Logic Programming & Prolog

(Formulas as Programs)

Hype for Types

February 16, 2026

What We'll Talk About

- A *different* connection between logic and computation

What We'll Talk About

- A *different* connection between logic and computation
- Proof search as computation (formulas-as-programs)

What We'll Talk About

- A *different* connection between logic and computation
- Proof search as computation (formulas-as-programs)
- Prolog: a programming language built on this idea

What We'll Talk About

- A *different* connection between logic and computation
- Proof search as computation (formulas-as-programs)
- Prolog: a programming language built on this idea
- Key concepts: unification, backtracking, modes

Computation vs. Deduction

Two Connections Between Logic and Computation

So far in this course:

- We explored **proofs-as-programs** (Curry–Howard)
- A constructive proof *is* a program
- e.g. $\lambda x : A.\ x$ proves $A \supset A$

Two Connections Between Logic and Computation

So far in this course:

- We explored **proofs-as-programs** (Curry–Howard)
- A constructive proof *is* a program
- e.g. $\lambda x : A.\ x$ proves $A \supset A$

Today: an entirely different idea!

- The **search for a proof** is the computation
- Logical rules *are* the program
- Proof search according to a fixed strategy *executes* the program

Computation vs. Deduction

Computation: Start from an expression, mechanically apply rules, get a result.

- $25 + 46 \rightarrow 71$ (no creativity needed)

Computation vs. Deduction

Computation: Start from an expression, mechanically apply rules, get a result.

- $25 + 46 \to 71$ (no creativity needed)

Deduction: Start from a conjecture, try to construct a proof using inference rules.

- For all $n > 2$: $a^n + b^n \neq c^n$ ... 357 years of hard work ... QED

Computation vs. Deduction

Computation: Start from an expression, mechanically apply rules, get a result.

- $25 + 46 \to 71$ (no creativity needed)

Deduction: Start from a conjecture, try to construct a proof using inference rules.

- For all $n > 2$: $a^n + b^n \neq c^n$ ... 357 years of hard work ... QED

Key Insight

If we *fix a strategy* for proof search, we remove the guesswork. Deduction becomes computation!

The Big Idea

Logic program computation

$=$

proof search according to a **fixed strategy**

The Big Idea

Logic program computation

$=$

proof search according to a **fixed strategy**

By knowing what the strategy is, we can implement algorithms in logic and execute them by proof search.

From Inference Rules to Programs

A Quick Refresher: Inference Rules

An inference rule has the form:

$$\frac{J_1 \quad \cdots \quad J_n}{J}$$

A Quick Refresher: Inference Rules

An inference rule has the form:

$$\frac{J_1 \quad \cdots \quad J_n}{J}$$

- J is the **conclusion**
- $J_1, \ldots, J_n$ are the **premises**
- If $n = 0$, it's an axiom

Example: Natural Numbers

Represent natural numbers as: $0, s(0), s(s(0)), \ldots$

Example: Natural Numbers

Represent natural numbers as: $0, s(0), s(s(0)), \ldots$

Define the predicate $even(N)$:

$$\frac{}{even(0)}\ (\textsc{evz}) \qquad \frac{even(N)}{even(s(s(N)))}\ (\textsc{evs})$$

Example: Natural Numbers

Represent natural numbers as: $0, s(0), s(s(0)), \ldots$

Define the predicate $even(N)$:

$$\frac{}{even(0)}\ (\text{EVZ}) \qquad \frac{even(N)}{even(s(s(N)))}\ (\text{EVS})$$

A proof that 4 is even:

$$\dfrac{\dfrac{\dfrac{}{even(0)}\ (\text{EVZ})}{even(s(s(0)))}\ (\text{EVS})}{even(s(s(s(s(0)))))}\ (\text{EVS})$$

Goal-Directed Search

Strategy: Given a goal, find rules whose conclusion matches, then recursively prove the premises.

Goal-Directed Search

Strategy: Given a goal, find rules whose conclusion matches, then recursively prove the premises.

Goal: $even(s(s(0)))$

1. Only EVS matches $\rightarrow$ subgoal: $even(0)$

Goal-Directed Search

Strategy: Given a goal, find rules whose conclusion matches, then recursively prove the premises.

Goal: $\mathit{even}(s(s(0)))$

1. Only EVS matches $\rightarrow$ subgoal: $\mathit{even}(0)$
2. Only EVZ matches $\rightarrow$ no premises $\rightarrow$ done!

Goal-Directed Search

Strategy: Given a goal, find rules whose conclusion matches, then recursively prove the premises.

Goal: $even(s(s(0)))$

1. Only EVS matches $\rightarrow$ subgoal: $even(0)$
2. Only EVZ matches $\rightarrow$ no premises $\rightarrow$ done!

Answer: **yes**!

Goal-Directed Search

Strategy: Given a goal, find rules whose conclusion matches, then recursively prove the premises.

Goal: $even(s(s(0)))$

1. Only EVS matches $\rightarrow$ subgoal: $even(0)$
2. Only EVZ matches $\rightarrow$ no premises $\rightarrow$ done!

Answer: **yes**!

Goal: $even(s(s(s(0))))$

1. Only EVS matches $\rightarrow$ subgoal: $even(s(0))$
2. No rule matches $\rightarrow$ **no**!

Computing Values: Answer Substitutions

Define addition: $plus(M, N, P)$ means $M + N = P$

$$\frac{}{plus(0, N, N)}\ ((\text{PZ})) \qquad \frac{plus(M, N, P)}{plus(s(M), N, s(P))}\ ((\text{PS}))$$

Computing Values: Answer Substitutions

Define addition: $\mathit{plus}(M, N, P)$ means $M + N = P$

$$\frac{}{\mathit{plus}(0, N, N)}\ ((\text{PZ})) \qquad \frac{\mathit{plus}(M, N, P)}{\mathit{plus}(s(M), N, s(P))}\ ((\text{PS}))$$

Query: $\mathit{plus}(s(0), s(0), R)$ ("what is $1 + 1$?")

Computing Values: Answer Substitutions

Define addition: $\mathit{plus}(M, N, P)$ means $M + N = P$

$$\frac{}{\mathit{plus}(0, N, N)}\ (\textsc{pz}) \qquad \frac{\mathit{plus}(M, N, P)}{\mathit{plus}(s(M), N, s(P))}\ (\textsc{ps})$$

Query: $\mathit{plus}(s(0), s(0), R)$ ("what is $1 + 1$?")

1. PS matches: $R = s(P)$, subgoal: $\mathit{plus}(0, s(0), P)$

Computing Values: Answer Substitutions

Define addition: $plus(M, N, P)$ means $M + N = P$

$$\frac{}{plus(0, N, N)}\ (\textsc{pz}) \qquad \frac{plus(M, N, P)}{plus(s(M), N, s(P))}\ (\textsc{ps})$$

Query: $plus(s(0), s(0), R)$ ("what is $1 + 1$?")

1. PS matches: $R = s(P)$, subgoal: $plus(0, s(0), P)$
2. PZ matches: $P = s(0)$

Computing Values: Answer Substitutions

Define addition: $\mathit{plus}(M, N, P)$ means $M + N = P$

$$\frac{}{\mathit{plus}(0, N, N)}\ (\textsc{pz}) \qquad \frac{\mathit{plus}(M, N, P)}{\mathit{plus}(s(M), N, s(P))}\ (\textsc{ps})$$

Query: $\mathit{plus}(s(0), s(0), R)$ ("what is $1 + 1$?")

1. PS matches: $R = s(P)$, subgoal: $\mathit{plus}(0, s(0), P)$
2. PZ matches: $P = s(0)$
3. So $R = s(s(0))$ — that's 2!

Computing Values: Answer Substitutions

Define addition: $plus(M, N, P)$ means $M + N = P$

$$\frac{}{plus(0, N, N)}\ (\textsc{pz}) \qquad \frac{plus(M, N, P)}{plus(s(M), N, s(P))}\ (\textsc{ps})$$

Query: $plus(s(0), s(0), R)$ ("what is $1 + 1$?")

1. PS matches: $R = s(P)$, subgoal: $plus(0, s(0), P)$
2. PZ matches: $P = s(0)$
3. So $R = s(s(0))$ — that's 2!

The unknowns (R, P) are called **logic variables**.
The result ($R = s(s(0))$) is the **answer substitution**.

Prolog

Meet Prolog (1972)

- **Pro**grammation en **Lo**gique
- The first (and most popular) logic programming language

Meet Prolog (1972)

- **Pro**grammation en **Lo**gique
- The first (and most popular) logic programming language
- Write inference rules $\rightarrow$ execute them by proof search

Meet Prolog (1972)

- **Pro**grammation en **Lo**gique
- The first (and most popular) logic programming language
- Write inference rules $\rightarrow$ execute them by proof search
- Great for: symbolic computation, search, parsing, type checking

Meet Prolog (1972)

- **Pro**grammation en **Lo**gique
- The first (and most popular) logic programming language
- Write inference rules $\rightarrow$ execute them by proof search
- Great for: symbolic computation, search, parsing, type checking
- Quirky: no types, failure-driven, extralogical features

Prolog Syntax

An inference rule:

$$\frac{J_1 \quad \cdots \quad J_n}{J}$$

becomes a **clause**:

```
J :- J1, ..., Jn.
```

Prolog Syntax

An inference rule:

$$\frac{J_1 \quad \cdots \quad J_n}{J}$$

becomes a **clause**:

```
J :- J1, ..., Jn.
```

Read as: "J **if** J_1 and $\cdots$ and J_n"

Prolog Syntax

An inference rule:

$$\frac{J_1 \quad \cdots \quad J_n}{J}$$

becomes a **clause**:

```
J :- J1, ..., Jn.
```

Read as: "J **if** J_1 and $\cdots$ and J_n"

- J is the **head**, $J_1, \ldots, J_n$ is the **body**
- An axiom (no premises) is just: `J.`
- **Upper case** = variables, **lower case** = constants/predicates

Even and Plus in Prolog

```
even(z).
even(s(s(N))) :- even(N).

plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

Even and Plus in Prolog

```
even(z).
even(s(s(N))) :- even(N).

plus(z, N, N).
plus(s(M), N, s(P)) :- plus(M, N, P).
```

Queries at the Prolog prompt:

```
?- even(s(s(s(s(z))))).
yes

?- plus(s(z), s(z), R).
R = s(s(z))
```

Running Programs "In Reverse"!

Same rules for `plus`, but a different *query*:

```
?- plus(M, s(z), s(s(z))).
M = s(z)
```

Running Programs "In Reverse"!

Same rules for `plus`, but a different *query*:

```
?- plus(M, s(z), s(s(z))).
M = s(z)
```

We just used addition to compute **subtraction**!

Running Programs "In Reverse"!

Same rules for `plus`, but a different *query*:

```
?- plus(M, s(z), s(s(z))).
M = s(z)
```

We just used addition to compute **subtraction**!

This is wild

In functional programming, you write one function per direction. In logic programming, the *same relation* can compute multiple directions.

Key Concepts

Unification

When we match a goal against a clause head, we need to make them identical by finding substitutions for variables.

Unification

When we match a goal against a clause head, we need to make them identical by finding substitutions for variables. Goal: $plus(s(0), s(0), R)$

Head: $plus(s(M), N, s(P))$

Unification

When we match a goal against a clause head, we need to make them identical by finding substitutions for variables. Goal: $plus(s(0), s(0), R)$

Head: $plus(s(M), N, s(P))$ Match corresponding subterms:

- $s(0) = s(M) \implies M = 0$
- $s(0) = N \implies N = s(0)$
- $R = s(P) \implies R = s(P)$ (P still unknown)

Unification

When we match a goal against a clause head, we need to make them identical by finding substitutions for variables. Goal: $plus(s(0), s(0), R)$

Head: $plus(s(M), N, s(P))$ Match corresponding subterms:

- $s(0) = s(M) \implies M = 0$
- $s(0) = N \implies N = s(0)$
- $R = s(P) \implies R = s(P)$ (P still unknown)

This process is called **unification**: find the simplest substitution that makes two terms equal.

Unification: The Occurs Check Problem

Consider: does $N = s(s(N))$ have a solution?

Unification: The Occurs Check Problem

Consider: does $N = s(s(N))$ have a solution?

No! The right side always has two more s's than the left.

Unification: The Occurs Check Problem

Consider: does $N = s(s(N))$ have a solution?

No! The right side always has two more s's than the left.

But Prolog skips this check (the **occurs check**) for efficiency and builds a *circular term* instead!

Unification: The Occurs Check Problem

Consider: does $N = s(s(N))$ have a solution?

No! The right side always has two more s's than the left.

But Prolog skips this check (the **occurs check**) for efficiency and builds a *circular term* instead!

Consequence

Prolog's unification is technically *unsound*. For correct unification, use `unify_with_occurs_check/2`.

Backtracking

What if multiple rules match a goal?

Backtracking

What if multiple rules match a goal? Prolog uses **backtracking**:

1. Try the *first* matching clause (in program order)

Backtracking

What if multiple rules match a goal? Prolog uses **backtracking**:

1. Try the *first* matching clause (in program order)
2. If that path fails, *undo* and try the next one

Backtracking

What if multiple rules match a goal? Prolog uses **backtracking**:

1. Try the *first* matching clause (in program order)
2. If that path fails, *undo* and try the next one
3. If all paths fail, the goal fails

Backtracking

What if multiple rules match a goal? Prolog uses **backtracking**:

1. Try the *first* matching clause (in program order)
2. If that path fails, *undo* and try the next one
3. If all paths fail, the goal fails

Choice Points

A point where multiple clauses match is called a **choice point**. Prolog systematically explores alternatives via depth-first search.

Backtracking Example

Query: $plus(M, s(0), s(s(0)))$ ("what minus 1 equals 2?")

Backtracking Example

Query: $plus(M, s(0), s(s(0)))$ ("what minus 1 equals 2?")
Both PZ and PS might match at some point:

1. Try PS first $\rightarrow$ subgoal: $plus(M_1, s(0), s(0))$

Backtracking Example

Query: $plus(M, s(0), s(s(0)))$ ("what minus 1 equals 2?")
Both PZ and PS might match at some point:

1. Try PS first $\rightarrow$ subgoal: $plus(M_1, s(0), s(0))$
2. Try PS again $\rightarrow$ subgoal: $plus(M_2, s(0), 0)$

Backtracking Example

Query: $plus(M, s(0), s(s(0)))$ ("what minus 1 equals 2?")
Both PZ and PS might match at some point:

1. Try PS first $\rightarrow$ subgoal: $plus(M_1, s(0), s(0))$
2. Try PS again $\rightarrow$ subgoal: $plus(M_2, s(0), 0)$
3. No rule matches! **Backtrack.**

Backtracking Example

Query: $plus(M, s(0), s(s(0)))$ ("what minus 1 equals 2?")
Both PZ and PS might match at some point:

1. Try PS first $\rightarrow$ subgoal: $plus(M_1, s(0), s(0))$
2. Try PS again $\rightarrow$ subgoal: $plus(M_2, s(0), 0)$
3. No rule matches! **Backtrack.**
4. Try PZ instead $\rightarrow$ $M_1 = 0$ $\rightarrow$ success!

Backtracking Example

Query: $plus(M, s(0), s(s(0)))$ ("what minus 1 equals 2?")
Both PZ and PS might match at some point:

1. Try PS first $\rightarrow$ subgoal: $plus(M_1, s(0), s(0))$
2. Try PS again $\rightarrow$ subgoal: $plus(M_2, s(0), 0)$
3. No rule matches! **Backtrack.**
4. Try PZ instead $\rightarrow$ $M_1 = 0$ $\rightarrow$ success!

Answer: $M = s(0)$ (i.e., 1)

Modes

A **mode** describes which arguments are inputs (+) and which are outputs (−).

Modes

A **mode** describes which arguments are inputs (+) and which are outputs (−).

The same predicate can be used in different modes:

- `plus(+, +, -)` → addition
- `plus(-, +, +)` → subtraction

Modes

A **mode** describes which arguments are inputs (+) and which are outputs (−).

The same predicate can be used in different modes:

- `plus(+, +, -)` → addition
- `plus(-, +, +)` → subtraction

Caveat

Modes aren't checked by Prolog! You have to verify them yourself, or risk nontermination or wrong answers.

Subgoal Order Matters

Define multiplication using $(m + 1) \times n = (m \times n) + n$:

$$\frac{}{times(0, N, 0)}\ ((\textsc{tz}))$$

$$\frac{times(M, N, P) \quad plus(P, N, Q)}{times(s(M), N, Q)}\ ((\textsc{ts}))$$

Subgoal Order Matters

Define multiplication using $(m + 1) \times n = (m \times n) + n$:

$$\frac{}{\mathit{times}(0, N, 0)}\ ((\textsc{tz})) \qquad \frac{\mathit{times}(M, N, P) \quad \mathit{plus}(P, N, Q)}{\mathit{times}(s(M), N, Q)}\ ((\textsc{ts}))$$

Rule TS has **two subgoals**. Prolog solves them left-to-right.

Subgoal Order Matters

Define multiplication using $(m+1) \times n = (m \times n) + n$:

$$\frac{}{times(0, N, 0)}\ (\textsc{tz}) \qquad \frac{times(M, N, P) \quad plus(P, N, Q)}{times(s(M), N, Q)}\ (\textsc{ts})$$

Rule TS has **two subgoals**. Prolog solves them left-to-right. Does the order matter? Let's compute $1 \times 2 = Q$, i.e. $times(s(0), s(s(0)), Q)$.

Left-to-Right: `times` First (Good!)

Goal: $times(s(0), s(s(0)), Q)$ — apply TS:

Left-to-Right: `times` First (Good!)

Goal: $times(s(0), s(s(0)), Q)$ — apply TS: Subgoals:

$\underbrace{times(0, s(s(0)), P)}_{\text{solve first}}$ then $plus(P, s(s(0)), Q)$

Left-to-Right: `times` First (Good!)

Goal: $times(s(0), s(s(0)), Q)$ — apply TS: Subgoals:

$\underbrace{times(0, s(s(0)), P)}_{\text{solve first}}$ then $plus(P, s(s(0)), Q)$

Step 1: Solve $times(0, s(s(0)), P)$

- Only TZ matches $\rightarrow$ forces $P = 0$

Left-to-Right: `times` First (Good!)

Goal: $times(s(0), s(s(0)), Q)$ — apply TS: Subgoals:

$\underbrace{times(0, s(s(0)), P)}_{\text{solve first}}$ then $plus(P, s(s(0)), Q)$

Step 1: Solve $times(0, s(s(0)), P)$

- Only TZ matches $\rightarrow$ forces $P = 0$

Step 2: Now solve $plus(0, s(s(0)), Q)$ — we *know* $P = 0$!

- Only PZ matches $\rightarrow$ forces $Q = s(s(0))$

Left-to-Right: `times` First (Good!)

Goal: $times(s(0), s(s(0)), Q)$ — apply TS: Subgoals:

$\underbrace{times(0, s(s(0)), P)}_{\text{solve first}}$ then $plus(P, s(s(0)), Q)$

Step 1: Solve $times(0, s(s(0)), P)$

- Only TZ matches $\rightarrow$ forces $P = 0$

Step 2: Now solve $plus(0, s(s(0)), Q)$ — we *know* $P = 0$!

- Only PZ matches $\rightarrow$ forces $Q = s(s(0))$

Answer: $Q = s(s(0))$, i.e. $1 \times 2 = 2$. **Done in 2 steps!**

What If We Solved Right-to-Left? `plus` First (Bad!)

Same goal, but imagine we solve `plus` *before* `times`:

What If We Solved Right-to-Left? `plus` First (Bad!)

Same goal, but imagine we solve `plus` *before* `times`: Subgoals:

$$times(0, s(s(0)), P) \quad \text{and} \quad \underbrace{plus(P, s(s(0)), Q)}_{\text{solve first}}$$

What If We Solved Right-to-Left? `plus` First (Bad!)

Same goal, but imagine we solve `plus` *before* `times`: Subgoals:

$$times(0, s(s(0)), P) \quad \text{and} \quad \underbrace{plus(P, s(s(0)), Q)}_{\text{solve first}}$$

Problem: P is **unknown**! We haven't solved `times` yet.

What If We Solved Right-to-Left? `plus` First (Bad!)

Same goal, but imagine we solve `plus` *before* `times`: Subgoals:

$$\mathit{times}(0, s(s(0)), P) \quad \text{and} \quad \underbrace{\mathit{plus}(P, s(s(0)), Q)}_{\text{solve first}}$$

Problem: P is **unknown**! We haven't solved `times` yet.

Both PZ and PS match $\mathit{plus}(P, s(s(0)), Q)$:

- PS is listed first, so Prolog tries it: $P = s(P_1)$, $Q = s(Q_1)$
- New subgoal: $\mathit{plus}(P_1, s(s(0)), Q_1)$ — *same situation!*

What If We Solved Right-to-Left? `plus` First (Bad!)

Same goal, but imagine we solve `plus` *before* `times`: Subgoals:

$$\mathit{times}(0, s(s(0)), P) \quad \text{and} \quad \underbrace{\mathit{plus}(P, s(s(0)), Q)}_{\text{solve first}}$$

Problem: P is **unknown**! We haven't solved `times` yet.

Both PZ and PS match $\mathit{plus}(P, s(s(0)), Q)$:

- PS is listed first, so Prolog tries it: $P = s(P_1),\ Q = s(Q_1)$
- New subgoal: $\mathit{plus}(P_1, s(s(0)), Q_1)$ — *same situation!*
- PS applies again: $P_1 = s(P_2),\ Q_1 = s(Q_2)$
- New subgoal: $\mathit{plus}(P_2, s(s(0)), Q_2)$ — *same situation!*

What If We Solved Right-to-Left? `plus` First (Bad!)

Same goal, but imagine we solve `plus` *before* `times`: Subgoals:

$$\mathit{times}(0, s(s(0)), P) \quad \text{and} \quad \underbrace{\mathit{plus}(P, s(s(0)), Q)}_{\text{solve first}}$$

Problem: P is **unknown**! We haven't solved `times` yet.

Both PZ and PS match $\mathit{plus}(P, s(s(0)), Q)$:

- PS is listed first, so Prolog tries it: $P = s(P_1),\ Q = s(Q_1)$
- New subgoal: $\mathit{plus}(P_1, s(s(0)), Q_1)$ — *same situation!*
- PS applies again: $P_1 = s(P_2),\ Q_1 = s(Q_2)$
- New subgoal: $\mathit{plus}(P_2, s(s(0)), Q_2)$ — *same situation!*
- ...

Right-to-Left: Infinite Loop!

Prolog keeps applying PS forever:

$$
\begin{aligned}
& plus(P, s(s(0)), Q) \\
\rightarrow\ & plus(P_1, s(s(0)), Q_1) \\
\rightarrow\ & plus(P_2, s(s(0)), Q_2) \\
\rightarrow\ & plus(P_3, s(s(0)), Q_3) \\
\rightarrow\ & \cdots
\end{aligned}
$$

Right-to-Left: Infinite Loop!

Prolog keeps applying PS forever:

$$
\begin{aligned}
& plus(P, s(s(0)), Q) \\
\rightarrow\ & plus(P_1, s(s(0)), Q_1) \\
\rightarrow\ & plus(P_2, s(s(0)), Q_2) \\
\rightarrow\ & plus(P_3, s(s(0)), Q_3) \\
\rightarrow\ & \cdots
\end{aligned}
$$

Nontermination! Without knowing P, there are infinitely many ways to decompose the first argument via PS.

Right-to-Left: Infinite Loop!

Prolog keeps applying PS forever:

$$\begin{aligned} & plus(P, s(s(0)), Q) \\ \to\ & plus(P_1, s(s(0)), Q_1) \\ \to\ & plus(P_2, s(s(0)), Q_2) \\ \to\ & plus(P_3, s(s(0)), Q_3) \\ \to\ & \cdots \end{aligned}$$

Nontermination! Without knowing P, there are infinitely many ways to decompose the first argument via PS.

Lesson

Two logically equivalent orderings of subgoals can mean the difference between **terminating in 2 steps** and **running forever**. This is why Prolog's left-to-right evaluation order is part of the "programming" in logic programming.

A Taste of Type Checking in Prolog

Proof Checking = Pattern Matching + Unification

Remember bidirectional type checking?

Proof Checking = Pattern Matching + Unification

Remember bidirectional type checking?

- `check(Ctx, Term, Type)` — all inputs given, succeed or fail
- `synth(Ctx, Term, Type)` — synthesize the type

Proof Checking = Pattern Matching + Unification

Remember bidirectional type checking?

- `check(Ctx, Term, Type)` — all inputs given, succeed or fail
- `synth(Ctx, Term, Type)` — synthesize the type

```
check(G, pair(M,N), and(A,B)) :-
    check(G, M, A),
    check(G, N, B).

check(G, fun(X,M), imp(A,B)) :-
    check([tp(X,A)|G], M, B).

check(G, R, A) :- synth(G, R, A).
```

Synthesis Rules

```
synth(O, fst(M), A) :-
    synth(O, M, and(A,B)).

synth(O, snd(M), B) :-
    synth(O, M, and(A,B)).

synth(O, app(R,M), B) :-
    synth(O, R, imp(A,B)),
    check(O, M, A).

synth([tp(X,A)|O], X, A).
synth([tp(Y,B)|O], X, A) :-
    Y \= X, synth(O, X, A).
```

It Actually Works!

```
?- check([], fun(x,x), imp(a,a)).
true

?- check([], fun(x,fun(y,x)),
          imp(a,imp(b,a))).
true

?- check([], fun(x,fun(y,y)),
          imp(a,imp(b,a))).
no
```

It Actually Works!

```
?- check([], fun(x,x), imp(a,a)).
true

?- check([], fun(x,fun(y,x)),
          imp(a,imp(b,a))).
true

?- check([], fun(x,fun(y,y)),
          imp(a,imp(b,a))).
no
```

That's a type checker in ∼15 lines of Prolog!

Bonus: Type *Inference* for Free!

Leave the type as a variable and let unification figure it out:

```
?- check([], fun(x,x), A).
A = imp(B, B)

?- check([], fun(f,fun(x,fun(y,
       app(f,pair(x,y))))), A).
A = imp(imp(and(B,C),D),
        imp(B, imp(C, D)))
```

Bonus: Type *Inference* for Free!

Leave the type as a variable and let unification figure it out:

```
?- check([], fun(x,x), A).
A = imp(B, B)

?- check([], fun(f,fun(x,fun(y,
       app(f,pair(x,y))))), A).
A = imp(imp(and(B,C),D),
        imp(B, imp(C, D)))
```

Prolog finds the **most general type**!
Unification accumulates constraints and solves them automatically.

The Big Picture

Two Paradigms, One Logic

Proofs-as-Programs	**Formulas-as-Programs**
Curry–Howard	Logic Programming
A proof *is* a program	Proof *search* is computation
Functional programming	Prolog
Type checking	Goal-directed search

Two Paradigms, One Logic

Proofs-as-Programs	**Formulas-as-Programs**
Curry–Howard	Logic Programming
A proof *is* a program	Proof *search* is computation
Functional programming	Prolog
Type checking	Goal-directed search

Both rest on constructive logic, but exploit *different* aspects of it!

Prolog: The Good, The Bad, The Ugly

The Good:

- Relations can run in multiple directions
- Built-in search and backtracking
- Unification is incredibly powerful
- Type checkers / proof checkers in a few lines

Prolog: The Good, The Bad, The Ugly

The Good:

- Relations can run in multiple directions
- Built-in search and backtracking
- Unification is incredibly powerful
- Type checkers / proof checkers in a few lines

The Bad / Ugly:

- no types (garbage in, garbage out)
- Unsound unification (missing occurs check)
- Clause & subgoal order affect termination
- "Declarative programming" is a bit of a myth

Takeaways

1. Logic programming = proof search with a fixed strategy

Takeaways

1. Logic programming = proof search with a fixed strategy
2. Prolog executes inference rules top-down, left-to-right

Takeaways

1. Logic programming = proof search with a fixed strategy
2. Prolog executes inference rules top-down, left-to-right
3. Unification + backtracking + logic variables = a surprisingly expressive computation model

Takeaways

1. Logic programming = proof search with a fixed strategy
2. Prolog executes inference rules top-down, left-to-right
3. Unification + backtracking + logic variables = a surprisingly expressive computation model
4. The same logical specification can serve as both a program *and* a correctness argument

Takeaways

1. Logic programming = proof search with a fixed strategy
2. Prolog executes inference rules top-down, left-to-right
3. Unification + backtracking + logic variables = a surprisingly expressive computation model
4. The same logical specification can serve as both a program *and* a correctness argument

Connection to This Course

We've seen the same inference rules as types (Curry–Howard), as substructural resources (linear logic), and now as executable programs (Prolog). Different readings of the same logical foundation!