Quick Sort

The merge sort partitions its input list rather blindly, alternating values between the two lists. What if we partitioned the input list based on values rather than positions? The quick sort does this. It selects a “pivot” value (the head of the input list) and divides the input into two sublists based on whether the values in the list are less than the pivot or greater than or equal to the pivot. Next the two sublists are recursively sorted. But now, after sorting, no merge phase is needed. Rather, the two sorted sublists can simply be appended, since we know all values in the first list are less than all values in the second list.

We need a Prolog relation that characterizes how we will do our partitioning. We define partition\((E,L1,L2,L3)\) to be true if \(L1\) can be partitioned into \(L2\) and \(L3\) using \(E\) as the pivot element. The necessary rules are:

\[
\text{partition}(E,[],[],[]) ;
\text{partition}(E,[A|T1],A|[T2],L3) :- A<E, \text{partition}(E,T1,T2,L3).
\text{partition}(E,[A|T1],L2,[A|T3]) :- A>=E, \text{partition}(E,T1,L2,T3).
\]

The first line defines a trivial partition of a null list. The second line handles the case in which the first element of the list to be partitioned is less than the pivot, while the final line handles the case in which the list head is greater than or equal to the pivot.

With our notion of partitioning defined, the quicksort itself requires only 2 lines:

\[
\text{qsort}([],[]).
\text{qsort}([A|T],L) :- \text{partition}(A,T,L1,L2), \text{qsort}(L1,S1), \text{qsort}(L2,S2), \text{append}(S1,[A|S2],L).
\]

The first line defines a trivial sort of an empty list. The second line says to sort a list that begins with \(A\) and ends with list \(T\), we partition \(T\) into sublists \(L1\) and \(L2\), based on \(A\). Then we recursively quick sort \(L1\) into \(S1\) and \(L2\) into \(S2\). Finally we append \(S1\) to \([A|S2]\) (\(A\) must be > all values in \(S1\) and \(A\) must be \(\leq\) all values in \(S2\)). The result is \(L\), a sorting of \([A|T]\).

Arithmetic in Prolog

The = predicate can be used to test bound variables for equality (actually, identity).

If one or both of =’s arguments are free variables, = forces a binding or an equality constraint.

Thus

\[
\begin{align*}
& ?- 1=2. & \text{no} \\
& ?- X=2. & X = 2 \\
& ?- Y=X. & Y = X = _10751 \\
& ?- X=Y, X=joe. & X = Y = joe
\end{align*}
\]
**Arithmetic Terms are Symbolic**

Evaluation of an arithmetic term into a numeric value must be forced. That is, \(1+2\) is an infix representation of the relation \(+ (1, 2)\). This term is not an integer!

Therefore

\[
?- 1+2=3.
\]

no

To force arithmetic evaluation, we use the infix predicate \(\text{is}\).

The right-hand side of \(\text{is}\) must be all ground terms (literals or variables that are already bound). No free (unbound) variables are allowed.

**Hence**

\[
?- 2 \text{ is } 1+1.
\]

yes

\[
?- x \text{ is } 3*4.
\]

\[
x = 12
\]

\[
?- y \text{ is } z+1.
\]

! Instantiation error in argument 2 of is/2

! goal: \(_{10712} \text{ is } _{10715}+1

The requirement that the right-hand side of an \(\text{is}\) relation be ground is essentially procedural. It exists to avoid having to invert complex equations. Consider,

\[
(0 \text{ is } (I**N)+(J**N)-K**N)), N>2.
\]

**Counting in Prolog**

Rules that involve counting often use the \(\text{is}\) predicate to evaluate a numeric value.

Consider the relation \(\text{len}(L, N)\) that is true if the length of list \(L\) is \(N\).

\[
\text{len}([], 0).
\]

\[
\text{len}([_|T], N) :-
\]

\[
\text{len}(T, M), N \text{ is } M+1.
\]

\[
?- \text{len}([1, 2, 3], X).
\]

\[
X = 3
\]

\[
?- \text{len}(Y, 2).
\]

\[
Y = [_{10903}, _{10905}]
\]

The symbols \(_{10903}\) and \(_{10905}\) are “internal variables” created as needed when a particular value is not forced in a solution.

**Debugging Prolog**

Care is required in developing and testing Prolog programs because the language is untyped; undeclared predicates or relations are simply treated as false.

Thus in a definition like

\[
\text{adj}([A, B | _]) :- \text{A} = \text{B}.
\]

\[
\text{adj}([_|B | T]) :- \text{adk}([B | T]).
\]

\[
?- \text{adj}([1, 2, 2]).
\]

no

(Quintus does warn when an undefined relation is referenced, but many other Prologs don’t).
Similarly, given

\[
\text{member}(A, [A|\_]). \\
\text{member}(A, [_|T]) :- \\
\text{member}(A, [T]). \\
| \text{?- member}(2, [1, 2]). \\
\text{Infinite recursion! (Why?)}
\]

If you're not sure what is going on, Prolog's trace feature is very handy.
The command

trace.

turns on tracing. (notrace turns tracing off).
Hence

| ?- trace. 
| yes 
| [trace] 
| ?- member(2, [1, 2]).

Termination Issues in Prolog

Searching infinite domains (like integers) can lead to non-termination, with Prolog trying every value.
Consider

\[
\text{odd}(1). \\
\text{odd}(N) :- \text{odd}(M), N \text{ is } M+2. \\
| \text{?- odd}(X). \\
X = 1 ; \\
X = 3 ; \\
X = 5 ; \\
X = 7
\]

A query

| ?- odd(X), X=2. 

going into an infinite search, generating each and every odd integer and finding none is equal to 2!
The obvious alternative, 
\[
\text{odd}(2) \text{ (which is equivalent to } X=2, \text{ odd}(X)\text{)} \text{ also does an infinite, but fruitless search.}
\]
We'll soon learn that Prolog does have a mechanism to “cut off” fruitless searches.
Definition Order can Matter

Ideally, the order of definition of facts and rules should not matter. But, in practice definition order can matter. A good general guideline is to define facts before rules. To see why, consider a very complete database of motherOf relations that goes back as far as motherOf(cain,eve).

Now we define

\begin{verbatim}
isMortal(X) :-
  isMortal(Y), motherOf(X,Y).
isMortal(eve).
\end{verbatim}

These definitions state that the first woman was mortal, and all individuals descended from her are also mortal.

But when we try as trivial a query as | ?- isMortal(eve).
we go into an infinite search!

Why?

Let’s trace what Prolog does when it sees

| ?- isMortal(eve).

It matches with the first definition involving isMortal, which is

\begin{verbatim}
isMortal(X) :-
  isMortal(Y), motherOf(X,Y).
\end{verbatim}

It sets X=eve and tries to solve

\begin{verbatim}
isMortal(Y), motherOf(eve,Y).
\end{verbatim}

It will then expand isMortal(Y) into

\begin{verbatim}
isMortal(Z), motherOf(Y,Z).
\end{verbatim}

An infinite expansion ensues.

The problem is that Clark Kent is from the planet Krypton, and hence won’t appear in our motherOf database.

Let’s trace the query.

It doesn’t match isMortal(eve).

We next try

\begin{verbatim}
isMortal(clarkKent) :-
  isMortal(Y), motherOf(clarkKent,Y).
\end{verbatim}

We try Y=eve, but eve isn’t Clark’s mother. So we recurse, getting:

\begin{verbatim}
isMortal(Z), motherOf(Y,Z), motherOf(clarkKent,Y).
\end{verbatim}

But eve isn’t Clark’s grandmother either! So we keep going further back, trying to find a chain of descendents that leads from eve to clarkKent. No such chain exists, and there is no
limit to how long a chain Prolog will try.
There is a solution though!
We simply rewrite our recursive definition to be
\( \text{isMortal}(X) :- \) motherOf(X,Y), isMortal(Y).
This is logically the same, but now we work from the individual \( X \) back toward \( \text{eve} \), rather than from \( \text{eve} \) toward \( X \). Since we have no motherOf relation involving \( \text{clarkKent} \), we immediately stop our search and answer \text{no}!

**Extra-logical Aspects of Prolog**

To make a Prolog program more efficient, or to represent negative information, Prolog needs features that have a procedural flavor. These constructs are called “extra-logical” because they go beyond Prolog’s core of logic-based inference.

**The Cut**

The most commonly used extra-logical feature of Prolog is the “cut symbol,” “!”. A ! in a goal, fact or rule “cuts off” backtracking.
In particular, once a ! is reached (and automatically matched), we may not backtrack across it. The rule we’ve selected and the bindings we’ve already selected are “locked in” or “frozen.”

For example, given
\( \text{x}(A) :- \text{y}(A,B), \text{z}(B), !, \text{v}(B,C) \).
Once the ! is hit we can’t backtrack to resatisfy \( \text{y}(A,B) \) or \( \text{z}(B) \) in some other way. We are locked into this rule, with the bindings of \( A \) and \( B \) already in place.
We can backtrack to try various solutions to \( \text{v}(B,C) \).
It is sometimes useful to have several !’s in a rule. This allows us to find a partial solution, lock it in, find a further solution, then lock it in, etc.
For example, in a rule
\( \text{a}(X) :- \text{b}(X), !, \text{c}(X,Y), !, \text{d}(Y) \).
We first try to satisfy \( \text{b}(X) \), perhaps trying several facts or rules that define the \( \text{b} \) relation. Once we have a solution to \( \text{b}(X) \), we lock it in, along with the binding for \( X \).
Then we try to satisfy \( \text{c}(X,Y) \), using the fixed binding for \( X \), but perhaps trying several bindings for \( Y \) until \( \text{c}(X,Y) \) is satisfied.
We then lock in this match using another !.
Finally we check if $d(Y)$ can be satisfied with the binding of $x$ already selected and locked in.

When are Cuts Needed?
A cut can be useful in improving efficiency, by forcing Prolog to avoid useless or redundant searches.
Consider a query like
\[
\text{member}(X,\text{list1}), \\
\text{member}(X,\text{list2}), \text{isPrime}(X).
\]
This asks Prolog to find an $x$ that is in list1 and also in list2 and also is prime.
$x$ will be bound, in sequence, to each value in list1. We then check if $x$ is also in list2, and then check if $x$ is prime.
Assume we find $x=8$ is in list1 and list2. isPrime(8) fails (of course).
We backtrack to member(X, list2) and try to resatisfy it with the same value of $x$.

But clearly there is never any point in trying to resatisfy member(X, list2).
Once we know a value of $x$ is in list2, we test it using isPrime(X).
If it fails, we want to go right back to member(X, list1) and get a different $x$.
To create a version of member that never backtracks once it has been satisfied we can use !.
We define
\[
\text{member1}(X, [X|\_]) :- !. \\
\text{member1}(X, [\_|Y]) :- \\
\text{member1}(X, Y).
\]
Our query is now
\[
\text{member}(X,\text{list1}), \\
\text{member1}(X,\text{list2}), \text{isPrime}(X).
\]
(Why isn’t member1 used in both terms?)

Expressing Negative Information
Sometimes it is useful to state rules about what can’t be true. This allows us to avoid long and fruitless searches.
fail is a goal that always fails. It can be used to represent goals or results that can never be true.
Assume we want to optimize our grandMotherOf rules by stating that a male can never be anyone’s grandmother (and hence a complete search of all motherOf and fatherOf relations is useless).
A rule to do this is
\[
\text{grandMotherOf}(X,GM) :- \\
\text{male}(GM), \text{fail}.
\]
This rule doesn't do quite what we hope it will! Why?
The standard approach in Prolog is to try other rules if the current rule fails. Hence we need some way to “cut off” any further backtracking once this negative rule is found to be applicable.
This can be done using
\[
\text{grandMotherOf}(X, GM) :- \\
\text{male}(GM), !, \text{fail}.
\]

Other Extra-Logical Operators

- assert and retract
These operators allow a Prolog program to add new rules during execution and (perhaps) later remove them. This allows programs to learn as they execute.
- findall
Called as \( \text{findall}(X, \text{goal}, \text{List}) \)
where \( X \) is a variable in \( \text{goal} \). All possible solutions for \( X \) that satisfy \( \text{goal} \) are found and placed in \( \text{List} \).
For example,
\[
\text{findall}(X, \\
(\text{append}(\_, [X|\_], [-1, 2, -3, 4]), (X<0)), \\
L).
\]
\( L = [-1, -3] \)

- var and nonvar
\text{var}(X) \text{ tests whether } \ X \text{ is unbound (free).}
\text{nonvar}(Y) \text{ tests whether } \ Y \text{ is bound (no longer free).}
These two operators are useful in tailoring rules to particular combinations of bound and unbound variables.
For example, the rule
\[
\text{grandMotherOf}(X, GM) :- \\
\text{male}(GM), !, \text{fail}.
\]
might backfire if \( GM \) is not yet bound. We could set \( GM \) to a person for whom \( \text{male}(GM) \) is true, then fail because we don't want grandmothers who are male!

To remedy this problem, we use the rule only when \( GM \) is bound. Our rule becomes
\[
\text{grandMotherOf}(X, GM) :- \\
\text{nonvar}(GM), \text{male}(GM), !, \text{fail}.
\]