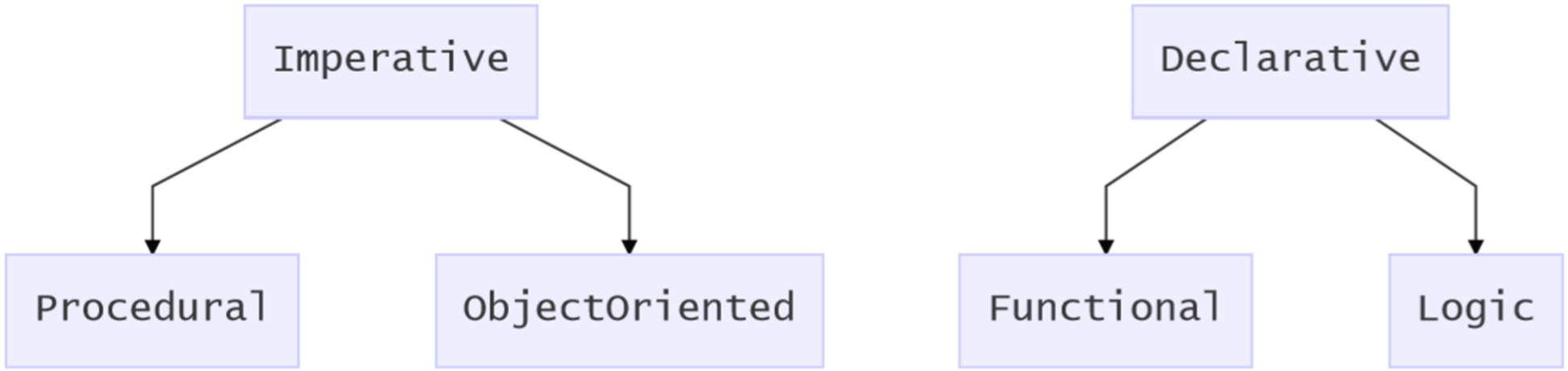

Logic
Programming
Paradigm

Logic Programming's Mathematical Framework

Just as functional programming paradigm is patterned from the formalisms of lambda calculus, logic programming is patterned from predicate calculus.


Imperative
Declarative
Procedural
ObjectOriented
Functional
Logic

Logic Programming

In terms of application this paradigm is more closely related to knowledge base programming languages like SQL.

Logic Programming

For this discussion we will use the programming language Prolog as the representative of logic paradigm.

Other logic programming families are answer set programming, ABSYS and Datalog.

Facts, Rules, and Queries

There are three basic constructs in Prolog, facts, rules and queries. A knowledge base is a collection of facts and rules in the same way a library is a collection of function definitions

Knowledge Base Example

```
firetype(charmander).
firetype(charizard).
watertype(squirtle).
flyingtype(charizard).
```

```
firetype(charmander).
firetype(charizard).
watertype(squirtle).
flyingtype(charizard).
```

Prolog query

```
?- firetype(charmander)
yes
```

Facts

The fact firetype(charmander) is basically a representation of the proposition, $firetype("charmander")$ where $firetype$ is a predicate and $charmander$ is a value assigned to the predicate.

Facts

Any fact that can be found on the knowledge base are basically true propositions and any proposition that is not in the knowledge base (and cannot be inferred from the knowledge base) are false.

```
firetype(charmander).
firetype(charizard).
watertype(squirtle).
flyingtype(charizard).
```

Prolog query

```
?- firetype(squirtle)
no
```

```
firetype(charmander).
firetype(charizard).
watertype(squirtle).
flyingtype(charizard).
```

Prolog query

```
?- grasstype(pigeot)
no
```

Rules

Aside from facts, you can also define rules in your knowledge base.

```
firetype(charmander).
firetype(charizard).
watertype(squirtle).
flyingtype(charizard).

resistanttofire(squirtle) :-
    watertype(squirtle).
```

Rules

A rule is basically an implication statement. A rule written as c :- h is equivalent to the implication statement $h \rightarrow c$.

```
firetype(charmander).
firetype(charizard).
watertype(squirtle).
flyingtype(charizard).

resistanttofire(squirtle)
:- watertype(squirtle).
```

Prolog query

```
?- resistanttofire(squirtle)
yes
```

Rules

Although resistanttofire(squirtle) is not written as a fact it can be inferred from the rule

```
resistanttofire(squirtle)
:- watertype(squirtle)
```

and the fact `watertype(squirtle).`

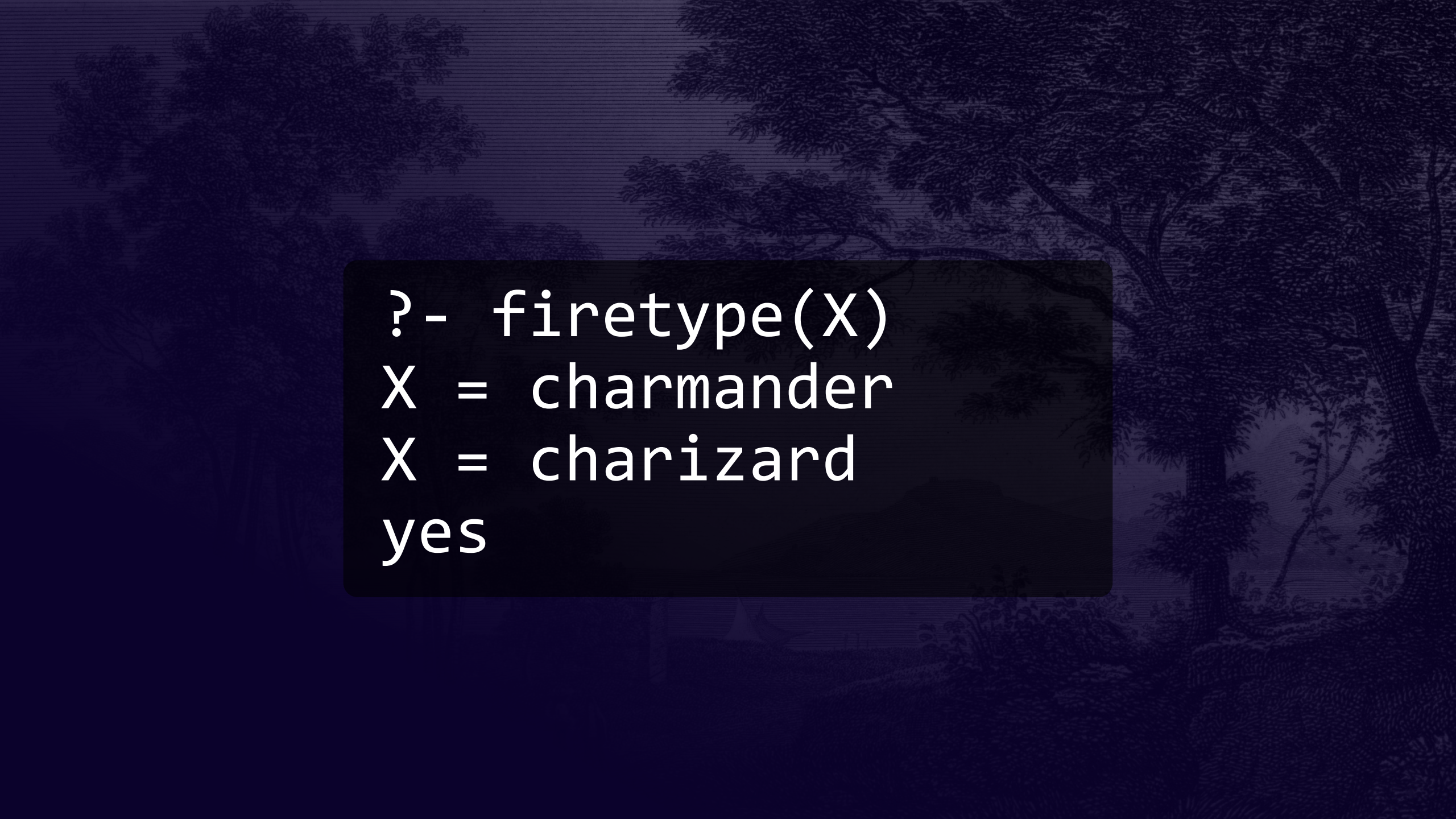
$$watertype(squirtle) \rightarrow resistanttofire(squirtle)$$
$$watertype(squirtle)$$
$$\therefore \quad resistanttofire(squirtle)$$

Variables

Another important thing about Prolog constructs is that you can write them with variables. For example, writing the query:

```
?- firetype(X)
```

Basically asks the question, which values substituted to X in the predicate `firetype(X)` will yield a true?


?- firetype(X)
X = charmander
X = charizard
yes

Variables

Variables inside facts and rules allows the creation of richer knowledge bases.

Instead of the rule

```
resistanttofire(squirtle)
:- watertype(squirtle).
```

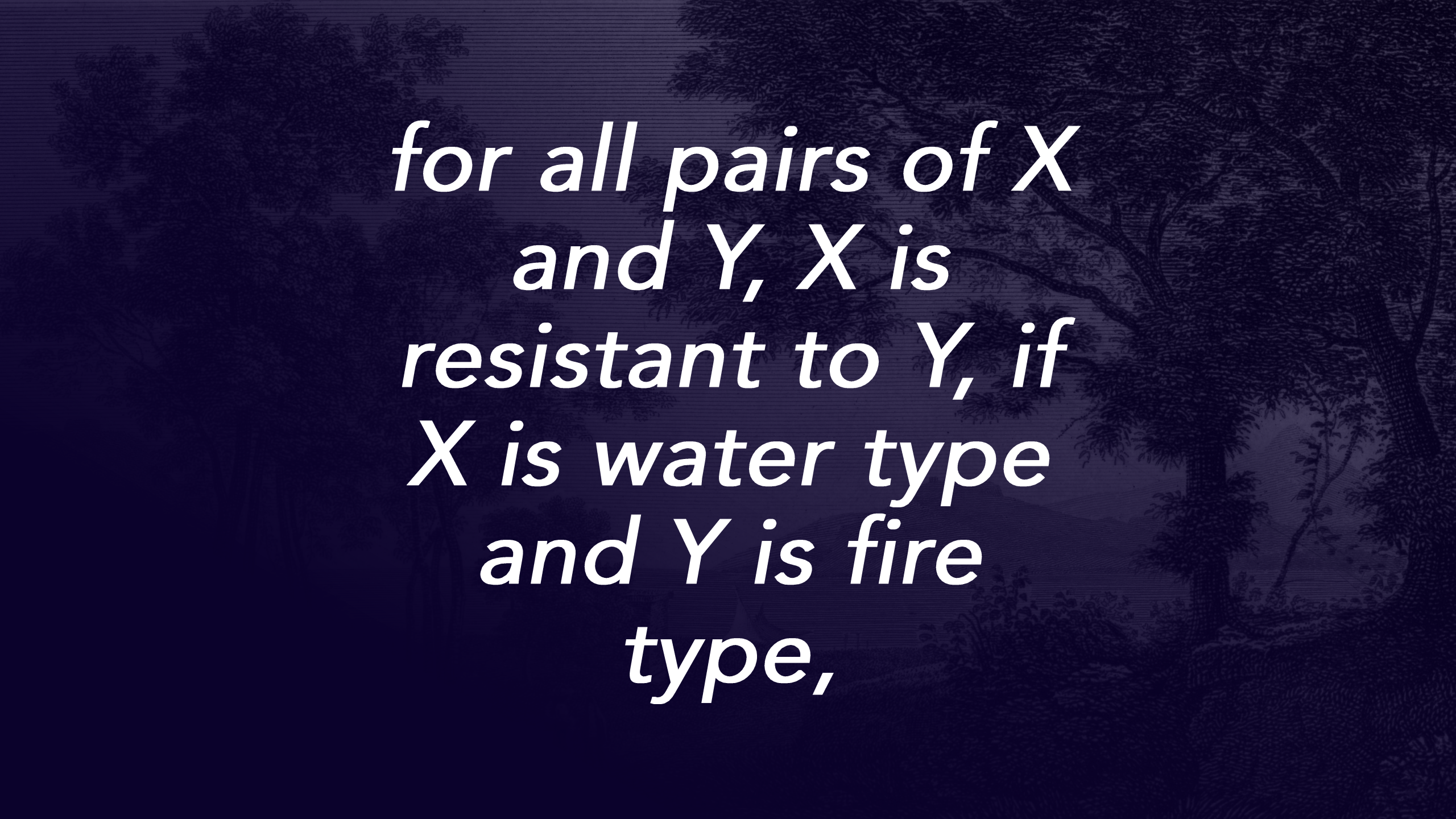
we can write a more general rule using variables:

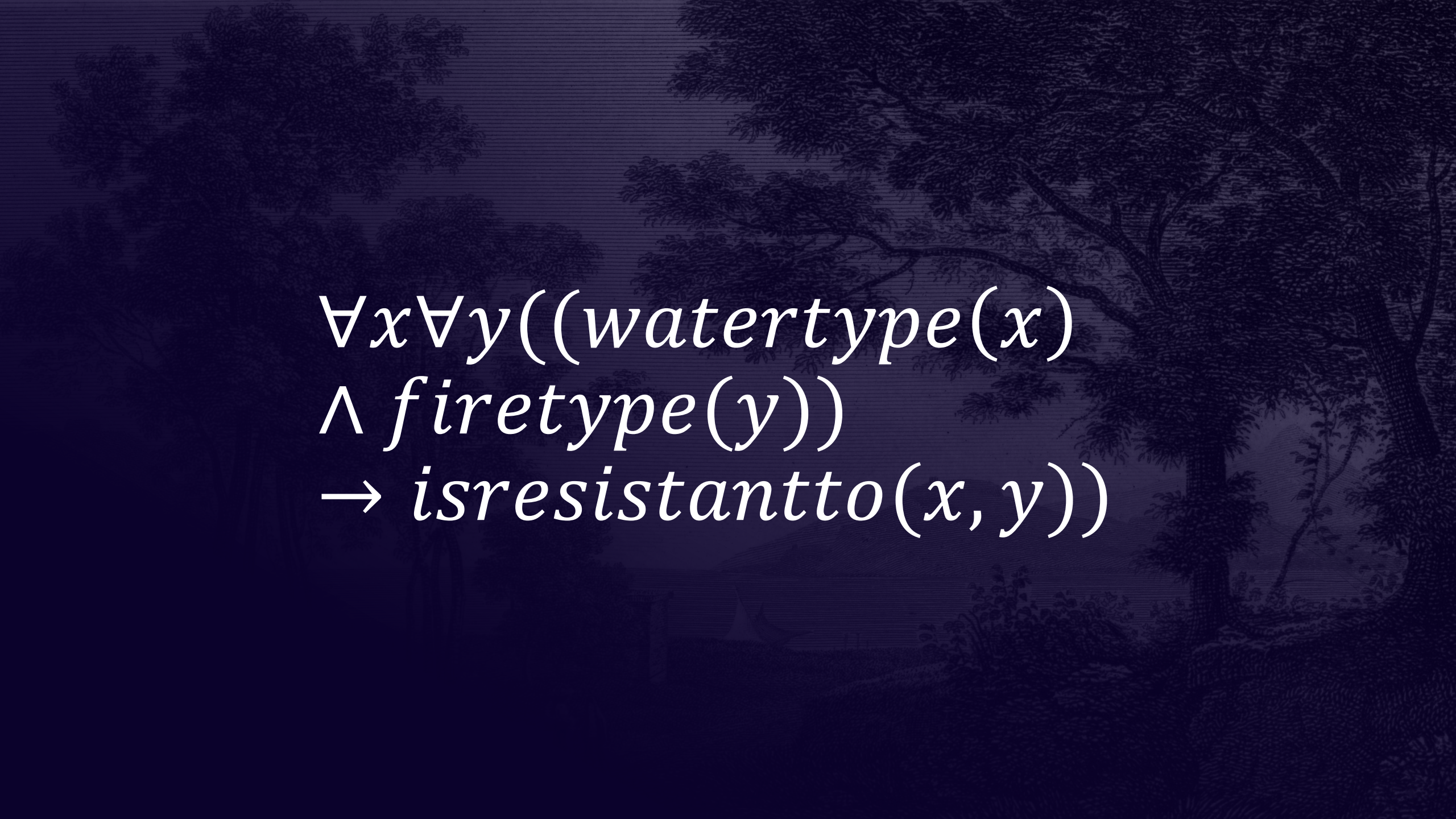
```
firetype(charmander).
firetype(charizard).
watertype(squirtle).
flyingtype(charizard).

isresistantto(X,Y) :-
    watertype(X),firetype(Y).
isresistantto(X,Y) :-
    watertype(X),watertype(Y).
```

Variables

This introduces a more complicated rule `isresistanto(X,Y) :- watertype(X),firetype(Y).` This rule's premise is a conjunction of predicates `watertype(X)` and `firetype(Y)`.


for all pairs of X and Y, X is resistant to Y, if X is water type and Y is fire type,


$\forall x \forall y((watertype(x) \wedge firetype(y)) \rightarrow isresistantto(x, y))$

```
?- isresistantto(squirtle,charmander)
yes
?- isresistantto(squirtle,charizard)
yes
?- isresistantto(squirtle,squirtle)
yes
```

Prolog Terms

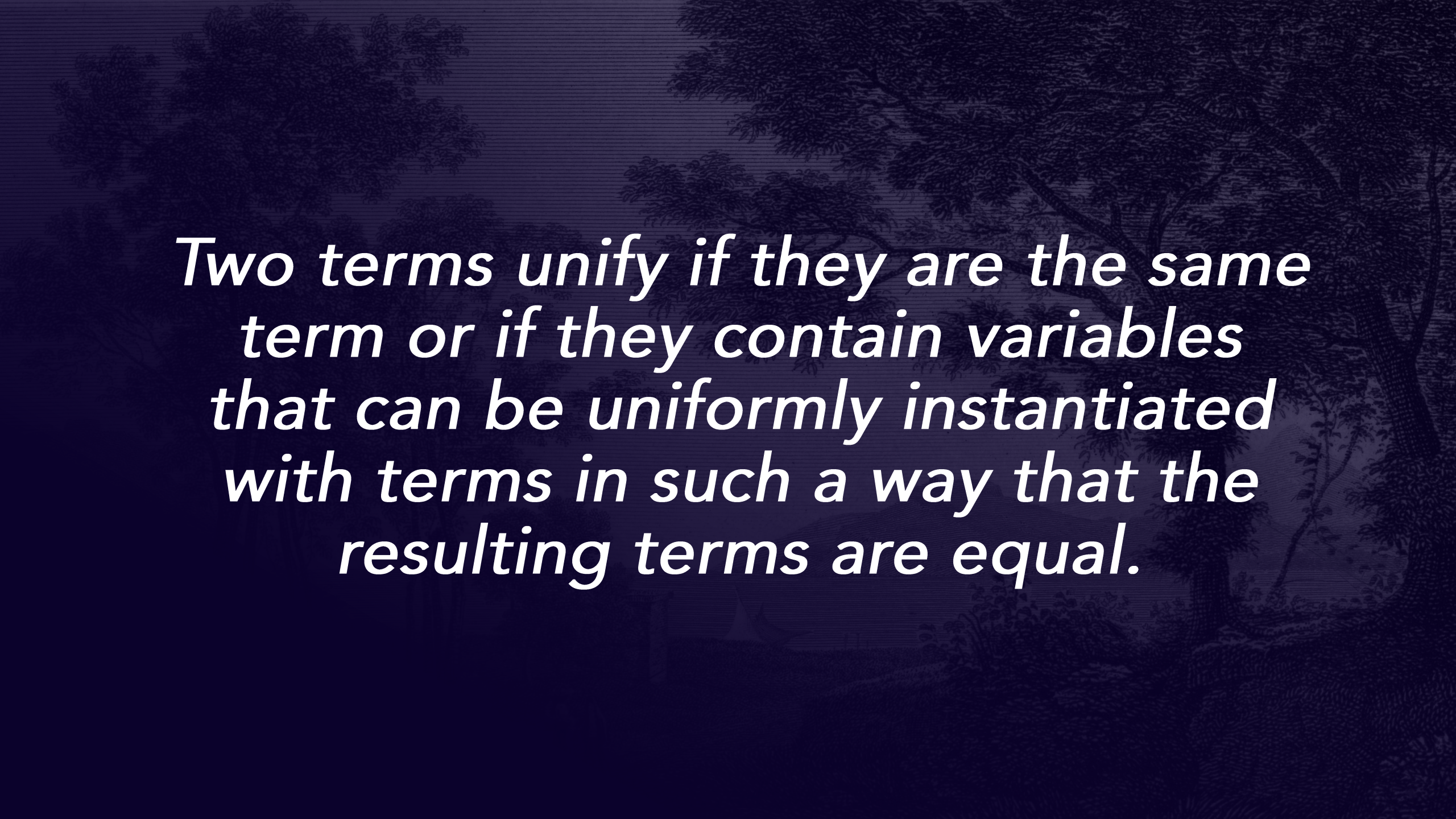
1. Constants. These can either be atoms (such as `squirtle`) or numbers (such as `24`).
2. Variables. (Such as `X`, `Z3`, `_4310`, and `List`.)
3. Complex terms. These have the form: `functor(term_1,...,term_n)`

Unification

The way prolog is able to respond to complex queries such as:

```
?- firetype(X)
X = charmander
X = charizard
```

is through the use of the concept known as **unification**.


Two terms unify if they are the same term or if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal.

Prolog Unification

Two terms a and b unify if and only if

1. a and b are constants and they are the same number or atom

Prolog Unification

Two terms a and b unify if and only if

2. a is a variable and b is any type of term (in this case a is instantiated to b) or b is a variable and a is any type of term (in this case b is instantiated to a).

Prolog Unification

Two terms a and b unify if and only if

3. a and b are complex terms and:
   - They have the same functors and the same number of arguments
   - all their corresponding arguments unify
   - the variable instatiations are compatible (you cannot instantiate x to some constant a when unifying a pair and instantiate x to another constant b when unifying another pair of arguments)

Programming with Unification

Unification is crucial with how one can write interesting logic programs.

By creating knowledge bases that take advantage of unification, you can generalize structures based on the facts and rules of its characteristics.

```
vertical(line(point(X,Y),point(X,Z))).
horizontal(line(point(X,Y),point(Z,Y))).
```

Proof Search

Prolog is able to unify any complicated query through the process of proof search.

This process is basically the core of prolog interpretation.

```
f(a).
f(b).

g(a).
g(b).

h(b).

k(X) :- f(X), g(X),
h(X).
```

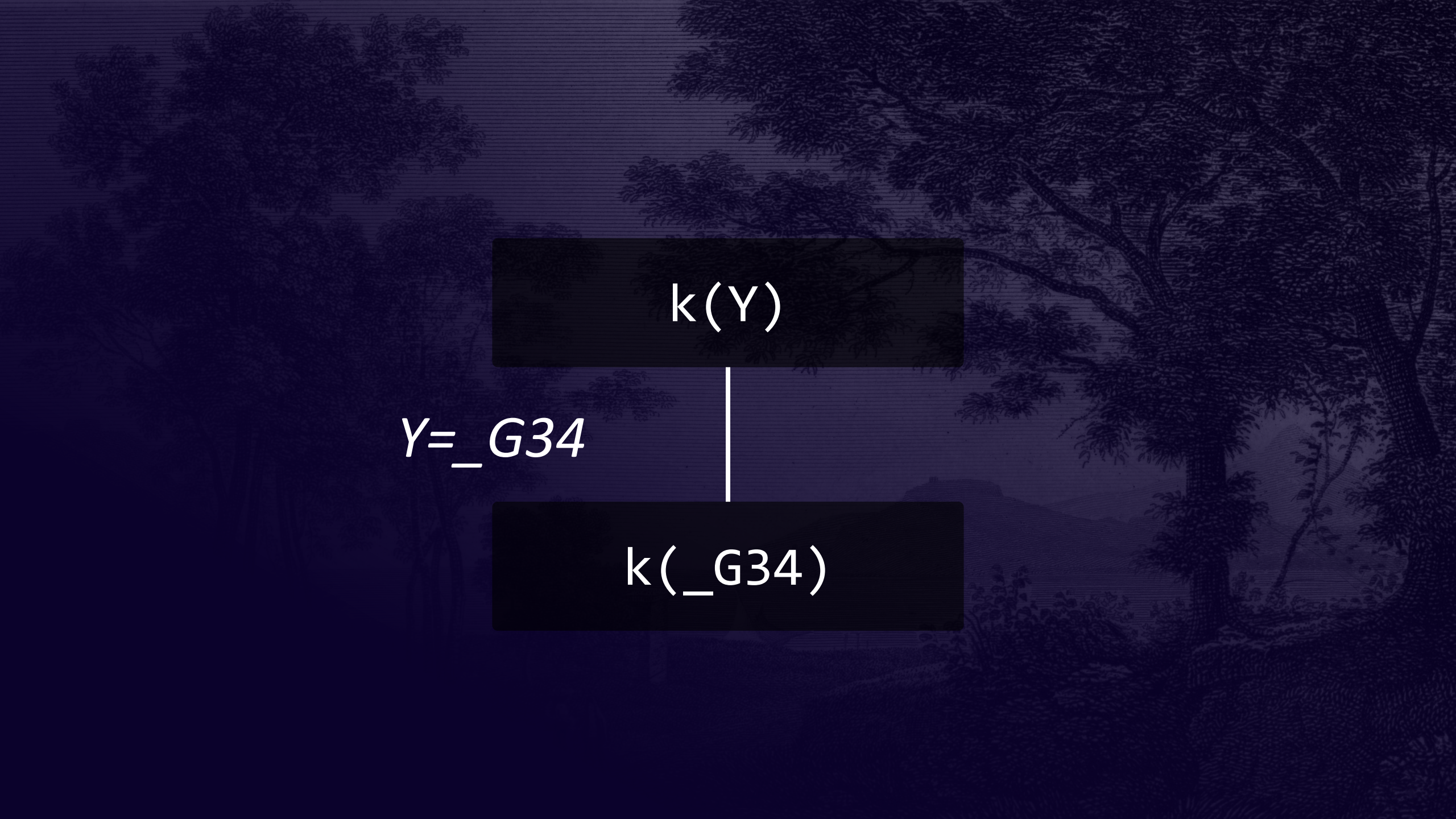
```
?- k(Y)
```

Proof Search

This gives prolog a **goal**, unifying k(Y) with all possible facts or inferable facts in the knowledge base.

Proof Search

Whenever prolog unifies queries containing variables, it creates a new internal variable to alias it with Y.


k(Y)
Y=_G34
k(_G34)

Proof Search

Prolog goes through the whole knowledge base from top to bottom and from left to right, attempting to unify the current goal, `k(_G34)` a fact or a head of a rule.

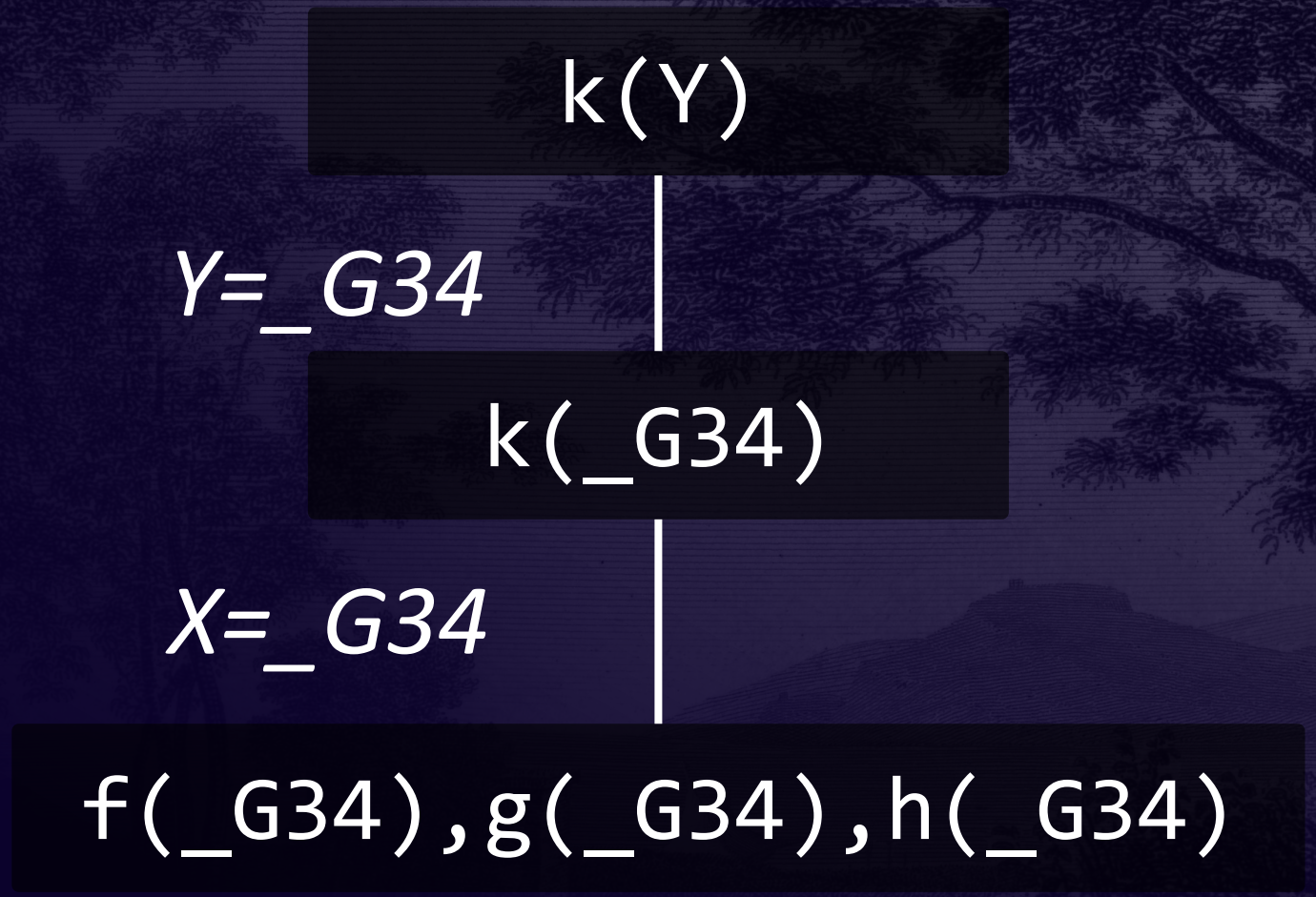
Proof Search

In this case since there are no facts `k(_G34)` can unify with, it unifies with the head (or the conclusion) of a rule, `k(X) :- f(X),g(X),h(X).`

Proof Search

Since this is a rule, we can prove that `k(_G34)` is true by proving the premises, `f(_G34), g(_G34), h(_G34).`

This gives prolog three new goals, proving the conjunction of the predicates, `f(_G34),g(_G34),h(_G34)`

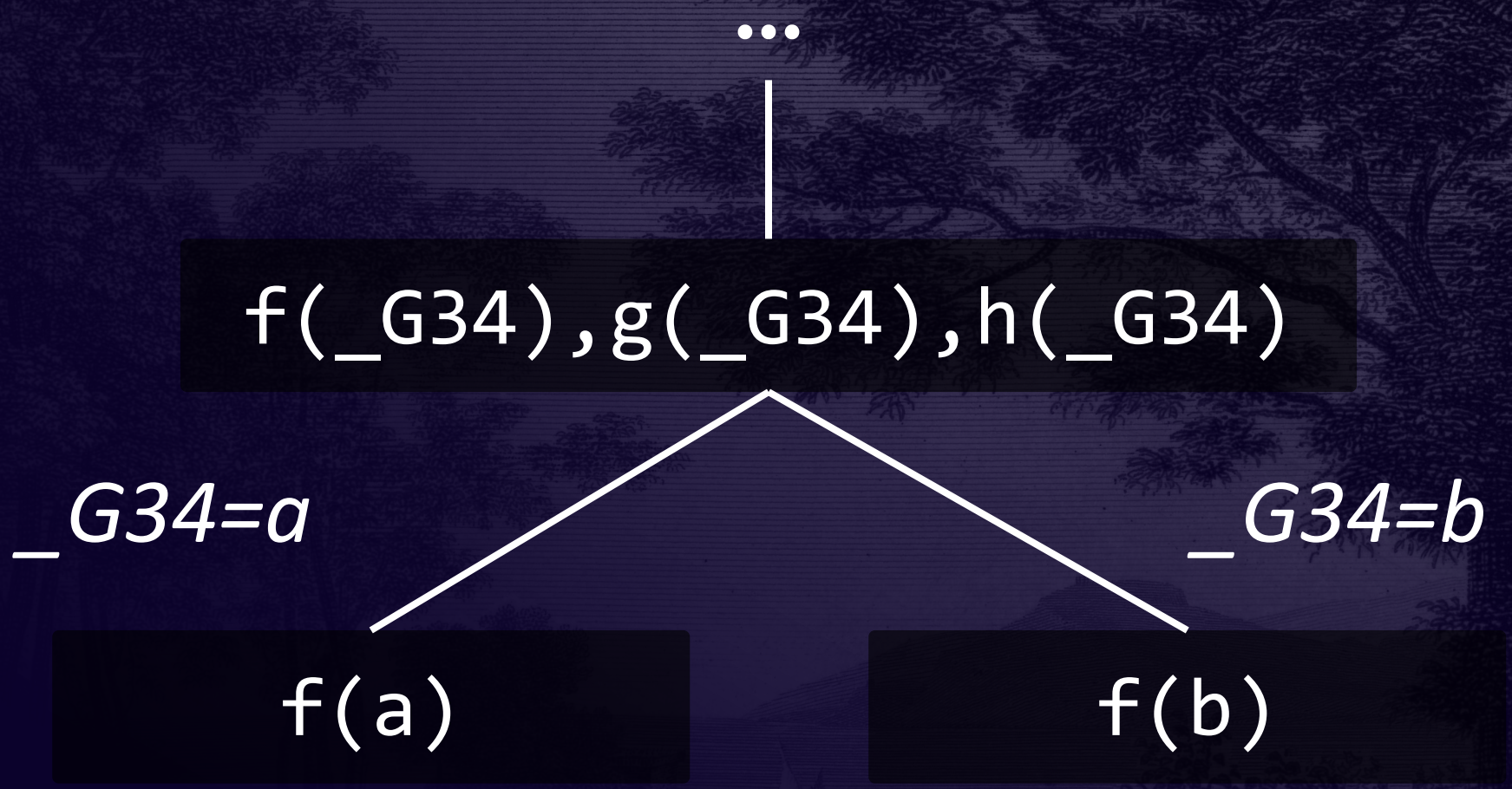

k(Y)
Y=_G34
k(_G34)
X=_G34
f(_G34),g(_G34),h(_G34)

Proof Search

Starting with the goal f(_G34), prolog searches the whole knowledge base again, attempting to unify f(_G34) with facts or rule heads.

Proof Search

Since the knowledge base contains both facts f(a) and f(b) there will now be two paths in this search, instantiating _G34 to a and instantiating _G34 to b.

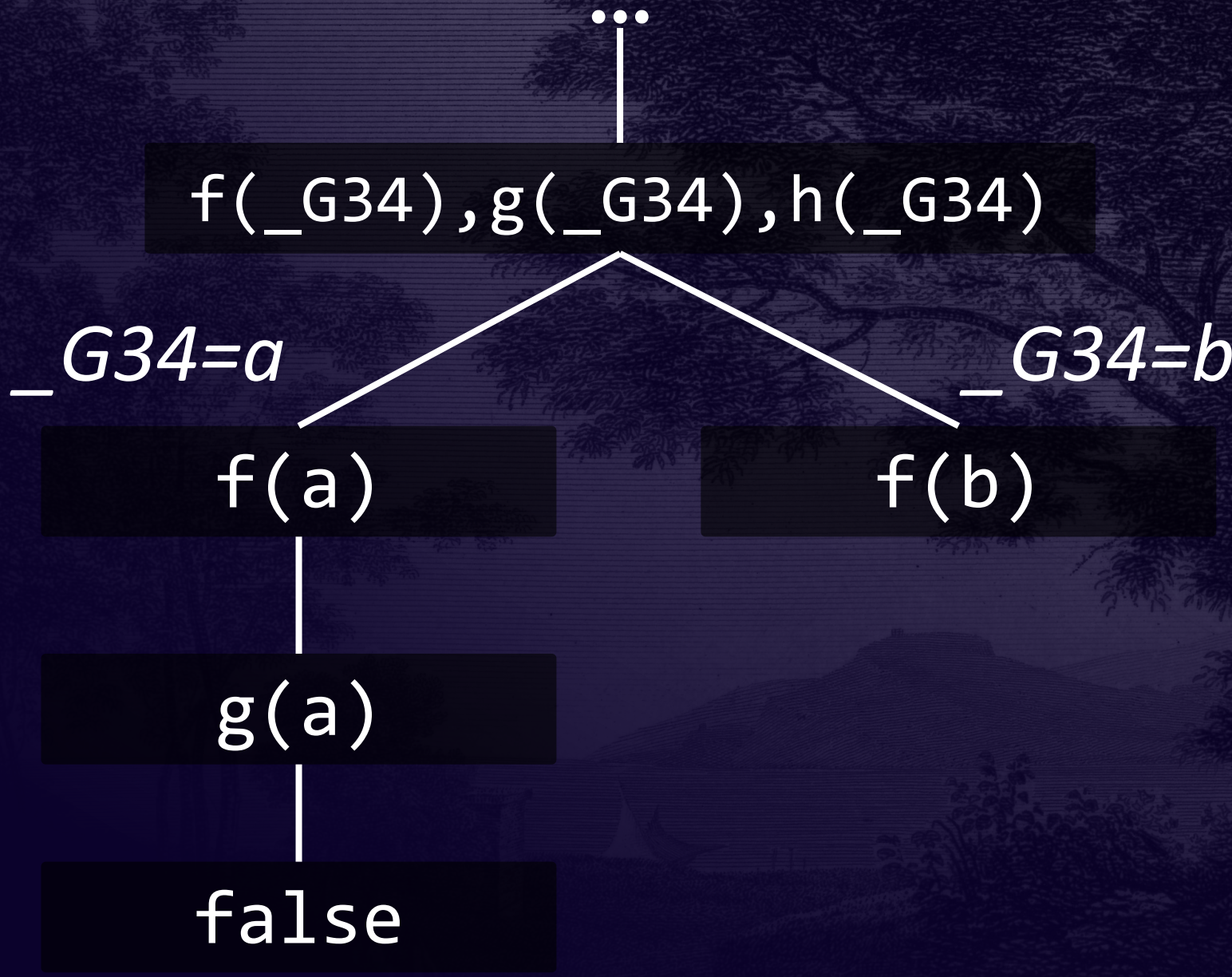

...
f(_G34),g(_G34),h(_G34)
_G34=a
_G34=b
f(a)
f(b)

Proof Search (Path 1 (_G34 = a))

From this instantiation, prolog is now left with the new goals g(a), h(a).

Starting with g(a), prolog searches the knowledge base again and unifies g(a) to g(a), reducing the goal toh(a).

Since h(a) cannot be unified with any fact or rule head, this path ends up unprovable.

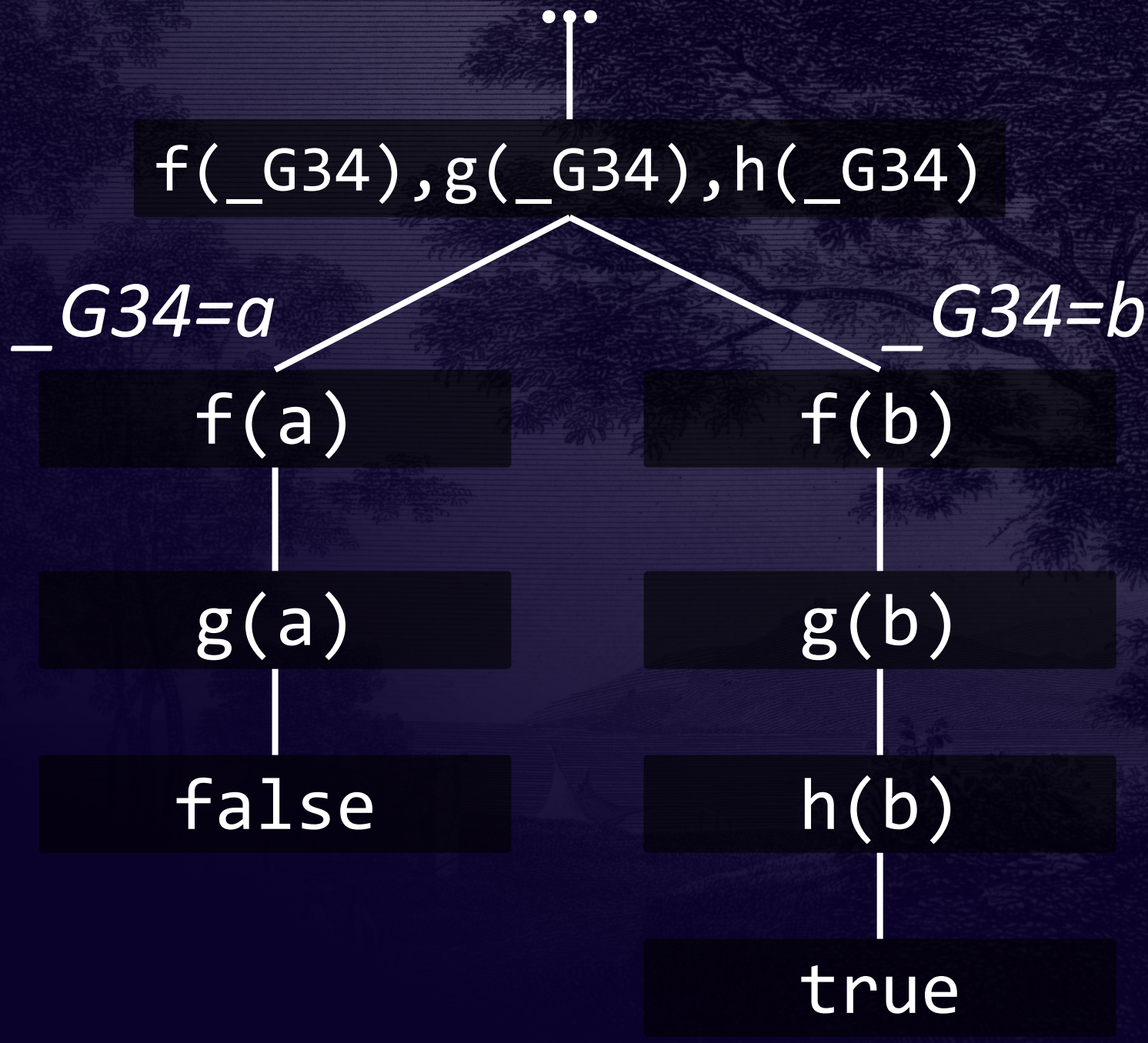

...
f(_G34),g(_G34),h(_G34)
_G34=a
_G34=b
f(a)
f(b)
g(a)
false

Proof Search (Path 2 (_G34 = b))

From this instantiation, prolog now has the new goals g(b), h(b).

Starting with g(b), prolog searches the knowledge base from the top again and finds the unification g(b) to g(b), reducing the goal to h(b).

It then finds the unification, h(b) thus completing the goal and proving the query for the instantiation_G34 = b.


f(_G34),g(_G34),h(_G34)
_G34=a
_G34=b
f(a)
f(b)
g(a)
g(b)
false
h(b)
true

Proof Search Response

By completing all possible paths prolog responds to the query:

```
Y = b
yes
```

Recursive Definitions

Similar to functional programming, logic programming represents repetition using recursion.

```
is_digesting(X,Y)  :-  just_ate(X,Y).
is_digesting(X,Y)  :-
    just_ate(X,Z),
    is_digesting(Z,Y).

just_ate(mosquito,blood(john)).
just_ate(frog,mosquito).
just_ate(stork,frog).
```

Recursive Definitions

You'll notice that the rule is_digesting is special since one of its goals is itself. You can interpret this rule as:

X is digesting Y if X just ate Y or X ate some Z that is digesting Y.

Advantages and Disadvantages of LP

Logic programming shares a lot of similarities with functional programming.

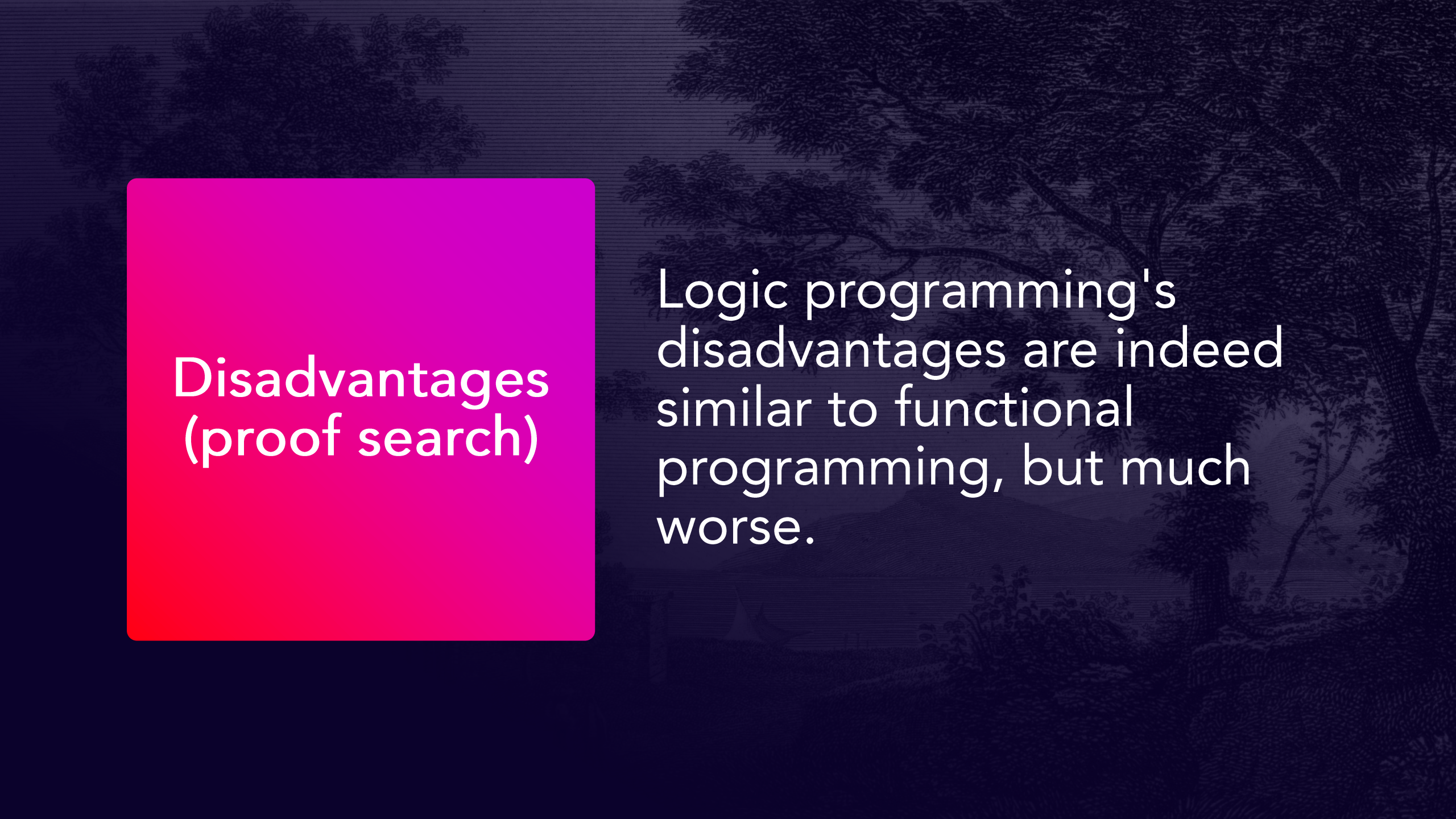
Both paradigms offer a safer and more consistent framework since they are both patterned form mathematical formalisms.

Advantages and Disadvantages of LP

Being non-imperative gives them an edge of automatically being immune to the perils of state and at the same time being prone to the perils of its absence.

Advantages (elegant knowledge representation)

The straight forward and elegant way of listing facts and rules makes it suitable for representing complex information that can be usually found in the domains of AI, NLP, and expert systems.


Disadvantages
(proof search)
Logic programming's disadvantages are indeed similar to functional programming, but much worse.

Disadvantages (proof search)

The obvious inefficiency due to the absence of state is much more evident in logic programming because of the thorough approach of backtracking in proof search.

Disadvantages (strangeness)

The strangeness of logic programming as compared to the imperative way of thinking is also much worse than functional programming.

Disadvantages (strangeness)

Because of these logic programming is relegated to solving niche problems in various domains.