Chapter 2
Web Prolog

> Imagine a dialect of Prolog with actors and mailboxes and send and receive – all the means necessary for powerful concurrent and distributed programming. Alternatively, think of it as a dialect of Erlang with logic variables, backtracking search and a built-in database of facts and rules – the means for logic programming, knowledge representation and reasoning. Also, think of it as a web programming language, and as a *lingua franca* for logic-based programming systems, standardised by the W3C. This is what **Web Prolog** is all about.
>
> *Web Prolog – the elevator pitch*

2.1 The essence of Prolog

Based on formal logic, a subject dating all the way back to the antiquity and tried and tested by generations of logicians and philosophers, logic programming forms a paradigm of it own, very different from the imperative or functional programming paradigms. Prolog is generally regarded as the first logic programming language, and is arguably the most important one. Consider the following program:

```
husband(Wife, Husband) :- wife(Husband, Wife).

wife(socrates, xantippa).
wife(aristotle, pythias).
```

We have here a *rule* that translates into the following formula in first-order predicate logic:

$$\forall x \forall y [wife(x, y) \rightarrow husband(y, x)]$$

The rule in combination with the two *facts* of the predicate `wife/2` (that need no translation) allow us to query the predicate `husband/2`. We may for example ask if it is true that Aristotle was the husband of Pythias:

```
?- husband(pythias, aristotle).
true.
?-
```

Or we can ask who was the husband of Pythias:

```
?- husband(pythias, Husband).
Husband = aristotle.
?-
```

Only one solution was found (so it appears that Pythias was not a bigamist, and nor was Aristotle):

```
?- husband(Wife, aristotle).
Wife = pythias.
?-
```

Enumerate the married couples one by one!

```
?- husband(Wife, Husband).
Wife = xantippa, Husband = socrates ;
Wife = pythias, Husband = aristotle.
?-
```

This simple example serves as a reminder that Prolog is not only a logic language, but also a *relational* language that can be used to check if a sentence is true, to look up the value of one argument given the value of another, or to enumerate all pairs of values. When querying a binary relation, these are the modes that exist.

Prolog allows the use of complex terms in the arguments of clauses, here in the form of *lists* in the well-known definition of `append/3`:

```
append([], L, L).
append([H|L1], L2, [H|L]) :- append(L1, L2, L).
```

Complex terms in the arguments of clauses ensures that Prolog is a *Turing complete* programming language. This is what makes it different from a language such as Datalog, which is similar in many ways, but is not Turing complete..

In order to be not only Turing complete but also a practical and efficient programming language, serious Prolog systems need to offer a lot more, and they normally do. In Section 2.1 of *Fifty Years of Prolog and Beyond*, the authors provide a conceptual and minimalist definition of the important features of Prolog, and are thus able to draw a line between what can be considered a Prolog implementation and what can not. Actually, they draw *two* lines, since they distinguish the *essential* features of Prolog from *important* (yet non-essential) features. Below, we reproduce their list of features, where 1-6 are considered essential features and 7-12 less essential. As it turns out, all the essential features except for 2) and 6) are involved when querying the relation between the married couples, or the relation between the lists, in the examples given above.

1. Horn clauses with variables in the terms and arbitrarily nested function symbols as the basic knowledge representation means for both programs (a.k.a. knowledge bases) and queries;
2. the ability to manipulate predicates and clauses as terms, so that meta-predicates can be written as ordinary predicates;
3. SLD-resolution (Kowalski, 1974) based on Robinson's principle (1965) and Kowalski's procedural semantics (Kowalski, 1974) as the basic execution mechanism;
4. unification of arbitrary terms which may contain logic variables at any position, both during SLD-resolution steps and as an explicit mechanism (e.g., via the built-in =/2);
5. the automatic depth-first exploration of the proof tree for each logic query;
6. some control mechanism aimed at letting programmers manage the aforementioned exploration;
7. negation as failure (Clark, 1978), and other logic aspects such as disjunction or implication;
8. the possibility to alter the execution context during resolution, via ad-hoc primitives;
9. an efficient way of indexing clauses in the knowledge base, for both the read-only and read-write use cases;
10. the possibility to express definite clause grammars (DCG) and parse strings using them;
11. constraint logic programming (Jaffar and Lassez, 1987) via ad-hoc predicates or specialized rules (Fruhwirth, 2009);
12. the possibility to define custom infix, prefix, or postfix operators, with arbitrary priority and associativity.

One cannot avoid noticing that, as far as we know, no other community in support of a programming language has found itself in the position of having to determine what should be considered a *real* language of this kind. As we found in Chapter 1, the problem with Prolog is that there are so many systems around that can rightly claim to implement it but still are incompatible with each other in important ways. This is what prompted the development of a standard. For almost thirty years now, the core features of Prolog has been standardized by ISO, documented in a report known as ISO/IEC 13211-1:1995.[1]

2.2 Web Prolog in a nutshell

Web Prolog is designed as a superset of a subset of the ISO Prolog core standard. The dialect defined by ISO is well understood, and is reasonably close to most of the dialects used in Prolog textbooks. The ISO Prolog syntax specification as well

[1] https://www.iso.org/obp/ui/#iso:std:iso-iec:13211:-1:ed-1:v1:en

as a large subset of its built-in predicates form an excellent point of departure for our attempt to create a design and a standard for Web Prolog.

The syntax of Web Prolog is exactly as in ISO Prolog, except that three infix operators (`!/2`, `if/2` and `@/2`) that are not in the standard are defined. They can easily be added by means of `op/3`, the ISO Prolog predicate allowing a programmer to define prefix, postfix or infix operators and specify their precedences. This means that syntactically well-formed Web Prolog programs can always be *read* (but not necessarily *run*) by a conforming implementation of ISO/IEC 13211-1:1995. It is easy to imagine circumstances where this might come in handy.

The capabilities of a programming language are to a large extent determined by the built-in constructs it provides. Rather than to make a long list, Figure 2.1 shows a "cloud" of built-in control constructs and predicates specified by the ISO standard.

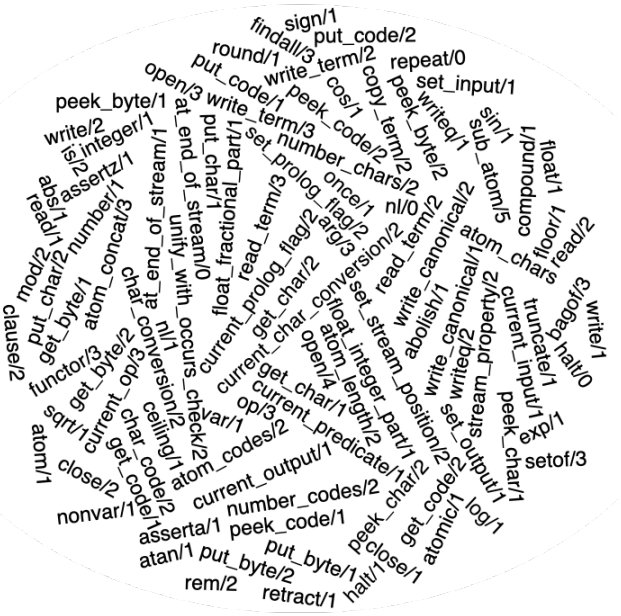

Fig. 2.1 A cloud of ISO Prolog built-in predicates.

Most of the predicates in this cloud *can* and *should* be supported by Web Prolog. (We will get to the exceptions a bit further ahead.) However, while most of them are necessary, they are not sufficient. Here is what else we think is needed:

- In addition to the built-in predicates offered by the ISO Prolog core, Web Prolog needs to support a set of *standard library predicates* that conforming nodes on

the Prolog Web must normally offer. Predicates such as `length/2`, `member/2` and `append/3` should be there – a programmer should not have to load them explicitly.[2] The public-domain set of libraries developed by the Prolog Commons Working Group provides a good start.[3]

- Item number 10 in the list of essential and important features points to the possibility to express Definite Clause Grammars (DCGs) and parse strings using them. DCGs are not yet part of the Prolog ISO standard but are important enough to warrant inclusion in the Web Prolog language.
- As we are aiming for a profile of Prolog suitable for programming on the Web, predicates for working with the Web, e.g. `http_open/3`, and support for serialization as JSON seems essential to have.
- Last but not least, we extend our subset of ISO Prolog with carefully crafted concurrency and distribution primitives heavily inspired by Erlang. When we write "heavily inspired," we really mean it. We try to stay as close to Erlang as possible, we use the same *kind* of actor as Erlang, and we use the same names for our concurrency-oriented predicates that Erlang uses for the corresponding functions.

We would expect the community to be able to agree on the first three points, and the last point in the list is likely to present the only major technical challenge. It is not *much* of a challenge however, since Erlang shows us we are lacking, and where we need to go.

There are features, predicates and libraries that may never make it into Web Prolog. The reason why may vary – they may be dangerous, or superfluous. Other features may be deemed not developed enough to be included in a standard.

- We only use a subset of ISO Prolog as it contains primitives that do not seem to "belong" on the Web and which may also compromise the security of a node, such as predicates that gives a program access to the operative system.
- Even though they are certainly relevant to the Web, there is definitely no need for predicates for building web servers, such as the ones available in SWI-Prolog's `library(http/http_server)`. After all, a Prolog node *is* a web server.
- Even though predicates for constraint logic programming is listed as an important Prolog feature, we do not believe time is ripe to include them in Web Prolog. With time, as they mature, this may of course change.
- No (or limited use) of modules [TODO: Expand on this]

We have characterized Web Prolog as a profile of Prolog suitable for programming on the Web. As we shall see, Web Prolog can be cut up into subsets that are themselves profiles, or sub-profiles if you will. Some such profiles of Web Prolog do not support predicates for database updates such as assert and retract, or predicates for reading from and writing to a terminal. Other profiles do not support `op/3`.

---

[2] https://www.complang.tuwien.ac.at/ulrich/iso-prolog/prologue lists member/2, append/3, length/2, between/3, select/3, succ/2, maplist/2-8, nth0/3, nth1/3, nth0/4, nth1/4, call_nth/2, foldl/4-6 and countall/2.

[3] http://prolog-commons.org

2.2.1 Web Prolog is inspired by Erlang

> I would prefer multi-threading in Prolog to look as much as possible like Erlang.
>
> *Richard O'Keefe*

Erlang is a general-purpose, concurrent, functional programming language developed by Joe Armstrong, Robert Virding and Mike Williams in 1986. Regarded as the first actor programming language to gain popularity, it started out as a proprietary language within Ericsson, but was released as open source in 1998.[4] Erlang was designed with the aim of improving the development of telephony applications, but has in recent years, thanks to its built-in support for massive concurrency, distribution and fault tolerance, been used as a very capable language for implementing the server-sides of web applications,[5] as well as for programming a wide range of soft real-time control problems [48].

Inspired by the list of essential features of Prolog given in the previous section, and if given the task of formulating a similar list defining what it means to be an Erlang-style programming language, we would likely include at least the following three essential requirements:

1. The ability to execute a large number of actor processes concurrently;
2. the ability to use message-passing for inter-process communication, avoiding mutable shared memory and locking issues;
3. the ability to support network-transparent concurrent programming where actors are allowed to create other actors on remote computers and enter into seamless communication with them.

These are three essential features that if added to the features of ISO Prolog will provide us with the most crucial parts of a foundation for Web Prolog, and indeed for the whole Prolog Trinity ecosystem. Of course, for this to work we must make sure that the two sets of features are compatible. Our current understanding is that they are, and we intend to demonstrate this in the book.

Why did we choose Erlang as a source of inspiration, rather than any alternative? After all, there are other actor programming languages – Akka, Pony and E, to name a few. The main reason for the choice of Erlang is that it is arguably the most *mature* concurrency-oriented language in existence, with a bigger community than the alternatives, and a community that includes industrial users. Many millions of lines of source code have been written in Erlang, many books teaching the language have been authored, and university courses teaching concurrent and distributed programming often uses Erlang. In our opinion, since we are aiming for a standard it makes a lot of sense to borrow the required concurrency-oriented features from a language as mature and battle-tested as Erlang.

---

[4] A good overview of the Erlang language, its design intent and the underlying philosophy, is given in the Ph.D. thesis of Joe Armstrong [3].

[5] By companies such as WhatsApp and Klarna for example.

Another reason for our choice is that Erlang and Prolog are in many ways remarkably similar, both when it comes to syntax, and the reliance on dynamic typing, assign-once variables, pattern matching and recursion. The similarities can be explained by the fact that historically, Erlang evolved from Prolog and the first version of Erlang was actually implemented as an interpreter in Prolog [4]. The following listings of Prolog code (to the left) and Erlang code (to the right) show some of the similarities as well as some of the differences between the two languages:

```
% append/3

append([], L, L).
append([H|L1], L2, [H|L]) :-
    append(L1, L2, L).
```

```
% append/2

append([], L) -> L ;
append([H|L1], L2) ->
    [H|append(L1, L2)].
```

Note the use of capital letters for variables, the familiar notation for lists, and the reliance on pattern matching and recursion. A major difference is that Erlang is a functional programming language, whereas Web Prolog is relational. Since a function is just a special case of a relation this difference must not be exaggerated, but it does show in the way function calls may be nested, something that cannot be done in Prolog (or Web Prolog) since arguments are used to represent outputs as well as inputs. What *can* be done in Prolog, however, but not in Erlang, is to call `append/3` in more than one *mode* – not only to append one list to another, but also, for example, to non-deterministically split a list up into two parts.

Despite such differences, as long as we do deterministic computation only, and no search is involved, logic programming and functional programming are fairly similar in the way they work, and methods used to achieve success with one often transpose to the other. Again, features such as pattern matching, assign-once variables and recursion, for example, typically play important roles in both kinds of languages.

Erlang was *losing* features that Prolog had (and still has), but *gained* powerful support for concurrent and distributed programming that Prolog did not have (and still does not have enough of). We are of course thinking of features such as 1, 2 and 3 in the above list of requirements. Key here is the concept of an actor, and the ways actors can be scripted. Below, the echo server written in Web Prolog is placed side-by-side with an implementation in Erlang – a predicate in Web Prolog and a function in Erlang – both making a recursive call that implements a loop.

```
echo_actor  :-
    receive({
        echo(From, Msg) ->
            From ! echo(Msg),
            echo_actor
    }).
```

```
echo_actor() ->
    receive
        {echo, From, Msg} ->
            From ! {echo,Msg},
            echo_actor()
    end.
```

The Web Prolog code on the left demonstrates the use of two constructs foreign to traditional Prolog, implementing the sending and receiving of messages in the style of Erlang. The programs have a similar look, and this is intentional. We have *strived*

to make Web Prolog look as similar as possible to Erlang within the constraints imposed by the syntax of ISO Prolog.[6]

A function call such as `Pid = spawn(fun() -> echo_actor() end)` can be made in order to spawn an echo server in Erlang, while doing it in Web Prolog would use something like `spawn(echo_actor, Pid)`. These calls look somewhat similar too, but while Erlang is a higher-order language in which the spawn function takes an anonymous function as its argument, Prolog (or Web Prolog) is not a higher-order language in this sense. In Web Prolog, `spawn/2` is a *meta predicate* which expects a callable goal to be passed in the first argument, a goal that when called will execute a procedure that determines how the actor will behave.

2.2.2 Web Prolog is a multi-paradigm programming language

> Prolog is different, but not that different.
>
> *Richard O'Keefe*

A common way to categorize programming languages is in terms of the programming paradigm(s) they support. In the very nicely organized and very well argued taxonomy of programming paradigms by Peter van Roy in [47], depicted in Figure 2.2,[7] it is indeed possible to pinpoint exactly where Web Prolog fits in. We have enclosed the relevant square boxes in boxes with rounded corners.

This suggest that we can regard Web Prolog as a language that supports three programming models: *relational logic programming* (like in Prolog), *imperative programming* (also like in Prolog), and (like in Erlang) *message-passing concurrent programming*. (A note on terminology is in order. In this book we prefer the term *actor-based programming*. Joe Armstrong used to refer to it as *concurrency-oriented programming*, or COP.).

With access to predicates for asserting and retracting clauses in the dynamic database, traditional Prolog has always, albeit somewhat reluctantly, been able to support imperative programming (and later additions such as `setarg/3` has of course amplified this). Also, the built-in backtracking *search* that is so useful in Prolog has little to do with logic. Thus Prolog has never really been a single-paradigm programming language, and this is reflected in van Roy's taxonomy. In [47] he suggests that this is not a bad thing:

> A good language for large programs must support several paradigms. One approach that works surprisingly well is the *dual-paradigm language*: a language that supports one paradigm for programming in the small and another for programming in the large.

With the addition of support for the actor programming model, Web Prolog becomes a much clearer as well as (in our view) a more interesting case of a multi-paradigm

[6] Note that `!/2` here is the Erlang-style *send* operator, rather than the Prolog cut (`!/0`).

[7] The diagram is available at https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.pdf

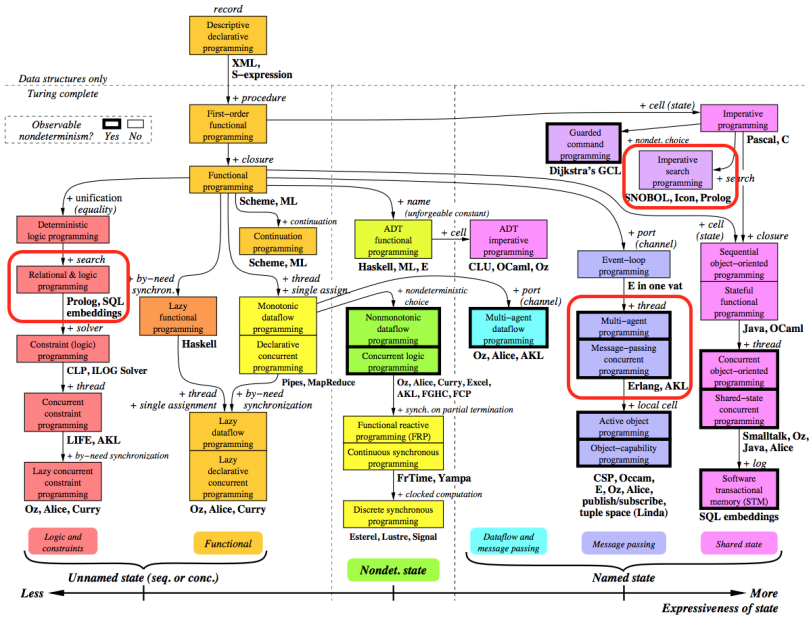

Fig. 2.2 A taxonomy of programming paradigms due to Peter van Roy.

programming language. In particular, it can be argued that the support for programming in the large has been greatly improved.

Still, it is probably wise to entertain a suspicion of unexpected interactions between language features and possible impedance mismatches between the two paradigms – between Prolog's relational, non-deterministic programming model based on logic and Erlang's functional and message passing model. How well do the Erlang-style constructs mix with Prolog – with backtracking for example, or with the features for imperative programming? What do we get if we combine them? A kludge, or something quite beautiful? This, as so many other things, might be in the eye of the beholder, but we know what *our* eyes tell us.

In theory, we should be on the safe side. Sequential Erlang is basically Erlang with its data types, one-way pattern matching, functions and control structures, but without spawn, send, receive, and other constructs used for concurrent programming. The idea behind Web Prolog can thus be described as an attempt to "plug out" the sequential part from Erlang and "plug in" sequential Prolog instead. There seems to be no principled reasons why we would not be able to replace the sequential functional language with a sequential relational language, e.g. a logic programming language such as Prolog. Indeed, in [2] Armstrong writes:

> Erlang is a concurrent programming language with a functional core. By this we mean that the most important property of the language is that it is concurrent and that secondly, the sequential part of the language is a functional programming language.

> The sequential subset of the language expresses what happens from the point it time where a process receives a message to the point in time when it emits a message. From the point of view of an external observer two systems are indistinguishable if they obey the principle of observational equivalence. *From this point of view, it does not matter what family of programming language is used to perform sequential computation.* (Our emphasis.)

Alternatively – and this has been our choice – the approach can be described as an attempt to *extend* Prolog with constructs such as spawn, send and receive. The idea is to keep everything that core Prolog has to offer, and extend it with a number of those primitives that make Erlang such a great language for programming message-passing concurrency. The choice between extending Prolog with Erlang-style constructs and extending Erlang with Prolog-style constructs is easy to make, and a lot has to do with syntax. Provided we can accept using a syntax which is relational rather than functional, precluding the nesting of function calls, it should be clear by now that the surface syntax of Prolog can easily be adapted to express the needed Erlang-style primitives. It is arguably a lot harder to express Prolog rules and other constructs using the syntax of Erlang.

Paradigms alone hardly capture all there is to a practical programming language. Languages may also be related by having a special purpose in common, or by constraints imposed upon them by a particular area of application. One such purpose, which comes with constraints pertaining to security, is the use of a language for programming the Web.

2.2.3 Web Prolog as a scripting language for the Web

> Scripting languages are a lot like obscenity. I can't define it, but I'll know it when I see it.
>
> *Larry Wall*

Over the years, the Web has become a key delivery platform for a variety of sophisticated interactive applications in just about any conceivable domain. As a consequence, JavaScript, more or less the only game in town for programming the front-end of a web application, has become among the most commonly used programming languages on Earth. It appears that even back-end developers are more likely to use it than any other language.[8] Indeed, JavaScript must be regarded as *the* web programming language of our times.

JavaScript started out as a very small, highly domain-specific scripting language that was limited to running within a web browser to dynamically modify the web page being shown, but has over time evolved into a widely portable general-purpose programming language. Modern JavaScript is a high-level, dynamically typed, interpreted multi-paradigm programming language with several essential features that

[8] http://stackoverflow.com/research/developer-survey-2016#most-popular-technologies-per-occupation

make it a versatile tool for web development. JavaScript's syntax supports various programming paradigms, including imperative, functional, and object-oriented programming styles. With features such as callbacks, promises, and async/await, JavaScript supports asynchronous programming, enabling non-blocking operations, especially useful in web applications. JavaScript's native format for data interchange, JSON, is lightweight and widely used for data transmission. JavaScript runs on almost all modern web browsers and platforms, making it universally applicable for web development. It easily interfaces with numerous web APIs (Application Programming Interfaces) for tasks like accessing the Document Object Model (DOM), making HTTP requests and handling events. This makes it an excellent tool for connecting disparate systems in web applications.

With the advent of Node.js,[9] JavaScript extended its reach to server-side development. This allowed for a unified language experience across both client and server sides, simplifying the development process by allowing the same language to be used throughout the entire stack. Not having to learn more than one language in order to become a so called "full-stack developer," and not having to switch languages when changing from working on the server-side to working on the client-side are important advantages. Also, Node.js is frequently used to build and connect microservices. Its non-blocking I/O model and event-driven architecture make it suitable for building scalable and efficient network applications.

It is clear that as a relational logic programming language with the essential and important features listed above, Prolog is very different from JavaScript in terms of paradigms supported. The addition of Erlang-style concurrency and asynchronous communication does not really narrow the gap, as those are features that work differently in Erlang. What matters here is the *role* that JavaScript plays on the Web. We want Web Prolog to play a similar role in the Prolog Trinity ecosystem as JavaScript does in the JavaScript ecosystem, just as clear-cut.

The connection between scripting languages and the Web has been noted before. For example, in his 1998 article "Scripting: higher level programming for the 21st Century,"[33] John Ousterhout writes (about the Internet rather than the Web, but that makes little difference):

> The growth of the Internet has also popularized scripting languages. The Internet is nothing more than a gluing tool. It does not create any new computations or data; it simply makes a huge number of existing things easily accessible. The ideal language for most Internet programming tasks is one that makes it possible for all the connected components to work together; that is, a scripting language.

So what about the suitability of Web Prolog for programming the Web? Can Prolog challenge JavaScript in this space, at least for some applications, such as web-based AI? For this to be possible, we must ensure that Web Prolog is

- viable as a scripting language,
- suitable for programming the Web,
- possible to implement in browsers, and

[9] https://en.wikipedia.org/wiki/Node.js

- secure.

As regards the viability of Web Prolog as a scripting language we note that while hardly a definition, Ousterhout characterizes scripting languages as follows in [33]:

1. They are suitable for "programming in the large,"
2. they are designed for "glueing" applications together,
3. they tend to be typeless, thus making it easier to connect components,
4. they are usually interpreted, thus providing rapid turnaround during development by eliminating compile times,
5. they are higher level than a system programming language in the sense that a single statement does more work on average, and
6. they allow rapid prototyping.

JavaScript does indeed fit this characterization, but given the development we described above, it should now probably be described as general-purpose languages which also happens to be suitable for scripting. By contrast, some languages are not suitable for that purpose – C++ and Java comes to mind.

What about Prolog? Does it satisfy Ousterhout's characterization? The points 3-6 in his list are probably true of the Prolog programming language in general. (With the possible exception of 4 – some systems compile, but do it very quickly.) Whether or not the points 1 and 2 applies varies between systems. For example, the people behind the commercial SICStus Prolog appears to regard Prolog more as a language for writing problem-solving modules to be embedded into other code, written in Java, C++, Python or what have you.[10] It is only a guess, but we suspect that the people behind SICStus Prolog would not be willing to characterise it as a scripting language, although that may simply be a business decision more than anything else. In contrast, acting as a glue language and a language for programming in the large is the ambition of SWI-Prolog, witness the following quote from the online manual:[11]

> SWI-Prolog positions itself primarily as a Prolog environment for 'programming in the large' and use cases where it plays a central role in an application, i.e., where it acts as 'glue' between components.

Interestingly, the people behind Erlang also think of it as a glue language:

> We use Erlang as the glue to handle all orchestration, and then we use Python, C, Julia, ... It is actually a language *intended* to act as a hub towards other languages. The interfaces could be protocols, could be RESTful APIs, or other programming languages. It's ideal for that.[12]

On the Web – the biggest distributed programming system ever constructed – Erlang-style network-transparent message-passing concurrency indeed appears to be ideal for programming in the large, and indeed for the orchestration of both local and remote processes. But recall van Roy's assertion that a good language for large programs must support several paradigms, one paradigm for programming in the

[10] Mats Carlsson, main developer of SICStus Prolog, personal communication

[11] http://www.swi-prolog.org/pldoc/man?section=swiprolog

[12] See https://www.youtube.com/watch?v=K8nxTSPHZhs5, 8:58 into the discussion.

small and another for programming in the large. With the addition of Erlang-style message-passing concurrency to Prolog's logic programming core supplemented by its imperative features, we end up with three paradigms. This cannot be a bad thing. It is likely that Erlang glue is of a different kind, with different properties, than Prolog glue. There is hope that the *combination* of Prolog and Erlang might result in an even stronger and more flexible glue of the kind required – a *superglue* if you will.

Despite the good fit with Ousterhout's characterization, neither Prolog nor Erlang are usually *advertised* as scripting languages, and they were not *designed* for the purpose of glueing applications together (or at least Prolog was not). They are both full-blown very flexible general-purpose programming languages with "batteries included." Like JavaScript, they should probably be regarded as general-purpose languages which also *happens* to be suitable for scripting.

Web Prolog is a very *general* special-purpose scripting language. Similar to most other scripting languages, Web Prolog can be used for purposes for which "scripting" is not really the right word. Web Prolog comes with a focus on web *logic* programming as well as web *agent* programming, and in client-side browsers as well as server-side. Using Web Prolog, the owner of a node is for example able to build knowledge bases consisting of many millions of clauses – facts as well as rules, and/or web agents that can make use of such knowledge bases.

Regarding our third point, for Web Prolog to become a viable web programming complement to JavaScript it is vital that it can be implemented in browsers too, since this comes with a number of advantages:

1. When network connection is slow, it is best to perform the majority of computations in the browser.
2. This is where, in most scenarios, the *state* of an interaction between a user and an application should preferably be represented.
3. Client-side computation reduces the demands placed on nodes.
4. This is also where we find browser APIs that lets a web developer manipulate the DOM, store data locally, and add features such as spoken interaction, video or graphics to an application.

For Web Prolog in the browser to play the same role as JavaScript in the browser currently does, it must allow Web Prolog source code to be loaded from any server on the Web by means of the HTML `<script>` element, inline or with a link.

We do not, however, go all the way in our attempt to replicate Javascript's abilities as a scripting language. One thing that we do not at this time aim for is to make Web Prolog into a language with primitives for scripting the Document Object Model (DOM) in web browsers. With time it may well be interesting to develop such capabilities, but we believe it is simply too early to try to standardize them. In fact, we suspect that JavaScript will reign supreme in that role, and that if a browser is running Web Prolog locally, it will be as a *web worker*,[13] with which the main JavaScript process is talking.

[13] https://en.wikipedia.org/wiki/Web_worker

As for security, similar to JavaScript, Web Prolog is a *sandboxed language*, open to the execution of untested or untrusted source code, possibly from unverified or untrusted clients without risking harm to the host machine or operating system. Therefore, Web Prolog does not include predicates for file I/O, socket programming or persistent storage, but must rely on the host environment in which it is embedded for such features.

Web Prolog does indeed share some of the properties that made JavaScript succeed on the Web. A visit to a web application server starts a JavaScript process on the client, running code that has been downloaded from the application's host to the user's client. Such a process must be allowed to run there (i.e. the owner of the client must allow it), and when it is (and most users allow it by default), it must execute in a way that does not harm the client. When a Web Prolog process is run on a node it must be allowed by its owner to do so, and it must run without harming the node. So one thing they share, Web Prolog and JavaScript, is the ability to run untested and untrusted source code, authored by unverified and untrusted programmers, in a sandbox on someone else's computer.

In summary, while JavaScript's initial role was confined to client-side scripting in web browsers, its capabilities have expanded significantly. Today, it serves as a versatile glue language that connects various components in both client and server environments, as well as in the broader web development ecosystem. When designing Web Prolog and the rest of the Prolog Trinity ecosystem, we must of course try to learn from such a popular and versatile tool for web development, with a focus on the good parts in particular.

2.3 Erlang-style message-passing concurrency in Web Prolog

> In a nutshell, if you were an actor in Erlang's world, you would be a lonely person, sitting in a dark room with no window, waiting by your mailbox to get a message. Once you get a message, you react to it in a specific way: you pay the bills when receiving them, you respond to Birthday cards with a "Thank you" letter and you ignore the letters you can't understand.
>
> *Fred Hébert*

Inspired by Erlang's notion of an actor, the *Prolog actor* is the fundamental unit of computation in the Prolog Trinity ecosystem. The diagram in Figure 3.2 shows three actors, one of which has just been spawned, and one that has been opened up so that we can have a look inside (where on this level of abstraction they all look the same). A Prolog actor is a *process*, capable of communicating with the world around it through messaging. An actor has a unique address – a *process identifier*, or *pid* for short. If we have the pid of an actor, we can message it. Since pids can themselves be part of a message, it allows other actor processes to communicate back. Furthermore, an actor has a *mailbox* that stores incoming messages. There is a *receive* operation that allows the actor to select and process messages from the mailbox, and a *send* operation that can be used to send messages to any other actor. Finally, there is a *spawn* operation that allows an actor to create other actors – child actors as it were.

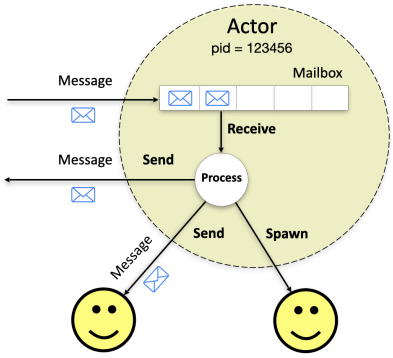

Fig. 2.3 The anatomy of a Prolog actor.

Corresponding to these operations there are built-in Web Prolog predicates such as `!/2`, `receive/1-2` and `spawn/2-3`. In this section, the use of these and other predicates will be demonstrated by examples. We choose to work with very simple examples, many of which are borrowed from tutorials and text books teaching Erlang to beginners. The major motivation for having it in this way is to try to satisfy two kinds of readers who might want to approach the material in two different ways. Readers who have a year or two of Prolog programming under their belt, but feel that time is ripe to have a look at concurrent programming, may want to look at the examples very carefully and perhaps work through them themselves using the Trinity Prolog implementation.

Readers who are very experienced Prolog programmers and teachers might want to ask themselves if this a good way to teach students of Prolog some concurrent programming techniques, or if something else might work better. Their role is as evaluators of potential teaching material rather than learners. We know what we believe; we think Web Prolog might be suitable for teaching both Prolog and Erlang-style actor programming in the same system, and potentially in a web-based playground – a great way to create as little hassle for a teacher as possible.

Then, of course, we expect the latter kind of reader to evaluate our proposal and to determine if Erlang-style concurrent programming makes sense in the context of Prolog.

2.3.1 Programmer talking to actor, actor talking to itself

If we need to hold a Prolog-style conversation with an actor, we may want to talk to a Prolog *shell*. By using a terminal attached to a shell, we can interact with it the way we normally interact with Prolog, by querying it, updating its dynamic database, and

so on. For example, here is how we instruct the shell to split a list up into a prefix and a postfix using `append/3`:

```
?- 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.
?-
```

This book, as we have warned, is not a Prolog tutorial, so how such queries are able to come up with results is not something we will explain. Our focus is rather on the concurrency-oriented, message-passing features of Web Prolog, so we begin instead by introducing a couple of simple messaging primitives. But first, using `self/1`, we need to determine the identity of the shell we will be talking to:

```
?- self(Self).
Self = 85234512.
?-
```

What we got back here is a *pid*, an unforgable and locally (but not necessarily globally) unique identifier. Our proposal here is to use random integers of a size that makes it impossible in practice for anyone to *guess* the pid of an actor.

As stated above, if we know the pid of an actor process we can send it a message. Just as any other kind of actor, the shell is equipped with a mailbox, and this is where the message will end up. The syntax and semantics of the send operator is easy to explain. With a call such as `Actor ! Message` the term `Message` is sent to the mailbox of the process identified as `Actor`, either by means of a pid or (as we shall see) using a registred *name*. For example, using `!/2` with the pid of our shell we can instruct it to send *itself* a message:

```
?- 85234512 ! hello.
true.
?-
```

Now we can use `receive/1` to make sure that the message we sent was received by the shell and is present in its mailbox. A single receive clause with a variable in the head and `true` in the body does the job:

```
?- receive({Message -> true}).
Message = hello.
?-
```

This is the simplest use of the receive operation we can think of, but `receive/1` really is more complex than this, so expect more to come further ahead in this chapter. One more thing about the above call to `receive/1` is worth a mention already at this point though: had the mailbox been empty `receive/1` would have *blocked* the execution of the process running as a shell, waiting for a message to show up.

2.3.2 Two utility tools for programming in the shell

and get aquainted with a couple of features that makes interaction convenient when programming against a Web Prolog shell

Having to copy and paste pids is not very convenient. Instead we can make use of a shell utility feature, borrowed from SWI-Prolog, which allows a variable binding resulting from the successful execution of a shell query to be reused in future shell queries if a dollar symbol is put in front of it, like so:

```
?- $Self ! hello.
true.
?-
```

Here, the shell kept track of the most recent binding of `Self` to a value (`85234512` in this case) and substituted the term `$Self` with that value before the query was run.

While being able to inspect the contents of the shell's mailbox during interactive programming is obviously important, using `receive/1` is not the most convenient way of doing it. After all, we cannot always be sure that there *are* any messages in the mailbox, and if it is empty `receive/1` will block and make normal interaction impossible. The call will just hang indefinitely, and the only way out may be to abort it using Control-C. Furthermore, we may want to see *all* messages in the mailbox which means that we need to run `receive/1` more than once, but we may not know how *many* messages there are, and then again run into the blocking problem. A more convenient tool here is the utility predicate `flush/0`. This predicate will show (and remove) all messages from the mailbox and will never block.

Below, we demonstrate both of these utilities by first sending yet another message to our shell and then use `flush/0` to inspect and empty the contents of its mailbox:

```
?- $Self ! goodbye.
true.
?- flush.
Shell got hello
Shell got goodbye
true.
?-
```

The $Var substitution mechanism (or "dollar notation") in combination with `flush/0` comes in very handy during interactive programming in the shell and we shall rely on them extensively when running examples through all of the book.

2.3.3 Actors talking to other actors

It takes two to tango, and an actor talking to itself is not very interesting. Fortunately, as already mentioned, an actor is capable of creating other actors and then start talking to them.

The built-in predicate `spawn/1-3` is used to create new actor processes. In a call such as `spawn(Goal,Pid)` it expects a callable goal to be passed in the first argument, and will bind the variable in the second argument to the pid of the actor that is created. Here is how this might look like:

```
?- spawn(echo_actor, Pid).
Pid = 72347585.
?-
```

The goal calls a predicate which determines the behavior of the actor. The actor process runs in parallel with the caller, the shell in this case. Here is an example script that implements a simple echo server:

```
echo_actor :-
    receive({
        echo(From, Msg) ->
            From ! echo(Msg),
            echo_actor
    }).
```

We have seen this piece of code before, and it is time to explain how it works. The predicate `echo_actor/0` defines a loop where a call to `receive/1` tries to select a message of the form `echo(From,Msg)` from the mailbox and send the echo message back to the actor referenced by the pid to which the variable `From` is bound, and then continue looping by making a recursive call.

To make our server return the echo message to the shell we must send it a message of the form `echo(Pid,Msg)` with `From` bound to the pid of the shell and `Msg` bound to the message. Here is how we do that:

```
?- self(Self), $Pid ! echo(Self, hi).
Self = 85234512.
?- flush.
Shell got echo(hi)
true.
?-
```

The sending is asynchronous, i.e. `!/2` does not block waiting for a response but continues immediately. Also, sending a message to a pid always succeeds and never throws an exception, even if the pid points to a non-existing process. In that case the message is simply discarded. This means that short of having the targeted actor return a confirmation message we will not always know if the message reached it. However, the above case did not leave us in the dark, as the actual echo served as a confirmation message.

2.3.3.1 Monitoring

When an actor process A_1 spawns an actor A_2, A_2 becomes the *child* of A_1, and A_1 the *parent* of A_2. Since A_2 may in turn spawn other processes, the actors involved may form a hierarchy. In our previous example, the shell and the echo server entered into precisely this relationship and thus formed a (very shallow!) hierarchy.

In a call such as `spawn(Goal,Pid,Options)` the optional third argument is a list of options used for the configuration of the actor. Two of these options – `monitor` and `link` – can be used to specify what should happen when the actor terminates, i.e what the parent might learn about the reason for its death, and how its children will be treated when it dies.

If the value of the `monitor` option is set to `true` the actor under construction is instructed to inform the parent about what eventually will become its fate. Let us look at a simple example:

```
?- spawn(append([a,b],[c,d],Zs), Pid, [
       monitor(true)
   ]).
Pid = 60367387.
?- flush.
Shell got down(60367387,true)
true.
?-
```

Here, the `monitor` option is set to `true`, instructing the actor to send a special-purpose `down` message carrying a small piece of information on how the run went to its parent process just before terminating. The idea is to allow the parent process to observe the child process and to detect if it has terminated for any reason. In this case, the `down` message was delived to its parent as soon as the goal terminated. Normally, an actor process will terminate and completely disappear when it has run out of things to do. It may be because the script succeeds, or fails, or throws an error. In this case, the atom `true` in the second argument of the `down` message tells us that it terminated in a way that can be considered normal.[14]

One more thing about this example deserves a comment. Note that although the goal `append([a,b],[c,d],Zs)` succeeds, the variable `Zs` is not bound. This is a consequence of the fact that the goal is *copied* before being run in the new process. (This is how it works in Erlang too.) Because of that, the only way for the parent process to get hold of the result of the call is to send it to the parent, like so:

```
?- self(Self),
   spawn((append([a,b],[c,d],Zs), Self ! Zs), Pid, [
       monitor(true)
   ]).
Self = 85234512,
Pid = 77129823.
```

[14] Erlang uses the atom `normal` to indicate this.

```
?- flush.
Shell got [a,b,c,d]
Shell got down(77129823,true)
true.
?-
```

2.3.3.2 Linking

If the value of the `link` option is `true` it means that when the actor terminates, any children that it might have are forced to terminate too. So in the following example, if the execution of `foo/0` spawns other actors (and the link only makes sense if it does), then should the `foo/0` call terminate, these actors will automatically be killed.

```
?- spawn(foo, Pid, [link(true)]).
Pid = 45092311.
?-
```

Note that it does *not* mean that the `foo/0` call must terminate if any child that it may have spawned terminates. The link is *uni-directional,* and makes the life of a child dependent on the life of its parent, but never the other way around. (Erlang is different here, as its links are bi-directional, and we shall discuss this difference further ahead.)

In the following example, because `link` is set to `false` our echo server will not necessarily terminate if the *shell* is terminated.

```
?- spawn(echo_actor, Pid, [link(false)]).
Pid = 66720916.
?-
```

In our proposal, the default value for `link` is `true`, as this is deemed to be what we usually want. In fact, and for a reason that will be explained later, in the context of distributed web programming it might be a good idea to *require* that the value of `link` is set to `true`.

2.3.3.3 Registering

Calling `register(Name,Pid)` associates the atom `Name` with `Pid`. The name can be used instead of the pid when calling `!/2`. For example, we can register our shell under the name `shell`:

```
?- self(Self), register(shell, Self).
Self = 85234512.
true.
?- shell ! hello.
true.
```

```
?- spawn(shell ! goodbye).
true.
?- flush.
Shell got hello
Shell got goodbye
true.
?-
```

Registering is useful when we want to offer a service that should always be available under a name that is easy to remember. Should a crash occur, all our system needs to do is to restart it and associate the same name with the pid of the new process.

2.3.3.4 Exiting

Web Prolog supports two predicates, `exit/1` and `exit/2`, that can be used to terminate an actor process. If the process calls `exit(Reason)` it will terminate immediately, and if monitored by its parent process, the parent will be sent a `down` message, containing the term that `Reason` was bound to when predicate was called.

Here is a simple and silly example where a process is spawned and monitored by the shell. Since the process is told to exit immediately with the reason `my_reason`, the shell receives a `down` message with the atom `my_reason` in its second argument:

```
?- spawn(exit(my_reason), Pid, [
       monitor(true)
   ]).
Pid = 91325643.
?- flush.
Shell got down(91325643, my_reason)
true.
?-
```

It sometimes happens that we need to terminate an actor process by force. Our echo server, for example, can only be terminated in this way. The predicate `exit/2` can be used to terminate any actor process with a known pid. If we do not know the pid, but it has a name that we know, we can use that instead. To see how this works, let us spawn a new monitored echo server and register it:

```
?- spawn(echo_actor, Pid, [
       monitor(true)
   ]),
   register(echo_actor, Pid).
Pid = 21562390.
?-
```

Now, let us say we change our minds and want to get rid of it again:

```
?- whereis(echo_actor, Pid),
   exit(Pid, 'We changed our minds!').
Pid = 21562390.
?- flush.
Shell got down(21562390,'We changed our minds!')
true.
?-
```

By calling `exit/2` with the pid in the first argument and a term detailing the reason for exiting in the second, we were able to terminate the process. Note that a call to `exit/2` will only accept a pid in its first argument, so if all we have is a name, the built-in predicate `whereis/2` must be used to locate it.

Note that the reason passed to `exit/1-2` can be any term, not just an atom. This means that it can be used to transfer an arbitrarily large chunk of information back to the parent. However, in most cases an atom is all that is needed, and it is worth noticing that using `true` as a reason can be used to suggest to the parent that the termination was *normal*, even though it was caused by a call to `exit/1-2`.

What might seem a bit odd is that even thought `21562390` is now provably dead, trying to send it a message using a pid or calling `exit/2` still succeeds without an error, although no `down` message is being sent:

```
?- self(Self), $Pid ! echo(Self, bye).
Self = 85234512.
?- exit($Pid, 'We changed our minds!').
true.
?- flush.
true.
?-
```

Treating `!/2` and `exit/2` as no-ops when the pid points to a non-existent process is consistent with how it works in Erlang. However, trying to send a message using the name of a non-existent process generates an error:

```
?- echo_actor ! echo($Self, bye).
Error: The name 'echo_actor' is not associated with a pid.
?-
```

This too is consistent with how Erlang works.

2.3.4 A closer look at receive/1-2

> I would argue that it is precisely the 'receive' construct in Erlang that makes Erlang such a joy to use.[15]
>
> *Richard O'Keefe*

[15] https://groups.google.com/forum/#!msg/erlang-programming/gjU-HCoq7dk/Mx_Af0iQ5P0J

As shown already, but only for a trivial case, an actor process uses the receive primitive to extract messages from its mailbox. Since the syntax and semantics of `receive/1-2` is fairly complex, a closer look and more examples are needed. Below we give several simple examples illustrating different ways to use the `receive/1-2` predicate in Web Prolog, demonstrating how to handle different *types* of messages, use *timeouts*, apply *guards*, and more. Other examples illustrate how messages are deferred and handled in subsequent `receive/1-2` calls, demonstrating the flexibility of passing and processing messages in Web Prolog.

2.3.4.1 Basic receive

In Web Prolog, just like in Erlang, the receive operation specifies an ordered sequence of *receive clauses* delimited by semicolons. A receive clause always has a *head* (a term) and a *body* of Prolog goals. Schematically, a receive call looks like this:

```
receive({
    Head1 -> Body1 ;
    Head2 -> Body1 ;
    ...
    HeadN -> BodyN
})
```

Often, the head is just a single term serving as a *pattern*. Any term will do, ground or non-ground, and even a bare variable is fine.

As in Erlang, `receive/1` scans the mailbox looking for the first message (i.e. the oldest) that matches a pattern in any of the receive clauses, blocking if no such message is found. If a matching clause is found, the message is removed from the mailbox and the body of the clause is called. In Web Prolog, just like in Erlang, values of any variables bound by the matching of the pattern with a message are available in the body of the clause.

In its simplest form, the `receive/1` call waits for a specific message and executes the corresponding code in the body of the receive clause if a message that matches the pattern shows up. For example, the following call waits for a message in the form of `hello(Name)` and prints a greeting when it appears:

```
receive({
    hello(Name) ->
        format("Hello, ~s!~n", [Name])
})
```

We can specify multiple patterns in a single `receive/1` call. For example, this call handles messages of either the form `hello(Name)` or the form `goodbye(Name)`, but only one of them:

```
receive({
    hello(Name) ->
```

```
        format("Hello, ~s!~n", [Name]) ;
    goodbye(Name) ->
        format("Goodbye, ~s!~n", [Name])
})
```

We can use the _ pattern to catch any message. In the following case, any message that does not match `hello(Name)` will be caught by the _ pattern:

```
receive({
    hello(Name) ->
        format("Hello, ~s!~n", [Name]) ;
    _ ->
        format("Unknown message received.~n")
})
```

We can nest `receive/1-2` calls. Here, after having received the `start(Name)` message, the call waits for a `continue(Msg)` message:

```
receive({
    start(Name) ->
        format("Starting with ~s.~n", [Name]),
        receive({
            continue(Msg) ->
                format("Continuing with ~s~n", [Msg])
        })
})
```

2.3.4.2 Messages deferred

If no pattern matches a message in the mailbox, the message is *deferred*, which means that the message does not match any of the patterns in the current `receive/1-2` call and remains in the process's mailbox, possibly to be handled later in the control flow of the process. The receive is still running, waiting for more messages to arrive, and for one that will match. Some simple examples illustrating this behavior are shown below.

In the following example, the `goodbye("Bob")` message – which is the oldest and therefore first in line – does not match the clause in the first receive call and is deferred. The `hello("Alice")` message matches that clause, and then the clause in the second receive call handles the `goodbye("Bob")` message

```
?- self(Self),
   Self ! goodbye("Bob"),
   Self ! hello("Alice"),
   receive({
       hello(Name1) ->
```

```
            format("Hello, ~s!~n", [Name1])
    }),
    receive({
        goodbye(Name2) ->
            format("Goodbye, ~s!~n", [Name2])
    }).
Hello, Alice!
Goodbye, Bob!
true.
?-
```

This behavior is particularly useful if we expect two messages but are not sure which one will arrive first. For example, if we insist on processing a message `foo` before `bar`, we can easily do that with receive, like so:

```
wait_foo :-
    receive({
        foo ->
            process_foo,
            wait_bar
    }).
```

```
wait_bar :-
    receive({
        bar ->
            process_bar
    }).
```

Even if `bar` arrives in the mailbox before `foo`, calling `wait_foo/0` would result in `foo` being selected and processed before `bar`. This is why the receive operator is often referred to as *selective* receive.

2.3.4.3 Guards

The head of a receive clause can, in addition to the pattern, optionally specify a *guard* in the form of a Prolog goal. The role of a head of the form `Pattern if Goal` is to make pattern matching more expressive. Here is an example that distinguishes between positive and non-positive numbers using guards:

```
receive({
    number(N) if N > 0 ->
        format("Positive number: ~p~n", [N]) ;
    number(N) if N =< 0 ->
        format("Non-positive number: ~p~n", [N])
})
```

That was simple, and will work in Erlang too, but here is another example, using a different *kind* of goal after the `if` operator:

```
?- self(Self),
   Self ! hello(xantippa),
   receive({
```

```
        hello(W) if husband(W, H) ->
            format("Hello, ~w, say hello to ~w!~n", [W,H])
    }).
Hello, xantippa, say hello to socrates!
true.
?-
```

Note that the variable `H` does not occur in the pattern. A guard like that cannot be used in Erlang. In Web Prolog, its value can be passed to the body of the clause and do a job there. So while readers familiar with Erlang may wonder why we choose `if` instead of `when`, which is the operator Erlang is using, we just happen to think that this difference is big enough to warrant a different name for the operator.

2.3.4.4 Timeouts

The optional second argument of `receive/1-2` expects a list of options. The `timeout` option takes an integer or float that specifies the number of seconds before the call will succeed anyway, even if no match has been found. The `on_timeout` option takes a goal that is called if timeout occurs. Here is an example of its use:

```
receive({
    hello(Name) ->
        format("Hello, ~s!~n", [Name])
},[ timeout(5),
    on_timeout(format("No match received in 5 seconds.~n"))
])
```

If no message of the form `hello(Name)` is received within 5 seconds, the code in the `on_timeout` option is executed.

Here is how a predicate `sleep/1` that suspends execution `Time` seconds can be defined:

```
sleep(Time) :-
    receive({},[
        timeout(Time)
    ]).
```

As we noted earlier, being able to inspect the contents of the shell's mailbox during interactive programming is important, and `flush/0` is a nice tool for doing just that. Its definition also serves as yet another example of the use of the `timeout` option:

```
flush :-
    receive({
        Message ->
            format("Shell got ~q~n",[Message]),
            flush
```

```
    },[
        timeout(0)
    ]).
```

The value `0` of the `timeout` option ensures that the loop terminates immediately if no messages remain in the mailbox. This is how the hanging of the `receive/1` call is avoided.

2.3.4.5 The receive predicate is semi-deterministic

The predicate `receive/1-2` is *semi-deterministic*, i.e. it either fails, or succeeds exactly once. The only way it will fail is if the goal in the *body* of one of its receive clauses fails, or if timeout occurs and the goal passed in the `on_timeout` option fails.

To see how it pans out in a simple corner case, consider the following two calls:

```
receive({foo(X) -> true})        receive({foo(X) -> fail})
```

The first call will succeed if a message matching the pattern `foo(X)` appears in the mailbox of the actor process executing the call, a term such as `foo(314)` for example. The second call will fail (and possibly cause backtracking) once `foo(314)` appears. Only by the first call will the variable `X` be bound (to `314`). Both calls will remove the matched message from the mailbox. In both cases, if a message appears that does not match the pattern, it is deferred.

To implement a looping behavior, Prolog programmers occasionally use a *failure-driven* loop that relies on backtracking rather than recursion. Using this technique, our echo server can be rewritten like so:

```
echo_actor  :-
    repeat,
    receive({
        echo(From, Msg) ->
            From ! echo(Msg),
            fail
    }).
```

For an Erlang programmer, this use of `receive/1` may come as a surprise and is not a technique that can be used in Erlang. In this particular case, a Web Prolog programmer would be advised to stick to the recursive version. However, there are cases when a failure-driven loop is the only way forward and we shall look at an important such case towards the end of this chapter.

2.4 Erlang-style programming in Web Prolog

> 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)

Not only do Web Prolog programs *look* a lot like Erlang, they *behave* a lot like Erlang too. A good way to demonstrate this is to translate a fair number of different Erlang programming examples borrowed from tutorials and text books into Web Prolog and show that they run just like in Erlang. In this section we shall look at a kind of priority queue, a count server, an event-driven state maching, a universal stateful server with hot code swapping, actors playing ping-pong, a programs executing tasks in parallel, and an approach to building simple supervision hierarchies.

While most of the demonstrated actors have a behavior that is fully determined at the time of their creation, we also show how generic actors can be programmed (or reprogrammed) post creation-time.

During the exploration of the examples, we point to the importance of the use of send and receive for implementing the *communication protocols* allowing actors acting as clients to, still within the bounds of one node, talk to actors acting as servers. We also explain why it is often a good idea to hide the details of such protocols behind a dedicated predicate API.

We have described Web Prolog as a language for distributed programming, but this chapter will only deal with local concurrency. Distributed programming will be dealt with in Chapter 4. For more exhaustive documentation of all built-in predicates available in Web Prolog, consult Appendix A.

2.4.1 A priority queue example

To demonstrate the use of the `if` operator and the use of two `receive/2` options that causes a goal to run on timeout, we show a priority queue example borrowed from Fred Hébert's textbook on Erlang [16]. The purpose is to build a list of messages with those with a priority above 10 coming first:

```
important(Messages) :-
    receive({
        Priority-Message if Priority > 10 ->
            Messages = [Message|MoreMessages],
            important(MoreMessages)
    },[ timeout(0),
        on_timeout(normal(Messages))
    ]).

normal(Messages) :-
    receive({
```

```
        _-Message ->
            Messages = [Message|MoreMessages],
            normal(MoreMessages)
    },[ timeout(0),
        on_timeout(Messages=[])
    ]).
```

The timeout set to `0` means that it will occur immediately, but the system tries all messages currently in the mailbox first.

Below, we test this program by first sending four messages to the toplevel process, and then calling `important/1`:

```
?- self(S),
   S ! 15-high, S ! 7-low, S ! 1-low, S ! 17-high.
S = 34871244.
?- important(Messages).
Messages = [high,high,low,low].
?-
```

Note that the implementation of the priority-queue example relies on the deferring behavior of `receive/1-2` and would not work without it.

2.4.2 An event-driven state machine

Web Prolog's receive primitive can be used to implement simple event-driven state machines in straightforward code. Patterns in receive clauses can be used to match against event messages, guards to encode conditions, and the bodies of receive clause to perform actions.

Figure 2.4 depicts a machine comprising two states and four transitions.

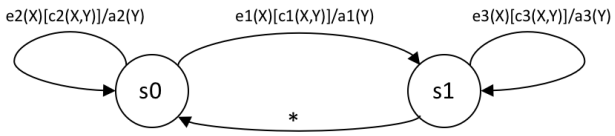

Fig. 2.4 A simple event-driven state machine

Using predicate names to encode states, here is how it can be implemented:

```
s0 :-
    receive({
       e1(X) if c1(X,Y) ->
           a1(Y),
```

```
s1 :-
    receive({
       e3(X) if c3(X,Y) ->
           a3(Y),
```

```
            s1 ;                              s1 ;
        e2(X) if c2(X,Y) ->               _AnyEvent  ->
            a2(Y),                            s0
            s0                          }).
    }).
```

In state `s0` the machine is waiting for an event message of the form `e1(X)` or `e2(X)` to appear in the mailbox of the current process. If `e1(X)` is matched and the condition `c1(X,Y)` holds, the action `a1(Y)` will be called and the machine will transition to `s1`, but if `e2(X)` is matched and `c2(X,Y)` holds, then `a2(Y)` is called and the machine stays in state `s0`. And so on.

As far as we aware, event-driven state machines have not been used much in the Prolog world. Perhaps the reason is that primitives for sending and receiving messages did not appear in Prolog until (a few) platforms implemented the ISO Prolog Threads draft standard. As can be seen by an Erlang/OTP behavior such as `gen_statem`,[16] Erlang takes event-driven state machines very seriously, and perhaps it is time for Prolog to follow suit. We shall assume that it is, and in Chapter 6 present a proposal for how to introduce Web Prolog as a scripting language for StateChart XML, a W3C standard which provides an XML-based notation for event-driven state machines of a particularly sophisticated kind.

2.4.3 A stateful count server

We have already looked at an example of a *stateless* server, namely the `echo_actor` presented earlier in this chapter. Let us now turn to *stateful* servers and show how state may be programmed. In the following example, we demonstrate how to script an actor that can keep a *count*:

```
count_actor(Count0) :-
    receive({
        count(From) ->
            Count is Count0 + 1,
            From ! count(Count),
            count_actor(Count) ;
        stop ->
            true
    }).
```

The predicate `count_actor/1` defines a loop where a call to the built-in predicate `receive/1` tries to select a message of the form `count(From)` from the mailbox, increment the counter, send the current count back to the actor referenced by the pid to which the variable `From` is bound, and continue looping by making a recursive

[16] http://erlang.org/doc/man/gen_statem.html

call. Note how the state of the counter – the current count, that is – is kept in the argument of the predicate.

Note that we added a second receive clause that will allow us to terminate the server without using `exit/2`. We just have to send it a message of the form `stop`.

Here is how an actor following this script can be made to perform when spawning it:

```
?- spawn(count_actor(0), Pid, [
       monitor(true)
   ]).
Pid = 60367387.
?-
```

Here, the `monitor` option is set to `true`, instructing the actor to send a special-purpose `down` message carrying a small piece of information on how the run went to its parent process just before terminating. Default is to not monitor.

Calling `self/1` determines the identity of the toplevel process – the process that just became the parent of the spawned actor:[17]

```
?- self(Self).
Self = 41167597.
?-
```

In the next step the send operator `!/2` is used for sending a message to the spawned actor, instructing it to increment the count and to return the result in the form of a message.

```
?- $Pid ! count($Self).
true.
?-
```

A call to `receive/1` can be made in order to collect the message arriving from the actor:

```
?- receive({Count -> true}).
Count = count(1).
?-
```

Here, `!/2` is used to send two messages to the actor that will end up in its mailbox, in the order they were sent:

```
?- $Pid ! count($Self), $Pid ! stop.
true.
?-
```

Finally, the utility predicate `flush/0` is used to inspect the contents of the mailbox belonging to the toplevel process:

[17] If you are an actor, this is how you find out who you are!

```
?- flush.
Shell got count(2)
Shell got down(60367387, true)
true.
?-
```

Because the actor was monitored, a `down` message was found in addition to the current count. The value `true` in the second argument means that `count_actor/1` succeeded.

2.4.4 A bigger, tastier example

As a tastier example of how a process can be made to hold an updatable state during a conversation we have adapted a fridge simulation example from Fred Hébert's introduction to Erlang [16]:[18]

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

The program creates a process allowing three operations: storing food in the fridge, taking food from the fridge, and terminating the fridge. It is only possible to take food that has been stored beforehand. Again, with the help of recursion the state of a process can be held entirely in the argument of the predicate. In this case we choose to store all the food as a list, and then look in that list when someone needs something.

Assuming the above program is already available, the following session creates the server process. We can then call `self/1`, `!/2` and `flush/0` from the shell in order to simulate the actions of a client:

[18] http://learnyousomeerlang.com/more-on-multiprocessing#state-your-state

```
?- spawn(fridge([]), Pid, [
       monitor(true)
   ]).
Pid = 77346122.
?- self(Me), $Pid ! store(Me, meat), $Pid ! store(Me, cheese).
Me = 97216744.
?- flush.
Shell got 77346122-ok
Shell got 77346122-ok
true.
?- self(Me), $Pid ! take(Me, cheese).
Me = 97216744.
?- flush.
Shell got 77346122-ok(cheese)
true.
?- $Pid ! terminate.
true.
?- flush.
Shell got down(77346122, true)
true.
?-
```

2.4.5 Hiding the details of protocols

In the previous example, we expected programmers to know the details of the protocol that must be followed when interacting with our fridge simulation, which forced them to make raw calls using the send operator in combination with the `flush/0` utility predicate. As suggested by Fred Hébert in his book, that is often a useless burden, and good way around it is to abstract messages away with the help of predicates (or in Hébert's case, functions) dealing with receiving and sending them:

```
store(Pid, Food, Response) :-
    self(Self),
    Pid ! store(Self, Food),
    receive({
        Pid-Response -> true
    }).

take(Pid, Food, Response) :-
    self(Self),
    Pid ! take(Self, Food),
    receive({
        Pid-Response -> true
    }).
```

Calling `receive/1` immediately after having sent the message guarantees that the communication between client and server stays synchronous. Using `store/3` and `take/3`, the interaction with the fridge becomes somewhat easier:

```
?- spawn(fridge([]), Pid, [
       monitor(true)
   ]).
Pid = 55289322.
?- store($Pid, cheese, Response).
Response = ok.
?- take($Pid, cheese, Response).
Response = ok(cheese).
?-
```

When dealing with actors with more complex protocols, such abstractions turns out to be of considerable value.

2.4.6 A universal stateful server with hot code swapping

Inspired by one of Joe Armstrong's lectures on Erlang,[19] this is how we may program a *generic* and *stateful* server in Web Prolog which can also handle *hot code swapping*:

```
server(Pred, State0) :-
    receive({
        rpc(From, Ref, Request) ->
            call(Pred, Request, State0, Response, State),
            From ! Ref-Response,
            server(Pred, State);
        upgrade(Pred1) ->
            server(Pred1, State0)
    }).
```

The first receive clause matches incoming `rpc` messages specifying a goal, performs the required computation, and returns the answer to the client that submitted the goal. The second clause is for upgrading the server.

As can be seen from the first receive clause, the generic server expects the definition of a predicate with four arguments to be present and callable from the server. In the case of our refrigerator simulation the expected predicate may be defined as follows in order to obtain the required specialisation of the generic server:

```
fridge(store(Food), FoodList, ok, [Food|FoodList]).
fridge(take(Food), FoodList, ok(Food), FoodListRest) :-
    select(Food, FoodList, FoodListRest), !.
fridge(take(_Food), FoodList, not_found, FoodList).
```

[19] http://youtu.be/0jsdXFUvQKE

Let us abstract from the message and make sure that the communication between client and server stays synchronous:

```
rpc_synch(To, Request, Response) :-
    self(Self),
    make_ref(Ref),
    To ! rpc(Self, Ref, Request),
    receive({
        Ref-Response -> true
    }).
```

Note the generation of a unique reference marker to be used to ensure that answers pair up with the questions to which they are answers. This code too is generic and follows a common idiom in Erlang that implements a synchronous operation on top of the asynchronous send and the blocking receive. The predicate `rpc_synch/3` waits for the response to come back before terminating. It inherits this blocking behavior from `receive/1`, and it is this behavior that makes the operation synchronous. (We will see more of this pattern later.)

Here is how we start a server process and then use `rpc_synch/3` to talk to it:

```
?- spawn(server(fridge, []), Pid).
Pid = 97216744.
?- rpc_synch($Pid, store(meat), Response).
Response = ok.
?- rpc_synch($Pid, take(meat), Response).
Response = ok(meat).
?-
```

Now, suppose we want to upgrade our server with a faster predicate for grabbing food from the fridge, perhaps one that uses an algorithm more efficient than the sequential search performed by `fridge/4`. Assuming a predicate `faster_fridge/4` is already loaded and present at the node we can make the upgrade without first taking down the server. This means that we can retain access to the state (and thus not risk losing any food in the process):

```
?- $Pid ! upgrade(faster_fridge).
true.
?-
```

Of course, the server can be reprogrammed in a much more radical way, and not only become faster, but also be given a totally different behavior.

2.4.7 Making promises, and keeping them

Remote procedure calls being synchronous means the caller is suspended until the computation terminates and we have to do an idle wait for the answer, although we

may have something better to do. An alternative approach may be to use a so called *promise*, an asynchronous variant of a remote procedure call. To implement this, we can split the client code above into two parts, and thus create two predicates, `promise/3` and `yield/2`:

```
promise(To, Request, Ref) :-
    self(Self),
    make_ref(Ref),
    To ! rpc(Self, Ref, Request).

yield(Ref, Response) :-
    receive({
        Ref-Response -> true
    }).
```

We can now separate the sending of a request from the receiving of the response, thus trading a somewhat messier code for a little bit of concurrency where the caller can perform the RPC, do something else and try to claim the computed value at a later time, when it may (or may not) be ready. Like so:

```
...
promise(To, store(meat), Ref),
... do something else here...
yield(Ref, Response),
...
```

With `promise/3` and `yield/2` defined, we can of course define `rpc_synch/3` as follows instead of as above:

```
rpc_synch(To, Request, Response) :-
    promise(To, Request, Ref),
    yield(Ref, Response).
```

Abstractions such as these can be compared to Erlang *behaviors*. They are not always easy to build, but once they are built they can be fairly easily instantiated and tailored to specific tasks.

2.4.8 Prolog actors playing ping-pong

Since the servers in the previous sections are running in parallel to the shell and are talking to it using asynchronous messaging, we have already demonstrated the use of concurrency. Below, in a probably more convincing example inspired by a user's guide to Erlang,[20] two processes are first created and then start sending messages to each other a specified number of times:

[20] See http://erlang.org/doc/getting_started/conc_prog.html

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

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

ping_pong :-
    spawn(pong, Pong_Pid),
    spawn(ping(3, Pong_Pid)).
```

When `ping_pong/0` is called the behavior of this program exactly mirrors the behavior of the original version in Erlang:

```
?- ping_pong.
true.
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Ping finished
Pong finished
?-
```

This is a concurrent program, but its execution is not done in parallel, but rather as is illustrated in Figure 2.5 where the upper line represent the “pinger” and the lower line the “ponger,” and where each gray bar represent a task consisting of the sending of a message (in light gray) and the reception of the response (in darker gray).

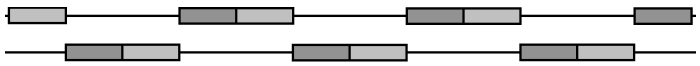

Fig. 2.5

In Appendix B a slightly modified version of this program is used for benchmarking send and receive in Web Prolog. It turns out that if we run this on a 2023 Apple iMac with the M3 chip with `N` set to `100,000` instead of `3` it runs in less than one second. This means that each task is performed in less than 5 microseconds. Appendix B also shows that Erlang is even faster.

2.4.9 Parallel execution of concurrent programs

In this section we will implement `parallel/1`, a meta-predicate that tries to run a list of goals in parallel. A call to `parallel(Goals)` should

1. block until all work has been done, but no longer,
2. succeed, with variable bindings, if all goals succeed,
3. fail as quickly as possible if any goal fails, and
4. rethrow any errors thrown by a goal, also as quickly as possible.

In other words, running a list of goals in parallel should behave in the same way as when running them sequentially, but finish faster.

For `parallel/1` to work properly, we must require something about the input too, namely that goals in the list are independent, i.e. they must not communicate using shared variables, or by any other means.

To make it easier to understand what is going on, we begin by writing a predicate `parallel/2` that only takes *two* goals and spawns *two* actor processes that solve them in parallel.

```
parallel(Goal1, Goal2) :-
    self(Self),
    spawn((call(Goal1), Self ! Pid1-Goal1), Pid1),
    spawn((call(Goal2), Self ! Pid2-Goal2), Pid2),
    receive({Pid1-Goal1 -> true}),
    receive({Pid2-Goal2 -> true}).
```

The actor `Pid1` calls `Goal1` and, if it succeeds, sends the pair of `Pid1` and the (now instantiated) goal term as a message to the parent `Self`. The actor `Pid2` does the same thing for `Goal2`.

If the message sent by `Pid1` reaches the parent's mailbox first, then the first receive clause is triggered, and execution steps to the second receive statement and waits for the second actor's message to arrive. If the message sent by `Pid2` arrives first, then it is deferred. Once the message from `Pid1` comes along, the first receive

statement is satisfied and the program steps to the second receive statement, which is triggered immediately by the deferred message. The result is that when `parallel/2` terminates, the messages will have been received irrespective of the order in which they were sent. The time spent waiting for the call to succeed is the longer of the response times from the two actors. As so often, asynchronous communication using send in combination with the selective receive is key to the kind of programming going on here.

It is important to remember that the goal in the first argument of `spawn/2-3` is always copied before being called. This means that the instantiation of a goal passed to `parallel/2` will not happen until the corresponding call to `receive/1` has selected the message sent from that call.

This program satisfies our requirements 1) and 2), but not 3) and 4). The problem is that if one goal fails or throws an error, the corresponding receive will suspend, waiting in vain for a message of the form `Pid-Goal` to show up. We will explain how to deal with this, but first show how our approach can be generalized into taking a *list* of goals instead of just two. For this, `maplist/3` comes in handy:

```
parallel(Goals) :-
    maplist(par_solve, Goals, Pids),
    maplist(par_yield, Pids, Goals).

par_solve(Goal, Pid) :-
    self(Self),
    spawn((call(Goal), Self ! Pid-Goal), Pid).

par_yield(Pid, Goal) :-
    receive({Pid-Goal -> true}).
```

This, by itself, does not help us satisfy the requirement 3) and 4), but here is a version of the program that does:

```
parallel(Goals) :-
    maplist(par_solve, Goals, Pids),
    maplist(par_yield(Pids), Pids, Goals).

par_solve(Goal, Pid) :-
    self(Self),
    spawn((call(Goal), Self ! Pid-Goal), Pid, [
        monitor(true)
    ]).

par_yield(Pids, Pid, Goal) :-
    receive({
        down(Pid, true) ->
            true ;
        down(_, false) ->
```

```
            tidy_up_all(Pids),
            !, fail ;
        down(_, error(E)) ->
            tidy_up_all(Pids),
            throw(E)
    }),
    receive({Pid-Goal -> true}).
```

In this version, all three predicates from the previous program have been modified. In `par_solve/2` each worker process running a call is now being monitored, and `par_yield/3` (which is `par_yield/2` with the addition of a third argument the purpose of which we shall explain in a bit) is set up to inspect the `down` messages of the form `down(Pid, Reason)` arriving from the worker processes and act appropriately. If `Reason` is `true` nothing needs to be done, if `Reason` is `false` we call `fail` as that will make `parallel/1` fail, and if `Reason` is of the form `error(E)`, `E` is rethrown.

Note the use of anonymous variables in the first arguments of the patterns matching `false` and `error(E)`. Using `Pid` here would work too, but might delay the failure of the call or the rethrowing of an error.

As soon as failure or error is detected, but before failing the entire call or rethrowing the error, the actor running `process/1` has a bit of tidying up to do, which will be performed by a call to `tidy_up_all/1`. For this it needs access to the complete list of pids for the worker actors which has been passed along since it was computed in the first call to `maplist/3` in the body of the clause defining `parallel/1`.

The predicate `tidy_up_all/1` can be implemented as follows:

```
tidy_up_all(Pids) :-
    maplist(tidy_up, Pids).

tidy_up(Pid) :-
    demonitor(Pid),
    exit(Pid, kill),
    mailbox_rm(Pid).

mailbox_rm(Pid) :-
    receive({
        Msg if arg(1, Msg, Pid) ->
            mailbox_rm(Pid)
    },[
        timeout(0)
    ]).
```

Here, one actor process at a time is killed by a call to `exit/2` (and recall that even if it is dead already, no error is raised). To avoid that the death of the process generates an additional `down` message, the monitoring of it must first be turned off. Finally, messages with `Pid` in its first argument that may have reached the mailbox during the execution of `parallel/1` are removed.

It is time to test our program and make sure that we get a speedup and that our four requirements are satisfied. In the following call to `parallel/1`, we simulate goals with long running times by combining simple unifications with calls to `sleep/1`.

```
?- _Goals = [(X=a,sleep(1)),(Y=b,sleep(3)),(Z=c,sleep(2))],
   time(parallel(_Goals)).
% 189 inferences, 0.000 CPU in 3.006 seconds
X = a,
Y = b,
Z = c.
?-
```

Here the call was done in just 3 seconds rather than the 6 seconds it would take if running them sequentially. This, of course, is as good as it gets. In terms of timelines, instead of seeing

Fig. 2.6 One actor performing three tasks sequentially.

we see this:

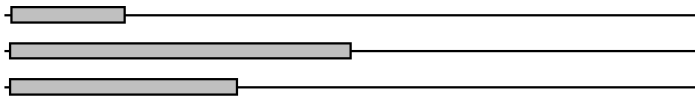

Fig. 2.7 Three actors performing three tasks in parallel.

If one of the goals fail the entire call fails immediately, just as required by 3):

```
?- _Goals = [(X=a,sleep(1)),(Y=b,fail),(Z=c,sleep(2))],
   time(parallel(_Goals)).
% 105 inferences, 0.000 CPU in 0.001 seconds
false
?-
```

And finally, as required by 4), if one of the goals is bad an error is thrown that we can catch:

```
?- _Goals = [(X=a,sleep(1)),(Y=b,sleep(a)),(Z=c,sleep(2))],
   time(catch(parallel(_Goals),E,true)).
% 126 inferences, 0.000 CPU in 0.001 seconds
E = error(type_error(float, a), context(system:sleep/1, _)).
?-
```

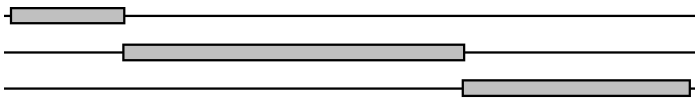

Fig. 2.8 Three actors performing three tasks concurrently, but not in parallel.

Consider the timeline in Figure 2.8. What if we have seen something like that instead? The processing would still be concurrent, but your computer might have just one core. A speedup can only be had if the computer on which the call is made is a multi-core machine.

Another thing to keep in mind is that if the goals in the list execute very quickly, the overhead of running `parallel/1` is likely to reduce any gains that might be had by running them in parallel. To make any noticable difference, the predicate must be fed goals that run for a considerable time.

A predicate such as `parallel/1` should probably be available as a built-in in Web Prolog, and it is encouraging to see that it can be implemented as succinctly as this.

2.4.10 Creating supervision hierarchies

A *supervision hierarchy* refers to a structural pattern commonly used in concurrent and distributed systems, particularly in actor-based models like those found in Erlang. In such a system, actors are organized in a tree-like hierarchy where parent actors supervise their children. This structure is deemed crucial for fault tolerance and system resilience. Supervisors monitor their child actors for errors and decide on strategies (restart, stop, etc.) when errors occur. This ensures that the system can recover from errors locally without affecting the entire system's stability.

Here is code for a simple restarter procedure that takes a goal, a name to be given to the actor executing this goal, and an integer specifying the number of times a restart should be attempted before giving up:

```
restarter(Goal, Name, Count) :-
   spawn(restarter_loop(Goal, Name, Count), _, [
      monitor(true)
   ]).

restarter_loop(Goal, Name, Count0) :-
   spawn(Goal, Pid, [
      monitor(true)
   ]),
   register(Name, Pid),
   receive({
```

```
        down(Pid, true) ->
           true ;
        down(Pid, _Anything) ->
           (   Count0 == 0
           ->  true
           ;   Count is Count0 - 1,
               restarter_loop(Goal, Name, Count)
           )
     }).
```

Here is how it will work ([TODO: expand this!]):

```
?- restarter(echo_actor, echo_actor, 3).
true.
?- self(Self), echo_actor ! echo(Self, hello).
Self = 77129834.
?- flush.
Shell got echo(hello)
true.
?- whereis(echo_actor, Pid), exit(Pid, restart).
Pid = 94203114.
?- self(Self), echo_actor ! echo(Self, hello).
Self = 77129834.
?- flush.
Shell got echo(hello)
true.
```

2.5 Erlang-style programming beyond what Erlang can do

2.5.1 Getting answers through backtracking

In Chapter 2 we stated that, at least in theory, unexpected interactions between language features and possible impedance mismatches between Prolog's relational, non-deterministic programming model and Erlang's functional and message passing model should not cause any problems. As we now turn to more examples that go beyond what Erlang can easily do, we need to see how well the Erlang-style constructs do mix with backtracking for example? In this section we show some examples suggesting that the mix is both sound and easy to understand.

Suppose the query given in the argument to `spawn/2` has several answers, a query such as `?-mortal(Who)` for example. Below, a goal containing this query is called, the first solution is sent back to the calling process, and `receive/1` is then used in order to listen for a message of the form `next` or `stop` before terminating:

```
?- self(Self),
```

```
    spawn(( mortal(Who),
            Self ! Who,
            receive({
                next -> fail ;
                stop -> true
            })
          ), Pid).
Pid = 76123351,
Self = 90054377.
?- flush.
Shell got socrates
true.
?- $Pid ! next.
true.
?- flush.
Shell got plato
true.
?- $Pid ! stop.
true.
?-
```

As this session illustrates, the spawned goal generated the solution `socrates`, sent it to the mailbox of the parent shell process, and then suspended and waited for more messages. When the message `next` arrived, the forced failure triggered backtracking which generated and sent `plato` to the mailbox of the toplevel shell process. The next message was `stop`, so the spawned process terminated. Note that the example demonstrated an actor adhering to what might be seen as a tiny communication protocol accepting only the messages `next` and `stop`.

Looking at the code in the first argument of the calls to `spawn/2` above, this is how we in Prolog often loop over the solutions to a query, using a failure-driven loop rather than a recursive one. Again, the most obvious way to make the code work as expected, is to allow receive to fail.

One needs to observe, however, that the goal to be solved in the above example is hard-coded into the program, that the protocol for the communication between client and actor is overly simplistic, and that neither failure of the spawned goal, nor error thrown by it, are handled. There is clearly a need for something more complete and more generic.

2.5.2 A simple Prolog toplevel actor

In essence, a Prolog toplevel is a failure-driven loop running inside a tail-recursive loop. The failure-driven inner loop allows a client to ask for one solution at a time to a goal, while the outer recursive loop allow it to call several goals in a row and run each of them to completion.

Below, we show how we can build a simple Prolog toplevel by using a meta-predicate (such as `call_cleanup/2`) and by specifying a small set of custom messages carrying answers and/or the state of the process that needs to be returned to the calling process. The predicate `call_cleanup/2` is here used not only to call a goal, but also to check if any choice points remain after the goal has been called or backtracked into.[21]

```
simple_toplevel(Pid) :-
    simple_toplevel(Pid, []).

simple_toplevel(Pid, Options) :-
    self(Self),
    spawn(session(Pid, Self), Pid, Options).

session(Pid, Parent) :-
    receive({
        '$call'(Template, Goal) ->
            (   call_cleanup(Goal, Det=true),
                (   var(Det)
                ->  Parent ! success(Pid, Template, true),
                    receive({
                        '$next' -> fail ;
                        '$stop -> true
                    })
                ;   Parent ! success(Pid, Template, false)
                )
            ;   Parent ! failure(Pid)
            )
    }),
    session(Pid, Parent).
```

A suitabla predicate API hiding the details of the protocol from the programmer can be written like so:

```
simple_toplevel_call(Pid, Template, Goal) :-
    Pid ! '$call'(Template, Goal).

simple_toplevel_next(Pid) :-
    Pid ! '$next'.

simple_toplevel_stop(Pid) :-
    Pid ! '$stop'.
```

Note that the code for `simple_toplevel/1-2` does not say anything about what should happen if an error is thrown, which would for example be the case if the

[21] http://www.swi-prolog.org/pldoc/man?predicate=call_cleanup/2

predicate called by the goal is not defined. If the spawned process is monitored, however, the error message will eventually reach the mailbox of the spawning process anyway, in the form of a `down` message. This means that the actor has terminated.

As we have seen, it is possible for a programmer to access and process different results of a non-deterministic computation from within a program. This is sometimes referred to as *encapsulated search*. But for encapsulated search to become a key feature of the ACTOR profile of Web Prolog, we need to provide something still more generic.

As it turns out, however, and as we shall demonstrate in Chapter 3, Web Prolog programmers do not need to define their own toplevel predicates, but can instead use a set of built-in predicates in order to create and control more powerful Prolog toplevels, which are actors adhering to a standardized and much improved protocol.

Chapter 3
Prolog agents

> Imagine a world of **Prolog agents**, some useful, others playful, bringing joy to games and virtual worlds; some short-lived, others long-lived, some simple, others complex – but maybe built from simpler ones. Written in Web Prolog, talking Web Prolog with other agents, using Web Prolog knowledge bases to guide their actions and conversations, making sure important capabilities of clever conversational agents, such as natural language understanding, knowledge representation, reasoning and real-time interaction, are accounted for.
>
> *Prolog agents – the elevator pitch*

Chapter 1 introduced the notion of a Prolog agent. In Chapter 2 we introduced the more precise concept of a Prolog actor – the most elementary form of Prolog agent in the Prolog Trinity ecosystem. In this chapter, the concept of a Prolog agent is further developed and two important kind of agents, Prolog toplevels and Prolog nodes, will be introduced and their roles in the ecosystem explained.

The reason for referring to both actors and nodes as agents is to focus on their similarities. The similarity that makes use refer to both of them as Prolog agents is that they are both processes that are capable of talking Prolog to other agents. Also, they both live on the Prolog Web.

There are of course differences as well. A node is typically long-lived. It is capable of serving many clients at the same time. It can be programmed, but only by its owner.

3.1 The concept of an agent

In Russell and Norvig [38] an agent is characterized as "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." It does not get more abstract than that, as it covers just about everything from a simple reactive agent such as a thermostat to a complex cognitive agent such as a human.

In this book, however, we shall be somewhat more concrete and care mostly about the concept of a *software agent*, i.e. an agent implemented in software and running

on computer hardware. The concept of software agent might be more appropriately defined as a software process capable of continious interaction with the world external to it. This definition of agenthood is admittedly very simple. It leaves out properties such as autonomy and goal-directedness, properties other researchers use in order to demarcate software agents from ordinary executing programs.[1] By our definition, any running program is an agent, as long as it is an *interactive* program.[2]

Software agents living in web-based environment form a category on its own. We shall refer to them as *web agents*. A web agent, in the context of the internet and web technology, typically refers to a program or script that performs automated tasks or interacts with web services on behalf of a user or another program. Some such agents are also known as *bots* or *web crawlers*. They can serve various purposes, such as web scraping, data collection, automated form submission, or even chatbots that interact with users on websites.

In the context of computer science and artificial intelligence, "spawning an agent" typically refers to the creation or instantiation of a new software agent from an existing one, resembling the parent-child relationship seen in biological reproduction. This process is akin to the notion of giving birth, where a "parent agent" generates a "child agent."

The parent agent is responsible for initiating and managing the child agent's execution. This can involve allocating resources, defining the child agent's initial state, and establishing communication channels between the parent and child agents. The child agent inherits certain characteristics or behaviors from its parent but may also possess its own unique attributes or functionalities.

This concept is frequently encountered in multi-agent systems, distributed computing environments, and parallel processing scenarios, where the dynamic creation of agents allows for increased flexibility and scalability in handling tasks and solving complex problems.

3.2 Prolog agents in a nutshell

A *Prolog agent* is an executing software process on the Prolog Web which is capable of continuous interaction with other Prolog agents in its environment using Web Prolog as a dedicated agent communication language (or an ACL, to use the commonly employed acronym). In this book, we deal primarily with Prolog agents that live on the Web, or in other words, that are *web agents*. Note that if taken as a definition it does not require that a Prolog agent is *written* in Web Prolog, only that it can *talk* Prolog.

---

[1] We agree with Russell and Norvig in [38] when they state that "[the] notion of an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents," so we do not elaborate further on this.

[2] Thus, Turing machines are not agents, since they are not interactive.

As suggested by the taxonomy in Figure 3.1, Prolog agents come in different shapes and sizes, and as suggested by the diagram, they can be arranged into a simple taxonomy.

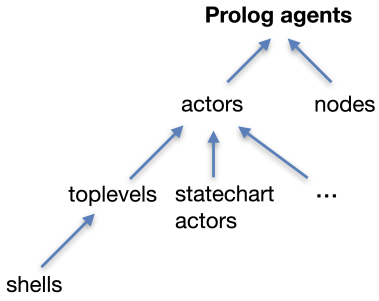

Fig. 3.1 A taxonomy of Prolog agents.

Prolog nodes (or just nodes) serve as the runtime systems of Web Prolog, and in the context of this book they will be regarded as agents. *Prolog actors* – very similar to what the Erlang community refer to as processes – are the loci of computation in the Prolog Trinity ecosystem. Prolog actors satisfy our definition of agenthood since they are processes capable of talking to other processes in their environment. As actors, they are all equipped with a mailbox from which messages received can be selected, and they can send messages to other agents. An actor can also spawn a child actor configured in all sorts of ways by means of options.

There are different *kinds* of actors such as *toplevels* and *statechart actors*, and there is a potential for other kinds of actors with other behaviors, some very specific, others generic. The ... in the diagram is a place holder for future such agents, and also covers actors such as echo actors and count actors. Chapter 2 looked at many examples of such actors in action.

An explanation of what we mean by a *statechart actor* will have to wait until Chapter 6. Suffice it to say that it is a way to program an actor using a (partly) visual programming language called *statecharts*, invented by David Harel (who is featured among the other inventors in Chapter 1). The relevancy to the Prolog Trinity ecosystem is that such actors can use Web Prolog as a data modelling and scripting language.

If toplevels and other actors are considered agents, it follows that they represent a rudimentary form of agency, characterized by minimal complexity in decision-making and autonomy. While these agents exhibit basic properties such as autonomy, interaction, and task-specific behavior, these characteristics alone may not suffice for what might be considered a fully autonomous agent. A more robust agent, such as a "soft robot" interacting with and adapting to its environment, would require additional cognitive capabilities, such as the ability to form desires and intentions, as outlined in the belief-desire-intention (BDI) framework.

Despite this, we remain satisfied with our simple agents, whether they function as nodes, toplevel actors, statechart actors, or other specialized entities. These agents have proven to be incredibly useful in their current form, particularly in web environments where their simplicity allows for ease of deployment and scalability. The utility of these agents is further enhanced by their ability to be composed into more complex systems, enabling the emergence of more sophisticated behaviors.

Moreover, it is important to recognize that "real" agents can be constructed from networks or societies of these simple agents. This modular approach enables the development of more advanced agents, such as those adhering to the BDI model, by building on the cooperative interactions of basic components.

3.3 More about Prolog actor agents

In the Erlang community, many actors, including actors with a very specific functionality such as echo actors and count actors, but also those that are generic such as compute servers, can be described in this way. Of, course, there are clients too, and other programs that cannot be described in this way.

Most actors comes with a communication protocol, i.e. a set of rules and conventions that govern how different actors (and other software processes) interact and communicate with each other.

3.3.1 An actor agent is equipped with a private Prolog database

So far, we have been a bit vague on where the code being executed came from. The only hint given was that it might be stored in the node's *shared database*, defining data and programs that can be used by any actor that lives there. We shall say more about such code further ahead in this chapter, but first we will introduce an area *inside* the actor where code might reside, as well as a way to make data and programs available to an actor.

An actor agent is equipped with a private Prolog database, where programs and data that may only be accessed and used by this particular actor is stored. All actors have such a database, but initially it may be empty. When it is not empty, its content should be contrasted with the programs and data that are shared among all actors that are present on a node, predicates that are loaded into the node's shared Prolog database.

When performing the spawn operation, the (soon-to-become) parent actor can choose to populate the private database of the child actor with new programs and data not present in the node's shared database.

The clauses in this database represent the agent's private beliefs and skills – in contrast to beliefs and skills that it shares with other actor agents running on the same node.

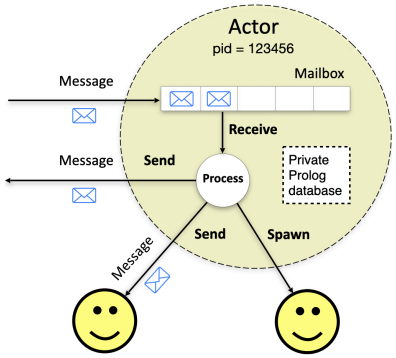

Fig. 3.2 The anatomy of a Prolog actor. Now complete with its private Prolog database.

Consider the following example where the `load_text` option is passed to `spawn/3`, specifying that the source code for our count server should be loaded into the actor's private database before calling the goal in the first argument:

```
?- spawn(count_actor(0), Pid, [
       load_text("
           count_actor(Count0) :-
               receive({
                   count(From) ->
                       Count is Count0 + 1,
                       From ! count(Count),
                       count_actor(Count) ;
                   stop ->
                       true
               }).
       ")
   ]).
Pid = 45092311.
?-
```

In addition to the `load_text` option, three other options can be used to populate the private database of an actor. The `load_list` options loads a list of clauses or directives, the `load_uri` option loads the content specified by a URI, and `load_predicates` takes a list of predicate indicators and loads the clauses for the indicated predicates that are accessible by the caller.

It is possible to pass an arbitray number of the `load_*` options to `spawn/3`, and possibly more than one instance of each. To ensure that clauses end up in a well-defined order, they will all be converted into Prolog source text before finally being loaded into the database. The order of clauses and directives in the source text

is determined by the order of the `load_*` options in the list of options passed to `spawn/3`.

Note that the code that will eventually be executed by the call to `spawn/3` depends not only on what has been loaded into the private database of an actor, but also on the contents of the node's shared Prolog database. We shall return to this later in this chapter.

—

3.3.2 The life-cycle of a typical Prolog actor agent

> We have the black boxes that do things, and we have the messages in between. What goes on inside the black boxes does not really matter; what takes place there is abstracted over.
>
> *Joe Armstrong*

The diagram in Figure 3.3 illustrates the life-cycle of a typical Prolog actor.

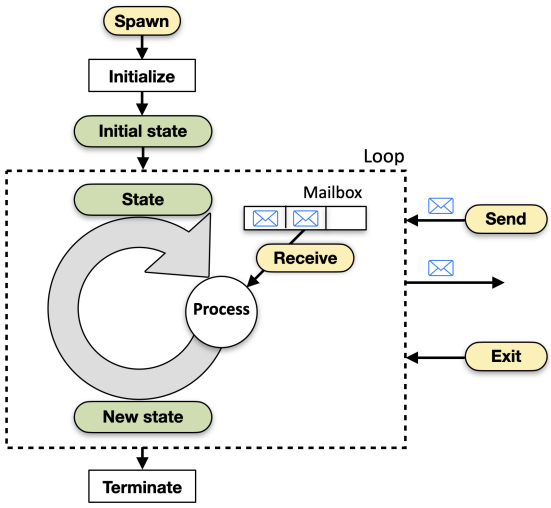

Fig. 3.3 The life-cycle of a Prolog actor.

When our count actor was spawned, the source code specified by the `load_text` option was loaded into the actor's private Prolog database. The pid signaling that the actor exists is returned.

The actor is now in its *initial state*, and the goal in the first argument of `spawn/3` (`count_actor(0)`) is called.

When two processes communicate over a protocol, interaction is driven by coordinating send and receive operations, with the protocol defining the rules. The sending process transmits messages in the specified format, and the receiving process captures and interprets them.

The protocol defines message formats, sequencing, and behaviors, ensuring synchronization and accurate data transfer. Discrepancies between sender and receiver expectations can lead to breakdowns, which the protocol should mitigate through error-handling or synchronization mechanisms. The Web Prolog receive operator, while blocking the process at the point of a `receive/1-2` call, allows for more flexible handling through mechanisms like selective pattern matching and timeouts. This means the process waits for a matching message but can specify a timeout to avoid being blocked indefinitely, enabling alternative actions if an expected message does not arrive in time. This helps reduce the chance of breakdowns and allows for more adaptive communication.

Effective communication relies on both processes adhering to shared expectations for reliable data exchange.

Whenever the actor runs a `receive/1-2` call, a message is selected from the mailbox and processed sequentially. Typically, this takes place in a loop. The result often depends on the state of the process, and as part of processing, a new state for the actor may be computed. This will be the current state for processing the next message selected.

The loop implements the communication protocol that clients that want to interact with the actor must follow.

3.3.3 The dynamic (and still private) Prolog database

As long as the actor process is alive, it is allowed to update its private database using predicates such as `assert/1`, `retract/1` and `retractall/1`. Following the ISO standard, predicates that are modified this way need to be declared using the `dynamic/1` directive. Updates will only affect that actor's workspace, not actors running elsewhere, and not even actors running on the same node.

This means that problems caused by two or more processes trying to update the same database simultaneously cannot arise. It also means that since database updates performed by one process is not seen by other processes, assert and retract cannot be used for inter-process communication. Web Prolog adheres to the idea that processes should "communicate to share memory, rather than share memory to communicate".[3]

Therefore, although the following goal is permitted, it is quite meaningless since the clause `foo(a)` disappears as the goal has run and the spawned process has terminated.

```
?- spawn(assert(foo(a)), Pid).
```

[3] A mantra attributed to CSP.

```
Pid = 32861299.
?-
```

The dynamic database provides another way to transfer from one state to the next. Here is an example:

```
?- spawn(count_actor, Pid, [
       load_text("
           :- dynamic cnt/1.

           cnt(0).

           count_actor :-
               receive({
                   count(From) ->
                       retract(cnt(Count0)),
                       Count is Count0 + 1,
                       From ! count(Count),
                       assert(cnt(Count)),
                       count_actor ;
                   stop ->
                       true
               }).
       ")
   ]).
Pid = 99801234.
?-
```

In this case, the use of the dynamic database is a very bad idea.

3.4 Prolog shells and other toplevel actors are agents

As we saw already in Chapter 2, if `self/1` is called in a terminal, its argument gets bound to an pid, and this pid points to a Prolog *shell*:

```
?- self(Pid).
Pid = 72097632.
?-
```

Here, the actor with the pid `72097632` is a shell. Shells are toplevels, and therefore also agents. Shells handle all the things that toplevels handle, but also adds a number of useful things for when we are talking to Prolog over a terminal. This includes I/O (read and write) and utilities such as `flush/1` and the dollar notation. Such features are supplied by the *node controller* rather than a toplevel alone. (We will say more about the node controller further ahead in this chapter.)

Here is how we instruct the shell to spawn a new toplevel:

```
?- toplevel_spawn(Pid).
Pid = 16226587.
?-
```

Note that `16226587` is a toplevel actor but not a shell. As with every other Prolog actor, it comes with a mailbox, a private Prolog database, the ability to send messages to other actors, as well as the ability to create other actors. The feature that distinguishes a toplevel from other actors is that it comes with a standardized built-in special-purpose communication protocol, the PTCP.

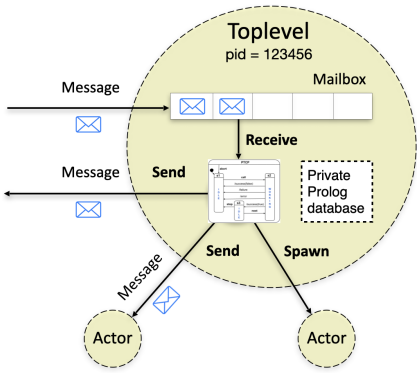

Fig. 3.4 The anatomy of a Prolog toplevel actor.

3.4.1 A Prolog toplevel is an actor with a built-in protocol

A programmer firing up a traditional Prolog system is likely met with a query prompt. In the literature, this is usually referred to as the *toplevel*. The reason we refer to it a *shell* is because we want to use the term *toplevel* to refer to toplevels that are not shells.

In traditional Prolog, a program cannot *internally* create a toplevel, pose queries and request solutions on demand, but this is something that Web Prolog allows. In traditional Prolog the toplevel is *lazy* in the sense that new solutions to a query are only computed on demand. As we shall see, this is how the toplevel actor in Web Prolog works too.

In Web Prolog, a toplevel actor is a programming abstraction modelled on the interactive toplevel of Prolog. A toplevel actor is like a first-class interactive Prolog toplevel, accessible from Web Prolog as well as from other programming languages such as JavaScript. We can also think of it as an *encapsulated Prolog session*, an abstraction aiming at making Prolog programmers feel right at home.

A toplevel is a kind of actor, and what distinguishes it from other kinds of actors is the *protocol* it follows when it communicates, i.e. the kind of messages it listens for, the kind of messages it sends and in what order, and the behavior this gives rise to.

The protocol must not only allow a client to submit queries and a toplevel to respond with answers, it must also allow the toplevel to prompt for input or produce output at any time, in an order and with a content as dictated by the program that it runs. All toplevels follow this protocol. The terminal adheres to it as well, and even a human user of a terminal talking to a shell must adapt to it in order to have a successful interaction.

The design of the toplevel actor in Web Prolog is in fact very much inspired by the informal communication protocol that we as programmers adhere to when we invoke a Prolog shell from our OS prompt, load a program, submit a query, are presented with a solution (or a failure or an error), type a semicolon in order to ask for more solutions, or hit return to stop. These are "conversational moves" that Prolog understands. There are even more such moves, since after having run one query to completion, the programmer can choose to submit another one, and so on. The session does not end until the programmer decides to terminate it. There are only a few moves a client can successfully make when the protocol is in a particular state, and the possibilities can easily be described, by a state machine for example, as will be shown in the next section.

3.4.2 The Prolog Toplevel Communication Protocol

Figure 3.5 depicts a statechart specifying the Prolog Toplevel Communication Protocol (PTCP) – a protocol for the communication between a client and a toplevel actor. The client can be any process (including another actor or (say) a JavaScript process) capable of sending the messages and signals in bold to the toplevel. The toplevel actor is responsible for returning the messages with a leading / back to the client. The use of a statechart allows us to show that no matter the current state of the protocol, **abort** will always take it to the state from which a new goal can be called and **exit** will always terminate the toplevel process.

A process is able to *spawn* a toplevel. After having done so, the process becomes the parent of the toplevel and can start communicating with it according to the protocol. Initially, the protocol is in state **s1**, where the toplevel rests idle, waiting for a **call** message containing a goal. When such a message arrives, the protocol transitions to state **s2**, where the toplevel is actually doing work. The protocol will remain in state **s2** until some work is done and the toplevel sends a message indicating either `/success`, `/failure`, `/error`, or (if the process is monitored) `/down` to the client. On the event of the toplevel sending `/failure` or `/error`, the protocol will transition back to state **s1**. This will be the case also for a `/success` message that indicates no more solutions to the query can be found (marked with false in the chart). However, if a `/success` message indicates more solutions may exist (marked

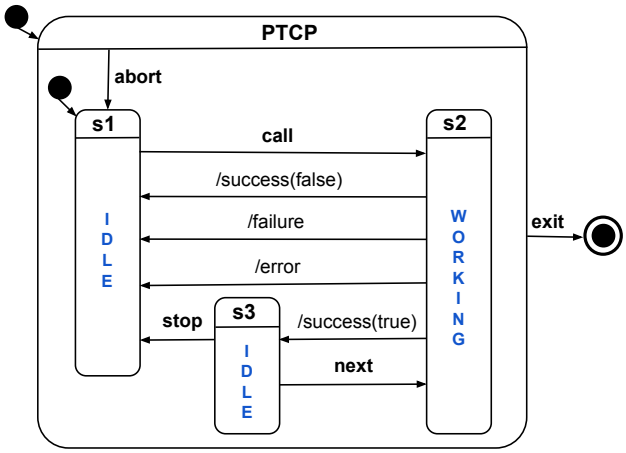

Fig. 3.5 Statechart specifying the PTCP for a successful conversation with a toplevel. The transitions are labeled with *message types*. Types in bold are sent from the client to the toplevel, whereas message types with a leading / goes in the opposite direction, from the toplevel to the client.

with true in the chart), the protocol transitions to state **s3**, where it will wait for a message **next**, **stop**, **abort** or **exit** to arrive from the client. If the message is **stop** or **abort** the protocol will transition back to state **s1**, if it is **next** it will transition to state **s2** and trigger the search for more solutions, and finally, if it is **exit**, it will (if the process is monitored) force the toplevel to send a message `/down` back to the client and then to terminate. Any other messages will be deferred to the program being executed by the toplevel.

Web Prolog comes with six built-in predicates which allow a client to spawn a toplevel actor and interact with it through its life-cycle:

```
toplevel_spawn/1-2    toplevel_call/2-3    toplevel_next/1-2
toplevel_stop/1-2     toplevel_abort/1     toplevel_exit/1
```

Such predicates are defined in terms of the more primitive `spawn/1-3`, `!/2` and `receive/1-2` predicates, but they also comes with a number of new options. Their uses will be demonstrated in the next section, and Appendix A contains an excerpt from the (draft) manual which covers most of the details of our proposal.

3.4.3 Shell talking to a Prolog toplevel

Below, we show an example of how to create and interact with a toplevel process from a shell. We start by spawning a new toplevel, using `toplevel_spawn/3` with

the `monitor` option to instruct the toplevel to send us a `down` message when the process eventually terminates. The `load_list` option is used to populate the private Prolog database belonging to the toplevel actor with two simple unit clauses `p(a)` and `p(b)`:

```
?- toplevel_spawn(Pid, [
       monitor(true),
       load_list([p(a),p(b)])
   ]).
Pid = 74122981.
?-
```

With this, the toplevel actor is initiated, and with the PTCP now in state **s1**, it is ready to accept messages and signals sent by the other `toplevel_*` predicates.

The `monitor` and `load_list` options are inherited from `spawn/2-3`. However, `toplevel_spawn/3` offers two new options that can be used to specify the behavior of toplevels that are spawned. The `session` option configures the toplevel to run a multi-query session rather than exit after just one query. It is true by default. The `target` option can be used to redirect the answer messages arriving from the toplevel to any actor of choice. By default, it is set to the pid of the parent. We say more about their use further ahead in this chapter.

3.4.3.1 Making the toplevel call a goal

Let us see what happens if we call `toplevel_call/2` with the default values for options:

```
?- toplevel_call($Pid, p(X)).
true.
?- flush.
Shell got success(74122981,[p(a),p(b)],false)
true.
?-
```

The answer is returned to the mailbox of the calling process in the form of a Prolog term with three arguments. The functor of the answer term represents its *type*. In this case, it shows that the goal succeeded. (In other cases it might indicate a failure or an error.) The first argument of the success term is the pid, and the list in the second argument represents the two solutions that was computed. The value `false` in the third argument of the term indicates that no more solutions exist.

After a brief visit to state **s2** for the execution of the goal, the PTCP is now back in state **s1**.

3.4.3.2 Using the template option

Below, `toplevel_call/3` is called with the goal `p(X)` and with the `template` option set to the variable `X`:

```
?- toplevel_call($Pid, p(X), [
       template(X)
   ]).
true.
?- flush.
Shell got success(74122981,[a,b],false)
true.
?-
```

Note how the value of the `template` option determined the form of the list of solutions in the second argument of the answer term. The relation to the Prolog built-in standard predicate `findall/3` is evident.

3.4.3.3 Using the offset and limit options

When querying a relational database using SQL or an RDF dataset using SPARQL, the use of `OFFSET` and `LIMIT` serves to control the subset of results that are returned by a query.

The `limit` option specifies the maximum number of solutions to return, while the `offset` option specifies the number of solutions to skip before starting to return solutions.

```
?- toplevel_call($Pid, between(1,infinite,I), [
       template(I),
       offset(100),
       limit(3)
   ]).
Pid = 74122981.
?- flush.
Shell got success(74122981,[101,102,103],true)
true.
?-
```

Calling `toplevel_next/1` produces three more solutions:

```
?- toplevel_next($Pid).
true.
?- flush.
Shell got success(74122981,[104,103,106],true)
true.
?-
```

However, `toplevel_next/2` accepts the `limit` option too:

```
?- toplevel_next($Pid, [
       limit(5)
   ]).
true.
?- flush.
Shell got success(74122981,[107,108,109,110,111],true)
true.
?-
```

...

```
?- toplevel_stop($Pid).
true.
?-
```

These options are particularly useful in the following scenarios:

- Pagination of results: When dealing with a large dataset, it is often impractical to retrieve and display all the records at once, especially in web applications. Instead, results are divided into pages, with each page showing a smaller subset of the data.
- Efficient data retrieval: In cases where only a portion of the dataset is needed for processing, using `offset` and `limit` helps in efficiently retrieving just that portion, reducing the load on the node and the network.
- Implementing infinite scrolling: In user interfaces that implement infinite scrolling (such as social media feeds), `offset` and `limit` can be used to load more data dynamically as the user scrolls down the page.
- Handle an infinite number of solutions:
- Avoiding recomputation:

A special case of pagination is when we choose to set `limit` to `1` in order to make it much easier to implement a traditional Prolog terminal in JavaScript.

3.4.3.4 A toplevel can do a kind of I/O

Whenever a toplevel is in state **s2** and doing some real work, it is able to send messages to its parent using the usual send operator. However, a better idea can often be to use `output/1`, a built-in predicate that in its simplest form can be defined like so:

```
output(Message) :-
    self(Self),
    parent(Parent),
    Parent ! output(Self, Message).
```

Since the message is wrapped in a binary term with the pid of the toplevel in its first argument, the parent will always know if the message came from the right toplevel.

There is also `output/2`, which expect the option `target` to specify a target different from the parent.

(By the way, `output/1-2` can be used from any actor, not only from a toplevel.)

Now, let us demonstrate how I/O works:

```
?- toplevel_call($Pid, output(hello)).
true.
?- flush.
Shell got output(74122981,hello)
Shell got success(74122981,[output(hello)],false)
true.
?-
```

Input can be collected by calling `input/2`, which sends a `prompt` message to the client, which in turn can respond by calling `respond/2`:

```
?- toplevel_call($Pid, input('Input', X)),
   receive({Answer -> true}).
Answer = prompt(74122981,'Input').
?- respond($Pid, hello),
   receive({Answer -> true}).
Answer = success(74122981,[input('Input',hello)],false).
?-
```

3.4.3.5 Aborting a non-terminating goal

The toplevel is still not dead so let us see what happens when we ask the toplevel to first update its private Prolog database with a silly recursive clause for a predicate `p/0` and then call it:

```
?- toplevel_call($Pid, assert((p :- p))),
   receive({Answer -> true}).
Answer = success(74122981,[assert((p:-p))],false)
?- toplevel_call($Pid, p).
true.
?-
```

Although nothing is shown in the terminal, it is clear that the toplevel is now just wasting CPU cycles to no avail. Fortunately, a non-terminating goal can be aborted by calling `toplevel_abort/1`:

```
?- toplevel_abort($Pid).
true.
?-
```

With this, the PTCP is back in state **s1**.

3.4.3.6 Redirecting answers

The option `target` allows us to instruct a toplevel actor to send answer terms to a destination different from the parent. To demonstrate how this works, we first create a simple actor that can serve as a target:

```
?- spawn(( repeat,
           receive({
               Msg ->
                   format("Received ~p~n", [Msg]),
                   fail
           })
         ), Pid0).
Pid0 = 98380209.
```

Its pid is used as the value of the target option:

```
?- toplevel_spawn(Pid, [
       target($Pid0)
   ]).
Pid = 97919106.
?-
```

The success term is printed, and calling `flush/0` shows that the mailbox belonging to the shell is empty:

```
?- toplevel_call($Pid, between(1,1000,N), [
       template(N),
       limit(5)
   ]).
true.
Received success(97919106,[1,2,3,4,5],true)
?- flush.
true.
?-
```

The `target` option is supported by all `toplevel_*` predicates, which allows us to direct the answer terms back to the shell. The only exception is `toplevel_stop/1`:

```
?- toplevel_stop($Pid).
Self = 34712309.
?- flush.
true.
?-
```

3.4.3.7 Exiting the toplevel

When we are done talking to the toplevel we can kill it. As the `monitor` option was set to `true`, we should expect to receive a `down` message:

```
?- toplevel_exit($Pid, goodbye),
   receive({Answer -> true}).
Answer = down(74122981,goodbye).
?-
```

Clearly, a Prolog toplevel is a kind of server (in the sense of Erlang – see Section 2.4.3). Also, note that a toplevel, even when not explicitly threading any state, nor using the dynamic database, must still be considered stateful. Rather than using an explicit data structure for holding the state, it is the underlying Prolog process *as such* that holds it, most clearly shown in the way the toplevel "remembers" its history and how its behavior is influenced by this, enabling it to react appropriately when a client requests the next solution to a query.

3.4.4 Toplevels and the message deferring mechanism

In the following example, a toplevel is spawned, and then `toplevel_next/1` is called. The protocol is obviously not in a state where it can react on the `next` message (see Figure 3.5). The message is therefore deferred and it is not until `toplevel_call/3` is called and the protocol changes states that it has an effect.

```
?- toplevel_spawn(Pid).
Pid = 78340943.
?- toplevel_next($Pid).
true.
?- flush.
true.
?- toplevel_call($Pid, member(X,[a,b,c]), [
       limit(1),
       template(X)
   ]).
true.
?- flush.
Shell got success(78340943, [a], true)
Shell got success(78340943, [b], true)
true.
?-
```

Note that messages sent to a toplevel will often (always?) be handled in the right order even if they arrive in the "wrong" order (e.g. `next` before `call`). This is due to the selective receive which defers their handling until the PTCP protocol permits it.

The messages `abort` and `exit`, however, will never be deferred. This is guaranteed by the fact that **abort** and **exit** are valid transitions from any state in the protocol (see Figure 3.5). (They are actually signals rather than messages).

One way to think of this is in terms of the so called *robustness principle*: "Be conservative in what you send, be liberal in what you accept."[4] Due to the deferring behavior a toplevel is liberal in this way, but, as implied by the principle, clients are advised not to rely on this behavior.

3.4.5 Reconstructing findall/3

If we did not already have a built-in predicate `findall/3` we could have implemented it like so:

```
findall(Template, Goal, Solutions) :-
    toplevel_spawn(Pid, [
        session(false)
    ]),
    toplevel_call(Pid, Goal, [
        template(Template),
        limit(none)
    ]),
    receive({
        success(Pid, Solutions, false) ->
            true ;
        failure(Pid) ->
            Solutions = [] ;
        error(Pid, Error) ->
            throw(Error)
    }).
```

It may not make much sense to do it like this, but at least it says something about the relation between `findall/3` and the `toplevel_*` predicates.

3.4.6 A synchronous predicate API to toplevels

The communication between a client and a toplevel illustrated above is asynchronous, but if we want, it is quite easy to define a synchronous alternative using a technique we have already seen. A predicate `wait_for_answer/3` can be defined like so:

```
wait_for_answer(Pid, Answer, Options) :-
    receive({
```

[4] https://en.wikipedia.org/wiki/Robustness_principle

```
            Answer if arg(1, Answer, Pid) ->
                true
        }, Options).
```

Then `toplevel_call_synch/3-4` can be implemented as follows:

```
toplevel_call_synch(Pid, Goal, Answer) :-
    toplevel_call_synch(Pid, Goal, Answer, []).

toplevel_call_synch(Pid, Goal, Answer, Options) :-
    toplevel_call(Pid, Goal, Options),
    wait_for_answer(Pid, Answer, Options).
```

and `toplevel_next_synch/2-3` like so:

```
toplevel_next_synch(Pid, Answer) :-
    toplevel_next_synch(Pid, Answer, []).

toplevel_next_synch(Pid, Answer, Options) :-
    toplevel_next(Pid, Options),
    wait_for_answer(Pid, Answer, Options).
```

Note that since `toplevel_stop/1` does not produce a response message, no `toplevel_stop_synch/2-3` is defined, and `toplevel_stop/1` can be used as it is.

Here is a simple test that shows how it works:

```
?- toplevel_spawn(Pid).
Pid = 67590967.
?- toplevel_call_synch($Pid, mortal(X), Answer, [limit(1)]).
Answer = success(67590967, [mortal(socrates)], true).
?- toplevel_next_synch($Pid, Answer).
Answer = success(67590967, [mortal(plato)], true).
?- toplevel_next_synch($Pid, Answer).
Answer = success(67590967, [mortal(aristotle)], false).
?-
```

3.5 Prolog nodes are also agents

As evident in Figure 3.1, we think of a Prolog node too as a kind of Prolog agent. The reason is that a node features a significant subset of the capabilities of a toplevel. In particular allowing clients to submit queries and getting solutions returned one at a time and in the order that Prolog finds them. A major difference between a node agent and an actor agent is that a node agent is capable of dealing with many clients at the same time.

The diagram in Figure 3.6 depicts the anatomy of an executing Prolog node accessible by anyone who knows its URI and is authorized to make use of its capabilities.

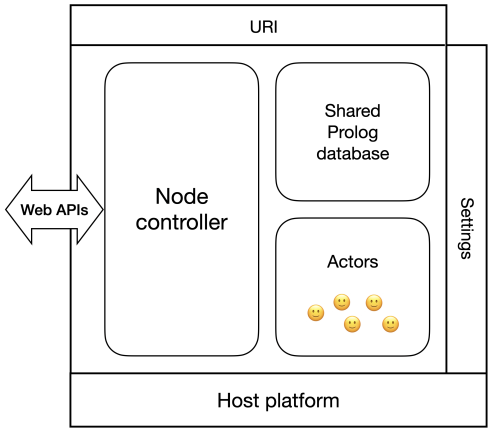

Fig. 3.6 The anatomy of a Prolog node.

As suggested by the diagram, a node consists of components such as a node controller, web APIs, a shared database holding node-resident programs and data, a container populated by actors, and a number of various settings allowing its owner to configure the node. There is also the system hosting the node, a Prolog system for example.

A Prolog node announces itself as a supporter of a particular *profile*. A profile is a set of capabilities. Some are language capabilities, others are interactional capabilities. The capabilities of nodes vary, and only the most capable nodes have all the components shown in the diagram. As we shall see in more details further ahead, the most capable kind of node is equipped with comprehensive web APIs using WebSocket and HTTP as transport protocols, over which a client with access to the URI can create actors running Web Prolog programs defined by the owner of the client, the owner of the node, or by contributions from both. Less capable nodes support only the HTTP API. We shall say more about profiles in Chapter 4.

In obvious contrast to a locally running Prolog process accessed over a traditional terminal, a node is normally (but not necessarily) capable of serving multiple clients simultaneously. In this respect, a node can be compared to an ordinary web server. Just like a web server, a node on the Prolog Web has an owner – a person or an organization responsible for the resources (including programs, data and compute) offered by the node, where the ownership is signaled by the domain of the URI.

To make this as concrete as possible, let us assume that someone uses Trinity Prolog to set up a node, using the following command:

```
$ tp-node --port=80 --src=../src/n7.pl --settings=s.pl
```

As a result, the node is created and access to it is given through a set of different endpoints:

Node endpoint	Target
`http://n7.org`	The shared Prolog database
`http://n7.org/call`	The stateless HTTP API
`ws://n7.org/actor`	The stateful WebSocket API
`http://n7.org/shell`	A simple shell environment

Table 3.1 Proposal for how to link URIs with targets.

Note that since a Prolog node is also a web server, it may offer other HTTP endpoints as well, some of which may not be relevant to its core functionality.

As always, the use of HTTPS in combination with a suitable SSL/TLS certificate to ensure secure access to a node is recommended. For the same reason, WSS is recommended.

3.5.1 The shared Prolog database

A node may host a set of Prolog predicates. The predicates – which will be referred to as the *node-resident* predicates – are typically maintained by the owner of the node. Any client connected with the node has access to these predicates in addition to the Web Prolog built-in predicates, and can pose queries over the web APIs formulated in terms of them.

The contents of the file `n2.pl` might be as follows:

```
:- use_module(philosopers, [philosopher/1]).

mortal(Who) :- human(Who).

human(socrates).
human(Who) :- philosopher(Who).
```

This is also what we get if we steer an ordinary browser to the URI `http://n2.org`. The clauses for `philosopher/1`, however, are not necessarily available.

3.5.2 The node controller

The role of the *node controller* is to:

- handle authentication and authorization of clients,
- manage WebSocket and HTTP connections,

- spawn actors requested by clients,
- impose resource restrictions on actors,
- manage registry, monitors and links,
- make sure messages returned to the client have the right format,
- terminate actors that no longer have a connection to a client,
- implement shell utilities,
- handle logging, etc.

The main task of the node controller is to support the web APIs. It consists of two components that are operating independently of each other, one component supporting the stateful WebSocket API, the other component supporting the stateless HTTP API.

Resource restriction on the size of an actor's private Prolog database, the time the actor is allowed to run, etc. Resource restrictions are specified by the values of settings decided by the owner of the node.

Clients cannot influence the workings of the node controller, only the node's owner can do that through various settings. In fact, from the outside the workings of the node controller is supposed to invisible, like an "ether" through which communication or information is transmitted in a more or less invisible manner.

3.5.3 The stateful WebSocket API

In order to enable a client to control all aspects of a set of toplevels and other actors, the most capable type of node offers a WebSocket sub-protocol. WebSocket is a real-time, low latency, bi-directional protocol for asynchronous communication between a client and a server. The WebSocket sub-protocol is as an application-layer protocol, which operates on top of the WebSocket protocol, leveraging the transport services provided by TCP. The WebSocket protocol differs from TCP in that it enables a stream of messages instead of a stream of bytes. With websockets, data is always sent as a whole message. The receipt of a message is event driven and the data payload in an event is always the entire message the other side sent.

For Web Prolog, all of this works to our advantage since we specifically want to use the message passing type of paradigm that the WebSocket protocol offers. For example, being already message-based, websockets save us from most of the work involved in implementing a suitable protocol.

The three phases in the life-cycle of a WebSocket connection is illustrated in Figure 3.7.

In the first phase (in blue in the diagram) a WebSocket connection is initiated via an HTTP connection which is then switched over to the WebSocket protocol. Websockets can operate on the same port as a Prolog node and there is a vast infrastructure for HTTP and HTTPS that already exists (proxies, firewalls, caches, and other intermediaries), ensuring that security requirements of the modern web are fulfilled and the browser and the node can validate each other.

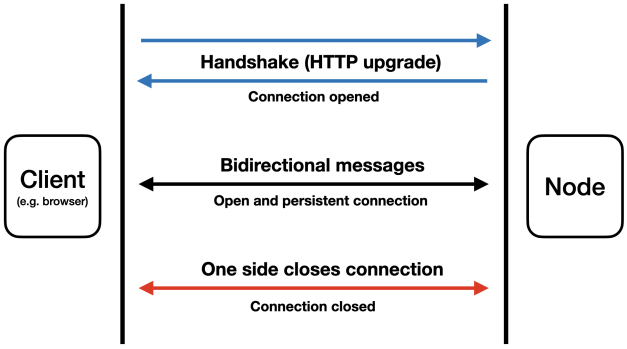

Fig. 3.7 The life-cycle of an interaction beween a client and a node over a WebSocket connection.

In the second phase (in black), the client and the node starts talking to each other. This where the sub-protocol comes in. Our proposal for a WebSocket sub-protocol mirrors the PTCP but they are not identical. The sub-protocol significantly bigger and messages are rendered as JSON. By design, the messages understood by the sub-protocol matches the `toplevel_*` predicates quite well, but there are messages for spawning ordinary actors and handle their messaging too. Table 3.2 shows just a part of it.

Command	Description	JSON Field	Type	Required
`toplevel_spawn`	Spawn a toplevel actor	`options`	string	No
`toplevel_call`	Request solutions for goal	`pid`	integer	Yes
`toplevel_call`	Request solutions for goal	`goal`	string	Yes
`toplevel_call`	Request solutions for goal	`options`	string	No
`toplevel_next`	Request more solutions	`pid`	integer	Yes
`toplevel_next`	Request more solutions	`options`	string	No
`toplevel_stop`	Request stop	`pid`	integer	Yes
`toplevel_stop`	Request stop	`options`	string	No
`toplevel_abort`	Abort running goal	`pid`	integer	No

Table 3.2 WebSocket Protocol JSON Message Structure

Here is an example of a message:

```
{  "command": "toplevel_spawn",
   "options": "[session(true),format('json-s')]"
}
```

Note that the `format` option is not a valid option in the predicate API.
Subject to the `format` option, answers and other kinds of messages arriving from the node are returned in the form of JSON or Prolog text. Client code written

in languages other than Prolog would normally request that they be encoded in JSON. The pid of the created toplevel is returned to the client, not in the form of the binding of a variable (as in the predicate API), but as a message of the form `{"type":"spawned","pid":<pid>}`.

The full set of options valid in the predicate API is valid in the web API as well. Since the options for `toplevel_spawn/2`, `toplevel_call/3` and `toplevel_next/2` are documented in Appendix A.2, we do not give the details here. A brief reminder should be sufficient. In the message corresponding to `toplevel_spawn/2` we can use options such as `exit`, `monitor`, `link`, `timeout`, `load_list`, `load_text`, `load_uri` and `load_predicates`. (Recall that the majority of these options are inherited from `spawn/3`.) For `toplevel_call/3` we can use `template`, `offset` and `limit`. For `toplevel_next/2` there is only one valid option, namely `limit`.

WebSocket client libraries are available for the most common general programming languages, SWI-Prolog included. However, the most common use of websockets is in communication between a web browser and a server. In all the major web browsers, the WebSocket object provides a JavaScript API for creating and managing a WebSocket connection to a server, as well as for sending and receiving data on the connection.[5] The role of JavaScript here is to create a new websocket, to define the necessary handlers for `onopen`, `onmessage`, `onerror` and `onclose` messages, and to call the methods `send` or `close` for sending messages or closing the connection. A synopsis that leaves out a lot of details can be given as follows:[6]

Constructor

```
let connection = new WebSocket(<URI>[,<protocol>);
```

Event listeners

```
connection.onerror = function(message) {...}
connection.onopen = function(message) {...}
connection.onmessage = function(message) {...}
connection.onclose = function(message) {...}
```

Methods

```
connection.send(message)
connection.close()
```

Examples:

```
connection.send(JSON.stringify({
    command: "toplevel_spawn",
    options: "[session(true), format('json-s')]"
}
```

...

[5] See https://www.w3.org/TR/websockets/

[6] For all the gory details as well as a good general introduction to WebSocket we recommend [31].

```
connection.onmessage = function(message) {
    var msg = JSON.parse(message.data);
    if (msg.type == "spawned") {
        pid = msg.pid;
    } else if (msg.type == "success") {
        if (msg.more) {
            ...
        } else {
            ...
        }
    } else if (msg.type == "failure") {
        ...
    } ...
    ...
}
```

3.5.4 Browser talking to a toplevel running on a node

Figure 3.8 illustrates a scenario in which a user engages with a Prolog toplevel process running on a node via a dedicated terminal implemented in JavaScript and executed within a web browser. The terminal's display captures the remnants of a session during which the user executed four queries. These queries were processed by the toplevel in the context of the shared Prolog database, supplemented by a contribution provided by the user.
A JavaScript program running in the browser initiates a WebSocket connection by creating a new `WebSocket` object.

```
let connection = new WebSocket("ws://n7.org/ws", "ptcp-0.1");
```

[TODO: Consider /ws -> /actor —
The browser sends a WebSocket handshake request to the node over HTTP. This request is upgraded to the WebSocket protocol. The node receives the handshake request and, if it accepts the connection, it sends back a handshake response. Once the handshake is complete, the connection is open and the client and node can now start sending and receiving messages using the protocol.
—

A description of the step-by-step evolution of the scenario in Figure 3.8 can be given as follows. After having established a websocket connection with the node, the client makes a request of the form:

```
{ "command": "toplevel_spawn",
  "options": "[session(true),format(json-s)]" }
```

This causes its node controller to spawn a toplevel and return its pid to the client in the form of a message rendered as JSON:

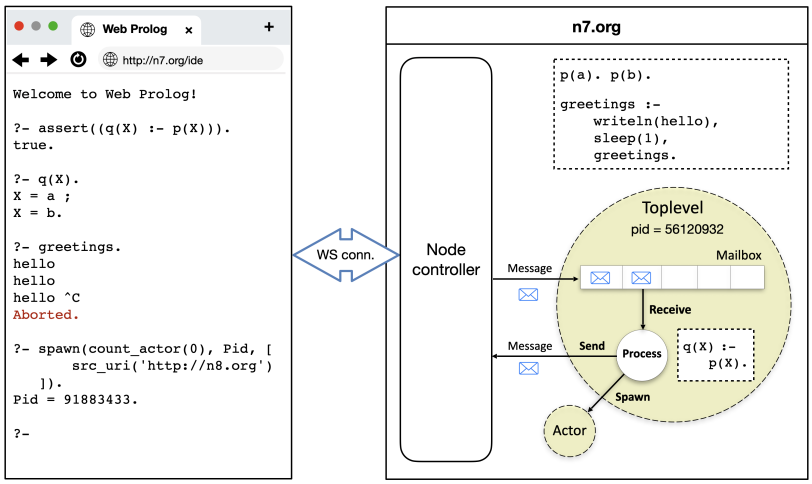

Fig. 3.8 A scenario involving a web browser in conversation with a remote node running a toplevel.

```
{ "type": "spawned",
  "pid" : 56120932 }
```

In the GUI, this produces the prompt `?-` which invites the user to enter a query

```
?- assert((q(X) :- p(X))).
```

which sends the following message to the node controller:

```
{ "command": "toplevel_call",
  "pid": 56120932,
  "goal": "assert((q(X):-p(X)))",
  "options": "[limit(1)]" }
```

It makes the toplevel add the clause `q(X) :- p(X)` to its private workspace and return the following message to the client:

```
{ "type": "success",
  "pid" : 56120932,
  "data": [{}],
  "more": false }
```

The user then submits the query `?-q(X)`, which the toplevel answers with a message encoding the first solution:

```
{ "type": "success",
  "pid" : 56120932,
  "data": [{"X":"a"}],
  "more": true }
```

The value `true` of the `more` property signifies that more solutions may be available. Asking for the next solution results in a similar message, except that the value of the `data` property becomes `[{"X":"b"}]` and the value of `more` becomes `false`.

Note that it takes the *combination* of the clause that was asserted into the toplevel's private database, and data and programs in the shared Prolog database to answer the query `?-q(X)`. As we have already noted, one way to think of this is to see the contents of the workspace as the "private beliefs" of the toplevel, and the node-resident programs as "shared knowledge."

The user calling `greetings/0` results in a series of messages of the following form to be sent back to the client, one per second:

```
{ "type": "output",
  "pid" : 56120932,
  "data": "hello" }
```

Here is where the node must *push* output to the client, which is easily done when a bi-directional transport such as WebSocket is employed.

The user hitting Control-C aborts the non-terminating query and results in a response in the form of this message:

```
{ "type": "abort",
  "pid" : 56120932 }
```

Finally, using source code fetched by means of the `load_uri` option, the user decides to ask the toplevel to spawn a count actor of the kind we saw in Chapter 2. As we leave the scenario, it is still running. However, should the user leave the terminal at `http://n7.org/ide` the toplevel process will first kill (by calling `exit/2`) the count actor and then die (since the closing of the websocket will kill it, again by a call to `exit/2`). There is a kind of supervision hierarchy in play here, and we shall say more about that in Chapter 4.

Bi-directionality and statefulness are properties that make WebSocket by far the most flexible transport protocol for the Prolog Web. Over a WebSocket connection, a client can be in almost total control of an actor such as a toplevel. (We write "almost" here since the owner of the node on which the actor is running always has the ultimate say when it comes to how much resources will be allocated to the running of an actor.)

As a reminder, note that despite what Figure 3.8 may suggest, the browser client may not be alone in interacting with this particular node. Other client may be talking to other actors running there. There may be tens of thousands of them. They are completely shielded from each other, unless the programs they are running have been written to allow (some of) them to communicate.

—

We have seen that when accessed from a terminal, the node controller in combination with a toplevel actor is capable of serving as a *shell*. The toplevel provides the core of the shell, while the node controller provides the $Var substition mechanism, the redefinition of I/O predicates, etc.

We are in direct communication with the toplevel, the node controller merely provides the "ether" through which the interaction is taking place.

3.5.5 The stateless HTTP API

As we shall see, the WebSocket API is supported only by the most capable kind of node, with the most powerful profile. Less capable profiles only supports a 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 parameters:

Parameter	Type	Description	Default Value
`goal`	callable	The goal to be called	None
`template`	term	The template to be used	Same as goal
`offset`	int >= 0	The offset in the virtual list of solutions	0
`limit`	int > 0	Max number of solutions to be returned	No limit
`load_text`	string	Source text to be injected	None
`load_uri`	URI	A URI pointing to source text to be injected	None
`format`	string	The format of answer responses	`json`

Table 3.3 HTTP API Parameters

The only required parameter is `goal`.

Such URIs are simple, they are meaningful, they are declarative, they can be bookmarked, and responses are cachable by intermediates. Most of these desirable properties are there because the client-node interaction involved during the process of resolving the URIs has been made stateless.

The semantics of the `offset` and `limit` parameters are borrowed from SQL and SPARQL, and as in these languages they expect integer values, where `offset` defaults to `0` and `limit` to `none`. 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 slice of solutions `Ss` to `Q`, starting at offset `N` and having a length of at most `M`. 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>}`.

Here are a couple of examples of its use, with requests matched up by responses:

```
GET http://n1.org/call?goal=wife(Husband,Wife)
{ "type":"success",
  "data":[{"Husband":"socrates","Wife":"xantippa"},
          {"Husband":"aristotle","Wife":"pythias"}],
  "more":false }
```

If the value of the `format` parameter is `prolog` in the request for the query `?-wife(Husband,Wife),` a binary term with functor `success` is returned, containing a list of instances of the query in the first argument and either `true` or `false` in the second argument, indicating whether additional soutions may exist or not.

```
GET http://n1.org/call?goal=wife(Husband,Wife)&format=prolog
success([wife(socrates,xantippa),
         wife(aristotle,pythias)],false)
```

A failure answer indicates that no solutions exist. Here is a request for solutions for a goal that fails.

```
GET http://n1.org/call?goal=wife(plato,Wife)
{ "type":"failure" }
```

If the value of the `format` parameter is `prolog` and the goal fails, the atom `failure` is returned:

```
GET http://n1.org/call?goal=wife(plato,Wife)&format=prolog
failure
```

Our next example illustrates source code injection using the `load_text` parameter. If the data exceeds URL length limitations imposed by browsers, servers, or intermediaries, POST rather than GET might be the only viable option. A node should therefore offer POST in addition to GET. A POST request such as

```
POST http://n1.org/call
goal=husband(Wife,Husband)
offset=1
limit=2
load_text=husband(Wife,Husband):-wife(Husband,Wife).
```

would produce the following JSON formatted response:

```
{ "type":"success",
  "data":[{"aristotle":"pythias"}],
  "more":false }
```