Lost API documentation

Lost Members

May 11, 2010

Contents

Chapter 1

An introduction to the lost framework

1.1 The lost framework

The lost framework is a collection of utilities and models for working with biological sequences written in PRISM/B-Prolog. The framework tries to unify all these bits and pieces and to provide a way to integrate them.

1.2 Obtaining a copy of the lost framework

The lost framework can be obtained using *git* if you have an account on the *mox* server. Assuming that you have *git* installed on your local machine, to get a copy you need to clone the central repository:

```
$ git clone ssh://your-username@mox.ruc.dk/var/git/lost.git
```

It is also possible to clone the repository directly from mox if you want to work there, rather than on your local machine. To do this, just type

```
$ git clone /var/git/lost.git
```

In the following we will use `$LOST` to refer to top-most directory of your copy of the framework.

1.3 Configuring your copy of the lost framework

After obtaining the lost framework, a little configuration is needed to get started. You will need to edit the file `$LOST/lost.pl`. In the beginning of the file there are two important facts you may need to change,

```
lost_config(prism_command,'prism').
lost_config(lost_base_directory, '/change/to/local/lost/dir/').
lost_config(platform, windows_or_unix).
```

The option `prism_command` should point to a the main PRISM executable binary. If it is in your *$PATH* then you can usually leave it unchanged. `lost_base_directory` should be the full path of the directory (including trailing /) containing the `lost.pl` file. Note, that even on windows platforms you should use forward slash rather than backslash in the path specification. The value of `platform` should be either `windows` or `unix`.

To get started you can examine and run `example.pl` which is located in the in the `$LOST/scripts/` directory.

1.3.1 File structure layout of the lost framework

The lost framework is divided into several parts:

- `$LOST/scripts/`: Contains scripts that run models. These scripts are just small Prolog programs. The scripts does not consult models directly, but instead use predicates that the framework provides.
- `$LOST/models/`: Contains all the biological sequence models. Each model is located in its own subdirectory of `$TLOST/models/` the name of which is also used as name of the model.
- `$LOST/lib/`: Contains shared libraries used multiple models.
- `$LOST/data/`: Contains sequences and data to be used by the models. Annotations etc. generated by model invoked by the framework stored in this directory and given the extension `.gen`
- `$LOST/tmp/`: Is used for temporary storage of files bt models.

Chapter 2

Creating lost models

2.1 Lost model conventions

Each model is located in its own subdirectory of the `$LOST/models/` directory, henceforth called `$MODELS`. So for instance, the sample model called `sample_model1` is located in `$MODELS/sample_model1/`. In the following, we will refer to the directory where a particular model resides as `$MODEL`.

Models are allowed to consult files with paths relative to the `$MODEL` directory, but should under normal circumstances only directly consult file which are located in the `$MODEL` directory or a subdirectory of it. The exception to this is the file `$LOST/lost.pl`. Consulting this file gives access to all the shared APIs (See chapter 3).

To integrate into the framework each model must provide a file called `interface.pl`, which must be located in the same directory as the model. In the file `interface.pl`, one or more of the predefined predicates

- `lost_best_annotation/3`.
- `lost_learn/3`

are implemented and these serve as entry points of using the model.

The general form of these predicates is that they take a list of input files, a list of options and the name of an output file. They generally read the input files, do some processing and then write a result to the output file. The type of processing depends on the predicate. The predicate `lost_best_annotation/3` is meant to be used when running the model in prediction mode, so for instance, it might produce a file containing the viterbi path for a given input sequence. The predicate `lost_learn` is meant to be used when training the model and the output file would usually be the learned parameters of the trained model.

Models should declare the *type* associated to the arguments of these predicates. These declarations include

- The formats of input files, see section **??**.

- The format of the output file, see section **??**.
- Options and default values for options, see section 2.1.2.
- Optionally, a set of possible values any declared option.

2.1.1 Model interface predicates

This section describes predicates, that when implemented in the `interface.pl` file provided by a model, allows the model provide functionalities that integrate into the general framework.

`lost_best_annotation(+InputFileNames,+Options,+OutputFilename)`

The framework calls this predicate to obtain a "best annotation" from the model. The model is free to provide this annotation in any way it sees fit. It is the responsibility of the model to save the annotation to `OutputFilename`, when `lost_best_annotation` is called. If the model for some reason is unable to write the annotation to `OutputFilename`, then it should `throw` an exception. The predicate takes the following arguments:

- `InputFileNames`: Is a list of filenames (with absolute paths), each containing an input to the model. There is no restriction on the format of the files, but the model should declare what file formats it expects. Predicates to parse a wide range of fileformats are supplied in the *io* API (see section 3.2).
- `Options`: Is a list of facts on the form, where the functor works as key and the first argument serves as the value of the option, egg. `key(Value)`. This list is used to parameterize the model in various ways. To check the value of a given option, the predicate `get_option` (see section **??**) should be used. The options must be declared as described in section .
- `OutputFilename`: Full filename which the resulting "best annotation" should be saved to. The model is expected to save the resulting annotation to this file before the completion of `lost_best_annotation`. The *io* API contains some common predicates for saving annotations (see section 3.2).

`lost_learn(+InputFileNames,+Options,+OutputFilename)`

This predicate is used for training models. The model is expected to save the result of the training session (e.g. a switch parameter file or similar) to `OutputFilename`. It expects three arguments

- `InputFileNames`: Is a list of filenames (with absolute paths), each containing an input to the model. These are used for providing the traning data. There is no restriction on the format of the files, but the model

should declare what file formats it expects. Predicates to parse a wide range of file formats are supplied in the *io* API (see section 3.2).

- `Options`: Is a list of facts, where the functor works as key and the first argument serves as the value of the option, eg. `key(Value)`. For convenience, option values can be checked an extracted using the predicates `get_option` (see section **??**).
- `OutputFilename`: Full filename which the resulting switch parameters or similar should be saved to. The model is expected to save the result to this file when `lost_learn` is called. If the model is unable to do this, it should throw an exception.

2.1.2 Declaration of options for interface predicates

Options to interface predicates should be declared in the `interface.pl` file. An option is declared by adding a fact

```
lost_option(+InterfaceGoal,+OptionName,+DefaultValue,+Description).
```

- `InterfaceGoal`: InterfaceGoal is one of
  - `lost_best_annotation`
  - `lost_learn`
- `OptionName`: An atom for the name of key. This is the functor identifying this particular option.
- `DefaultValue`: If the option is left unspecified then it will take this value.
- `Description`: A textual description of the purpose of the option. Should be in single quotes.

As example, consider an option declaration:

```
lost_option(lost_best_annotation,debug,no,'Enable/disable debug prints').
```

In this declaration, we state that the goal `lost_best_annotation` take the option `debug`. If this option is not a member of the `Options` list, then it will be added with the default value `no` before `lost_best_annotation` is called by the framework.

For instance, assuming the model is called `my_model`, it might be called from a script with

```
get_annotation_file(my_model,['/some/path/infile'],[],FileOut),
```

which will start a new PRISM process and invoke

```
lost_best_annotation(['/some/path/infile'],[debug(no)],FileOut),
```

in `$MODELS/my_model/interface.pl`. Note that the option `debug` has been inserted with the default value. A default value is only inserted if the option is left unspecified.

Declaring option values

If the values of an option can be enumerated, then the model should provide a declaration of by defining a predicate,

```
lost_option_values(+Predicate,+OptionName,-ValuesList).
```

In our example above, the option `debug` might have two possible values, `no` and `yes`, so a declaration might look like,

```
lost_option_values(lost_best_annotation, debug, [yes,no]).
```

Option checking

When a model is invoked by the framework, options are checked before invoking the model. If a model is called with an option which has not been declared by the model, then a warning will be issued. Similarly, if a declared option for a predicate also has a `lost_option_values` declaration and the a predicate is called with the option set to value which is not part of the declaration, the a warning is issued.

2.1.3 Declaration of file formats for interface predicates

Input file formats

For each model interface predicate the format of input files and output files should be declared as facts in the `interface.pl` file for the model. To declare the format of the input files, a fact,

```
lost_input_format(PredicateName, [ Format1, .., FormatN ]).
```

where `PredicateName` is the name of the predicate to which the declaration belong. The second argument is a list of format identifiers as described in chapter **??**. For instance, we may have a declaration,

```
lost_input(lost_best_annotation,
    [ text(prolog(sequence(dna))), text(prolog(sequence(amino_acids)))]).
```

The last entry of the list of formats may have the special star quantifier, which works similar to a kleene star: It states that the last entry is a placeholder for between zero or more entries of the specified type. For instance,

```
lost_input(lost_best_annotation,
    [ text(prolog(sequence(dna))), star(text(prolog(sequence(dna))))]).
```

specifies that `lost_best_annotation` takes one or more input files of the type `text(prolog(sequence(dna)))`.

Output file formats

Contrary to the input file formats, the output file format can depend on the options given to the model. That means, the model may produce files in different output formats depending on the values of the options it is called with.

The output format is thus declared with a predicate,

```
lost_output_format(PredicateName,Options, Format)
```

This predicate should specify the relation between a set of `Options` and a particular `Format` for the interface predicate `PredicateName`. `Format` should unify to one of the formats described in chapter **??**.

Checking of file formats

The framework checks that file formats are declared according to the scheme described above. If a model neglects to declare file formats or declares them wrongly, then a warning will be issued.

Chapter 3

Lost shared APIs

To use the lost APIs, the file `$LOST/lost.pl` must be consulted. To make the models independent of the absolute path of the `$LOST` directory, they should consult it with a path relative to the model path (e.g. `:- ['../../lost.pl'].`). Then, APIs, which are located in the `$LOST/lib` directory can be consulted using the goal,

```
lost_include_api(+APIName)
```

where `APIName` is the name of a Prolog file located in the `$LOST/lib/` directory except the `.pl` extension.

3.1 The interface API: `interface`

The API provides the interface to lost models following the conventions described in section 2.1.

The predicate,

```
get_annotation_file(Model, Inputs, Options, OutputFile)
```

is used to retrieve the best annotation generated by a specified `Model` invoked with the specified `Inputs` and `Options`. `Inputs` is a list of file names containing input for the model. `Options` is a list of facts on the form `key(Value)`, where `key` is the name of an option and `Value` is a concrete value.

As a consequence of calling this predicate the output *best annotation*) from the model is written to the file `OutputFile`. If the model has been previously invoked with exactly the same `Inputs` and `Options` , then `OutputFile` will unify with the outcome of the previous invokation. In this way, the results of running a model are cached.

The generated annotation files are named according to a convention, where all annotation files will be placed in the `$LOST/data/` directory and the `Filename` is construed according to the following convention:

```
{Modelname}_{Id}.gen
```

x

The first time a model `OutputFile` is generated, the file `annotation.idx` will be created in `$LOST/data/`. This file serves as a database to map the filename of a generated annotation file to the model, input files and options that generated the particular annotation. This database file contains Prolog facts of the form

```
fileid(Id,Filename,Model,Options,InputFiles).
```

The annotation index is automatically maintained by `get_annotation_file/4` and should not be edited by hand.

If an annotation for a particular run of a model is not present then `get_annotation_file/4` will start a new PRISM process that invokes the `lost_best_annotation` predicate provided be the model `interface.pl` file. By the contract of model conventions, the model will generate the annotation and save it to the file indicated by the provided filename.

3.2 Input-Output API: `io`

In this module (`io.pl`), several predicates are defined to manipulate `*.seq` files :

- loading information from files that extracts from a file data information used as input of models (sequence annotation for example);
- saving information into a file;
- and maybe more.

3.2.1 Loading information from files

- `load_annotation_from_file(++Type_Info,++Options,++File,--Annotation)`: Generate from `File` a sequence of `Annotation`. It is assumed that `File` is composed of terms. `Type_Info` is used to specify what format of information into file
  - `sequence` means that information is stored into a list. For example,

    ```
    data(Key_Index,1,10,[a,t,c,c,c..]).
    ```

  - `db` means that information is represented by a set of range that specified specific zone (coding region for example)

    ```
    gb(Key_Index,1,10).
    ```

  For each `Type_Info`, several options are available represented by the list `Options`. Options available for `sequence`:

  - [] (default): data list is the 2^{th} argument of the terms and these lists of data are appended;

- data_position(Num) specified that data list is `Num`th argument of term;
- `range(Min,Max)` extracts from the list of the complete annotation the sublist from position `Min` to position `Max`;
- `all_lists`: generate a list of each data list by term. Warning: `range(Min,Max)` is not support by this option.

File Example `toto.seq`:

```
data(Key_Index,1,5,[1,2,3,4,5]).
data(Key_Index,6,10,[6,7,8,9,10]).
data(Key_Index,11,15,[11,12,13,14,15]).
```

Results of request are:

```
| ?- load_annotation_from_file(sequence,[data_position(4)],'toto.seq',R).
R = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] ?
| ?- load_annotation_from_file(sequence,[data_position(4),range(4,10)],'toto.seq',R).
R = [4,5,6,7,8,9,10] ?
load_annotation_from_file(sequence,[data_position(4),all_lists],'toto.seq',R).
R = [[1,2,3,4,5],[6,7,8,9,10],[11,12,13,14,15]] ?
```

Options available for `db`:

- [] (default): first and the second element of the term defined a range. A list of 0-1 values is generated, 0 when you are outside ranges and 1 you are inside at least one;
- `in_db(Letter)` replaces the default value 1 by `Letter`;
- `out_db(Letter)` replaces the default value 0 by `Letter`;
- `range_position(Min,Max)` allows to specify the position argument number of a term of the minimal and maximal value of term;
- `range(Min,Max)` extracts from the list of the complete annotation the sublist from position `Min` to position `Max`;

File Example `toto.seq`:

```
gb(3,5).
gb(7,9).
gb(8,11). % Overlap ;)
```

Results of request are:

```
| ?- load_annotation_from_file(db,[],'toto.seq',R).
R = [0,0,1,1,1,0,1,1,1,1,1] ?
| ?- load_annotation_from_file(db,[in_db(c),out_db(nc)],'toto.seq',R).
```

```
R =  [nc,nc,c,c,c,nc,c,c,c,c,c]?
| ?- load_annotation_from_file(db,[range(3,7)],R).
R = [1,1,1,0,1] ?
| ?- load_annotation_from_file(db,[in_db(c),out_db(nc),range(8,16)],'toto.seq',R).
R =  [c,c,c,nc,nc,nc,nc,nc]?
```

- `get_sequence_from_file(++File,++Options,--Sequence)`: Generate from `File` of data a sequence (read: a list) of data `Sequence`. This predicate is a more efficient implementation of `load_annotation_from_file\4` with the type option `sequence` when several ranges of data have to be computed (option `ranges`)

  We assumed that data facts of `File` are composed of facts with the format:

  ```
  Term =.. [_Data_Functor,_Key,Range_Min,Range_Max,Data]
  ```

  File Example `toto.seq`:

  ```
  data(toto,1,10,[1,2,3,4,5,6,7,8,9,10]).
  data(toto,11,20,[11,12,13,14,15,16,17,18,19,20]).
  data(toto,21,30,[21,22,23,24,25,26,27,28,29,30]).
  ```

  Result of the request is:

  ```
  | ?- get_data_from_file('toto.seq',[],R).
  R = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                            21,22,23,24,25,26,27,28,29,30] ?
  ```

  Different options are available to specify the format of the data term:

  - `data_position(Pos)` to specify an other position for the data;
  - `left_position(Left)` to specify an other position for the left bound of the range that specifies the position of the data into the data file. When this left bound is not available, the keyword `none` for `Pos` can be used;
  - `right_position(Right)` to specify an other position for the right bound of the range that specifies the position of the data into the data file. When this right bound is not available, the keyword `none` for `Pos` can be used.

  File Example `toto2.seq`:

  ```
  data(1,10,[1,2,3,4,5,6,7,8,9,10]).
  data(11,20,[11,12,13,14,15,16,17,18,19,20]).
  data(21,30,[21,22,23,24,25,26,27,28,29,30]).
  ```

  Result of the request is:

```
| ?- get_data_from_file('toto2.seq',
                        [left_position(1),
                         right_position(2),
                         data_position(3)],R).
R = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                           21,22,23,24,25,26,27,28,29,30] ?
```

To ask for specific ranges of data, two options are available:

- `range(Min,Max)` computes the sequence of data between `Min` and `Max`;
- `ranges(List_Ranges)` computes sequences of data for a list of ranges with a format `[[Min1,Max1],[Min2,Max2],...]`. Note that the ranges should be ordered, i.e `Min1` $\leq$ `Min2` must be satisfied.

Result of the request is:

```
| ?- get_data_from_file('toto.seq',[range(5,10)],R).
R = [5,6,7,8,9,10] ?
| ?- get_data_from_file('toto.seq',[ranges([5,10],[5,8],[9,15])],R).
R = [[5,6,7,8,9,10],[5,6,7,8],[9,10,11,12,13,14,15]] ?
```

3.2.2 Saving information to files

The io API also contains some predicates for saving sequences to a file.

- `save_annotation_to_sequence_file(+KeyIndex,+ChunkSize,+Annotation,+File)`: Saves the data in the list given by `Annotation` to the file `File` in *sequence* format which is number of Prolog facts, on the form:

  ```
  data(KeyIndex,1,5, [a,t,c,c,g]).
  ```

  The first argument `KeyIndex` is used as an identifier. `ChunkSize` is the number of elements from `Annotation` to store with each fact. The second and third argument of a stored fact, corresponds to the start and end position in `Annotation`. The fourth argument of the fact is a list containing the relevant elements of the `Annotation` list. Note that if the the length of `Annotation` is not a multiple of `ChunkSize`, then the last fact stored will have a shorter range.

3.3 The Stats API: `stats`

In this module (`stats.pl`), several predicates are defined to automatically compute frequencies, probabilities of occurrences of nucleotides, codons ... This computation is based on a simple counting method given a simple input data.

Variable `Data_Type` is used to specify the type of the input data: Data type available are:

- `nucleotide` where input data are composed of letter from $\{a, c, g, t\}$;
- `codon` where a codon is represented by a list of three letters from $\{a, c, g, t\}$;
- `amino_acid` where input data are composed of letter from $\{a, c, d, e, f, g, h, i, k, l, m, n, p, q, r, s, t, v, w, y\}$;
- `length`. This type is used to compute frequencies and statistics about length of specific region of a genome.

Here is the description of main predicates of `stats.pl`

- stats(++Data_Type,++Options,++Data,++Input_Counting,–Result)
  `Data_Type` setting define on which counting procedure is initialized. User must take care then to give the right `Data`. For {`nucleotide,codon,amino_acid`} data type, `Data` must be a Prolog list with the right element, For `length` data type, `Data` must be a list of ranges represented by a list of two integer values ([Min,Max]). Two options are available:
  - `order(Num)` defines the size of the past stored to perform the counting
  - `past(List)` allows to give as input of the computation a previous past. This option is useful to make connection between two counting computations.

  `Input_Counting` is used as well to perform a counting computation on an input data divided into a set of lists. By default, counting computation is initialized when is a variable `Input_Counting`. However, the counting can restart from `Input_Counting` if this variable is unified to the result of a previous result. Finally, `Result` has the following format:

```
[(Past1,[(Elt1,Count11),(Elt2,Count12),...]),
 (Past2,[(Elt1,Count21),(Elt2,Count22),...]),
 ...]
```

  For example let consider this following data:

```
data('U00096',1,5,[a,g,c,t,t]).
data('U00096',6,10,[c,c,c,c,c]).
```

  Here is the result of this following request

```
| ?- stats(nucleotide,[],[a,g,c,t,t],_,R).
R = [([],[(a,1),(c,1),(g,1),(t,2)])] ?
| ?- stats(nucleotide,[order(1)],[a,g,c,t,t],_,R).
R = [([a],[(a,0),(c,0),(g,1),(t,0)]),
     ([c],[(a,0),(c,0),(g,0),(t,1)]),
     ([g],[(a,0),(c,1),(g,0),(t,0)]),
     ([t],[(a,0),(c,0),(g,0),(t,1)])]?
```

```
| ?- stats(nucleotide,[order(2)],[a,g,c,t,t],_,R1),
     stats(nucleotide,[order(2),past([t,t])],[c,c,c,c,c],R1,R).
R = [([a,a],[(a,0),(c,0),(g,0),(t,0)]),
     ([a,c],[(a,0),(c,0),(g,0),(t,1)]),
     ([a,g],[(a,0),(c,1),(g,0),(t,0)]),
     ([a,t],[(a,0),(c,0),(g,0),(t,0)]),
     ([c,a],[(a,0),(c,0),(g,0),(t,0)]),
     ([c,c],[(a,0),(c,3),(g,0),(t,1)]),
     ([c,g],[(a,0),(c,1),(g,0),(t,0)]),
     ([c,t],[(a,0),(c,0),(g,0),(t,1)]),
     ([g,a],[(a,0),(c,0),(g,0),(t,0)]),
     ([g,c],[(a,0),(c,0),(g,0),(t,1)]),
     ([g,g],[(a,0),(c,1),(g,0),(t,0)]),
     ([g,t],[(a,0),(c,0),(g,0),(t,1)]),
     ([t,a],[(a,0),(c,0),(g,0),(t,0)]),
     ([t,c],[(a,0),(c,0),(g,0),(t,1)]),
     ([t,g],[(a,0),(c,0),(g,0),(t,0)]),
     ([t,t],[(a,0),(c,1),(g,0),(t,0)]),
     ]?
```

- stats(++Data_Type,++Options,++Data,++Input_Counting,–Past,–Result) `stats`/6 computes exactly the same counting. This predicate allows only to record into `Past` the last past of the computing. This predicate is usefull to compute statistics on a series of data input. For example, this following request

  ```
  | ? - stats(nucleotide,[order(2)],[a,g,c,t,t],_,Past,R1),
        stats(nucleotide,[order(2),past(Past)],[c,c,c,c,c],R1,Past2,R).
  Past = [t,t],
  Past2 = [c,c],
  R = ... ?
  ```

  can be used to obtain counting information on the previous data.

- normalize(++Data_Type,++List_Counting,–Probabilities) From the result of a counting procedure, `normalize`/3 modified `List_Counting` to compute domains and probabilities distributions.

For example, here is the result of several requests:

```
| ?- stats(nucleotide,[],[a,g,c,t,t],_,R),
     normalize(nucleotide,R,S).
S = [([],([a,c,g,t],[0.2,0.2,0.2,0.4]))]
| ?- stats(nucleotide,[order(1)],[a,g,c,t,t],_,R),
     normalize(nucleotide,R,S).
S = [([a],([a,c,g,t],[0,0,1,0])),
     ([c],([a,c,g,t],[0,0,0,1])),
```

```
    ([g],([a,c,g,t],[0,1,0,0])),
    ([t],([a,c,g,t],[0,0,0,1]))] ?
```

To get a representation of the statistics where each count is represented as a single fact, the predicate

```
build_stat_facts(+Stats,-StatsFacts)
```

can be useful. For instance,

```
| ?- stats(nucleotide,[],[a,g,c,t,t],_,R), build_stat_facts(R,F).
R = [([],[(a,1),(c,1),(g,1),(t,2)])]
F = [stat([a],1),stat([c],1),stat([g],1),stat([t],2)] ?
```

3.4 The `regex` API

The `regex` API provides POSIX-like regular expressions for matching Prolog strings and atoms. A regular expression is created using

```
re_compile(+Regex, -CompiledRegex)
```

The input argument `Regex` is an atom or a list of symbols representing a particular regular expression. The output argument `CompiledRegex` is used in subsequent matching with this regular expression.

Matching with a regular expression is then done using

```
re_match(+CompiledRegex,+String,-Matches)
```

The goal `re_match/3` is true whenever the `CompiledRegex` matches the `String`. The `String` may be an atom or a list of symbols. If the goal is true, then `Matches` is a list containing the matching parts of the `String` corresponding to particular match groups (paranthesized sub-expressions) of the regular expression.

Currently the regular expressions can be expressed using a significant subset of the well-known POSIX regular expression operators. The following operators are supported:

- *Concatenation:* is expressed by the concatenation of two sub expressions. For instance, the regular expression `ab` matches an `a` followed by a `b`.
- *Alternation:* is expressed with the vertical bar (`|`). For instance the expression `a|b` matches either `a` or `b`.
- *Repetition:* Repetition operators indicate the number of times a preceding regular expression may be matched. The following repetition operators are supported:
  - `*` representing the Kleene star, meaning that the preceding expression is matched zero or more times.

- `+` the preceding expression one or more times.
- `?` The preceding expression zero times or one time.

Note that repetition binds stronger than alternation and alternation binds stronger than concatenation. To enforce a different binding order, enclosing sub-expressions in parentheses (which binds strongest) are used.

Parenthesised sub-expressions can also be used to create *match groups.* An outer-most paranthesized sub-expression, where the paranthesis are superflouos is interpreted as a match group. When an expression is matched, then the part of the string matched by the paranthesized expression is extracted.

Additionally, ranges are supported by the regular expression syntax. A range is expressed using square brackets. For instance, `[a-z0-9]` is range expression which matches a lower case alphanumeric character or a digit.

3.5 The genecode API

This API contains a set of genetic code tables. Each table defines a relation between codons and amino acids. The main predicate of the API is

```
genecode(+Code, ?Codon, ?AminoAcid)
```

The argument code is an integer specifying the genetic code to use. Currently, only genetic code `11` (bacterial) is supported. `Codon` is a list of of three nucleotides symbols from the alphabet `[a,g,c,t]`. The argument `AminoAcid` is a symbol from the alphabet

```
[a,c,d,e,f,g,h,i,k,l,m,n,p,q,r,s,t,v,w,y,*]
```

Where all symbols except `*` represents an amino acid (the name of which starts which that letter). The special symbol `*` signifies a stop codon in the specified genetic code.

Other predicates in the genecode API include

```
genecode_start_codons(+GeneCode,-StartCodons)
```

The `genecode_start_codons/2` predicate unifies `StartCodons` with a list of all valid start codons for the given `GeneCode`.

```
genecode_stop_codons(+GeneCode,-StopCodons)
```

The `genecode_stop_codons/2` predicate unifies `StopCodons` with a list of all valid stop codons for the given `GeneCode`.

3.6 The sequence API: sequence

The `sequence` API is library containing predicates for doing common tasks with biological sequences.

3.6.1 Complementing DNA sequences

A complementary dna sequence can be achieved using the predicate

```
dna_seq_complement(+NucleotideSeq,-ComplNucleotideSeq)
```

When given the input sequence `NucleotideSeq` is a list of symbols from the alphabet `[a,g,c,t]`, then `ComplNucleotideSeq` is unified to a list of complementary symbols, e.g.

```
| ?- dna_seq_complement([a,g,a,c,t,a],X).
X = [t,c,t,g,a,t] ?
yes
```

Typically, when complementing a DNA sequence, we are actually interested in the *reverse* complement. To achieve this, use the built-in predicate `reverse/2` before `dna_seq_complement/2`.

3.6.2 Translating a DNA sequence to amino acids

The predicate,

```
dna_amino_acid(+Genecode, ?DNA_Sequence,?AminoAcids)
```

Translates a DNA sequence to a sequence of amino acids or vice versa; produces all possible DNA sequences for a given amino acids sequence. `Genecode` is an integer for the genetic code to be used in the translation. `DNA_Sequence` is a list of symbols from the alphabet `[a,g,c,t]` and length of the list is expected to be divisible by three. `AminoAcids` is a list of symbols from the alphabet,

```
[a,c,d,e,f,g,h,i,k,l,m,n,p,q,r,s,t,v,w,y,*].
```

Chapter 4

Models

Pre-defined models are introduced. These models are used to build more complex models.

4.1 Parsers of Biological Data

To extract information from different Biological database, several parsers have been designed to parse report of analyses (Blast, Easygene and Genemark) and database (Genbank). These parsers generated a series of Prolog terms that can be used after that input of different probabilistic models. `script_parser.pl` collects different scripts to generate different data files.

4.1.1 Parser_fna

*.fna file of Genbank is composed of a complete genome in the FASTA format. Parser_fna permits to parse this *fna.file from Genbank and generate list of terms. These terms store the genome into a Prolog list composed of $\{a, c, g, t\}$. Two scripts are implemented:
`parser_fna(++Name_FNA_File,++Name_GBK_File,++Options)` and
`parser_fna(++Name_FNA_File,++Name_GBK_File,++Options,--OutputFile)`.
Note that *.gbk file is necessary as well. This file is used to automatically extract genome information (Genbank key and size of genome).

Output format of the generated terms are:
`data(Genebank_Key,Start,End,List_of_Data)`. Option `list(Number)` can be used to divide the complete genome into several lists with a length defined by the parameter `Number`. Example with the E.Coli K12 genome:

```
>gi|48994873|gb|U00096.2| Escherichia coli .....
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGG.....
.....
```

Result by default:

```
%>gi|48994873|gb|U00096.2| Escherichia coli ....
data('U00096',1,4639675,[a,g,c,t,t,t, ...]).
```

Result when `list(280)` option is used

```
%>gi|48994873|gb|U00096.2| Escherichia .....
data('U00096',1,280,[a,g,c,t,...]).
data('U00096',281,560,[c,c,c,...]).
....
```

4.1.2 Parser_ptt

A PTT file is the NCBI format for representation of a *protein table*. Typically, ptt files are composed of information about known or predicted protein coding genes in a genome. Each such gene is represented by a single line where tabulation separated fields represent various properties of the gene such as the position, length, name and protein product. The `parser_ptt` model parses PTT files and generates a Prolog term in the output file for each gene in the input (ptt) file. We use the format identifier, `text(ptt)`, for PTT files, which is also the input format of `parser_ptt`. The output format is `text(prolog(ranges(gene)))`. The model does not declare any options.

Consider the following extract of a PTT file:

```
Location Strand Length PID Gene Synonym Code COG Product
190..255 + 21 1786182 thrL b0001 - - thr operon leader peptide
```

The first line is just a header describing the names of the individual fields, but the second is an entry for a particular gene. The corresponding Prolog term generated by the parser for this gene is then

```
gb(190,255,'+',1,[
    gene_name(thrL),length(21),pid(1786182),
    synonym(b0001),code('-'),cog('-'),
    product('thr operon leader peptide')]
).
```

Additionaly, the file `$SCRIPTS/script_parser.pl` implements two helper predicates to make it easy to use this parser:
`parser_ptt(++Name_PTT_File)` and
`parser_ptt(++Name_PTT_File,,--OutputFile)`.

4.1.3 Parser_Easygene

This parser allows to generate from the GFF format of Easygene prediction several Prolog terms with the following format: `eg(Left,Right,Dir,Frame,[])`.

Two scripts are implemented:
`parser_easygene(++Report_Name)` and
`parser_easygene(++Report_Name,--OutputFile)`.

4.1.4 Parser_Genemark

This parser allows to generate from a report of Genemark.HMM of Easygene prediction several Prolog terms with the following format: `gm(Left,Right,Dir,Frame,[])`. Genemark report of prediction coordinates is parsed.

Two scripts are implemented:
`parser_genemark(++Report_Name)` and
`parser_genemark(++Report_Name,--OutputFile)`.

4.1.5 Parser_Blast

This parser translated into Prolog terms the XML report of Tblastn. The parser a fact for each hits detected in the XML report.

4.2 Models for measurement and statistical reports

4.2.1 accuracy_report

The model `accuracy_report` can be used the produce a report of various measures of the accuracy of particular gene predictions compared with a golden standard such a genebank.

To use the the model, you need to call `get_annotation_file/4` with the following arguments,

```
get_annotation_file(accuracy_report,
    [ReferenceFile,PredictionFile],
    [
     option(reference_functor,RefFunctor),
     option(prediction_functor,PredFunctor),
     option(start,StartPos),
     option(end,EndPos)
    ],
    OutputFile),
```

`ReferenceFile` must be the full path to a file with facts representing the "correct predictions". `PredictionFile` must be the full path to a file with facts representing the predictions. Both files must be a a `db` type format, with facts on the following form,

```
functor(To, From, Strand, ReadingFrame, Name).
```

Each such fact represent a prediction in the `PredictionFile` or correct gene in the in `ReferenceFile`. The `functor` is a any given functor, but the `ReferenceFile` and the `PredictionFile` should use different functors. The *To* argument represents the position in the genome where the prediction begins (inclusive) and the `From` argument represents the end position of the prediction

`ReadingFrame` is a integer in the range $\{1, 2, 3\}$ `Strand` is either `+` for the forward strand or `-` for the reverse strand.

`get_annotation_file` for the `accuracy_report` model must be called with four mandatory options:

- `reference_functor`: The functor used in the `ReferenceFile`
- `prediction_functor`: The functor used in the `PredictionFile`
- `start`: An integer corresponding to the beginning of the range on which accuracy should be measured.
- `end`: An integer corresponding to the end of the range (inclusive) on which accuracy should be measured.

4.2.2 The `range_stats` model

The `range_stats` model calculates statistics for each gene or similar from a file in the format `text(prolog(ranges(gene)))`. The output file is also in the `text(prolog(ranges(gene)))` format, with an entry for each range in the original file, but with some statistics facts appended to the `ExtraList`.

The model takes two input files,

- A *ranges file* in `text(prolog(ranges(gene)))` format.
- A *data file* in `text(prolog(sequence(gene)))` format.

The model can generate statistics basic nucleotide frequencies, amino acids frequencies and length. The nucleotide sequence for the range is automatically extracted from the *data file* and translated or reverse complemented as necessary. It is possible to adjust which kinds of statistics that should be generated by supplying the model with relevant options:

- `amino_acid_stats`: Whether to add statistics based on amino acid frequency. Values are `yes` (default) or `no`.
- `nucleotide_stats`: Whether to add statistics based on nucleotide frequency.Values are `yes` (default) or `no`.
- `length_stats`: Whether to add statistics based on the length of the range.Values are `yes` (default) or `no`.
- `max_nucleotide_order(N)`: Is an integer that determines the maximal order of nucleotide n-grams to calculate statistics for. Given this option the model will calculate statistics for nucleotide n-grams of size N+1 down to size 1. The default value is 1, which that single nucleotide frequency and di-nucleotide frequency will be calculated.

- `max_amino_acid_order(N)`: Is an integer that determines the maximal order of n-grams to calculate statistics for. Given this option the model will calculate statistics for amino acid n-grams of size N+1 down to size 1. The default value is 0 (e.g. only single amino acid frequencies).
- `genecode`: An integer which identifies the gene code table to use when translating to amino acid sequence. Default is 11 (bacterial).

Depending the on the options for the model a subset of the following facts are appended to the `ExtraList` of each gene in the output file:

- `nucleotide_stats(L)` where `L` is list of elements of the type `stat(Seq,Freq)` and `Seq` is list of nucleotides signifying a particular n-gram (n being the length of list) and `Freq` is the frequency of the n-gram in the gene.
- `amino_acid_stats(L)` where `L` is a list of elements of the type `stat(Seq,Freq)` and `Seq` is list of amino acid symbols signifying a particular n-gram (n being the length of list) and `Freq` is the frequency of the n-gram in the gene.
- `normalized_gene_length(F)` where `F` is a number in the range $0 \cdots 1$ representing the proportion of the length of the range relative to the longest gene in the file. For instance, if the longest gene in the file is 2000 nucleotides long and the gene in question is 1000 nucleotides long, then the normalized gene length will be $1000/2000 = 0.5$.

4.3 Prediction/Gene filter models

Common for these models is that they all take files in the `text(prolog(ranges(_)))` format and produce output in the same format. Usually, the output will be a subset of the entries from the input file.

4.3.1 longest_predication_per_stop_codon

Given an input file with gene predictions, this model selects for each stop codon the longest matching prediction. Only the longest prediction for each stop coden will be written to the output file.

For the goal `lost_best_annotation/3`, the following files and options are expected:

- Input files: `text(prolog(ranges(gene)))`
- Output file: `text(prolog(ranges(gene)))`

The model only takes one option, `file_functor`, which is used to set the functor to identify the gene predictions in the input file. If the input file contains only facts with the same functor the default value `auto` can be used and the model will infer the functor automatically.

4.3.2 best_prediction_per_stop_codon

Given an input file with gene predictions this model selects, for each stop codon, the prediction with the highest score and write such predictions to the output file. The score is for each prediction is assumed to be present the `Extra` list argument of the prediction facts in the input file.

For the goal `lost_best_annotation/3`, the following files and options are expected:

- Input files: `text(prolog(ranges(gene)))`
- Output file: `text(prolog(ranges(gene)))`

The model only takes two option:

- `file_functor`: Is used to set the functor to identify the gene predictions in the input file. If the input file contains only facts with the same functor the default value `auto` can be used and the model will infer the functor automatically.
- `score_functor`: Is used to specify how the score is represented in the `Extra` list. For instance, if the Extra list looks like `[...,score(0.75),..]`, then the `score_functor` option should be set to `score`. The argument of the score functor (`0.75` in the example) should be a number or at least something comparable with the standard less-than-or-equal-to Prolog operators.

4.3.3 gene_filter

The `gene_filter` model is a model that filters a set of genes based on various matching criteria. The input to the model is a file in `text(prolog(ranges(gene)))` format containing the list of genes to be filtered and a file in `text(prolog(sequence(dna)))` that includes the nucleotide sequence for the entire genome to which the genes belong.

The model supports filtering based on a wide range of criteria, specified using the options given to the model. The model is invoked using `get_annotation_file/4`.

The options supported by the model are:

- `match_frames`: A list of valid frames. Genes in other frames will be filtered. The list must be a subset of the *default value*: `[1,2,3]`.
- `match_strands`: A list of valid strands. Genes occuring an other strand will be filtered. The list must be a subset of the *default value*: `['+','-']`.
- `exact_match_extra_fields`: A list of Prolog terms terms each of which must unify with an element of the `Extra` list of the gene terms. Genes which do not are filtered.

- `exact_no_match_extra_fields`: A list of Prolog terms terms any of which *may not* unify with any element of the `Extra` list of the gene terms. Genes which do are filtered.

- `regex_match_extra_fields`: A list of regular field names in the `Extra` list and regular expressions to match those fields. For example, a regular expression matcher for the `name` field would be specified as `name('^y.*$')`, here matching genes with a name starting with the letter y. The syntax of the regular expressions are as specified in section 3.4. Gene tems that do not match each of the specified regular expressions are filtered.

- `regex_no_match_extra_fields`: A list of field names in the extra and regular expressions to match those fields. If a gene is matched by any of these regular expression, it will be filtered.

- `match_protein_coding`: Must be either `yes` or `no`. Default is `no`. If set to `yes`, this will filter gene terms which do not have a valid start and stop codon. The codon table to be used is specified using the the `genecode` option.

- `genecode`: An integer specifying what genecode to use. Default is `11`, which is the bacterial gene code. See 3.5 for a list of valid gene code tables.

- `invert_results`: Must be either `yes` or `no`. Default is `no`. If set to `yes`, then only the otherwise non-filtered results are filtered.

4.4 Model that integrate other programs

4.4.1 Genemark

To be documented...

4.4.2 Glimmer3

To be documented...

4.5 Various models

4.5.1 hard_to_find_genes

The model `hard_to_find_genes` ranks a given set of genes according to how many genefinders that find the gene.

The input to the model is a list of files, of which the first element is taken to the *golden standard* - a file containing a set of *true* genes. The rest of the n files in the input file list are files that contain gene predictions of various

genefinders (one genefinder per input file). All the files are expected to be in the `text(prolog(ranges(gene)))` format.

For each gene in the *golden standard* file, the model checks which of the gene finders predicts this gene. It writes each of these genes to the output file, but with facts added to the `Extra` list. The added facts are

- `found_by_genefinders(F)`: Where `F` is a list of n entries containing
  - `1` if the nth gene finder predicts the gene.
  - `0` if the nth gene finder did not predict the gene.
- `gene_finding_difficulty_score(S)`: Where `S` is a decimal number between zero and one representing the inverse of the proportion of genefinders that predicts this gene. E.g. if it is zero, no gene finders predicts the gene, and if it is one, then all n gene finders predicts the gene.

The model take a number of options:

- `start`: Is a positive integer indicating the start of the range in which to consider genes. Only genes starting/ending after this position are included in the output file. The default value `min` is taken to mean the start of the file, e.g. position 1.
- `end`: Is a positive integer indicating the end of the range in which to consider genes. Only genes with a right border below this value included in the output file. The default value `max` is taken to mean the maximal position of any gene in the *golden standard* file.
- `gene_match_criteria`: Is used to determine what is required for a gene prediction to *match* a gene. If the value is set to `start_and_stop` (default), then both start codon and stop codon needs to match. If the value is set to `stop` then only the stop codon needs to match.

Chapter 5

File formats

The chapter documents the different types of file formats used in the lost framework. The naming of file formats follow a simple scheme using nested Prolog functors where the outermost functor is the most general and the innermost is the most specific description of the file format. For instance, consider the naming for a DNA sequence expressed as a Prolog file,

```
text(prolog(sequence(dna)))
```

The outermost functor `text` specifies that the file is a text file, and the next functor `prolog` says that the text file contains Prolog code. The next one, `sequence`, specifies the type of data (sequence data) we expect to be expressed in the prolog facts finally, the innermost functor `dna` specifies the type of sequence that we are dealing with.

5.1 text(prolog(_))

`text(prolog(_))` formats contain Prolog facts. The functor and arity of those facts cannot be determined by knowing that it is `text(prolog(_))`, but needs specification using further embbed functors.

5.1.1 text(prolog(sequence(_)))

This format should be specified for sequence data expressed as prolog facts. For instance, a file of this format may contain facts like the ones below, which specifies the alphabet,

```
data(alphabet,1,10, [a,b,c,d,e,f,g,h,i,j]).
data(alphabet,11,20],[k,l,m,n,o,p,q,r,s,t]).
data(alphabet,21,27,[u,v,w,x,y,z]).
```

The form of these facts are,

```
functor(Identifier,To,From,SequenceElementList).
```

The format does not dictate the `functor` of the facts (e.g. `data`), nor the size of the data list in the fourth argument. However, the range expressed by `To` and `From` should correspond to the numer of elements in `SequenceElementList`.

`text(prolog(sequence(dna)))`

Like the above format but restricts the alphabet of the data elements in the `SequenceElementList` to the set {`a,g,c,t`}.

`text(prolog(sequence(rna)))`

Like the above format but restricts the alphabet of the data elements in the `SequenceElementList` to the set {`a,g,c,u`}.

`text(prolog(sequence(amino_acids)))`

Like the above format but restricts the alphabet of the data elements in the `SequenceElementList` to the set {`a,c,d,e,f,g,h,i,k,l,m,n,p,q,r,s,t,v,w,y`}.

5.1.2 `text(prolog(ranges(_)))`

The format consists of Prolog facts, each of which contain an annotation for a particular range of some sequence

```
F =.. [ Functor, LeftEnd, RightEnd | _ ]
```

The functor may vary depending on the type of annotation but is expected to be same within a file. The first two arguments, `LeftEnd` and `RightEnd` are positive integers which specifies the range relative to the sequence. Both are inclusive.

`text(prolog(ranges(gene)))`

The format consists of Prolog facts, each of which contain an gene annotation for a particular range of some DNA sequence (all facts refer to the same sequence).

The facts are on the form,

```
functor(LeftEnd,RightEnd,Strand,Frame,ExtraList).
```

The functor may vary depending on the type of annotation but is expected to be same within a file. The first two arguments, `Start` and `End` are positive integers which specifies the range of the annotation relative to the DNA sequence. The `Strand` argument is `+` for the forward strand or`-` for the reverse strand. The `Frame` argument is one of {`1,2,3`}. The final argument, `ExtraList` is a list possibly containing extra information.

5.1.3 `text(prolog(prism_switches))`

This is the format used by PRISM to save and load parameter files.

5.1.4 text(ptt)

TODO ptt files

5.1.5 text(easygene_report)

TODO ptt files

5.1.6 text(fasta(SequenceType)

A sequence in FASTA format begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than ("¿") symbol in the first column. The word following the "¿" symbol is the identifier of the sequence, and the rest of the line is the description (both are optional). There should be no space between the "¿" and the first letter of the identifier. It is recommended that all lines of text be shorter than 80 characters. The sequence ends if another line starting with a "¿" appears; this indicates the start of another sequence.

The format identifier `text(fasta(_))` is used to refer to any type of file in FASTA format, but we distinguish between different subtypes with `SequenceType`. A ground value for `SequenceType` may be one of the following:

- `fasta`: Specifies generic fasta format. This is the same as specifying `text(fasta(_))`.
- `fna`: fasta nucleic acid.
- `ffn`: FASTA nucleotide coding regions. Contains coding regions for a genome.
- `ffa`: fasta amino acid.

Chapter 6

Notes and stuff

6.1 Feature wish list

- Division of models into models (probabilistic models) and nodes (which are just data processing).
-
- Document gene database file format and make sure that all models adhere to this format.
- Option, input and output formats for all models. (Matthieu, Christian)
- Data outside git (Christian)
- Models should `lost_tmp_directory` for intermediary files.
- Document all of undocumented models. (Matthieu, Christian)

Some things that we have discussed, but decided not to do

- Do something to avoid multiple declaration of lost_include_api everywhere... (Matthieu).