Exercise 1.1

9. After typing a semicolon to initiate backtracking, forget the previous `X`, and attempt to re-satisfy the first goal.
10. The first goal fails.
11. Prolog returns `false.` because there are no more solutions.

Exercise 1.2

We can obtain all two answers to `sister_of/2` by just querying for it:

```
 ?- sister_of(X, Y).
X = alice,
Y = edward ;
X = Y, Y = alice ;
false.
```

Exercise 1.3

in back of book

```
 is_mother(Mum) :- mother(Mum, _Child).
is_father(Dad) :- father(Dad, _Child).
is_son(Son):- parent(_Par, Son), male(Son).
sister_of(Sis, Pers) :-
    parent(Par, Sis), parent(Par, Pers),
    female(Sis), diff(Sis, Pers).
grandpa_of(Gpa, X) :- parent(Par, X), father(Gpa, Par).
sibling(Sl, S2) :-
    parent(Par, _Si), parent(Par, S2), diff(Sl, S2).
```

Note that we are using the predicate `diff` in the definition of `sister_of` and `sibling`. This prevents the system from concluding that somebody can be a sister or sibling of themselves. You will not be able to define `diff` at this stage.

Exercise 1.4

In the text, `sister_of/2` is defined on page 18 as:

```
sister_of(X, Y) :-
    female(X),
    parents(X, M, F),
    parents(Y, M, F).
```

It is possible for an object to be its own sister because we do not have a clause to exclude this situation. To resolve this, we can add a `diff` clause to the end of the rule like so:

```
sister_of(X, Y) :-
    female(X),
    parents(X, M, F),
    parents(Y, M, F),
    diff(X, Y).
```

Exercise 2.1

`pilots(A, london) = pilots(london, paris).` The goal fails.

`point(X, Y, Z) = point(X1, Y1, Z1).` The goal succeeds and causes `X` to co-refer with `X1`, `Y` with `Y1`, and `Z` with `Z1`.

`letter(C) = word(letter).` The goal fails.

`noun(alpha) = alpha.` The goal fails.

`'student' = student.` The goal succeeds.

`f(X, X) = f(a, b).` The goal fails.

`f(X, a(b, c)) = f(Z, a(Z, c)).` The goal succeeds and causes `X` and `Z` to co-refer and `Z` to be instantiated to `b`.

Exercise 3.1

Using `sometimes_better`, every car is preferred over every other car, resulting in all possible `3*3=9` solutions:

```
?- setof([X,Y], prefer(X,Y), L), length(L, Z).
L = [[guzzler, guzzler], [guzzler, prodigal], [guzzler, waster], [prodigal, guzzler], [prodigal, prodigal], [prodigal, waster], [was
Z = 9.
```

If we instead require at least one 10% improvement and swap `sometimes_better` for `significantly_better`:

```
significantly_better([Con1|_], [Con2|_]) :-
    UpperLimit is 0.9 * Con2,
    Con1 < UpperLimit.
significantly_better([_|Con1], [_|Con2]) :-
    significantly_better(Con1, Con2).

prefer(Car1, Car2) :-
    fuel_consumed(Car1, Con1),
    fuel_consumed(Car2, Con2),
    significantly_better(Con1, Con2).
```

then we find that there are only 4 preferred pairs:

```
?- setof([X,Y], prefer(X,Y), L), length(L, Z).
L = [[guzzler, waster], [prodigal, guzzler], [prodigal, waster], [waster, guzzler]],
Z = 4.
```

Exercise 4.1

If the cut is left out, i.e. the first clause of `sum_to/2` is `sum_to(1, 1).` instead of `sum_to(1, 1) :- !.`, then backtracking on the `sum_to/2` goal results in an infinite loop because we no longer have a base case for our recursion.

```
?- sum_to(3, X).
X = 6 ;
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 1.0Gb, global: 32Kb, trail: 1Kb
ERROR:   Stack depth: 11,183,521, last-call: 0%, Choice points: 3
ERROR:   Possible non-terminating recursion:
ERROR:     [11,183,521] user:sum_to(-11183508, _8448)
ERROR:     [11,183,520] user:sum_to(-11183507, _8468)
```

Exercise 5.1

The program `read_in/1` is reproduced below. Comments in `/* */` are given by C&M; those in `%` by me.

```
/* Read in a sentence */
% read_in([W|Ws]) is satisfied if W is a word and Ws is a list of words obtained by
% (1) reading a character C
% (2) using C to read in the next word W and the first character C1 after W
% (3) using W and C1 to read in the rest of the sentence Ws
read_in([W|Ws]) :- get_char(C), readword(C, W, C1), restsent(W, C1, Ws).

/* Given a word and the character after it, read in the rest of the sentence */
% a word W ends a sentence by having no subsequent characters if it is a lastword
restsent(W, _, []) :- lastword(W), !.
% given the first character C after a word, the rest of the sentence has
% first word W1 and remaining words Ws if
% (1) W1 is the word with first character C and the next word's first character is C1
% (2) the rest of the words in the sentence is Ws
restsent(_, C, [W1|Ws]) :- readword(C, W1, C1), restsent(W1, C1, Ws).
```

```
/*
 Read in a single word, given an initial character, and
 remembering which character came after the word.
*/
% given a character C, it forms an entire word if it is a single character
% and C1 is the first character of the next word
readword(C, C, C1) :- single_character(C), !, get_char(C1).
% given an initial character C, it forms a word W followed by a character C2 if
% (1) C is a valid character whose display value is NewC
% (2) C1 is a newly read-in character
% (3) Cs is the remaining characters of the current word and the next word's first character is C2
% (4) W is the combination of the first character of the word NewC
%     with the remaining characters in the word Cs
readword(C, W, C2) :-
    in_word(C, NewC),
    !,
    get_char(C1),
    restword(C1, Cs, C2),
    atom_chars(W, [NewC|Cs]).
% if C failed to satisfy in_word in the previous rule, then the next word W
% and the character following it C2 is given by
% (1) reading the next character C1
% (2) getting the word W that starts with C1 and the first character of the next word C2
readword(_, W, C2) :- get_char(C1), readword(C1, W, C2).

% given an initial character C, the rest of the word has first NewC
% followed by remaining characters Cs, followed by the first character of the next word C2 if
% (1) C is a valid character whose display value is NewC
% (2) C1 is a newly read-in character
% (3) Cs is the remaining characters of the current word and the next word's first character is C2
restword(C, [NewC|Cs], C2) :-
    in_word(C, NewC),
    !,
    get_char(C1),
    restword(C1, Cs, C2).
% give an initial character C that failed to satisfy in_word,
% this represents that first character of the next word
restword(C, [], C).

/*
 These characters can appear within a word. The second
 in_word clause converts letters to lower-case.
*/
% in_word(Character, DisplayCharacter) is satisfied
% if Character is a valid character to appear in a word
% and DisplayCharacter is what should be outputted

% lower-case letters C are valid and should be displayed as themselves
in_word(C, C) :- letter(C, _).          /* a b...z */
% upper-case letters C are valid and should be displayed
% as their corresponding lower-case letter L
in_word(C, L) :- letter(L, C).          /* A B...Z */
% digits C are valid and should be displayed as themselves
in_word(C, C) :- digit(C).              /* 1 2...9 */
% special characters C are valid and should be displayed as themselves
in_word(C, C) :- special_character(C).  /* '-' ""  */

/* Special characters */
special_character('-').
special_character("").

/* These characters form words on their own */
single_character(',').      single_character(':').
single_character('.').      single_character('?').
single_character(';').      single_character('!').
```

```
/* Upper and lower case letters */
letter(a, 'A').             letter(n, 'N').
letter(b, 'B').             letter(o, 'O').
letter(c, 'C').             letter(p, 'P').
letter(d, 'D').             letter(q, 'Q').
letter(e, 'E').             letter(r, 'R').
letter(f, 'F').             letter(s, 'S').
letter(g, 'G').             letter(t, 'T').
letter(h, 'H').             letter(u, 'U').
letter(i, 'I').             letter(v, 'V').
letter(j, 'J').             letter(w, 'W').
letter(k, 'K').             letter(x, 'X').
letter(l, 'L').             letter(y, 'Y').
letter(m, 'M').             letter(z, 'Z').

/* Digits */
digit('0').                 digit('5').
digit('1').                 digit('6').
digit('2').                 digit('7').
digit('3').                 digit('8').
digit('4').                 digit('9').

/* These words terminate a sentence */
lastword('.').
lastword('!').
lastword('?').
```

Exercise 5.2

in back of book

The following program reads in characters (from the current input file) indefinitely, printing them out again with `a`'s changed to `b`'s.

```
 go :- repeat, get_char(C), deal_with(C), fail.
deal_with(a) :- !, put_char(b).
deal_with(X) :- put(X).
```

The "cut" in the first `deal_with` rule is essential to avoid entering the second rule and potentially calling `put(a)` when backtracking.

Exercise 6.1

The intention of `get_non_space/1` is to read characters until it finds the next non-space character and then unify this character with `X`. However, if `X` is already instantiated, then this changes the behavior of `get_non_space/1` to read until the particular character `X` is found, i.e. a membership check rather than a value grab.

Exercise 7.1

The dictionary insertion order determines how the tree is built and what shape it has.

For example, an insertion order of (massinga, braemar, nettleweed, panorama) yields the tree:

```
      massinga
       / \
      /   \
     /     \
braemar  nettleweed
              \
               \
                \
              panorama
```

An insertion order of (adela, braemar, nettleweed, massinga) yields the tree:

```
adela
  \
   \
    \
   braemar
       \
        \
         \
      nettleweed
          /
         /
        /
   massinga
```

Exercise 7.2

```
% annotated program that prints messages
go(X, X, _) :- format('found telephone in room ~w', X).
go(X, Y, T) :-
    ((d(X, Z); d(Z, X))),
    \+ member(Z, T),
    format('entering room ~w~n', Z),
    go(Z, Y, [Z|T]).
```

Exercise 7.3

Yes, alternate paths can be found by the given program. To prevent more than one path from being found, we can add a cut at the end:

```
?- hasphone(X), go(a, X, []), !.
```

Exercise 7.4

The order of the facts `d/2` determines the order in which rooms are searched.

Exercise 7.5

Intuitively, we know that `permutation(L1, L2)` will generate all permutations of `L1` exactly once as the alternative values of `L2` by just looking carefully at each line of the program. Comments are mine:

```
permutation([], []).        % base case

permutation(L, [H|T]) :-    % a list L can be permuted into a list [H|T] by
    append(V, [H|U], L),    % letting H be an arbitrary element of L
    append(V, U, W),
    permutation(W, T).  % letting T by the permutation of the concatenated sublists V,U
                % where V and U are, respectively, the elements before and after H
```

Practically, we can execute `permutation/2` for the sample case of `[a,b,c,d]` and observe that all expected `4! = 24` cases are returned:

```
?- findall(X, permutation([a,b,c,d], X), L), length(L, Y).
L = [[a, b, c, d], [a, b, d, c], [a, c, b, d], [a, c, d, b], [a, d, b, c], [a, d, c, b], [b, a, c, d], [b, a, d, c], [b, c, a, d], [
Y = 24.
```

Note that the solutions are generated in lexicographic order wrt the input order, i.e. if the input order is alphabetical then the output order will also be alphabetical.

Exercise 7.6

We ignore the hint's suggestion to compute length inside of `split/4` and instead use the system predicate `length/2`. We implement the hybrid approach by adding a new rule that uses `insort/2` when the length of the list is less than 10, which falls within the 5-15 range recommended by Sedgewick and Wayne.

```
quisort2([], []).
quisort2(X, S) :-
    length(X, L),
    L < 10,
    insort(X, S).
quisort2([H|T], S) :-
    length([H|T], L),
    L >= 10,
    split(H, T, A, B),
    quisort2(A, A1),
    quisort2(B, B1),
    append(A1, [H|B1], S).
```

Exercise 7.7

`random_pick(X, Y)` instantiates `Y` to a randomly-chosen element of list `X`

```
random_pick(X, Y) :-
    length(X, L1),
    L2 is L1 - 1,
    random_between(0, L2, R),
    nth0(R, X, Y).
```

Optionally, we could take the hint's suggestion to define our own version of `nth0(Index, List, Elem)` :

```
my_nth0(0, [X|_], X).
my_nth0(N1, [_|X], Y) :-
    N2 is N1 - 1,
    my_nth0(N2, X, Y).
```

Exercise 7.8

Given the goal `findall(X, G, L)`, if there are uninstantiated variables not sharing with X in G, then `findall(X, G, L)` will enter an infinite loop. This happens because, referring to the definition of `findall/3` on page 168, `call(G)` never fails because we have that uninstantiated variable, so the first clause keeps looping and we never enter the second clause.

Exercise 7.9

in back of book

Here is a program that generates Pythagorean triples:

```
pythag(X, Y, Z) :-
    intriple(X, Y, Z),
    Sumsq is X * X + Y * Y,
    Sumsq is Z * Z.

intriple(X, Y, Z) :-
    is_integer(Sum),
    minus(Sum, X, Suml),
    minus(Suml, Y, Z),
    X>0, Y>0, Z>0. % required condition missing from back of book

minus(Sum, Sum, 0).
minus(Sum, Dl, D2) :-
    Sum > 0, Suml is Sum - 1,
    minus(Suml, Dl, D3), D2 is D3 + 1.

is_integer(0).
is_integer(N) :- is_integer(Nl), N is Nl + 1.
```

The program uses the predicate `intriple/3` to generate possible triples of integers X, Y, Z. It then checks to see whether this triple really is a Pythagorean triple. The definition of `intriple/3` has to guarantee that all triples of integers will eventually be generated. It first of all generates an integer that is the sum of `X`, `Y` and

`Z`. Then it uses a non-deterministic subtraction predicate, minus, to generate values of `X`, `Y` and `Z` from that.

Exercise 9.1

in back of book

Note: I was unable to get these programs to run successfully

Here is the program for translating a simple grammar rule into Prolog. It is assumed here that the rule contains no phrase types with extra arguments, no goals inside curly brackets, and no disjunctions or cuts.

```
 ?- op(1199,xfx,-->).

translate((Pl-->P2),(G1:-G2)) :-
    left_hand_side(Pl,SO,S,Gl),
    right_hand_side(P2,S0,S,G2).

left_hand_side(P0,S0,S,G) :-
    nonvar(PO), tag(P0,S0,S,G).

right_hand_side((Pl,P2),S0,S,G) :-
    !,
    right_hand_side(Pl,SO,Sl,Gl),
    right_hand_side(P2,Sl,S,G2),
    and(Gl,G2,G).
right_hand_side(P,SO,S,true) :-
    islist(P),
    !,
    append(P,S,SO).
right_hand_side(P,SO,S,G) :- tag(P,S0,S,G).

tag(P,S0,S,G) :- atom(P), G =..[P,S0,S].

and(true,G,G) :-!.
and(G,true,G) :- !. and(Gl,G2,(Gl,G2)).

islist([]) :- !. islist(LU).

append([A|B],C,[A|D]) :- append(B,C,D).
append([],X,X).
```

In this program, variables beginning with `P` stand for phrase descriptions (atoms, or lists of words) in grammar rules. Variables beginning with `G` stand for Prolog goals. Variables beginning with `S` stand for arguments of the Prolog goals (which represent sequences of words). In case you are interested, there follows a program that will handle the more general cases of grammar rule translation. One way in which a Prolog system can handle grammar rules is to have a modified version of `consult`, in which a clause of the form `A --> B` is translated before it is added to the database. We have defined a pair of operators to act as curly brackets "{" and "}", but some Prolog implementations may have built-in definitions, so that the term `{X}` is another form of the structure `'{}'(X)`.

```
 ?- op(1101,fx,").
?- op(1100,xf,").
?- op(1199,xfx,-->).

translate((P0-->Q0),(P:-Q)) :-
    left_hand_side(P0,S0,S,P),
    right_hand_side(Q0,S0,S,Q1),
    flatten(Q1,Q).

left_hand_side((NT,Ts),S0,S,P) :-
    !,
    nonvar(NT),
    islist(Ts),
    tag(NT,S0,S1,P),
    append(Ts,S,S1).
left_hand_side(NT,S0,S,P) :-
    nonvar(NT), tag(NT,S0,S,P).

right_hand_side((X1,X2),S0,S,P) :-
    !,
    right_hand_side(X1,S0,S1,P1),
    right_hand_side(X2,S1,S,P2),
    and(P1,P2,P).
right_hand_side((X1;X2),S0,S,(P1;P2)) :-
    !, or(X1,S0,S,P1), or(X2,S0,S,P2).
right_hand_side(P,S,S,P) :- !.
right_hand_side(!,S,S,!) :- !.
right_hand_side(Ts,S0,S,true) :-
    islist(Ts),
    !,
    append(Ts,S,S0).
right_hand_side(X,S0,S,P):- tag(X,S0,S,P).

or(X,S0,S,P) :-
    right_hand_side(X,S0a,S,Pa),
    ( var(S0a), S0a = S, !,
        S0=S0a, P=Pa; P=(S0=S0a,Pa) ).

tag(X,S0,S,P) :-
    X =.. [F|A], append(A,[S0,S],AX), P =.. [F|AX].

and(true,P,P) :- !.
and(P,true,P) :- !.
and(P,Q,(P,Q)).

flatten(A,A) :- var(A), !.
flatten((A,B),C) :- !, flattenl(A,C,R), flatten(B,R).
flatten(A,A).

flattenl(A,(A,R),R) :- var(A), !.
flattenl((A,B),C,R) :-
    !, flattenl(A,C,R1), flattenl(B,R1,R).
flattenl(A,(A,R),R).

islist([]) :- !.
islist([_|_]).

append([A|B],C,[A|D]) :- append(B,C,D).
append([],X,X).
```

Exercise 9.2

in back of book

The definition of the general version of `phrase/2` is as follows:

```
phrase(Ptype, Words) :-
    Ptype =.. [Pred|Args],
    append(Args, [Words, []],Newargs),
    Goal =.. [Pred|Newargs],
    call(Goal).
```

where `append/3` is defined as in Section 3.6, i.e.

```
append([], L, L).
append([X|L1], L2 [X|L3]) :- append(L1, L2, L3).
```

Exercise 9.3

This question seems wrong to me and I am probably mis-understanding it.

Given a sequence of tokens, it *is* possible to write a userful parser using DCG. In fact, the entire purpose of DCG is to parse sentences of the specified grammar, so I am confused why this question states otherwise.

Exercise 9.4

We run the given sample goals, then clarify the output by renaming variables and adding whitespace.

every man loves a woman

```
?- sentence(X, [every,man,loves,a,woman], []).
X = all(_1746, man(_1746)->exists(_1764, woman(_1764)&loves(_1746, _1764))) ;
false.
```

```
all(
    X,
    man(X) -> exists(
        Y,
        woman(Y) & loves(X, Y)
    )
)
```

Note that the program only produces the first of two possible interpretations: 1. there is a single woman that every man loves 2. there are (possibly) different woman that each man loves

This is because the program assumes all nouns are singular and so interprets "a woman" to be a single person, rather than a member of a group.

every man that lives loves a woman

```
?- sentence(X, [every,man,that,lives,loves,a,woman], []).
X = all(_3148, man(_3148)&lives(_3148)->exists(_3176, woman(_3176)&loves(_3148, _3176))) ;
false.
```

```
all(
    X,
    man(X) & lives(X) -> exists(
        Y,
        woman(Y) & loves(X, Y)
    )
)
```

every man that loves a woman lives

```
?- sentence(X, [every,man,that,loves,a,woman,lives], []).
X = all(_4562, man(_4562)&exists(_4586, woman(_4586)&loves(_4562, _4586))->lives(_4562)) ;
false.
```

```
all(
    X,
    man(X) & exists(
        Y,
        woman(Y) & loves(X, Y)
    ) -> lives(X)
)
```