

11 – Logic Paradigm and Prolog

Logic:

Programs are built by defining logical rules and goals.

The runtime environment tries to achieve the goals by logical deduction.

Examples: Prolog, ASP, Datalog, Florid, Logtalk

There are other paradigms as well.

This section focuses on the **logic** paradigm, sometimes called “declarative” programming. In this paradigm, programs are built by defining logical rules and goals, and the runtime environment tries to achieve the goals by logical deduction. By far the most common logic programming language is Prolog, although there are others (ASP, Datalog, Florid, Logtalk, for example).

There are also many languages and systems that are derived from Prolog, such as the CLIPS expert system shell we use in CSc-180.

You are probably already familiar with *Boolean* logic. In Boolean logic, names are given to things that are either TRUE or FALSE. The operations AND, OR, and NOT then allow us to build a rich set of rules of inference, such as DeMorgan’s rule, and Modus Ponens. Boolean logic is sufficiently powerful for building the digital circuitry used in a CPU.

As a programming tool, however, Boolean logic has serious limitations. The only values available are TRUE and FALSE, which becomes problematic if we want to use other values, such as numbers or strings. It also is difficult in Boolean logic to express logical rules that can be applied generally depending on values that aren’t TRUE and FALSE. For example, if we want to create a rule that expresses that a person is eligible for social security if their age is over 62, it is difficult because there is no mechanism in Boolean logic for associating a number (such as age) with a person.

In *Predicate*, or *First-Order* Logic, we can also incorporate variables of other types. For instance, in predicate logic we can express things like:

$$\text{Age}(\text{Person}, Y) \wedge Y \geq 62 \rightarrow \text{Eligible}(\text{Person})$$

Some of these identifiers are Boolean (Age, Eligible), some are strings or numbers (Person, Y). A rule like this can generalize, because it can be applied in different cases, and for different people it will be true or false.

For this reason, most logic programming languages use predicate logic. In Prolog, statements are written as *Horn clauses*, of the following form:

$$a_1 \wedge a_2 \wedge a_3 \wedge \dots \wedge a_n \rightarrow b$$

The syntax for this in Prolog is actually written *in reverse*, and with commas for the AND's (\wedge), and “:-” for implication, as follows:

`b :- a1, a2, a3.`

Note also the use of a period (.) as a terminator.

Writing Prolog Programs

The basic strategy for building a prolog program is as follows:

1. enter “rules” (things that are always true)
2. enter “facts” (data about a particular situation)
3. “query” the program (ask questions)
4. The runtime system searches for an answer

Rules can contain variables, values, ANDs, and the implication symbol (:-). Variables start with an uppercase letter, string values start with a lowercase letter. The scope of a variable is extremely short – just the rule it is in! Compound terms can occur on either side of the implication, such as:

`born(terry, 1983).`

Here is a simple Prolog program that contains two rules and three facts:

```
person(X) :- mother_of(Y,X).  
person(Y) :- mother_of(Y,X).  
mother_of(mary, carmen).  
mother_of(mary, alexander).  
mother_of(mary, andre).
```


} *rules*
} *facts*

In the program above, the intent of the first rule is to indicate that anyone who *has* a mother is a person, and the second rule indicates that anyone who *is* a mother is a person. (Actually, there is no way to know, for certain, that the first parameter of “mother_of” is the mother, and the second is the child - it could be the other way around. The programmer needs to decide the meanings of parameters, document them, and stay consistent.)

Note that because of the Prolog scope rule, that variables X and Y in the first rule are different variables than the X and Y in the second rule.

Now let's try querying the above program. Most Prolog systems provide a prompt for interactive querying – often a question mark. In these examples, the user's query is shown in red, Prolog's response is in blue:

```
? - person(carmen) .
true
? - person(andre) .
true
? - person(mary) .
true
? - person(herb) .
false
```

 *Note: some Prolog systems will list this answer three times!*

The reason that the query `person(herb)` returns “false”, is not because Prolog proved that herb wasn't a person – it is because Prolog failed to prove that herb is a person. That is an important point: Prolog tries to prove your query true; if it can, it answers “true”. If it can't it answers “false”.

What distinguishes a “rule” from a “fact”? Well, in truth, Prolog makes no distinction. However, it is a very useful programming technique to organize the code in such a manner. Usually, facts are rules that contain no variables and no implication, but there are exceptions.

When a *query* contains a variable, Prolog tries to find a value for the variable that would make the query true. Here are some examples:

```
? - mother_of(Mom, carmen) .
Mom = mary
? - mother_of(mary, Kid) .
Kid = carmen
Kid = alexander;
Kid = andre;
? - person(P) .
P = carmen
P = alexander;
P = andre;
P = mary;
```

Prolog tries to find all possible answers to a query. After giving the first answer, some runtime systems (such as SWI-Prolog) require the user to specify if they want to see more answers. In SWI-Prolog, the user does this by typing a semi-colon (;). The semi-colons are shown in red in the above example, because they are entered by the user.

Here is a different set of facts and rules for deducing family relationships:

```
parent(carmen, herb, mary).
parent(alexander, herb, mary).
parent(andre, herb, mary).
female(carmen).
male(alexander).
male(andre).
sister(X,Y) :- female(X), parent(X,F,M), parent(Y,F,M).
```

In this program, parent facts are intended to be: **parent(child,father,mother)**, and the **sister** rule to be true if *X is a sister of Y*. (note that is different than saying *X and Y are sisters*, which requires both X and Y to be female).

Now let's try some queries:

```
? - sister(carmen, alexander).
true
? - sister(alexander, carmen).
false
? - sister(carmen, carmen).
true
```

Oops, we probably didn't intend for someone to be their own sister. At first, it might seem that this shouldn't happen, since X and Y are different variables. But here, X=carmen and Y=carmen both satisfy the requirements for sister(X,Y). We can easily fix this bug as follows:

```
sister(X,Y) :- female(X), parent(X,F,M), parent(Y,F,M), X/=Y.
```

Numeric computations can be done in Prolog. The assignment operator is the word "is". Here is a program that computes factorial:

```
factorial(1,1).
factorial(N,Res) :- N1 is N-1, factorial(N1,Res2), Res is Res2*N.
```

Here is a query that asks query to find the factorial of 5:

```
? - factorial(5,F).
F = 120
```

This example illustrates that compound terms are not the same as functions. They don't "return" values – any desired output must be bound to a variable in the query, and the program must be written to do that. (sometimes such terms are called "functors".)

Prolog's Search Mechanism

Prolog's process for answering queries is a somewhat simplified version of a logical inference technique called *resolution* and *unification*. "Resolution" refers to matching the left and right sides of an implication, cancelling out a term on both sides. "Unification" refers to the binding of variables to values.

Here is an example program, along with a trace of what happens during a query. (taken from Louden & Lambert, "Programming Languages" ©2012)

```
[1] ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
[2] ancestor(X,X).
[3] parent(amy,bob).
    ? - ancestor(H,bob).
```

Prolog starts by placing the goal on the righthand side of an implication:

```
:- ancestor(H,bob).
```

Prolog now searches the rules list for anything with a compatible lefthand side.

It finds rule #1, adds all of its elements to the goal, unifies variables, and resolves the two sides:

```
ancestor(X,Y) :- ancestor(H,bob), parent(X,Z), ancestor(Z,Y).      [**]
```

```
ancestor(H,bob) :- ancestor(H,bob), parent(H,Z), ancestor(Z,bob).
```

```
ancestor(H,bob) :- ancestor(H,bob), parent(H,Z), ancestor(Z,bob).
```

```
:- parent(H,Z), ancestor(Z,bob).
```

Prolog now has TWO goals to satisfy. It starts a new search on *parent(H,Z)*, and **finds rule #3**:

```
parent(amy,bob) :- parent(H,Z), ancestor(Z,bob).
```

```
parent(amy,bob) :- parent(amy,bob), ancestor(bob,bob).
```

```
:- ancestor(bob,bob)
```

Prolog starts a new search on *ancestor(bob,bob)*, and **finds rule #1**. [*]

We use (') to differentiate different instances of variables named X, Y, and Z.

```
ancestor(X',Y') :- ancestor(bob,bob), parent(X',Z'), ancestor(Z',Y').
```

```
ancestor(bob,bob) :- ancestor(bob,bob), parent(bob,Z'), ancestor(Z',bob).
```

```
:- parent(bob,Z'), ancestor(Z',bob).
```

Prolog starts a new search on *parent(bob,Z')*. This search fails.

So it backtracks and continues the most recent incomplete search (indicated above with [*]).

It continues that search, and **finds rule #2**.

```
ancestor(X'',X'') :- ancestor(bob,bob).
```

```
ancestor(bob,bob) :- ancestor(bob,bob).
```

```
:- Here, Prolog reports "true", and unifications for variables in the original query: H=amy
```

If the user enters a ":", Prolog backtracks and continues searching for more answers.

Here, it backtracks to the most recent incomplete search (indicated above with [*]).

It continues that search, and **finds rule #2**.

```
ancestor(X''',X''') :- ancestor(H,bob).
```

Note that X'''=H=bob

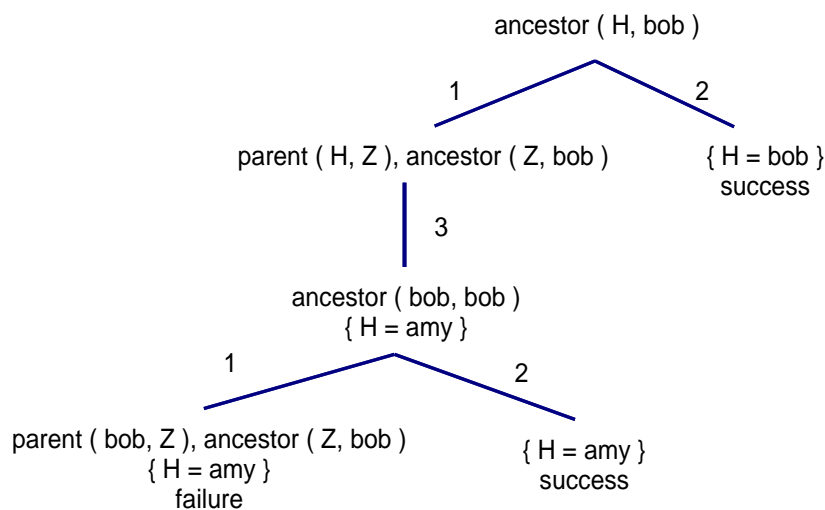
```
ancestor(bob,bob) :- ancestor(bob,bob).
```

```
:- and again, Prolog reports "true", this time with H=bob
```

At each step, when Prolog combines the goal with a compatible rule, it must unify the corresponding terms. The rules for unification are:

- If the two terms are values, they must be the same value.
- If one term is a variable and the other is a value, it unifies them by binding the variable to the value.
- If both terms are variables, they become aliases of each other.

The example search given on the previous page can be summarized in tree form, showing decisions, bindings, and results, as follows:



List Structures in Prolog

Prolog offers the same list functionality as Scheme and LISP, but with a very different (and as we shall see, rather clever) notation. In Prolog, lists are surrounded by square brackets `[]`, and elements in the list are separated by commas. For example: `[2,6,12,9]`. Prolog lists may also include the metasymbol `|` (vertical bar), and that single symbol can be used to specify the `car`, `cdr`, or `cons`, depending on where it is used.

For example, when used directly, it can perform a `cons` operation:

```

[3 | [6,2]] produces [3,6,2]
[3 | [ ] ] produces [3]
  
```

However, if used in a query, the unification process can result in the same symbol effectively extracting the `car` and `cdr` from an existing list.

For example, consider the following facts, and an accompanying query:

```
favoriteNumbers(terry, [2,9,42]).  
favoriteNumbers(alex, [3,18,30,65]).  
? - favoriteNumbers(terry, [X|Y]).
```

Prolog responds with `X=2, Y=[9,42]`. That is, `X` and `Y` are set to the `car` and `cdr` of `[2,9,42]`. Here, the “|” decomposed the list into its `car` and `cdr` in order to unify the variables in the query.

With this notation, we can implement the list processing functions that we previously solved in Scheme, into Prolog.

Example 1 – Consider our solution to “lastElement” that we did in Scheme:

```
(define (lastElement L)  
  (if (null? (cdr L)) (car L)  
      (lastElement (cdr L))))
```

We can translate this solution into a set of Prolog rules. In general, the process for doing this is as follows:

- The various if-then-else cases become different rules
- We add an additional parameter to hold the output value

Our solution in Prolog becomes:

```
lastElement([X],X).  
lastElement([X|Y],Z) :- lastElement(Y,Z).
```

Now we can try running our program with a query, such as:

```
? - lastElement([7,8,9,3], L).  
L=3
```

Sometimes it is important to organize the rules so that the base case appears first (recall Prolog’s search process, that starts at the top and works its way downward through the rules). In the “lastElement” program, it turns out that it doesn’t matter which order the rules are listed. But as we will see, in some of the next examples placing the rules in the wrong order can lead to an infinite loop (actually, an infinite *search*).

Let’s do another example:

Example 2 - Consider our solution to “concatenate” that we did in Scheme:

```
(define (concatn8 L M)
  (if (null? L) M
      (cons (car L) (concatn8 (cdr L) M))))
```

We can translate this solution to Prolog in the same manner:

```
concatn8([],M,M).
concatn8([H|X],M,[H|T]) :- concatn8(X,M,T).
```

Now we can try running our program with a query, such as:

```
? - concatn8([7,8,9,3], [1,4,2], L).
L=[7,8,9,3,1,4,2]
```

It is often preferable to try and find Prolog solutions are as “general” as possible, meaning that they are queryable in multiple directions. For example, notice the results when we query “concatn8” in this way:

```
? - concatn8(L, M, [7,3,4,2]).
L=[], M=[7,3,4,2] ;
L=[7], M=[3,4,2] ;
L=[7,3], M=[4,2] ;
L=[7,3,4], M=[2] ;
L=[7,4,4,2], M=[]
```

Generate-and-Test

A very common design methodology for Prolog programs is called *generate-and-test*. Such a program includes both a mechanism for generating combinations of values, and another mechanism for testing to see if those values comprise a solution. It then relies on Prolog’s search strategy to search all possible combinations of potential solutions.

Consider for example the problem of finding “Pythagorean Triples”. These are values X , Y , Z , such that $X^2 + Y^2 = Z^2$. It is pretty easy to write a Prolog program that can “test” three values to see if they are a triple:

```
triple(X,Y,Z) :- W is X*X+Y*Y, V is Z*Z, W=V.
```

We can query this rule with `triple(3,4,5)` and it will properly return `true`. But querying it with something like `triple(A,B,C)`, in the hopes that it *finds* triples, fails – because it cannot compute W when X and Y are uninitialized.

We need to add code that “generates” values for X , Y , and Z .

One very simple way of generating values is to just state the possible values as facts, as follows:

```
num(1) .  
num(2) .  
num(3) .  
etc.
```

The trick now is to combine this with the “test” code we just saw, preceded by bindings for X, Y, and Z using calls to “num”:

```
num(1) .  
num(2) .  
num(3) .  
num(4) .  
num(5) .  
num(6) .  
num(7) .  
num(8) .  
num(9) .  
num(10) .  
triple(X,Y,Z) :- num(X), num(Y), num(Z),  
                  W is X*X+Y*Y, V is Z*Z, W=V.
```

Now when we submit a query such as `triple(A,B,C)`, it first tries to prove `num(A)`. It satisfies this when it finds the rule `num(1)`, thus setting A to 1. It does the same thing for Y and Z, and thus the first triple it “tests”, is the triple (1,1,1). This of course, fails the test. So it backtracks, binding Z to 2. It will test (1,1,2), then (1,1,3), and so on, eventually finding (3,4,5) which passes. And it will continue, finding all triples for values between 1 and 10.

Of course, our program will become unwieldy if, for example, we wanted to find all triples for values in the range 1 to 1000. In class we will examine some ways to write this program more efficiently.

Generate-and-test can be used to solve problems without actually having to solve them ourselves. Imagine we wanted to sort a list of numbers, but we don’t know any sorting algorithms. If we had a way of *testing* to see if a list is sorted, and a way of *generating* permutations of our list, we can use generate-and-test to have Prolog search for a sorted version. For example:

```

      generate           test
sort(S,T) :- permute(S,T) , sorted(T) .

permute([],[]) .
permute(X,[Y|Z]) :- concat(U,[Y|V],X) , concat(U,V,W) , permute(W,Z) .

sorted([]) .
sorted([X]) .
sorted([X,Y|Z]) :- X=<Y, sorted([Y|Z]) .

```

In this case, writing a Prolog program to find permutations of a list is arguably just as difficult as writing a sort algorithm! However, the concept of programming by coding how to *recognize* the answer, rather than how to *find* the answer, is partly what gives logic programming its appeal.

Unfortunately, it also gives rise to one of Prolog's drawbacks. Although the above sorting algorithm works, it suffers from terrible performance. This is because it works by generating every possible permutation of the list, resulting in performance far worse even than a naive bubble sort.

CUT (!) and FAIL

There are times when we may want to exercise more control over Prolog's search. In particular, it is often useful to be able to stop it from backtracking when we know that there cannot be any additional answers, or if the backtracking would lead to an infinite search.

The cut operator (denoted "!") can be added on the right-hand side of any rule, to indicate a spot where Prolog has already found the only correct bindings. It tells the runtime system to *not backtrack to the left of the cut*.

One use for cut is to avoid infinite backtracking. Consider our previous solution to factorial:

```

factorial(1,1) .
factorial(N,Res) :- N1 is N-1, factorial(N1,Res2), Res is Res2*N.

```


While we have seen that a query such as `?-factorial(5,X)` does work correctly (Prolog responds with `X=120`), if the user enters a `;`, Prolog enters an infinite loop, looking for factorial of 0, -1, -2, etc. There are several ways to stop this from happening – one of them is to add a cut (!) at an appropriate location in the rules, such as shown here:

```
factorial(1,1).  
factorial(N,Res) :- N1 is N-1, factorial(N1,Res2), !, Res is Res2*N.
```



Another solution is to add the cut here:

```
factorial(1,1) :- !.  
factorial(N,Res) :- N1 is N-1, factorial(N1,Res2), Res is Res2*N.
```



Cut can also be used to build a sort of if-then-else structure, as follows:

```
A :- B, !, C.  
A :- D.
```

Rule A, here, is analogous to *if A then B else C*. This is because A works by first testing B. If B is true, then C is applied. If B is not true, then rule A fails and backtracking causes the second rule A to be tried, resulting in D being applied. Under ordinary circumstances (i.e., without the cut), the second rule would be tried even if B were true. But by adding the cut, once B is proven to be true, backtracking won't occur.

Here is a very simplified example that illustrates using this methodology to determine whether to buy a laptop or a desktop computer:

```
worksOnRoad(Frank).  
worksOnRoad(Terry).  
worksInOffice(Lance).  
worksInOffice(Cary).  
purchase(Person,Computer) :- worksOnRoad(Person), !, Computer is laptop.  
purchase(Person,Computer) :- Computer is desktop.
```

Now, the query: `purchase(Terry,X)` would return “X=laptop”, while the query `purchase(Lance,X)` would return “X=desktop”. Without the cut, the query `purchase(Terry,X)` would return both “X=laptop” and “X=desktop”.

Another use of “cut” is in conjunction with another feature of Prolog: “fail”. Fail is a reserved word in Prolog that causes a goal to fail. It is used to build rules that use “negative logic”; that is, that specify conditions under which something *isn't* true, rather than conditions when something is true.

When the term “fail” is encountered, the rule immediately fails. A typical way of using this, is to pair it with other rules, at least one that succeeds.

Consider this simple example that determines who is and isn't a taxpayer:

```
taxpayer(X) :- person(X), nonResident(X), !, fail.  
taxpayer(X) :- person(X), pension(X, Income), Income < 5000, !, fail.  
taxpayer(X) :- person(X).  
person(jim).  
person(john).  
person(judy).  
nonresident(john).  
pension(judy, 8000).  
pension(jim, 1000).
```

Now, the query:

?-taxpayer(X)

would return a list of taxpayers. The strategy here is that it is far easier to specify who *doesn't* pay taxes, than it is to specify who *does* pay taxes (since it is only under rare exceptions that people don't have to pay taxes). The rules above specify the two conditions under which people don't pay taxes. It uses generate-and-test to generate all of the people, and then tests to see if they *don't* have to pay tax. Note that the cut (!) immediately before the "fail" is required, otherwise non-taxpayers would still end up becoming taxpayers.