CS 381: Programming Language Fundamentals

Summer 2015

Introduction to Logic Programming in Prolog
July 29, 2015

Outline

Programming paradigms

Logic programming basics
- Introduction to Prolog
- Predicates, queries and rules

Understanding the query engine
- Goal search and unification
- Structuring recursive rules

Complex terms and lists

Programming paradigms

What is a programming paradigm?

Paradigm: A conceptual model underlying the theories and practice of a scientific subject

scientific subject = *programming*

Programming paradigm: A conceptual model underlying the theories and practice of programming

Imperative paradigm

Imp.hs

Imperative model

data	set of state variables	`type State = [(Name,Val)]`
computation	transformation of state	`State -> State`

Needs two sub-languages:

- **expressions** to describe values to store in variables (**`Expr`**)
- **statements** to describe state changes and control flow (**`Stmt`**)

Semantic functions:

- `semE :: Expr -> State -> Val`
- `semS :: Stmt -> State -> State`

Object-oriented paradigm

An extension/refinement of the imperative paradigm

Obj.hs

Object-oriented model

data	set of objects with state	`type Object = (State,[Method])`
		`type Method = (Name, State -> State)`
computation	evolution of objects	`[Object] -> [Object]`

Needs **expression** and **statement** sub-languages, but also extended statements with:

- constructs to **create objects** and **invoke methods**

New statement semantic function:

- `semS :: Stmt -> [Object] -> [Object]`

Functional paradigm

Functional model

data	structured values	`data Val = ...`
computation	functions over values	`Val -> Val`

Generally just one language (e.g. lambda calculus):

- **expressions** describe functions and values (`Expr`)

Semantic functions:

- `sem :: Expr -> Val`

Oregon State University

Logic paradigm

Logical model

data	set of values and relations	`type Known = [(Val,...,Val)]`
computation	query over relations	`type Query = Known -> Known`

Generally just one language:

- **relations** describe both knowledge and queries (**`Rel`**)

Semantic functions:

- **`sem :: Rel -> Query`**

Comparison of programming paradigms

Paradigm	**View of computation**
imperative	*sequence of state transformations*
object-oriented	*simulation of interacting objects*
functional	*function mapping input to output*
logic	*queries over logical relations*

Outline

Programming paradigms

Logic programming basics
- Introduction to Prolog
- Predicates, queries and rules

Understanding the query engine
- Goal search and unification
- Structuring recursive rules

Complex terms and lists

What is Prolog?

- an **untyped logic** programming language
- programs are **rules** that define **relations** on values
- run a program by formulating a **goal** or **query**
- result of a program: a true/false answer and a **binding of free variables**

Logic: a tool for reasoning

Syllogism (logical argument) — Aristotle, 350 BCE

Every human is mortal.

Socrates is human.

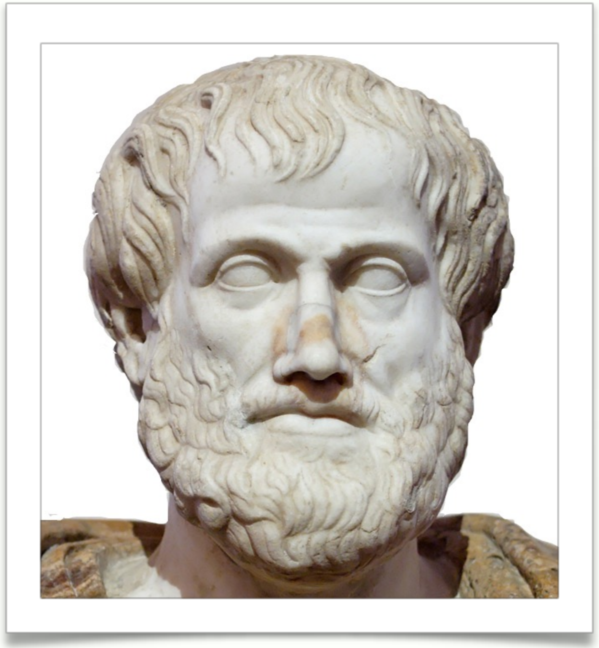
Therefore, Socrates is mortal.

First-order logic — Gottlob Frege, 1879 *Begriffsschrift*

$$\forall x. Human(x) \rightarrow Mortal(x)$$

$$Human(Socrates)$$

$$\therefore Mortal(Socrates)$$



Logic and programming

rule	$\forall x. Human(x) \rightarrow Mortal(x)$	→	*logic program*
fact	$Human(Socrates)$	→	*logic program*
goal/query	$\therefore Mortal(Socrates)$	→	*logic program execution*

Prolog program

```
mortal(X) :- human(X).
human(Socrates).
```

Prolog query (interactive)

```
?- mortal(Socrates).
true.
```

Outline

Programming paradigms

Logic programming basics
- Introduction to Prolog
- Predicates, queries and rules

Understanding the query engine
- Goal search and unification
- Structuring recursive rules

Complex terms and lists

SWI-Prolog logistics

Predicate	Description
`[myfile].`	load definitions from "myfile.pl"
`listing(P).`	lists facts and rules related to predicate **P**
`trace.`	turn on tracing
`nodebug.`	turn off tracing
`help.`	open help window (requires X11 on Mac)
`halt.`	quit

GNU-Prolog uses the same commands — except **`help`**!

Atoms

An **atom** is just a primitive value

- string of characters, numbers, underscores starting with a **lowercase letter**:
  - **`hello`**, **`socrates`**, **`uP_aNd_4tOm`**
- any single quoted string of characters:
  - **`‘Hello world!’`**, **`‘Socrates’`**
- numeric literals: **`123`**, **`-345`**
- empty lists: **`[]`**

Variables

A variable can be used in rules and queries

- string of characters, numbers, underscores starting with an uppercase letter or an underscore
  - `X`, `SomeHuman`, `_g_123`, `This_Human`
- special variable: _ (just an underscore)
  - unifies with anything — “don’t care”

Predicates

Basic entity in Prolog is a **predicate** $\cong$ **relation** $\cong$ **set**

Unary predicate

```
hobbit(bilbo).
hobbit(frodo).
hobbit(sam).
```

```
hobbit = {bilbo, frodo, sam}
```

Binary predicate

```
likes(bilbo, frodo).
likes(frodo, bilbo).
likes(sam, frodo).
likes(frodo, ring).
```

```
likes = {(bilbo, frodo), (frodo, bilbo)
         (sam, frodo), (frodo, ring)}
```

Simple goals and queries

Predicates are:

- **defined** in a file *the program*
- **queried** in the REPL *running the program*

Response to a query is a `true/false` answer *(or* **`yes/no`***)*

when `true`, provides a **binding** for each variable in the query

Is `sam` a hobbit?

```
?- hobbit(sam).
true.
```

Is `gimli` a hobbit?

```
?- hobbit(gimli).
false.
```

Who is a `hobbit?`

```
?- hobbit(X).
X = bilbo ;
X = frodo ;
X = sam .
```

Type `;` after each response to search for another

Querying relations

hobbits.pl

You can query **any argument** of a predicate

- this is fundamentally different from passing arguments to functions!

Definition

```
likes(bilbo, frodo).
likes(frodo, bilbo).
likes(sam, frodo).
likes(frodo, ring).
```

```
?- likes(frodo, Y).
Y = bilbo ;
Y = ring .
```

```
?- likes(X, frodo).
X = bilbo ;
X = sam .
```

```
?- likes(X, Y).
X = bilbo,
Y = frodo ;
X = frodo,
Y = bilbo ;
X = sam,
Y = frodo ;
X = frodo,
Y = ring .
```

Overloading predicates

hobbits.pl

Predicates with the **same name** but **different arities** are **different predicates**!

```
hobbit/1
hobbit(bilbo).
hobbit(frodo).
hobbit(sam).
```

```
?- hobbit(X).
X = bilbo ;
X = frodo ;
X = sam .
```

```
hobbit/2
hobbit(bilbo, rivendell).
hobbit(frodo, hobbiton).
hobbit(sam, hobbiton).
hobbit(merry, buckland).
hobbit(pippin, tookland).
```

```
?- hobbit(X,_).
 ...
X = merry ;
X = pippin .
```

Conjunction

Comma (,) denotes **logical and** of two predicates

Do `sam` **and** `frodo` **like each other?**

```
?- likes(sam,frodo), likes(frodo,sam).
true.
```

Do `merry` **and** `pippin` **live in the same place?**

```
?- hobbit(merry,X), hobbit(pippin,X).
false.
```

Do any hobbits live in the same place?

```
?- hobbit(H1,X), hobbit(H2,X), H1 \= H2.
H1 = frodo, X = hobbiton, H2 = sam.
```

```
likes(frodo, sam).
likes(sam, frodo).
likes(frodo, ring).

hobbit(frodo, hobbiton).
hobbit(sam, hobbiton).
hobbit(merry, buckland).
hobbit(pippin, tookland).
```

H1 and **H2** must be different!

Oregon State University

Rules

hobbits.pl

Rule: `head :- body`

The **head** is true **if** the **body** is true

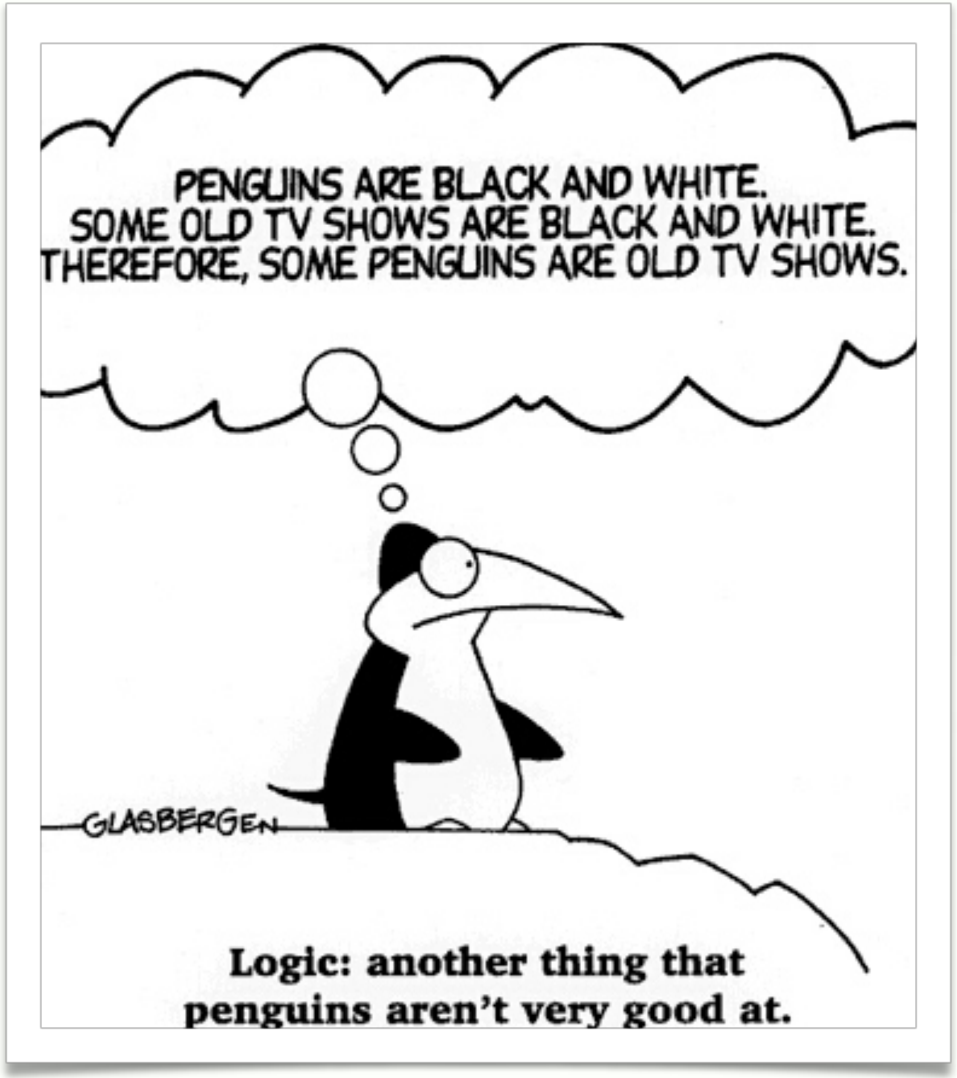
Examples

```
likes(X,beer)  :- hobbit(X,_).
likes(X,boats) :- hobbit(X,buckland).

danger(X) :- likes(X,ring).
danger(X) :- likes(X,boats), likes(X,beer).
```

Note that **disjunction** is described by multiple rules

Prolog


PENGUINS ARE BLACK AND WHITE.
SOME OLD TV SHOWS ARE BLACK AND WHITE.
THEREFORE, SOME PENGUINS ARE OLD TV SHOWS.
GLASBERGEN
Logic: another thing that
penguins aren't very good at.

Outline

Programming paradigms

Logic programming basics
- Introduction to Prolog
- Predicates, queries and rules

Understanding the query engine
- Goal search and unification
- Structuring recursive rules

Complex terms and lists

How does Prolog solve queries?

Basic algorithm for solving a (sub)goal

1. linearly **search** data base for candidate facts/rules
2. attempt to **unify** candidate with goal

   if unification is **successful**:

   - if a **fact** — we're done with this goal!
   - if a **rule** — add body of rule as new subgoal

   if unification is **unsuccessful**: keep searching
3. backtrack if we reach the end of the database

1. linearly search the database for candidate facts/rules

What is a candidate fact/rule?

- **fact**: predicate matches the goal
- **rule**: predicate of its **head** matches the goal

Example goal: `likes(merry,Y)`

Candidates

```
likes(sam,frodo).
likes(merry,pippin).
likes(X,beer) :- hobbit(X).
```

Not candidates

```
hobbit(merry,buckland).
danger(X) :- likes(X,ring).
likes(merry,pippin,mushrooms).
```

2. attempt to unify candidate with goal

Unification

Find an **assignment of variables** that makes its arguments **syntactically equal**

Prolog: `A = B` means attempt to **unify** `A` and `B`

Candidates

```
?- likes(merry,Y) = likes(sam,frodo).
false.

?- likes(merry,Y) = likes(merry,pippin).
Y = pippin .

?- likes(merry,Y) = likes(X,beer).
X = merry ; Y = beer .
```

2a. if **fail**, try next candidate

2b. if **success**, add new subgoal(s)

Tracking subgoals

Deriving solutions through rules

1. maintain a list of goals that need to be solved
   - when this list is empty we're finished!
2. if current goal unifies with a rule **head**, add **body** as subgoals to list
3. after unification, **substitute variables** in all goals in the list!

Database

```
1 lt(one,two).
2 lt(two,three).
3 lt(three,four).
4 lt(X,Z) :- lt(X,Y), lt(Y,Z).
```

Sequence of goals for `lt(one,four)`

```
                     lt(one,four)
4: X=one,Z=four      lt(one,Y1), lt(Y1,four)
1: Y1=two            lt(two,four)
4: X=two,Z=four      lt(two,Y2), lt(Y2,four)
2: Y2=three          lt(three,four)
3: true              done!
```

3. Backtracking

For each subgoal, Prolog maintains:

- the **search state** (goals + assignments) before it was produced
- a **pointer** to the rule that produced it

When a **subgoal fails**:

- **restore** the previous state
- **resume** search for previous goal from the pointer

When the **initial goal fails**: return **false**

Outline

Programming paradigms

Logic programming basics
- Introduction to Prolog
- Predicates, queries and rules

Understanding the query engine
- Goal search and unification
- Structuring recursive rules

Complex terms and lists

Potential for infinite search

Why care about how goal searches work?

One reason: so we can write **recursive rules** that don't loop forever!

Contra-example: symmetry

```
likes(frodo,sam).
likes(merry,pippin).
likes(frodo,bilbo).
likes(X,Y) :- likes(Y,X).
```

```
?- likes(bilbo,merry).
ERROR: Out of local stack
```

Contra-example: transitivity

```
lt(one,two).
lt(two,three).
lt(three,four).
lt(X,Z) :- lt(X,Y), lt(Y,Z).
```

```
?- lt(three,one).
ERROR: Out of local stack
```

Strategies for writing recursive rules

How to avoid infinite search

1. always list **non-recursive cases first**
2. use "helper" predicates to **enforce progress** during search

Example: symmetry

```
likesP(frodo,sam).
likesP(merry,pippin).
likesP(frodo,bilbo).
likes(X,Y) :- likesP(X,Y).
likes(X,Y) :- likesP(Y,X).
```

```
?- likes(bilbo,merry).
false.
```

Example: transitivity

```
ltP(one,two).
ltP(two,three).
ltP(three,four).
lt(X,Y) :- ltP(X,Y)
lt(X,Z) :- ltP(X,Y), lt(Y,Z).
```

```
?- lt(three,one).
false.
```


Oregon State UNIVERSITY

Outline

Programming paradigms

Logic programming basics
- Introduction to Prolog
- Predicates, queries and rules

Understanding the query engine
- Goal search and unification
- Structuring recursive rules

Complex terms and lists

Representing structured data

Can represent structured data by **nested predicates**

Example database

```
rides(gandalf, horse(white)).
rides(sam, donkey(grey)).
rides(frodo, pony(grey)).
rides(strider, horse(black)).
```

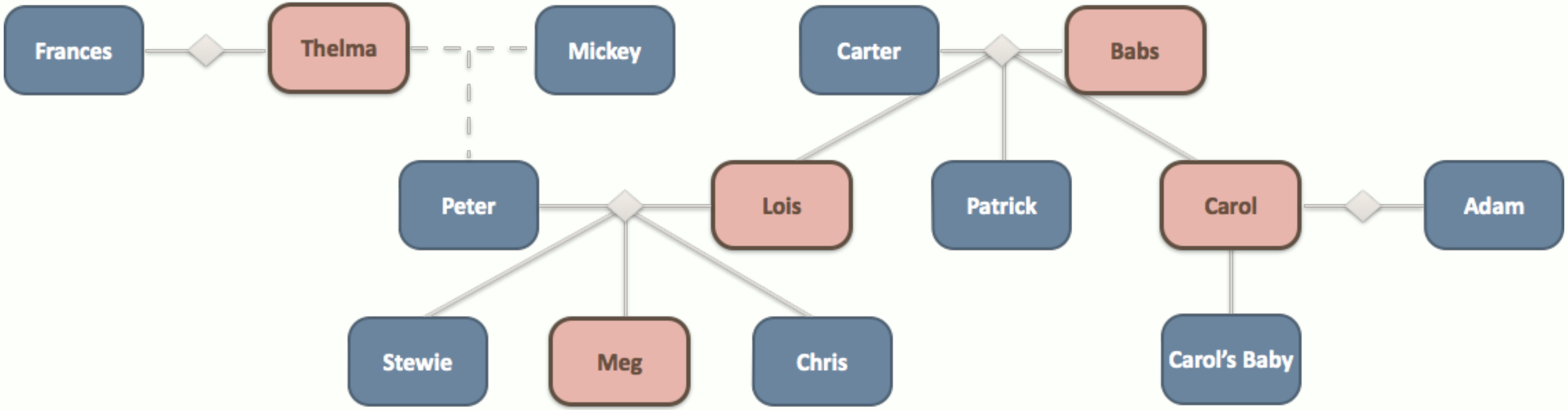
```
?- rides(gandalf, X).
X = horse(white) .
```

```
?- rides(X, horse(Y)).
X = gandalf, Y = white ;
X = strider, Y = black .
```

Variables *cannot* be used for predicates:

`?- rides(X, Y(grey)).` ← **illegal!**

Extended example: family tree



Equality

Different forms of equality between terms

1. unification `=` *and* `\=`
2. equivalence `==` *and* `\==`
3. evaluation `=:=` *and* `=\=`

Unification equality

Two terms are *equal* when they can be *instantiated* so that they *become identical*

```
?- 3=3.
true.
```

```
?- X=3.
X = 3.
```

```
?- X=Y.
X = Y.
```

evaluation equality

```
?- likes(X,red)=likes(john,Y).
X = john,
Y = red.
```

```
?- 3=4.
false.
```

```
?- 3+1=4.
false.
```

```
?- car(red)=car(X).
X = red.
```

term

Unification

A *unifier* for two terms `T` and `T'` is a *substitution* for variables **σ**, such that:

`σ(T) = σ(T')`

```
?- 3=3.
true.
```

```
σ = {}
σ(3) = 3 = σ(3)
```

```
?- X=3.
X = 3.
```

```
σ = {X -> 3}
σ(X) = 3 = σ(3)
```

```
?- X=Y.
X = Y.
```

```
σ = {X -> Y}
σ(X) = Y = σ(Y)
```

```
?- likes(X,red)=likes(john,Y).
X = john,
Y = red.
```

```
σ = {X -> john, Y -> red}
σ(likes(X,red)) = likes(john,red)
                = σ(likes(john,Y))
```

```
?- car(red)=car(X).
X = red.
```

```
σ = {X -> red}
σ(car(red)) = σ(car(X))
```


Oregon State University

Equivalence

Two terms are *equivalent* if they are *identical*.

```
?- 3==3.
true.
```

```
?- X==3.
false.
```

```
?- X==Y.
false.
```

```
?- X=3, X==3.
X = 3.
```

```
?- X==Y, X=Y.
false.
```

```
?- X=Y, X==Y.
X = Y.
```

different from object/reference equality

Oregon State UNIVERSITY

Evaluation equality

Two terms are *evaluation equivalent* if they *evaluate* to the same number.

```
?- 3+1=:=4.
true.
```

```
?- 3+1==4.
false.
```

```
?- 3+1=4.
false.
```

```
?- X=3, X*2=:=X+3.
X = 3.
```

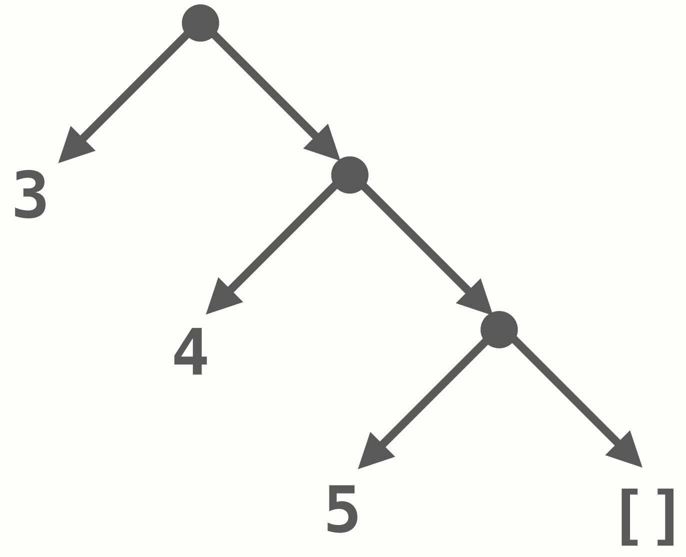
```
?- 3=:=X.
ERROR: =:=/2: Arguments are not
sufficiently instantiated
```

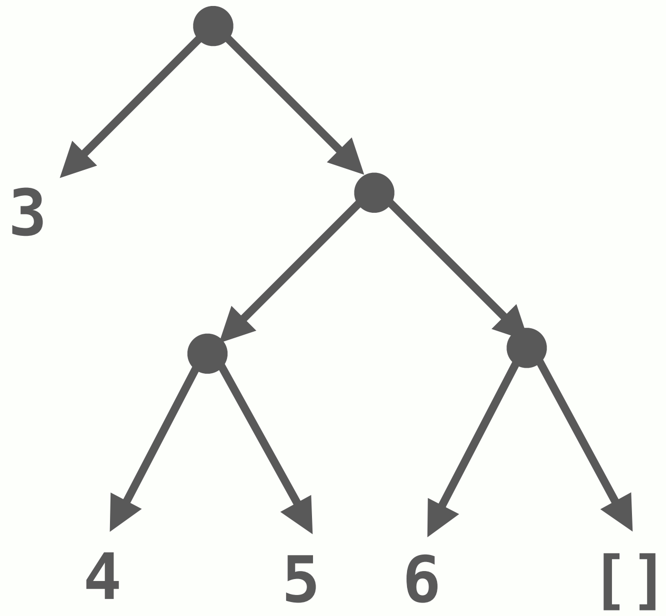
```
?- X*2=:=X+3, X=3.
ERROR: =:=/2: Arguments are not
sufficiently instantiated
```

List construction

list.pl

Lists are **terms** with special syntax





$[3,4,5] \equiv 3 . (\ 4 . (\ 5 . (\ [\ ]\ )\ )\ )$

[3, [4, 5], 6]

Cons Functor   Empty List

List patterns

```
story([3,little,pigs])
```

Head Tail

```
?- story([X|Y]).

X = 3,
Y = [little,pigs].
```

```
?- story([X,Y|Z]).

X = 3,
Y = little,
Z = pigs.
```

```
?- story([X,Y,Z]).

X = 3,
Y = little,
Z = pigs.
```

```
?- story([X,Y,Z|V]).

X = 3,
Y = little,
Z = pigs,
V = [].
```

```
?- story([X,Y,Z,V]).

false.
```

Haskell	Prolog
x:y:z	[X,Y|Z]
[x,y,z]	[X,Y,Z]

List predicates

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

wildcard — matches anything

```
?- member(3,[2,3,4,3]).
true ;
true.
```

```
?- member(3,[2,3,4,3,1]).
true ;
true ;
false.
```

```
?-member(2,[2,3,4,3]).
true ;
false.
```

```
?-member(3,L).
L = [3|A] ;
L = [A,3|B] ;
L = [A,B,3|C]
…
```

```
?-member(X,[3,4]).
X = 3 ;
X = 4.
```