KnowOS: The (Re)Birth of the Knowledge Operating System

Mike Travers, JP Massar, and Jeff Shrager[1]

We describe a Lisp-based Knowledge Operating System (KnowOS) that supports several real applications, including BioLingua, a knowledge-rich biocomputing workbench, and CACHE, a tool for collaborative information analysis. The KnowOS is essentially a web-based Lisp environment, providing apparent persistence of complex data and processes, internal and external application integration, multi-user management, and user interface tools for dealing with networks of complex objects. We explain the KnowOS vision, how we approached it, the pros and cons of what we did, how the implementation stands up to the vision, and where we are going with this project.

The KnowOS Vision and Abstract Architecture

Both operating systems and programming languages use abstraction to provide services such as data management, user and process isolation and intercommunication, and interfaces to hardware services and to other software. In historical practice, the broad concepts of operating systems and programming languages overlap with one another in a variety of ways. For example, since one of the most important services provided by operating systems is application integration – the ability to pass the results of one application program on to another – operating systems often provide custom "scripting" languages for this purpose [1]. Deriving from IBM's Job Control Language (JCL) in the 60s, the concept was popularized by Unix shell scripts, and has evolved a number of rich languages such as Python.

Both scripting languages and full-strength programming languages define a virtual environment consisting of abstract objects and operations to manipulate them. They differ primarily in the nature of their domain of objects, and in their sense of the persistence of their data state. Scripting languages generally range over persistent object spaces composed of relatively simple objects. The objects in Unix are files organized into directories, and these are highly persistent. Similarly, relational database management systems (RDBMSs), such as Oracle, range over persistent tables. Contrast this with classical programming where what one does is to build up networks of complex structured "intermediate" knowledge, use these in calculations, and then drop them when the program terminates.

It is now widely recognized that networks of complex structured knowledge are critical to modern software applications, but the tools contained in mainstream languages and operating systems are not particularly oriented towards this type of data. Operating systems deal mainly with serial files, programming languages deal with networks of objects but do not provide persistence. RDBMs provide persistence but are poor at dealing with complex, evolving , messily-structured domains. Herein lies the vision that

[1] Corresponding author: JShrager@Stanford.edu; The Carnegie Inst. of Washington, Dept. of Plant Biology, 260 Panama St., Stanford, CA, 94306.

we call a “Knowledge Operating System” or “KnowOS”: Complex persistent knowledge should be the fundamental domain of both the user interface (UI) and of the way that programs interact with one another (i.e., the API).

In what follows we describe an open source KnowOS designed to support the development of heavily knowledge-based applications. We begin by describing the KnowOS as we envisioned it, what it is for, and then how we approached each major component. For each of these we examine the pros and cons of our implementation choices. Finally, we assess how the implementation stands up to the vision, and talk about where we are going with the KnowOS project.

The central idea of a knowledge operating system is that knowledge – that is, arbitrarily complex, interconnected, semi-structured data of all types – is omnipresent. This knowledge is immediately and easily available to both users and programs at all times, and the user is, or appears to be, “living in” this universe, just as the user of Unix lives in the directory tree, and the user of Oracle lives in a world of tables. All of this knowledge should be manipulable through an equally-omnipresent programming language through which one can easily and efficiently work with complex knowledge structures. In this vision, everything known to the OS is a part of the knowledge space – basic data, programs, process objects, long text objects (analogous to files), etc., and the programming language has access to all of these objects equally. The classical example of what we have thus far described is the Lisp Machine [2], variously instantiated at MIT and Xerox PARC in the 70s and 80s.

In addition to enabling one to work and program in a rich knowledge world, a true KnowOS would provide all of the “standard” computational tools that one needed for one’s particular domain of computation. In our initial domain of computational genomics, these include tools such as sequence comparison and alignment, the formation of phylogenetic trees, and a wide variety of other programs. Our goal in building a KnowOS was to smoothly integrate these and any other tools desired by users into the knowledge space, all under the control of a simple and efficient programming paradigm.

Our KnowOS is divided into three major architectural layers: (1) An apparently-persistent knowledge base centered around a frame system, flexible enough to represent complex knowledge domains; (2) An efficient, highly extensible, yet easy to use programming language (Lisp, with extensions for frames and domain specific functions) that manages the operation of the whole system, permitting both scripting and real programming; and (3) A web-based interface to the programming language, and thence to the knowledge and other KnowOS services. These architectural layers are implemented on top of standard hardware and software (e.g., Linux) so that most existing standard programs, such as the bioinformatic tools described above, can be fluently integrated into one’s work.

Knowledge-based Scientific Computing by the Scientists Themselves

The concept of a Knowledge Operating System came upon us rather than us coming upon it. In 2002, in collaboration with NASA's Computational and Fundamental Biology division, we set out to build a biologist's programming environment, which eventually became BioLingua [3]. First made public in mid 2003, the fundamental insight in BioLingua was that biologists and other scientists needed to be knowledge programmers.

As an example, a common computational operation in modern biology is to analyze and interpret the data coming from DNA microarray experiments. DNA microarrays indirectly measure the relative expression level of every gene in an organism under various conditions (such as different nutrient concentrations). The mathematical analyses of this data involve relatively simple statistical calculations. However, the *interpretation* of these results is a heavily knowledge-based operation. A very small organism might have a few thousand genes, and the amount of knowledge about each might vary from none to detailed knowledge about the systems in which each gene participates and the roles that it plays in these systems, and about what other genes each is similar to, in this or other organisms. This knowledge is highly diverse, often uncertain, and never simple. Interpreting a microarray experiment thus involves seeing the simple statistical results through the lens of the rich knowledge of the organism and its genes.

Biologists know that they need to do this, and they know more-or-less what they want to do, yet they have essentially no tools with which to do it. Biologists very often try to do this sort of thing with their most powerful tool: Excel. Indeed, they usually give up at this sort of analysis and hire undergraduate programmers to write one-off analysis programs. In order to enable biologists to do work of this sort themselves, the energy barrier, so to speak, standing between biologists and complex knowledge-based programming needed to be greatly reduced. Many factors contribute to the height of this barrier, but the primary one is the need for integrated manipulation of a great many different kinds of knowledge from many different sources. The work required to gather this knowledge into a common computational framework is beyond the time and knowledge resources of most biologists.

BioLingua – the first instantiation of our Knowledge Operating System – was intended to meet this challenge by providing a powerful programming language living in a pre-loaded knowledge system; In just a few lines of code, in just a few minutes, biologists who knew BioLingua could get exploratory knowledge work done that would otherwise take even a knowledgeable programmer days or weeks, and which would be essentially impossible using tools such as Excel. BioLingua grew somewhat organically, as do most large software systems programmed by a very small group. Yet the fundamental vision was always clear: by surrounding scientists with the relevant knowledge and tools, and a simple means of manipulating these through programming, we could greatly facilitate scientific computing. Of course, reality is more complex. In what follows we describe our concrete implementation choices in terms of the KnowOS knowledge model and interface technology, and assess how well these implementations stand up to the KnowOS vision.

The Omnipresence of Complex Knowledge

As we have said, the most fundamental component of our vision is that users have direct, tangible access to knowledge and that it be represented in a very flexible manner. This focus led us to several important choices. First, we chose to build our own frame-based knowledge representation system. We did this not because we couldn't have pulled one of the shelf, but because the frame system is so central to the KnowOS vision that we wanted to have control over all of its specific features. For the sake of integrating language and interface, we wanted the frames to be accessible by name. For the sake of power we wanted to be able to associate arbitrary functions with slot types and to be able to control inheritance very simply and flexibly in order to make it easy to write custom inference methods. And we wanted to make it easy to eventually back the knowledge base with a persistence system. Also, at the time that we built the preliminary frame system we had in mind writing a Lisp tutorial book, so part of the spec was that the frame system should be simple enough to be explained in detail as a chapter of such a text.

Since frame names in our system are guaranteed to be unique, we needed to impose a structured naming scheme. In practice, we have taken to creating frame names that are prefixed with the domain in which the frame lives, like a required package prefix in Common Lisp, an email address, or java class name. For example, in BioLingua, the Gene Ontology frames are all prefixed with "GO.", and the frames that come from the BioCyc models (which are cast in the Ocelot frame language) are all: "OC." [4]. There are two completely different frames, one called "GO.GLUCOKINASE" and another called "OC.GLUCOKINASE" and these coexist happily together. Users are free, of course, to produce their own naming and lookup conventions, and a general frame search mechanism is provided that enables one to look up frames by any test on the contents of any (or all) slots.

The idea of data persistence in programming languages is ancient [5], perhaps its earliest implementation being APL [6] workspaces in the 60s. Unfortunately, persistence contains a fundamental tension: although is seems obvious that one wants to retain the state of one's data, and possibly one's processes, this enables one to do permanent damage to important data. In the KnowOS we have, at the moment, implemented what we call "apparent persistence" (AP). That is, everything persists until the server crashes or is taken down. At reboot, the knowledge base is reloaded into Lisp memory automatically. Because the KnowOS UI is merely a thin client to a Lisp-based server (discussed in more detail below), and because of ACL's excellent stability, the KnowOS *appears* to be a persistent knowledge store where users' states of work, as well as all their processes, persist from one "login" to the next, because to log into the system is merely to connect to the already-running server. The quasi-persistent based knowledge – knowledge of the organisms and various basic biological knowledge, in the case of BioLingua – is provided by preloads that come from MySQL and/or flat files, and which are loaded either at system boot, or on demand. Some of the most complex knowledge is loaded "just in time" by the frame system from its sources when it is referenced.

So, persistence in our current KnowOS is illusory – data and process state are only maintained as long as the server does not reboot. (The same is true for process state in any operating system, with the exception of certain research projects, notably CoyotOS; www.coyotos.org) Regardless, we recognize that we need more than Apparent Persistence for some applications, and are moving toward delivering it through an integration with AllegroCache, a new Franz object database. AllegroCache has a commit-based architecture, which raises some difficult problems in our current model where a single Lisp image serves multiple users running multiple threads. There arises the issue of how to avoid committing everyone's content at the same time when one user does a commit. One approach to this involves differentiating temporary from permanent frames; Although all are committed, the frames marked as temporary are deleted later from the persistent store in what amounts to a garbage collection phase. Another solution that has been proposed is for each user to have a separate connection to the store within the same Lisp image, or for each user to be running an entirely separate Lisp image in the server, all talking to a central AllegroCache server. Once a marriage has been made between the KnowOS and AllegroCache, many other issues will arise. Most interesting among these, we expect the KnowOS to be able to provide the equivalent of versioning for knowledge – a sort of CVS for frames. Exactly how this will be done is under discussion, but it is an important facility for biologists, and for many other domains.

In sum, the KnowOS knowledge model, as it stands, is simple to use, and efficiently carries out most of our needs, yet has significant limitations. We recognize these, esp. apparent v. true persistence, but it is not so simple to make this work well from the user's point of view.

Web-Based Interface and Multi-User Management

The central user interface of our current KnowOS is a web-based Lisp Listener and associated dynamic web pages, which users interact with through a standard web browser (Fig. 1). The choice to use standard "thin" browser technology rather than building a complex "thick" GUI interface has significant pros and cons. The most important pro is accessibility; users have access to the full power of a Lisp-based environment without a complex installation process, and independent of any particular software platform. Another advantage is ease of use and development. Using dynamic HTML generation and associated technologies, it is easy to generate a fairly powerful user interface and adapt it to new uses. However, the resulting UI is somewhat less rich in functionality than that provided by a thick-client GUI. We have made some forays into extending the interaction capabilities of the browser, for instance using JavaScript running in the browser to provide rudimentary argument lists and paren balancing, but these are difficult and are so far just in their infancy.

Through judicious extensions to Lisp functionality, and through a complex set of dynamic web pages, we have recreated through the browser what amounts to a programming environment in which one can create and test functions, gather them together, and save them out as modules into the KnowOS's own captive part of the underlying Linux file system. Users can go back and forth between their code in the editor and the KnowOS listener, and conduct editing, evaluation, and exploration of the knowledge space all at the same time. Modules containing complexes of data and functionality can be composed and shared, and KnowOS users can even create their own simple dynamic web pages which call in to user-written functions.

Apparent time sharing is managed by ACL's process system, and we use packages to separate user's data. Each user is given a unique package in which to create his or her objects. This works surprisingly well and provides opportunities for information sharing that we have taken advantage of in collaborative tasks. Of course it also permits any user's state to be examined and modified by any other user. We have consistently made the tradeoff toward functionality over security in the current KnowOS.

Describing a UI can never come close to experiencing it, so we encourage the reader to go to www.biolingua.org where a fully-operational demo version of the BioLingua instance of the KnowOS is available for you to explore. Some interesting features of the UI are: two independent input regions, and an evaluation history; frame objects that are clickable (a small step towards the goal of everything being live); session logs that save all activity for later examination; two or more users being able to log in as a single user and completely share their session. Experienced Lisp users will quickly note that there is no interactive debugging facility; this rarely seems to be a hindrance given the type of programming done using the interface. (Fig. 1 is a snapshot of the main parts of the UI.)

Code Base and Installations

The KnowOS is an open source project under the "MIT Open Source" license, and we encourage the Lisp community to contribute to it. It is built nearly entirely in Common Lisp. Although we have focused on ACL 7.0, we have remained very close to the Common Lisp specification, and most of the KnowOS runs in Lispworks and SBCL.

As discussed above, the KnowOS was originally developed for knowledge-based biocomputing, and appeared first in the guise of BioLingua [3]. In BioLingua the knowledge bases loaded into the KnowOS include the Gene Ontology, and various organism-specific genomic knowledge, including pathway models from the KEGG and BioCyc knowledge bases [4]. BioLingua instances have been developed for a number of model organisms, including a collection of 13 cyanobacteria, and *Arabidopsis thaliana*, the standard model plant. There are now three BioLingua severs running at Stanford, and three at Virginia Commonwealth University under the auspices of Dr. Jeff Elhai who uses BioLingua to teach Bioinformatics to high school, undergraduate, and graduate students.

The KnowOS is also being used in a research application called CACHE (the "Collaborative Analysis of Competing Hypotheses Environment"), wherein the biological knowledge base is replaced by a commonsense knowledge base (SUMO – the Suggested Upper Merged Ontology, www.ontologyportal.org). CACHE takes special advantage of the client server model combined with knowledge integration to provide a distributed, collaborative information analysis platform. The CACHE architecture facilitates hypothesis sharing between groups of analysts by mapping simple decision tables into an influence network where hypotheses are supported by a whole network of underlying evidence and other hypotheses. Because of the rich knowledge-based and knowledge-base integration supported by the KnowOS and SUMO, researchers who are working on similar problems can find one another even if they are using different vocabularies to describe their analyses.

There are a number of proposals under consideration for KnowOS applications. A National Genome Technology Center has proposed to build a KnowOS cluster as the underlying engine for several large knowledge-based biocomputing projects. Another interesting proposal is that the KnowOS form the base of a biological model store and analysis platform called "HyBrow"[7]. Similar to CACHE, HyBrow will enable a community of biologists to develop complex models that are grounded in knowledge and experimentation, while retaining the flexibility to respond to changing concepts and new information. The KnowOS is attractive in these applications because of its facility with symbolic computing, and its simple web-based multi-user client-server architecture.

Directions

Although the KnowOS is currently employed by a number of users to get real work and real teaching done, there remains enormous room for improvement. As mentioned above, we are working to extend Apparent Persistence to at least a partially true persistence model by integrating the KnowOS with AllegroCache. This will open up numerous opportunities for experiments in knowledge versioning and other issues that will become possible and important under a true persistence model. Most importantly, our current design assumes benign users as it implements no security. It is extremely difficult to secure a true "Lisp all the way down" architecture. One approach to improved security and user-based persistence is for each user to have their own private Lisp image running on the server, and to access a shared persistent object database (i.e., the AllegroCache server). This would significantly simplify security at the cost of users being able to easily interact with one another's workspaces, which is unfortunately exactly what makes the KnowOS attractive to some communities.

We are always seeking ways to reduce the difficultly of complex programming in the KnowOS (i.e., in Lisp). There is significant room for creative work regarding basic debugging in a web-based programming system, and we have implemented a few "toy" programming aids. The most promising directions in this area are in graphical programming through a Behave!-like interface [8], which allows Lisp-like expressions to be assembled graphically. Another promising direction is in quasi-natural-language "programming by resolution", a collaboration with researchers at SRI who have done this in the Gemini question-answering system [9]. Here NL descriptions are translated into theorems which are then "proved" by the SNARK theorem prover [10]. These theorems constitute chunks of program. Aside from helping users query the complex knowledge space, we would like to use this to support extraction of content from XML structures that have not previously been encountered. This is a very common problem in the KnowOS paradigm where users access various external programs through interfaces that contain complex XML.

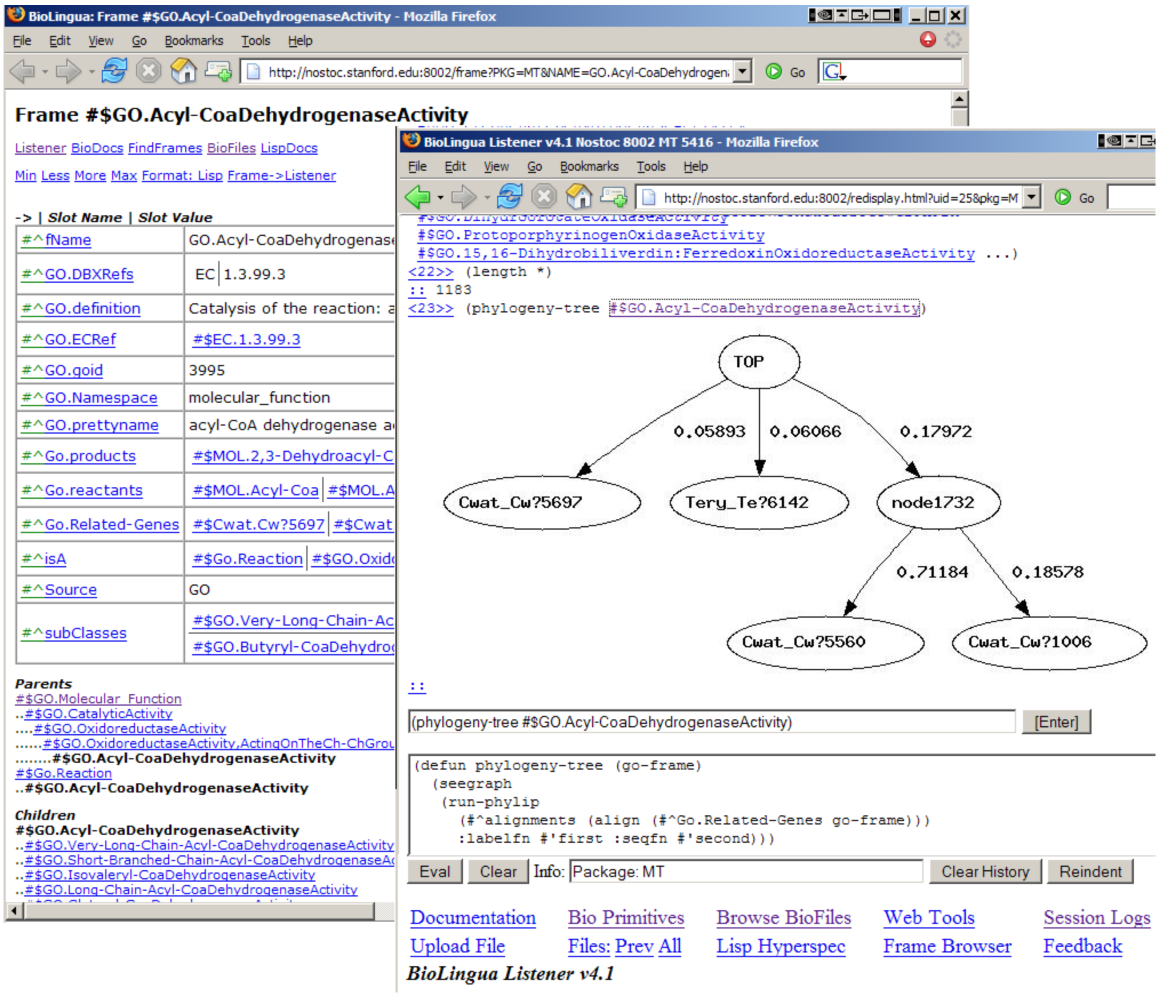
A Tool for Thought in the Age of Computational Knowledge

The goal of the KnowOS is to provide *tools for thought* that enable scientists, engineers, and other analysts to conveniently utilize a variety of representational and computational methods to investigate their domains. To do this we have done what amounts to turning a programming language into an operating system by endowing it with (apparent) persistence and multi-user capabilities, with the ability to integrate external applications, and with simple web-accessibility. As a result the KnowOS provides both some of the desirable characteristics of an operating system, and at the same time, those of a true programming language, and we are in process of integrating high-level knowledge tools, such as theorem-proving and quasi-natural-language programming capabilities. The KnowOS is neither fish nor fowl – neither merely a persistent object system nor merely a lisp-based scripting language; instead, it is something nearly novel: A Knowledge Operating System – perhaps the next stage in systems evolution in this age of computational knowledge.

Acknowledgements
We especially thank Andrew Pohorille of NASA's Division of Astrobiology and Fundamental Biology, who provided seed support for BioLingua. Others who have contributed significantly include Jeff Elhai, Bob Haxo, Pat Langley, Marc Santoro and Mark Slupesky. We thank Franz, Inc. and Lispworks for contributing their products and support, and Edi Weitz for CL-PPCRE.

References

1. JK Ousterhout (1998) Scripting: Higher-Level programming for the 21st Century. IEEE Computer, 31(3).
2. GL Steele, Jr., DA Moon (May, 1978) CADR. Artificial Intelligence Memo\Working Paper unnumbered; Massachusetts Institute of Technology, Artificial Intelligence Laboratory.
3. JP Massar, M Travers, J Elhai, J Shrager (2005) BioLingua: A programmable knowledge environment for biologists. BioInformatics 21(2), 199-207.
4. PD Karp, S Paley, P Romero (2002) The Pathway Tools Software. Bioinformatics 18, S225-S232.
5. MP Atkinson, PJ Bailey, KJ Chisholm, WP Cockshotr, R Morrison (1983) An approach to persistent programming. Comput. J. 26(4).
6. IBM (August, 1968) APL\360 User's Manual. IBM TJ Watson Research Center.
7. N Fedoroff, S Racunas, J Shrager (2005) Tools for Thought in the Age of Biological Knowledge. The Scientist.
8. M Travers (1998) Behave!, A visual programming system for the Virtual Fishtank exhibit, The Computer Museum, Boston.
9. R Waldinger, DE Appelt, , J Fry, DJ Israel, P Jarvis, D Martin, S Riehemann, ME Stickel, M Tyson, J Hobbs, JL Dungan, (2004) Deductive question answering from multiple resources. In MT Maybury (Ed.) New Directions in Question Answering. AAAI Press.
10. ME Stickel, RJ Waldinger, VK Chaudhri (2000) A guide to SNARK. Technical report, SRI International,


BioLingua: Frame #$GO.Acyl-CoaDehydrogenaseActivity - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://nostoc.stanford.edu:8002/frame?PKG=MT&NAME=GO.Acyl-CoaDehydrogen
Go
BioLingua Listener v4.1 Nostoc 8002 MT 5416 - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://nostoc.stanford.edu:8002/redisplay.html?uid=25&pkg=M
Go
TOP
0.05893
0.06066
0.17972
Cwat_Cw?5697
Tery_Te?6142
node1732
0.71184
0.18578
Cwat_Cw?5560
Cwat_Cw?1006
[Enter]
Eval
Clear
Clear History
Reindent

Frame #$GO.Acyl-CoaDehydrogenaseActivity

Listener BioDocs FindFrames BioFiles LispDocs

Min Less More Max Format: Lisp Frame->Listener

-> | *Slot Name* | *Slot Value*

#^fName	GO.Acyl-CoaDehydrogenas
#^GO.DBXRefs	EC | 1.3.99.3
#^GO.definition	Catalysis of the reaction: a
#^GO.ECRef	#$EC.1.3.99.3
#^GO.goid	3995
#^GO.Namespace	molecular_function
#^GO.prettyname	acyl-CoA dehydrogenase a
#^Go.products	#$MOL.2,3-Dehydroacyl-C
#^Go.reactants	#$MOL.Acyl-Coa \| #$MOL.A
#^Go.Related-Genes	#$Cwat.Cw?5697 \| #$Cwat
#^isA	#$Go.Reaction \| #$GO.Oxid
#^Source	GO
#^subClasses	#$GO.Very-Long-Chain-Ac #$GO.Butyryl-CoaDehydro

Parents
#$GO.Molecular_Function
..#$GO.CatalyticActivity
....#$GO.OxidoreductaseActivity
......#$GO.OxidoreductaseActivity,ActingOnTheCh-ChGrou
........**#$GO.Acyl-CoaDehydrogenaseActivity**
#$Go.Reaction
..**#$GO.Acyl-CoaDehydrogenaseActivity**

Children
#$GO.Acyl-CoaDehydrogenaseActivity
..#$GO.Very-Long-Chain-Acyl-CoaDehydrogenaseActivity
..#$GO.Short-Branched-Chain-Acyl-CoaDehydrogenaseAc
..#$GO.Isovaleryl-CoaDehydrogenaseActivity
..#$GO.Long-Chain-Acyl-CoaDehydrogenaseActivity

```
 #$GO.ProtoporphyrinogenOxidaseActivity
 #$GO.15,16-Dihydrobiliverdin:FerredoxinOxidoreductaseActivity ...)
<22>> (length *)
:: 1183
<23>> (phylogeny-tree #$GO.Acyl-CoaDehydrogenaseActivity)
```

```
::
```

(phylogeny-tree #$GO.Acyl-CoaDehydrogenaseActivity)

```
(defun phylogeny-tree (go-frame)
  (seegraph
   (run-phylip
     (#^alignments (align (#^Go.Related-Genes go-frame)))
     :labelfn #'first :seqfn #'second)))
```

Info: Package: MT

Documentation Bio Primitives Browse BioFiles Web Tools Session Logs
Upload File Files: Prev All Lisp Hyperspec Frame Browser Feedback

BioLingua Listener v4.1

Figure 1. Examples of the two most important pages from the KnowOS user web interface.

The front page illustrates the Lisp listener, with frame hyperlinks and the ability to embed computed graphics. Notice the two listener text entry fields, with "Enter" v. "Eval" buttons. Short expressions can be evaluated in the "Enter" field, and longer expressions (e.g., function definitions) can be evaluated in the multi-line "Eval" field. Paren balancing, dynamic argument display, etc. take place in the small text box labeled "Info". Evaluation history scrolls upward, so that the latest expression evaluated and its result are displayed nearest the bottom of the history.

The occluded page illustrates the frame browser displaying a Gene Ontology concept, and including its place in the GO hierarchy (bottom of the page). Notice that nearly everything is clickable, usually opening another frame browser. In this way the knowledge base can be conveniently explored. The left hand column of the frame table displays the slot names; Each is a frame that can be examined. The slot values are in the right hand column (partially occluded), and can also be examined if they contain frames, structures, or CLOS objects.