Prolog Intermediate/Advanced

A Complete 8-Chapter Programming Course

Topics covered:

findall/bagof/setof · The dynamic database · Negation as failure

Difference lists · DCGs · A meta-interpreter · CLP(FD)

Capstone: an N-Queens solver, closing with Prolog vs. Haskell

Exercises: 24 hands-on exercises with worked solutions

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

Course 2 of 2 · Completes the full Prolog track

Table of Contents

CHAPTER 1 OF 8

findall, bagof, setof — Collecting All Solutions

COURSE 2 · CH 1

findall, bagof, setof — Collecting All Solutions

Turning backtracking's one-at-a-time answers into a single concrete list

Course 1 ended with `prolog1-4`'s backtracking and `prolog1-8`'s cut — both about controlling a search that hands back **one solution at a time**, via `;`. Course 2 opens with the natural next question: what if you want *all* of them, at once, as an ordinary list you can pass around? That's exactly what `findall/3`, `bagof/3`, and `setof/3` do — three meta-predicates, genuinely different from each other, that collect a backtracking search's results into one concrete value.

The Problem: Backtracking Gives You Answers One At A Time

Recall the family facts from Course 1:

```
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(liz, jim).

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

Each `;` press asks Prolog to backtrack and try again. That's fine at the interactive top level, but it's useless inside a program — a predicate that wants "every child of tom, as a list" has no way to press `;` for you. Something has to drive the backtracking to exhaustion and gather the results.

findall/3 — The Simplest Collector

```
?- findall(X, parent(tom, X), Children).
Children = [bob, liz].
```

`findall(Template, Goal, List)` runs `Goal`, collecting `Template` for *every* solution found by backtracking, and unifies `List` with the result — all in one call, no interactive `;` needed. This is genuinely close to the mental model Haskell's own list monad gives you (`haskell2-3`): both collect every result of a nondeterministic computation into one concrete list. The difference is Haskell builds that list through `do`-notation over `[]` as a monad instance; Prolog's `findall` is a dedicated meta-predicate that drives Prolog's own SLD search directly.

`findall` has one important, very forgiving property: if `Goal` has **no** solutions at all, `findall` still succeeds — with an empty list, not a failure.

```
?- findall(X, parent(nobody, X), L).
L = [].
```

bagof/3 — Respecting Free Variables

`bagof/3` looks like a drop-in replacement for `findall` at first glance, but it behaves genuinely differently whenever the goal contains a variable — like `Parent` below — that isn't in the template:

```
?- bagof(Child, parent(Parent, Child), Children).
Parent = tom, Children = [bob, liz] ;
Parent = bob, Children = [ann, pat] ;
Parent = liz, Children = [jim].
```

Instead of one combined list, `bagof` **groups** the results by every possible binding of the free variable `Parent`, backtracking through each group separately. This is a genuine, easy-to-miss surprise coming from `findall` — the same query, one predicate swapped for another, produces three separate answers instead of one merged list.

To get `findall`-style behaviour — one combined list, ignoring who the parent was — use the `^` operator to tell `bagof` to treat `Parent` as existentially quantified ("for *some* Parent," not "grouped by Parent"):

```
?- bagof(Child, Parent^parent(Parent, Child), AllChildren).
AllChildren = [bob, liz, ann, pat, jim].
```

> **BAGOF FAILS ON ZERO SOLUTIONS — FINDALL DOESN'T**
>
> Unlike `findall`, `bagof` genuinely **fails** (not an empty list) when `Goal` has no solutions at all: `?- bagof(X, parent(nobody, X), L).` reports `false`, not `L = []`. A predicate that unconditionally expects a list back can be broken by this — code written against `bagof` has to account for the goal possibly having zero solutions, or fall back to `findall`, which always succeeds.

setof/3 — Sorted, Deduplicated Results

`setof/3` shares `bagof`'s free-variable grouping behaviour exactly, with two extra guarantees: the resulting list is **sorted** (Prolog's standard order of terms) and has all **duplicates removed**.

```
?- setof(Child, Parent^parent(Parent, Child), Sorted).
Sorted = [ann, bob, jim, liz, pat].
```

Compare this to what `findall` would have produced for the same query — `[bob, liz, ann, pat, jim]`, in whatever order the facts were tried and with any duplicates left in. `setof` is the right choice specifically when the caller needs a canonical, duplicate-free answer rather than a raw trace of the search.

PREDICATE	ZERO SOLUTIONS	FREE VARIABLES	ORDER
`findall/3`	succeeds, List = []	ignored — one combined list	search order, duplicates kept
`bagof/3`	fails	grouped, unless ^-quantified	search order, duplicates kept
`setof/3`	fails	grouped, unless ^-quantified	sorted, duplicates removed

DEFAULT TO FINDALL UNLESS YOU SPECIFICALLY NEED GROUPING OR SORTING

In real code, `findall` is usually the safer default precisely because it can't fail out from under you on an empty result — reach for `bagof` / `setof` deliberately, when the free-variable grouping or the sorted/deduplicated guarantee is actually the behaviour you want, not by habit.

Coding Challenges

CHALLENGE 1

Using the parent/2 facts from this chapter, write a findall/3 query that collects every Parent-Child pair in the whole database as a list of Parent-Child terms (e.g. tom-bob), and show the result.

View solution

CHALLENGE 2

Write a bagof/3 query (without ^) that groups each parent's children separately, then rewrite it with Parent^ to combine every child into one list, and explain in a comment why the two queries produce different-shaped results.

View solution

CHALLENGE 3

Add a duplicate fact (e.g. another parent with an already-listed child) to the database, then write both a findall/3 and a setof/3 query for all children, comparing the two results in a comment to show setof's sorting and deduplication in action.

View solution

CHAPTER 1 QUICK REFERENCE

- **findall(Template, Goal, List)** — collects every Template for every solution of Goal; always succeeds, [] on zero solutions
- **bagof(Template, Goal, List)** — like findall, but groups results by any free variable in Goal not in Template; fails on zero solutions
- **setof(Template, Goal, List)** — like bagof, plus sorted order and duplicates removed
- `Var^Goal` tells bagof/setof to treat Var as existentially quantified, combining groups into one list — the findall-shaped behaviour
- findall is the safer default for general use; reach for bagof/setof deliberately for grouping or sorted/deduplicated output
- Genuinely comparable in spirit to Haskell's own list monad (haskell2-3) — both collect a nondeterministic computation's results into one concrete list, by very different mechanisms

CHAPTER 2 OF 8

The Dynamic Database — assert & retract

COURSE 2 · CH 2

The Dynamic Database — assert & retract

Mutating Prolog's own program at runtime, and what it genuinely costs

Every predicate this course has used so far — from Course 1's `parent/2` facts through `prolog2-1`'s `findall` queries — has been **fixed**: written once in the source file, unchanging for the life of the program. This chapter introduces a genuinely different capability — changing the fact database itself, *while the program runs* — and is honest about what that costs.

assertz/1 and asserta/1 — Adding Facts at Runtime

```
:- dynamic(score/2).

?- assertz(score(alice, 10)).
true.

?- assertz(score(bob, 20)).
true.

?- score(X, Y).
X = alice, Y = 10 ;
X = bob, Y = 20.
```

`assertz/1` adds a new fact (or rule) to the end of the database for that predicate — as if it had been written in the source file all along, but happening live, mid-query. `asserta/1` does the same thing at the **beginning** instead, so it's tried first on the next query. Neither existed anywhere in this course's vocabulary before now — every predicate up to this point was defined once, statically, and stayed that way.

retract/1 — Removing Facts at Runtime

```
?- retract(score(alice, 10)).
true.

?- score(X, Y).
X = bob, Y = 20.
```

`retract/1` removes the first fact matching its argument from the database. Combined, `assertz` and `retract` give a predicate the ability to **update** a fact — retract the old value, assert the new one — something no purely declarative set of facts and rules can do to itself.

A Practical Use — a Runtime Counter

The clearest demonstration of what this actually buys: a predicate that remembers how many times it's been called, something genuinely impossible using only Course 1's tools.

```
:- dynamic(counter/1).
counter(0).

increment :-
    retract(counter(N)),
    N1 is N + 1,
    assertz(counter(N1)).

?- increment, increment, increment.
true.

?- counter(X).
X = 3.
```

`counter(0)` is retracted and a new `counter(N1)` asserted on every call to `increment` — real, persistent, mutable state living inside Prolog's own database, surviving across separate top-level queries the way a variable in an imperative language would.

> **DECLARE IT DYNAMIC FIRST**
>
> A predicate that has never appeared in the loaded source file will raise a "procedure does not exist" permission error the first time `assertz` tries to add a fact for it. The `:- dynamic(counter/1).` directive at the top tells Prolog in advance that this predicate is allowed to be modified at runtime, even before any clause for it exists yet.

The Real Tension With Declarative Purity

`prolog1-8`'s cut already cost this course one honest admission — that a predicate's *search* could depend on execution order. `assert`/`retract` goes further: it means the **facts themselves** can differ depending on *when* a query is run relative to other goals, not just how the search through them is pruned. A query against `score/2` run before an `assertz` call can produce a genuinely different answer than the identical query run after it — the same textual query, two different truths, depending purely on timing. That is about as far from "a fixed set of relations, true regardless of when you ask" as Prolog gets.

This deserves the same honesty this site has given every other real language tradeoff — most recently `haskell2-4`'s own acknowledgment that `unsafePerformIO` genuinely can break Haskell's IO-tracking guarantee, not just theoretically. `assert`/`retract` is Prolog's own version of that same kind of escape hatch: a real, practical, frequently-used tool that knowingly steps outside the paradigm's own core promise.

FACTS	MEANING OVER TIME	WHAT COURSE 1 OFFERED
Static (source file)	`fixed — same truth regardless of when queried`	`everything through prolog1-8`
Dynamic (assert/retract)	`mutable — truth depends on timing relative to other goals`	`nothing — genuinely new in Course 2`

ASSERT/RETRACT INSIDE A GOAL THAT LATER BACKTRACKS

Because `assertz` and `retract` are ordinary goals that succeed once and commit immediately — they are **not** undone if Prolog later backtracks past them the way a variable binding would be. A goal that asserts a fact and then fails, forcing backtracking, leaves that fact in the database permanently. This is a real, quiet source of bugs: pure logic guarantees that redoing a goal changes nothing observable; a database mutation breaks that guarantee outright.

Coding Challenges

CHALLENGE 1

Declare a dynamic predicate seen/1, use assertz/1 to add three different values to it one at a time, then query seen(X) and show all three results returned via backtracking.

View solution

CHALLENGE 2

Write the counter/1 and increment predicates exactly as shown in the chapter, call increment five times, query counter(X), and explain in a comment why this behavior would have been impossible using only Course 1's tools.

View solution

CHALLENGE 3

Write a short comment explaining, with a concrete example query, why a fact added via assertz is NOT automatically removed if the goal that asserted it is later backtracked into and fails — contrasting this with how an ordinary variable binding behaves on backtracking.

View solution

CHAPTER 2 QUICK REFERENCE

- **assertz(Fact)** — adds Fact to the end of the database for its predicate
- **asserta(Fact)** — adds Fact to the beginning, tried first on the next query
- **retract(Fact)** — removes the first matching fact from the database
- :- dynamic(Name/Arity). must be declared before assert/retract can target a predicate with no existing clauses
- A runtime counter (retract old value, assert new one) is genuinely impossible using only static facts and rules
- assert/retract mean the database's own truth can now depend on execution timing — a real cost to declarative purity, matching haskell2-4's own honest unsafePerformIO acknowledgment
- assert/retract are NOT undone on backtracking — a goal that asserts then fails leaves the assertion permanently in place

CHAPTER 3 OF 8

Negation as Failure

COURSE 2 · CH 3

Negation as Failure

\+ and the closed-world assumption — a real, well-documented limitation

`prolog1-8` ended with a brief preview: Course 2 would properly explain `\+`, Prolog's negation operator, commonly built internally from exactly the cut-based pattern that chapter showed. This chapter delivers on that promise — and, in keeping with this course's own honesty about assert/retract last chapter, is equally direct about what `\+` genuinely does *not* guarantee.

What \+ Actually Means

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

Read this the way `prolog1-8` taught you to read cut. If `Goal` succeeds, the first clause's body runs: the cut commits, then `fail` forces the whole thing to fail — so `\+ Goal` fails whenever `Goal` itself succeeds. If `Goal` fails, the first clause never gets past its own body, so Prolog falls through to the second clause, `\+ Goal.`, which unconditionally succeeds. The net effect: `\+ Goal` succeeds exactly when `Goal` cannot be proven, and fails exactly when it can.

```
bird(tweety).
bird(polly).
bird(pingu).
penguin(pingu).

can_fly(X) :- bird(X), \+ penguin(X).

?- can_fly(tweety).
true.

?- can_fly(pingu).
false.
```

This is the classic, genuinely useful shape: "assume something is true unless it can specifically be shown otherwise." `can_fly(pingu)` fails because `penguin(pingu)` succeeds, so `\+ penguin(pingu)` fails — the

negation working exactly as intended here.

The Closed-World Assumption

Notice what `\+ Goal` is **actually** testing: not "Goal is false," but "Goal cannot currently be proven from what's in the database." Prolog operates under the **closed-world assumption** — anything not provable from the known facts and rules is treated as false, full stop, with no third option for "unknown." That's a genuinely different, stronger claim than classical logical negation makes, and it's the real, well-documented limitation this chapter is named for.

```
flight(paris, london).
flight(london, paris).
flight(paris, rome).

?- \+ flight(paris, tokyo).
true.
```

This succeeds — but only because this particular `flight/2` database happens not to list a Paris-Tokyo route, not because a Paris-Tokyo flight is provably impossible. **Absence of evidence isn't evidence of absence**: `\+` can only ever report "not provable from what I currently know," and an incomplete database will report that honestly, misleadingly, exactly like a complete one would.

A Concrete Gotcha — Unbound Variables Inside a Negated Goal

```
?- \+ flight(paris, X).
false.

?- \+ flight(rome, X).
true.
```

The first query fails, because `flight(paris, X)` genuinely does succeed — for `X = london`, among others — so its negation fails. The second succeeds, because nothing matches `flight(rome, X)` at all in this database. The gotcha: with a free variable inside `Goal`, `\+ Goal` is really asking "does *no* value of X make this true," existentially quantified over every possible binding — not "is this specific unknown X not a flight destination." And whichever way it resolves, `X` itself comes back **unbound** either way — `\+` never exposes any binding `Goal` might have found internally, even when it fails because of one.

	CLAIM BEING MADE	HANDLES AN INCOMPLETE DATABASE?
Classical logical negation	`Goal is actually false`	`yes — true/false/unknown are distinct`

	CLAIM BEING MADE	HANDLES AN INCOMPLETE DATABASE?
Prolog's \+ (negation as failure)	`Goal cannot currently be proven`	`no — "not provable" and "false" are collapsed into one`

GROUND YOUR GOAL BEFORE NEGATING IT WHEN POSSIBLE

`\+` behaves most predictably when every variable inside `Goal` is already bound (ground) by the time it runs — the "generate, then test" idiom: find a candidate value first with an ordinary goal, and only then check `\+ some_condition(Candidate)` against it, rather than leaving free variables for `\+` itself to reason about existentially.

AN INCOMPLETE DATABASE CAN MAKE \+ REPORT A CONFIDENT, WRONG-FEELING "TRUE"

Every use of `\+` is only ever as trustworthy as the completeness of the facts and rules it's checking against. A genuinely missing flight route, a genuinely-not-yet-recorded fact, and a genuinely impossible one all look **identical** to `\+` — this is the real, unavoidable cost of the closed-world assumption, not a bug to be fixed.

Coding Challenges

CHALLENGE 1

Write the bird/1, penguin/1, and can_fly/1 predicates exactly as shown in the chapter, add a new fact bird(ostrich) and penguin-style fact flightless(ostrich), update can_fly/1 to also check \+ flightless(X), and confirm can_fly(ostrich) correctly fails.

View solution

CHALLENGE 2

Using the flight/2 facts from the chapter, write a query using \+ that appears to confirm there is no flight from london to tokyo, then write a comment explaining why this result does not actually prove such a flight is impossible in reality.

View solution

CHALLENGE 3

Write a short comment explaining, using \+ flight(paris, X) as the concrete example, why X remains unbound after the query runs regardless of whether the query itself succeeds or fails.

View solution

CHAPTER 3 QUICK REFERENCE

- **\+ Goal** succeeds when Goal cannot be proven, fails when Goal succeeds — built internally from Goal, !, fail exactly as prolog1-8 previewed
- This is negation *as failure*, not classical logical negation — "not provable" is treated as "false," with no separate "unknown"
- The closed-world assumption: anything absent from the database is treated as false, even if it's simply missing rather than genuinely impossible
- \+ Goal with a free variable asks "does no value make Goal true" — existentially quantified, not testing one specific unknown value
- \+ never exposes any variable bindings from Goal, whether it succeeds or fails
- Ground a goal (bind its variables first) before negating it when possible, for predictable results

CHAPTER 4 OF 8

Difference Lists

COURSE 2 · CH 4

Difference Lists

An open-ended list representation that makes concatenation genuinely O(1)

`prolog1-5` and `prolog1-6` covered ordinary Prolog lists — `[H|T]`, recursion over them, the everyday tools. This chapter revisits those same lists with a performance question those chapters never raised: what happens when you need to build up a large list by repeatedly concatenating pieces onto it? The naive answer turns out to be genuinely slow — and difference lists are the real, standard technique for fixing that.

The Problem — Naive append/3 Is O(n), and Repeated Appending Is Worse

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

Every call to `append/3` walks the *entire* first list, one element at a time, before it can even begin producing the combined result — a call on a list of length *n* costs *n* steps. That's fine for one call. It becomes a real problem the moment you need to build a list by appending onto it repeatedly inside a loop or a recursive accumulation — say, flattening a list of ten thousand small lists one at a time. Each `append` call re-walks everything accumulated *so far*, so the total cost isn't 10,000 cheap steps — it's **O(n^2)**, the same quadratic blowup that shows up whenever repeated concatenation is done the naive way in any language.

The Idea — An Open List With a Known "Hole"

A **difference list** represents a list not as one closed term, but as a *pair*: the list itself, and a variable marking its own currently-unbound tail — conventionally written `List-Hole`. For example, `[a, b, c | Hole]` paired with that same `Hole` represents the list `[a, b, c]`, but with its tail left deliberately open rather than closed off with `[]`.

The entire point of leaving that tail open: concatenating two difference lists no longer requires walking either one. It's a single unification.

```
dl_concat(A-B, B-C, A-C).
```

Read this literally: the first difference list is "some list `A`, whose open tail is `B`." The second is "some list `B`, whose open tail is `C`." Concatenating them means unifying the first list's hole (`B`) directly with the second list's own list (also written `B`) — which *is* the second list, spliced straight into the first one's open tail, for free.

A Concrete Worked Example

```
?- L1 = [1, 2, 3 | H1], DL1 = L1-H1,
   L2 = [4, 5 | H2], DL2 = L2-H2,
   dl_concat(DL1, DL2, Combined).

Combined = [1, 2, 3, 4, 5 | H2]-H2.
```

`H1` — the first list's hole — unifies with `L2`, the entire second list, so `L1` becomes `[1, 2, 3, 4, 5 | H2]` without a single element of either original list ever being walked or copied. The combined difference list still has an open tail, `H2` — concatenation didn't close anything, it just spliced two open lists into one, still open.

Closing a Difference List

An open difference list isn't yet an ordinary Prolog list — closing one just means unifying its remaining hole with `[]`:

```
?- Combined = ClosedList-H2, H2 = [].
ClosedList = [1, 2, 3, 4, 5].
```

Only at this final step does the term become a genuine, closed `[1, 2, 3, 4, 5]` — indistinguishable at that point from a list built the ordinary way.

TECHNIQUE	COST PER CONCATENATION	COST OF N REPEATED CONCATENATIONS
Naive append/3	**O(length of first list)**	**O(n²) overall**
Difference lists	**O(1) — one unification**	**O(n) overall**

THIS EXACT TECHNIQUE REAPPEARS, HIDDEN, IN THE NEXT CHAPTER

`prolog2-5`'s Definite Clause Grammars translate every `-->` rule into ordinary clauses threading a pair of list arguments through the whole grammar — that threaded pair genuinely **is** a difference list, doing exactly the same job it does here, just generated automatically by DCG notation instead of written out by hand.

AN UNCLOSED HOLE IS FRAGILE — BINDING IT TOO EARLY SILENTLY BREAKS EVERYTHING

The entire technique depends on the hole staying a genuinely **unbound** variable until the moment you deliberately close it. If something accidentally unifies that hole with `[]` (or anything else) before every intended concatenation has happened, the list is prematurely closed — any later `dl_concat` call expecting to splice something into that hole will simply fail to unify, often with no obvious error pointing back at the real cause.

Coding Challenges

CHALLENGE 1

Write the dl_concat/3 predicate exactly as shown in the chapter, build two difference lists representing [a, b] and [c, d, e], concatenate them, and close the result to show the final ordinary list [a, b, c, d, e].

View solution

CHALLENGE 2

Extend Challenge 1 to concatenate three difference lists together (rather than two) using dl_concat/3 twice, and explain in a comment why this stays O(1) per concatenation regardless of how long each individual list is.

View solution

CHALLENGE 3

Write a short comment demonstrating, with a concrete query, what goes wrong if a difference list's hole is unified with [] before a second dl_concat/3 call tries to splice another list into it.

View solution

CHAPTER 4 QUICK REFERENCE

- Naive append/3 costs O(length of the first list) per call — repeated appending in a loop becomes $O(n^2)$ overall
- A difference list is represented as List-Hole, where Hole is the list's own currently-unbound tail variable
- **dl_concat(A-B, B-C, A-C).** — concatenation is one unification (the first hole becomes the second list), genuinely O(1)
- Close a difference list by unifying its remaining hole with []
- DCGs (prolog2-5) generate exactly this pattern automatically via the --> notation's threaded list-pair arguments
- The hole must stay genuinely unbound until deliberately closed — binding it early silently breaks any later concatenation

CHAPTER 5 OF 8

DCGs — Definite Clause Grammars

notation as sugar over the difference-list threading from the previous chapter, nonterminals calling nonterminals, the {}/1 escape hatch for ordinary Prolog code, and a practical worked expression-summing grammar ========== -->

COURSE 2 · CH 5

DCGs — Definite Clause Grammars

Prolog's own built-in parser-generator sugar — a real, practical strength unique among the languages on this site

`prolog2-4` ended with a promise: the difference-list threading built by hand there reappears, hidden, inside Prolog's own `-->` notation. This chapter delivers on that — DCGs are Prolog's built-in grammar-writing sugar, and no other language covered on this site has anything comparable baked directly into the language itself, not layered on as a separate parser-combinator library.

The --> Notation

```
greeting --> [hello], [world].

?- phrase(greeting, [hello, world]).
true.

?- phrase(greeting, [hello, there]).
false.
```

A DCG rule looks like an ordinary Prolog clause, but with `-->` instead of `:-`, and its body written as a sequence of **terminals** — literal list elements in square brackets — to be matched against the input, left to right. `phrase/2` is how you actually run one: `phrase(NonTerminal, List)` checks whether `List` can be fully consumed by the grammar rule `NonTerminal`.

What --> Is Actually Sugar Over

Every DCG rule is translated, automatically, into an ordinary Prolog clause with two extra hidden arguments — an input-so-far list and a remaining-list, threaded through exactly like `prolog2-4`'s own `List-Hole` pair. Conceptually, `greeting --> [hello], [world].` becomes:

```
greeting(S0, S) :-
    S0 = [hello|S1],
    S1 = [world|S].
```

Each terminal peels one element off the front of the "remaining input so far" list, threading the leftover tail into the next goal — `S0` to `S1` to `S`, the exact same open-list-with-a-hole pattern the previous chapter built by hand. `phrase(greeting, [hello, world])` is really just calling `greeting([hello, world], [])` — the whole list in, nothing left over.

Nonterminals — Rules Calling Rules

```
sentence --> noun_phrase, verb_phrase.
noun_phrase --> [the], noun.
verb_phrase --> verb, noun_phrase.
noun --> [cat].
noun --> [dog].
verb --> [chased].

?- phrase(sentence, [the, dog, chased, the, cat]).
true.
```

A DCG rule body can reference other DCG rules, not just literal terminals — each nonterminal call receives whatever input is left after the rules before it, and passes along whatever remains after it consumes its own piece. This is exactly how a real grammar is meant to compose: small rules for individual pieces (`noun`, `verb`), combined into larger ones (`noun_phrase`, `sentence`) without any of them needing to know how the input-threading actually works underneath.

Embedding Ordinary Prolog Code — Curly Braces

```
even_number(N) --> [N], { number(N), 0 is N mod 2 }.

?- phrase(even_number(N), [4]).
N = 4.

?- phrase(even_number(N), [7]).
false.
```

Anything inside `{ }` inside a DCG body is treated as an **ordinary Prolog goal**, run as-is rather than being interpreted as a grammar symbol to match against the input list — the escape hatch that lets a grammar rule check arithmetic, call another predicate, or do anything else Prolog can normally do, mid-parse.

A Practical Use — Summing an Expression List

```
expr(Sum) --> term(T), expr_rest(T, Sum).

expr_rest(Acc, Sum) -->
    [plus], term(T),
    { Acc1 is Acc + T },
    expr_rest(Acc1, Sum).
expr_rest(Sum, Sum) --> [].

term(T) --> [T], { number(T) }.

?- phrase(expr(Sum), [1, plus, 2, plus, 3]).
Sum = 6.
```

A genuinely useful grammar, not just a toy: `expr` parses a term, then repeatedly looks for `plus` followed by another term, accumulating the running total with `prolog1-7`'s own `is/2` inside a curly-brace escape at each step, until `expr_rest`'s empty-list base case is reached and the accumulated sum is unified with the final result.

	UNDERLYING MECHANISM	WHAT YOU WRITE
Hand-written difference lists	**List-Hole pairs, threaded manually**	**every S0/S argument, explicitly**
DCGs (--> notation)	**the exact same List-Hole threading**	**grammar rules only — the threading is generated for you**

PHRASE/2 AND PHRASE/3 ARE HOW DCG RULES ARE ACTUALLY CALLED

A DCG rule compiles to a predicate with two extra hidden arguments, so it's never called directly by its plain name — `phrase(NonTerminal, List)` handles that translation for you. `phrase/3` additionally accepts a "leftover" argument, useful when the grammar is only meant to match a *prefix* of the input rather than the whole list.

FORGETTING THE CURLY BRACES AROUND ORDINARY PROLOG GOALS

Writing `N is Acc + T` directly inside a DCG body — without wrapping it in `{ }` — doesn't run it as Prolog code at all; it gets treated as a *terminal*, an attempt to match a literal grammar symbol against the input list, which either fails outright or raises a type error. Every ordinary Prolog goal inside a `-->` body needs its own `{ }`, without exception.

Coding Challenges

CHALLENGE 1

Write the sentence/noun_phrase/verb_phrase grammar exactly as shown in the chapter, add a second noun (e.g. [mouse]) and a second verb (e.g. [saw]), and confirm phrase/2 accepts a new valid sentence combining them.

View solution

CHALLENGE 2

Write the expr/term/expr_rest grammar exactly as shown in the chapter, then use phrase/2 to sum a longer list such as [5, plus, 10, plus, 15, plus, 20], confirming the correct total.

View solution

CHALLENGE 3

Write a short comment demonstrating what happens if the { } braces around { Acc1 is Acc + T } in expr_rest/2 are accidentally removed, explaining specifically why Prolog then tries to treat it as a grammar terminal rather than running it as arithmetic.

View solution

CHAPTER 5 QUICK REFERENCE

- **Rule --> Body.** defines a grammar rule; terminals are written as literal lists (e.g. [hello])
- **phrase(NonTerminal, List)** runs a DCG rule against List — really calling the rule's own hidden two-argument compiled form
- Every --> rule compiles to a predicate threading an input-so-far/remaining-input pair — the exact same difference-list pattern from prolog2-4
- DCG rules can call other DCG rules as nonterminals, composing small grammar pieces into larger ones
- **{ Goal }** runs Goal as ordinary Prolog code inside a DCG body, rather than treating it as a grammar terminal to match
- Forgetting { } around a Prolog goal inside a DCG body is a common, easy-to-miss source of failures or type errors

CHAPTER 6 OF 8

Building a Meta-Interpreter

COURSE 2 · CH 6

Building a Meta-Interpreter

A Prolog interpreter written in Prolog — the real payoff of genuine homoiconicity

`haskell2-8` closed the Haskell track with a capstone interpreter: a hand-built `Expr a` GADT representing a small custom expression language as data, plus an `evalM` function pattern-matching over it. That interpreter needed real, deliberate design — Haskell had to invent a data representation of "a program" from scratch, entirely separate from actual Haskell syntax. This chapter builds a Prolog interpreter *in* Prolog, and it comes out startlingly short — for a reason worth naming precisely.

What "Meta-Interpreter" Means

A meta-interpreter is a program that executes other programs — an interpreter written in the same language it interprets. Here, that means writing a Prolog predicate, `solve/1`, that itself carries out the job Prolog's own engine normally does: given a goal, prove it by finding and running matching clauses.

Homoiconicity — Code Is Data

A clause body like `parent(X, Y), parent(Y, Z)` isn't secretly encoded as something else internally — it **is** the ordinary term `','(parent(X,Y), parent(Y,Z))`, built from the same `,/2` functor that appears anywhere else a comma joins two things. Prolog source code is, structurally, just Prolog terms — the same terms every other chapter in this course has manipulated with unification, `findall`, and pattern matching. This is **homoiconicity**: code and data share one representation, with no translation step required to go from "a program" to "a value my own language can inspect."

The Core Meta-Interpreter

```
solve(true) :- !.
solve((A, B)) :- !, solve(A), solve(B).
solve(A) :- clause(A, Body), solve(Body).
```

Three clauses, cut-guarded so each case commits once matched, exactly the way `prolog1-8` taught you to read cut. `clause(Head, Body)` is a built-in that looks up a user-defined clause matching `Head`, unifying `Body` with its body — `true` for an ordinary fact, backtracking over every matching clause if there's more than one.

Tracing solve/1 By Hand

```
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(liz, jim).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).

?- solve(grandparent(tom, Z)).
Z = ann ;
Z = jim.
```

`solve(grandparent(tom, Z))` falls through to the third clause: `clause(grandparent(tom, Z), Body)` unifies `Body` with `(parent(tom, Y), parent(Y, Z))`. Recursing into `solve/1` on that body hits the second clause instead — a conjunction — splitting into `solve(parent(tom, Y))` followed by `solve(parent(Y, Z))`. The first call looks up `parent(tom, Y)` via `clause/2` again, finds `Body = true` for `Y = bob` (or, on backtracking, `Y = liz`), and `solve(true)` succeeds via the first clause. The second call repeats the same process for `parent(Y, Z)`, producing `Z = ann` or `Z = jim` depending on which `Y` was chosen. Nothing here is special-cased for `grandparent/2` specifically — `solve/1` genuinely re-derives the answer the exact same way Prolog's own engine would, using only `clause/2` lookups and ordinary unification.

The Real Comparison — haskell2-8's Interpreter vs. This One

`haskell2-8`'s `Expr a` GADT needed dedicated constructors for every piece of its toy language — `IntLit`, `BoolLit`, `Add`, `Div`, `If` — and its `evalM` function had to pattern-match each one out by hand, because Haskell source code and Haskell *data* are two entirely separate things; representing "a program" as a value required inventing a whole new type just for that purpose. `solve/1` needed no such invention. It interprets **actual Prolog** — real conjunctions, real user-defined clauses, not a small toy subset reinvented for the exercise — because a Prolog goal already *is* exactly the kind of term Prolog itself is built to unify, inspect, and recurse over. Three clauses were enough precisely because nothing needed translating first.

Extending the Interpreter — Disjunction and Negation

```
solve(true) :- !.
solve((A, B)) :- !, solve(A), solve(B).
solve((A ; B)) :- !, (solve(A) ; solve(B)).
solve(\+ A) :- !, \+ solve(A).
solve(A) :- clause(A, Body), solve(Body).
```

Two new clauses, inserted **before** the generic fallback so their own cuts get the chance to commit first. The disjunction clause simply solves either branch. The negation clause is the most telling one — it reuses the host language's own `\+` directly on the result of a recursive `solve` call, rather than reimplementing `prolog2-3`'s negation-as-failure logic from scratch. `solve/1` stays a thin, faithful layer over the real Prolog engine, not a full reimplementation of one.

	REPRESENTATION OF "A PROGRAM"	WHAT HAD TO BE BUILT
haskell2-8's Expr a interpreter	a hand-designed GADT, separate from real Haskell syntax	a new data type, plus a pattern-matching evaluator over it
This chapter's solve/1	ordinary Prolog terms — real clauses, unchanged	three to five clauses, reusing clause/2 and \+ directly

EACH CLAUSE'S OWN CUT IS DOING REAL WORK

Without the cut in `solve(true) :- !.`, a later `solve(A) :- clause(A, Body), ...` call could still be tried for `A = true` on backtracking, attempting a pointless (and likely failing) `clause(true, Body)` lookup — the same "commit once the right case is identified" pattern `prolog1-8` taught, now put to genuine, load-bearing use inside an interpreter's own dispatch logic.

THIS TOY INTERPRETER DOESN'T HANDLE BUILT-INS LIKE IS/2

Calling `solve(X is 1 + 1)` falls through to the generic clause, which tries `clause(X is 1 + 1, Body)` — and fails, because `is/2` is a built-in predicate with no user-defined clauses for `clause/2` to find. A genuinely complete meta-interpreter needs explicit cases recognizing built-ins and calling them directly (e.g. `solve(G) :- built_in(G), !, call(G).`) rather than routing everything through `clause/2` — an honest scope boundary, left out here to keep the core idea visible.

Coding Challenges

CHALLENGE 1

Write the three-clause solve/1 from the chapter along with the parent/2 and grandparent/2 facts and rule, then query solve(parent(tom, bob)) and solve(grandparent(tom, Z)), confirming both produce the results shown in the chapter's trace.

View solution

CHALLENGE 2

Add the disjunction and negation clauses from the chapter to solve/1, define a predicate likes(tom, pizza) and likes(tom, sushi), then query solve((likes(tom, pizza) ; likes(tom, tacos))) and solve(\+ likes(tom, tacos)), confirming both succeed correctly.

View solution

CHALLENGE 3

Write a short comment explaining why solve(X is 1 + 1) fails with this chapter's interpreter, and what a solve/1 clause recognizing built-in predicates specially (rather than routing them through clause/2) would need to look like.

View solution

CHAPTER 6 QUICK REFERENCE

- **solve(true) :- !.** — the base case, an empty body means the goal is already proven
- **solve((A, B)) :- !, solve(A), solve(B).** — a conjunction is solved by solving each half in turn
- **solve(A) :- clause(A, Body), solve(Body).** — the general case, looking up and recursing into a user-defined clause
- clause(Head, Body) looks up a matching user-defined clause, giving Body = true for a plain fact
- Homoiconicity — Prolog code already is Prolog data — is why solve/1 needed no invented representation the way haskell2-8's Expr a GADT did
- Disjunction and negation extend solve/1 by directly reusing Prolog's own ; and \+, staying a thin layer over the real engine
- This toy interpreter has an honest scope gap: built-in predicates like is/2 have no clause/2 entries and need explicit special-case handling

CHAPTER 7 OF 8

Constraint Logic Programming, A First Look

COURSE 2 · CH 7

Constraint Logic Programming, A First Look

CLP(FD) — a real, practical tool for genuine constraint-satisfaction problems

A lighter chapter than the last — a first look, not an exhaustive tour — at a genuinely practical extension: constraint logic programming over finite domains, **CLP(FD)**. It's exactly the right tool for problems shaped like puzzles — Sudoku, N-Queens, scheduling — and it previews the technique this course's own capstone will lean on.

The Problem With Plain Backtracking Search

```
?- between(1, 9, X), between(1, 9, Y), X + Y =:= 10.
X = 1, Y = 9 ;
X = 2, Y = 8 ;
-- and so on
```

Ordinary Prolog can solve this — `between/3` generates candidate values one at a time, and `=:=` checks the constraint *after* both values are already committed to. That's fine here, but the pattern doesn't scale: for a harder problem with many variables and many constraints, this "generate first, test after" approach can waste enormous effort fully constructing candidates that were doomed from the very first choice, only discovering that on the very last check.

CLP(FD) — Constraints Applied Before Values Are Chosen

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

?- X in 1..9, Y in 1..9, X + Y #= 10, X #< Y, label([X, Y]).
X = 1, Y = 9 ;
X = 2, Y = 8 ;
X = 3, Y = 7 ;
-- and so on, but with far less wasted search than the between/3 version once problems get larger
```

`X in 1..9` declares `X` as a domain variable — not yet a number, but a variable whose eventual value is constrained to that range. Posting `X + Y #= 10` and `X #< Y` immediately narrows both domains through

constraint propagation, before either variable has been assigned any specific value at all — the underlying CLP(FD) solver actively removes impossible values from consideration up front, rather than waiting for backtracking to stumble onto them one by one.

#= vs. is vs. = — A Fourth Operator, Genuinely Different Again

`prolog1-7` distinguished `is`, `=`, and `==`. CLP(FD) adds a fourth, `#=`, distinct from all three: unlike `is/2`, which demands its right-hand side already be fully ground arithmetic or raises an error, `#=` can be posted between variables whose values aren't known yet — it behaves bidirectionally, immediately constraining both sides' *domains* relative to each other, rather than requiring one side to already be a concrete number.

```
?- X is Y + 1.
-- ERROR: Arguments are not sufficiently instantiated

?- X #= Y + 1, Y in 1..5.
X in 2..6,
Y in 1..5.
```

`is/2` fails outright with an unbound `Y`. `#=` instead immediately derives a constrained domain for `X` from whatever domain `Y` already has — real, useful information, produced before any concrete value has been chosen for either variable.

label/1 — Forcing Concrete Values

A domain variable stays exactly that — a range of still-possible values, narrowed by propagation — until something forces the search to actually commit to specific numbers. `label(Vars)` does that: it backtracks through concrete values for each variable in `Vars`, but only within whatever each domain has *already* been narrowed down to by constraint propagation, not the original full range.

	WHEN CONSTRAINTS APPLY	VALUES REQUIRED
Plain Prolog (is/2, =:=)	only after values are fully instantiated	ground numbers on both sides, or an error
CLP(FD) (in, #=, #<, …)	immediately, propagating across domains before any value is chosen	none — works over still-unbound domain variables

THIS IS EXACTLY THE RIGHT TOOL FOR GENUINE CONSTRAINT-SATISFACTION PUZZLES

N-Queens, Sudoku, and scheduling problems are all naturally shaped like "many variables, many simultaneous constraints, find values satisfying all of them" — precisely CLP(FD)'s own strength. Course 2's own capstone (`prolog2-8`) leans directly on this chapter's `in` / `#=` / `label` pattern for exactly that reason.

CLP(FD) IS FINITE-DOMAIN INTEGERS ONLY, AND LABEL/1 IS EASY TO FORGET

The "FD" stands for finite domain — CLP(FD) works over integers with a bounded range, not a general-purpose replacement for `is/2` or real/floating-point arithmetic. And forgetting to load `library(clpfd)`, or forgetting to call `label/1` at the end of a query, is a genuinely common beginner mistake — without `label`, the query reports the narrowed *domains* rather than concrete numbers, which can look like an answer while not actually being one.

Coding Challenges

CHALLENGE 1

Load library(clpfd), declare three domain variables X, Y, Z each in 1..5, post the constraint X + Y + Z #= 9, and use label/1 to find all combinations satisfying it.

View solution

CHALLENGE 2

Write a query using #= where one variable's domain is derived from another's before either is labeled (similar to the chapter's X #= Y + 1 example), show the resulting narrowed domains, and explain in a comment why the equivalent is/2 version would raise an error at that point.

View solution

CHALLENGE 3

Write a short comment explaining what a query like X in 1..9, Y in 1..9, X + Y #= 10 actually reports if label([X, Y]) is left off the end, and why that result isn't yet a usable concrete answer.

View solution

CHAPTER 7 QUICK REFERENCE

- :- use_module(library(clpfd)). loads CLP(FD) support
- **X in Low..High** declares a finite-domain variable, not yet a concrete number
- **#=, #<, #>, …** post constraints that propagate across domains immediately, before any value is chosen

- #= differs from is/2 by working over unbound variables — is/2 demands ground arithmetic or errors
- **label(Vars)** forces concrete values, searching only within each variable's already-narrowed domain
- CLP(FD) is finite-domain integers specifically — not a general is/2 replacement
- The natural fit for genuine constraint-satisfaction puzzles — N-Queens, Sudoku, scheduling — and the technique this course's own capstone builds on

CHAPTER 8 OF 8

Capstone: Building a Small Project

COURSE 2 · CH 8 · CAPSTONE

Capstone: Building a Small Project

An N-Queens solver, and the Prolog track's own closing word on two declarative paradigms

The final chapter of the full Prolog track. This capstone builds a real N-Queens solver — combining `prolog2-7`'s CLP(FD), `prolog1-8`'s cut, `prolog2-1`'s `findall`, and `prolog2-5`'s DCG notation into one working project — and closes with the comparison this entire track was framed around from its very first chapter: two languages, both routinely called "declarative," solving computation in genuinely different ways.

The Problem — N-Queens

Place *N* queens on an *N*×*N* chessboard so that no two attack each other — no shared row, no shared column, no shared diagonal. A classic constraint-satisfaction problem, and exactly the shape `prolog2-7` named as CLP(FD)'s natural strength.

Modeling With CLP(FD)

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

queens(N, Queens) :-
    length(Queens, N),
    Queens ins 1..N,
    safe_queens(Queens),
    label(Queens).

safe_queens([]).
safe_queens([Q|Qs]) :-
    safe_queens(Qs, Q, 1),
    safe_queens(Qs).

safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
    Q0 #\= Q,
    abs(Q0 - Q) #\= D0,
    D1 #= D0 + 1,
    safe_queens(Qs, Q0, D1).
```

`Queens` is a list of *N* domain variables, one per column, each holding that column's queen's row — column position is implicit in the list index, so "no shared column" is automatic. `ins` is `prolog2-7`'s own `in`, pluralized to apply one domain declaration to an entire list at once. `safe_queens/1` walks the list, posting a `#\=` (row clash) and a diagonal-distance constraint between every pair of queens — real unification and real constraint propagation, narrowing the search before `label/1` ever commits to a single value.

One Solution vs. All Solutions

```
?- queens(4, Qs), !.
Qs = [2, 4, 1, 3].  -- one valid placement; the exact solution found first depends on the solver's own s

?- findall(Qs, queens(6, Qs), AllSolutions), length(AllSolutions, Count).
Count = 4.
```

The cut after `queens(4, Qs)` is `prolog1-8`'s own backtracking-control tool, put to genuine use — it commits to the first solution `label/1` finds rather than leaving choice points open for a caller who only wants one answer. `findall` — `prolog2-1`'s own meta-predicate — does the opposite: it drives `queens/2` to exhaustion, collecting **every** valid 6-queens board into one list. Four is the well-known correct count of distinct solutions for *N* = 6.

Rendering a Board With a DCG

```
row_symbols(Queens, Row, Symbols) :-
    length(Queens, N),
    phrase(row(Queens, Row, 1, N), Symbols).

row(_, _, C, N) --> { C > N }, [].
row(Queens, Row, C, N) -->
    { C =< N, nth1(C, Queens, R) },
    symbol(R, Row),
    { C1 is C + 1 },
    row(Queens, Row, C1, N).

symbol(Row, Row) --> !, [q].
symbol(_, _) --> [e].
```

`prolog2-5`'s `-->` notation, reused to *generate* a row rather than parse one — DCGs run perfectly well in either direction. For each column `C`, `symbol/2` emits `q` if that column's queen sits in the row currently being rendered, otherwise `e`.

The cut in `symbol(Row, Row) --> !, [q].` is genuinely load-bearing, not just tidy — and it's a **red** cut by `prolog1-8`'s own test. Without it, backtracking into `symbol/2` at a queen's own column could still also match the catch-all second clause, producing an *additional*, factually wrong reading of that cell as empty. Removing this cut would genuinely change which results the predicate can produce — the defining test for red vs. green.

```
print_solution(Queens) :-
    length(Queens, N),
    numlist(1, N, Rows),
    forall(member(Row, Rows), print_row(Queens, Row)).

print_row(Queens, Row) :-
    row_symbols(Queens, Row, Symbols),
    maplist(symbol_char, Symbols, Chars),
    atomic_list_concat(Chars, RowAtom),
    writeln(RowAtom).

symbol_char(e, '. ').
symbol_char(q, 'Q ').

?- queens(4, Qs), !, print_solution(Qs).
. . Q .
Q . . .
. . . Q
. Q . .
```

One last, quiet connection: `forall/2` is standardly defined as `forall(Cond, Action) :- \+ (Cond, \+ Action).` — double negation, built directly from `prolog2-3`'s own `\+`. This capstone never calls `\+` by

name, but it's running underneath every row printed, exactly the way `prolog2-6` showed a meta-predicate can be a thin layer reusing something more fundamental.

Chapter Attribution

PIECE	CHAPTER
Domain variables, #\=, #=, labeling	**prolog2-7 — CLP(FD)**
Committing to the first solution	**prolog1-8 — Cut**
Collecting and counting all solutions	**prolog2-1 — findall**
The board-rendering grammar	**prolog2-5 — DCGs**
The red cut inside symbol/2	**prolog1-8 — green vs. red cuts, applied for real**
forall/2's own internal definition	**prolog2-3 — Negation as Failure (used indirectly)**
Unification, backtracking throughout	**Course 1 — prolog1-3, prolog1-4**

STILL OUT OF SCOPE, HONESTLY

`prolog2-2`'s assert/retract has no natural role here — this solver needs no mutable state remembered across calls, since the whole problem is expressed and solved as one self-contained relation. `prolog2-4`'s difference lists are used only invisibly, underneath the DCG translation, never written out by hand in this chapter. And `prolog2-6`'s meta-interpreter — a teaching device for demonstrating homoiconicity — has no reason to reappear in a capstone that's solving a real problem rather than interpreting one; `solve/1` was never meant to replace Prolog's own engine for everyday use.

Two Declarative Paradigms — Closing the Prolog Track

Every earlier chapter that named Haskell did so piece by piece — unification against pattern matching, backtracking against the `[]` monad, cut and `assert` against `unsafePerformIO`. This capstone is the place to say the whole comparison plainly, using N-Queens itself as the concrete anchor.

A Haskell N-Queens solver would be a genuine **function** — most naturally built over the `[]` monad's own `do`-notation (`haskell2-3`), generating candidate placements, filtering out attacking ones, and returning a concrete list of valid boards as its result. The search is real, but it's expressed as ordinary values flowing through ordinary function composition — `solve(N) :: [[Int]]` is a pure computation, evaluated to produce a value, the same as any other Haskell expression.

This chapter's `queens/2` is not a function computing a value — it's a **relation**, and `Queens` is not returned so much as *discovered*: a set of variable bindings that happen to satisfy every constraint posted against them, found by an engine built to search exactly that way. CLP(FD) makes the search efficient, but it doesn't change

what's actually happening underneath — Prolog was never evaluating an expression to a value; it was finding bindings that make a relation true.

	WHAT "SOLVING N-QUEENS" MEANS	HOW NONDETERMINISM IS EXPRESSED
Haskell	`a pure function evaluated to a list of results`	`the [] monad — do-notation over ordinary values`
Prolog	`a relation, satisfied by a discovered set of bindings`	`backtracking search, native to the language itself`

Both are honestly "declarative" — neither language makes you write an explicit loop or a mutable counter to find these boards. But they earn that label in two structurally different ways, and this whole track, from `prolog1-1`'s very first "no `main`" surprise to this capstone's own solver, has been the concrete demonstration of exactly what that difference looks like in practice.

Coding Challenges

CHALLENGE 1

Write queens/2 and safe_queens/1 and safe_queens/3 exactly as shown in the chapter, query queens(5, Qs), !, and confirm the result satisfies every constraint (no shared row, no shared diagonal) by hand.

View solution

CHALLENGE 2

Use findall/3 to collect every solution to queens(5, Qs), report the total count, and confirm it matches the well-known correct value of 10 distinct 5-queens solutions.

View solution

CHALLENGE 3

Write a short comment explaining, with a concrete backtracking query into row_symbols/3, what extra (incorrect) result the cut inside symbol/2 is specifically preventing, and why this makes it a red cut rather than a green one by the chapter's own test.

View solution

CHAPTER 8 QUICK REFERENCE — PROLOG INTERMEDIATE/ADVANCED COMPLETE

- queens/2 models N-Queens as domain variables, #\= and diagonal-distance constraints, and label/1 — prolog2-7's CLP(FD) put to real use
- Cut after queens(N, Qs) commits to one solution; findall/3 collects and counts every solution instead
- A DCG can generate output, not just parse input — row_symbols/3 reuses prolog2-5's --> notation to render a board
- symbol/2's cut is a genuine red cut — removing it would let an incorrect extra "empty" reading of a queen's own cell through
- forall/2 is itself built from double negation (\+), a quiet reappearance of prolog2-3 underneath a capstone that never calls \+ directly
- Haskell solves N-Queens as a pure function evaluated to a value; Prolog solves it as a relation, satisfied by bindings a search engine discovers — the track's own throughline, made concrete one final time
- **Prolog Intermediate/Advanced is now complete. The full Prolog track — 16 chapters across 2 courses — is done.**