Appendix A
Excerpt from the draft manual

A.1 Predicates for programming with actors

Predicate: `self/1` ACTOR

```
self(-Pid) is det.
```

Binds `Pid` to the process identifier of the calling process.

Predicate: `spawn/1-3` ACTOR

```
spawn(+Goal) is det.
spawn(+Goal, -Pid) is det.
spawn(+Goal, -Pid, +Options) is det.
```

Creates a new Web Prolog actor process running `Goal`. Valid options are:

- `node(+URI)`
  Creates the process on the node pointed to by the URI. Default is `localhost`.
- `monitor(+Boolean)`
  If `true`, sends a `down` message to the parent process when the spawned process terminates. Default is `false`.
- `link(+Boolean)`
  If `true`, terminates all child processes (if any) upon termination of the spawned process. Default is `true`.
- `timeout(+IntegerOrFloat)`
  Terminates the spawned process (or the process of spawning a process) after `IntegerOrFloat` seconds.
- `load_text(+AtomOrString)`
  Loads the clauses specified by a Web Prolog source text into the actor's private Prolog database before calling `Goal`.
- `load_list(+ListOfClauses)`
  Loads a list of Web Prolog clauses into the actor's private Prolog database before calling `Goal`.

- `load_uri(+URI)`
  Loads the clauses specified by the Web Prolog source text at `URI` into the actor's private Prolog database before calling `Goal`. `Goal`.
- `load_predicates(+ListOfPredicateIndicators)`
  Loads the local predicates denoted by `ListOfPredicateIndicators` into the actor's private Prolog database before calling `Goal`.
- `type(+Atom)`
  Indicates the type of the source to be injected into the process. Default is `'web-prolog'`. Note that some `load_*` options may not be compatible with other values of this option.

Predicate: `monitor/1` ACTOR

```
monitor(+Pid) is det.
```

Begin monitoring the process `Pid`.

Predicate: `demonitor/1` ACTOR

```
demonitor(+Pid) is det.
```

Stop monitoring the process `Pid`.

Predicate: `register/2` ACTOR

```
register(+Name, +Pid) is det.
```

Register a process under a name, where `Name` is an atom and `Pid` identifies the actor process. The association between the name and the pid will be removed when the process terminates.

Predicate: `whereis/2` ACTOR

```
whereis(+Name, ?Pid) is det.
```

Locate the process associated with the name. Returns `undefined` if the process does not exist.

Predicate: `unregister/1` ACTOR

```
unregister(+Name) is det.
```

Remove the association between the name and the process.

Predicate: `exit/1` ACTOR

```
exit(+Reason) is det.
```

Exit the calling process with `Reason`.

Predicate: `exit/2` ACTOR

```
exit(+Pid, +Reason) is det.
```

Exit the process identified as `Pid` with `Reason`.

Predicate: `!/2, send/2-3` ACTOR

```
+PidOrName ! +Message is det.
send(+PidOrName, +Message) is det.
send(+PidOrName, +Message, +Options) is det.
```

Sends `Message` to the mailbox of the process identified as `PidOrName`. `PidOrName` must have the form `Pid@Node` or `Name@Node`. `Pid@localhost` and `Name@localhost` refer to actors running on the current node, and can often be abbreviated to `Pid` or `Name`, respectively. Valid options for `send/3` are:

- `delay(+IntegerOrFloat)`
  Delays the sending with a specified number of seconds. Default is `0`.
- `id(+ID)`
  `ID` is a user supplied identifier that can be used by `cancel/1` to stop the sending of the message to happen.

Predicate: `cancel/1` ACTOR

```
cancel(+ID) is det.
```

Tries to cancel the sending of *all* delayed messages with the specified ID. This cannot be guaranteed to succeed since a message may already have been sent by the time the call is made.

Predicate: `raise/1` ACTOR

```
raise(+Message) is det.
```

Sends `Message` to the mailbox of the current process. Bootstrapped as

```
raise(Message) :-
    self(Pid),
    Pid ! Message.
```

Predicate: `output/1-2` ACTOR

```
output(+Data) is det.
output(+Data, +Options) is det.
```

Sends a message `output(Pid,Data)` to the target process. `Pid` is the pid of the current process. Valid option:

- `target(+Pid)`
  Send the message to `Pid`. Default is the parent process.

Note that this is just a convenience predicate. A toplevel, just like any other actor, may use `!/2` to send any term to any process to which it has a pid.

Predicate: `input/2` ACTOR

```
input(+Prompt, -Data) is det.
input(+Prompt, -Data, +Options) is det.
```

Sends a message `prompt(Pid,Prompt)` to the target process and waits for its input. `Prompt` may be any term (i.e. even a compound term). `Pid` is the pid of the current process. `Data` will be bound to the term that the target process sends using `respond/2`. Valid option:

- `target(+Pid)`
  Send the `prompt` message to `Pid`. Default is the parent process.

Predicate: `respond/2` ACTOR

```
respond(+Pid, +Input) is det.
```

Sends a response in the form of the term `Input` to a process that has prompted its parent process for input.

Predicate: `receive/1-2` ACTOR

```
receive(+Clauses) is semidet.
receive(+Clauses, :Options) is semidet.
```

`Clauses` is a sequence of receive clauses delimited by a semicolon:

```
{  Pattern1 [if Guard1] ->
         Body1 ;
   ...
   PatternN [if GuardN] ->
         BodyN
}
```

Each pattern in turn is matched against the first message (the one that has been waiting longest) in the mailbox. If a pattern matches and the corresponding guard succeeds, the matching message is removed from the mailbox and the body of the receive clause is called. If the first message is not accepted, the second one will be tried, then the third, and so on. If none of the messages in the mailbox is accepted, the process will wait for new messages, checking them one at a time in the order they arrive. Messages in the mailbox that are not accepted are *deferred*, i.e. left in the mailbox without any change in their contents or order. Valid options:

- `timeout(+IntegerOrFloat)`
  If nothing appears in the current mailbox within `IntegerOrFloat` seconds, the predicate succeeds anyway. Default is no timeout.
- `on_timeout(+Goal)`
  If the timeout occurs, `Goal` is called.

A.2 Predicates for programming with toplevel actors

Predicate: `toplevel_spawn/1-2` ACTOR

```
toplevel_spawn(-Pid) is det
toplevel_spawn(-Pid, +Options) is det
```

Spawns a toplevel and binds `Pid` to its pid. With just two exceptions, all options that can be passed to `toplevel_spawn/2` are inherited from `spawn/3`. The only new options are `session` and `target`.

- `session(+Boolean)`
  If set to `false`, the toplevel actor will terminate after having run a goal to completion. If `true`, further interaction is expected. Defaults to `false`.
- `target(+Pid)`
  Send all answer terms to `Pid`. Default is the pid of the parent.

Predicate: `toplevel_call/2-3` ACTOR

```
toplevel_call(+Pid, +Goal) is det.
toplevel_call(+Pid, +Goal, +Options) is det
```

Asks the toplevel `Pid` for solutions to `Goal`. Valid options are:

- `template(+Template)`
  `Template` is a term sharing variables with the goal. By default, the template is identical to the goal.
- `offset(+Integer)`
  Collect only the slice of solutions starting from `Integer`. Default is `0`.
- `limit(+Integer)`
  By default, `toplevel_call/2-3` requests that *all* solutions to `Goal` be computed and returned as a list of solutions embedded in an answer term of type `success`. By passing the limit option, the length of this list can be restricted to `Integer`.
- `target(+Pid)`
  Send the answer term to `Pid`. Default is the value of `target` when passed as an option to `toplevel_spawn/2`.

Variables in `Goal` will not be bound. Instead, solutions and other kinds of output will be returned in the form of answer messages delivered to the mailbox of the process that called `toplevel_spawn/2-3`.

- `success(Pid, Data, More)`
  `Pid` refers to the toplevel process that succeeded in solving the goal. `Data` is a list holding instantiations of `Template`. `More` is either `true` or `false`, indicating whether or not we can expect the toplevel to be able to return more solutions, would we call `toplevel_next/1-2`.
- `failure(Pid)`
  `Pid` is the pid of the toplevel process that failed for lack of (more) solutions.

- `error(Pid, Data)`
  `Pid` is the pid of the toplevel throwing the error. `Data` is the error term.

Note that nothing stops a toplevel from sending messages of a form different from the above to the target.

Predicate: `toplevel_next/1-2` ACTOR

```
toplevel_next(+Pid) is det.
toplevel_next(+Pid, +Options) is det
```

Asks toplevel `Pid` for more solutions to `Goal`. Valid options:

- `limit(+Integer)`
  By default, the value of the `limit` option is the same as for `toplevel_call/2-3`.
- `target(+Pid)`
  Send the answer term to `Pid`. Default is the value of `target` when passed as an option to `toplevel_call/3`.

The messages delivered to the mailbox of the target are the same as for `toplevel_call/2-3`.

Predicate: `toplevel_stop/1` ACTOR

```
toplevel_stop(+Pid) is det.
```

Asks toplevel `Pid` to stop searching for more solutions.

Predicate: `toplevel_abort/1` ACTOR

```
toplevel_abort(+Pid) is det.
```

Tells toplevel `Pid` to abort the execution of any goal it currently runs.

Predicate: `toplevel_exit/1-2` ACTOR

```
toplevel_exit(+Reason) is det.
toplevel_exit(+Pid, +Reason) is det.
```

Same as `exit/1` and `exit/2`.

A.3 Built-in Predicates for RPC

Predicate: `rpc/2-3` ISOBASE

```
rpc(+URI, +Goal) is nondet.
rpc(+URI, +Goal, +Options) is nondet.
```

Semantically equivalent to the sequence below, except that the goal is executed in (and in the Prolog context of) the node referred to by `URI`, rather than locally.

```
copy_term(Goal, Copy),
call(Copy),      % executed on node at URI
Goal = Copy.
```

The following options are valid:

- `limit(+Integer)` ISOBASE
  By default, `rpc/2-3` will only make one trip to the remote node at `URI` in which it will (try to) compute all solutions to `Goal` in order to cache them at the client. A goal with n solutions and `limit` set to 1 would require n roundtrips if we wanted to see them all. With `limit` set to i, the same goal would only require $ceiling(n/i)$ roundtrips.
- `timeout(+IntegerOrFloat)` ISOBASE
  Terminates the spawned process (or the process of spawning a process) after `IntegerOrFloat` seconds.
- `load_text(+AtomOrString)` ISOTOPE
  Loads the clauses specified by a Web Prolog source text into the underlying actor's private Prolog database before calling `Goal`.
- `load_list(+ListOfClauses)` ISOTOPE
  Loads a list of Web Prolog clauses into the underlying actor's private Prolog database before calling `Goal`.
- `load_uri(+URI)` ISOTOPE
  Loads the clauses specified by the Web Prolog source text at `URI` into the underlying actor's private Prolog database before calling `Goal`.
- `load_predicates(+ListOfPredicateIndicators)` ISOTOPE
  Loads the local predicates denoted by `ListOfPredicateIndicators` into the underlying actor's private Prolog database before calling `Goal`.
- `monitor(+Boolean)` ACTOR
  Default is `false`, i.e. to not monitor. The node at `URI` must be another ACTOR node.
- `protocol(+Atom)` ACTOR
  If `Atom` is `http` (default), the HTTP protocol will be used as transport, and if `Atom` is `ws`, a WebSocket connection will be used.
- `pid(-Pid)` ACTOR
  The `pid` option is passed with a free variable `Pid` which will be bound to the pid of the remote toplevel when the call returns. Using the `pid` option breaks the abstraction for remote procedure calling, so it should be used with care. Note that if `transport` is `http`, `Pid` will be bound to `anonymous`. The node at `URI` must be another ACTOR node.

Predicate: `promise/3-4` ISOBASE

```
promise(+URI, +Goal, -Ref) is det.
promise(+URI, +Goal, -Ref, +Options) is det.
```

Makes an asynchronous RPC call to node `URI` with `Goal`. This is a type of RPC which does not suspend the caller until the result is computed. Instead, a reference `Ref` is returned, which can later be used by `yield/2-3` to collect the answer. The reference can be viewed as a promise to deliver the answer. Valid options are:

- `template(+Template)`
  `Template` is a term sharing variables with the goal. By default, the template is identical to the goal.
- `offset(+Integer)`
  Collect only the slice of solutions starting from `Integer`. Default is `0`.
- `limit(+Integer)`
  By default, `promise/3-4` requests that *all* solutions to `Goal` be computed and returned as a list of solutions embedded in an answer term of type `success`. By passing the limit option, the length of this list can be restricted to `Integer`.

Predicate: `yield/2-3` ISOBASE

```
yield(+Ref, ?Answer) is det.
yield(+Ref, ?Answer, +Options) is det.
```

Returns the promised answer from a previous call to `promise/3-4`. If the answer is available, it is returned immediately. Otherwise, the calling process is suspended until the answer arrives from the node that was called. Note that this predicate must be called by the same process from which the previous call to `promise/3-4` was made, otherwise it will not return. Valid options:

- `timeout(+IntegerOrFloat)`
  If nothing appears in the current mailbox within `IntegerOrFloat` seconds, the predicate succeeds anyway. Default is no timeout.
- `on_timeout(+Goal)`
  If the timeout occurs, `Goal` is called.

A.4 The stateless HTTP API

In our proposal for an HTTP query API, the URI in a GET request for one or more solutions to a query has the following form:

```
BaseURI/call?goal=G&template=T&offset=O&limit=L&format=F
```

The URI denotes a resource in the form of a (possibly only partial) answer to the goal `G` as given by the node `BaseURI`. The template `T` works as in `findall/3` and the semantics of the `offset` and `limit` parameters are borrowed from SQL and SPARQL, As in these languages they expect integer values, where `offset` defaults to `0` and `limit` to `infinite`. A client may also use a parameter `load_text` in order to send along source code to complement the goal. Responses are returned as Prolog terms or as Prolog variable bindings encoded in JSON.

A response contains a success answer, a failure answer, or an error answer. A success answer contains a list of solutions of the form `T` to `G`, starting at offset `O` and having a length of at most `L`. In addition, an indication whether more solutions may exist is given. By default, answers are rendered as JSON, where the slice of solutions is represented as a list of pairs of the form `{<var>:<value>,..,<var>:<value>}`. A failure answer indicates that no (more) solutions exists, and an error answer signals an error and carries an error message.

[TODO: Needs more work!]

A.5 The stateful WebSocket API

[TODO: Needs work!]