Week 13: Logic Programming

Most computation is directed, going from input to output.

In functional programming, this is made very explicit, since input = function argument and output = function result.

We already saw one exception, in constraint solving.

There, we defined a set of relations, which the computer could "solve" in several directions.

Logic programming adds to this relational programming paradigm two ideas:

- The idea that a solution is found by a *search* which may try several alternatives.
- A kind of symbolic pattern matching called *unification*.

Prolog

The most widespread logic programming language is *Prolog*.

Prolog is an acronym for "Programming in Logic".

It was developed in the 70's by Alain Colmerauer, originally for natural language parsing.

Prolog is used in artificial intelligence applications, such as expert systems, knowlegde bases, natural language parsing.

Like Lisp, Prolog is a small language with simple syntax and no static types.

Example: *append*

The *append* function is written in Scala as follows.

```
def append[a](xs: List[a], ys: List[a]): List[a] = xs match {
  case Nil ⇒ ys
  case x :: xs1 ⇒ x :: append(xs1, ys)
}
```

This function can be regarded as a translation into Scala of the following two rules.

1. For any list *ys*, appending the empty list and *ys* gives *ys*.
2. For any *x*, *xs1*, *ys*, *zs*, if appending *xs1* and *ys* yields *zs*, then appending *x* :: *xs1* to *ys* yields *x* :: *zs*.

In Prolog these two rules can be written as follows.

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


Remarks:

- Variables and parameters in Prolog start with an upper case letter, e.g. *X*, *Xs*, *Ys*.
- [...|...] is list cons, e.g. [*X*|*Xs*] is *X* :: *Xs* in Scala.

Predicates

append is called a *predicate* in Prolog.

Roughly, a predicate is a procedure which may succeed or fail.

Note what was the "function result" in Scala now becomes an additional parameter.

Clauses

Predicates are defined by *clauses*, which can be axioms and rules.

- $append([\,], Ys, Ys)$ is an axiom; it states that appending $[\,]$ and Ys yields Ys (for any Ys).
- $append([X|Xs], Ys, [X|Zs])$:– $append(Xs, Ys, Zs)$ is a rule; it states that appending $[X|Xs]$ and Ys yields $[X|Zs]$, *provided* appending Xs and Ys yields Zs (for any X, Xs, Ys, Zs).

Hence, :– can be read as backwards implication $\Leftarrow$.

Queries

A Prolog query is a predicate which can contain variables as parameters.

The Prolog interpreter will attempt to find assignments to the variables that make the predicate true.

For instance, the call of $append(List(1), List(2, 3))$ in Scala would be modelled by

$$append([1], [2, 3], X)$$

in Prolog. It would lead to the answer $X = [1, 2, 3]$.

But it is also possible to put variables in other places.

For instance,

- $append(X, [2, 3], [1, 2, 3])$ yields $X = [1]$.
- $append([1, 2], Y, [1, 2, 3])$ yields $Y = [3]$.

- $append(X, Y, [1, 2, 3])$ yields several solutions:

  $X = [\,], Y = [1, 2, 3]$, or
  $X = [1], Y = [2, 3]$, or
  $X = [1, 2], Y = [2]$, or
  $X = [1, 2, 3], Y = [\,]$.

- $append([1], Y, Z)$ yields a solution template which contains a variable: $Y = X, Z = [1|X]$.

This strategy, when it works, can be very flexible.

It closely resembles query languages for databases.

Indeed, Prolog is often used as a language for information retrieval over some database in particular when deduction is also necessary.

Deductive Information Retrieval

Here is a little database of a family tree.

$female(mary)$.	$married(fred, mary)$.
$female(ann)$.	$married(peter, elaine)$.
$female(elaine)$.	$married(sue, alfred)$.
$female(jane)$.	$married(alfred, ann)$.
$female(susan)$.	
$child(bob, fred)$.	$child(sue, mary)$.
$child(bob, mary)$.	$child(jane, sue)$.
$child(peter, fred)$.	$child(jane, alfred)$.
$child(peter, mary)$.	$child(jessica, alfred)$.
$child(sue, fred)$.	$child(jessica, ann)$.
	$child(paul, jane)$.

We can access the information in the database in the tree by asking queries.

A query starts with a question mark, followed by a predicate.

Here is a transcript of a session with a little prolog interpreter (which was written in Scala):

$prolog\!> \ ?child\,(bob,\ fred\,).$
yes
$prolog\!> \ ?child\,(bob,\ bob\,).$
no
$prolog\!> \ ?child\,(bob,\ X\,).$
$[X = fred\,]$
$prolog\!> \ ?more.$
$[X = mary\,]$
$prolog\!> \ ?more.$
no
$prolog\!> \ ?child\,(X,\ bob\,).$
no

The special query *?more.* asks for further solutions to the previous query.

One can also define rules to derive facts that are not coded directly in the database.

For instance:

$prolog\!> \ sibling\,(X,\ Y\,) \ :- \ child\,(X,\ Z\,),\ child\,(Y,\ Z\,).$

Then:

$prolog\!> \ ?sibling\,(peter,\ bob\,).$
yes
$prolog\!> \ ?sibling\,(bob,\ jane\,).$
no
$prolog\!> \ ?sibling\,(bob,\ X\,).$
$[X = peter\,]$
$prolog\!> \ ?more.$
$[X = susan\,]$
$prolog\!> \ ?more.$
$[X = bob\,]$
$prolog\!> \ ?more.$
$[X = peter\,]$
$prolog\!> \ ?more.$
$[X = susan\,]$
$prolog\!> \ ?more.$
no

Question: Why does each sibling appear twice in the solutions?

The previous query is not quite what we want, because *bob* is a sibling of himself.

This can be corrected by defining:

$prolog\!> \ sibling\,(X,\ Y\,) \ :- \ child\,(X,\ Z\,),\ child\,(Y,\ Z\,),\ not\,(same\,(X,\ Y\,)\,).$

Here, the predicate *same* is simply defined as

$same\,(X,\ X\,).$

The operator *not* is special (and somewhat problematic!) in Prolog.

$not\,(P\,)$ succeeds if the original predicate P fails.

For instance, to define that a person is male, we can use:

$male\,(X\,) \ :- \ not\,(female\,(X\,)\,).$

Recursive Rules

Rules can also be recursive.

For instance to define that X is an ancestor of Y:

$parent\,(X,\ Y\,) \ :- \ child\,(Y,\ X\,).$
$ancestor\,(X,\ Y\,) \ :- \ parent\,(X,\ Y\,).$
$ancestor\,(X,\ Y\,) \ :- \ parent\,(X,\ Z\,),\ ancestor\,(Z,\ Y\,).$

The possibility to define recursive rules distinguishes logic programming from database query languages.

Exercise: Define the predicate that X is an uncle of Y.

Exercise: Define the predicate that X and Y are cousins.

Implementation of Prolog

The Prolog interpreter seems to have quite a bit of "intelligence".

We will now uncover how this is achieved.

Essentially, there are two main ingredients:

- A mechanism for pattern matching which is based on unification.
- A mechanism for searching derivations.

Representation of Terms

We can represent Prolog terms by a class *Term* with two subclasses *Var* for variables and *Constr* for constructors.

```
abstract class Term {
  def tyvars: List[String] = ...
  def map(s: Subst): Term = ...
}
case class Var(a: String) extends Term;
case class Constr(a: String, ts: List[Term]) extends Term;
```

For instance, the variable X is represented as *Var("X")*.

Or, the term *cons(X, nil)* is represented as

Constr("cons", List(Var("X"), Constr("nil", List())))

Prolog also has syntactic sugar for list terms, but this can be expanded out as follows.

$[\,]$	$= nil$
$[S \mid T]$	$= cons(S, T)$
$[T_1, \ldots, T_n]$	$= cons(T_1, \ldots\ cons(TS_n, nil) \ldots\ )$

The class *Term* defines two methods.

- *tyvars* returns the list of all names of type variables in the term.
- *map* applies a substitution to a term (see below).

Simple Pattern Matching

When faced with a query such as *?child(peter, X)*, the interpreter tries to find a fact in the database which *matches* the query.

Matching means: Assign terms to the variables in the query so that query and fact become equal.

In our example, the assignments $[X = fred]$ or $[X = mary]$ would do, since *child(peter, fred)* and *child(peter, mary)* are facts in the database.

Assignments to variables (or: substitutions) are modelled by lists of bindings.

Each binding associates a variable name to a term:

```
type Subst = List[Binding];
case class Binding(name: String, term: Term);
```

We can define a function *lookup* that searches for the binding of a given name in a substitution:

```
def lookup(s: Subst, name: String): Option[Term] = s match {
  case List() => None
  case b :: s1 => if (name == b.name) Some(b.term)
                  else lookup(s1, name)
}
```


Substitutions as Mappings

The function *map* applies a substitution to a term.

This is defined as follows.

```
class Term {
  def map(s: Subst): Term = match {
    case Var(a) = lookup(s, a) match {
      case Some(b) => b map s
      case None => this;
    }
    case Constr(a, ts) => Constr(a, ts map (t => t map s))
  }
  ...
}
```

That is, substitutions can be seen as idempotent mappings over terms, which map all but a finite number of variables to themselves.

The *match* Functions

Now, we can implement the match algorithm by a pair of functions.

Here's the first one:

```
def match(pattern: Term, term: Term, s: Subst): Option[Subst] =
  Pair(pattern, term) match {
    case Pair(Var(a), _) => lookup(s, a) match {
      case Some(term1) => match(term1, term, s)
      case None => Some(Binding(a, term) :: s)
    }
    case Pair(Constr(a, ps), Constr(b, ts)) if a == b =>
      match(ps, ts, s)
    case _ => None
  }
```


Explanations: The *match* function takes as arguments:

- a pattern (i.e. a term with variables in it),
- a term (which is assumed to have no variables in it), and
- a substitution which contains the assignment to variables that was determined so far.

If the match succeeds, it yields as result *Some*(*s*) where *s* is a substitution.

If the match fails, the function yields as result the constant *None*.

The *match* algorithm proceeds by a pattern match over the pair of pattern and term.

- If the pattern is a variable, we need to check first whether the variable has already been assigned.
- If yes, we need to continue matching with the term that was assigned to the variable.
- If no, we extend the substitution with a binding that associates the variable's name with the term.

- If both pattern and term have the same constructor at the top level, the pattern match proceeds recursively with their elements, using the second match procedure.

Exercise: Implement the second pattern matching procedure, whose signature is:

```
def match (patterns: List[Term], terms: List[Term], s: Subst)
    : Option[Subst]
```

This procedure should return *Some* (*s1*) where *s1* is a substitution that extends *s* and that matches corresponding patterns with terms.

It should return *None* if no such substitution exists, or if the two lists have different lengths.

Unification

The pattern matching algorithm works fine for retrieving facts, but fails for rules.

Rules contain left hand sides (or: heads) which have themselves variables in them.

To match against a rule, we have to assign to variables in both the head of the rule and the query.

For instance, given the rule

$$sibling(X, Y) :- child(X, Z), child(Y, Z), not(same(X, Y)).$$

and the query $?sibling(peter, Z)$, we have to match $sibling(X, Y)$ against $sibling(peter, Z)$, which leads to the assignment $[X = peter, Y = Z]$ (or: $[X = peter, Z = Y]$).

The pattern matching algorithm needs to be generalized so that it becomes symmetric.

The resulting algorithm is called *unification*.

Unifying two terms x and y means finding a substitution s so that x *map* s and y *map* s become equal.

Example: Here are some examples of unification.

$$\begin{array}{ll}
unify(sibling(peter, Z), sibling(X, Y)) & = [X = peter, Z = Y] \\
unify(same(X, X), same(mary, Y)) & = [X = mary, Y = mary] \\
unify(cons(X, nil), cons(X, Y)) & = [Y = nil] \\
unify(cons(X, nil), cons(X, a)) & = \langle failure \rangle \\
unify(X, cons(1, X)) & = \langle failure \rangle
\end{array}$$

The last case is a bit subtle. Here, unification fails because there is no finite term T such that $T = cons(1, T)$.

However, there is an *infinite* term which satisfies the equation, namely the term that represents an infinite list of *1*'s.

Normal Prolog interpreters compute only with finite terms consisting of variables and constructors (so-called *Herbrand terms*, after the logician Jacques Herbrand (1908-1931)).

Implementation of *unify*

```
def unify (x: Term, y: Term, s: Subst): Option[Subst] = Pair(x, y) match {
  case Pair(Var(a), Var(b)) if a == b ⇒
    Some(s)
  case Pair(Var(a), _) ⇒ lookup(s, a) match {
    case Some(x1) ⇒ unify(x1, y, s)
    case None ⇒ if (y.tyvars contains a) None
                else Some(Binding(a, y) :: s)
  }
  case Pair(_, Var(b)) ⇒ lookup(s, b) match {
    case Some(y1) ⇒ unify(x, y1, s)
    case None ⇒ if (x.tyvars contains b) None
                else Some(Binding(b, x) :: s)
  }
  case Pair(Constr(a, xs), Constr(b, ys)) if a == b ⇒
    unify(xs, ys, s)
  case _ ⇒ None
}
```

As for pattern matching, we implement an *incremental* version of *unify*, where an intermediate substitution is passed in as a third parameter.

The main changes w.r.t. pattern matching are:

- We now have to test the case where both sides are the same variable. In this case, unification succeeds with the given substitution.
- The case where one side is a variable has been duplicated to make it symmetric.
- There is a check that variables do not appear in the terms they are bound to, in order to prevent infinite trees.

  (this is often called the *occurs-check*).

Complexity of Unification

Without the occurs check, the complexity of unification is linear in the size of the two terms to be unified.

This might be surprising, because the size of the result of the unification can be exponential in the size of its terms!

Example: Unifying

$$seq\,(X1,\ b\,(X2,\ X2),\ X2,\ d\,(X3,\ X3)\,)$$
$$seq\,(a\,(Y1,\ Y1\,),\ Y1,\ c\,(Y2,\ Y2),\ Y2)$$

yields

$$\begin{aligned} seq\,(&a\,(b\,(c\,(d\,(X3,\ X3),\ d\,(X3,\ X3)\,),\ c\,(d\,(X3,\ X3),\ d\,(X3,\ X3)\,)\,),\\ &\qquad b\,(c\,(d\,(X3,\ X3),\ d\,(X3,\ X3)\,),\ c\,(d\,(X3,\ X3),\ d\,(X3,\ X3)\,)\,)\,),\\ &b\,(c\,(d\,(X3,\ X3),\ d\,(X3,\ X3)\,),\ c\,(d\,(X3,\ X3),\ d\,(X3,\ X3)\,)\,),\\ &c\,(d\,(X3,\ X3),\ d\,(X3,\ X3)\,),\\ &d\,(X3,\ X3)\,) \end{aligned}$$

It is easy to see that, as the sequence is extended, the first term in the unifier grows exponentially with the length of the sequences.

However, unification is still linear, because it shares trees instead of copying them.

Complexity of the occurs check

With the occurs check as implemented, unification becomes worst-case exponential in the size of its input, because subtrees are traversed multiple times.

By marking subtrees that have already been visited, one can speed up the occurs check so that it is linear in the graph-size of the term visited.

Hence, unification becomes a quadratic algorithm.

It is possible to do even better, and make unification an $O(n \log n)$ algorithm.

...And the Sad Reality

Unfortunately, most Prolog interpreters simply leave out the occurs check in the interest of efficiency.

This would be OK, if the universe of discourse was simply extended to infinite trees.

This is the approach taken by a successor of standard Prolog called Prolog 3 (also developed by Colmerauer).

But most standard Prolog interpreters behave unpredictably when a variable is assigned a term that contains the variable itself.

For instance, they might go into infinite loops.

Furthermore, different interpreters behave in different ways.

Summary

Logic programming searches for solutions to queries, given a program consisting of facts and rules.

Computation need not to be directed; often a rule can be used in several ways with input and output parameters.

An interpreter for logic programming tries to apply rules to solve a query, using unification as the basic inference step.

Still to come next week:

- How to search for solutions.
- How all this is (not?) related to logic.