Addendum: A Simple Horn Clause Interpreter

- We construct an interpreter for the core of Prolog, with Horn clauses and logic variables.
- The interpreter is based on unification.
- The interpreter produces all solutions, one by one.
- Necessary preliminary: streams

Streams

A stream is a list which is constructed lazily.

These can be constructed simply with let.

Delay evaluation of a cons until the a field of the list object is selected.

Example: The list of all integers starting from n:

```
def ints (n) = {
  let l = n :: ints (n+1)
  l
}
```

((::) is infix Cons.)

Example: Prime numbers

As a more involved example we construct a list of all prime numbers:

```
let primes: List[Int] = {

  def primesFrom (n: Int) = {
    let primes' = primes (n + 2)
    var ps: List[Int] := primes
    while (ps.head * ps.head < n && n % ps.head != 0)
      ps := ps.tail
    if (p.head * p.head < n) primes'
    else n :: primes'
  }

  2 :: primesFrom (3)
}
```


Example: A lazy append function

Task: Given two streams, construct an append function which concatenates the two streams.

Evaluate as lazily as possible.

```
def appendStream [a] (xs: List[a], ys: List[a]): List[a] = {
  let result = xs.match {
    def Nil = ys
    def Cons (x, xs1) = List.Cons (x, appendStream (xs1, ys))
  }
  result
}
```

PROLOG Programs

Symbols:

- Term symbols f, g, h.
- Predicate symbols p, q.
- Variables X, Y, Z.

Syntax:

Terms:	S, T	$::=$	$f(T_1, \ldots, T_n) \mid X$
Predicates	P, Q	$::=$	$p(T_1, \ldots, T_n)$
Clauses	C	$::=$	$P \leftarrow Q^*$
Program			C^*
Query			P^*

Representation of PROLOG Programs

We merge term symbols and predicate symbols. Hence, predicates and terms become the same.

Terms and predicates are represented by an algebraic data type with two cases (use the usual encoding in terms of match):

```
class Term = {
  case VAR (name: String)
  case CON (ts: List[Term])
  static newVar: Term = ...
}
```

We also need a function newVar to create a new variable.

This is all completely analogous to the situation where we unified types.

Clauses are pairs of left and right hand sides.

We also need a method newInstance which renames all variables in the current clause with fresh variable names, yielding a new clause.

```
type Clause = {
  def lhs: Term
  def rhs: List[Term]
  def newInstance: Clause
}
```

Queries are lists of terms.

The PROLOG Interpreter Interface

A PROLOG interpreter tries to find all substitutions which satisfy a query wrt to a program.

Hence, we want to write a function:

```
def interprete (query: List[Term], clauses: List[Clause]): List[Subst] = {
  ...
}
```

Failure is signalled by returning the empty list of substitutions.

Contrast:

- Return empty list = failure
- Returning single-element list with empty substitution = success.

At any point in time, some variables will already be instantiated by a substitution.

So it makes sense to define a local function solve in interprete which solves the given query with the current substituion.

```
def interprete (query: List[Term], clauses: List[Clause]): List[Subst] = {
    def solve (query: List[Term], s: Subst): List [Subst] = {
        ...
    }
    solve (query, Subst.empty)
}
```

How should solve be implemented?

Step through list of predicates in query and try to match each one against all clauses:

```
def solve (query: List[Term], s: Subst): List [Subst] = {
    query.match {
        def Nil =
            s :: Nil
        def Cons (q, query') =
            val c <- clauses
            val s <- match (q, c.newInstance, s)
            solve (query', s)
    }
```


Meaning of val <– ...

The construct

```
val x <- xs ; E
```

means:

- Step through list the xs and perform E for each of its elements. The result of E should be a list.
- Concatenate all the lists resulting from E together.
- Do all this lazily, so that the lists will not be constructed unless their elements are demanded.
- Implementation will be discussed later.

Implementation of match

The match function tries to match a query predicate with the left hand-side of a clause (using unification).

If the match succeeds, match tries to solve the query consting of the right-hand side of the clause.

Otherwise, match fails.

Implementation:

```
def match (q: Term, c: Clause, s: Subst): List [Subst] = {
    val s' <- unify (q, c.lhs, s)
    solve (c.rhs, s')
}
```

Unification

We assume a unification algorithm which signals failure by returning an empty substitution.

In case of success, unify returns a list consisting of a single element, namely the most general unifier substitution.

Signature:

```
def unify (t1: Term, t2: Term, s: Subst): List[Subst] = ...
```


Definition of val <− ...

val <− ... is not built into Funnel, but is expanded using operator overloading.

Normal case of operator overloading:

> A + B is interpreted as A.+ B.
>
> That is, we look for a method named + in A, and pass the result of B as parameter.

val case of operator overloading:

> **val** x <− A ; B is interpreted as A.<−(x|B).
>
> That is, we look for a method named <− in A and pass the anonymous function (x | B) as parameter.

Instead of + and <− any other operator name can be used.

Implementation of <− for lists

```
class List [a] = {
    ...
    def <− [b] (k: a → List[b]) = this.match {
        def Nil =
            Nil
        def Cons (x, xs) = {
            val ys = k (x)
            let zs = xs.<− (k)
            appendStream (ys, zs)
        }
    }
}
```


Putting it all together:

```
def interprete (query: List[Term], clauses: List[Clause]): List[Subst] = {
    def solve (query: List[Term], s: Subst): List [Subst] = {
        query.match {
            def Nil =
                s :: Nil
            def Cons (q, query’) =
                val c <− clauses
                val s <− match (q, c.newInstance, s)
                solve (query’, s)
        }
    }
    def match (q: Term, c: Clause, s: Subst): List [Subst] = {
        val s’ <− unify (q, c.lhs, s)
        solve (c.rhs, s’)
    }
    solve (query, Subst.empty)
}
```