Intro to Web Prolog for Erlangers

Torbjörn Lager

Department of Philosophy, Linguistics and Theory of Science
University of Gothenburg
Sweden

Why Prolog? What can we do with Prolog that we cannot easily do with Erlang?

How about symbolic AI?

- Logic-based knowledge representation and reasoning
- Meta-interpreters for e.g. expert systems
- Parsing and generation with DCG — a unification-based grammar formalism
- Problem solving of various kinds — perhaps using CLP.
- ... and much more.

With Web Prolog, all this can be done in a concurrent and distributed manner – on the Web!

Pure Prolog is a subset of FOPL

```
p(X) :- q(X), r(X).

q(a).
q(b).  r(b).
q(c).  r(c).

?- p(X).
```

$$\forall x[q(x) \wedge r(x) \rightarrow p(x)]$$

$$q(a)$$
$$q(b) \quad r(b)$$
$$q(c) \quad r(c)$$

$$\vdash \exists x[p(x)]$$

Pure Prolog is a subset of FOPL

```
p(X) :- q(X), r(X).

q(a).
q(b).  r(b).
q(c).  r(c).

?- p(X).
X = b
```

$$\forall x[q(x) \wedge r(x) \rightarrow p(x)]$$

$$q(a)$$
$$q(b) \quad r(b)$$
$$q(c) \quad r(c)$$

$$\vdash \exists x[p(x)]$$

Pure Prolog is a subset of FOPL

```
p(X) :- q(X), r(X).

q(a).
q(b).  r(b).
q(c).  r(c).
```

$$\forall x[q(x) \wedge r(x) \rightarrow p(x)]$$

$$q(a)$$
$$q(b) \quad r(b)$$
$$q(c) \quad r(c)$$

```
?- p(X).
X = b ;
X = c.

?-
```

$$\vdash \exists x[p(x)]$$

Appending lists in pure Prolog

```
append([],L,L).
append([H|T],L2,[H|L3]) :-
    append(T,L2,L3).
```

```
?- append([a,b],[c,d],Xs).
Xs = [a,b,c,d].

?- append(Xs,Ys,[a,b,c]).
Xs = [], Ys = [a,b,c] ;
Xs = [a], Ys = [b,c] ;
Xs = [a,b], Ys = [c] ;
Xs = [a,b,c], Ys = [] ;
false.

?-
```

Problem solving: the n-queens puzzle

```
:- use_module(library(clpfd)).

queens(N, Queens) :-
    length(Queens, N),
    Queens ins 1..N,
    safe_queens(Queens),
    labeling([ff], Queens).

safe_queens([]).
safe_queens([Queen|Queens]) :-
    safe_queens(Queens, Queen, 1),
    safe_queens(Queens).

safe_queens([], _, _).
safe_queens([Queen|Queens], Queen0, D0) :-
    Queen0 #\= Queen,
    abs(Queen0 - Queen) #\= D0,
    D1 #= D0 + 1,
    safe_queens(Queens, Queen0, D1).
```

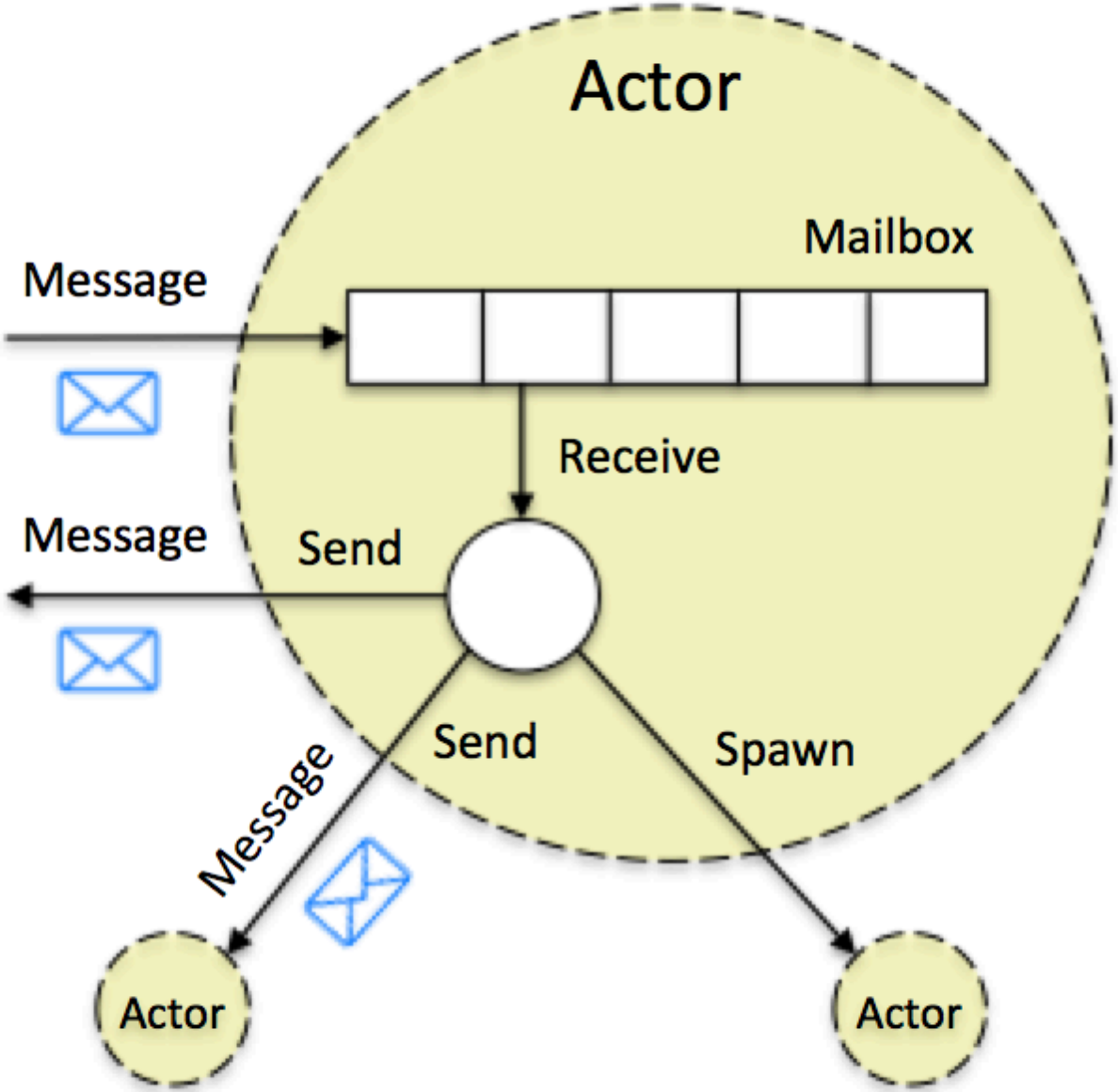
```
?- queens(8, Qs).
Qs = [1, 5, 8, 6, 3, 7, 2, 4] ;
Qs = [1, 6, 8, 3, 7, 4, 2, 5] ;
Qs = [1, 7, 4, 6, 8, 2, 5, 3] ;
Qs = [1, 7, 5, 8, 2, 4, 6, 3] ;
Qs = [2, 4, 6, 8, 3, 1, 7, 5] ;
Qs = [2, 5, 7, 1, 3, 8, 6, 4] ;
Qs = [2, 5, 7, 4, 1, 8, 6, 3] ;
Qs = [2, 6, 1, 7, 4, 8, 3, 5] ;
Qs = [2, 6, 8, 3, 1, 4, 7, 5] ;

. . .
```

```
?- queens(100, Qs).
Qs = [1, 3, 5, 57, 59, 4, 64, 7, 58, 71, 81, 60, 6, 91, 82,
90, 8, 83, 77, 65, 73, 26, 9, 45, 37, 63, 66, 62, 44, 10, 48,
54, 43, 69, 42, 47, 18, 11, 72, 68, 50, 56, 61, 36, 33, 17,
12, 51, 100, 93, 97, 88, 35, 84, 78, 19, 13, 99, 67, 76, 92,
75, 87, 96, 94, 85, 20, 14, 95, 32, 98, 55, 40, 80, 49, 52,
46, 53, 21, 15, 41, 2, 27, 34, 22, 70, 74, 29, 25, 30, 38,
86, 16, 79, 24, 39, 28, 23, 31, 89]
```

Parsing with DCG

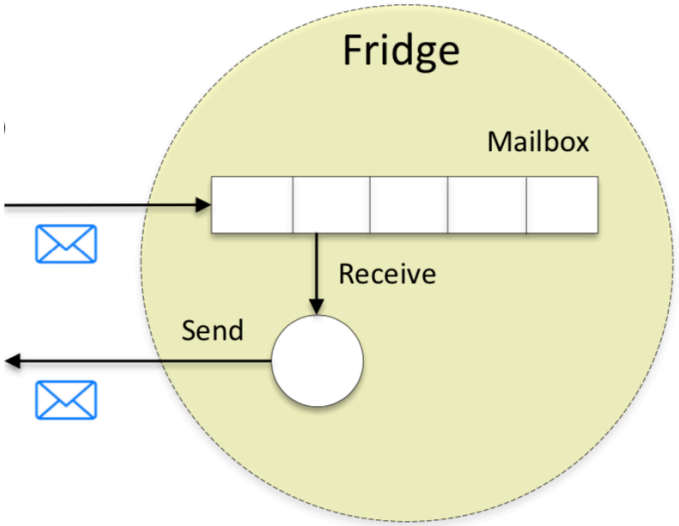
```
s(s(NP,VP)) --> np(NP, Num), vp(VP, Num).

np(NP, Num) --> pn(NP, Num).
np(np(Det,N), Num) --> det(Det, Num), n(N, Num).
np(np(Det,N,PP), Num) --> det(Det, Num), n(N, Num), pp(PP).

vp(vp(V,NP), Num) --> v(V, Num), np(NP, _).
vp(vp(V,NP,PP), Num) --> v(V, Num), np(NP, _), pp(PP).

pp(pp(P,NP)) --> p(P), np(NP, _).

det(det(a), sg) --> [a].

pn(pn(john), sg) --> [john].

n(n(man), sg) --> [man].
n(n(telescope), sg) --> [telescope].

v(v(saw), _) --> [saw].

p(p(with)) --> [with].


?- phrase(s(Tree), [john,saw,a,man,with,a,telescope]).
Tree = s(pn(john),vp(v(saw),np(det(a),n(man),pp(p(with),np(det(a),n(telescope)))))) ;
Tree = s(pn(john),vp(v(saw),np(det(a),n(man)),pp(p(with),np(det(a),n(telescope))))) ;
false.


?-
```

Algorithm = Logic + Control


Actor
Mailbox
Message
Receive
Message
Send
Send
Spawn
Message
Actor
Actor

Fred's fridge simulation example

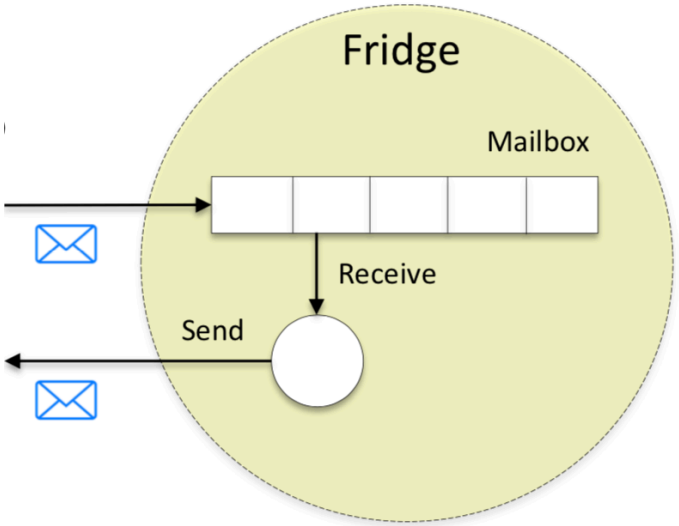
```
fridge(FoodList0) :-
    receive({
        {From, {store, Food}} ->
            self(Self),
            From ! ok(Self),
            fridge([Food|FoodList0]);
        {From, {take, Food}} ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! ok(Self, Food),
                fridge(FoodList)
            ;   From ! not_found(Self),
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```



Inspired by Erlang example from "*Learn You Some Erlang for great good!" by Fred Hébert*

Fred's fridge simulation example

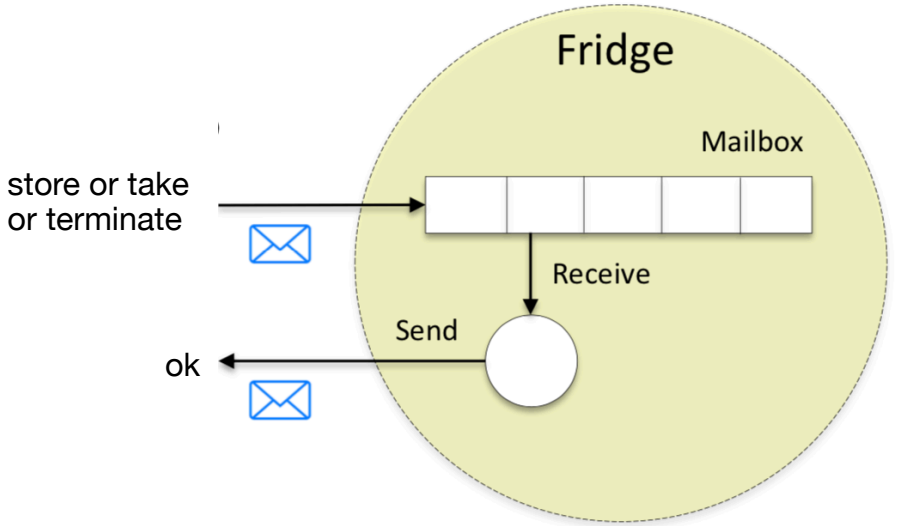
```
fridge(FoodList0) :-
    receive({
        {From, {store, Food}} ->
            self(Self),
            From ! ok(Self),
            fridge([Food|FoodList0]);
        {From, {take, Food}} ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! ok(Self, Food),
                fridge(FoodList)
            ;   From ! not_found(Self),
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```



Inspired by Erlang example from "*Learn You Some Erlang for great good!" by Fred Hébert*

Fred's fridge simulation example

```
fridge(FoodList0) :-
    receive({
        store(From, Food) ->
            self(Self),
            From ! ok(Self),
            fridge([Food|FoodList0]);
        take(From, Food) ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! ok(Self, Food),
                fridge(FoodList)
            ;   From ! not_found(Self),
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```



Inspired by Erlang example from "*Learn You Some Erlang for great good!*" by Fred Hébert

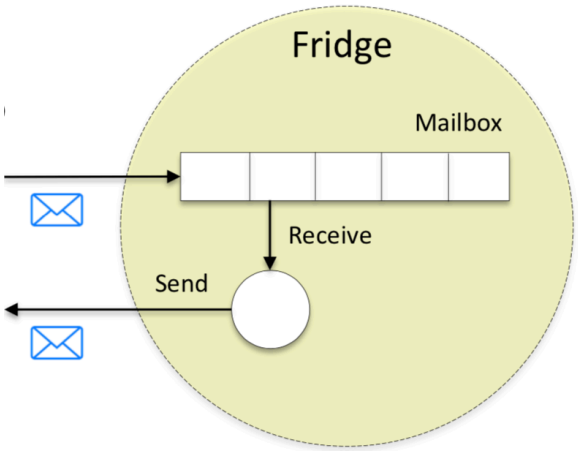
Reading the code was fun — I had to do a double take — was I reading Erlang or Prolog — they often look pretty much the same.

Joe Armstrong (p.c. June 18, 2018)

Fred's fridge simulation example

```
fridge(FoodList0) :-
    receive({
        store(From, Food) ->
            self(Self),
            From ! ok(Self),
            fridge([Food|FoodList0]);
        take(From, Food) ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! ok(Self, Food),
                fridge(FoodList)
            ;   From ! not_found(Self),
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```

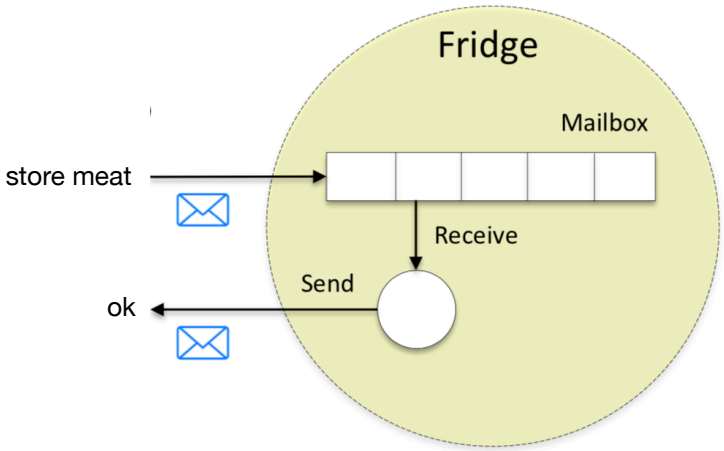
```
?- spawn(fridge([]), Pid, [
        node('http://fridge.org'),
        monitor(true)
   ]).
Pid = 6734313@'http://fridge.org'.

?-
```



Fred's fridge simulation example

```
fridge(FoodList0) :-
    receive({
        store(From, Food) ->
            self(Self),
            From ! ok(Self),
            fridge([Food|FoodList0]);
        take(From, Food) ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! ok(Self, Food),
                fridge(FoodList)
            ;   From ! not_found(Self),
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```

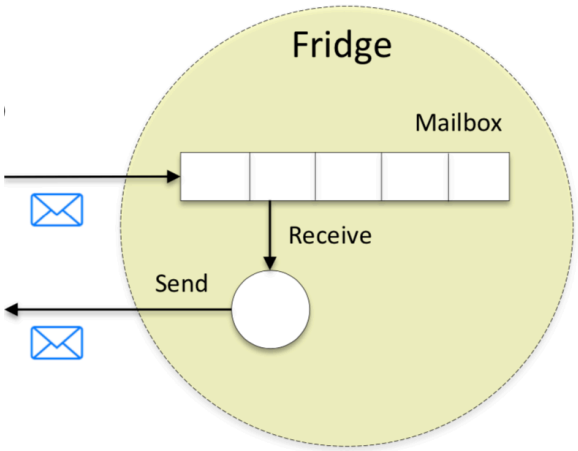
```
?- spawn(fridge([]), Pid, [
        node('http://fridge.org'),
        monitor(true)
   ]).
Pid = 6734313@'http://fridge.org'.

?- self(Me), $Pid ! store(Me, meat),
   $Pid ! store(Me, cheese).
Me = 9201673@'http://local.org'.

?-
```



Fred's fridge simulation example

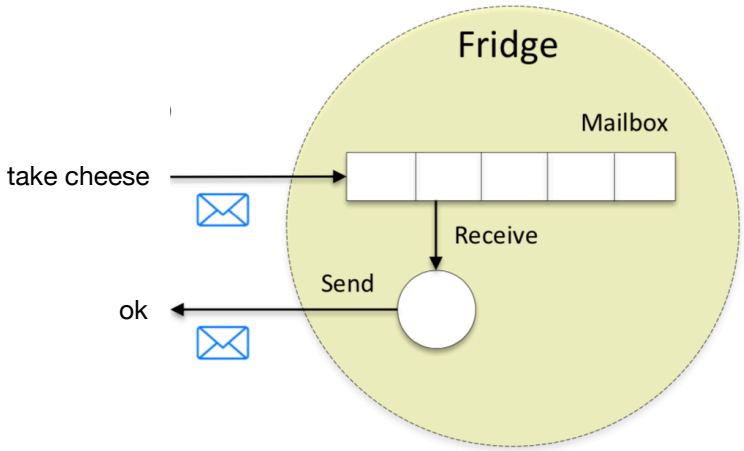
```
fridge(FoodList0) :-
    receive({
        store(From, Food) ->
            self(Self),
            From ! ok(Self),
            fridge([Food|FoodList0]);
        take(From, Food) ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! ok(Self, Food),
                fridge(FoodList)
            ;   From ! not_found(Self),
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```

```
?- spawn(fridge([]), Pid, [
        node('http://fridge.org'),
        monitor(true)
   ]).
Pid = 6734313@'http://fridge.org'.

?- self(Me), $Pid ! store(Me, meat),
   $Pid ! store(Me, cheese).
Me = 9201673@'http://local.org'.

?- flush.
Shell got ok(6734313@'http://fridge.org')
Shell got ok(6734313@'http://fridge.org')
true.

?-
```



Fred's fridge simulation example

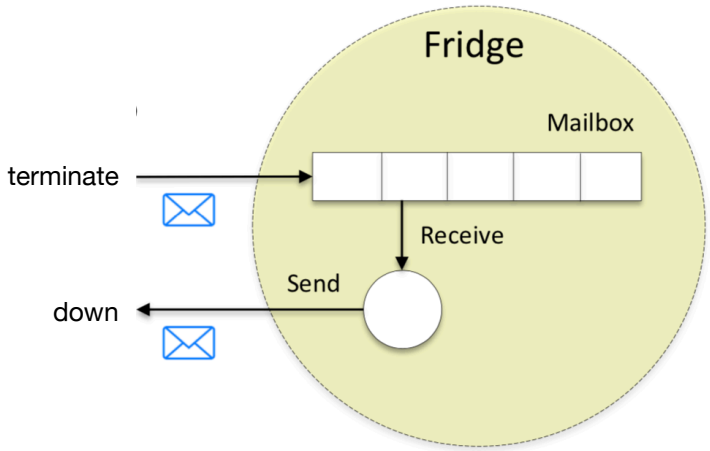
```
fridge(FoodList0) :-
    receive({
        store(From, Food) ->
            self(Self),
            From ! ok(Self),
            fridge([Food|FoodList0]);
        take(From, Food) ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! ok(Self, Food),
                fridge(FoodList)
            ;   From ! not_found(Self),
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```




```
?- spawn(fridge([]), Pid, [
        node('http://fridge.org'),
        monitor(true)
   ]).
Pid = 6734313@'http://fridge.org'.

?- self(Me), $Pid ! store(Me, meat),
   $Pid ! store(Me, cheese).
Me = 9201673@'http://local.org'.

?- flush.
Shell got ok(6734313@'http://fridge.org')
Shell got ok(6734313@'http://fridge.org')
true.

?- $Pid ! take($Me, cheese).
true.

?- flush.
Shell got ok(6734313@'http://fridge.org',cheese)
true.

?-
```

Fred's fridge simulation example

```
fridge(FoodList0) :-
    receive({
        store(From, Food) ->
            self(Self),
            From ! ok(Self),
            fridge([Food|FoodList0]);
        take(From, Food) ->
            self(Self),
            (   select(Food, FoodList0, FoodList)
            ->  From ! ok(Self, Food),
                fridge(FoodList)
            ;   From ! not_found(Self),
                fridge(FoodList0)
            );
        terminate ->
            true
    }).
```




```
?- spawn(fridge([]), Pid, [
        node('http://fridge.org'),
        monitor(true)
   ]).
Pid = 6734313@'http://fridge.org'.

?- self(Me), $Pid ! store(Me, meat),
   $Pid ! store(Me, cheese).
Me = 9201673@'http://local.org'.

?- flush.
Shell got ok(6734313@'http://fridge.org')
Shell got ok(6734313@'http://fridge.org')
true.

?- $Pid ! take($Me, cheese).
true.

?- flush.
Shell got ok(6734313@'http://fridge.org',cheese)
true.

?- $Pid ! terminate.
true.

?- flush.
Shell got down(6734313@'http://fridge.org',exit).
true.

?-
```

Ping-pong example

Erlang

```
-module(tut15).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_Pid) ->
    Pong_Pid ! finished,
    io:format('Ping finished');
ping(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
    receive
        pong ->
            io:format('Ping received pong')
    end,
    ping(N - 1, Pong_Pid).

pong() ->
    receive
        finished ->
            io:format('Pong finished');
        {ping, Ping_Pid} ->
            io:format('Pong received ping'),
            Ping_Pid ! pong,
            pong()
    end.

start() ->
    Pong_Pid = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_Pid]).
```

Web Prolog

```
ping(0, Pong_Pid) :-
    Pong_Pid ! finished,
    io:format('Ping finished').
ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong ->
            io:format('Ping received pong')
    }),
    N1 is N - 1,
    ping(N1, Pong_Pid).

pong :-
    receive({
        finished ->
            io:format('Pong finished’);
        ping(Ping_Pid) ->
            io:format('Pong received ping'),
            Ping_Pid ! pong,
            pong
    }).

start :-
    spawn(pong, Pong_Pid, [
        % node(‘http://ex.org'),
        src_predicates([pong/0])
    ]),
    spawn(ping(3, Pong_Pid), _, [
        src_predicates([ping/2])
    ]).
```

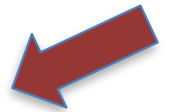
Ping-pong example

Erlang

```
-module(tut15).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_Pid) ->
    Pong_Pid ! finished,
    io:format("ping finished");
ping(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong")
    end,
    ping(N - 1, Pong_Pid).

pong() ->
    receive
        finished ->
            io:format("Pong finished");
        {ping, Ping_Pid} ->
            io:format("Pong received ping"),
            Ping_Pid ! pong,
            pong()
    end.

start() ->
    Pong_Pid = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_Pid]).
```

Web Prolog

```
ping(0, Pong_Pid) :-
    Pong_Pid ! finished,
    io:format('Ping finished').
ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong ->
            io:format('Ping received pong')
    }),
    N1 is N - 1,
    ping(N1, Pong_Pid).

pong :-
    receive({
        finished ->
            io:format('Pong finished’);
        ping(Ping_Pid) ->
            io:format('Pong received ping'),
            Ping_Pid ! pong,
            pong
    }).

start :-
    spawn(pong, Pong_Pid, [
        node(‘http://remote.org’),
        src_predicates([pong/0])
    ]),
    spawn(ping(3, Pong_Pid), _, [
        src_predicates([ping/2])
    ]).
```



Handling non-determinism

```
?- self(Self),
   spawn(( p(X),
           Self ! X,
           receive({
               next -> fail;
               stop -> Self ! stopped
           })
         ), Pid).
Pid = 1035231,
Self = 3122459@'http://local.org'.

?-
```

```
% Program loaded


p(a).
p(b).
p(c).
```

Handling non-determinism

```
?- self(Self),
   spawn(( p(X),
           Self ! X,
           receive({
               next -> fail;
               stop -> Self ! stopped
           })
         ), Pid).
Pid = 1035231,
Self = 3122459@'http://local.org'.

?- flush.
Shell got a
true.

?-
```

```
% Program loaded


p(a).
p(b).
p(c).
```

Handling non-determinism

```
?- self(Self),
   spawn(( p(X),
           Self ! X,
           receive({
               next -> fail;
               stop -> Self ! stopped
           })
         ), Pid).
Pid = 1035231,
Self = 3122459@'http://local.org'.

?- flush.
Shell got a
true.

?- $Pid ! next.
true.

?-
```

```
% Program loaded


p(a).
p(b).
p(c).
```

Handling non-determinism

```
?- self(Self),
   spawn(( p(X),
           Self ! X,
           receive({
               next -> fail;
               stop -> Self ! stopped
           })
         ), Pid).
Pid = 1035231,
Self = 3122459@'http://local.org'.

?- flush.
Shell got a
true.

?- $Pid ! next.
true.

?- flush.
Shell got b
true.

?-
```

```
% Program loaded


p(a).
p(b).
p(c).
```

Handling non-determinism

```
?- self(Self),
   spawn(( p(X),
           Self ! X,
           receive({
               next -> fail;
               stop -> Self ! stopped
           })
         ), Pid).
Pid = 1035231,
Self = 3122459@'http://local.org'.

?- flush.
Shell got a
true.

?- $Pid ! next.
true.

?- flush.
Shell got b
true.

?- $Pid ! stop,
   receive({Message -> true}).
Message = stopped.

?-
```

```
% Program loaded


p(a).
p(b).
p(c).
```

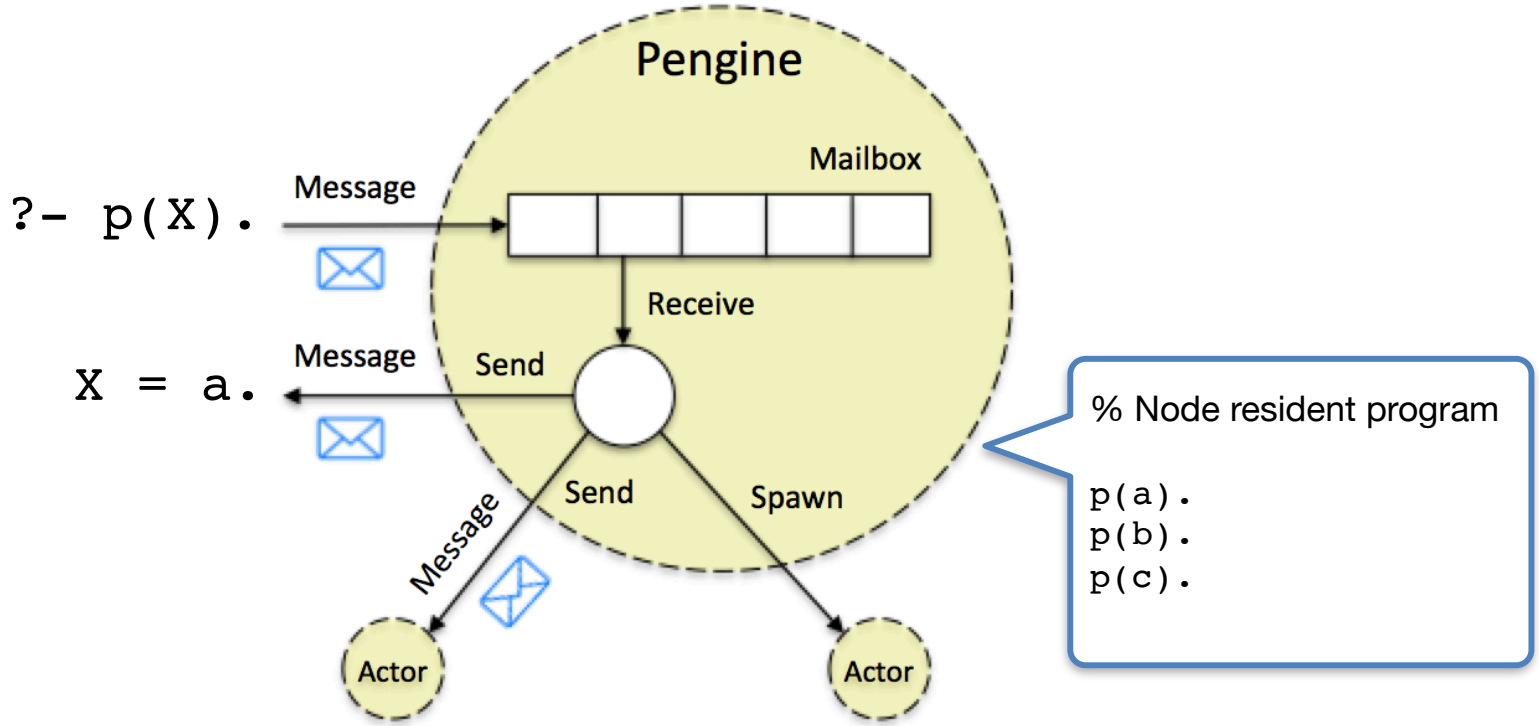
Handling non-determinism

```
?- self(Self),
   spawn(( p(X),
           Self ! X,
           receive({
               next -> fail;
               stop -> Self ! stopped
           })
         ), Pid).
Pid = 1035231,
Self = 3122459@'http://local.org'.

?- flush.
Shell got a
true.

?- $Pid ! next.
true.

?- flush.
Shell got b
true.

?- $Pid ! stop,
   receive({Message -> true}).
Message = stopped.

?-
```

`receive/1-2` is **semi-deterministic** (i.e. either fails or succeeds exactly once), so calling `fail/0` here causes it to fail and triggers backtracking

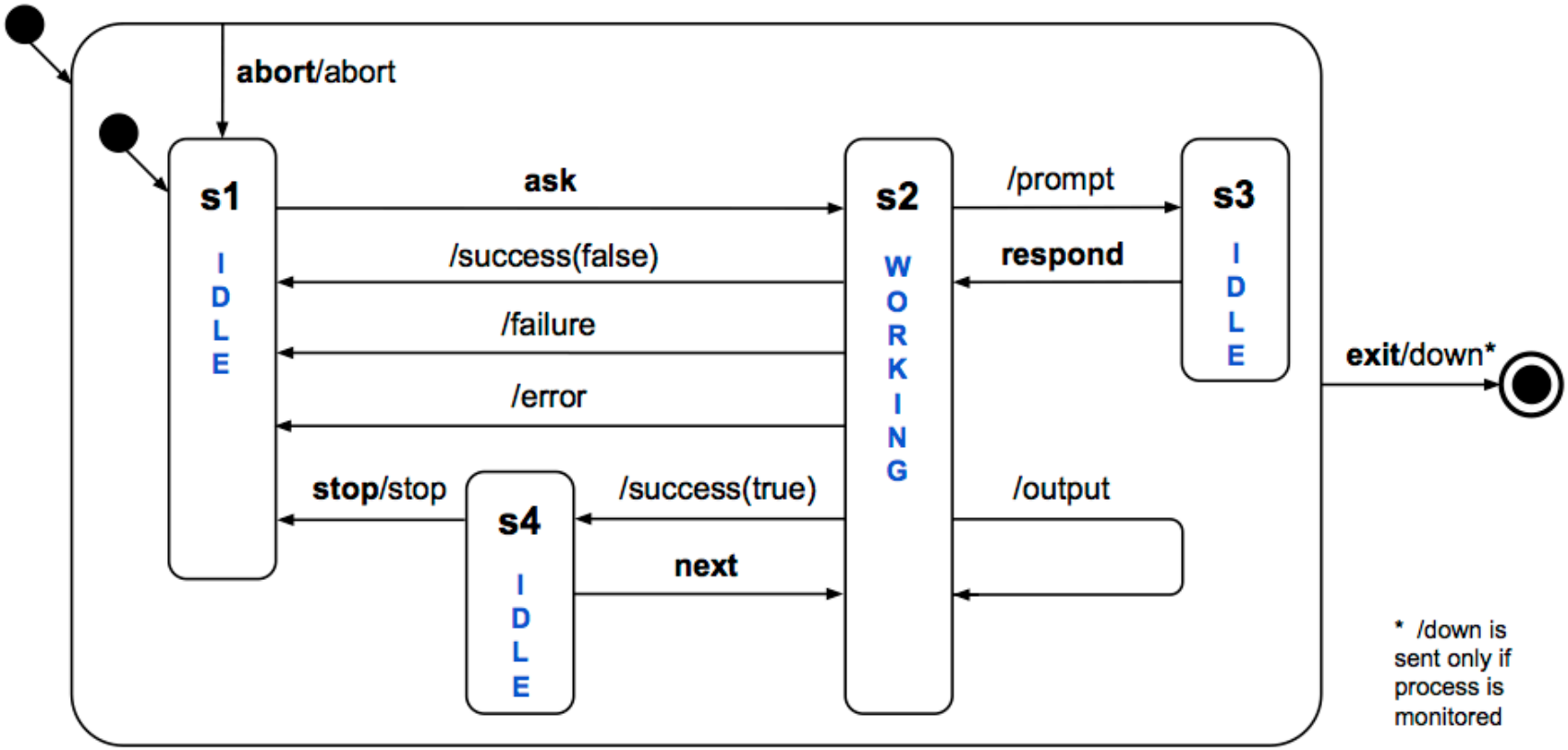
The pengine (Prolog engine)

- A pengine sends back answers, or other forms of messages, to a parent asking queries or issuing commands.



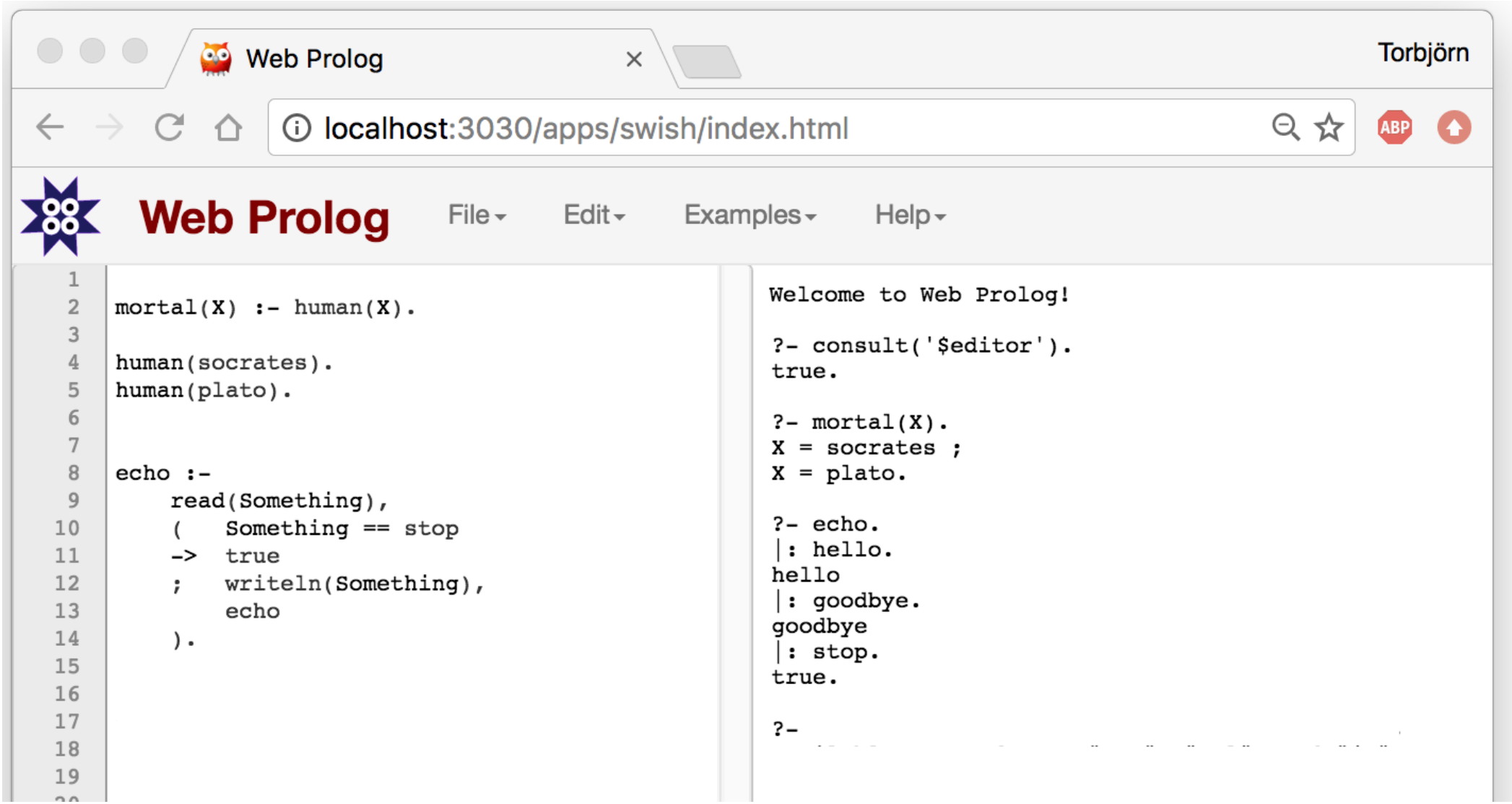
The Pengine Communication Protocol

- A pengine is special *kind* of actor, characterised by its protocol – the Pengine Communication Protocol (PCP)



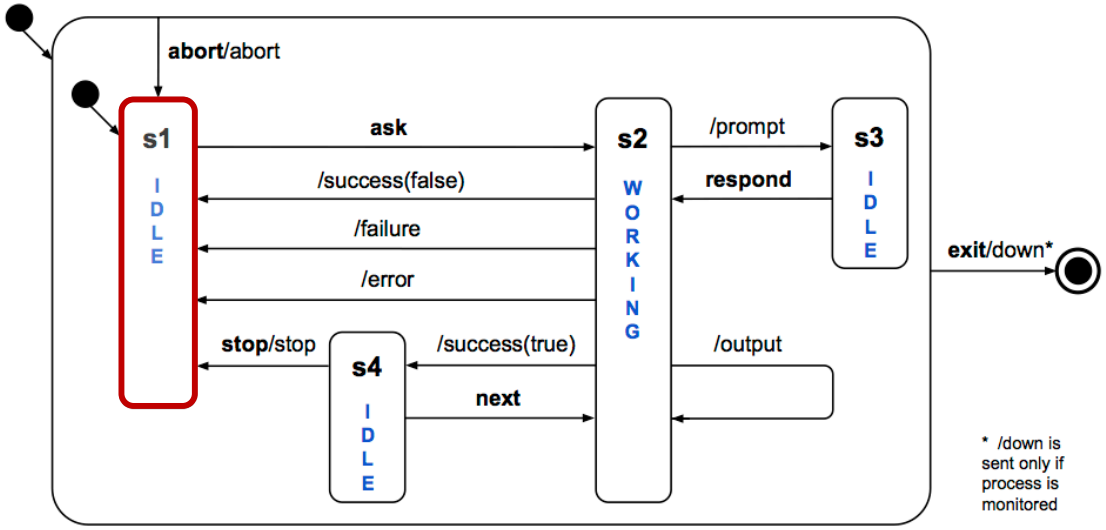
The proof-of-concept IDE

See `https://github.com/Web-Prolog`



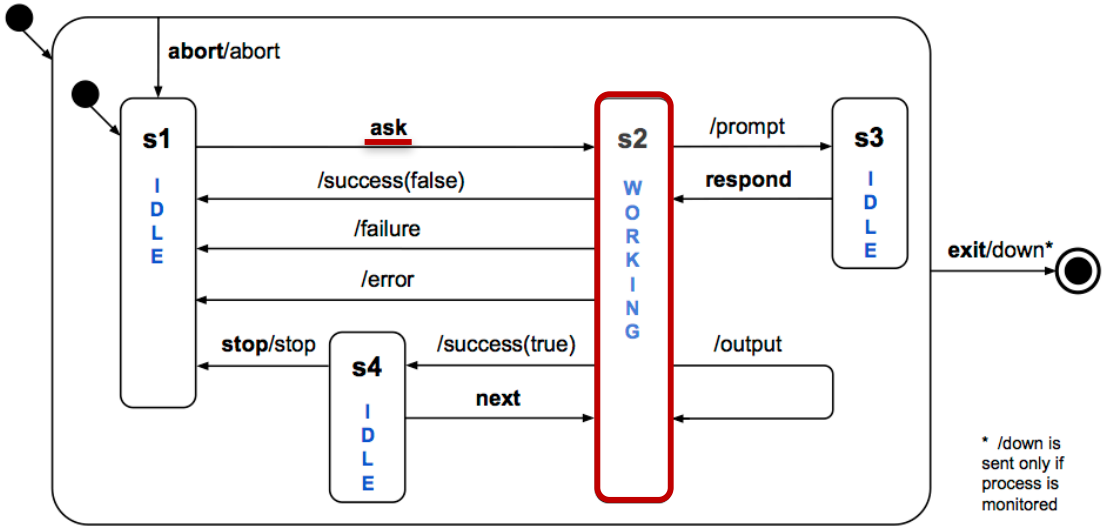
Working with pengines

```
?- pengine_spawn(Pid, [
        node('http://remote.org'),
        monitor(true)
    ]).
Pid = 752872@'http://remote.org'.

?-
```



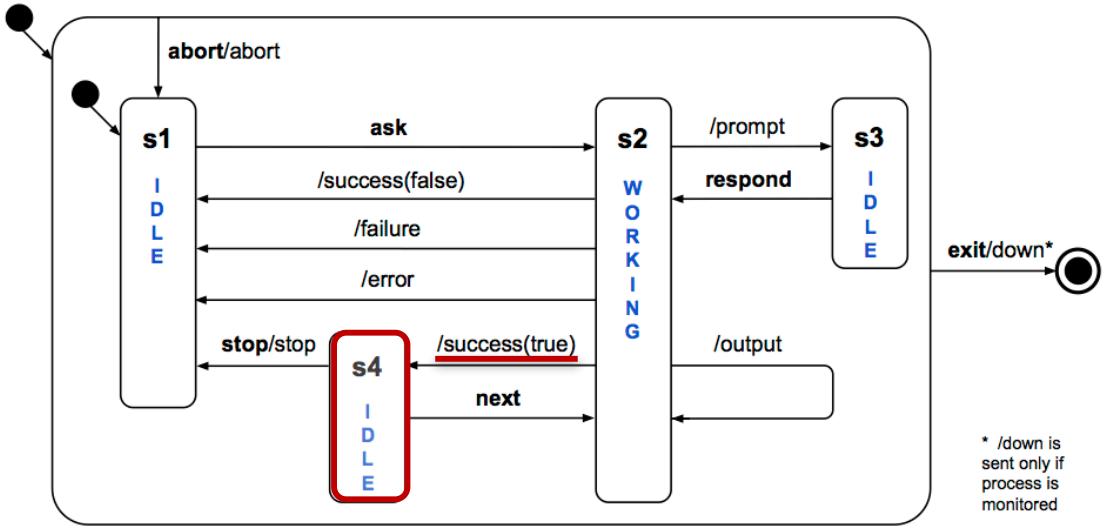
Working with pengines

```
?- pengine_spawn(Pid, [
       node('http://remote.org'),
       monitor(true)
   ]).
Pid = 752872@'http://remote.org'.

?- pengine_ask($Pid, p(X), [
       template(X)
   ]).
true.

?-
```



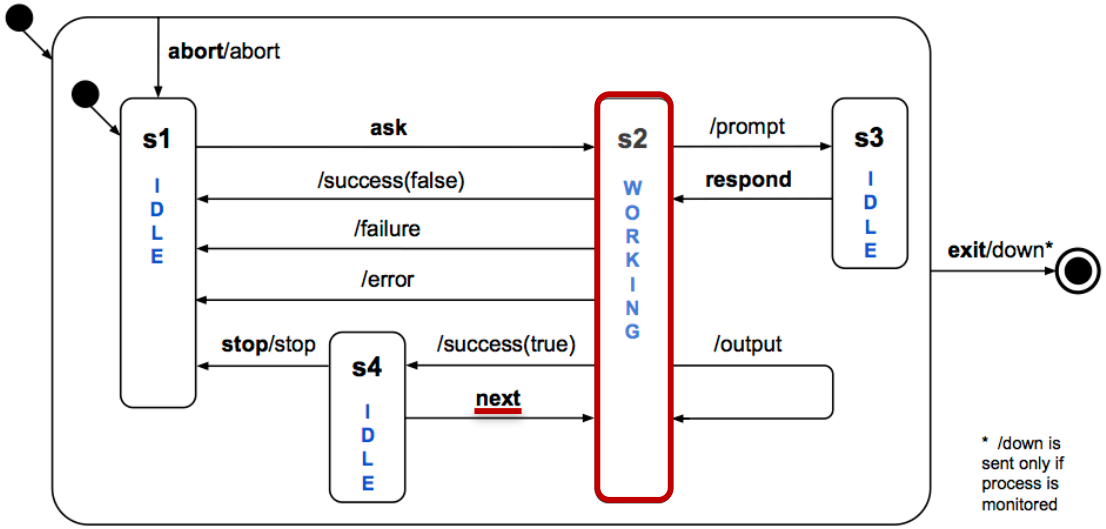
Working with pengines

```
?- pengine_spawn(Pid, [
        node('http://remote.org'),
        monitor(true)
    ]).
Pid = 752872@'http://remote.org'.

?- pengine_ask($Pid, p(X), [
        template(X)
    ]).
true.

?- flush.
Shell got success(752872@'http://remote.org',[a], true)
true.

?-
```



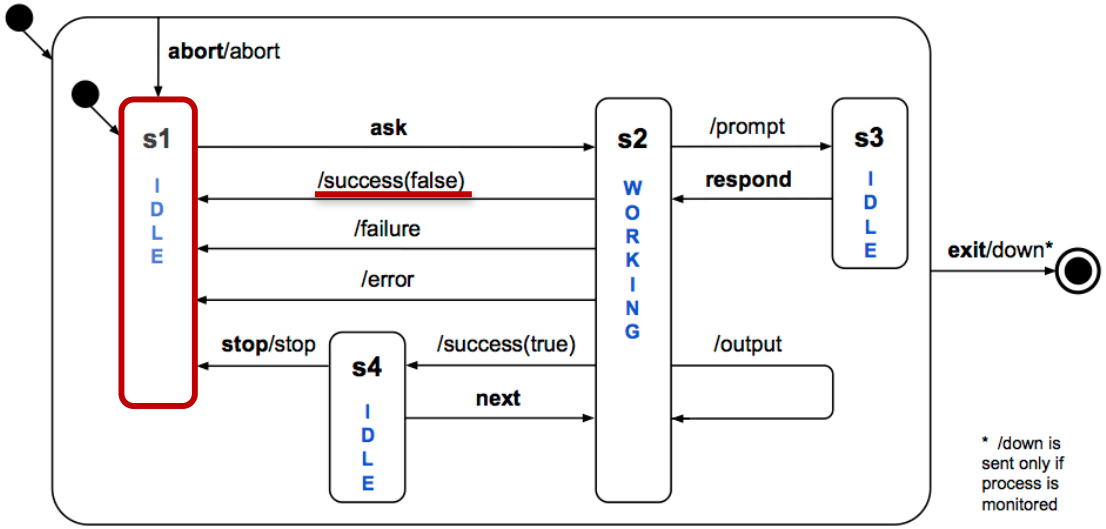
Working with pengines

```
?- pengine_spawn(Pid, [
        node('http://remote.org'),
        monitor(true)
    ]).
Pid = 752872@'http://remote.org'.

?- pengine_ask($Pid, p(X), [
        template(X)
    ]).
true.

?- flush.
Shell got success(752872@'http://remote.org',[a], true)
true.

?- pengine_next($Pid, [
        limit(10)
    ]).
true.

?-
```



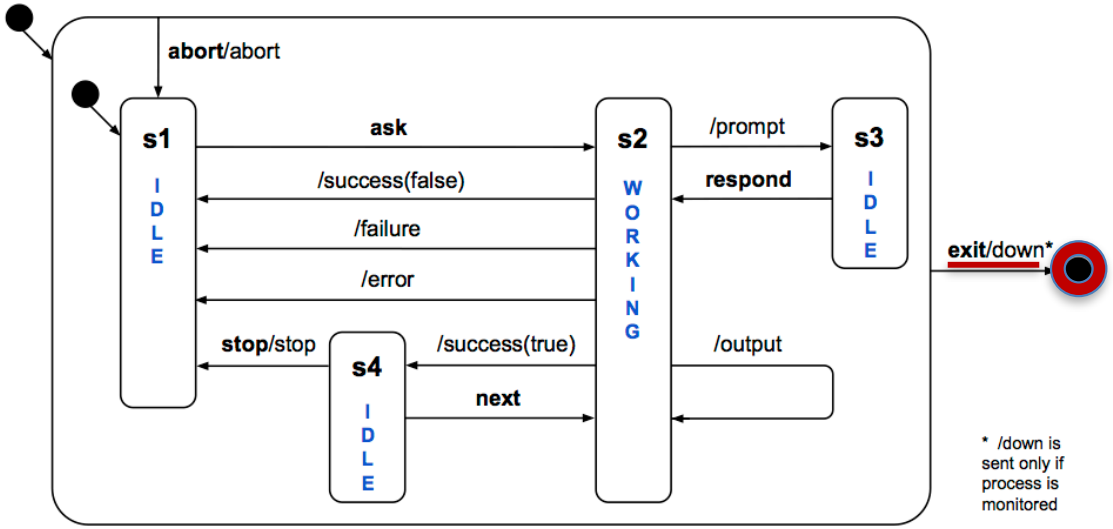
Working with pengines

```
?- pengine_spawn(Pid, [
       node('http://remote.org'),
       monitor(true)
   ]).
Pid = 752872@'http://remote.org'.

?- pengine_ask($Pid, p(X), [
       template(X)
   ]).
true.

?- flush.
Shell got success(752872@'http://remote.org',[a], true)
true.

?- pengine_next($Pid, [
       limit(10)
   ]).
true.

?- flush.
Shell got success(752872@'http://remote.org',[b,c], false)
true.

?-
```



Working with pengines

```
?- pengine_spawn(Pid, [
       node('http://remote.org'),
       monitor(true)
   ]).
Pid = 752872@'http://remote.org'.

?- pengine_ask($Pid, p(X), [
       template(X)
   ]).
true.

?- flush.
Shell got success(752872@'http://remote.org',[a], true)
true.

?- pengine_next($Pid, [
       limit(10)
   ]).
true.

?- flush.
Shell got success(752872@'http://remote.org',[b,c], false)
true.

?- exit($Pid, goodbye).
true.

?- flush.
Shell got down(752872@'http://remote.org', goodbye)
true.
```



Non-deterministic RPC (NDRPC)

- The `rpc/2-3` predicate allows a process running in a node A to call and try to solve a query in the Prolog context of another node B, taking advantage of the data and programs being offered by B, just as if they were local to A.

- Very simple - no concurrency involved.

Exemplifying NDRPC

```
?- rpc('http://remote.org', p(X)).
X = a ;
X = b ;
X = c.
```

Has to make three network round trips

```
?- rpc('http://remote.org', p(X), [
        limit(10)
   ]).
X = a ;
X = b ;
X = c.
```

Needs only one network round trip

```
?- rpc('http://remote.org', (p(X),q(X)), [
        src_text("q(a). q(b).")
   ]).
X = a ;
X = b.
```

Inject code

```
?- rpc(localnode, p(X), [
        src_uri('http://remote.org/src')
   ]).
X = a ;
X = b ;
X = c.
```

Runs locally, with code fetched from a URI. (Needs only one network round trip)

How is NDRPC implemented?

```
rpc(URI, Query, Options) :-
    pengine_spawn(Pid, [
         node(URI),
         exit(true),
         monitor(false)
       | Options
    ]),
    pengine_ask(Pid, Query, Options),
    wait_answer(Query, Pid).

wait_answer(Query, Pid) :-
    receive({
        failure(Pid) -> fail;
        error(Pid, Exception) ->
            throw(Exception);
        success(Pid, Solutions, true) ->
            (   member(Query, Solutions)
            ;   pengine_next(Pid),
                wait_answer(Query, Pid)
            );
        success(Pid, Solutions, false) ->
            member(Query, Solutions)
    }).
```

Programming abstractions

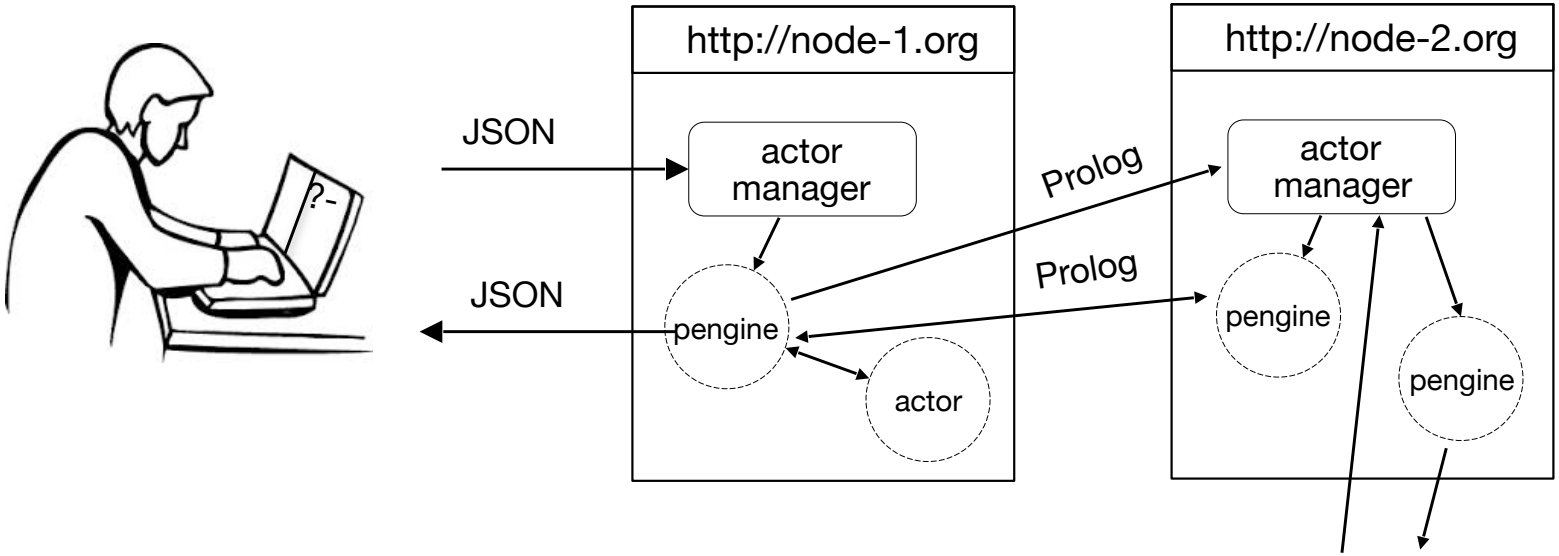
- Actors — the loci of computation in Web Prolog
- Pengines — actors adhering to the PCP protocol
- Non-deterministic remote procedure calls (NDRPC)

They are all related!

Communicating Prolog engines is a great idea — this is more or less what Erlang started as — but I didn't like the idea of backtracking over nodes.

Joe Armstrong (p.c. June 16, 2018)

In conversation with the Prolog Web

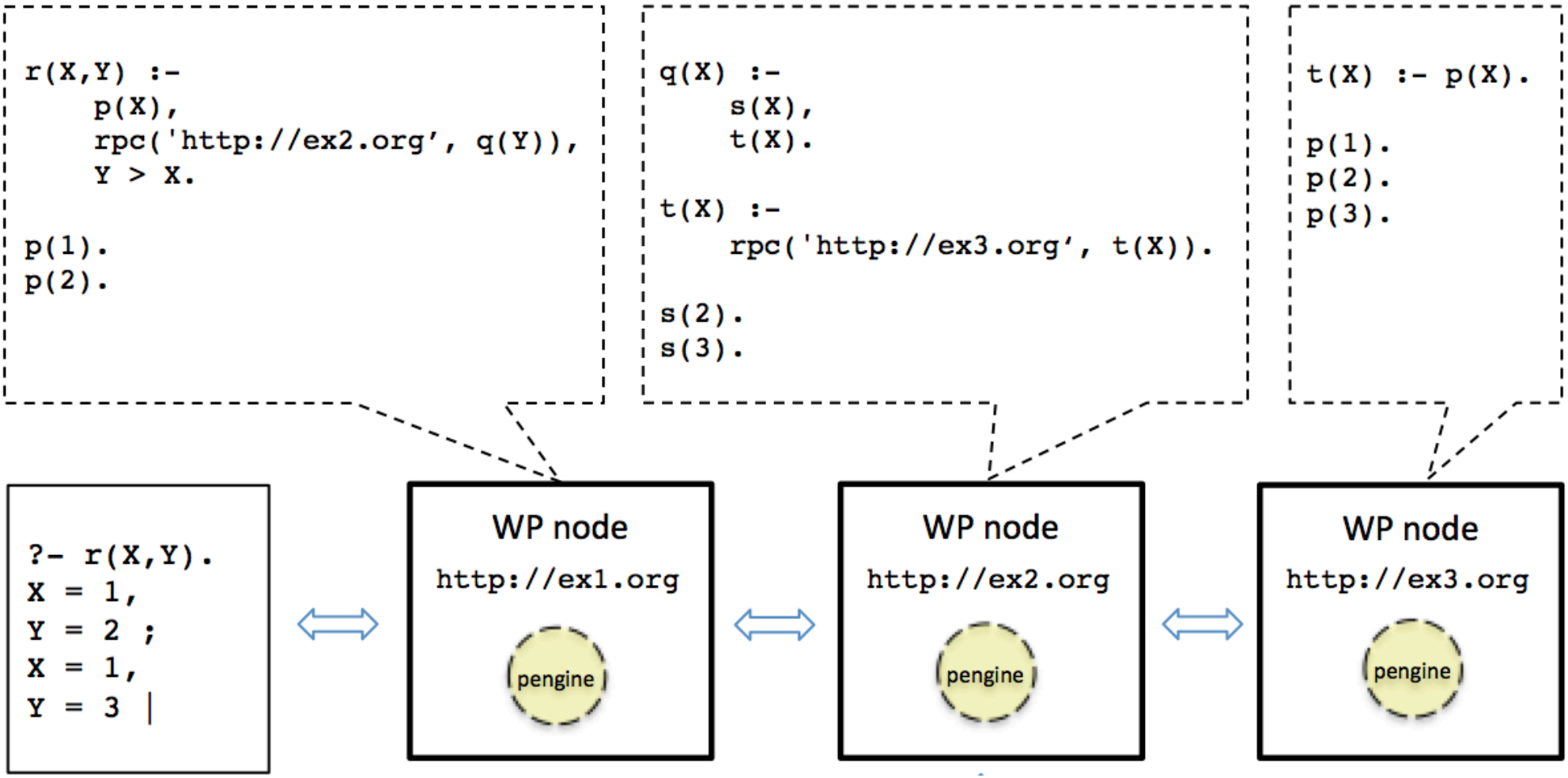



```
$ curl http://node-2.org/api?query=q(X)&offset=1&limit=10&format=json
{ "type":"success",
  "data”:[{"X":3},{"X”:8}],
  "more":false
}
```

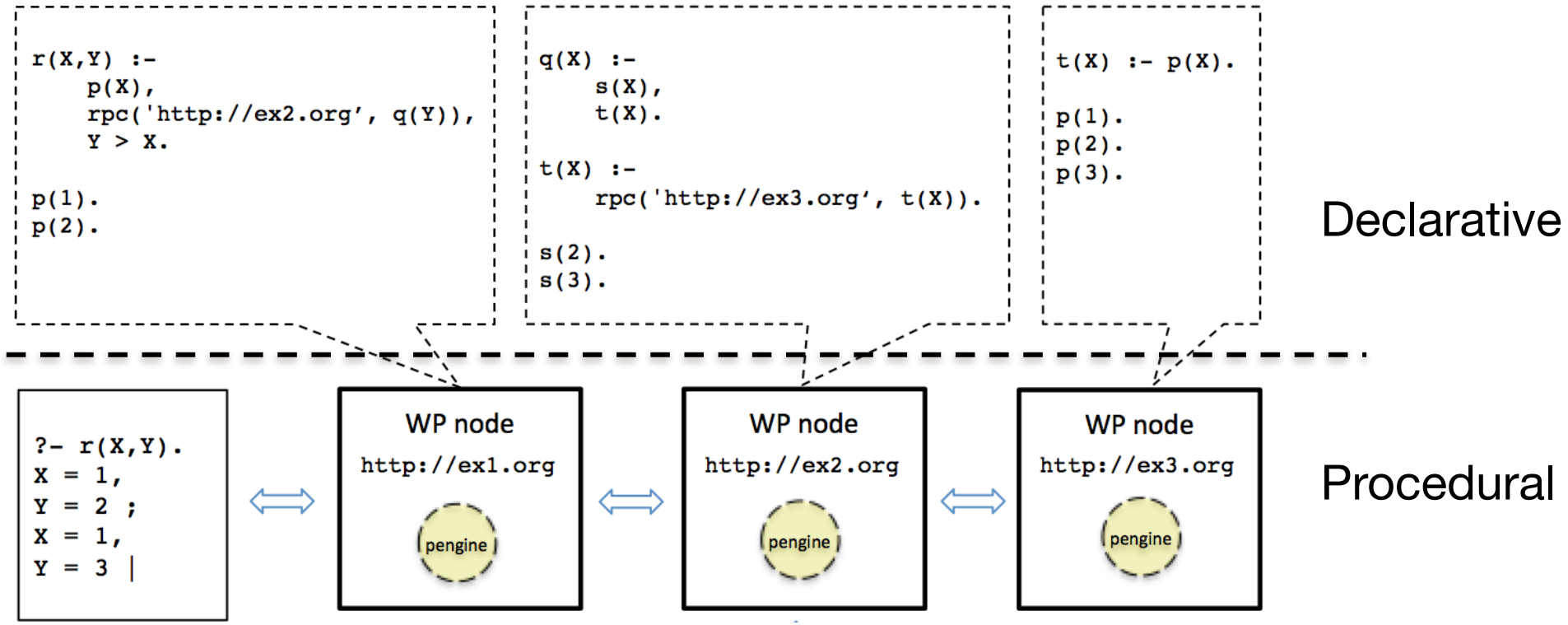
Using the stateless HTTP API

```
GET http://remote.org/ask?query=append(Xs,Ys,[a,b,c])&limit=2
{ "type":"success",
  "pid":"anonymous",
  "data":[{"Xs":[], "Ys":["a","b","c"]},{"Xs":["a"], "Ys":["b","c"]}],
  "more":true
}

GET http://remote.org/ask?query=append(Xs,Ys,[a,b,c])&offset=2&limit=2
{ "type":"success",
  "pid":"anonymous",
  "data":[{"Xs":["a","b"], "Ys":["c"]},{"Xs":["a","b","c"], "Ys":[]}],
  "more":false
}
```

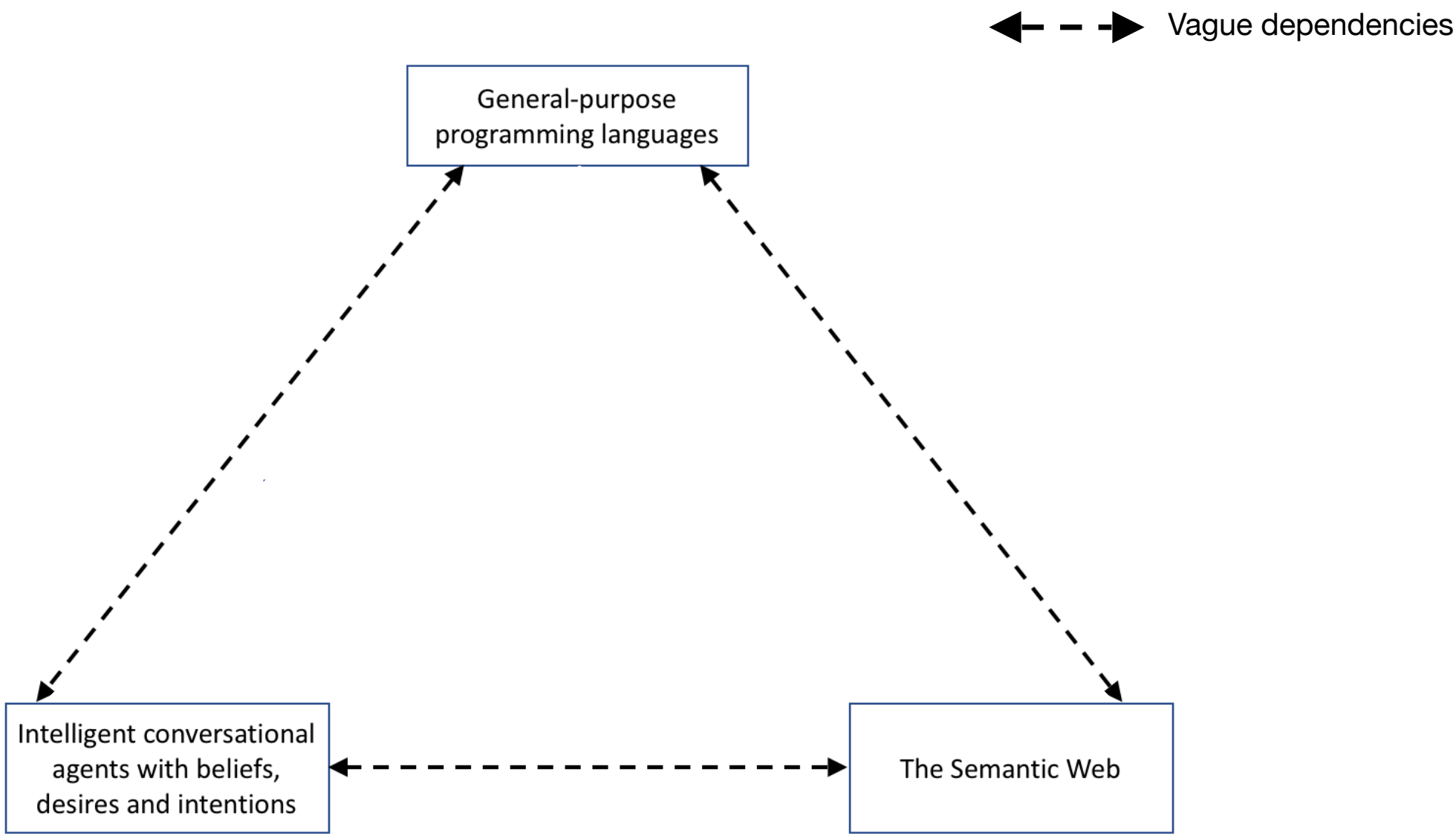
Querying the *pure* Prolog Web



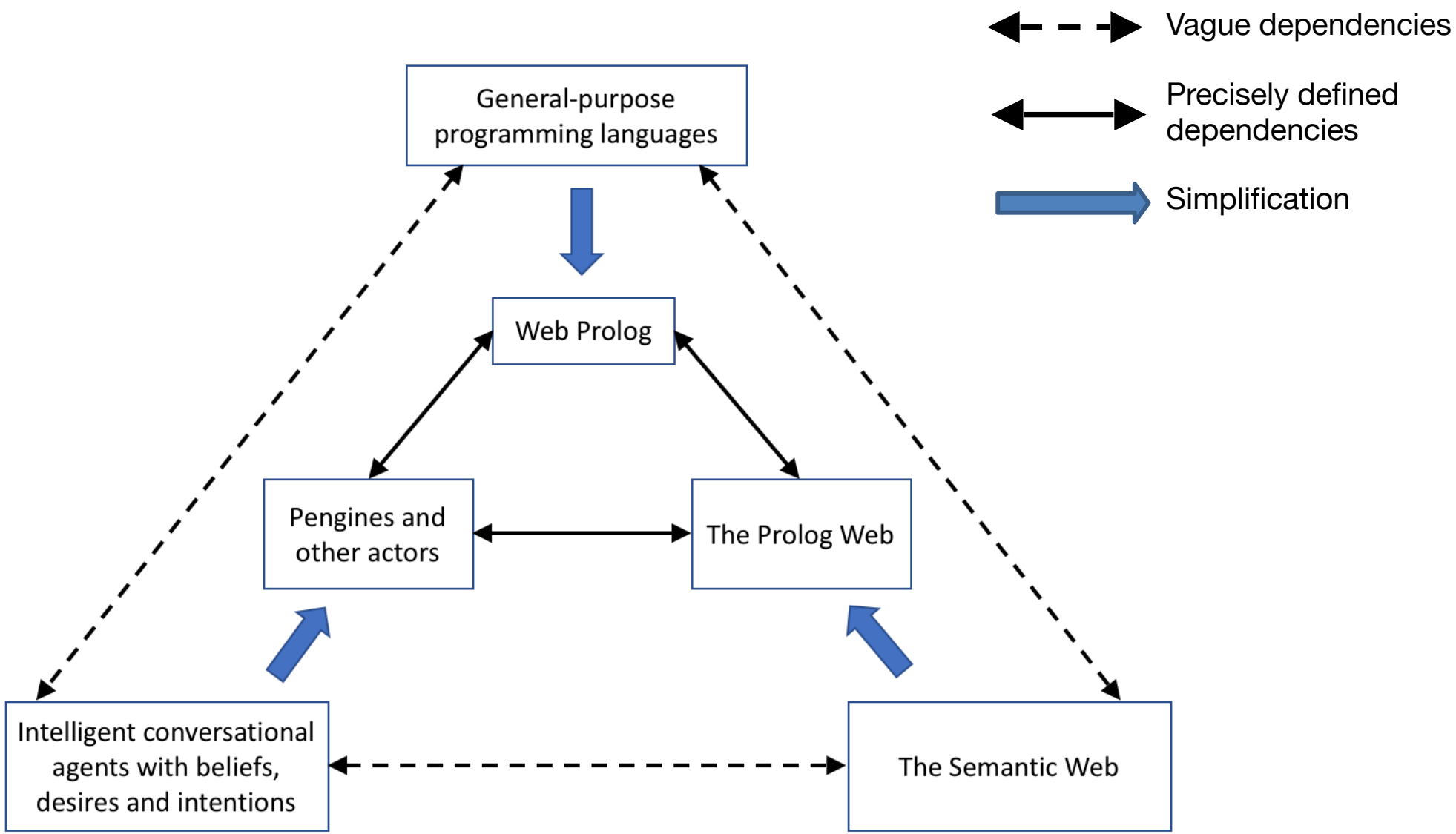
Two levels of abstraction



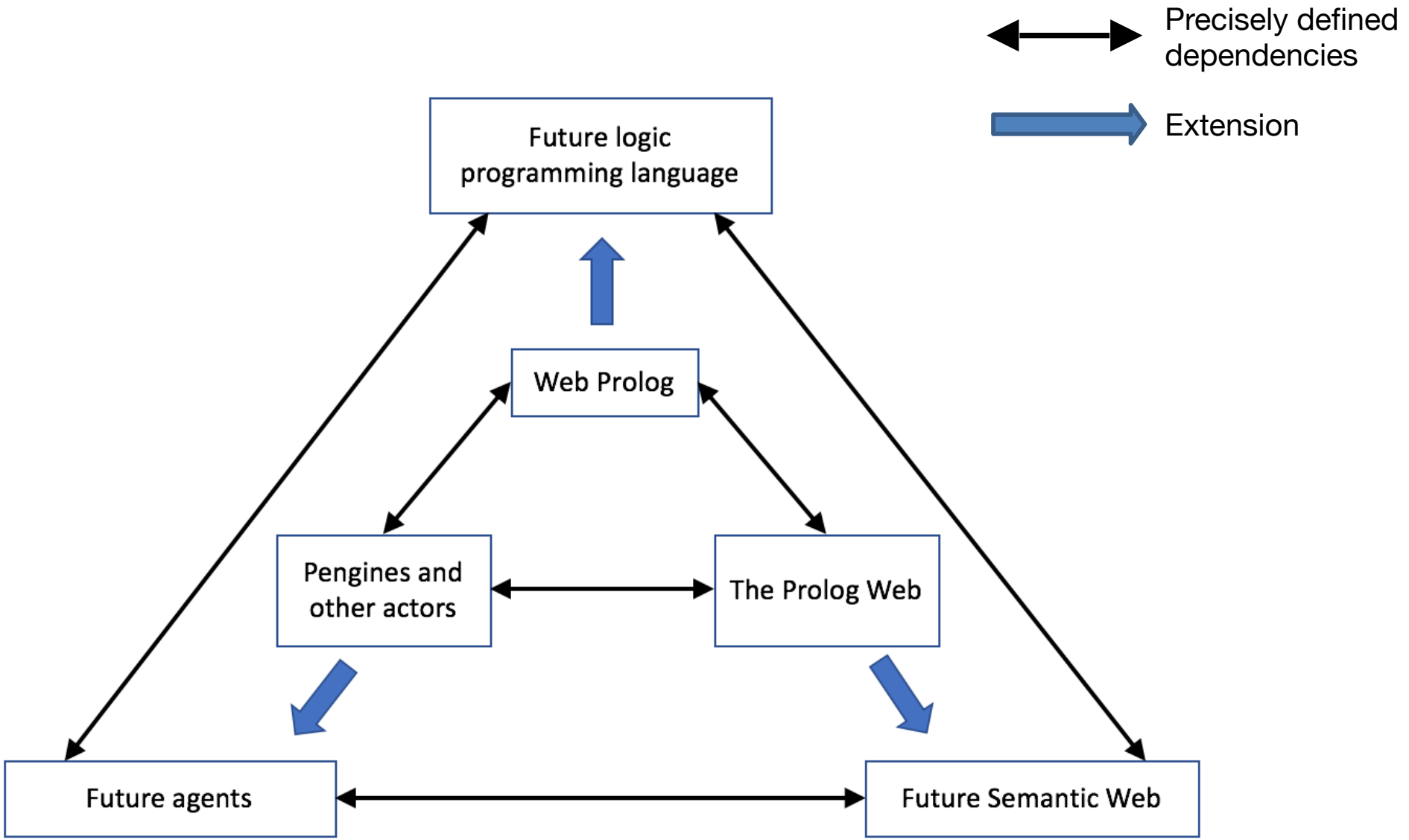
The big picture



The big picture



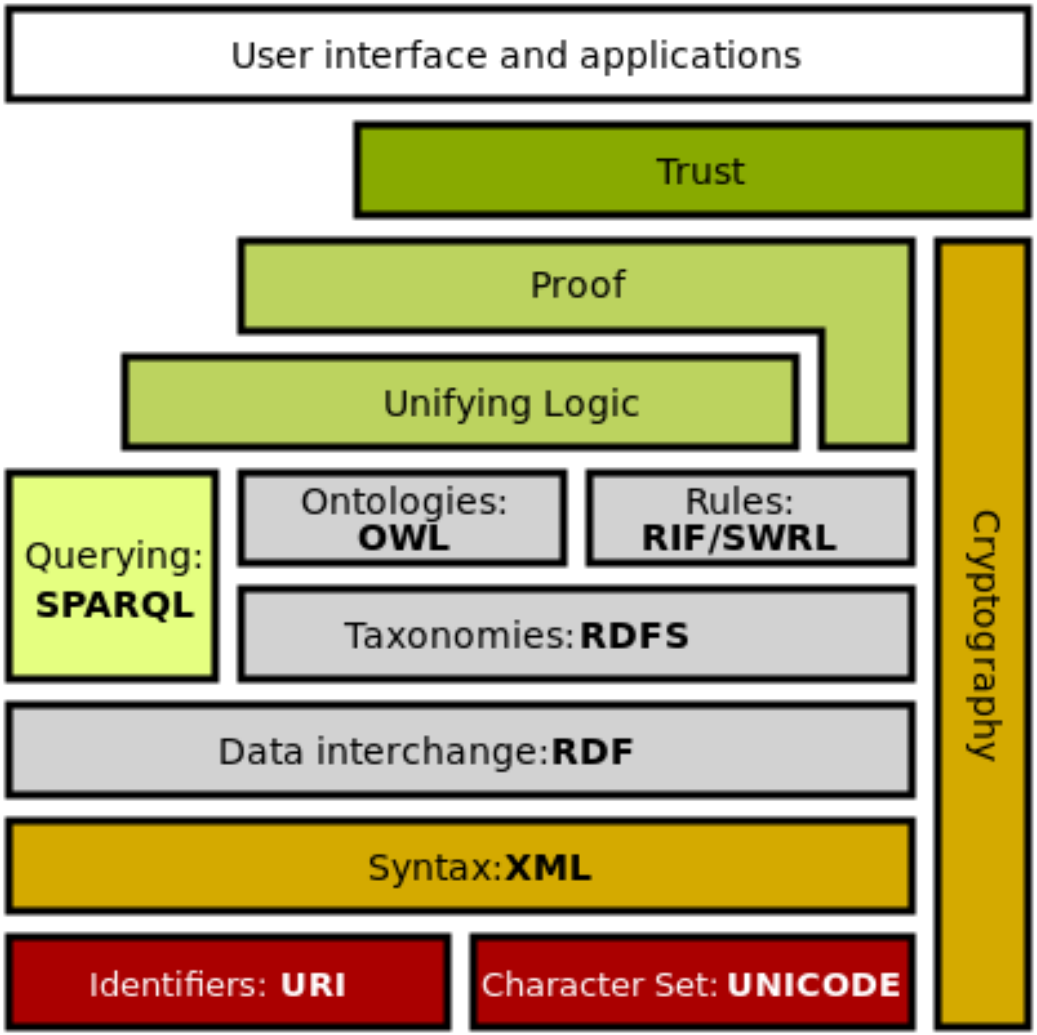
The big picture in a potential future



Web Prolog “killer application” areas

- Semantic web programming
- Agents - intelligent conversational agents in particular

Compare with the Semantic Web



Comparing with the Semantic Web

The Prolog Web

- Same language for knowledge representation and querying
- Web Prolog is a programming language
- The Prolog Web comes with an architecture

The Semantic Web

- Different languages for knowledge representation and querying
- Semantic Web languages are not programming languages
- The Semantic Web doesn't come with an architecture

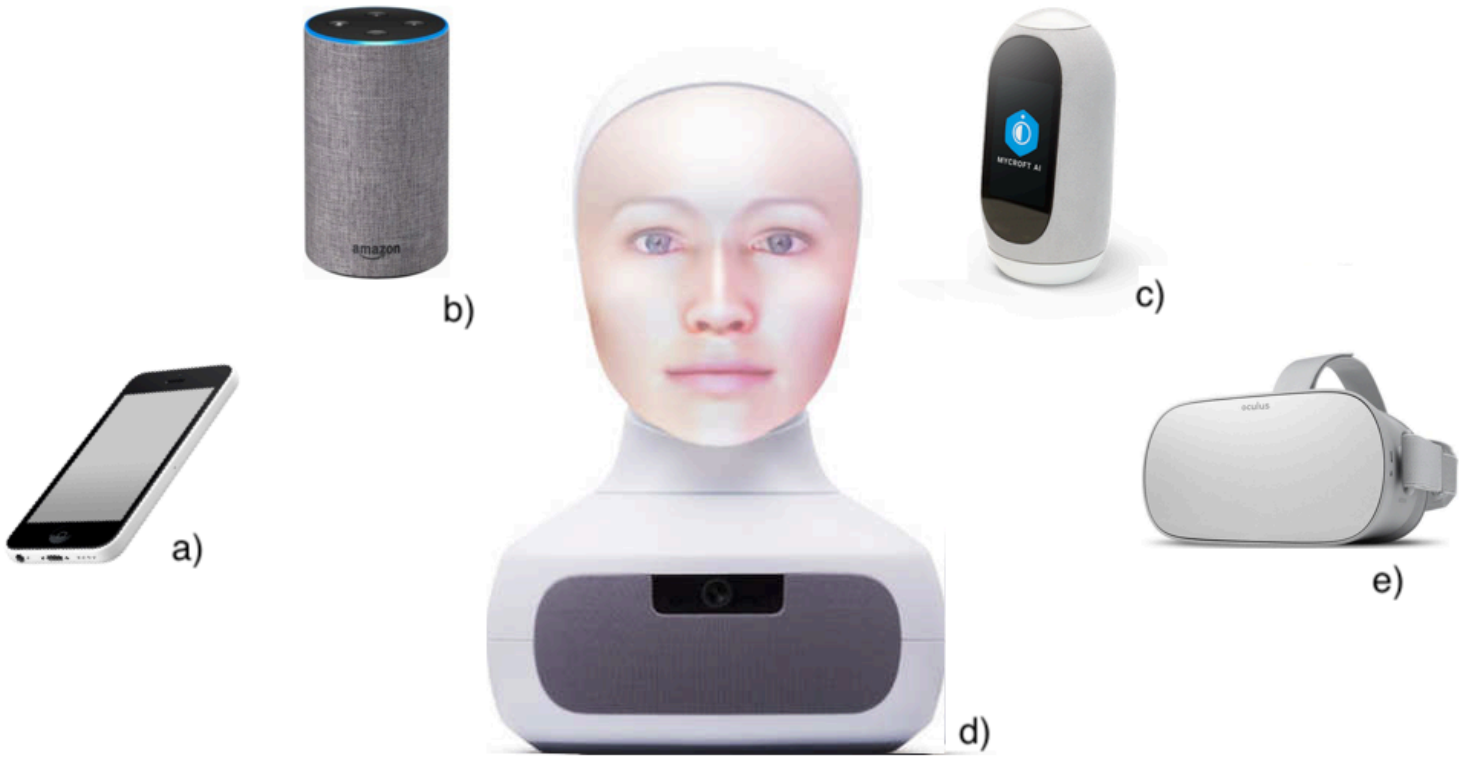
Towards a Semantic Web logic programming language

- Wielemaker et al. 2016 - Cliopatria: A prolog infrastructure for the semantic web:

  > “We do not consider SPARQL adequate for creating rich semantic web applications. SPARQL often needs additional application logic that is located near the data to provide a task-specific API that drives the user interface. Locating this logic near the data is required to avoid protocol and latency overhead. *RDF-based application logic is a perfect match for Prolog and the RDF data is much easier queried through the Prolog RDF libraries than through SPARQL.*”

- Avoids impedance mismatch

Intelligent conversational agents



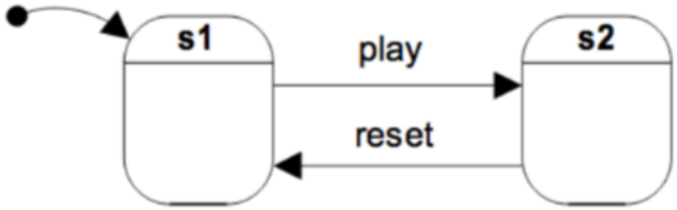
They seem to need **knowledge representation** and **reasoning, natural language processing** as well as **means for fine-grained real-time interaction** …

Algorithm = Logic + Control

Intelligent conversation =

Logic + Even more control

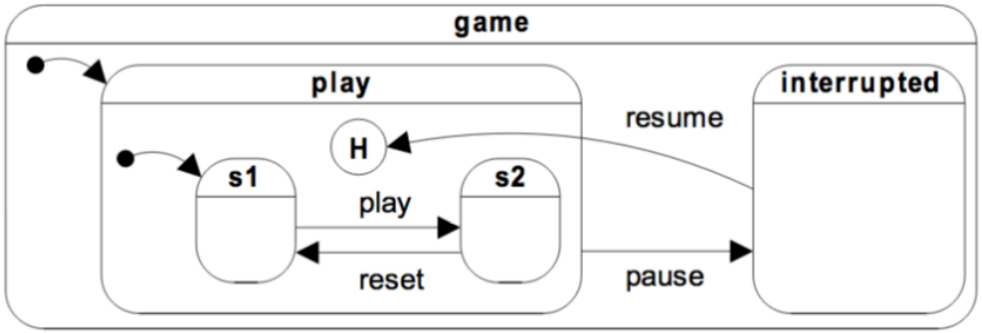
Running a statechart actor



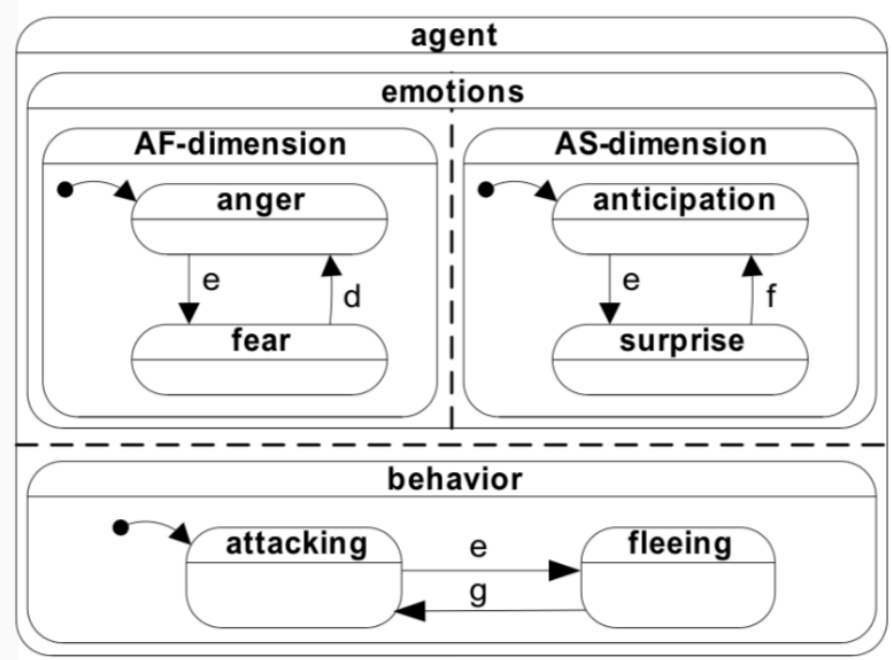

```
<scxml datamodel="web-prolog" initial="s1">
    <state id="s1">
        <onentry>return('IDLE')</onentry>
        <transition event="play" target="s2"/>
    </state>
    <state id="s2">
       <onentry>return('PLAYING')</onentry>
       <transition event="reset" target="s1"/>
    </state>
</scxml>
```

```
?- spawn(run([]), Pid, [
       type(scxml),
       src_uri('http://src.org/game.scxml'),
       monitor(true)
   ]).
Pid = 341293.

?- flush.
Shell got 'IDLE'
true.

?- $Pid ! play.
true.

?- receive({What -> true}).
What = 'PLAYING.

?- $Pid ! reset.
true.

?- receive({What -> true}).
What = ‘IDLE’.
?-
```

Hierarchy and parallelism




```
<scxml datamodel="web-prolog" initial="play">
    <state id="play" initial="s1">
        <history id="h">
            <transition target="s1"/>
        </history>
        <state id="s1">
            <transition event="play" target="s2"/>
        </state>
        <state id="s2">
            <transition event="reset" target="s1"/>
        </state>
        <transition event="pause" target="interrupted"/>
    </state>
    <state id="interrupted">
        <transition event="resume" target="h"/>
    </state>
</scxml>
```




```
<scxml datamodel="web-prolog" initial="agent">
    <parallel id="agent">
        <parallel id="emotions">
            <state id="AF-dimension">
                <state id="anger">
                    <transition event="e" target="fea
                </state>
                <state id="fear">
                    <transition event="d" target="ang
                </state>
            </state>
            <state id="AS-dimension">
```

Collaboration?

Things we have in common

- Actor-based programming
- A lot of syntax and programming patterns
- Some of the technology behind the BEAM

WAM, JAM, BAM and BEAM

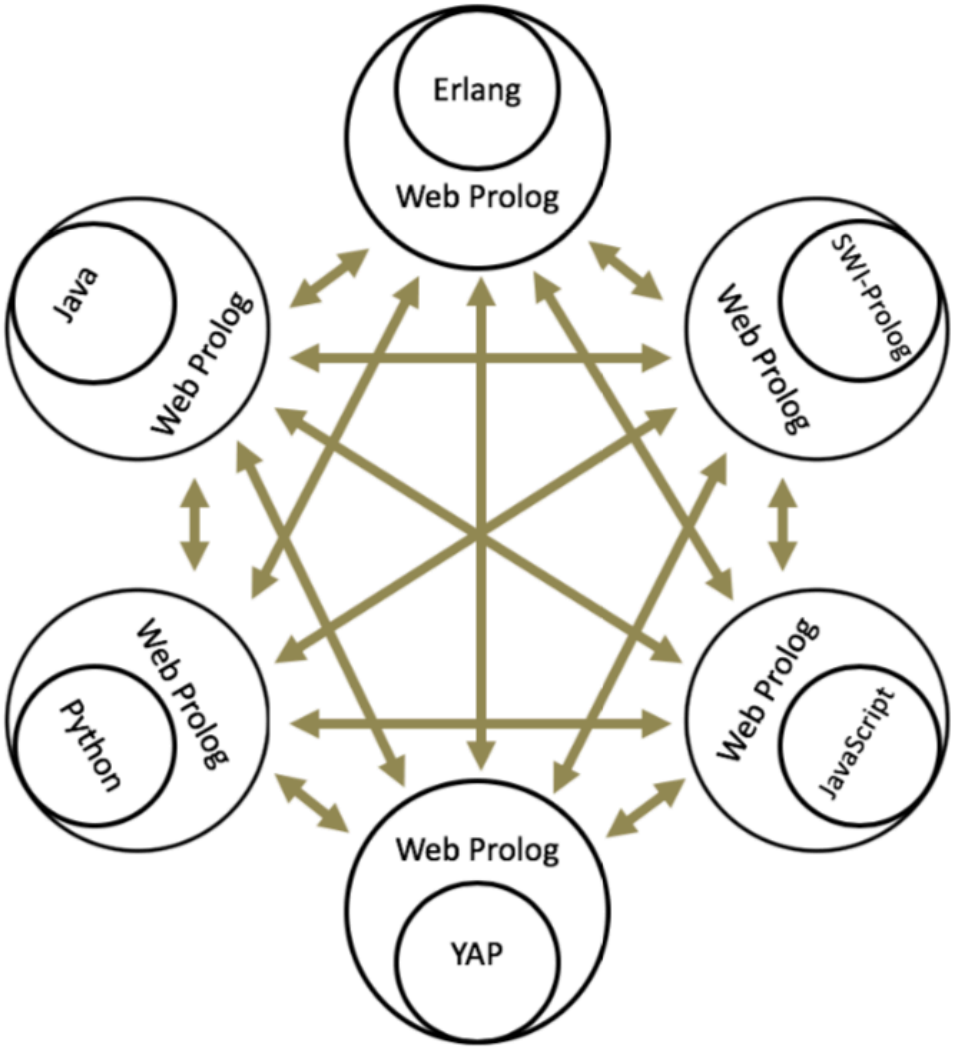
- **The paper:** The holy grail for a Web Prolog runtime system is a compiler targeting a *virtual machine with BEAM-like properties*, capable of producing code which when run will create processes as small and efficient as Erlang processes, yet with the useful capabilities that Prolog offers. *We do not dare to guess whether building such a virtual machine is feasible.*

- **Richard O'Keefe:** I do! It is!

  Let's face it, processes in Logix were *tiny* compared with Erlang ones. And the BEAM was a reaction to JAM, which was inspired by the WAM, and Aquarius Prolog used the BAM which had some similarities to BEAM.

Standardisation of Web Prolog

Web Prolog as a *lingua franca*



A proposal for the Prolog and Erlang communities

2022

Celebrate Prolog's 50th anniversary in 2022 —
Marry Prolog with Erlang and spawn Web Prolog!

A proposal for the Prolog and Erlang communities

2022

Celebrate Prolog's 50th anniversary in 2022 —
Marry Prolog with Erlang and spawn Web Prolog!

Joe Armstrong 1950 - 2019

And honour
the inventors!

Alain Colmerauer 1941 - 2017

Rebranding Prolog

- as a **web logic programming language**
- as a dialect of Prolog based on **a subset of the ISO Prolog core extended with primitives inspired by Erlang**
- as a dialect of Erlang (*sic*!) with Prolog-like capabilities
- as a language serving as a ***lingua franca*** allowing different Prolog platforms running on the Web to **communicate**, **interoperate** and **cooperate** in the service of human users of the **Prolog Web**
- as a language which gives the Prolog community **a Prolog dialect in common**
- as a language that might **appeal to the logic programming community at large**
- as a **new standard** under the auspices of the **W3C**
- as a language targeting a **new application area – spoken (and possibly multi-modal) conversational systems.**

Small signs of success…

- Likes from Joe Armstrong, Markus Triska, and Richard O'Keefe.
- An invitation to join the ISO Prolog standardisation committee.
- O'Keefe wrote:

  "I should tell you that I am liking what I see very much
  and would enjoy using this."

- Interest from the Web world, in the form of Damon Sicore, a former Vice President of Engineering at Mozilla, and Wikimedia Foundation.

Thank you!