.

What is Skippy?

200 (((?REPLY . #$Thing)) ((?REPLY . #$Individual)) ((?REPLY . #$Dog)))

Is skippy a mammal ?

Right. I agree.

Is skippy a animal ?

Affarmitive. I beleive that.

is skippy a person ?

No. Not that I know of.

What is rover ?

200 (((?REPLY . #$Thing)) ((?REPLY . #$Individual)) ((?REPLY . #$Dog)))

Forget Skippy is a Dog

()

What is Skippy ?

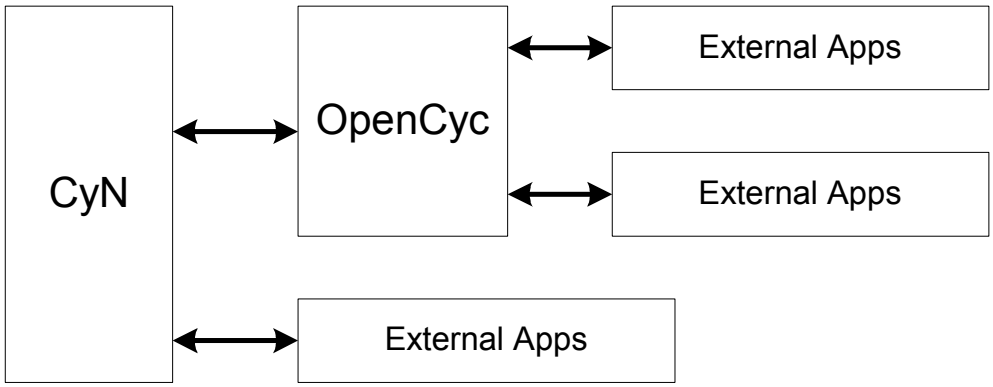
FALSE

Is skippy a mammal ?

Negative. I can't prove that.

6. FUTURE EXTENSION IDEAS AND OPEN ISSUES

One possibility opened up is the use of external applets and agents that are interfaced through the microtheory system and controllable through AIML. Internal rules can monitor and control systems while system status can be queried via AIML, along with instruction for change. Anything invokeable through the SubL interface (like the planner, future parsing tools, explanation mechanisms…) should be accessible through CyN.



The pattern matching system coupled with <topic> can be used to provide a manual solution to disambiguation. One word or phrase may have multiple meanings, but context provides a solution to selecting the right one. This can be controlled by placing different interpretations in different microtheories and then selecting the proper set of microtheories to use based on context. Allowing such context changes or dynamic redefinition of the microtheory hierarchy to remains to be implemented, but is fairly straightforward. Such implementation would involve a simple retraction then reassertion of the link between microtheories.

The system can easily make use of additional linguistic resources. An English to CycL parser is being developed in the form of Daxtron Labs' **Rosetta** parser, and Cycorp's yet-to-be-released natural language tools. Using it with either tool or others would add additional flexibility in the interface categories. Another possibility is to construct a simple noun-phrase parser, which is slated for earlier release by the OpenCyc project. One possibility to be explored is to parse the atomic AIML patterns (those without "*"'s in them) with an CycL aware parser, and generate categories that make the corresponding assertions in the OpenCyc KB. Such a catalog of parses can be fine tuned or contextualized as required by the target application.

I believe that the <cyccondition> and <guard> tag has the most potential for providing an effective link between system. With them the AIML flow of execution can be controlled via CycL logical statements. This has additional power when coupled with external applications making assertions to the OpenCyc KB. A great deal of processing can be hidden in a single statement. For a tongue-in-cheek example:

```
<category>

<pattern>REPORT SYSTEM STATUS</pattern>

<template>

       <cyccondition query="(fi-ask '(#$processstatus #$reactor
       #$critical))">Evacuate facility! The reactor is melting. Run!!!
       </cyccondition>

        <cyccondition query="(fi-ask '(#$processstatus #$reactor
       #$normal))">Reactor is fine</cyccondition>

       <cyccondition query="(fi-ask '(#$processstatus #$security
       #$compromised))"> Intruders are detected</cyccondition>

       <cyccondition query="(fi-ask '(#$processstatus #$security
       #$normal))">Everything is secure</cyccondition>

</template>

</category>


<category>

<pattern>WHAT DO YOU SEE </pattern>

<template>

       <cyccondition>

              <li query="(fi-ask '(#$sees #$self #$Dog))"> I see a doggy. </li>

              <li query="(fi-ask '(#$sees #$self #$Cat))"> I see a kitty. </li>

              <li query="(fi-ask '(#$and (#$sees #$self ?SOMEONE)(#$isa
              #$?SOMEONE #$Friend)))"> I see someone neat. :-)  </li>

              <li >I don't know how to describe what I see</li>

       </cyccondition>

</template>

</category>
```

Each query through OpenCyc could trigger either a simple lookup or a very long proof process using any number of the OpenCyc supported inference processes. But the desired information and its response are compactly represented in a way that allows easy scripting.

The other possibility brought up by the <cyccondition> and <guard> tag is to possibly create a generic tag that specifies external evaluation in general and not just to OpenCyc. Thus <externalcondition method="{sql,lisp,prolog,cycl…}" query="query text in method language"> would do the same thing as <cyccondition> but for any of the supported target languages. The same benefits should result but with even wider usage possibilities. AIML methods exist for generation of clarification dialogs and can be reused with the new tag set to provide for basic knowledge entry functions. One natural consequence of storing responses in OpenCyc (as with using <cycrandom> tag) is you can just assert new responses, including rules. Also you can store responses that in turn make assertions in OpenCyc for use on the AIML side.

One shortcoming of the existing system is that the CycL is opaque strings to CyN; text transformed according to the rules of AIML and passed to OpenCyc. This allows flexibility, but

it also ignores CycL syntax. Detailed error checking of CycL on the AIML side was not in the scope of the project.

And finally, is the question of actually creating a new AIML set that has the coverage of Alice with the appropriate hooks to OpenCyc. Some of this could be done through automated tools parsing the existing AIML set and producing results for review. But ultimately it will require lots of bot masters familiar both with AIML and CycL writing or reviewing lots of categories. The one advantage of involving the AIML/ CycL communities is they both are built on large, mostly manually entered knowledge bases, and both recognize that effort is required for achieving the results they desire. I hope this project will convince some that a merger of efforts is useful to goals of both groups.

7. CONCLUSION

It is indeed possible to interface OpenCyc and AIML. While most of the basic features could be implemented using the <system> tag, but the few additions simplify the interface task for the AIML/CycL writer, and speed up the interaction. The direct coupling of the AIML predicate store to a Cyc microtheory requires further investigation at this time. The addition of <cyccondition> and <guard> tags opens up other levels of context sensitivity made possible by combinations of complex facts.

Chatbots as a user interface could have uses in many domains, some of which are outside the normal social conversation system. For better or worse, people expect a system to both handle real, complex tasks as well as to be sociable. CyN hopes to bridge that gap by providing access to a powerful first-order predicate logic engine in the form of OpenCyc, coupled with the personality scripting polish of AIML. Both the left-brain and right-brain are needed to make a complete system.

The early LISP programs of SIR and ELIZA had the ability to evaluate arbitrary expressions and invoke any LISP program. CyN brings that talent to AIML, through its coupling with the SubL interface. Hundreds of person-years of Cycorp commonsense research is now accessible through an easy-to-use scripting front-end.

The ultimate question is does such a coupling between systems create something that is more useful than either alone. For the application area of creating models using Chatbots, we believe that this is the case.

APPENDIX A — ACKNOWLEDGEMENTS

ALICE A.I Foundation, Inc. and Dr. Richard Wallace for Alice and AIML

Cycorp and Douglas Lenat for Cyc and OpenCyc

Gary Debuque for Program N

The AIML and OpenCyc Forums and User communities

Douglas Miles for insight into English parsing using OpenCyc

APPENDIX B — REFERENCES

Prolog References from Dr. Wallace's PSYCH paper:

1. Prolog Tutorial
http://cs.wwc.edu/~cs_dept/KU/PR/Prolog.html
2. A Short Tutorial on Prolog
http://cbl.leeds.ac.uk/~tasmin/prologtutorial/
3. Prolog Programming: A First Course
http://cbl.leeds.ac.uk/~paul/prologbook/
4. GNU Prolog site
http://pauillac.inria.fr/~diaz/gnu-prolog.

AIML References:

1. Documentation
http://alicebot.org/documentation
2. Don't Read Me: Program dB
http://alicebot.org/articles/wallace/dont-dB.html
3. AIML Specification (Working Draft)
http://alicebot.org/TR/2001/WD-aiml

OpenCyc KB in DAML format (they forgot to update the date side bar so it looks old):

http://www.cyc.com/2002/04/08/cyc.daml

ThoughtTreasure in CycL format (lots of assertions about everyday stuff):

http://www.signiform.com/tt/ttkb/tt0.00022.cycl.gz

Also some additional KB pointers at:

http://www.daxtron.com/Ontology.htm

http://www.wisdominajar.com/prolog/prolog01.html shows how to create a "Prolog"-style reasoning system in AIML.

http://www.wisdominajar.com/prolog/prolog02.html is a description of an experiment linking the Prolog interpreter PSYCH to Program D using <system>.

Dr. Wallace posted an article about this, "PSYCH - Activating Prolog from AIML":
http://list.alicebot.org/pipermail/alicebot-style/2001-November/000041.html

He also taught us that we can do it all in AIML:
http://list.alicebot.org/pipermail/alicebot-style/2001-December/000013.html

An very early system that combined pattern matching with semantic processing is found in Bertram Raphael's thesis "SIR: A Computer Program for Semantic Information Retrieval", June 1964 paper "AITR-220" on
http://www.ai.mit.edu/research/publications/browse/0200browse.shtml

ftp://publications.ai.mit.edu/ai-publications/pdf/AITR-220.pdf (14 Meg snapshots)

APPENDIX C — AIML CATEGORIES

```
<aiml>


<!-- Convert YOU to Self --->


<category>
<pattern>* YOU *</pattern>
<template><srai><star index="1"/> SELF <star index="2"/></srai></template>
</category>


<category>
<pattern>* YOU</pattern>
<template><srai><star index="1"/> SELF</srai></template>
</category>


<category>
<pattern>YOU *</pattern>
<template><srai> SELF <star index="1"/></srai></template>
</category>


<!-- Basic Text cases -->


<category>
<pattern>CYCSET *</pattern>
<template>
<think><set name="cycvar"><star/></set></think>
I tried to set cycvar to <star/>.
</template>
</category>


<category>
<pattern>CYCGET *</pattern>
<template>
I tried to get cycvax as <get name="cycvax"/>.
</template>
</category>
```

```
<category>
<pattern>CYCISA * A *</pattern>
<template>
<cycsystem>(cyc-query '(#$genls <cycterm><star index="1"/></cycterm> <cycterm><star
index="2"/></cycterm>) #$EverythingPSC )</cycsystem>
</template>
</category>

<category>
<pattern>COMPLETE *</pattern>
<template>
<cycsystem>(fi-complete "<star index="1"/>" )</cycsystem>
</template>
</category>

<category>
<pattern>EXPLAIN *</pattern>
<template>
<cycrandom>(fi-ask '(#$comment <cycterm><star index="1"/></cycterm> ?REPLY)
#$EverythingPSC )</cycrandom>
</template>
</category>

<category>
<pattern>TRANSLATE *</pattern>
<template>
<cycsystem>(fi-ask '(#$denotation #$<sentence><star index="1"/></sentence> ?TEXT ?TYPE
?CYCOBJECT) #$EverythingPSC)</cycsystem>
</template>
</category>

<category>
<pattern>GUESS *</pattern>
<template>
I think <star/> means <cycterm><star index="1"/></cycterm> to Cyc
</template>
</category>
```

```
<category>
<pattern>CYCLI *</pattern>
<template>
<cycrandom>(fi-ask '(#$aimlResponse #$response1 ?REPLY) #$AimlContextMt)</cycrandom>
</template>
</category>


<category>
<pattern>CYCIT *</pattern>
<template>
<cycrandom>(fi-ask '(#$denotation #$<sentence><star index="1"/></sentence>-TheWord
?TEXT ?TYPE ?REPLY) #$EverythingPSC)</cycrandom>
</template>
</category>




<!-------------- "NORMAL" USE ------------>
<category>
<pattern>WHAT IS A *</pattern>
<template>


Generalizations:
<cycsystem>(cyc-query '(#$genls <cycterm><star index="1"/></cycterm> ?WHAT )
#$EverythingPSC )</cycsystem>


Instance Of:
<cycsystem>(cyc-query '(#$isa <cycterm><star index="1"/></cycterm> ?WHAT )
#$EverythingPSC )</cycsystem>
</template>
</category>


<category>
<pattern>NAME SOME *</pattern>
<template>
<cycrandom>(cyc-query '(#$genls ?REPLY <cycterm><star index="1"/></cycterm>)
#$EverythingPSC )</cycrandom>
</template>
</category>
```

```
<category>
<pattern>WHO IS A *</pattern>
<template>
I think <cycrandom>(cyc-query '(#$and (#$isa ?REPLY <cycterm><star
index="1"/></cycterm>) (#$isa ?REPLY #$Person) )#$EverythingPSC )</cycrandom> is a
<star/>.
</template>
</category>

<category>
<pattern>WHO INFLUENCES *</pattern>
<template>
I think <cycrandom>(cyc-query '(#$and (#$influencesAgent ?REPLY <cycterm><star
index="1"/></cycterm>) (#$isa ?REPLY #$Person) )#$EverythingPSC )</cycrandom> can
influence <star/>.
</template>
</category>

<category>
<pattern>WHO DOES * INFLUENCE</pattern>
<template>
I think <star/> can influence <cycrandom>(cyc-query '(#$and (#$influencesAgent
<cycterm><star index="1"/></cycterm> ?REPLY) (#$isa ?REPLY #$Person) )#$EverythingPSC
)</cycrandom>.
</template>
</category>

<category>
<pattern>WHO DOES * BOSS</pattern>
<template>
I think <cycrandom>(cyc-query '(#$and (#$boss ?REPLY <cycterm><star
index="1"/></cycterm> ) (#$isa ?REPLY #$Person) )#$EverythingPSC )</cycrandom> works
for <star/>.
</template>
</category>

<category>
<pattern>WHO ELSE IS *</pattern>
<template>
<srai>WHO IS <star/></srai>
```

```
</template>

</category>


<category>

<pattern>* IS THE * OF WHO</pattern>

<template>

I think <star/> is the <star index="2"/> of <cycrandom>(cyc-query '(#$and
(<cycterm><star index="2"/></cycterm> ?REPLY <cycterm><star index="1"/></cycterm> )
(#$isa ?REPLY #$Person) )#$EverythingPSC )</cycrandom>.

</template>

</category>


<category>

<pattern>WHO IS THE * OF *</pattern>

<template>

I think <cycrandom>(cyc-query '(#$and (<cycterm><star index="1"/></cycterm>
<cycterm><star index="2"/></cycterm> ?REPLY) (#$isa ?REPLY #$Person) )#$EverythingPSC
)</cycrandom> is the <star index="1"/> of <star index="2"/>.

</template>

</category>


<category>

<pattern>WHO IS AN *</pattern>

<template>

I think <cycrandom>(cyc-query '(#$and (#$isa ?REPLY <cycterm><star
index="1"/></cycterm>) (#$isa ?REPLY #$Person) )#$EverythingPSC )</cycrandom> is a
<star/>.

</template>

</category>


<category>

<pattern>WHO IS * MATE</pattern>

<template>

I think <cycrandom>(cyc-query '(#$and  (#$or (#$mate <cycterm><star
index="1"/></cycterm> ?REPLY)(#$mate ?REPLY <cycterm><star index="1"/></cycterm>))
(#$isa ?REPLY #$Person) )#$EverythingPSC )</cycrandom> is mate of <star/>.

</template>

</category>


<category>

<pattern>WHO IS * LOVE INTEREST</pattern>
```

```
<template>

I think <cycrandom>(cyc-query '(#$and  (#$or (#$romanticInterest <cycterm><star
index="1"/></cycterm> ?REPLY)(#$romanticInterest ?REPLY <cycterm><star
index="1"/></cycterm>)) (#$isa ?REPLY #$Person) )#$EverythingPSC )</cycrandom> is
romantic interest of <star/>.

</template>

</category>



<category>

<pattern>WHO DOES * KNOW</pattern>

<template>

I think <star/> is aquainted with <cycrandom>(cyc-query '(#$and  (#$or
(#$acquaintedWith <cycterm><star index="1"/></cycterm> ?REPLY)(#$acquaintedWith ?REPLY
<cycterm><star index="1"/></cycterm>)) (#$isa ?REPLY #$Person) )#$EverythingPSC
)</cycrandom>.

</template>

</category>



<category>

<pattern>ARE YOU A *</pattern>

<template>

<srai>CYCREPLY <cycsystem>(cyc-query '(#$isa #$Self <cycterm><star
index="1"/></cycterm>) #$EverythingPSC )</cycsystem></srai>

</template>

</category>



<category>

<pattern>WHY IS A * A *</pattern>

<template>

<cycsystem>(why-isa? <cycterm><star/></cycterm> <cycterm><star index="2"/></cycterm> )
</cycsystem>

</template>

</category>



<category>

<pattern>WHY IS A * AN *</pattern>

<template>

<cycsystem>(why-isa? <cycterm><star/></cycterm> <cycterm><star index="2"/></cycterm> )
</cycsystem>

</template>

</category>
```

```
<category>
<pattern>WHY IS AN * A *</pattern>
<template>
<cycsystem>(why-isa? <cycterm><star/></cycterm> <cycterm><star index="2"/></cycterm> )
</cycsystem>
</template>
</category>

<category>
<pattern>WHY IS AN * AN *</pattern>
<template>
<cycsystem>(why-isa? <cycterm><star/></cycterm> <cycterm><star index="2"/></cycterm> )
</cycsystem>
</template>
</category>

<category>
<pattern>WHAT DO YOU WANT</pattern>
<template>
<srai>WHAT DOES SELF WANT</srai>
</template>
</category>

<category>
<pattern>WHAT DOES SELF WANT</pattern>
<template>
<srai>SELFWANTS <cycrandom>(fi-ask '(#$desires #$Self ?REPLY ) #$DAX-SelfAwareMt
3)</cycrandom></srai>
</template>
</category>

<category>
<pattern>WHAT DOES * WANT</pattern>
<template>
<cycsystem>(fi-ask '(#$desires <cycterm><star/></cycterm> ?REPLY ) #$EverythingPSC
3)</cycsystem>
</template>
</category>

<category>
```

```
<pattern>SELFWANTS LIKES-GENERIC * *</pattern>
<template> I want <star index="1"/> to like <star index="2"/>. </template>
</category>

<category>
<pattern>SELFWANTS *</pattern>
<template> I want whatever ( <star index="1"/> ) means in English. </template>
</category>

<!--------- The PSYCH ISA Categories ----------->
<!--------- note OpenCyc uses #$isa and #$genls  for 'is a' -------->

<category>
<pattern>IS A * A *</pattern>
<template>
<srai>CYCREPLY <cycsystem>(cyc-query '(#$or (#$isa <cycterm><star
index="1"/></cycterm> <cycterm><star index="2"/></cycterm>)(#$genls <cycterm><star
index="1"/></cycterm> <cycterm><star index="2"/></cycterm>)) #$EverythingPSC
)</cycsystem></srai>
</template>
</category>

<category>
<pattern>IS A * AN *</pattern>
<template><srai>IS A <star/> A <star index="2"/></srai></template>
</category>

<category>
<pattern>IS AN * A *</pattern>
<template><srai>IS A <star/> A <star index="2"/></srai></template>
</category>

<category>
<pattern>IS AN * AN *</pattern>
<template><srai>IS A <star/> A <star index="2"/></srai></template>
</category>

<category>
```

```
<pattern>IS * A *</pattern>
<template><srai>IS A <star/> A <star index="2"/></srai></template>
</category>

<category>
<pattern>IS * AN *</pattern>
<template><srai>IS A <star/> A <star index="2"/></srai></template>
</category>

<category>
<pattern>IS * *</pattern>
<template><srai>IS A <star/> A <star index="2"/></srai></template>
</category>

<!------ If you want to SARI into a AIML based processor set --------->

<category>
<pattern>CYCREPLY *</pattern>
<template>
<star/>
</template>
</category>

<category>
<pattern>CYCREPLY 500 * IS NOT AN EXISTING CONSTANT</pattern>
<template>
I am sorry. Cyc does not have a Constant called <star/>.
</template>
</category>

<category>
<pattern>CYCREPLY TRUE</pattern>
<template>
<random>
<li>Yes. </li>
<li>Affarmitive. </li>
```

```
<li>Right. </li>
</random>
<random>
<li>I beleive that.</li>
<li>I agree.</li>
<li>I think so.</li>
</random>
</template>
</category>

<category>
<pattern>CYCREPLY FALSE</pattern>
<template>
<random>
<li>No. </li>
<li>Negative. </li>
<li>Nope. </li>
</random>
<random>
<li>I can't prove that.</li>
<li>I can't say that.</li>
<li>Not that I know of.</li>
</random>
</template>
</category>

</aiml>
```

APPENDIX D — OPENCYC KE TEXT

```
Default Mt: UniversalVocabularyMt.


Constant: AimlContextMt.

isa: DataMicrotheory.

genlMt: BaseKB.


comment: "#$AimlContextMt contains storage location in OpenCyc for AIML variables".



Default Mt: AimlContextMt.


Constant:aimlPred.

isa:BinaryPredicate.

arg1Isa:Thing.

arg2Isa:Thing.

comment:" (#$aimlPred $#aimlvar-var 'aimlval' ) means that The AIML Variable
#$aimlvar-var is associated with it value 'aimlval' ".


Constant:aimlTopic.

isa:BinaryPredicate.

arg1Isa:Thing.

arg2Isa:Thing.

comment:" (#$aimlTopic $#aimldomain 'aimlval' ) is used to control the <topic> tag in
the aiml interperter. ".


constant:aimlResponse.

isa:BinaryPredicate.

arg1Isa:Thing.

arg2Isa:Thing.

comment:" (#$aimlResponse $#key 'reply') response to ?REPLY for <cycrandom> testing".


constant:response1.

aimlResponse:"response A".

aimlResponse:"response B".

aimlResponse:"response C".

aimlResponse:"response D".
```

APPENDIX E — AIMLCONTEXTMT SNAPSHOT

AIMLContext Snapshot:

```
Microtheory : AimlContextMt


Bookkeeping Assertions :


(myCreator AimlContextMt Kino) in BookkeepingMt

(myCreationPurpose AimlContextMt OpenCycProject) in BookkeepingMt

(myCreationTime AimlContextMt 20031211) in BookkeepingMt

(myCreationSecond AimlContextMt 222945) in BookkeepingMt


GAF Arg : 1


Mt : UniversalVocabularyMt

isa :  DataMicrotheory

genlMt :  NameStringDefinitionMt BaseKB

comment :  "AimlContextMt contains storage location in OpenCyc for AIML variables"
```

Microtheory Contents:

```
(aimlResponse response1 "response D")

(aimlResponse response1 "response C")

(aimlResponse response1 "response B")

(aimlResponse response1 "response A")

(comment aimlPred " (aimlPred $#aimlvar-var 'aimlval' ) means that The AIML Variable
#$aimlvar-var is associated with it value 'aimlval' ")

(comment aimlResponse " (aimlResponse $#key 'reply') response to ?REPLY for
<cycrandom> testing")

(argIsa aimlResponse 2 Thing)

(arg2Isa aimlResponse Thing)

(argIsa aimlResponse 1 Thing)

(arg1Isa aimlResponse Thing)

(aimlPred aimlVar-cycvar "test5")
```

```
(aimlPred aimlVar-cycvax "Hello from CycVax Var value thingy")

(comment aimlTopic " (aimlTopic $#aimldomain 'aimlval' ) is used to control the
<topic> tag in the aiml interperter. ")

(argIsa aimlTopic 2 Thing)

(arg2Isa aimlTopic Thing)

(argIsa aimlTopic 1 Thing)

(arg1Isa aimlTopic Thing)

(comment aimlPred " (aimlPred $#aimlvar-var 'aimlval' ) means that The AIML Variable
#$aimlvar-var is associated with it value 'aimlval' ")

(argIsa aimlPred 2 Thing)

(arg2Isa aimlPred Thing)

(argIsa aimlPred 1 Thing)

(arg1Isa aimlPred Thing)
```