Instantiation as "return"

```
foodWithColor(Color) :- food(F), color(F,Color).
```

Prolog has no analog for "`return x`" as is found in most languages. In Prolog there is no way to say something like this,

```
?- Color = foodWithColor(), writeln(Color), fail.
```

or this,

```
?- writeln(foodWithColor()), fail.
```

Instead, predicates "return" values by instantiating logical variables.

```
?- foodWithColor(C), writeln(C), fail.
red
green
...
```


Instantiation as "return", continued

Here's a one-rule predicate that produces structures with food/color pairs:

```
get_fwc(Color, Result) :-
    food(F), color(F,Color), Result = fwc(F,Color).
```

Usage:

```
?- get_fwc(green, R).
R = fwc(broccoli, green) ;
R = fwc(lettuce, green) ;
No

?- get_fwc(_, R).
R = fwc(apple, red) ;
R = fwc(broccoli, green) ;
R = fwc(carrot, orange) ;
R = fwc(lettuce, green) ;
R = fwc(rice, white) ;
No
```

How could we make the predicate more concise?

Instantiation as "return", continued

First version and usage:

```
get_fwc(Color, Result) :-
    food(F), color(F,Color), Result = fwc(F,Color).

?- get_fwc(green,R).
R = fwc(broccoli, green)
```

Above, `R` is unified with `Result`, and `Result` is unified with `fwc(F,Color)`, (transitively) unifying `R` with `fwc(broccoli, green)`.

Instead of unifying `Result` with a structure in the body of the rule, we can eliminate the "middle-man" `Result` and specify the structure in the head, unifying it with whatever is specified in the query:

```
get_fwc(Color, fwc(F,Color)) :- food(F), color(F,Color).
```


Computing with facts

Consider these two nearly-identical predicates:

```
equal_e(X,Y) :- X == Y.    % equality

equal_u(X,Y) :- X = Y.     % unification
```

How does their behavior differ? What's something we could do with one that we couldn't do with the other?

Computing with facts, continued

```
equal_e(X,Y) :- X == Y.    % equality

equal_u(X,Y) :- X = Y.     % unification
```

Usage:

```
?- equal_e(abc,abc).
Yes

?- equal_e(abc,X).
No

?- equal_u(abc,X).
X = abc

?- equal_u(X,abc).
X = abc
```

What is the result of the following queries?

```
?- A = 1, equal_u(A,B).
?- equal_u(B, A), A = 1, equal_u(B, 2).
```

Can `equal_u` be shortened?

Computing with facts, continued

At hand:

```
equal_u(X,Y) :- X = Y.
```

The right way to do it:

```
equal_u(X,X).
```

`equal_u` is a fact that performs computation via unification and instantiation.

Below are some more facts that perform computation. Describe what they do.

```
square(rectangle(S, S)).

rotate(rectangle(W, H), rectangle(H, W)).

width(rectangle(W, _), W).

height(rectangle(_, H), H).
```


Computing with facts, continued

```
square(rectangle(S, S)).
rotate(rectangle(W, H), rectangle(H, W)).
width(rectangle(W, _), W).
```

Usage:

```
?- square(rectangle(3,3)).
Yes

?- rotate(rectangle(3,5),R2).
R2 = rectangle(5, 3)
Yes

?- rotate(rectangle(3,5),R2), width(R2, W).
R2 = rectangle(5, 3)
W = 5

?- rotate(muffler,M2).
No
```

Problem: Using only `rotate/2`, write `square/1`.

Computing with facts, continued

At hand:

```
rotate(rectangle(W, H), rectangle(H, W)).
```

One way to think about a square is that it's a rectangle whose 90-degree rotation equals itself:

```
square(R) :- rotate(R,R).
```

The thing to appreciate is that `rotate` describes a relationship between two `rectangle` structures. With a single fact Prolog can create a rotated rectangle, determine if one rectangle is a rotation of another, and more.

Sidebar: Describing predicates

Recall this predicate: **`between(1,10,X)`**

Here is what `help(between)` shows:

```
between(+Low, +High, ?Value)
  Low and High are integers, High >= Low.  If Value is an integer,
  Low =< Value =< High. When Value is a variable it is successively
  bound to all integers between Low and High.
```

If an argument has a plus prefix, like `+Low` and `+High`, it means that the argument is an input to the predicate. A question mark indicates that the argument can be input or output.

The documentation indicates that `between` can (1) generate values and (2) test for membership in a range.

```
?- between(1,10,X).
X = 1
...

?- between(1,10,5).
Yes
```


Describing predicates, continued

How would the arguments of these predicates be described?

```
rotate(Rect1, Rect2)

square(Rectangle)

equal_u(X, Y)
```

Here is the synopsis for `format/2`:

```
format(+Format, +Arguments)
```

Speculate: What does `sformat/3` do?

```
sformat(-String, +Format, +Arguments)
```

Amusement: What does the following query do?

```
?- between(1,100,X), format('Considering: ~d\n', X),
   between(50,52,X), format('Found ~d\n', X), fail.
```


Arithmetic

Just as we saw with \==, Prolog builds structures out of arithmetic expressions:

```
?- display(1 + 2 * 3).
+(1, *(2, 3))
Yes

?- display(1 / 2 * ( 3 + 4)).
*(/(1, 2), +(3, 4))
Yes

?- display(300.0/X*(3+A*0.7**Y)).
*(/(300.0, _G373), +(3, *(_G382, **(0.7, _G380))))
```

Unlike \==, there are no predicates for the arithmetic operators. Example:

```
?- +(3,4).
ERROR: Undefined procedure: (+)/2

?- *(10,20).
ERROR: Undefined procedure: * /2
```

Question: Why would predicates be awkward for arithmetic expressions?

Arithmetic, continued

The predicate `is/2` evaluates a structure representing an arithmetic expression and unifies the result with a logical variable:

```
?- is(X, 3+4*5).

X = 23

Yes
```

`is/2` is usually used as an infix operator:

```
?- X is 3 + 4, Y is 7 * 5, Z is X / Y.

X = 7
Y = 35
Z = 0.2
```

All variables in the structure being evaluated by `is/2` must be instantiated:

```
?- A is 3 + X.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Arithmetic, continued

It is not possible to directly specify an arithmetic value as an argument of most predicates:

```
?- write(3+4).
3+4
Yes

?- 3+4 == 7.
No

?- between(1, 5+5, 7).
ERROR: between/3: Type error: `integer' expected, found `5+5'
```


Arithmetic, continued

A full set of arithmetic operations is available:

`-X`	negation
`X+Y`	addition
`X-Y`	subtraction
`X*Y`	multiplication
`X/Y`	division—produces float quotient
`X//Y`	integer division
`X mod Y`	integer remainder
`integer(X)`	truncation to integer
`float(X)`	conversion to float
`sign(X)`	sign of `X`: -1, 0, or 1

```
?- X is 43243432422342123234 / 77777777777777777.
X = 555.987

?- X is 10 // 3.
X = 3

?- X is e ** sin(pi).
X = 1.0
```


Arithmetic, continued

Here are some predicates that employ arithmetic. Remember that we have to "return" values via instantiation.

```
add(X, Y, Sum) :- Sum is X + Y.

around(Prev,X,Next) :- Prev is X - 1, Next is X + 1.

area(rectangle(W,H), A) :- A is W * H.

area(circle(R), A) :- A is pi * R ** 2.

?- add(3,4,X).
X = 7

 ?- around(P, 7, N).
P = 6
N = 8

?- area(circle(3), A).
A = 28.2743

?- area(rectangle(2*3, 2+2), Area).
Area = 24
```


Comparisons

There are several numeric comparison operators:

`X =:= Y`	numeric equality
`X =\= Y`	numeric inequality
`X < Y`	numeric less than
`X > Y`	numeric greater than
`X =< Y`	numeric equal or less than (note order!)
`X >= Y`	numeric greater than or equal

Just like `is/2`, they evaluate their operands. Examples of usage:

```
?- 3 + 5 =:= 2*3+2.
Yes

?- X is 3 / 5, X > X*X.
X = 0.6

?- X is random(10), X > 5, writeln(X).
6
```

Note that the comparisons produce no value; they simply succeed or fail.

Comparisons, continued

Some predicates for grading:

```
grade(Score,Grade) :- Score >= 90, Grade = 'A'.
grade(Score,Grade) :- Score >= 80, Score < 90, Grade = 'B'.
grade(Score,Grade) :- Score >= 70, Score < 80, Grade = 'C'.
grade(Score,Grade) :- Score < 70, Grade = 'F'.

print_grade(Score) :- grade(Score,Grade),
    format('~w -> ~w\n', [Score, Grade]).
```

Usage:

```
?- grade(95,G).
G = 'A'
Yes

?- print_grade(80).
80 -> B
Yes

?- print_grade(55).
55 -> F
Yes
```