

# Polyome: A Learning System for Extracting Bioinformatic Data

Bob Myers<sup>1,2</sup>, Trevor I Dix<sup>1,2</sup>, Ross L Coppel<sup>1,3</sup> and David G Green<sup>1,2</sup>

<sup>1</sup>Victorian Bioinformatics Consortium, Monash University, Australia

<sup>2</sup>Clayton School of Information Technology, Monash University, Melbourne, Australia

<sup>3</sup>Dept. of Microbiology, Monash University, Melbourne, Australia

*Abstract: The exponential growth in the quantity of publicly available genetic data and the proliferation of bioinformatic databases mean that scientists need computerized tools more than ever. Existing approaches to the problem all suffer from one or more basic problems. This paper describes Polyome, the core of a system for the integration and querying of data sources, designed to overcome these problems.*

Keywords: Bioinformatics, Data Extraction, Data Integration.

## 1. INTRODUCTION

The integration and querying of multiple and diverse data sources is a major problem in many areas, but particularly so in the bioinformatics domain. Galperin[1] lists 858 publicly available bioinformatics data sources in his 2006 review, an increase of 19% over 2005 and 56% over 2004. Also, the quantity of genetic data produced each year has been growing exponentially. The International Nucleotide Sequence Database Collaboration announced in August 2005 that their databases now contained over 100 Gigabases ( $10^{11}$ )[2]. An inevitable effect of the bioinformatics data explosion is a proliferation of data sources of many types. It is possible that the molecular biology world will eventually decide on a common standard (probably using a ‘survival of the fittest’ technique), but that is at best still a long way off, and something is needed to fill the gap in the mean time. Researchers need computerized tools to assist them in making best use of an exponentially expanding body of information.

Some approaches that have been used to tackle this problem include Data Warehousing, Standardized

Databases, Federated Databases, Web-Services, Database Wrappers and the Semantic Web. These all have various restrictions and problems as discussed in [3-7] and many others. Some of the main problems encountered with these approaches are:

- *Fragility*, where even small changes in a data source can cause the system to malfunction. For example, a fieldname in a relational database is changed or the layout of a web-page is updated. This can lead to an unreasonably high maintenance load.
- *Breadth*, where each new data source needs to be individually programmed into the system in detail. Considering the number of bioinformatic data sources, this leads to an unacceptably large effort if the interface to each data source has to be constructed from scratch.
- *Complexity*, where creating components of the system requires much expertise in biology and computing. If the skills required are rare, interfaces are unlikely to be created for many data sources.

Other systems that tackle the problem of data integration include:

- BioQuery[8], which uses database wrappers.
  - myGrid[9], uses grid technology.
  - TAMBIS[10], uses wrappers
  - Discovery Net[11], uses grid technology,
  - ISYMOD[6], uses data warehouse concepts.
  - GUS[12, 13], GMOD[14] and AceDB[15], are attempts at building standardized databases
  - K2[13]
- and many others.

This paper describes Polyome, the core of a system for the integration and querying of data sources, designed to overcome these problems. It does this by the use of metadata to describe data sources and a con-

versational process to seek clarification from the user if necessary and make corrections to the metadata. The reuse of lower-level building-blocks and the conversational process allow the construction of new functionality from existing components, making accessing new data sources simple. Integration of data is accomplished by the use of a logical database to store it in a common format.

## 2. METHODS

For a bioinformatics tool to be truly useful it would have to be able to access as many different types of data sources as possible, including:

- Structured data (relational databases, XML etc.)
- Semi-structured like (HTML etc.)
- Unstructured (text, papers etc.)

It would also need to be able to cope with changes in the target data sources, be simple to use and extend its functionality, and must be able to learn to enable it to access new data sources with minimal re-programming.

The approach taken by the authors in designing and building such a system was to create a generalized, simple, low-level ‘parser’, that makes as few assumptions as possible about the form of input it would receive, and to make it extendable by the addition of rules and libraries of low-level functions. It was decided to avoid full natural language processing since a lot of the input would not be natural language.

For the user interface it was decided to use a combination of free text and GUI components. Some interactions between the user and the system are best handled by a GUI component, like an options box, if for example, the system needs to restrict the user into selecting one of a small set of replies. On the other hand, when dealing with many different forms of input it is often impossible to design a GUI that allows adequate flexibility. In these situations the free text interface would be used.

The components of Polyome are written as a Java application. Java was chosen because it is easily portable across different hardware and operating systems, and there a great many ‘add-on’ libraries available to extend Java’s capabilities.

A Prolog logical database stores the basic data and metadata for the system - a natural choice to use for representing many different forms of data in a common format. The public domain Kernel Prolog[16] was chosen because it is written in Java, integrates well with the rest of the system, and supports ‘BuiltIns’ (a method of adding Prolog functions written in Java). The system, in its current form, is described in detail in section 3.

## 3. ARCHITECTURE

### 3.1. Architecture Overview

The Converser consists of three parts:

- **Tokenize:** Receives text input from the user, breaks it into tokens and allocates a token-type tag to each using the dictionary.
- **Interpret:** Matches the tokens against a set of translation rules, producing a number of interpretations of the input.
- **Evaluate:** Checks each interpretation, selects the most suitable, and runs it, passing the result back to the user.

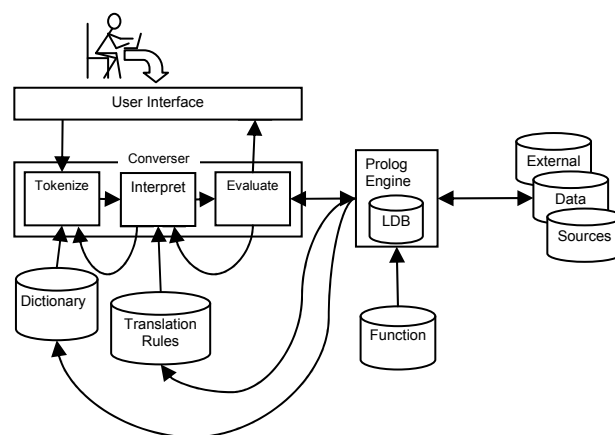


Fig. 1 Architecture Overview

The interpretations contain script language modules (called PScripts) which perform actions on the Prolog Engine, querying and updating the Prolog logical database (LDB), and formulating replies to the user.

The LDB stores data and metadata about the system’s area of expertise.

The Function Library contains a set of Prolog functions to extend the Prolog engine’s functionality in a given area of expertise. Functions may be written in Prolog or Java.

The User Interface handles all interaction between the system and the user via a combination of text and graphical methods.

The flow of a simple interaction between a user and Polyome might be: The user enters a query via the user interface, the Converser uses the dictionary and translation rules to convert it into a PScript, which is run, sending commands to the Prolog engine (which updates the LDB) and sends an appropriate response back to the user.

### 3.2. Components and Metadata

**User Interface:** The Polyome user interface consists of a combination of text and graphical compo-

nents. In the primary interface (see Fig. 2) the user types a statement or query in plain text in the bottom panel, and responses from Polyome appear in the top panel.

Various graphical interface modules are provided for situations where the system needs to constrain the users response to one of several options or to display results as tables of text, images etc. For example, the system can display an option-box as shown in Fig. 3 to force the users reply to one of the two options provided.

Other graphical modules include list-boxes, image-display and html-display.

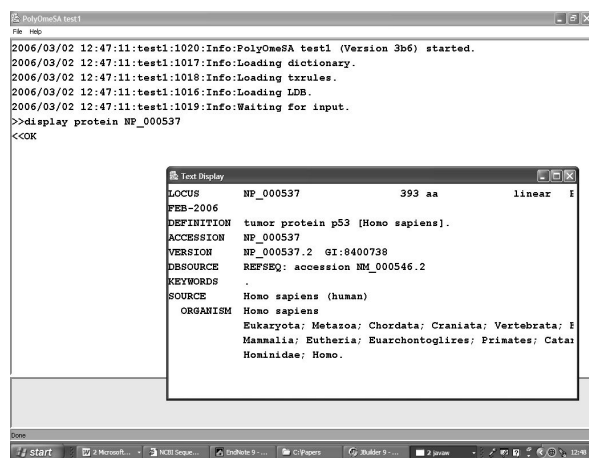


Fig. 2 Text Interface

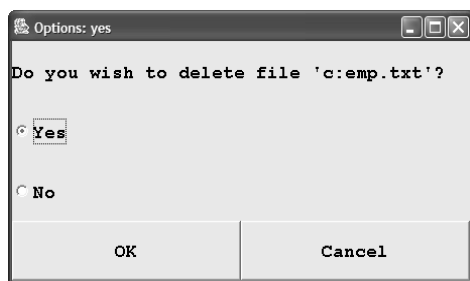


Fig. 3 Options Box Interface

**Dictionary:** The dictionary is effectively a list of pairs with each pair consisting of a token and a tag. A token can be any string of characters, for example: a word, a short phrase, HTML elements etc. A tag can be anything used to categorize the token. For example, tags for English words would generally be parts-of-speech, but for HTML elements you might categorize them simply as 'open', 'close' or 'other'.

It is possible for a token to have more than one tag, e.g. the token 'arc' can be tagged as 'noun' and 'verb', and both can be recorded in the dictionary.

This data is stored as XML and read into a data structure when the system is started. The data struc-

ture can be represented by a graph<sup>1</sup> as shown in Fig. 4 where each node contains a single character and, optionally, a tag.

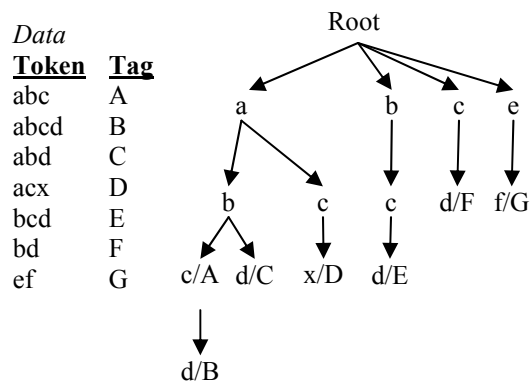


Fig. 4 Dictionary data structure

Updates can be made to the data structure during a session and the data is written back to XML at the end. The dictionary is pre-loaded with over 6000 pairs corresponding to common English words and various HTML elements.

**Tokenize:** The tokenize algorithm finds all pairs that match the beginning of the input string. So, using the data in Fig. 4, for an input of 'abcdef' tokenize would find pairs 'abc/A' and 'abcd/B'. After these pairs have been matched (see Interpret below) the remainder of the input string is tokenized and matched, and so on until all the input is consumed.

**Translation Rules:** Each translation rule consists of two parts: a *template* and an executable *script* (PScript). The template is used to match to the pairs generated by the tokenize routine and is made up of a series of elements, each of which is a token/tag pair. The template allows for repetition of elements or groups of elements, optional elements or groups and 'wild-card' elements that can match any token. For example:

- (abc/A) would match to a single pair
- (abc/A)+ would match to 1 or more instances of that pair
- (abcd/B)(ef/G)? matches the first pair and the second is optional
- (%x%/A) would match a single pair with any token and tag=A (%x% is the wild-card)
- (abc/?) would represent a wild-card tag.

The PScript is a set of executable statements that

<sup>1</sup> Fig. 4 shows a tree. However, cycles are used for repetition, such as [0-9]+ for an integer, thus leading to a cyclic directed graph.

are to be actioned when the pattern has been matched to some input. For example: “Prolog(assert(isa(a,b))” would cause the assert statement to be executed. The PScript may contain wild-card tokens that occur in the pattern. E.g. “Prolog(assert(isa(%x%,b))”. The value in the input that matched the ‘%x%’ in the pattern is substituted into the PScript prior to execution of the PScript.

An important PScript command is Input(..text..). This submits the text to the Converser (see Fig. 1) for processing. By making nested calls to the Converser in this way a PScript can process input from an external data source into the LDB for later querying, and integrate simple translation rules into more complex structures. See Appendix B for an example of a PScript.

The data for the translation rules is stored in XML (see Appendix A for an example) and read into a data structure at startup. The data structure is a graph with each node corresponding to a token/tag pair. The data is copied back to XML (with any changes) when the system closes.

**Interpret:** The interpret algorithm takes pairs generated by the tokenize routine and matches them to nodes in the translation rules data structure working from the root node down. When a match occurs, the string corresponding to the matched pair’s token is stripped off the front of the input, and matching continues down the tree with the remainder of the input. When a complete translation rule is matched (i.e. a node is reached that contains a PScript) that translation rule is appended to the ‘interpretation’ of this input, and pair matching continues from the root node again. An interpretation of that input (i.e. a set of translation rules that translate that series of pairs into a series of PScripts) has been found when all the input has been consumed. It is of course possible (even likely) that multiple pairs can be generated at different points in the input, and that multiple translation rule nodes may match a pair. The interpret algorithm follows the ‘strongest’ match at these points. For example the pair (abcd/A) matches to translation rule node (abcd/A) more strongly than to node (%x%/A), i.e. it is more specific. Having found a strongly matched interpretation, the algorithm then backtracks to find any weaker ones.

**Evaluate:** The evaluate algorithm takes the interpretations generated by Interpret and ranks them according to a simple scoring system that takes into account things like the number of translation rules in the interpretation, any gaps (i.e. parts of the input that did not match any translation rule), the number of pairs used and the strength of the matching. The PScript for each interpretation is also executed to make sure that it produces functionally correct and logical Prolog

statements. After each execution, the effects of the PScript (e.g. updates to the LDB) are removed using a checkpoint/rollback facility that has been implemented on the Prolog Engine.

Having selected the best interpretation the algorithm either executes it, and returns a message (which is generated by the PScript in the interpretation) to the user, or, if some problem is detected with the selected interpretation, a message is formulated requesting clarification which is sent back to the user instead. The user’s reply to this is processed in the same way as before. The original input from the user is then re-processed, with the expectation that the clarification provided by the user will have updated the system in some way so that the original input can now succeed. In this way the system conducts a conversation with the user, seeking more information to enable it to process the input more reliably.

**Prolog Engine and LDB:** The Prolog engine and Logical Database are used to store and query data and metadata about the system and its area of expertise. For example it may contain the following fact to remember how to access a particular database :

```
hasa(genBase,url,'www.genbase.edu.au/query/main')
```

It is also used to store facts related to particular queries input by the user, for example, details about the contents of an HTML page. The contents of the LDB are saved at system closedown and re-loaded at the next startup.

**Function Library:** The Function Library is a set of Prolog functions set up to extend the systems functionality. For example:

- getHtml(url,Html) reads in HTML from a given URL
- addDictEntry(token,tag) adds a pair to the dictionary.
- submit(formID,HTML) submits an HTML form.

## 4. EXAMPLE

### Submitting an amino acid sequence to SignalP

This example shows how a request can be submitted to an external data source (here the SignalP[17] web-site), and the results displayed to the user.

The user enters the request in the user interface:

```
send ASTPGHTIIYEAVCLHNDRTTIP to signalp
```

The system breaks this into a series of pairs, e.g.:

```
(send,verb)
(ASTPGHTIIYEAVCLHNDRTTIP,?)
(to,prep)
(signalp,noun)
```

Note that because some tokens may have multiple tags, there may be several different sets of pairs generated for the input. All alternatives will be interpreted and the best ones used.

As each of these pairs is generated, the Interpret module matches them against the templates of the translation rules. The template of the rule that matches this set of pairs looks like:

```
('send','verb')
('%prot%','????')
('to','prep')
('signalp','noun')
(('with','prep')('%p%','????')('=','????')('%v%','
????')+)?
```

Note that the last line is optional because of the '?' at the end. It is provided to allow the user to specify values for the various radio buttons on the SignalP submission page (see Fig. 5). For example, the default value of the Organism Group set of radio buttons is set to Eukaryotes.

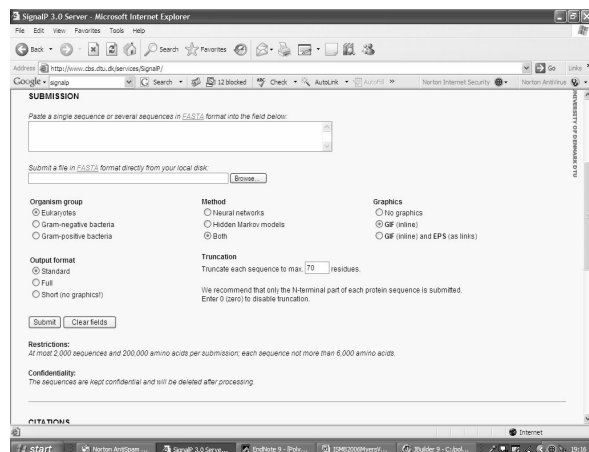


Fig. 5 SignalP submission page

If the user wished to submit a request that specified Gram-negative bacteria, they could enter:

```
send ASTPGHTIIYEAVCLHNDRTTIP to signalp
with orgtype = gram-
```

In our example the optional part does not match any input and hence plays no part. The XML for the full translation rule is listed in Appendix A.

The PScript for this rule performs the following actions:

- 1) Checks to see if the SignalP request page has already been read in. If not, looks at the metadata in the LDB to find the URL of the SignalP request page.
- 2) Runs a Function Library module to read that web-page from the server.

- 3) Treating that HTML as input, it processes it (in the same way as input from the user). In this case the input is HTML, not natural language, and there are entries in the dictionary and translation rules for processing HTML. The contents of the incoming HTML are decomposed into Prolog facts and stored in the LDB.
- 4) The retrieved HTML is validated against the metadata in the LDB.
- 5) If the name of the text-area changes (because the web-page has been updated for example) the PScript returns a message to the user to ask him/her to specify the new name by showing a list of available text-areas on that page. The system then updates the metadata to reflect the new value, and attempts to re-process the original request.
- 6) Changes are made to these facts to reflect the particular request to be sent to SignalP. In this case it is only the contents of the text-area that need to be set to the value of the amino acid string. The system knows the name of this text-area because it is stored in the LDB as metadata.
- 7) The web-page metadata is then used to create a form which is sent to the SignalP server.
- 8) The response from the server is displayed to the user in a separate window.

The code corresponding to these actions is marked: (#n) in the full listing in Appendix B.

The basic process is therefore: The query is processed into a PScript, which performs actions, validating the results against the metadata in the LDB. If a problem is found, the system sends a message to the user, processes the reply and updates the metadata. The original query is then re-processed from scratch. In this way the system copes with changes in the data source (i.e. the Fragility problem) by updating the metadata rather than the PScript.

Above is an example of a query executed through Polyome. It can be viewed as a stand-alone process, or as a component which can be combined with others into larger, more complex modules.

## 5. CONCLUSION

In this paper, Polyome, the core of a system for the integration and querying of bioinformatic data was presented. Polyome has been designed to tackle the three common problems of fragility, breadth and complexity.

Polyome tackles the problem of fragility by the use of metadata (stored in the LDB) to describe key features of the data sources to be accessed. It then uses a conversational process to seek clarification if necessary and make updates to the metadata.

Polyome manages the complexity and breadth problems through the reuse of translation rules and the conversational process which enables the construction of new functionality from existing components.

Integration of data from different sources is accomplished by the use of the Prolog LDB, enabling information from many different databases to be stored in a common format as facts.

Querying the data consists of retrieving facts from the LDB and displaying them using standard, supplied modules.

A pre-loaded library of basic functions is provided, and the system is extendable by the addition of new functions, making it simple to access new types of data sources as they become available.

Efficient data structures and algorithms have been used to maintain good response times in Polyome. Of course there is little that can be done about delays at the data source sites. Finally, the use of Java throughout development of this system makes the Polyome system portable to other hardware platforms and operating systems.

Development of the system is continuing with the addition of more translation rules and function library modules, and further development of the system's functionality.

## 6. ACKNOWLEDGMENT

Bob Myers was supported by the Victorian Bioinformatics Consortium under a Victorian Government STII grant.

## 7. REFERENCES

- [1] Galperin, M.Y., The Molecular Biology Database Collection: 2006 update. Nucl. Acids Res., 2006. 34(suppl\_1): p. D3-5.
- [2] Health, U.N.I.o., Public Collections of DNA and RNA Sequence Reach 100 Gigabases, Aug 2005, [http://www.nlm.nih.gov/news/press\\_releases/dna\\_rna\\_100\\_gig.html](http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html)
- [3] Stein, L., Integrating Biological Databases. 2003.
- [4] Philippi, S., Light-weight integration of molecular biological databases. Bioinformatics, 2004. 20(1): p. 51-57.
- [5] Siepel, A., et al., ISYS: a decentralized, component-based approach to the integration of heterogeneous bioinformatics resources. Bioinformatics, 2001. 17(1): p. 83-94.
- [6] Chabalier, J., et al., ISYMOD: a knowledge warehouse for the identification, assembly and analysis of bacterial integrated systems. Bioinformatics, 2005. 21(7): p. 1246-1256.
- [7] Myers, B., Dix, T.I., Coppel R.L. and Green D.G. Database Integration and Querying in the Bioinformatics Domain. in The ICWE 2005 Workshop on Web Information Systems Modeling. 2005. Sydney, Australia: University of Wollongong School of IT and Computer Science.
- [8] Brundage, J.M. and C. Dubay, BioQuery: an object framework for building queries to biomedical databases, in Bioinformatics. 2003. p. 901-902.
- [9] Stevens, R.D., A.J. Robinson, and C.A. Goble, myGrid: personalised bioinformatics on the information grid, in Bioinformatics. 2003. p. 302i-304.
- [10] Baker, P.G., et al., TAMBIS: Transparent Access to Multiple Bioinformatics Information Sources, in 6th Int. Conf. on Intelligent Systems for Molecular Biology. 1998. p. 25--34.
- [11] Rowe, A., et al., The discovery net system for high throughput bioinformatics. Bioinformatics, 2003. 19(90001): p. 225i-231.
- [12] CBIL and others, The GUS Platform, 2005, <http://www.gusdb.org>
- [13] Davidson, S.B., et al., K2/Kleisli and GUS: experiments in integrated access to genomic data sources. IBM Syst. J., 2001. 40(2): p. 512-531.
- [14] GMOD, Generic Model Organism Database Construction Set, 2005, <http://www.gmod.org/>
- [15] Sanger-Institute, The Sanger Institute: AceDB Database, 2005, <http://www.acedb.org>
- [16] Binnet, Kernel Prolog - Open Source Project, 2006 mar, <http://www.binnetcorp.com/OpenCode/kernelprolog.html>
- [17] Bendtsen JD, N.H., von Heijne G, Brunak S., Improved prediction of signal peptides: SignalP 3.0. J Mol Biol., 2004. 340(4): p. 783-95

## APPENDIX

### A. Example Translation Rule.

```

<txrule>
  <template>
    <pair repeat="1">
      <token>%prot%</token>
      <tag>????</tag>
    </pair>
    <pair repeat="1">
      <token>to</token>
      <tag>prep</tag>
    </pair>
    <pair repeat="1">
      <token>signalp</token>
      <tag>noun</tag>
    </pair>
    <group repeat="?">
      <pair repeat="1">
        <token>with</token>
        <tag>prep</tag>
      </pair>
      <group repeat="+">
        <pair repeat="1">
          <token>%p%</token>
          <tag>????</tag>
        </pair>
        <pair repeat="1">
          <token>=</token>
          <tag>symp</tag>
        </pair>
        <pair repeat="1">
          <token>%v%</token>
          <tag>????</tag>
        </pair>
      </group>
    </group>
  </template>
  <pscript>File(signalp)</pscript>
</txrule>

```

### B. PScript (as contained in file signalp)

```

// check if srp is in LDB, if not then read
// it in by getting the URL from the LDB and
// calling getHTML/2 and feeding the
// retrieved HTML into an Input command

```

```

Branch(
  On(Prolog('hasa(srp,html,X)'))          (#1)
  'no'(Branch(
    On(Prolog('hasa(srp,url,U)'))
    'no'(Return(Class('Message')
      Content('?What is the
        URL of the SignalP
        request page')
      Type('q'))))
    Else(Prolog('hasa(srp,url,U),
      getHTML(U,H),          (#2)
      assert(var(x,H)'))
      Prolog('tid(I),
        push(I),
        assert(hasa(srp,html,I)),
        assert(isa(I,html)'))
      Input(Body(Head(H)      (#3)
        List(
          Prolog('var(x,H)'))))
      Prolog('pop()'))))
  // validate the HTML (in the LDB)
  // make sure it has some <form>s

Branch(                                     (#4)
  On(Prolog('hasa(srp,html,X),
    hasa(X,form,Y)'))
  'no'(Return(Class('Message')
    Content('!That page has no
      forms')
    Type('a'))))

  // put a fact into the LDB to indicate the id
  // of the form that contains a textarea
  // control with type=radio button and the
  // correct name (as stored in the metadata)

Prolog('hasa(srp,html,X),
  hasa(X,form,Y),
  hasa(Y,input,Z),
  hasa(srp,textareaname,Z),
  hasa(Z,type,radio),
  assert(var(form,Y)'))

  // Check that the form has a textarea control
  // with name=format

Branch(                                     (#4)
  On(Prolog('var(form,Y),
    hasa(Y,name,format),
    hasa(Y,type,textarea)'))
  'no'(Return(Class('Message')
    Content('!The format field has
      changed')
    Type('a'))))

  // check that the textarea control has not
  // changed

Branch(                                     (#5)
  On(Prolog('hasa(srp,html,X),
    hasa(X,input,Y),
    hasa(srp,textareaname,Z),
    hasa(Y,name,Z),
    hasa(Y,type,textarea)'))
  'no'(Return(Class('Message')
    Content('!The textarea has
      changed, what is the
      new name')
    Type('q'))))

  // check that any field names supplied by the

// user (eg: 'with orgtype=gram+') are valid

Expand(                                     (#6)
  Branch(
    On(Prolog('var(form,Y),
      hasa(Y,name,%p)'))
    'no'(Return(Class('Message')
      Content('!Do not know field
        %p')
      Type('a'))))

  // check that any field values supplied by
  // the user (eg: 'with orgtype=gram+')
  // are valid and if so, add to LDB

Expand(                                     (#6)
  Branch(
    On(Prolog(
      'var(form,X),
      hasa(X,input,Y),
      hasa(Y,name,%p),
      or(hasa(Y,value,V),
        and(
          not(hasa(Y,type,checkbox)),
          not(hasa(Y,type,radio)))
        ),
      assert(hasa(Y,checked,%v)'))
    'no'(Return(Class('Message')
      Content('%v is not a valid
        option for %p')
      Type('a'))))

  // add the given protein string to the
  // textarea control

Expand(                                     (#6)
  Prolog('var(form,X),
    hasa(X,input,Y),
    hasa(srp,textareaname,Z),
    hasa(Y,name,Z),
    retract(hasa(Y,selected,A)),
    assert(hasa(Y,selected,%prot)'))

  // set the required format for the results
  // page to 'full' (if that's a valid option)

Branch(
  On(Prolog('var(form,X),          (#6)
    hasa(X,input,Y),
    hasa(Y,name,format),
    hasa(Y,value,full),
    retract(hasa(Y,selected,A)),
    assert(hasa(Y,selected,full)'))
  'no'(Return(Class('Message')
    Content('!Full is an invalid
      option for input
      field format.')
    Type('a'))))

  // submit the form get the results page HTML
  // and display it in a separate window

Prolog('var(form,X),
  submit(X,H),          (#7)
  assert(var(y,H)'))

Return(Class('MessageHtmlDisplay')      (#8)
  Content(Body(Head(R)
    List(
      Prolog('var(y,R)'))))
  Type('a'))

```