Conveniences that I wished prolog had

Term deconstructor and alias:

```
my_pred(X=(Part1, Part2)) :-
  …
```

An elegant (in my opinion) alternative to adding another rule to the body of the predicate; Also easier to understand at a glance what kind
of term holds for the predicate.

Ignore empty commas (or colon) in a predicate body:

```
my_pred(...) :-
  rule1,
  rule2, % This comma is ignored
  .
```

Useful when you are quickly debugging a predicate.

A have a skip predicate:

```
my_pred(...):-
 rule1,
 skip,
 rule2. % Only rule 1 is checked
```

Useful for debugging only part of a predicate without wrangling with comments.

Note:

metalevel says that you can produce such behavior using the false atom:

```
my_pred(...):-
 rule1,
 false,
 rule2.
```

However in my tests `my_pred(X)` always evaluates to false, that makes intuitive sense to me since the above rule can be rewritten as

`my_pred(...) <- rule1 and false and rule2`, which should evaluate to `false` unconditionally.

I could emulate a skip predicate with `…; false,...`:

```
my_pred(...):-
 rule1;
 false,
 rule2.
```

Due to operator precedence rules the above is equivalent to: `my_pred(...) <- rule1 or (false and rule2)` which should evaluate to `my_pred(...) <- rule1`.

Type hints

Currently prolog and prolog linters only check for a predicate arity to statically determine if a predicate holds. Knowing if you even provided a variable
of the correct type to a predicate can only be determined during execution.

```
my_child_pred(X=type_tuple(type_clpf_integer, type_integer),
Y=type_list(all, type_string)) :- …

my_parent_pred(X=type_clpf_integer):-
  my_child_pred((X, 1), [hello]).
```

A type checker would accuse that I'm using the incorrect types because I'm trying to pass a list of atoms to the second term of `my_child_pred`, which expects a list of strings.

I think this could be implemented with a library if term deconstructor aliases were implemented to the language. And I believe it would be able to be done statically.