Declarations in λProlog

We consider four declarations now

```
kind
type
infixl
infixr
```

and present two more later.

```
localkind
local
```

It is likely that various implementations of λProlog will add various other declarations.

Each declaration associates to a token or list of tokens some expression, either a *kind expression*, a *type expression*, or an infix priority.

Built-in Types

Primitive Types

```
kind  int     type.   % integers
kind  real    type.   % real numbers
kind  string  type.   % strings
kind  o       type.   % formulas
```

Types Constructors

```
kind  ->     type -> type -> type.  % function types
```

Common Definable Type Constructors

```
kind pair  type -> type -> type.
type pr     A -> B -> pair A B.

kind   list    type -> type.
type   nil     list A.
type   ::      A -> list A -> list A.
infixr ::      5.
```

Except for the pair type, these declarations are located in some system module and need not be entered by the user.

Priorities in Infix Declarations

Higher priority means that binding is tighter.

Higher priority symbols appear *lower* in a parse tree.

```
a + b * c          =        a + (b * c)
  7   8


      +                    *
     / \                  / \
    a   *                +   c  (wrong)
       / \              / \
      b   c            a   b
```

```
p :- N is M + N * P , X = Y :: L
   1    4   7   8   2   4    5
```

```
(p :- ((N is (M + (N * P))) , (X = (Y :: L))))
```

Types in λProlog

On the surface, types in λProlog resemble types in ML. There are important differences however.

Minor: Type constructors in λProlog are prefix operators while in ML they are postfix operators.

```
member : A -> list A -> o.
member : 'a -> 'a list -> bool.
```

Minor: Types in λProlog are almost always written in curried form while in ML types are often written in uncurried form. Compare:

```
member : A -> list A -> o.
member : ('a * 'a list) -> bool.
```

Major: Types in λProlog describe syntactic domains while in ML types describe semantics domains (the distinction between these domains is not always immediate).

For example, both languages have the type `int`.

In ML, the intended meaning of `int` is the collection of integers.

In λProlog, the intended meaning is the set of *integer expressions*.

in ML: `5 = 2 + 3` returns `true`.

in λProlog: `?- 5 = 2 + 3` fails to be provable.

The Syntax of Formulas

```
type  true              o.
type  ,                 o -> o -> o.
type  :-, =>, ; , &     o -> o -> o.

infixr  ,  2.
infixr  &  2.
infixr  ;  3.
infixr  :- 1.
infixr  => 4.

type  pi, sigma    (A -> o) -> o.
```

Notice that there are two conjunctions — `,` `&` — and two implications — `:-` `=>`.

This duplication is adopted because:

(1) It aids readability since certain occurrence of these connectives have different operational reading. The convention is to use `&` and `:-` at the top-level of program clauses and to use `,` and `=>` at the top-level of goal formulas.

(2) In recent work on how λProlog can be "refined" using linear logic, the behavior of these two conjunctions and implications diverge: in the linear linear logic setting they are different.

$$p \wedge q \supset r \qquad\qquad p \wedge (p \supset q) \wedge (q \supset r) \supset r$$

```
r :- p, q.        r :- p, p => q, q => r.
p & q => r.       p & (q :- p) & (r :- q) => r.
```

Signatures

A *signature* is an ordered list of kind, type, and infix declarations for which the following restrictions apply.

1. If a token is given a kind declaration in this list, that declaration must appear prior to any use of it in a type declaration.
2. If a token is given an infix declaration, that token must be given a type declaration in the signature prior to the infix declaration. Also, the declared type must have at least two argument types.
3. A token cannot be given two kind declarations, two type declarations, or two infix declarations. A token may be given both a kind and type declaration. If such a token also has an infix declaration associated with it, that parsing directive applies only to the token given a type declaration.

Signatures are often denoted using the symbol Σ.

A Σ-*term of type* τ is a closed expression of type τ all of whose non-logical constants come from Σ.

A Σ-*formula* is a Σ-term of type o.

The Syntax of Quantification

Let Σ be the following simple signature.

```
kind  i      type.
type  a      i.
type  p, q   i -> o.
```

`q x => p y` is a $\Sigma \cup \{x : i, y : i\}$-formula.

`y\(q x => p y)` is a $\Sigma \cup \{x : i\}$-term of type `i -> o`.

`sigma y\(q x => p y)` is a $\Sigma \cup \{x : i\}$-formula.

`x\(sigma y\(q x => p y))` is a Σ-term of type `i -> o`.

`pi x\(sigma y\(q x => p y))` is a Σ-formula.

This last formula would be written in mathematical notion as

$$\forall x \exists y (qx \supset px).$$

Order of Formulas

Clausal order.

$$
\begin{array}{rcl}
\mathit{clausal}(A) & = & 0 \quad \text{provided } A \text{ is atomic or } \top \\
\mathit{clausal}(B_1 \wedge B_2) & = & \max(\mathit{clausal}(B_1), \mathit{clausal}(B_2)) \\
\mathit{clausal}(B_1 \vee B_2) & = & \max(\mathit{clausal}(B_1), \mathit{clausal}(B_2)) \\
\mathit{clausal}(B_1 \supset B_2) & = & \max(\mathit{clausal}(B_1) + 1, \mathit{clausal}(B_2)) \\
\mathit{clausal}(\forall x.B) & = & \mathit{clausal}(B) \\
\mathit{clausal}(\exists x.B) & = & \mathit{clausal}(B)
\end{array}
$$

Combined clausal and quantificational order.

$$
\begin{array}{rcl}
\mathit{order}(A) & = & 0 \quad \text{provided } A \text{ is atomic or } \top \\
\mathit{order}(B_1 \wedge B_2) & = & \max(\mathit{order}(B_1), \mathit{order}(B_2)) \\
\mathit{order}(B_1 \vee B_2) & = & \max(\mathit{order}(B_1), \mathit{order}(B_2)) \\
\mathit{order}(B_1 \supset B_2) & = & \max(\mathit{order}(B_1) + 1, \mathit{order}(B_2)) \\
\mathit{order}(\forall x : \tau.B) & = & \max(\mathit{ord}(\tau) + 1, \mathit{order}(B)) \\
\mathit{order}(\exists x : \tau.B) & = & \max(\mathit{ord}(\tau), \mathit{order}(B))
\end{array}
$$

First-Order Horn Clauses

$$\begin{aligned} G &::= \quad A \mid G \wedge G \\ D &::= A \mid G \supset A \mid \forall x\ D \end{aligned} \tag{1}$$

That is, goal formulas are conjunctions of atomic formulas and program clauses are of the form

$$\forall x_1 \ldots \forall x_m [A_1 \wedge \ldots \wedge A_n \supset A_0]$$

for $m, n \geq 0$. (If $m = 0$ then we do not write any universal quantifiers, and if $n = 0$ then we do not write the implication.)

A richer formulation is given by the following definition.

$$\begin{aligned} G &::= \top \mid A \mid G \wedge G \mid G \vee G \mid \exists x\ G \\ D &::= A \mid G \supset D \mid D \wedge D \mid \forall x\ D. \end{aligned} \tag{2}$$

Here, the connectives $\top$, $\vee$, and $\exists$ are permitted in goals and $\wedge$ and $\forall$ can be mixed in definite formulas. Also, the "head" of a definite clause does not have to immediately present at the top-level of a clause: it might be buried to the right of implications and conjunctions and under universal quantifiers.

A compact presentation of Horn clauses can be given simply as

$$D ::= A \mid A \supset D \mid \forall x\ D. \tag{3}$$

This is the same as the set of formulas D over $\wedge$, $\supset$, and $\forall$ such that

$$\mathit{order}(D) \in \{0, 1\}.$$

The `lists` module

```
module lists.

kind list     type -> type.

type id       list A -> list A -> o.
type memb     A -> list A -> o.
type append   list A -> list A -> list A -> o.

id nil nil.
id (X::L) (X::K) :- id L K.

memb X (X::L).
memb X (Y::L) :- memb X L.

append nil K K.
append (X::L) K (X::M) :- append L K M.

type    member_and_rest     A -> list A -> list A -> o.

member_and_rest X (X::L) L.
member_and_rest X (Y::K) (B::L) :- member_and_rest X K L.
```

The pairs module

```
module pairs.

kind pair    type -> type -> type.
type pair    A -> B -> pair A B.

type assoc, assok   A -> B -> list (pair A B) -> o.
type domain         list (pair A B) -> list A -> o.
type range          list (pair A B) -> list B -> o.

assoc X Y ((pair X Y)::L).
assoc X Y (P::L) :- assoc X Y L.

assok X Y ((pair X Y)::L) :- !.
assok X Y (P::L) :- assok X Y L.

domain nil nil.
domain ((pair X Y)::Alist) (X::L) :- domain Alist L.

range nil nil.
range ((pair X Y)::Alist) (Y::L)  :- range Alist L.
```