The Cut

The cut “!” is a special type of atom used to control Prolog’s backtracking behaviour. Suppose a program contains a rule

\[ H : \neg A_1, \ldots, A_n, !, B_1, \ldots, B_m. \]

1. When first encountered during resolution, the cut succeeds (with empty answer substitution). That is, a goal

\[ \bot : = !, B_1, \ldots, B_m, C_1, \ldots, C_k \]

has descendent

\[ \bot : = B_1, \ldots, B_m, C_1, \ldots, C_k \]

2. When the cut is reached during backtracking, all subsequent branches descended from the goals with initial atoms \( A_1, \ldots, A_n \) and \( H \) are omitted from the search.
In other words, backtracking over the cut forces a `fail` on the all atoms between the cut and the head of the rule it occurs in, and shifts the focus of backtracking to the point just before the `H` was called.

No further ways of satisfying $A_1, \ldots, A_n$ are considered.

Nor are any further rules for `H` considered.

Example:

```
a(X) :- b(X).
a(X) :- c(X), !, d(X).
a(X) :- e(X).

b(1).
b(2).
c(3).
c(4).
d(3).
d(4).
e(5).
```
(Note that the cut has destroyed the logical purity of the program: the “answers” 4 and 5 are not discovered by the general query $a(X)$.)

The cut does not restrict backtracking on goals “higher up” in the proof tree.

\[
\begin{align*}
a(Y) & : = b(Y), \rightarrow. \\
   b(1). \\
   b(2). \\
\end{align*}
\]

: $b(X), a(Y)$?

$X = 1$
$Y = 1$

$X = 2$
$Y = 1$
Application: eliminating search that is bound to fail.

There is only one way to reverse a list. Thus, once we have found a solution, we can cut the rest of the search:

\[
\text{reverse}(X, Y) :- \text{rev2}(X, [], Y).
\]

\[
\text{rev2}([], L, L) :- !.
\]

\[
\text{rev2}([X|Y], Z, T) :- \text{rev2}(Y, [X|Z], T), !.
\]

: reverse([2,3,4], X)?

X = [4, 3, 2]

Application: restricting the set of answers returned.

If we write a predicate for membership in a list as follows:

\[
\text{mem}(X, [X|\_]).
\]

\[
\text{mem}(X, [\_|R]) :- \text{mem}(X, R).
\]

then we get the behaviour:

: mem(2, [2,3,2,2,5]), X=1?

X = 1

X = 1

X = 1
Using cut...

\[\text{mem}(X, [X|\_]) :\! : bid .
\text{mem}(X, [\_|R]) :\! : \text{mem}(X, R).\]

we get

\[\text{mem}(2, [2,3,2,2,5]), X=1?\]

\[X = 1\]

\[\text{mem}(X, [4,5,6,7])?\]

\[X = 4\]

(Note: we have lost the ability to “generate all possible answers!”)

Using cut to return just a single answer in a generate and test program:

\[\text{solve}(\text{Problem}, \text{Soln}) :\! : \text{generate(Problem, Soln)},
\text{test(problem, Soln)}, !.\]
Australian citizens are to pay 10% GST. US Citizens are exempt from GST, i.e., pay 0% GST. People with dual citizenship of Australia and the USA are to be treated as Australian citizens, i.e., they pay 10% GST.

\[
gst(P, 10) :- \text{citizen}(P, \text{australia}), !. \\
gst(P, 0) :- \text{citizen}(P, \text{usa}).
\]

citizen(jones, australia).
citizen(jones, usa).

: gst(jones, X)?

\[
X = 10
\]

Prolog I/O: Terms

\[
\begin{align*}
\text{write} & (X) & \text{— write the term } X \\
\text{nl} & \text{— write a newline} \\
\text{read} & (X) & \text{— read a term, followed by “.”, and unify with } X.
\end{align*}
\]

Each of these succeeds just once.

: read(a(X,Y)), Z is X+Y, write("The sum is \\
\quad \text{....}"), \\
\quad \text{nl}, write(Z), \text{nl}? \\
a(1,2).
\]

The sum is 

\[
3 \\
X = 1 \\
Y = 2 \\
Z = 3
\]
Prolog I/O: Characters

IProlog:

\texttt{getc}(X) – read the next character
\texttt{putc}(X) – write the next character
\texttt{ratom}(X) – read the next ‘atom’

Standard Prologs:

\texttt{get}(X) – unify X with the ASCII code of the next character on the input stream, skipping non-printing characters.
\texttt{get0}(X) – unify X with the next character on the input stream
\texttt{put}(X) – write character with ASCII code X on the output stream

File Handling

\texttt{see}(X) – open file X for reading
\texttt{seeing}(X) – what file are we reading?
\texttt{seen} – close the current open file, revert to user
\texttt{tell}(X) – open file X for writing
\texttt{telling}(X) – what file are we writing?
\texttt{told} – close the current open file, revert to user
Parsing and Grammars

A grammar is a set of rules describing the structure of the sentences of a language. There are many different approaches to doing this. A context-free grammar is a set of rules such as the following:

Sentence --> Noun-Phrase VerbPhrase
Noun-Phrase --> Article Adjective Noun
Verbphrase --> Verb Adverb

Article --> [a]   Adjective --> [black]
Article --> [the] Adjective --> [brown]
Noun --> [dog]
Noun --> [cat]
Verb --> [runs]   Adverb --> [slowly]
Verb --> [swims]   Adverb --> [quickly]

By repeatedly applying these rules we can generate sentences:

Sentence --> Noun-Phrase Verb-phrase
   --> Article Adjective Noun Verb-phrase
   --> Article Adjective Noun Verb Adverb
   --> [a] Adjective Noun Verb Adverb
   --> [a] Adjective Noun [runs] Adverb
   --> [a, black] Noun [runs] Adverb
   --> [a, black, cat, swims] Adverb
   --> [a, black, cat, swims, slowly]

Conversely, given a context-free grammar and a list of words or letters, we can ask, is there some way of generating this using the grammar?

This is called the parsing problem.
Encoding the parsing problem in Prolog:

```prolog
sentence(X) :- concat(X1,X2,X),
              nounphrase(X1), verbphrase(X2).
nounphrase(X) :- concat(X1,X2,T), concat(T,X3,X),
                article(X1), adjective(X2), noun(X3).
verbphrase(X) :- concat(X1,X2,T), verb(X1), adverb(X2).
```

- `article([a])`. `adjective([black])`.
- `article([the])`. `adjective([brown])`.
- `noun([cat])`. `verb([runs])`.
- `noun([dog])`. `verb([swims])`.
- `adverb([slowly])`. `adverb([quickly])`.

Consider what happens for the rule

```prolog
sentence(X) :- concat(X1,X2,X), nounphrase(X1), verbphrase(X2).
```

Here given a list of words X, the atom `concat(X1,X2,X)` is used generate possible answers for X1,X2, that are then tested.

E.g. `concat(X1,X2,[the, brown, dog, runs, quickly])` generates the possible answers

- X1 = [] X2 = [the, brown, dog, runs, quickly]
- X1 = [the] X2 = [brown, dog, runs, quickly]
- X1 = [the, brown] X2 = [dog, runs, quickly]
- X1 = [the, brown, dog] X2 = [runs, quickly]

before the last of these succeeds.
A smarter approach:

use a predicate `nounphrase(S0,S1)` such that if `S0` is a list and `S1` is a variable, succeeds if it finds a nounphrase at the front of `S0` and returns the remainder of the list in `S1`.

Similarly for the remaining types of phrases...

definitions:

sentence(S0,S2) :- nounphrase(S0,S1), verbphrase(S1,S2).
nounphrase(S0,S3) :- article(S0,S1),
                   adjective(S1,S2), noun(S2,S3).
verbphrase(S0,S2) :- verb(S0,S1), adverb(S1,S2).

article(['a|S'],S). adjective(['black|S'],S).
article(['the|S'],S). adjective(['brown|S'],S).
noun(['cat|S'],S). verb(['runs|S'],S).
noun(['dog|S'],S). verb(['swims|S'],S).

: sentence(['the, brown, dog, runs, quickly'],[])?
**yes**
We can use extra arguments to extract information from the subphrases. For example, we can use this to return a parse tree or build up an expression:

```prolog
equation(E,S0,S1) :- integer(E,S0,S1).
expression(plus(E1,E2),S0,S5) :-
    S0 = ['('|S1], equation(E1,S1,S2),
    S2 = ['+'|S3], equation(E2,S3,S4),
    S4 = [')'|S5].
expression(times(E1,E2),S0,S5) :-
    S0 = ['('|S1], equation(E1,S1,S2),
    S2 = ['+'|S3], equation(E2,S3,S4),
    S3 = [')'|S5].
integer(E,S0,S1) :- digit(E,S0,S1).
digit(0,
We can also use the extra arguments to place conditions on the subphrases. E.g. the following is a grammar for the set of sequences of the form a^n b^n c^n:

```prolog
sentence(S0,S3) :-
    seq(a,N,S0,S1), seq(b,N,S1,S2), seq(c,N,S2,S3).
seq(L,1,S0,S1) :- letter(L,S0,S1).
seq(L,N,S0,S2) :- letter(L,S0,S1), seq(L,M,S1,S2), N is M+1.
letter(L,[L|S],S).
```
Most Prologs (including XSB, but not iprolog, it seems), have a special notation to simplify writing grammar rules:

\[ H(\text{ArgsH}) \rightarrow B_1(\text{ArgsB}_1), \ldots, B_n(\text{ArgsB}_n), \{C_1, \ldots, C_m\}. \]

is translated by the Prolog compiler to

\[ H(\text{ArgsH}, S_0, S_n) :- B_1(\text{ArgsB}_1, S_0, S_1), \ldots, B_n(\text{ArgsB}_n, S_{n-1}, S_n), C_1, \ldots, C_m. \]

If \( B_i \) is \([L_1, \ldots, L_n]\) it is translated to \( S_{i-1} = [L_1, \ldots, L_n|S_i] \).

Example: we can write the arithmetic expression example as follows:

```prolog
expression(E) --> integer(E).
expression(plus(E1,E2)) --> ['('], expression(E1), ['+'], expression(E2), [')'].
expression(times(E1,E2)) --> ['('], expression(E1), ['*'], expression(E2), [')'].
integer(E) :- digit(E).
digit(X) -> [X], {(X=1; X=2; \ldots; X=9)}. (";" in a Prolog rule body means OR. This can be used in grammar rule bodies also.)
```

12