Week 3

Magnus Madsen

Friday 14th March, 2025 at 15:00

Week 3: Outline

Tuesday

Lecture (45min)

- Introduction to Prolog
- Introduction to Unification

Exercises (45min)

- Work on the assignment alone or together in small groups.

Thursday

Lecture (45min)

- A Larger Example: The Wolf, Goat, and Cabbage Problem

Exercises (45min)

- Work on the assignment alone or together in small groups.

Quote of the Day

"As Will Rogers would have said, *there is no such thing as a free variable.*"

— Alan Perlis

Pull Requests are Welcome

You can improve the course material!

- Exercises are in `src/weekX.md`
- Slides are in `slides/weekX.tex`

PRs can be submitted on GitHub:

`https://github.com/magnus-madsen/advprog/`

Introduction to Prolog

Prolog

Prolog: *Pro*gramming *Lo*gic

From Datalog to Prolog: A Shift in Perspective

Datalog is based on *bottom-up* tabling:

- Datalog has a unique minimal model.
- Datalog offers strong guarantees about termination.
- Limited expressive power.

From Datalog to Prolog: A Shift in Perspective

Datalog is based on *bottom-up* tabling:

- Datalog has a unique minimal model.
- Datalog offers strong guarantees about termination.
- Limited expressive power.

Prolog is based on *top-down* search:

- Prolog is goal-driven.
- Prolog programs are imperative: the order of evaluation matters.
- Prolog is Turing-complete – and hence programs may fail to terminate.

From Datalog to Prolog: A Shift in Perspective

Datalog is based on *bottom-up* tabling:

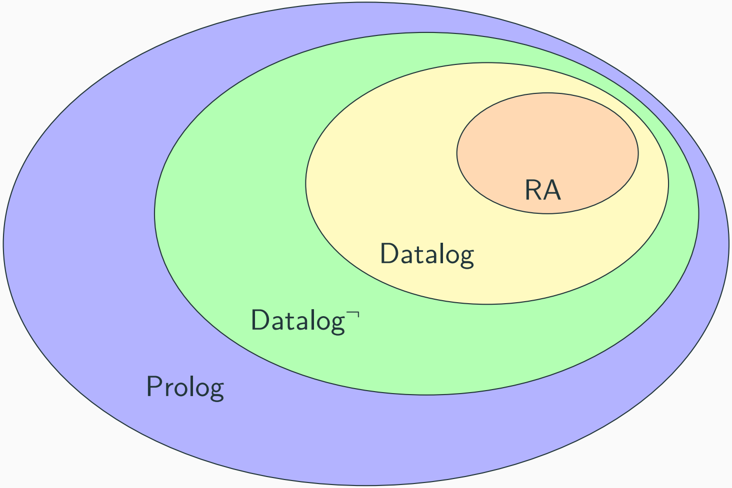
- Datalog has a unique minimal model.
- Datalog offers strong guarantees about termination.
- Limited expressive power.

Prolog is based on *top-down* search:

- Prolog is goal-driven.
- Prolog programs are imperative: the order of evaluation matters.
- Prolog is Turing-complete – and hence programs may fail to terminate.

Prolog *is* a logic programming language, but it is like the C of logic languages.

Expressive Power



Syntax Update

In Flix the syntax of a rule is:

```
Path(x, z) :- Edge(x, y), Path(y, z).
```

In Prolog the syntax of the same rule is:

```
path(X, Z) :- edge(X, Y), path(Y, Z).
```

Syntax Update

In Flix the syntax of a rule is:

```
Path(x, z) :- Edge(x, y), Path(y, z).
```

In Prolog the syntax of the same rule is:

```
path(X, Z) :- edge(X, Y), path(Y, Z).
```

Moreover, in Prolog lowercase names are constants:

```
parent(emma, magnus).
parent(emma, daniela).
```

Syntax Update

In Flix the syntax of a rule is:

```
Path(x, z) :- Edge(x, y), Path(y, z).
```

In Prolog the syntax of the same rule is:

```
path(X, Z) :- edge(X, Y), path(Y, Z).
```

Moreover, in Prolog lowercase names are constants:

```
parent(emma, magnus).
parent(emma, daniela).
```

Warning: You will screw this up. Remember to check your casing.

Prolog is Query Driven

To write a Prolog program:

- We state the facts and rules of the domain.
- We ask a query (with zero or more free variables).

Prolog computes a ***single*** solution answering **yes** or **no**.

Here we can see Prolog's roots in expert systems and artificial intelligence.

A Few Simple Queries (1/3)

Given the facts:

```
parent(emma, magnus).
parent(emma, daniela).
```

A Few Simple Queries (1/3)

Given the facts:

```
parent(emma, magnus).
parent(emma, daniela).
```

We can ask:

```
?- parent(emma, magnus).
yes
```

A Few Simple Queries (1/3)

Given the facts:

```
parent(emma, magnus).
parent(emma, daniela).
```

We can ask:

```
?- parent(emma, magnus).
yes
```

And we can ask:

```
?- parent(emma, augustus).
no
```

A Few Simple Queries (2/3)

We can also ask:

```
?- parent(emma, X).
X = magnus ? ;
X = daniela ? ;
no
```

We use the semicolon **;** to prompt Prolog for additional solutions.

Prolog says **no** at the end because there are no more solutions!

A Few Simple Queries (3/3)

In Prolog `parent` is a relation, not a function, so we can also ask:

```
?- parent(X, magnus).
X = emma ? ;
no
```

Note that we are asking for “an input” that matches “an output”.

A Few Simple Queries (3/3)

In Prolog `parent` is a relation, not a function, so we can also ask:

```
?- parent(X, magnus).
X = emma ? ;
no
```

Note that we are asking for "an input" that matches "an output".

Question: What is the answer to the query `parent(X, Y)`?

Recursion in Prolog (1/2)

Prolog supports recursion:

```
edge(a, b).
edge(b, c).

path(X, Y) :- edge(X, Y).
path(X, Z) :- edge(X, Y), path(Y, Z).
```

Recursion in Prolog (1/2)

Prolog supports recursion:

```
edge(a, b).
edge(b, c).

path(X, Y) :- edge(X, Y).
path(X, Z) :- edge(X, Y), path(Y, Z).
```

And we can ask:

```
?- path(a, c).
yes

?- path(a, X).
X = b ? ;
X = c ? ;
no
```

Recursion in Prolog (2/2)

But we have to be careful. If we change the program to:

```
path(X, Z) :- path(Y, Z), edge(X, Y).
path(X, Y) :- edge(X, Y).
```

Recursion in Prolog (2/2)

But we have to be careful. If we change the program to:

```
path(X, Z) :- path(Y, Z), edge(X, Y).
path(X, Y) :- edge(X, Y).
```

And ask:

```
?- path(a, c).
```

Recursion in Prolog (2/2)

But we have to be careful. If we change the program to:

```
path(X, Z) :- path(Y, Z), edge(X, Y).
path(X, Y) :- edge(X, Y).
```

And ask:

```
?- path(a, c).
```

The program loops! But why?

Recursion in Prolog (2/2)

But we have to be careful. If we change the program to:

```
path(X, Z) :- path(Y, Z), edge(X, Y).
path(X, Y) :- edge(X, Y).
```

And ask:

```
?- path(a, c).
```

The program loops! But why?

Answer: Prolog uses top-to-bottom, left-to-right evaluation.

Constructors in Prolog (1/2)

Prolog supports constructors:

- Allows us to construct compound value (lists, trees, etc).
- Allows us to construct infinite values (oops.)

Like in functional programming, we use constructors to build data structures.

Constructors in Prolog (2/2)

We can write:

```
networth(person(steve, carrel),   80). % in millions USD
networth(person(steve, jobs),    250). % in millions USD
networth(person(jeff, bezos), 186000). % in millions USD
```

Constructors in Prolog (2/2)

We can write:

```
networth(person(steve, carrel),   80). % in millions USD
networth(person(steve, jobs),    250). % in millions USD
networth(person(jeff, bezos), 186000). % in millions USD
```

And then we can ask:

```
?- networth(X, 80).
X = person(steve,carrel)
```

Constructors in Prolog (2/2)

We can write:

```
networth(person(steve, carrel),   80). % in millions USD
networth(person(steve, jobs),    250). % in millions USD
networth(person(jeff, bezos), 186000). % in millions USD
```

And then we can ask:

```
?- networth(X, 80).
X = person(steve,carrel)
```

We can also ask:

```
?- networth(person(steve, X), Y).
X = carrel, Y =  80 ? ;
X = jobs,   Y = 250 ? ;
no
```

Lists in Prolog (1/2)

We can use constructors to encode lists:

```
len(nil, 0).
len(cons(_, Xs), R) :- len(Xs, N), R is N + 1.
```

Lists in Prolog (1/2)

We can use constructors to encode lists:

```
len(nil, 0).
len(cons(_, Xs), R) :- len(Xs, N), R is N + 1.
```

And then we can ask:

```
len(cons(1, cons(2, cons(3, nil))), N).
N = 3 ? ;
```

Lists in Prolog (1/2)

We can use constructors to encode lists:

```
len(nil, 0).
len(cons(_, Xs), R) :- len(Xs, N), R is N + 1.
```

And then we can ask:

```
len(cons(1, cons(2, cons(3, nil))), N).
N = 3 ? ;
```

Note: Please use the built-in lists: `[]` and `[Head|Tail]` in real programs.

Lists in Prolog (2/2)

We can also write:

```
appnd(nil, Ys, Ys).
appnd(cons(X, Xss), Ys, cons(X, Rs)) :- appnd(Xss, Ys, Rs).
```

Lists in Prolog (2/2)

We can also write:

```
appnd(nil, Ys, Ys).
appnd(cons(X, Xss), Ys, cons(X, Rs)) :- appnd(Xss, Ys, Rs).
```

And then we can ask:

```
?- appnd(cons(1, cons(2, nil)), cons(3, cons(4, nil)), R).
R = cons(1,cons(2,cons(3,cons(4,nil)))) ? ;
```

Arithmetic in Prolog

We wrote:

```
R is N + 1.
```

because wanted to force Prolog to evaluate `R` to `N + 1`.

Note that:

```
?- 1 + 2 = 3.   no
?- 3 is 1 + 2.  yes
```

Arithmetic in Prolog

We wrote:

```
R is N + 1.
```

because wanted to force Prolog to evaluate `R` to `N + 1`.

Note that:

```
?- 1 + 2 = 3.    no
?- 3 is 1 + 2.   yes
```

But

```
?- 1 + 2 is 3.   no
```

The `is` operator forces evaluation *on the right-hand side*.

Prolog is Dynamically Typed (1/2)

Prolog is dynamically typed, so if we write:

```
appnd(apple, cons(1, nil), R).
```

Prolog just tells us **no**. This may be okay.

Prolog is Dynamically Typed (1/2)

Prolog is dynamically typed, so if we write:

```
appnd(apple, cons(1, nil), R).
```

Prolog just tells us **no**. This may be okay.

But it can also get weird:

```
?- appnd(nil, apple, R).
R = apple ? ;
```

Here `R` is not a list! Oops!

Prolog is Dynamically Typed (2/2)

Like in Scheme, we can add dynamic type checks. We define the list "type":

```
is_list(nil).
is_list(cons(_, Xs)) :- is_list(Xs).
```

Prolog is Dynamically Typed (2/2)

Like in Scheme, we can add dynamic type checks. We define the list “type”:

```
is_list(nil).
is_list(cons(_, Xs)) :- is_list(Xs).
```

And then we update our implementation of `appnd`:

```
appnd(nil, Ys, Ys) :- is_list(Ys).
appnd(cons(X, Xss), Ys, cons(X, Rs)) :-
   is_list(Xss), is_list(Ys), is_list(Rs),  appnd(Xss, Ys, Rs).
```

Now our silly query `?- appnd(nil, apple, R)` returns **no**.

Prolog Grammar

The core Prolog grammar is almost equivalent to the Datalog grammar:

$$
\begin{aligned}
p \in \mathit{Program} &= c_1 \cdots c_n \\
c \in \mathit{Constraint} &= A_0 \Leftarrow B_1, \cdots, B_n. \\
A \in \mathit{HeadAtom} &= p(t_1, \cdots, t_n) \\
B \in \mathit{BodyAtom} &= p(t_1, \cdots, t_n) \mid \texttt{\textbackslash+}\ p(t_1, \cdots, t_n) \\
t \in \mathit{Term} &= k \mid x \mid c(t_1, \cdots, t_n)
\end{aligned}
$$

$$
\begin{aligned}
p \in \mathit{Predicates} &= \text{is a finite set of predicate symbols.} \\
x \in \mathit{Variables} &= \text{is a finite set of variable symbols.} \\
c \in \mathit{Constructors} &= \text{is a finite set of constructors.} \\
k \in \mathit{Constants} &= \text{is a finite set of constants.}
\end{aligned}
$$

Unification

Matching a Goal to a Rule

Assume we have a Prolog program with the facts and rules:

```
len([], 0).
len([_Head|Tail], R) :- len(Tail, N), R is N + 1.
```

Matching a Goal to a Rule

Assume we have a Prolog program with the facts and rules:

```
len([], 0).
len([_Head|Tail], R) :- len(Tail, N), R is N + 1.
```

And we ask Prolog the query:

```
? len([1, 2], X).
```

which is really:

```
? len([1 | [2 | []]], X).
```

Question: How does Prolog know which rule to evaluate?

Question: And what should be the values of the variables in the rule?

Matching a Goal to a Rule

Assume we have a Prolog program with the facts and rules:

```
len([], 0).
len([_Head|Tail], R) :- len(Tail, N), R is N + 1.
```

And we ask Prolog the query:

```
? len([1, 2], X).
```

which is really:

```
? len([1 | [2 | []]], X).
```

Question: How does Prolog know which rule to evaluate?

Question: And what should be the values of the variables in the rule?

Answer: Prolog uses ***unification*** to match the goal with the head atom.

What is Unification (1/2)

A ***substitution*** is a map from variables to terms.

What is Unification (1/2)

A ***substitution*** is a map from variables to terms.

For example, we can have the substitution:

$$s = \{X \mapsto 21, Y \mapsto [1, 2, 3]\}$$

What is Unification (1/2)

A ***substitution*** is a map from variables to terms.

For example, we can have the substitution:

$$s = \{X \mapsto 21, Y \mapsto [1, 2, 3]\}$$

We can apply a substitution to a term. For example,

if $t = node(X, X, Y)$ then

$$s(t) = node(21, 21, [1, 2, 3])$$

What is Unification? (1/2)

A ***substitution*** is a map from variables to terms.

What is Unification? (1/2)

A ***substitution*** is a map from variables to terms.

For example, we can have the substitution:

$$s = \{X \mapsto 21, Y \mapsto [1, 2, 3, W]\}$$

What is Unification? (1/2)

A ***substitution*** is a map from variables to terms.

For example, we can have the substitution:

$$s = \{X \mapsto 21, Y \mapsto [1, 2, 3, W]\}$$

We can apply a substitution to a term. For example:

If $t = node(X, X, Y)$ then

$$s(t) = node(21, 21, [1, 2, 3, W])$$

What is Unification? (2/2)

Unification: Given two terms t_1 and t_2, find a substitution s such that:

$$s(t_1) = s(t_2)$$

The substitution, when applied to both terms, makes them *syntactically* equal.

- We call such a substitution a *unifier*. It may not always exist.
- But if there is a unifier then there is a *most-general unifier* (MGU).

What is Unification? (2/2)

Unification: Given two terms t_1 and t_2, find a substitution s such that:

$$s(t_1) = s(t_2)$$

The substitution, when applied to both terms, makes them *syntactically* equal.

- We call such a substitution a *unifier*. It may not always exist.
- But if there is a unifier then there is a *most-general unifier* (MGU).

Sidenote: A cool idea is when $=$ is replaced by $\equiv$ leading to **E-unification**, i.e. unification modulo some equational theory.

A Unification Algorithm (in Flix) (1/6)

We define the language of terms:

```
enum Term {
    case Var(String),
    case Cst(Int32),
    case Pair(Term, Term)
}
```

A real term language, like the one used in Prolog, is richer.

However, the above term language is sufficient to illustrate the major points.

A Unification Algorithm (in Flix) (2/6)

We define a substitution as:

```
type alias Subst = Map[String, Term]
```

A Unification Algorithm (in Flix) (2/6)

We define a substitution as:

```
type alias Subst = Map[String, Term]
```

And we define a function that applies a substitution to a term:

```
def applySubst(s: Subst, t: Term): Term = match t {
    case Term.Var(x)          => Map.getWithDefault(x, t, s)
    case Term.Cst(c)          => Term.Cst(c)
    case Term.Pair(t1, t2)    =>
        Term.Pair(applySubst(s, t1), applySubst(s, t2))
}
```

A Unification Algorithm (in Flix) (3/6)

Given two substitutions s_1 and s_2, we define a function to compose them.

The new substitution should morally have the effect of applying s_1 to the term and then applying s_2 to that, i.e. we want:

$$\text{compose}(s_1, s_2)(t) = s_2(s_1(t))$$

Implementation: Left as an exercise for the reader.

Remark: Most bugs happen when implementing compose.

A Unification Algorithm (in Flix) (4/6)

We can now write a function to unify two terms:

```
def unify(t1: Term, t2: Term): Subst = match (t1, t2) {
    case (Term.Cst(c1), Term.Cst(c2)) if c1 == c2 => Map.empty()
    case (Term.Var(x), _) => Map.singleton(x, t2)
    case (_, Term.Var(y)) => Map.singleton(y, t1)
    case (Term.Pair(t11, t12), Term.Pair(t21, t22)) =>
        let s1 = unify(t11, t21);
        let s2 = unify(applySubst(s1, t12), applySubst(s1, t22));
        compose(s1, s2)
    case _ => unsafe throw new Exception("Unification failed")
}
```

A Unification Algorithm (in Flix) (4/6)

We can now write a function to unify two terms:

```
def unify(t1: Term, t2: Term): Subst = match (t1, t2) {
    case (Term.Cst(c1), Term.Cst(c2)) if c1 == c2 => Map.empty()
    case (Term.Var(x), _) => Map.singleton(x, t2)
    case (_, Term.Var(y)) => Map.singleton(y, t1)
    case (Term.Pair(t11, t12), Term.Pair(t21, t22)) =>
        let s1 = unify(t11, t21);
        let s2 = unify(applySubst(s1, t12), applySubst(s1, t22));
        compose(s1, s2)
    case _ => unsafe throw new Exception("Unification failed")
}
```

Question: Why apply `s1` to `t12` and `t22` before the recursive call to `unify`?

A Unification Algorithm (in Flix) (5/6)

Lets try it out:

```
unify(Cst(123), Var("x"))
    => Map#{x => Cst(123)}
```

A Unification Algorithm (in Flix) (5/6)

Lets try it out:

```
unify(Cst(123), Var("x"))
    => Map#{x => Cst(123)}
```

```
unify(Pair(Var("x"), Var("x")), Pair(Cst(123), Var("y")))
    => Map#{x => Cst(123), y => Cst(123)}
```

A Unification Algorithm (in Flix) (5/6)

Lets try it out:

```
unify(Cst(123), Var("x"))
    => Map#{x => Cst(123)}
```

```
unify(Pair(Var("x"), Var("x")), Pair(Cst(123), Var("y")))
    => Map#{x => Cst(123), y => Cst(123)}
```

What about:

```
unify(Var("x"), Pair(Cst(123), Var("x")))
    => Map#{x => Pair(Cst(123), Var(x))}
```

A Unification Algorithm (in Flix) (5/6)

Lets try it out:

```
unify(Cst(123), Var("x"))
    => Map#{x => Cst(123)}
```

```
unify(Pair(Var("x"), Var("x")), Pair(Cst(123), Var("y")))
    => Map#{x => Cst(123), y => Cst(123)}
```

What about:

```
unify(Var("x"), Pair(Cst(123), Var("x")))
    => Map#{x => Pair(Cst(123), Var(x))}
```

Ooos! This is incorrect. What's the problem?

A Unification Algorithm (in Flix) (6/6)

We modify the two cases for variables as follows:

```
case (Term.Var(x), t2) =>
    if (Set.memberOf(x, freeVars(t2)))
        unsafe throw new Exception("Occurs Check")
    else
        Map.singleton(x, t2)

// The other case is symmetric.
```

Here the `freeVars` function is defined in the obvious way.

The **occurs check** ensures that we do not construct substitutions where a variable occurs recursively within the term it is unified with.

A Unification Algorithm (in Flix) (6/6)

We modify the two cases for variables as follows:

```
case (Term.Var(x), t2) =>
    if (Set.memberOf(x, freeVars(t2)))
        unsafe throw new Exception("Occurs Check")
    else
        Map.singleton(x, t2)

// The other case is symmetric.
```

Here the `freeVars` function is defined in the obvious way.

The **occurs check** ensures that we do not construct substitutions where a variable occurs recursively within the term it is unified with.

Note: The occurs check is expensive, so some Prolog implementations omit it.

Matching a Goal to a Rule – Continued (1/2)

Recall that we had:

```
len([], 0).
len([_Head|Tail], R) :- len(Tail, N), R is N + 1.
```

Matching a Goal to a Rule – Continued (1/2)

Recall that we had:

```
len([], 0).
len([_Head|Tail], R) :- len(Tail, N), R is N + 1.
```

And we ask:

```
? len([1 | [2 | []]], X).
```

Matching a Goal to a Rule – Continued (1/2)

Recall that we had:

```
len([], 0).
len([_Head|Tail], R) :- len(Tail, N), R is N + 1.
```

And we ask:

```
? len([1 | [2 | []]], X).
```

Prolog uses unification to the determine that:

1. We cannot unify $[]$ with $[1|[2|[]]]$ so the first rule (fact) is not applicable.

Matching a Goal to a Rule – Continued (1/2)

Recall that we had:

```
len([], 0).
len([_Head|Tail], R) :- len(Tail, N), R is N + 1.
```

And we ask:

```
? len([1 | [2 | []]], X).
```

Prolog uses unification to the determine that:

1. We cannot unify $[]$ with $[1|[2|[]]]$ so the first rule (fact) is not applicable.
2. We *can* unify $[\_Head|Tail]$ with $[1|[2|[]]]$ using the substitution $\{\_\text{Head} \mapsto 1, \text{Tail} \mapsto [2|[]]\}$.

Matching a Goal to a Rule – Continued (1/2)

Recall that we had:

```
len([], 0).
len([_Head|Tail], R) :- len(Tail, N), R is N + 1.
```

And we ask:

```
? len([1 | [2 | []]], X).
```

Prolog uses unification to the determine that:

1. We cannot unify $[]$ with $[1|[2|[]]]$ so the first rule (fact) is not applicable.
2. We *can* unify $[\_Head|Tail]$ with $[1|[2|[]]]$ using the substitution $\{\text{\_Head} \mapsto 1, \text{Tail} \mapsto [2|[]]\}$.
3. We then apply the substitution to the rule body to obtain the new goals: $\text{len}([2|[]], R)$ and R is $X + 1$, and recurse.

Matching a Goal to a Rule – Continued (2/2)

To recap:

Matching a Goal to a Rule – Continued (2/2)

To recap:

- Prolog searches for a fact or rule *from top to bottom* of the file where the current goal unifies with the head atom.

Matching a Goal to a Rule – Continued (2/2)

To recap:

- Prolog searches for a fact or rule *from top to bottom* of the file where the current goal unifies with the head atom.
- If a match is found, Prolog applies the found substition to the rule body which now become the sub-goals of the query.

Matching a Goal to a Rule – Continued (2/2)

To recap:

- Prolog searches for a fact or rule *from top to bottom* of the file where the current goal unifies with the head atom.
- If a match is found, Prolog applies the found substition to the rule body which now become the sub-goals of the query.
- The sub-goals are evaluated from left to right.

Matching a Goal to a Rule – Continued (2/2)

To recap:

- Prolog searches for a fact or rule *from top to bottom* of the file where the current goal unifies with the head atom.
- If a match is found, Prolog applies the found substition to the rule body which now become the sub-goals of the query.
- The sub-goals are evaluated from left to right.
- A goal is satisfied once we reach a fact.

Matching a Goal to a Rule – Continued (2/2)

To recap:

- Prolog searches for a fact or rule *from top to bottom* of the file where the current goal unifies with the head atom.
- If a match is found, Prolog applies the found substition to the rule body which now become the sub-goals of the query.
- The sub-goals are evaluated from left to right.
- A goal is satisfied once we reach a fact.

Hence: Be careful about evaluation order. It matters!

Print Debugging is Back! (1/2)

If we write:

```
edge(1, 2).
edge(2, 3).
path(X, Y) :- edge(X, Y), write('rule1\n').
path(X, Z) :- edge(X, Y), path(Y, Z), write('rule2\n').
```

Print Debugging is Back! (1/2)

If we write:

```
edge(1, 2).
edge(2, 3).
path(X, Y) :- edge(X, Y), write('rule1\n').
path(X, Z) :- edge(X, Y), path(Y, Z), write('rule2\n').
```

And ask:

```
-? path(1, 3).
```

Print Debugging is Back! (1/2)

If we write:

```
edge(1, 2).
edge(2, 3).
path(X, Y) :- edge(X, Y), write('rule1\n').
path(X, Z) :- edge(X, Y), path(Y, Z), write('rule2\n').
```

And ask:

```
-? path(1, 3).
```

Prolog prints:

```
rule1
rule2
yes
```

Print Debugging is Back! (2/2)

We can explore Prolog's evaluation order by writing:

```
path(X, Z) :- write('A'), edge(X, Y), write('B'), path(Y, Z), write('C').
```

And asking:

```
-? path(1, 3).
```

Print Debugging is Back! (2/2)

We can explore Prolog's evaluation order by writing:

```
path(X, Z) :- write('A'), edge(X, Y), write('B'), path(Y, Z), write('C').
```

And asking:

```
-? path(1, 3).
```

Question: What does this print?

Prolog Extensions

Prolog comes with several extensions:

- Cuts (a mechanism to control backtracking)
- Higher-order predicates (e.g. `findall`)
- Reflection (e.g. `clause`)
- Tabling (ala Datalog)

You will most likely need some of these for any serious Prolog programming.

A Larger Example

The Wolf, Goat, and Cabbage Problem (1/2)

"The wolf, goat, and cabbage problem is a river crossing puzzle. It dates back to at least the 9th century and has entered the folklore of several cultures." – *Wikipedia*

The Wolf, Goat, and Cabbage Problem (2/2)

Via Wikipedia:

> *"A **farmer** with **a wolf**, **a goat**, and **a cabbage** must cross a river by boat. The boat can carry only the farmer and a single item. If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage. How can they cross the river without anything being eaten?"*

We can solve this problem with Prolog.

The Wolf, Goat, and Cabbage Problem (2/2)

Via Wikipedia:

> *"A **farmer** with **a wolf**, **a goat**, and **a cabbage** must cross a river by boat. The boat can carry only the farmer and a single item. If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage. How can they cross the river without anything being eaten?"*

We can solve this problem with Prolog.

Question: But can you solve it with your brain? (and some paper...)

Wolf, Goat, and Cabbage in Prolog (1/8)

We consider the river to have two sides: the **west bank** (`w`) and **east bank** (`e`).

Initially, everyone will be on the west bank.

We can then define that one can travel from west to east and east to west:

```
travel(w, e).
travel(e, w).
```

We further assume we have four constants: `farmer`, `wolf`, `goat`, `cabbage`.

Wolf, Goat, and Cabbage in Prolog (2/8)

We define a configuration of the problem as a 4-tuple (using a list): We order the travelers as: `farmer`, `wolf`, `goat`, `cabbage`. Now:

- The list: `[w, w, w, w]` means that everyone is on the west bank.
- The list: `[e, e, e, e]` means that everyone is on the east bank.
- The list: `[e, e, e, w]` means that the farmer, wolf, and goat is on the east bank whereas the cabbage is on the west bank — which is okay.
- The list: `[e, w, w, e]` means that the farmer and cabbage is on the east bank whereas the wolf and the goat is on the east bank — which is NOT OKAY!

Wolf, Goat, and Cabbage in Prolog (3/8)

We now define a ternary relation: `move(state1, moved, state2)`:

If the farmer and wolf are on the west bank then they can travel to the east bank.

The goat and cabbage stay where they are.

We can capture this with the fact:

```
move([w, w, G, C], wolf, [e, e, G, C]).
```

Wolf, Goat, and Cabbage in Prolog (3/8)

We now define a ternary relation: `move(state1, moved, state2)`:

If the farmer and wolf are on the west bank then they can travel to the east bank.

The goat and cabbage stay where they are.

We can capture this with the fact:

```
move([w, w, G, C], wolf, [e, e, G, C]).
```

They can also travel back, so we need:

```
move([e, e, G, C], wolf, [w, w, G, C]).
```

Wolf, Goat, and Cabbage in Prolog (3/8)

We now define a ternary relation: `move(state1, moved, state2)`:

If the farmer and wolf are on the west bank then they can travel to the east bank.

The goat and cabbage stay where they are.

We can capture this with the fact:

```
move([w, w, G, C], wolf, [e, e, G, C]).
```

They can also travel back, so we need:

```
move([e, e, G, C], wolf, [w, w, G, C]).
```

This is a bit tedious, so let us make use of `travel`:

```
move([X, X, G, C], wolf, [Y, Y, G, C]) :- travel(X, Y).
```

Wolf, Goat, and Cabbage in Prolog (4/8)

We have:

```
move([X, X, G, C], wolf, [Y, Y, G, C]) :- travel(X, Y).
```

Wolf, Goat, and Cabbage in Prolog (4/8)

We have:

```
move([X, X, G, C], wolf, [Y, Y, G, C]) :- travel(X, Y).
```

and we need three more rules:

```
move([X, W, X, C], goat,    [Y, W, Y, C]) :- travel(X, Y).
move([X, W, G, X], cabbage, [Y, W, G, Y]) :- travel(X, Y).
move([X, W, G, C], nothing, [Y, W, G, C]) :- travel(X, Y).
```

Wolf, Goat, and Cabbage in Prolog (4/8)

We have:

```
move([X, X, G, C], wolf, [Y, Y, G, C]) :- travel(X, Y).
```

and we need three more rules:

```
move([X, W, X, C], goat,    [Y, W, Y, C]) :- travel(X, Y).
move([X, W, G, X], cabbage, [Y, W, G, Y]) :- travel(X, Y).
move([X, W, G, C], nothing, [Y, W, G, C]) :- travel(X, Y).
```

Question: Why do we need the last rule?

Wolf, Goat, and Cabbage in Prolog (5/8)

Recall: We want to ensure that the (a) wolf does not eat the goat, and (b) the goat does not eat the cabbage.

We define the states that are safe. There are two cases:

(1) The goat is on the same bank as the farmer:

```
safe([X, _, X, _]).  // Recall: farmer, wolf, goat, cabbage
```

(2) Or the wolf and cabbage are on the same bank as the farmer:

```
safe([X, X, _, X]).
```

Wolf, Goat, and Cabbage in Prolog (5/8)

Recall: We want to ensure that the (a) wolf does not eat the goat, and (b) the goat does not eat the cabbage.

We define the states that are safe. There are two cases:

(1) The goat is on the same bank as the farmer:

```
safe([X, _, X, _]).  // Recall: farmer, wolf, goat, cabbage
```

(2) Or the wolf and cabbage are on the same bank as the farmer:

```
safe([X, X, _, X]).
```

Question: Are (1) and (2) equivalent to (a) and (b)?

Wolf, Goat, and Cabbage in Prolog (6/8)

We can now define a solution: A solution is a pair of a state and a list of moves that brings everyone safely to the east bank.

Wolf, Goat, and Cabbage in Prolog (6/8)

We can now define a solution: A solution is a pair of a state and a list of moves that brings everyone safely to the east bank.

If everyone is already on the east bank there is nothing to be done:

```
solution([e, e, e, e], []).
```

Wolf, Goat, and Cabbage in Prolog (6/8)

We can now define a solution: A solution is a pair of a state and a list of moves that brings everyone safely to the east bank.

If everyone is already on the east bank there is nothing to be done:

```
solution([e, e, e, e], []).
```

We can move to a new state provided that it is (i) safe and (ii) solvable:

```
solution(State, [FirstMove | OtherMoves]) :-
    move(State, FirstMove, NextState),
    safe(NextState),
    solution(NextState, OtherMoves).
```

Wolf, Goat, and Cabbage in Prolog (7/8)

We can now ask Prolog to compute a solution:

```
? solution([w, w, w, w], X).
```

Wolf, Goat, and Cabbage in Prolog (7/8)

We can now ask Prolog to compute a solution:

```
? solution([w, w, w, w], X).
```

And then we wait ...

Wolf, Goat, and Cabbage in Prolog (7/8)

We can now ask Prolog to compute a solution:

```
? solution([w, w, w, w], X).
```

And then we wait ...

And then we wait some more ...

Wolf, Goat, and Cabbage in Prolog (7/8)

We can now ask Prolog to compute a solution:

```
? solution([w, w, w, w], X).
```

And then we wait ...

And then we wait some more ...

And, hey, what's happening?

Wolf, Goat, and Cabbage in Prolog (7/8)

We can now ask Prolog to compute a solution:

```
? solution([w, w, w, w], X).
```

And then we wait ...

And then we wait some more ...

And, hey, what's happening?

Problem: We have infinitely many moves leading to no solution. The farmer is just going back and forth, forever.

Wolf, Goat, and Cabbage in Prolog (8/8)

Solution: We (indirectly) bound the recursion depth.

Wolf, Goat, and Cabbage in Prolog (8/8)

Solution: We (indirectly) bound the recursion depth.

How? We fix the length of the list with moves:

```
? length(X, 7), solution([w, w, w, w], X).
```

Wolf, Goat, and Cabbage in Prolog (8/8)

Solution: We (indirectly) bound the recursion depth.

How? We fix the length of the list with moves:

```
? length(X, 7), solution([w, w, w, w], X).
```

Prolog now answers:

```
X = [goat, nothing, wolf, goat, cabbage, nothing, goat]
```

Wolf, Goat, and Cabbage in Prolog (8/8)

Solution: We (indirectly) bound the recursion depth.

How? We fix the length of the list with moves:

```
? length(X, 7), solution([w, w, w, w], X).
```

Prolog now answers:

```
X = [goat, nothing, wolf, goat, cabbage, nothing, goat]
```

Question: Why did I choose 7? How did I know what number to choose?

Wolf, Goat, and Cabbage in Prolog (8/8)

Solution: We (indirectly) bound the recursion depth.

How? We fix the length of the list with moves:

```
? length(X, 7), solution([w, w, w, w], X).
```

Prolog now answers:

```
X = [goat, nothing, wolf, goat, cabbage, nothing, goat]
```

Question: Why did I choose 7? How did I know what number to choose?

Example inspired by UCSD: `https://cseweb.ucsd.edu/classes/fa09/cse130/misc/prolog/goat_etc.html`

Getting Started with Prolog

Prolog Dialects and Implementations

As with Datalog there are many Prolog dialects and implementations.

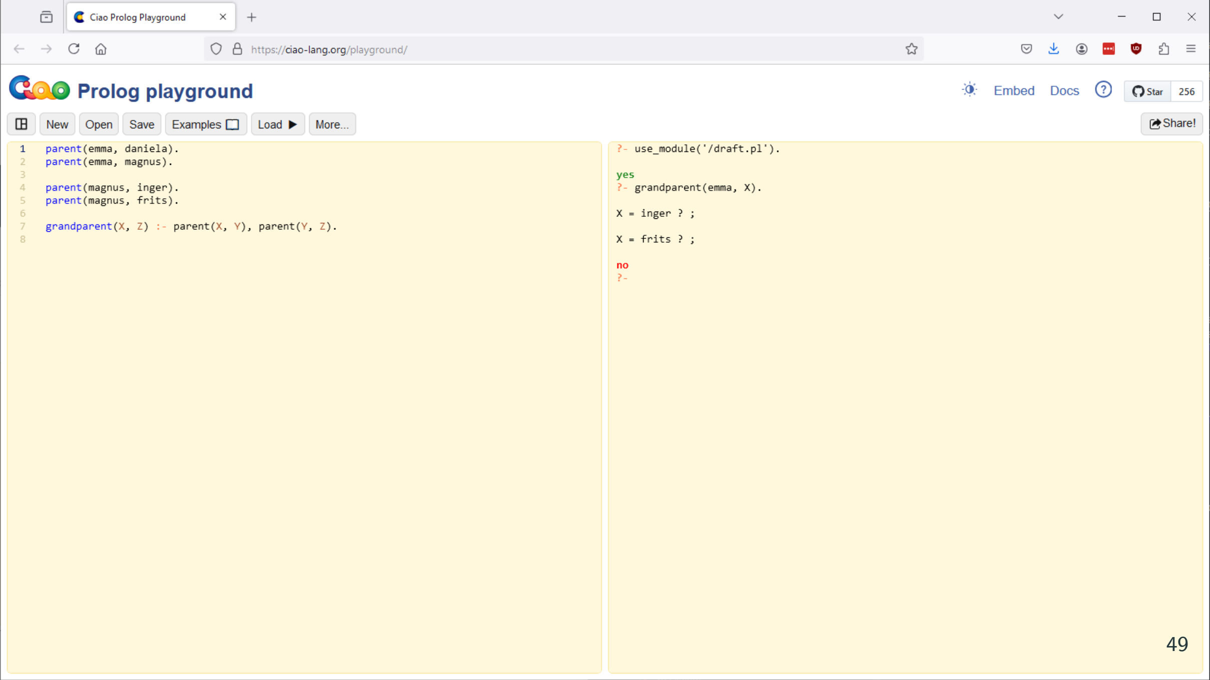
The most popular and battle-tested Prolog implementations are:

- **Ciao Prolog** is an open-source research project from UPM and IMDEA
  `https://ciao-lang.org/`
- **Gnu Prolog** is an open-source Prolog implementation
  `http://www.gprolog.org/`
- **SWI Prolog** is an open-source Prolog implementation
  `https://www.swi-prolog.org/`
- **XSB Prolog** is a commercial Prolog implementation with tabling
  `https://xsb.com/xsb-prolog/`

Ciao Prolog

I recommend that you use **Ciao Prolog** (with SWI Prolog as backup)

- Ciao has been developed for more than 40 years.
- Ciao is large research project with many cool ideas.
- Runs in the browser via WebAssembly:
  `https://ciao-lang.org/playground/`


Ciao Prolog Playground
https://ciao-lang.org/playground/
Prolog playground
Embed
Docs
Star
256
New
Open
Save
Examples
Load
More...
Share!
parent(emma, daniela).
parent(emma, magnus).
parent(magnus, inger).
parent(magnus, frits).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
?- use_module('/draft.pl').
yes
?- grandparent(emma, X).
X = inger ? ;
X = frits ? ;
no
?-

Ciao Playground

Your first task:

Run the Wolf, Goat, and Cabbage program on the playground.

`https://ciao-lang.org/playground/`

Summary

Prolog is a **goal-driven** logic programming language:

- Datalog is for tabling. Prolog is for search.

Summary

Prolog is a **goal-driven** logic programming language:

- Datalog is for tabling. Prolog is for search.

Prolog is Turing-complete. We get the good and the bad:

- We can express what we want, but programs may fail to terminate.

Summary

Prolog is a **goal-driven** logic programming language:

- Datalog is for tabling. Prolog is for search.

Prolog is Turing-complete. We get the good and the bad:

- We can express what we want, but programs may fail to terminate.

Prolog is less declarative than Datalog: evaluation order matters.

Summary

Prolog is a **goal-driven** logic programming language:

- Datalog is for tabling. Prolog is for search.

Prolog is Turing-complete. We get the good and the bad:

- We can express what we want, but programs may fail to terminate.

Prolog is less declarative than Datalog: evaluation order matters.

Prolog supports compound data types (lists, trees, ...).