Prolog Fundamentals

A Complete 8-Chapter Programming Course

Topics covered:

SWI-Prolog & the ?- REPL · Facts, rules & predicates

Unification & the real contrast with pattern matching · Backtracking & search

Lists & recursion · is vs. = vs. == · Cut, treated honestly

Exercises: 24 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed as a direct counterpoint to Haskell's own declarative style

Course 1 of 2 · Intermediate/Advanced follows

Table of Contents

CHAPTER 1 OF 8

Getting Started

COURSE 1 · CH 1

Getting Started

No main, no function to call — a Prolog program is a database of true statements, and running it means asking it questions

Haskell's own course, just completed, spent sixteen chapters showing what "declarative" means when a language evaluates pure expressions to values. This course is the direct counterpoint: Prolog is also called declarative, but it means something genuinely different — describing relations that hold true, then asking the system to search for what satisfies them.

SWI-Prolog & the ?- REPL

```
$ swipl
?- 2 + 2 =:= 4.
true.
```

SWI-Prolog is the standard modern implementation; `swipl` launches its interactive top-level. Similar in spirit to GHCi's own REPL — but every query here ends in a period, and the answers look nothing like a function's return value.

Facts — The Most Basic Unit

```
parent(tom, bob).
parent(tom, liz).
```

A genuinely different starting point from `haskell1-1`'s own function-first orientation. There's no function being defined here at all — `parent(tom, bob).` is a **fact**, a statement simply asserted as true and added permanently to Prolog's own database. A Prolog program begins as a collection of facts about relations, not a sequence of function definitions.

Querying — Asking Questions of the Database

```
?- parent(tom, bob).
true.

?- parent(tom, X).
X = bob.
```

Here's this chapter's central reveal: a query isn't "calling a function" — it's asking, literally, "is this true?" or "for what `X` is this true?" `parent(tom, X)` genuinely asks Prolog to search its own database and find a value for `X` that makes the statement hold. This is a fundamentally different orientation from every other language on this site, where calling something always means "run this code and give me back a value."

Multiple Facts, Multiple Answers

```
?- parent(tom, X).
X = bob ;
X = liz.
```

With two matching facts in the database, `parent(tom, X)` genuinely has two possible answers — typing `;` at the prompt asks Prolog for the next one. A first, gentle preview of Chapter 4's own full backtracking treatment, without naming the underlying machinery yet.

No main, No Entry Point — A Genuinely Different Program Shape

Stated plainly: there's no equivalent to `haskell1-1`'s own `main :: IO ()` at this level at all. A Prolog "program" *is* its database of facts and rules — "running" it means posing queries against that database, not executing a fixed starting point. There's no single place execution begins the way there always was in every language covered before this one.

ASPECT	HASKELL (HASKELL1-1)	PROLOG
Program shape	`function definitions, one entry point`	`a database of facts/rules, no fixed entry point`
"Running" the program	`executing main`	`posing a query`
What a call/query does	`evaluates to a single value`	`searches for zero, one, or many satisfying answers`

> **THINK OF A FACT AS A TRUE STATEMENT, NOT STORED DATA**

`parent(tom, bob).` isn't "storing a value in a variable" the way an assignment would — it's adding a permanently-true statement to Prolog's own knowledge base, which every later query can then search against.

A FORGOTTEN PERIOD IS A VERY COMMON, VERY CONFUSING FIRST MISTAKE

Every fact, rule, and query must end in a period. Forgetting it doesn't produce an immediate, obvious error — `swipl` simply sits waiting for more input, since it assumes the statement isn't finished yet. If the prompt seems stuck, check for a missing period first.

Coding Challenges

CHALLENGE 1

Add three facts to Prolog's database describing which fruits are which colors (e.g. color(banana, yellow).), then query for the color of one specific fruit.

View solution

CHALLENGE 2

Add two facts asserting that two different fruits are both red, then run a single query that finds ALL red fruits, pressing ; to see each answer in turn.

View solution

CHALLENGE 3

Write a short comment explaining why "querying a Prolog database" and "calling a Haskell function" are fundamentally different operations, specifically addressing what happens when a query could have more than one true answer.

View solution

CHAPTER 1 QUICK REFERENCE

- swipl launches SWI-Prolog's interactive ?- top-level
- A fact (parent(tom, bob).) is a statement asserted as permanently true, not a function definition or a variable assignment
- A query asks "is this true?" or "for what X is this true?" — genuinely different from calling a function for its return value
- A query can have zero, one, or many answers — ; requests the next one when more than one exists
- There is no main and no fixed entry point — the program IS the database, and running it means querying it

- Every fact, rule, and query ends in a period — a missing one leaves swipl waiting silently
- **Next chapter:** facts and rules — building relations beyond plain facts

CHAPTER 2 OF 8

Facts & Rules

COURSE 1 · CH 2

Facts & Rules

Multiple clauses, one predicate — genuinely comparable to haskell1-4's own multiple-equations-one-function shape

`prolog1-1` covered plain facts. This chapter adds rules — facts that depend on other goals being true — and gets precise about what actually organizes a Prolog program.

Rules — Facts That Depend on Other Facts

```
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

`:-` reads as "if." This rule says: `grandparent(X, Z)` is true *if* `parent(X, Y)` is true *and* `parent(Y, Z)` is true. The comma is logical AND — a genuinely different reading from every other language's own comma usage on this site.

The Body Is Just More Goals

A rule's body is itself a sequence of goals, each one potentially a plain fact or another rule. Rules can call other rules — genuinely comparable in spirit to function composition — but querying a rule works *exactly* the same way as querying a plain fact. The caller can't tell the difference from outside at all, a real, clean uniformity.

Predicates — The Real Unit of Organization

```
parent(tom, bob).
parent(tom, liz).
-- both facts above, TOGETHER, define one predicate: parent/2
```

A **predicate** is a name plus arity (argument count) — `parent/2`. Every fact and rule sharing that same name and argument count together define one predicate. Genuinely comparable to how `haskell1-4`'s own

multiple function equations (`sumList`'s base case and recursive case) together define one function — Prolog's version has this exact "multiple clauses, one definition" shape.

Recursive Rules — Building ancestor from parent

```
ancestor(X, Y) :- parent(X, Y).                    -- base case
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).     -- recursive case
```

A real, genuinely recursive rule — base case plus recursive case, directly comparable to `haskell1-4`'s own base-case/recursive-case shape for functions. The difference: this recursively defines a *relation*, not a function computing a single value.

Variables Are Scoped to a Single Clause

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
-- the X in line 1 has NOTHING to do with the X in line 2 — completely fresh variables each clause
```

A real, easy-to-miss detail: a variable's name is scoped to a single clause only. Reusing `X` across two clauses of the same predicate does not link them — each clause gets its own completely independent set of variables, a real departure from ordinary lexical scoping in every other language covered so far.

Anonymous Variables — The Underscore

```
likes(mary, _).   -- "mary likes something" — the something is never bound to a name
```

`_` means "some value exists here, but I don't care what it is, and I won't bind it to a name." Directly comparable to `haskell1-4`'s and `haskell1-7`'s own discard pattern — a real, genuine syntactic and conceptual convergence between the two languages.

ASPECT	HASKELL (HASKELL1-4/HASKELL1-8)	PROLOG
Multiple definitions, one unit	`multiple function equations`	`multiple clauses (facts/rules) of one predicate`
Discard/don't-care binding	`_ in a pattern`	`_ as an anonymous variable`

ASPECT	HASKELL (HASKELL1-4/HASKELL1-8)	PROLOG
What defines "the same unit"	`function name`	`name + arity together (parent/2)`

THINK AND DOCUMENT IN TERMS OF NAME/ARITY

`parent/2` and a hypothetical `parent/3` would be two completely separate, unrelated predicates in Prolog, despite sharing a name — get in the habit of naming predicates by their arity too when thinking or writing about them.

REUSING A VARIABLE NAME ACROSS CLAUSES DOES NOT LINK THEM

Writing `X` in two different clauses of the same predicate looks like it might connect them — it doesn't. Each clause's variables are completely independent, a genuine, real source of early confusion coming from languages where a reused name usually means something.

Coding Challenges

CHALLENGE 1

Write parent/2 facts for a small three-generation family, then write a grandparent/2 rule using two parent/2 goals, and query it for a specific grandparent-grandchild pair.

View solution

CHALLENGE 2

Write a recursive ancestor/2 predicate (base case + recursive case) using your own parent/2 facts, and query it for an ancestor relationship that spans three generations, requiring the recursive case to fire.

View solution

CHALLENGE 3

Write a short comment explaining why writing X in two separate clauses of the same predicate does NOT create any connection between them, and what a programmer should do instead if they genuinely need to compare or relate two values across separate parts of a rule.

View solution

CHAPTER 2 QUICK REFERENCE

- :- reads as "if"; a comma in a rule body is logical AND
- A rule's body is just more goals — calling a rule looks identical to calling a plain fact from outside
- A predicate is a name+arity pair (parent/2) — every fact/rule sharing both together defines one predicate, the same "multiple clauses, one unit" shape as haskell1-4's own function equations
- Recursive rules (ancestor/2) use the same base-case/recursive-case shape as haskell1-4's own recursive functions, but define a relation rather than compute a value
- Variables are scoped to a single clause — the same name in two clauses is NOT connected
- _ is the anonymous variable — "something exists, I don't care what," directly comparable to haskell1-4's/haskell1-7's own discard pattern
- **Next chapter:** unification — Prolog's real central mechanism, contrasted directly with Haskell's own pattern matching

CHAPTER 3 OF 8

Unification

COURSE 1 · CH 3

Unification

Prolog's real central mechanism — bidirectional, unlike haskell1-7's own one-directional pattern matching

This is the chapter this whole course's own framing has been pointing toward. Unification looks, at a glance, like Haskell's pattern matching — both languages check whether a value has a certain shape and bind pieces of it to names. They are not the same mechanism, and the difference is real and structural.

What Unification Actually Does

```
?- X = 5.
X = 5.

?- 5 = 5.
true.

?- 5 = 6.
false.
```

Two terms **unify** if they can be made identical by substituting values for variables. `X = 5` doesn't assign — it *attempts to unify* the unbound variable `X` with `5`, which succeeds by binding `X`. `5 = 5` succeeds trivially (already identical); `5 = 6` fails outright — no substitution could ever make them equal.

Unifying Two Variables

```
?- X = Y.
X = Y.   -- neither bound to a concrete value — LINKED together instead
```

When both sides are unbound, unification succeeds by *linking* them — if either is later bound to something concrete, the other becomes bound too. Genuinely different from every other language's own assignment, which always gives a value *to* a variable, never links two unknowns to each other.

Unifying Compound Terms — Structural Matching

```
?- point(X, Y) = point(3, 4).
X = 3, Y = 4.

?- point(X, Y) = point(3, 4, 5).
false.   -- arity mismatch — fails immediately
```

Genuinely comparable to Haskell's own pattern matching on a constructor's shape — same functor name, same arity required, corresponding arguments unified pairwise.

The Central Contrast — Bidirectional vs. One-Directional

```
?- point(3, 4) = point(X, Y).   -- identical behavior to the reverse order
X = 3, Y = 4.
```

Here's the explicit, promised reveal: `haskell1-7`'s own pattern matching always matches a *known, already-computed* value against a pattern — the pattern side can never "reach backwards" and solve for an unknown piece of the value being matched. Prolog's unification has no such asymmetry at all. There is no "the pattern" and "the value" in Prolog — only two terms being made equal, in either order, or with variables on both sides at once. Genuinely bidirectional, where Haskell's matching is genuinely one-directional.

A Concrete Demonstration — Solving What Pattern Matching Never Could

```
?- f(X, Y) = f(g(Z), 5), Z = 3.
X = g(3), Y = 5, Z = 3.
```

`X` is bound to `g(Z)` — a compound term that *itself still contains an unbound variable* — and later, `Z` gets unified with `3`, retroactively completing `X`'s own value. Values can be partially known and progressively refined. Haskell's pattern matching has no equivalent to this "partially solved, filled in later" behavior at all — matching there is all-or-nothing against an already-complete value.

The Unification Algorithm, Briefly

A real, honest technical note: the full unification algorithm includes an "occurs check" to detect a variable unifying with a term that contains itself — but SWI-Prolog, by default, does **not** perform this check, for performance reasons. Worth naming plainly rather than glossing over, matching this site's established pattern of honest technical caveats.

ASPECT	HASKELL PATTERN MATCHING (HASKELL1-7)	PROLOG UNIFICATION
Direction	one-directional — value vs. pattern	bidirectional — no fixed roles
Can solve for unknowns on both sides	no	yes
Partially-bound compound results	not possible	routine (X bound to g(Z), Z unbound)

READ = AS "ATTEMPT TO UNIFY," NEVER AS "ASSIGN"

Treating `=` as an imperative assignment is the single most common early Prolog mistake — `prolog1-7`'s own `is` / `=` / `==` chapter builds directly on getting this distinction right now.

SWI-PROLOG'S DEFAULT LACK OF AN OCCURS CHECK IS A REAL, DOCUMENTED GOTCHA

`X = f(X)` should, in principle, fail — a variable can never legitimately unify with a term containing itself. Without the occurs check, SWI-Prolog instead silently creates a cyclic term by default. Genuinely surprising behavior, not a bug, worth knowing about before it causes real confusion.

Coding Challenges

CHALLENGE 1

In swipl, unify point(X, Y) with point(7, 9) and then, separately, unify point(7, 9) with point(X, Y) (reversed order), showing both produce identical bindings.

View solution

CHALLENGE 2

Write a single query that unifies pair(X, Y) with pair(f(Z), 10) and then unifies Z with 42 in the same query (using a comma), printing the final bindings for X, Y, and Z.

View solution

CHALLENGE 3

Write a short comment giving a concrete example of a task unification can do that Haskell's pattern matching genuinely cannot, explaining specifically what "bidirectional" buys in that example.

📄 View solution

CHAPTER 3 QUICK REFERENCE

- = attempts unification, not assignment — two terms unify if a substitution can make them identical
- Two unbound variables unify by linking together, not by one taking a value from the other
- Compound terms unify structurally — same functor, same arity, arguments unified pairwise
- THE central contrast: unification is bidirectional (no fixed pattern/value roles), Haskell's own pattern matching (haskell1-7) is strictly one-directional
- A variable can be bound to a compound term that itself still contains unbound variables, refined later — no Haskell pattern-matching equivalent
- SWI-Prolog skips the occurs check by default — X = f(X) creates a cyclic term rather than failing
- **Next chapter:** backtracking and the search tree — how multiple answers actually get explored

CHAPTER 4 OF 8

Backtracking & the Search Tree

COURSE 1 · CH 4

Backtracking & the Search Tree

Nondeterminism baked into the entire execution model — not one opt-in type, the way haskell2-3's own [] monad is

`prolog1-1` showed `;` producing a second answer without explaining how. Here's the real machinery underneath — and a genuinely rich comparison to a mechanism Haskell only offers as one specific, opt-in monad.

The Search Tree — How Prolog Actually Finds Answers

When a goal has multiple clauses that could match, Prolog tries the **first** one. If that path later leads to failure, Prolog automatically **backtracks** to try the next alternative. A useful mental model: a tree of choices, explored via depth-first search, backing up and trying a different branch whenever the current one dead-ends.

Conjunction and the Order Goals Are Tried

```
?- parent(X, Y), age(Y, 25).
```

Goals are tried left to right — and if a later goal fails, Prolog backtracks into an *earlier* goal to try its next alternative there, not just moving on independently. Genuinely sequential dependency, matching `haskell2-3`'s own `>>=` chaining in spirit (each step can depend on what came before) — but Prolog explores every alternative automatically, with no explicit lambda written per step.

; for Explicit Alternatives (Disjunction)

```
likes(mary, wine) ; likes(mary, beer).
```

`;` is logical OR, usable directly inside a rule body — not just something the top-level REPL shows between answers. Worth naming explicitly: the top-level's own `;` prompt and rule-body `;` are genuinely the *same* underlying mechanism, just triggered in two different places.

Comparison to Haskell's Own [] Monad

A genuinely rich comparison: `haskell2-3`'s own list monad represents nondeterminism — `>>=` on `[]` explores every combination of possible results, conceptually similar in spirit to Prolog's backtracking search. But Prolog's version is baked into the *entire language's* execution model by default — every goal call is implicitly a potential source of backtracking, not something wrapped in one specific type a programmer opts into. The same distinction `haskell1-5` drew between Java's opt-in Stream laziness and Haskell's own language-wide laziness, appearing again here in a new context: Haskell's `[]` monad is one specific tool; Prolog's backtracking is the language's own default way of running everything.

A Practical Example — Generating Combinations

```
?- member(X, [1,2,3]), member(Y, [a,b]).
X = 1, Y = a ;
X = 1, Y = b ;
X = 2, Y = a ;
-- ...and so on — every combination, generated automatically
```

Real, useful combinatorial work — no explicit nested loop was written anywhere. Backtracking generates every pairing automatically, on demand.

Backtracking Can Be Expensive — A Real, Honest Performance Note

Searching a large tree with many choice points can be genuinely slow if the search space isn't constrained somehow. Chapter 8's own `!` (cut) is one real tool for controlling this — foreshadowed here without going deep into it yet.

ASPECT	HASKELL [] MONAD (HASKELL2-3)	PROLOG BACKTRACKING
Scope	one specific type, opted into	the language's own default execution model
Exploration	explicit >>= chains	automatic, on every goal call
Getting the "next" result	the list itself already holds all results	; requests the next one on demand

PREFER SEPARATE CLAUSES OVER INLINE ; WHEN READABILITY ALLOWS

Multiple rules for the same predicate often read more clearly than one rule with an inline `;` disjunction — reach for `;` specifically when the alternatives are genuinely local to one small piece of logic.

"SLOW" AND "WILL NEVER TERMINATE" CAN LOOK IDENTICAL WHILE WAITING

A deeply recursive predicate with many choice points can search a genuinely huge space before failing completely — from the outside, a program that's merely slow and one that will never finish look exactly the same while you're waiting on it. A real, practical debugging concern worth being aware of early.

Coding Challenges

CHALLENGE 1

Write three color/1 facts and three size/1 facts, then run a single query combining color(X), size(Y) and use ; to retrieve all combinations.

View solution

CHALLENGE 2

Write a rule using an inline ; disjunction (e.g. drink(mary, X) :- X = wine ; X = beer.) and query it, retrieving both answers.

View solution

CHALLENGE 3

Write a short comment explaining the real difference in SCOPE between Prolog's own backtracking and Haskell's [] monad, using a concrete example of ordinary Prolog code (not wrapped in anything special) that is automatically backtracking-capable.

View solution

CHAPTER 4 QUICK REFERENCE

- Prolog explores a search tree depth-first, automatically backtracking to the next alternative on failure
- Goals in a conjunction are tried left to right, with backtracking able to reach back into an earlier goal — sequential dependency in spirit like haskell2-3's own >>=, but automatic
- ; is logical OR, usable directly in a rule body — the same mechanism as the top-level's own "next answer" prompt
- Prolog's backtracking is the language's own default execution model — a real scope difference from Haskell's own opt-in [] monad, echoing haskell1-5's Java-Streams-vs-language-wide-laziness distinction
- A large, unconstrained search space can be genuinely slow — "slow" and "never terminates" look identical while waiting

- **Next chapter:** lists — Prolog's own [H|T] syntax, a genuine visual convergence with Haskell's own (x:xs)

CHAPTER 5 OF 8

Lists

COURSE 1 · CH 5

Lists

[H|T] — a genuine independent convergence with haskell1-4's own (x:xs)

Two languages, built from entirely different premises, looking at lists and landing on almost the same notation. This chapter is that convergence, made explicit — plus a real Prolog-specific twist unification adds on top.

List Syntax — [], [H|T]

```
[]              -- the empty list
[H|T]           -- a list with a Head element and a Tail (the rest)
[1,2,3]         -- really: [1|[2|[3|[]]]]
```

An empty list, or a head consed onto a tail — `[1,2,3]` is genuinely sugar for the fully nested cons form.

The Genuine Convergence with Haskell's (x:xs)

Here's this chapter's central comparison: `haskell1-4`'s own `(x:xs)` splits a list into head `x` and tail `xs` using `:`. Prolog's `[H|T]` does exactly the same conceptual split using `|`. Both languages, working from the identical foundational insight — a list is either empty, or a head attached to a tail — arrived at visually similar notation completely independently, not through shared ancestry. Worth naming as a genuine convergence rather than a coincidence.

Multiple Elements Before the Tail

```
[H1, H2|T] = [1,2,3,4].   -- H1=1, H2=2, T=[3,4]
```

A real, practical convenience — splitting off more than one leading element in a single pattern.

Building Lists via Unification, Not Construction

```
?- [H|T] = [1,2,3].
H = 1, T = [2,3].

?- H = 1, T = [2,3], L = [H|T].
L = [1,2,3].   -- same [H|T] syntax, now BUILDING instead of taking apart
```

A real Prolog-specific angle, tying directly back to `prolog1-3`'s own central theme: since `=` is unification, not assignment, `[H|T]` can be used to *take apart* a known list, or to *build* a new one from known pieces — the identical syntax, in either direction, because unification never cared which side supplied the concrete values.

Common List Predicates — length/2, append/3, member/2

```
?- append([1,2], [3,4], X).
X = [1,2,3,4].          -- ordinary "computation" direction

?- append(X, Y, [1,2,3]).
X = [], Y = [1,2,3] ;
X = [1], Y = [2,3] ;
X = [1,2], Y = [3] ;
X = [1,2,3], Y = []. -- backtracking finds every way to split [1,2,3] in two!
```

A genuinely great illustration of bidirectionality applied to lists: the exact same `append/3` definition computes a concatenation forward, or — run "backwards" — searches for every possible way to split a list into two pieces, via backtracking. One predicate definition, two genuinely different uses.

ASPECT	HASKELL (HASKELL1-4)	PROLOG
Head/tail split syntax	`(x:xs)`	`[H|T]`
Origin	the language's own foundational design	the language's own foundational design — independently convergent
Concatenation (++/append)	one-directional only	bidirectional — can also split, via backtracking

[H|T] IS THE SAME INSTINCT AS (X:XS), IF THAT BACKGROUND IS FAMILIAR

The pattern-matching-on-recursive-structure instinct transfers directly — recognizing `[H|T]` as "Prolog's own version of what I already know" is a genuinely accurate way to think about it, not a loose analogy.

[H|T] REQUIRES AT LEAST ONE ELEMENT TO UNIFY SUCCESSFULLY

Unifying `[H|T]` against `[]` fails outright — a real, concrete parallel to needing a separate base case for the empty list, the same base-case/recursive-case split `haskell1-4` 's own recursive functions require.

Coding Challenges

CHALLENGE 1

In swipl, unify [H1, H2|T] against a five-element list of your choosing, printing H1, H2, and T separately.

View solution

CHALLENGE 2

Use append/3 in its "forward" direction to concatenate two lists, and separately use it "backwards" on a four-element list to find all the ways it can be split into two pieces, showing all the resulting splits.

View solution

CHALLENGE 3

Write a short comment explaining why [H|T] failing to unify against [] is a genuine parallel to needing a separate base case in a recursive Haskell function, referencing haskell1-4 directly.

View solution

CHAPTER 5 QUICK REFERENCE

- [] is the empty list; [H|T] splits a list into head and tail — [1,2,3] is really [1|[2|[3|[]]]]
- [H|T] and haskell1-4's own (x:xs) are a genuine independent convergence, not shared ancestry
- Because = is unification (prolog1-3), [H|T] can build a list from known pieces just as easily as it takes one apart
- append/3 genuinely runs bidirectionally — forward to concatenate, backward via backtracking to find every possible split
- [H|T] fails against [] — the same real base-case/recursive-case distinction haskell1-4's own recursive functions require
- **Next chapter:** recursion over lists — no traditional loops here either, revisiting haskell1-4's own surprise for a different reason

CHAPTER 6 OF 8

Recursion Over Lists

COURSE 1 · CH 6

Recursion Over Lists

haskell1-4's own "wait, what?" moment — happening again, for a completely different reason

Prolog also has no `for`, no `while`. Same surface fact as `haskell1-4`'s own Haskell. The reason underneath is genuinely different, and worth being precise about.

No Traditional Loops in Prolog Either

Here's the real distinction: Haskell has no loops because it has no mutation for a loop counter to increment — `haskell1-3` made that structural argument directly. Prolog is different: it genuinely *can* have mutable-feeling state, via the dynamic database (`assert`/`retract`, covered in full in Course 2). The absence of loops here isn't because mutation is impossible — it's because Prolog's entire computational model *is* the relational/search model. Recursion isn't a workaround for a missing loop construct; it's the natural, direct way to define a relation across a list structure, which a loop simply isn't shaped to express at all.

Length — A Genuinely Recursive Relation

```
myLength([], 0).
myLength([_|T], N) :- myLength(T, N1), N is N1 + 1.
```

Base case plus recursive case, structurally parallel to `haskell1-4`'s own shape — but this *defines a relation* between a list and its length, rather than "computing and returning" a length the way a Haskell function would.

Sum — Threading an Accumulator

```
mySum([], Acc, Acc).
mySum([H|T], Acc, Sum) :- Acc1 is Acc + H, mySum(T, Acc1, Sum).
```

Genuinely comparable to a tail-recursive accumulator pattern in any language — but here it's built directly into how the relation threads a running value through recursive calls via an extra argument, not a special "accumulator" language feature.

Recursion Explores the Search Tree Too

A real, honest connecting thread back to `prolog1-4`: a recursive predicate call is itself just another goal — meaning every recursive call is potentially a fresh source of backtracking, not merely "a loop iteration." A genuinely different mental model from imperative-style recursion in every other language, where a call is understood as deterministic unless explicitly made otherwise.

A Concrete Example — Filtering

```
evens([], []).
evens([H|T], [H|Rest]) :- H mod 2 =:= 0, evens(T, Rest).
evens([H|T], Rest)     :- H mod 2 =\= 0, evens(T, Rest).
```

The exact same conceptual operation as `haskell1-4`'s own `filter` — keep only elements satisfying a condition — expressed as a genuinely different *kind* of construct: three clauses defining a relation, rather than a single higher-order function call.

ASPECT	HASKELL (HASKELL1-4)	PROLOG
No traditional loops — surface fact	`true`	`true`
WHY loops don't exist	`no mutation exists at all — nothing to increment`	`the relational/search model, not a mutation restriction — assert/retract genuinely exists`
What recursion produces	`a computed value`	`a defined relation, searched for a satisfying case`

THINK IN TERMS OF DEFINING A RELATION, NOT PROCESSING STEP BY STEP

A recursive Prolog predicate defines what holds true across an entire list structure — a different framing from "processing a list step by step," even compared to a recursive Haskell function that at least computes a single value at the end.

A MISSING BASE CASE FAILS THE SAME WAY, WITH A PROLOG-SPECIFIC WRINKLE

The same infinite-recursion bug class already covered for Haskell (`haskell1-4`) and Python (the site's own `python_lesson_infinite_loops.html`) applies here too — a missing or unreachable base case recurses without end. In Prolog, this can also interact with backtracking: the failure mode can look like exhausting the stack while trying

alternative choice points, not just recursing once down a single path — a real, Prolog-specific twist on a bug class already met twice on this site.

Coding Challenges

CHALLENGE 1

Write myLength/2 exactly as shown in the chapter, and query it against a list with at least five elements to confirm the correct length.

View solution

CHALLENGE 2

Write a mySum/2 predicate (calling the three-argument accumulator version from the chapter with an initial accumulator of 0) and query it against a list of numbers to confirm the correct total.

View solution

CHALLENGE 3

Write a short comment explaining why Prolog's own lack of traditional loops is NOT for the same underlying reason as Haskell's, given that Prolog genuinely does have a way to hold mutable-feeling state.

View solution

CHAPTER 6 QUICK REFERENCE

- Prolog has no traditional loops either — but genuinely NOT for haskell1-4's own immutability-driven reason, since Prolog's dynamic database allows real mutable-feeling state
- Recursion is the natural way to define a relation across a list, not a workaround for a missing loop construct
- myLength/2 and mySum/2-3 define relations rather than computing and returning values the way Haskell functions do
- Every recursive call is itself just another goal — a fresh source of potential backtracking, not merely "an iteration"
- A missing base case fails the same way as Haskell's own version, with a real Prolog-specific twist involving backtracking exhaustion
- **Next chapter:** arithmetic — the classic is/=/== beginner trap

CHAPTER 7 OF 8

Arithmetic: is vs. = vs. ==

COURSE 1 · CH 7

Arithmetic: is vs. = vs. ==

The classic beginner trap — foreshadowed since prolog1-3's own tip-box, fully unpacked here

`prolog1-3`'s tip-box promised this chapter directly. Three operators, all doing something with equality-shaped names, all genuinely different — and mixing them up is the single most common early Prolog mistake.

= — Unification, Not Arithmetic

```
?- X = 2 + 3.
X = 2+3.   -- NOT 5! X is unified with the unevaluated TERM 2+3
```

`X = 2 + 3` does not compute anything. It unifies `X` with the actual compound term `2+3` — functor `+`, two arguments — never evaluated as arithmetic at all. Genuinely surprising for anyone assuming `=` means "compute and assign."

is — Forcing Arithmetic Evaluation

```
?- X is 2 + 3.
X = 5.

?- X is Y + 1.
-- ERROR: Y is not bound to a number — arithmetic cannot proceed
```

`is` evaluates its right side as a genuine arithmetic expression, then unifies the left side with the resulting number. A real asymmetry unlike unification's own flexibility: `is`'s right side must already resolve to a number — an unbound variable there is a real error, not something `is` can work around the way unification handles unbound variables freely.

== — Structural Equality, No Unification Side Effects

```
?- X = Y.
X = Y.        -- succeeds — LINKS X and Y together (prolog1-3)

?- X == Y.
false.        -- fails — two DIFFERENT unbound variables are not IDENTICAL
```

`==` checks whether two terms are *already* identical, with zero unification side effects — nothing gets bound. Two different unbound variables aren't identical, even though `=` on the same two would succeed by linking them.

The Three Side by Side

```
?- X = 2+3.        X = 2+3.    -- unifies with the unevaluated term
?- X is 2+3.        X = 5.     -- evaluates, then unifies with the number
?- 2+3 == 2+3.     true.      -- structurally identical terms, no evaluation needed
?- 2+3 == 5.       false.     -- the compound term 2+3 is NOT the number 5, structurally
```

Comparison Operators for Numbers

```
?- 2+2 =:= 4.   true.    -- forces arithmetic evaluation on BOTH sides, then compares
?- 2+2 == 4.    false.   -- 2+2 is structurally a compound term, not the atom 4 — no evaluation
```

`=:=`, `=\=`, `<`, `>`, `=<`, `>=` all force arithmetic evaluation on both sides — the same spirit as `is`, genuinely different from `==`'s own pure structural comparison. A second, concrete demonstration of the exact same underlying distinction.

OPERATOR	WHAT IT DOES	FORCES EVALUATION?	CAN BIND UNBOUND VARIABLES?
`=`	attempts unification	no	yes
`is`	evaluates right side, unifies left with the number	yes (right side)	yes (left side only)
`==`	checks structural identity	no	no
`=:=`	evaluates both sides, compares as numbers	yes (both sides)	no

ASK WHAT YOU'RE ACTUALLY TRYING TO DO

Checking if two things are already the same? Use `==`. Computing a number? Use `is`. Binding a variable, or matching a structure? Use `=`. Comparing two arithmetic expressions numerically? Use `=:=`. Each real intent maps to exactly one operator.

2+2 == 4 FAILING IS A GENUINELY COMMON POINT OF CONFUSION

Coming from any language where `+` is always eagerly evaluated, seeing `2+2 == 4` report `false` is a real surprise. In Prolog, `2+2` is just an unevaluated compound term until something — `is` or `=:=` — explicitly forces it to become a number.

Coding Challenges

CHALLENGE 1

In swipl, run X = 3 * 4 and separately X is 3 * 4, printing what X actually contains in each case, and explain the difference in a comment.

View solution

CHALLENGE 2

Write a query that unifies X = Y, then a separate query that checks X == Y for two fresh, unbound variables, showing the two results differ and explaining why in a comment.

View solution

CHALLENGE 3

Write a query attempting X is Y + 1 where Y is a genuinely unbound variable, show the resulting error, and explain why is has this restriction while = does not.

View solution

CHAPTER 7 QUICK REFERENCE

- = unifies with the unevaluated term — X = 2+3 gives X = 2+3, not 5
- is evaluates its right side as arithmetic, then unifies the left side with the resulting number — the right side must already resolve to a number
- == checks structural identity with zero unification side effects — two different unbound variables are not identical, even though = would link them
- =:=/=\=/</>/=</>= force arithmetic evaluation on both sides, unlike =='s pure structural comparison
- 2+2 == 4 is false — 2+2 stays an unevaluated compound term unless is or =:= forces it into a number

- **Next chapter:** cut (!) — controlling backtracking, treated honestly

CHAPTER 8 OF 8

Cut (!) — Controlling Backtracking, Honestly

COURSE 1 · CH 8

Cut (!) — Controlling Backtracking, Honestly

A genuinely controversial feature, treated the way this course treats every real tradeoff — plainly

Fundamentals closes with Prolog's most debated single character. Cut is real, useful, and genuinely controversial within the Prolog community itself — this chapter doesn't pick a side, it explains exactly what's actually being traded.

What Cut Actually Does

`!`, when reached during execution, **commits** to every choice made so far in the current clause — permanently pruning any remaining backtracking alternatives, both for goals earlier in the same clause and for the choice of which other clause to try for the predicate itself.

A Concrete Example — Cutting Off Alternatives

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).

?- max(3, 5, Result).
Result = 5.

?- max(3, 5, Result), Result == 3.  -- without the cut, backtracking could try clause 1 anyway and produ
```

Without the cut, asking for further solutions via `;` after the correct answer could backtrack into the first clause despite `3 >= 5` already having failed for those bindings in one call, or produce unintended additional results in more complex predicates. The cut commits: once `X >= Y` succeeds, this is the answer — no exploring further.

Green Cuts vs. Red Cuts — An Honest Distinction

A real, well-known distinction worth naming explicitly. A **green cut** only improves efficiency — it prunes alternatives that would have failed anyway, changing nothing about the program's actual logical meaning. A **red cut** genuinely changes the set of solutions a predicate produces compared to removing it. Red cuts are the controversial ones — a predicate relying on one can no longer be read as pure logic alone; understanding which kind you've written requires understanding your own program's behavior with and without the cut present.

The Real Cost to Declarative Purity

Stated plainly, matching this whole course's own established pattern (the same honesty `haskell2-6` applied to monad transformer stacks): a predicate using cut can no longer be freely reordered, and its logical reading now depends on Prolog's own specific left-to-right, depth-first execution order. A real, genuine departure from pure declarative logic, where a set of facts and rules should mean the same thing regardless of execution strategy. Cut ties correctness to execution order — a real, controversial tradeoff, not a free efficiency win.

Cut in Rule Bodies — A Common, Practical Pattern

```
classify(X, negative) :- X < 0, !.
classify(X, zero)     :- X =:= 0, !.
classify(X, positive).
```

The "if-then-else"-shaped idiom — genuinely practical, very common real-world Prolog code, not just a toy example. Each cut commits to that branch once its own condition succeeds, preventing later clauses from ever being tried for that same call.

Negation as Failure, Previewed

```
\+ Goal :- Goal, !, fail.
\+ Goal.
```

A brief, honest forward-pointer: Course 2's own `\+` (negation as failure) is commonly implemented internally using exactly this cut-based pattern — worth naming now, without going deep, since Course 2 covers `\+` properly.

APPROACH	LOGICAL MEANING	EFFICIENCY
No cut	full, pure logical reading	may explore unnecessary alternatives
Green cut	unchanged — same solutions either way	improved — genuinely free win

APPROACH	LOGICAL MEANING	EFFICIENCY
Red cut	`genuinely changed — a real tradeoff`	`improved, but at a real cost to purity`

ASK: WOULD REMOVING THIS CUT CHANGE WHICH SOLUTIONS COME BACK, NOT JUST HOW MANY ALTERNATIVES GET TRIED?

That question is the real test for green vs. red — if the answer is "the solutions themselves would differ," you've written a red cut, and it's worth knowing that consciously rather than by accident.

A CUT'S SCOPE IS THE ENCLOSING CLAUSE ONLY

`!` does not reach back and prune choice points created by goals in a *caller's* own body, before the current clause was even entered — its reach stops at the clause it's written in. A real, genuine source of confusion about how "far" a given cut actually extends.

Coding Challenges

CHALLENGE 1

Write the max/3 predicate exactly as shown in the chapter, query it for max(3, 5, Result), and press ; to confirm the cut prevents any second, incorrect answer from appearing.

View solution

CHALLENGE 2

Write the classify/2 predicate from the chapter, query it for a negative, a zero, and a positive number, and explain in a comment whether each cut used is a green cut or a red cut.

View solution

CHALLENGE 3

Write a short comment explaining, with a concrete example, why a predicate relying on a red cut can no longer be reordered freely the way pure Prolog facts/rules normally could be.

View solution

CHAPTER 8 QUICK REFERENCE — COURSE 1 COMPLETE

- ! commits to every choice made so far in the current clause, pruning remaining alternatives for both earlier goals and other clauses
- A green cut only improves efficiency — same solutions either way; a red cut genuinely changes which solutions come back
- Cut ties a predicate's correctness to Prolog's own specific execution order — a real, honest departure from pure declarative logic
- The if-then-else cut pattern (classify/2) is genuinely common, practical real-world Prolog
- Course 2's own \+ (negation as failure) is commonly implemented internally using this exact cut-based pattern
- A cut's scope stops at its own enclosing clause — it never reaches back into a caller's own choice points
- **Prolog Fundamentals is now complete.** Course 2 (Intermediate/Advanced) begins with findall/bagof/setof — collecting backtracking's multiple solutions into a single concrete list.