CSE 663 Term Project

Contextual Vocabulary Acquisition using OpenCyc

Project Report

December 19, 2002

Submitted by: ANUROOPA SHENOY

Abstract: *The project is to enable SNePS to link to Cyc and make use of the information stored in Cyc. Cyc can be used as a source of the general background information needed for doing Contextual vocabulary acquisition.*

Introduction:

Contextual vocabulary acquisition (CVA) is active, deliberate acquisition of word meanings from text by reasoning about using contextual cues, background knowledge, and hypotheses developed from prior encounters with the word, but without external sources of help such as dictionaries or people. [1]

OpenCyc is a general knowledge base and commonsense reasoning engine. It can be considered as a large collection of "commonsense" information and reasoning rules. [2]

A software agent called "Cassie" has been developed as the CVA system. Cassie consists of the SNePS-2.6 semantic-network knowledge-representation and reasoning (KRR) system and a knowledge base (KB) of background information representing the knowledge that a reader (e.g., Cassie) bring to the text containing the unknown term. Currently, the KB is hand-coded. [3]

The goal of this project is to automate the formation of the background information by linking SNePS to Cyc and forming the knowledge base using the information present in Cyc.

Contextual Vocabulary Acquisition (CVA):

The computational natural language processing systems should be robust. They should not break down when they encounter a word or expression that they do not know about. Also they should not stop every time they encounter such a situation and ask a human user for instructions. Also these systems should not rely on a fixed lexicon. They should be able to find out the meaning of the new words or expressions they encounter by them. This is the computational significance of Contextual Vocabulary Acquisition.

Cassie's input consists of the information from the text being read, which is parsed and incorporated in the knowledge representation formalism (in SNePS). Each node in the SNePS network represents a concept or mental object, linked by labeled arcs. All information, including propositions, is represented by nodes, and propositions about propositions can be represented without limit. Arcs form the underlying syntactic structure of SNePS. Paths if arcs can be defined allowing for path based inference, including property inheritance within generalization hierarchies. There is a one to one correspondence between nodes and represented concepts. This uniqueness principle guarantees that nodes will be shared whenever possible and that nodes represent intensional objects, i.e., concepts, propositions, properties, and such objects of thought as fictional entities non existents, and impossible objects. This representational ability is appropriate for CVA from arbitrary texts, whose subject matter could range from factual science to science fiction. [3]

The output of Cassie consists of a report of Cassie's current definition of the word in its context. The meaning of the word is taken as the position of that word in a single, highly interconnected network of words, propositions, and other concepts, consisting of the reader's background knowledge integrated with his or her mental model of the text being read. The word's dictionary definition usually contains less information that that. Hence algorithms are used for hypothesizing a definition by deductively searching the network for information appropriate to a dictionary like definitions.

SNeRE:

SNeRE, The SNePS Rational Engine, is a package that allows for the smooth incorporation of acting into SNePS-based agents. SNeRE recognizes a node with an action arc to be a special kind of node called an *act node*. Since an act usually consists of an action and one or more objects of the action, an act node usually has additional arcs pointing to the nodes that represent the objects of the action.

OpenCyc:

CycL is Cyc's language for expressing common sense knowledge. CycL is a formal language whose syntax derives from first-order predicate calculus (the language of formal logic) and from Lisp. In order to express common sense knowledge, however, it goes far beyond first order logic. The vocabulary of CycL consists of terms. The set of terms can be divided into constants, non-atomic terms (NATs i.e. functions), variables, and a few other types of objects. Terms are combined into meaningful CycL expressions, which are used to make assertions in the Cyc knowledge base. Every CycL atomic formula must begin with a predicate in order to be well-formed. [2]

In the Cyc KB, a truth value is a value attached to an assertion which indicates its degree of truth. There are five possible truth values:

- monotonically true (100)

  True always and under all conditions. This is normally reserved for things that are true by definition.

- default true (T)

  The assertion is assumed true, but subject to exceptions. Most rules in the KB are default true.

- unknown (~)

  Not known to be true, and not known to be false.

- default false (F)

  The assertion is assumed false, but subject to exceptions.

- monotonically false (0)

  The assertion is false always and under all conditions.

A microtheory is a Cyc constant denoting assertions which are grouped together because they share a set of assumptions. Microtheories are also called contexts. Every assertion is contained in some microtheory. A particular formula may be asserted into (or concluded in) more than one microtheory; when this is the case, there will be an assertion which has that formula in each of those microtheories.

One of the functions of microtheories is to separate assertions into consistent bundles. Within a microtheory, the assertions must be mutually consistent. This means that no hard contradictions are allowed, and any apparent contradictions must be resolvable by evaluation of the evidence visible in that microtheory. In contrast, there may be inconsistencies *across* microtheories.

CycL:

A collection is a type of thing, a kind of thing, or a class of things. Things which belong to a collection are called its *instances*. To express that something is an instance of a collection in CycL, the predicate #$isa is used. Everything belongs to at least one collection. To express that one collection is subsumed by another, we use the CycL constant #$genls. A formula of the form (#$genls X Y) means that every instance of the first collection, X, is also an instance of the second collection, Y. The #$genls relationship is transitive. Also, #$isa transfers through #$genls.

#$arity denotes the number of arguments that a predicate must have. The #$argxIsa predicate imposes semantic constraints. It constrains the meanings of the terms that are legal arguments for that predicate.

CycL also provides the logical connective #$or, #$and, #$not, #$implies and the quantifiers #$forall and #$thereexists.

These concepts are fundamental to knowledge representation in CycL.

Connecting to Cyc

Cyc has implemented two protocols to connect to their server; the first one is an incomplete telnet-like protocol. The second is a complete binary protocol, which has been currently implemented by java classes. The Cyc server provides API services by binding two TCP ports and accepting TCP connections at those ports. The default installation installs API servers accepting at ports 3601 (ASCII) and 3614 (CFASL). The actual port numbers used can be specified at installation time.

Refer to http://www.opencyc.org/doc/cycapi [4] for detailed information.

1. ASCII API

1.1 TCP Port Security

The Cyc server does not provide secure TCP connections. Applications using Cyc should reside within the fire walled network or on the same host as Cyc. But, remote access to Cyc should be performed using secure shell (SSH) to forward the ports Cyc uses, over an encrypted channel.

1.2 Establishing a connection

A client application establishes an API connection to a Cyc server by opening a TCP connection to one of the API ports on the machine running the Cyc server. The socket established is used to communicate API messages to Cyc and read results from Cyc.

1.3 Server Algorithm

Once the socket to be used for communication is established, the Cyc API server goes into a loop performing the following steps in order until the connection is closed:

- input one API request from the socket
- evaluate the request
- output the result to the socket

The protocol of each step is described below for the ASCII communication mode.

1.4 ASCII Message protocols

All messages used to input an API request and output an API result are ASCII text messages. Input messages are sent from the client to the socket and at the Cyc server read from its input stream for the socket. Output messages are sent from the Cyc server to its output stream for the socket and are read from the socket by the client application.

The Cyc server flushes its output stream for the socket after each output message is sent in order to ensure the client application can read a complete output message without blocking.

The client application is similarly reminded to flush its socket after each input message is sent in order to ensure the Cyc server can read a complete input message without blocking.

1.4.1 Input message protocol

The default input message protocol is an ASCII text message which can be read in from the socket using the SubL "read" protocol. More precisely, the SubL function READ is called on the input stream for the socket in order to produce a SubL form. This form represents the request to be evaluated.

Each API request is thus a textual sequence of the form

(<API function> <arg1> ... <argN>) <whitespace>

For example:

(constant-id #$Thing)

The arguments to an API request may themselves be API requests. For example:

(constant-id (find-constant "Thing"))

1.4.2 Evaluating an API request

The API request read in is evaluated according to the SubL "eval" protocol. More precisely, the SubL function EVAL is called on the form in order to produce a single SubL result. This result is outputted using the output message protocol. If a function returns multiple values, only the first value is used, however the SubL function MULTIPLE-VALUE-LIST can be used to gather multiple values into a single list.

If an error condition occurs during evaluation, the evaluation is aborted and a string representing the error condition is used as the result of the evaluation.

1.4.3 Output message protocol

The default API output message protocol consists of two parts. First, a code is output indicating whether or not the API request succeeded or generated an error.

The code output is either the textual sequence

200 <whitespace>

For a successful evaluation, or the sequence

500 <whitespace>

if an error occurred during evaluation. This code can be used by the client application to interpret the second part of the output protocol.

Second, the result of the API request is output according to the SubL "print" protocol. More precisely, the SubL function PRINT is called on the result of the evaluation.

Thus, the result will be a textual sequence of the form

<API result> <whitespace>

1.5 Closing a connection

A client application closes an API connection to a Cyc server by simply executing the API request

(api-quit) <whitespace>

This will cause the Cyc server to halt the connection and close the socket.

1.6 Multithreading

As in Cyc's multithreaded implementation, each API connection spawns a separate thread which is completely dedicated to handling the API server connection. When the connection is broken, the thread exits. Future API connections will reflect the modifications made to Cyc KB via the API server.

1.7 Useful API requests

The Cyc API has been implemented in a language called SubL. Some of the API requests useful in the CVA project are:

1.7.1 Constants

function **`CONSTANT-P`** `: (object)`
Return T iff the argument is a CycL constant.
Single value returned satisfies BOOLEANP.

function **`FIND-CONSTANT`** `: (name)`
Return the constant with NAME, or NIL if not present.
NAME must satisfy STRINGP.
Single value returned satisfies CONSTANT-P or is NIL.

function **`CONSTANT-COMPLETE`** `: (prefix &optional case-sensitive? exact-length? start end)`
Return all valid constants with PREFIX as a prefix of their name.
When CASE-SENSITIVE? is non-nil, the comparison is done in a case-sensitive fashion.
When EXACT-LENGTH? is non-nil, the prefix must be the entire string. Optionally the START and END character positions can be specified, such that the PREFIX matches characters between the START and END range. If no constant exists, return NIL.
PREFIX must satisfy STRINGP.
CASE-SENSITIVE? must satisfy BOOLEANP.

EXACT-LENGTH? must satisfy BOOLEANP.
START must satisfy FIXNUMP.

function **CONSTANT-APROPOS** : (substring &optional case-sensitive? start end)
Return all valid constants with SUBSTRING somewhere in their name.
When CASE-SENSITIVE? is non-nil, the comparison is done in a case-sensitive fashion.
Optionally the START and END character positions can be specified, such that the SUBSTRING matches characters between the START and END range. If no constant exists, return NIL.
SUBSTRING must satisfy STRINGP.
CASE-SENSITIVE? must satisfy BOOLEANP.
START must satisfy FIXNUMP.

1.7.2 isa support

function **MIN-ISA** : (term &optional mt tv)
Returns most-specific collections that include TERM (inexpensive).
TERM must satisfy HL-TERM-P.
Single value returned is a list of elements satisfying FORT-P.

function **MAX-NOT-ISA** : (term &optional mt tv)
Returns most-general collections that do not include TERM (expensive).
TERM must satisfy HL-TERM-P.
Single value returned is a list of elements satisfying FORT-P.

function **ALL-ISA** : (term &optional mt tv)
Returns all collections that include TERM (inexpensive).
TERM must satisfy HL-TERM-P.
Single value returned is a list of elements satisfying FORT-P.

function **UNION-ALL-ISA** : (terms &optional mt tv)
Returns all collections that include any term in TERMS (inexpensive).
TERMS must satisfy LISTP.
Single value returned is a list of elements satisfying FORT-P.

function **ALL-NOT-ISA** : (term &optional mt tv)
Returns all collections that do not include TERM (expensive).
TERM must satisfy HL-TERM-P.
Single value returned is a list of elements satisfying FORT-P.

function **ALL-INSTANCES** : (col &optional mt tv)
Returns all instances of COLLECTION (expensive).
COL must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

function **MAP-ALL-ISA** : (fn term &optional mt tv)
Apply FUNCTION to every all-isa of TERM.
(FUNCTION must not effect the current sbhl search state)
FN must satisfy FUNCTION-SPEC-P.
TERM must satisfy HL-TERM-P.

function **ANY-WRT-ALL-ISA** : (function term &optional mt tv)
Return the first encountered non-nil result of applying FUNCTION to the all-isa of TERM.
(FUNCTION may not effect the current sbhl search state)
FUNCTION must satisfy FUNCTION-SPEC-P.
TERM must satisfy HL-TERM-P.

function **MAP-ALL-INSTANCES** : (fn col &optional mt tv)
Apply FUNCTION to each unique instance of all specs of COLLECTION.
FN must satisfy FUNCTION-SPEC-P.
COL must satisfy EL-FORT-P.

function **MAP-INSTANCES** : (function term &optional mt tv)
apply FUNCTION to every (least general) #$isa of TERM.
FUNCTION must satisfy FUNCTION-SPEC-P.
TERM must satisfy EL-FORT-P.

function **ISA?** : (term collection &optional mt tv)
Returns whether TERM is an instance of COLLECTION.
COLLECTION must satisfy EL-FORT-P.
Single value returned satisfies BOOLEANP.

function **ISA-ANY?** : (term collections &optional mt tv)
Returns whether TERM is an instance of any collection in COLLECTIONS.
TERM must satisfy HL-TERM-P.
COLLECTIONS must satisfy LISTP.
Single value returned satisfies BOOLEANP.

function **ANY-ISA-ANY?** : (terms collections &optional mt tv)
Returns booleanp; whether any term in TERMS is an instance of any collection in COLLECTIONS.
TERMS must satisfy LISTP.
COLLECTIONS must satisfy LISTP.
Single value returned satisfies BOOLEANP.

function **NOT-ISA?** : (term collection &optional mt tv)
Returns boolenap; whether TERM is known to not be an instance of COLLECTION.
TERM must satisfy HL-TERM-P.
COLLECTION must satisfy EL-FORT-P.
Single value returned satisfies BOOLEANP.

function **INSTANCES?** : (collection &optional mt tv)
Returns whether COLLECTION has any direct instances.
COLLECTION must satisfy EL-FORT-P.
Single value returned satisfies BOOLEANP.

function **INSTANCES** : (col &optional mt tv)
Returns the asserted instances of COL.
COL must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

function **ISA-SIBLINGS** : (term &optional mt tv)
Returns the direct isas of those collections of which TERM is a direct instance.
TERM must satisfy HL-TERM-P.
Single value returned is a list of elements satisfying FORT-P.

function **MAX-FLOOR-MTS-OF-ISA-PATHS** : (term collection &optional tv)
Returns in what (most-genl) mts TERM is an instance of COLLECTION.
TERM must satisfy HL-TERM-P.
COLLECTION must satisfy EL-FORT-P.

function **WHY-ISA?** : (term collection &optional mt tv behavior)
Returns justification of (isa TERM COLLECTION)
TERM must satisfy HL-TERM-P.
COLLECTION must satisfy EL-FORT-P.
Single value returned satisfies LISTP.

function **WHY-NOT-ISA?** : (term collection &optional mt tv behavior)
Returns justification of (not (isa TERM COLLECTION)).
TERM must satisfy HL-TERM-P.
COLLECTION must satisfy EL-FORT-P.
Single value returned satisfies LISTP.

function **ALL-INSTANCES-AMONG** : (col terms &optional mt tv)
Returns those elements of TERMS that include COL as an all-isa.
COL must satisfy HL-TERM-P.
TERMS must satisfy LISTP.
Single value returned is a list of elements satisfying FORT-P.

function **ALL-ISA-AMONG** : (term collections &optional mt tv)
Returns those elements of COLLECTIONS that include TERM as an all-instance.
TERM must satisfy HL-TERM-P.
COLLECTIONS must satisfy LISTP.
Single value returned is a list of elements satisfying FORT-P.

function **ALL-ISAS-WRT** : (term isa &optional mt tv)

Returns all isa of term TERM that are also instances of collection ISA (ascending transitive closure; inexpensive).
TERM must satisfy EL-FORT-P.
ISA must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

function **`INSTANCE-SIBLINGS`** `: (term &optional mt tv)`
Returns the direct instances of those collections having direct isa TERM.
TERM must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

1.7.2 genls support

function **`MIN-GENLS`** `: (col &optional mt tv)`
Returns the most-specific genls of collection COL.
Single value returned is a list of elements satisfying FORT-P.

function **`MAX-NOT-GENLS`** `: (col &optional mt tv)`
Returns the least-specific negated genls of collection COL.
COL must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

function **`MAX-SPECS`** `: (col &optional mt tv)`
Returns the least-specific specs of collection COL.
COL must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

function **`MIN-NOT-SPECS`** `: (col &optional mt tv)`
Returns the most-specific negated specs of collection COL.
COL must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

function **`GENL-SIBLINGS`** `: (col &optional mt tv)`
Returns the direct genls of those direct spec collections of COL.
COL must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

function **`SPEC-SIBLINGS`** `: (col &optional mt tv)`
Returns the direct specs of those direct genls collections of COL.
COL must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

function **`ALL-GENLS`** `: (col &optional mt tv)`
Returns all genls of collection COL (ascending transitive closure; inexpensive).
COL must satisfy EL-FORT-P.
Single value returned is a list of elements satisfying FORT-P.

1.7.4 Assertion support

function **ASSERTION-MENTIONS-TERM** : (assertion term)
Return T iff Assertion's formula or mt contains TERM.
If assertion is a meta-assertion, recurse down sub-assertions. By convention, negated gafs do not necessarily mention the term #$not.
ASSERTION must satisfy ASSERTION-P.
TERM must satisfy HL-TERM-P.
Single value returned satisfies BOOLEANP.

1.7.5 Querying

function **CYC-QUERY** : (sentence &optional mt properties)
Query for bindings for free variables which will satisfy SENTENCE within MT.
Properties: :backchain NIL or an integer or T
:number NIL or an integer
:time NIL or an integer
:depth NIL or an integer
:conditional-sentence boolean
If :backchain is NIL, no backchaining is performed. If :backchain is an integer, then at most that many backchaining steps using rules are performed. If :backchain is T, then inference is performed without limit on the number of backchaining steps when searching for bindings.
If :number is an integer, then at most that number of bindings are returned.
If :time is an integer, then at most that many seconds are consumed by the search for bindings.
If :depth is an integer, then the inference paths are limited to that number of total steps.
Returns NIL if the operation had an error. Otherwise returns a (possibly empty) binding set. In the case where the SENTENCE has no free variables, the form (NIL), the empty binding set is returned, indicating that the gaf is either directly asserted in the KB, or that it can be derived via rules in the KB. If it fails to be proven, NIL will be returned. The second return value indicates the reason why the query halted.
If SENTENCE is an implication, or an ist wrapped around an implication, and the :conditional-sentence property is non-nil, cyc-query will attempt to prove SENTENCE by reductio ad absurdum.
SENTENCE must satisfy POSSIBLY-SENTENCE-P.
MT must satisfy (NIL-OR HLMT-P).
PROPERTIES must satisfy QUERY-PROPERTIES-P.
Single value returned satisfies QUERY-RESULTS-P.

function **CYC-CONTINUE-QUERY** : (&optional query-id properties)
Continues a query started by xref cyc-query.
If QUERY-ID is :last, the most recent query is continued.
QUERY-ID must satisfy QUERY-ID-P.

PROPERTIES must satisfy QUERY-PROPERTIES-P.
Single value returned satisfies QUERY-RESULTS-P.

1.7.6 Using the requests

1. **find-constant request**

   (find-constant “Fido”)

2. **all-isa request**

   (all-isa #$Dog)

3. **all-genls request**

   (all-genls #$Dog #$BiologyMt)

4. **cyc-query**

   (cyc-query '(#$genls #$Dog ?Col) #$BiologyMt)

1.7.7 An lisp program to connect to Cyc

```
;; Author: Anuroopa Shenoy
;; Utilities
(defun send-line (stream line)
  "Send a line of text to the stream, terminating it with
CR+LF."
  (princ line stream)
  (princ #\Return stream)
  (princ #\Newline stream)
  (force-output stream))
(defun read-response (stream)
  ;; Read response and output it
  (format t "> Received response:~%")
  (loop
    (setq line (read-line socket nil nil))
    (unless line (return))
    (format t "~a~%" line)
    (return)))
;;; Client
```

```
(defun connectCyc ()
  ;;Send an api request to the Cyc server on localhost at
ASCII port 3601
  ;;Print the contents of the returned answer to standard
output."
  ;; Open connection
(setq server "localhost")
   (setq port 3601)
   (setq socket (SOCKET:SOCKET-CONNECT port server))
   (unwind-protect
       (progn
    (format t "> Sending connect request to Cyc at ~a:~a...~%"
server port)
    ;; Send request
    ;;(send-line socket "(fi-find \"Dog\")")  --  using
deprecated api reques
    (send-line socket "(find-constant \"Dog\")")
    (read-response socket)
    ;;(send-line socket "(fi-ask '(#$isa #$Dog ?COL))") --
using deprecated api request
    (send-line socket "(all-isa #$Dog)")
    (read-response socket)
    ;;(send-line socket "(fi-ask '(#$genls #$Dog ?COL))") --
using deprecated api request
    (send-line socket "(all-genls #$Dog #$BiologyMt)")
    (read-response socket)
    (send-line socket "(api-quit)")
   ))
     ;; Close socket before exiting.
    (close socket))
```

2. Binary API

The binary protocol, named CFASL for which the reference client implementation has been given by CfaslInputStream and CfaslOutputStream java classes.

Refer to http://www.cyc.com/doc/opencyc_api/java_api/ [4] for more information.

2. 1 Useful Java API methods

2.1.1 Constants

getKnownConstantByName

public CycConstant **getKnownConstantByName**(java.lang.String constantName)
throws java.io.IOException, java.net.UnknownHostException, CycApiException
Gets a known CycConstant by using its constant name.
Parameters:
`constantName` - the name of the constant to be instantiated
Returns:
the complete `CycConstant` if found, otherwise throw an exception

getConstantByName

public CycConstant **getConstantByName**(java.lang.String constantName)
throws java.io.IOException, java.net.UnknownHostException, CycApiException
Gets a CycConstant by using its constant name.
Parameters:
`constantName` - the name of the constant to be instantiated
Returns:
the complete `CycConstant` if found, otherwise return null

2.1.2 isa support

getIsas

public CycList **getIsas**(CycFort cycFort)
throws java.io.IOException, java.net.UnknownHostException, CycApiException
Gets a list of the isas for a CycFort.

getMinIsas

public CycList **getMinIsas**(CycFort cycFort)
throws java.io.IOException, java.net.UnknownHostException, CycApiException
Gets a list of the most specific collections (having no subsets) which contain a CycFort term.

getAllIsa

public CycList **getAllIsa**(CycFort cycFort)

throws java.io.IOException, java.net.UnknownHostException, CycApiException
Gets a list of the collections of which the CycFort is directly and indirectly an instance.

2.1.3 genls support

getGenls

public CycList **getGenls**(CycFort cycFort)
throws java.io.IOException, java.net.UnknownHostException, CycApiException
Gets a list of the directly asserted true genls for a CycFort collection.

getMinGenls

public CycList **getMinGenls**(CycFort cycFort)
throws java.io.IOException, java.net.UnknownHostException, CycApiException
Gets a list of the minimum (most specific) genls for a CycFort collection

getAllGenls

public CycList **getAllGenls**(CycFort cycFort)
throws java.io.IOException, java.net.UnknownHostException, CycApiException
Gets a list of all of the direct and indirect genls for a CycFort collection.

2.1.4 Assertions

askWithVariable

public CycList **askWithVariable**(CycList query, CycVariable variable, CycFort mt)
throws java.io.IOException, java.net.UnknownHostException, CycApiException

Returns a list of bindings for a query with a single unbound variable.
Parameters:
`query` - the query to be asked in the knowledge base
`variable` - the single unbound variable in the query for which bindings are sought
`mt` - the microtheory in which the query is asked
Returns:
a list of bindings for the query

askWithVariables

public CycList **askWithVariables**(CycList query, java.util.ArrayList variables, CycFort mt)

throws java.io.IOException, java.net.UnknownHostException, CycApiException

Returns a list of bindings for a query with unbound variables.
Parameters:
`query` - the query to be asked in the knowledge base
`variables` - the list of unbound variables in the query for which bindings are sought
`mt` - the microtheory in which the query is asked
Returns:
a list of bindings for the query

2.2 Using Java API methods

Make new access object for connection to Cyc Server
CycAccess cycAccess = new CycAccess();

Searching for a constant Fido
CycFort fido = cycAccess.getKnownConstantByName("Fido");

Getting all the collections for which BillClinton is an instance of.
CycList isas = cycAccess.getIsas(cycAccess.getKnownConstantByName("BillClinton"));

Getting all the collections for which Dog is a subcollection (subclass) of.
CycList genls = cycAccess.getGenls(cycAccess.getKnownConstantByName("Dog"));

2.2 A Java Program to connect to Cyc

A working demo is available as a java class ApiDemo.java in the src/org/opencyc/api directory in the home directory where Cyc is installed. However, to run this, there is a complied version in lib directory of home directory of Cyc. It relies on jakarta-oro-2.0.6 to make the demo work (which I have installed on the Plato machine).

Conversion from Cyc to SNePS:

The information in a microtheory in Cyc KB can be put in a context in SNePS. The #$isa predicate can be converted to a member – class frame in SNePS. The #$genls can be represented as a subclass – superclass frame in SNePS. In this case the transitivity of the #$genls has to be specified in SNePS by paths so that path-based inference denotes the

transitivity. The hierarchy among the microtheories in Cyc can be represented in SNePS as the context hierarchy.

A listing of all the case-frames currently handled by CVA and the predicate(s) in Cyc that can be used to get the information needed for the case-frame out of Cyc is:

Case-frame Agent – Act		
Predicate	**Use**	**Explanation**
Actors	(#$actors EVENT ACTOR)	ACTOR is somehow saliently (directly or indirectly) involved in EVENT during EVENT
preActors	(#$preActors EVENT OBJECT)	OBJECT is a participant (see #$actors) in EVENT and that OBJECT exists before EVENT begins
postActors	(#$postActors EVENT OBJECT)	OBJECT is a participant in EVENT (so that (#$actors EVENT OBJECT) also holds), and that OBJECT exists after EVENT ends
deliberateActors	(#$deliberateActors ACT ACTOR)	#$Agent ACTOR is conscious, volitional, and purposeful in the #$Event ACT
nonDeliberateActors	(#$nonDeliberateActors ACT ACTOR)	ACTOR plays some role in the #$Event ACT, but is not acting deliberately
doneBy	(#$doneBy EVENT DOER)	DOER is the doer in the event EVENT: some activity on the part of DOER causes or carries out EVENT
performedBy	(#$performedBy ACT DOER)	DOER deliberately does ACT
performedByPart	(#$performedByPart ACT ORG)	ORG is considered to be the performer of the #$Action ACT, though in fact only some subordinate part of ORG (i.e., a member or a sub-organization), rather than all of the organization, is directly involved in ACT
bodilyDoer	(#$bodilyDoer EVENT DOER)	DOER does EVENT (i.e., DOER is not merely subjected to EVENT by external forces), but DOER does EVENT non-deliberately
actorPartsInvolved	(#$actorPartsInvolved ACT PART)	PART is one of the parts (see the predicate #$anatomicalParts) of an organism that is playing an active role (see the predicate #$doneBy and its specializations) in the event ACT, and, moreover, that PART is somehow involved in the event ACT
levelOfMentalExertion	(#$levelOfMentalExertion AGNT ACT LEVEL)	AGNT performs ACT, exerting a level of mental effort described by LEVEL

levelOfPhysicalExertion	(#$levelOfPhysicalExertion AGNT ACT LEVEL)	AGNT does ACT with a level of physical exertion LEVEL

Special case of case-frame Agent – Act		
Predicate	**Use**	**Explanation**
directingAgent	(#$directingAgent EVENT DIRECTOR)	DIRECTOR deliberately controls or directs EVENT. AGENT might or might not perform EVENT directly

Predicates related to the case-frame Agent – Act		
Predicate	**Use**	**Explanation**
skillLevel	(#$skillLevel OBJ ACT-TYPE ROLE PERF-ATT LEVEL)	OBJ has the ability to play the role ROLE in instances of the type of #$Event ACT-TYPE, with LEVEL degree of PERF-ATT
skillRequired	(#$skillRequired ACT-TYPE OTHER-TYPE PERF-ATT LEVEL)	one is to successfully perform an instance of some kind of action (ACT-TYPE), then one must be capable of performing instances of another kind of action (OTHER-TYPE) with the performance attribute PERF-ATT at a level of at least LEVEL
performanceLevel	(#$performanceLevel OBJ EVT ROLE PERF-ATT LEVEL)	individual OBJ plays the role ROLE in the action EVT, and does so with the performance attribute PERF-ATT to the degree LEVEL
significantSubEvents	(#$significantSubEvents RESULT-TYPE COMPONENT-TYPE RESULT-ATTRIB COMPONENT-ATTRIB SIGNIF)	doing an action of type COMPONENT-TYPE with a #$HighToVeryHigh level of the #$ScriptPerformanceAttribute COMPONENT-ATTRIB contributes to performing an action of RESULT-TYPE with a #$HighToVeryHigh level of the performance attribute RESULT-ATTRIB

Case-frame Act – Object		
Predicate	**Use**	**Explanation**
objectActedOn	(#$objectActedOn EVENT OBJECT)	OBJECT is altered or affected in EVENT, and the change that OBJECT undergoes is central or focal to understanding EVENT
objectOfStateChange	(#$objectOfStateChange EVENT OBJECT)	#$PartiallyTangible OBJECT undergoes some kind of intrinsic change of state (such as a change in one of its physical properties) in the

		#$IntrinsicStateChangeEvent EVENT
bodilyActedOn	(#$bodilyActedOn EVENT ORG)	ORG is a living organism (i.e., an #$Organism-Whole) that is being affected in EVENT
deviceUsed	(#$deviceUsed EVENT OBJECT)	#$PhysicalDevice OBJECT plays an instrumental role in the #$Event EVENT
objectControlled	(#$objectControlled EVENT OBJ)	object OBJ is being controlled in the #$Event EVENT
stuffUsed	(#$stuffUsed EVENT STUFF)	STUFF is a portion of an instance of #$ExistingStuffType which plays an instrumental role in EVENT

Case-frame Object – Possesser		
Predicate	**Use**	**Explanation**
possesses	(#$possesses AGENT OBJECT)	OBJECT is in the physical possession of AGENT
owns	(#$owns AGENT OBJECT)	AGENT has full ownership of OBJECT

Case-frame Object – Property		
Predicate	**Use**	**Explanation**
hasAttributes	(#$hasAttributes THING ATT)	ATT characterizes THING
genlAttributes	(#$genlAttributes SPEC-ATT GENL-ATT)	SPEC-ATT generalizes to GENL-ATT in the sense that anything that possesses the former attribute possesses the latter as well
negationAttribute	(#$negationAttribute ATT1 ATT2)	nothing has, or could have, both ATT1 and ATT2 as attributes at the same time
oppositeAttributes	(#$oppositeAttributes ATTR1 ATTR2)	ATTR1 is the directly opposite #$AttributeValue of ATTR2 (and vice versa).

Case-frame Member – Class		
Predicate	**Use**	**Explanation**
isa	(#$isa THING COL)	THING is an instance of the collection COL

memberOfSpecies	(#$memberOfSpecies ORG SPECIES)	organism ORG is a member of the #$BiologicalSpecies SPECIES
hasGender	(#$hasGender ORGANISM SEX)	sex of ORGANISM is SEX

Case-frame Superclass – Subclass		
Predicate	**Use**	**Explanation**
genls	(#$genls SUBCOL SUPERCOL)	SUPERCOL is a supercollection of SUBCOL

Predicates related to case-frame Antonym – Antonym		
Predicate	**Use**	**Explanation**
different	(#$different THING1..THINGn)	for any THINGi and THINGj (where 0 <= i <= n, 0 <= j <= n, and i =/ j), THINGi is not identical with THINGj

Case-frame Object Location		
Predicate	**Use**	**Explanation**
objectFoundInLocation	(#$objectFoundInLocation OBJ LOC)	OBJ has the location LOC
inRegion	(#$inRegion OBJ1 OBJ2)	OBJ1 is located at or in OBJ2. OBJ1 might or might not be a part (see #$parts) of OBJ2
geographicalSubRegions	(#$geographicalSubRegions SUPER SUB)	SUPER and SUB are both elements of #$GeographicalRegion, and the area SUB lies wholly within the region SUPER (see #$inRegion).
onPath	(#$onPath THING PATH)	THING is located along (on or adjacent to) the #$Path-Generic PATH. THING could be a moving object, or it could be a stationary point

Predicates related to case-frame Object – Location		
Predicate	**Use**	**Explanation**
perpendicularObjects	(#$perpendicularObjects OBJ1 OBJ2)	the longest axis of OBJ1 is perpendicular to the longest axis of OBJ2

parallelObjects	(#$parallelObjects OBJ1 OBJ2)	the lengthwise axes of OBJ1 and OBJ2 are parallel to each other
inFrontOf-Generally	(#$inFrontOf-Generally FORE AFT)	FORE is in front of the tangible object AFT
inFrontOf-Directly	(#$inFrontOf-Directly FORE AFT)	FORE is directly in front of tangible object AFT.
behind-Generally	(#$behind-Generally AFT FORE)	AFT is behind FORE
behind-Directly	(#$behind-Directly AFT FORE)	AFT is directly behind tangible object FORE.
above-Directly	(#$above-Directly ABOVE BELOW)	(1) the volumetric center of ABOVE is directly above some point of BELOW, if ABOVE is smaller than BELOW; or that (2) some point of ABOVE is directly above the volumetric center of BELOW, if ABOVE is larger than, or equal in size to, BELOW
above-Touching	(#$above-Touching ABOVE BELOW)	ABOVE is located over BELOW and they are touching
above-Overhead	(#$above-Overhead ABOVE BELOW)	ABOVE is directly above BELOW (see the predicate #$above-Directly), all points of ABOVE are higher than all points of BELOW, and ABOVE and BELOW do _not_ touch.
above-Higher	(#$above-Higher OBJ-A OBJ-B)	OBJ-A is at a greater altitude (from some common reference point) than OBJ-B
above-Generally	(#$above-Generally OBJ1 OBJ2)	#$SpatialThing-Localized OBJ1 is more or less above the #$SpatialThing-Localized OBJ2
surroundsCompletely	(#$surroundsCompletely OUTSIDE INSIDE)	OUTSIDE completely surrounds INSIDE
surroundsHorizontally	(#$surroundsHorizontally OUTSIDE INSIDE)	OUTSIDE surrounds a horizontal cross-section of INSIDE
near	(#$near THIS THAT)	distance between THIS and THAT is such that -- given the situation at hand and the sorts of things that THIS and THAT are -- they would be considered near each other by most observers
adjacentTo	(#$adjacentTo OBJ1 OBJ2)	OBJ1 and OBJ2 are touching, and that their region of contact is (at least for practical purposes, relative to the objects' dimensions and shapes) a line (i.e. the contact region is not a point, though the line of contact might actually have some height)
touches	(#$touches THIS THAT)	THIS and THAT are in contact, either directly or

		indirectly
touchesDirectly	(#$touchesDirectly THIS THAT)	THIS and THAT are in direct physical contact
on-Physical	(#$on-Physical OVER UNDER)	object OVER is above, supported by, and touching the object UNDER
supportedBy	(#$supportedBy OBJECT SUPPORT)	SUPPORT is at least partially responsible for holding OBJECT up and maintaining its vertical position

Case-frame Part – Whole		
Predicate	**Use**	**Explanation**
parts	(#$parts WHOLE PART)	PART is in some sense a part of WHOLE
physicalDecompositions	(#$physicalDecompositions WHOLE PART)	PART is a spatial part or component of WHOLE, in a very broad sense of `part' whereby PART might or might not be spatially continuous or discrete
physicalParts	(#$physicalParts WHOLE PART)	WHOLE is an at least partially tangible object and PART is one of its distinct, non-diffuse, identifiable, partially tangible parts
anatomicalParts	(#$anatomicalParts ORGANISM PART)	#$OrganismPart PART is an anatomical part of the #$Organism-Whole ORGANISM
physicalPortions	(#$physicalPortions WHOLE PART)	PART is a representative physical part of WHOLE, in the sense that the intrinsic physical properties of WHOLE are also properties of PART
physicalExtent	(#$physicalExtent WHOLE PART)	PART is the complete part of WHOLE
intangibleComponent	(#$intangibleComponent WHOLE PART)	PART is the entire intangible part of the #$CompositeTangibleAndIntangibleObject WHOLE
surfaceParts	(#$surfaceParts BIG LITTLE)	LITTLE is an external physical part (see #$externalParts) of a surface of BIG, or that LITTLE is a physical part of BIG itself and a surface of LITTLE is part of a surface of BIG.
externalParts	(#$externalParts OBJ PART)	OBJ has PART as one of its external #$physicalParts

References:

[1] Contextual Vocabulary Acquisition: description, http://www.cse.buffalo.edu/~rapaport/CVA/cvadescription.html

[2] www.opencyc.org/

[3] Rapaport, William J., & Ehrlich, Karen (2000), "A Computational Theory of Vocabulary Acquisition", in Łucja Iwanska, & Stuart C. Shapiro (eds.), Natural Language Processing and Knowledge Representation: Language for Knowledge and Knowledge for Language (Menlo Park, CA/Cambridge, MA: AAAI Press/MIT Press): 347-375.

[4] www.opencyc.org/doc/cycapi

[5] www.opencyc.org/doc/javaapi

[6] http://www.cyc.com/cycdoc/vocab/vocab-toc.html