Recursion in Logic Programming

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

November 1, 2021

Table of Contents

Table of Contents

SWI-Prolog

We will be using SWI-Prolog as our Prolog implementation for this course. SWI-Prolog is open source and is available on Windows, macOS, and Linux.

SWI-Prolog Demo

I will be providing a live demonstration on the basics of SWI-Prolog.

Table of Contents

Motivation

So far, the rules and queries we defined have been quite simple. Allowing recursion enables us to define more complex rules and queries.

Defining Natural Numbers with Logic Programming

- Now, Prolog does have built-in features for handling natural numbers that leverages the CPU's arithmetic functionality.
- However, it is instructive to define numbers in terms of logic programming rules.

Defining Natural Numbers with Logic Programming

Definition:

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).
```

where `s` is the successor function of arity 1. Please note that the notation `:-` is equivalent to $\leftarrow$; when you are typing your Prolog programs, you would use `:-`.

Examples of Natural Numbers

0	0
$s(0)$	1
$s(s(0))$	2
$s(s(s(0)))$	3
...	...
$s^n(0)$	n

Example: Let's run the query `natural_number(s(s(0)))`?

What if we wanted to state that natural number X is less than or equal to natural number Y?

Let's define `X <= Y`, which is equivalent to `'<='(X, Y)`. Note that this is not valid SWI-Prolog code; this is an example from the textbook.

Let's define `X <= Y`, which is equivalent to `'<='(X, Y)`. Note that this is not valid SWI-Prolog code; this is an example from the textbook.

```
0 <= X :- natural_number(X).
s(X) <= s(Y) :- X <= Y.
```

Let's also define `plus(X,Y,Z)`, where `Z` is the sum of `X` and `Y`:

Let's also define `plus(X,Y,Z)`, where `Z` is the sum of `X` and `Y`:

```
plus(0,X,X) :- natural_number(X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Let's do a few examples of `plus(X,Y,Z)` on the whiteboard.

We can also take advantage of conjunction in our recursive definitions. For example, below is the definition of `times(X,Y,Z)`, where `Z` is the product of `X` and `Y`:

We can also take advantage of conjunction in our recursive definitions. For example, below is the definition of `times(X,Y,Z)`, where `Z` is the product of `X` and `Y`:

```
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,XY),plus(XY,Y,Z).
```

What this does is perform the addition of `Y` to itself `X` times (e.g., $4 * 5 = 5 + 5 + 5 + 5$).

Table of Contents

Lists in Prolog

- Prolog has built-in support for lists.
- As in Scheme and most other Lisp-derived languages, a list is made up of pairs, where the first element is the head of the list and the second element contains the rest of the list; the final pair's second element contains an empty list.

Lists in Prolog

- The fact describing a pair is `.(X,Y)`, where `X` is the head of the list and `Y` is the second element. This is called the *dot functor*.
- Prolog provides syntactic sugar for pairs in the form of `[X|Y]`; this usage is far more common.
- The dot functor is the equivalent of a Lisp cons cell.

Defining a List in Prolog

```
list([]).
list([X|Xs]) :- list(Xs).
```

Let's show an example of `list` on the whiteboard.

How do we determine whether an element X is a member of a list?

How do we determine whether an element X is a member of a list?

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

Note that the first clause could also be

```
member(X,[X|Xs]) :- list(Xs).
```

Here are some possible queries that we could perform:

- `member(a, [a, b, c])?`
  - Is `a` a member of the list `[a, b, c]`?
- `member(X, [a, b, c])?`
  - What are the members of the list `[a, b, c]`?
- `member(a, X)?`
  - Which lists contain `a`?
    - Yes, we could do this in logic programming.

Finding the Prefix and Suffix of a List

`prefix` – Does the second list begin with the first list?

```
prefix([],Ys).
prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).
```

`suffix` – Is the end of the second list the first list?

```
suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) :- suffix(Xs,Ys).
```

Let's do some examples of `prefix` and `suffix` on the whiteboard.

`append` – Appends the second list to the first list

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Let's do an example of `append` on the whiteboard.

Interestingly, we could define `prefix` and `suffix` in terms of `append`:

```
prefix(Xs,Ys) :- append(Xs,As,Ys).
suffix(Xs,Ys) :- append(As,Xs,Ys).
```

Reversing a List

We could leverage append to reverse a list:

```
reverse([],[]).
reverse([X|Xs],Zs) :- reverse(Xs,Ys),append(Ys[X],Zs).
```

Reversing a List

However, we could make its execution more efficient by using an accumulator.

```
reverse(Xs,Ys) :- reverse(Xs,[],Ys).

reverse([X|Xs], Acc, Ys) :- reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

A Note About the Ordering of Rules in Prolog

The order of rules in Prolog matters; when executing queries Prolog looks up rules from the first defined to the last. For the examples in this slide rule ordering should not matter, but in more complex programs, rule order matters.

Table of Contents

Resolution and Unification

- In logic programming, an interpreter uses *resolution* and *unification* to answer queries.
- Borrowing from Professor Tom Austin's definitions from his Spring 2019 CS 152 slides:
  - "Resolution is the process of matching facts and rules to perform *inferencing*, the derivation of logical conclusions from the rules."
  - "Unification is the instantiation of variables via pattern matching."

Reading for Next Lecture

Please read Chapters 4 and 5 of *The Art of Prolog*. For supplemental reading, the textbook we used for Scheme, *Structure and Interpretation of Computer Programs*, has information about an entire logic programming engine built in Scheme in Chapter 4.4.