# Prolog: Lists

CS 331 Programming Languages
Lecture Slides
Monday, April 14, 2025

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

# Unit Overview
# The Prolog Programming Language

Topics

- ✓ PL feature: execution model
- ✓ PL category: logic PLs
- ✓ Introduction to Prolog
- ✓ Prolog: simple programming
- Prolog: lists
- Prolog: flow of control
- Prolog: interaction

# Review

# PL Category: Logic PLs

In **logic programming**, a computer program is a **knowledge base**. It typically contains two kinds of knowledge.

- **Facts.** Statements that are known to be true.
- **Rules.** Ways to find other true statements from those known.

Execution is then driven by a **query**.

**Logic programming languages** are PLs based on the ideas underlying logic programming. **Prolog** is the most important logic programming language.

# Prolog: Simple Programming — Facts, Queries, Rules, Goals

**Fact** (this is true; usually in source file):

```
abc.
def(a, 28).
```

**Rule** (this is true if these are true; usually in source file):

```
ghi(X, Y) :- X =< 3, Y is X+5.
```

**Query** (is this true? OR how to make it true? interactive):

```
?- ghi(3, 8).
?- ghi(3, Y), Y > 9.
```

A query sets up one or more **goals**. Rule bodies set up **subgoals.**

# Prolog: Simple Programming — Negation

"\+" is a 1-argument predicate that works as a prefix operator. It succeeds if its argument fails, so it means negation.

```
?- \+ 3 = 5.
true.
```

We can write "\+" in Prolog. Eventually, we will.

# Prolog: Lists

# Prolog: Lists
# Preliminaries [1/2]

A few useful things:

*For code from this topic, see* `list.pl`.

`_` (underscore)

Dummy variable. Unifies with any term, and indicates that its value will not be used. (Prolog calls unused variables "**singleton variables**" and warns you when it finds one. Use "`_`" to avoid such warnings.)

`var`/1

Takes any argument. Succeeds if it is a *free* variable.

`nonvar`/1

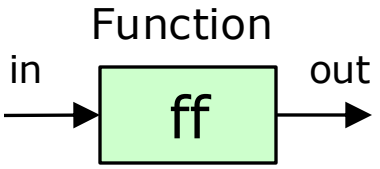
Takes any argument. Succeeds if it is *not* a free variable.

`call`/≥1

Arguments are a predicate & its arguments OR a single compound term. Calls the predicate/term and succeeds if it succeeds. Allows calling of a predicate stored in a variable: `X = foo, call(X, 3).`

# Prolog: Lists
## Preliminaries [2/2]

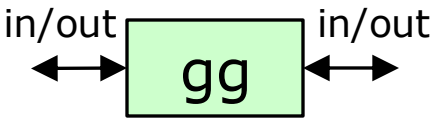
Recall that we can simulate a function with a Prolog predicate.

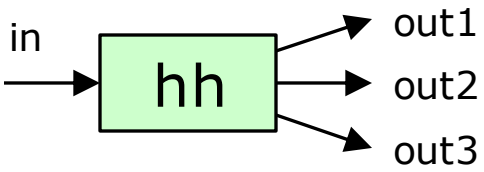



```
ff(in, Out)
```

We can use the same idea to simulate **generalized functions**: function-like entities that can be used in reverse, swapping input and output, or which can have more than one output for a single input, or some combination of these.

Generalized Functions




TO DO

- Write some Prolog predicates that simulate simple generalized functions.

*Done. See* `list.pl`.

# Prolog: Lists
## Lists & Tuples [1/3]

Prolog's basic list syntax is much like that of Haskell and Python.

```
[1,2,3]
```

Items in Prolog lists are terms. Different kinds of terms can be mixed in the same list.

```
?- X = [1, 'big dog', Y, 7+8, call, write(6), [3, 4.0]].
X = [1,'big dog',Y,7+8,call,write(6),[3,4.0]].
```

# Prolog: Lists
## Lists & Tuples [2/3]

Prolog lists are structured just like Scheme lists. And Prolog has the equivalent of Scheme dot (pair) notation, using "|".

Same as `(1 2)` →

| Scheme | Prolog |
|---|---|
| `(1 2 3)` | `[1, 2, 3]` |
| `()` | `[]` |
| `(1 (2 3))` | `[1, [2, 3]]` |
| `(1 . 2)` | `[1 \| 2]` |
| `(1 . (2 . ()))` | `[1 \| [ 2 \| []]` |

← Same as `[1, 2]`

We can place the bar just before the final item, like dot in Scheme.

| Scheme | Prolog |
|---|---|
| `(1 2 3 . 4)` | `[1, 2, 3 \| 4]` |

# Prolog: Lists
## Lists & Tuples [3/3]

Prolog also has tuples: comma-separated sequences. Parentheses are not required, unless there are precedence issues. However, I always include them.

```
?- X = (1,2,abc).
X = (1,2,abc).
?- X = [(1,2,3),4].
X = [(1,2,3),4].
```

All of the list & tuple notations can be used as patterns. That is, all the different kinds of notation support unification.

TO DO

- Write Prolog predicates to find the head and tail of a list.
- Can we write each of these using a single fact, not a rule?

*Done. See* `list.pl`.

# Lists & Recursion

The basic list-recursion pattern that we saw in Haskell and Scheme works well in Prolog, too:

- Base case: empty list.
- Recursive case: nonempty list. Do something with the head, and make a recursive call on the tail.

TO DO

- Write a Prolog predicate to find the length of a list.
- Write a Prolog predicate to concatenate two lists.
- Can concatenation be written as a generalized function?

*Done. See* `list.pl`.

## Lists & Encapsulated Loops

As in Haskell and Scheme, encapsulated loops work well in Prolog.

TO DO

- Write *map* in Prolog.
- Write *filter* in Prolog.
- Write *zip* in Prolog.
- Try our *map* with generalized functions.

*Done. See* `list.pl`.