Some odds and ends

Reversing failure with \+

Cut (!)

The "cut-fail" idiom

Green cuts and red cuts

Reversing failure with \+

The query `\+goal` succeeds if *goal* fails.

```
?- food(computer).
No

?- \+food(computer).
Yes
```

\+ is sometimes read as "can't prove" or "fail if".

Example: *What foods are not green?*

```
?- food(F), \+color(F, green).
F = apple ;
F = carrot ;
F = rice ;
No
```

Problem: Write a predicate `inedible(X)` that succeeds if something is not a food.

Speculate: What will the query `?- inedible(X).` do?

Reversing failure, continued

`inedible/1` and some examples of use:

```
?- listing(inedible).

inedible(A) :- \+food(A).

?- inedible(computer).
Yes

?- inedible(banana).
Yes

?- inedible(X).
No
```

Any surprises?

Cut (!)

Backtracking can be limited with the "cut" operator, represented by an exclamation mark.

A cut always succeeds when evaluated, but inhibits backtracking.

```
?- food(F), writeln(F), fail.
apple
broccoli
carrot
lettuce
rice

No

?- food(F), writeln(F), !, fail.
apple

No
```

One way to picture a cut is like a one-way gate: control can go through it from left to right, but not from right to left.

Cut, continued

Consider these facts:

```
f(' f1 ').   g(' g1 ').
f(' f2 ').   g(' g2 ').
f(' f3 ').   g(' g3 ').
```

Queries and cuts:

```
?- f(F), write(F), g(G), write(G), fail.
 f1  g1  g2  g3  f2  g1  g2  g3  f3  g1  g2  g3

?- f(F), write(F), !, g(G), write(G), fail.
 f1  g1  g2  g3

?- f(F), write(F), g(G), write(G), !, fail.
 f1  g1
```

What's the output of the following query?

```
?- !, f(F), !, write(F), !, g(G), !, write(G), !, fail.
```

Another analogy: A cut is like a door that locks behind you.

Cut, continued

In fact, cuts are rarely used at the query prompt. The real use of cuts is in rules. Here's an artificial example:

```
ce1(one,1) :- true.            % Equivalent to ce1(one,1).
ce1(two,2) :- true, !.
ce1(_, 'something') :- true.
```

Usage:

```
?- ce1(one,X).
X = 1 ;
X = something ;
No

?- ce1(two, X).
X = 2 ;
No
```

Note that the third clause is not tried for `ce1(two,X)`. That's due to the cut.

In a rule, a cut still acts as a one-way gate in the rule itself but it also prevents consideration of subsequent clauses for the current call of that predicate. It's "do-or-die" (succeed or fail) with the rule at hand.

Cut, continued

Control never backtracks through a cut but backtracking can occur between goals on each side of a cut.

Consider this abstract rule,

```
x :- a, b, c, !, d, e, f, !, g, h, i.
```

and a query:

```
?- x, y.
```

Control may circulate between `a`, `b`, and `c` but once `c` is proven (and the cut passed through), `a`, `b`, and `c` will be not be considered again during a particular call of `x`. Similarly, once `f` succeeds, we are further committed.

However, if `x` succeeds and `y` fails, control will backtrack into `g`, `h`, and `i` if they contain unexplored alternatives.

Cut, continued

Below is a faulty "improvement" for the `grade/2`, in the box at right.

```
grade2(Score, 'A') :- Score >= 90.
grade2(Score, 'B') :- Score >= 80.
grade2(Score, 'C') :- Score >= 70.
grade2(_, 'F').
```

Usage:

```
?- grade2(85,G).
G = 'B' ;
G = 'C' ;
G = 'F' ;

No
```

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

Cut, continued

Here is a procedure based on `grade2`:

```
grade2(Score, 'A') :- Score >= 90.
grade2(Score, 'B') :- Score >= 80.
grade2(Score, 'C') :- Score >= 70.
grade2(_, 'F').
```

```
do_grades(Students) :-
    member(Who-Avg, Students),
    grade2(Avg,Grade),
    format('~w: ~w~n', [Who, Grade]), fail.
```

Usage:

```
?- do_grades([bob-87, mary-92]).
bob: B
bob: C
bob: F
mary: A
mary: B
mary: C
mary: F
```

What's the problem? How can we fix it?

Cut, continued

Recall:

In a rule, a cut still acts as a one-way gate in the rule itself but it also prevents consideration of subsequent clauses for the current call of that predicate. It's "do-or-die" (succeed or fail) with the rule at hand.

Solution:

```
grade3(Score, 'A') :- Score >= 90, !.
grade3(Score, 'B') :- Score >= 80, !.
grade3(Score, 'C') :- Score >= 70, !.
grade3(_, 'F').
```

Usage:

```
?- grade3(85,G).
G = 'B' ;
No
```

```
do_grades(Students) :-
    member(Who-Avg, Students),
    grade3(Avg,Grade),
    format('~w: ~w~n', [Who, Grade]), fail.
```

```
?- do_grades([bob-87, mary-92]).
bob: B
mary: A
```

Problem: Write a predicate `agf(-F)` that finds at most one green food.

Cut, continued

Here is an example from *Clause and Effect* by Clocksin:

```
max(X,Y,X) :- X >= Y, !.
max(_,Y,Y).
```

What's the other way to write it?

Also borrowed from *Clause and Effect*, here's a variant of `member/2`:

```
xmember(X, [X|_]) :- !.
xmember(X, [_|T]) :- xmember(X,T).
```

How does its behavior differ from the standard version? (below)

```
member(X, [X|_]).
member(X, [_|T]) :- member(X,T).
```

There's a built-in predicate, `memberchk/2`, that has the same behavior as `xmember`. When might it be appropriate to use it instead of `member/2`?

Cut, continued

`memberchk` is useful to avoid exploring alternatives with values that are known to be worthless. Here is an artificial example that represents a computation that selects several occurrences of the same value (3) from a list and tries each.

```
?- X = 3, member(X,[1,3,5,3,3]), writeln('Trying X...'), X > 3.
Trying X...
Trying X...
Trying X...

No
```

In fact, if the first 3 doesn't work then's no reason to hope that a subsequent 3 will work, assuming no side-effects. Observe the behavior with `memberchk`:

```
 ?- X = 3, memberchk(X,[1,3,5,3,3]), writeln('Trying X...'),
    X > 3.
Trying X...

No
```

The "cut-fail" idiom

Predicates naturally fail when a desired condition is absent but sometimes we want a predicate to fail when a particular condition is present.

Here is a recursive predicate that succeeds iff all numbers in a list are positive:

```
allpos([X]) :- X > 0.
allpos([X|T]) :- X > 0, allpos(T).
```

Another way to write it is with a "cut-fail":

```
allpos(L) :- member(X, L), X =< 0, !, fail.
allpos(_).
```

Remember that a cut effectively eliminates all subsequent clauses for the active predicate. If a non-positive value is found the cut eliminates `allpos(_).` and then the rule fails.

```
?- allpos([3,-1,5]).
No
```

Another way to think about cut-fail: "My final answer is No!"

"cut-fail", continued

The SWI documentation for `is_list` includes its implementation:

```
is_list(X) :- var(X), !, fail.
is_list([]).
is_list([_|T]) :-  is_list(T).
```

A cut-fail is used in the first rule to say that if `X` is a free variable (i.e., it is uninstantiated), look no further and fail.

Problem: Add a clause for `food/1` that "turns off" all the foods:

```
?- food(F).

No
```


Green cuts and red cuts

A cut is said to be a "green cut" if it simply makes a predicate more efficient. By definition, adding or removing a green cut does not effect the set of results for any call of a predicate.

A "red cut" affects the set of results produced by a predicate.

Are there any examples of green cuts in the preceding examples?

Database manipulation

`assert` and `retract`

A simple command interpreter

Word tally

Unstacking blocks

`assert` and `retract`

A Prolog program is a database of facts and rules.

The database can be changed dynamically by adding facts with `assert/1` and deleting facts with `retract/1`.

A predicate to establish that certain things are food:

```
makefoods :-
    assert(food(apple)),
    assert(food(broccoli)),
    assert(food(carrot)),
    assert(food(lettuce)),
    assert(food(rice)).
```


`assert` and `retract`, continued

Evaluating `makefoods` adds facts to the database:

```
?- food(X).
ERROR: Undefined procedure: food/1

?- makefoods.
Yes

?- food(X).
X = apple
```

A food can be "removed" with `retract`:

```
?- retract(food(carrot)).
Yes

?- food(carrot).
No
```


`assert` and `retract`, continued

In a predicate like `makefoods`, use `retractall` to start with a clean slate:

```
makefoods :-
    retractall(food(_)),
    assert(food(apple)),
    assert(food(broccoli)),
    assert(food(carrot)),
    assert(food(lettuce)),
    assert(food(rice)).
```

Without `retractall`, each call to `makefoods` creates an identical, duplicated set of foods.

Here's the behavior of the first version of `makefoods`:

```
?- makefoods, makefoods, makefoods.
Yes

?- findall(F, food(F), Foods).
Foods = [apple, broccoli, carrot, lettuce, rice, apple,
broccoli, carrot, lettuce|...]
```

assert and retract, continued

Asserts and retracts are NOT undone with backtracking.

```
?- assert(f(1)), assert(f(2)), fail.
No

?- f(X).
X = 1 ;
X = 2 ;
No

?- retract(f(1)), fail.
No

?- f(X).
X = 2 ;
No
```

There is no notion of directly changing a fact. Instead, a fact is changed by retracting it and then asserting it with different terms.

assert and retract, continued

A rule to remove foods of a given color (assuming the `color/2` facts are present):

```
rmfood(C) :- food(F), color(F,C),
       retract(food(F)),
        write('Removed '), write(F), nl, fail.
```

Usage:

```
?- rmfood(green).
Removed broccoli
Removed lettuce
No

?- food(X).
X = apple ;
X = carrot ;
X = rice ;
No
```

The color facts are not affected—`color(broccoli, green)` and `color(lettuce,green)` still exist.

A simple command interpreter

Imagine a simple interpreter to manipulate an integer value:

```
?- run.
? print.
0
? add(20).
? sub(7).
? print.
13
? set(40).
? print.
40
? exit.

Yes
```

Note that the commands themselves are Prolog terms.

A simple command interpreter, continued

Here is a loop that simply reads and prints terms:

```
run0 :- repeat, write('? '), read(X), format('Read ~w~n', X),
        X = exit.
```

Usage:

```
?- run0.
? a.
Read a
? ab(c,d,e).
Read ab(c, d, e)
? exit.
Read exit

Yes
```

`repeat/0` always succeeds. If `repeat` is backtracked into, it simply sends control back to the right. (Think of its `redo` port being wired to its `exit` port.)

The predicate `read(-X)` reads a Prolog term and unifies it with `X`.

Problem: Explain the operation of the loop.

A simple command interpreter, continued

Partial implementation:

```
init :-
    retractall(value(_)),
    assert(value(0)).

do(set(V)) :-
    retract(value(_)),
    assert(value(V)).

do(print) :- value(V), write(V), nl.

do(exit).

run :-
    init,
    repeat, write('? '), read(X), do(X), X = exit.
```

```
?- run.
? print.
0
? add(20).
? sub(7).
? print.
13
? set(40).
? print.
40
? exit.

Yes
```

How can `add(N)` and `sub(N)` be implemented?

Challenge: Add `mul` and `div` without having any repetitious code. Hint: Think about building a structure to evaluate with `is/2`.

Word tally

Manipulation of facts is very efficient in most Prolog implementations. We can use facts like we might use a Java map or a Ruby hash.

Imagine a word tallying program in Prolog:

```
?- tally.
|: to be or
|: not to be ought not
|: to be the question
|: (Empty line ends the input)

-- Results --
be          3
not         2
or          1
ought       1
question    1
the         1
to          3

Yes
```


Word tally, continued

Implementation:

```
readline(Line) :-
    current_input(In), read_line_to_codes(In, Codes),
    atom_chars(Line, Codes).

tally :- retractall(word(_,_)),
    repeat, readline(Line), do_line(Line), Line == '',
    show_counts.

do_line('').
do_line(Line) :-
        concat_atom(Words, ' ', Line), % splits Line on blanks
        member(Word, Words),
        count(Word), fail.
do_line(_).

count(Word) :-
    word(Word,Count0), retract(word(Word,_)),
    Count is Count0+1, assert(word(Word,Count)), !.

count(Word) :- assert(word(Word,1)).
```


Word tally, continued

`show_counts` shows the results.

```
show_counts :-
    writeln('\n-- Results --'),
    findall(W-C, word(W,C), Pairs),
    keysort(Pairs, Sorted),
    member(W-C, Sorted),
    format('~w~t~12|~w~n', [W,C]),
    fail.
show_counts.
```

Problem: Modify the above so that words are ordered by count.

Is `keysort` really needed in the version shown?

Challenge: Create `show_counts(+Which)` where `Which` is `w` or `c` to cause ordering by words or counts.