Problem 1

The current directory contains files in C or C++ (that is to say, files with extension .c or .cpp or .c++). We need a Linux command to print – sorted lexicographically by file names – all lines in those files which contain the keyword `switch` and an opening parenthesis. The only command which does that is:

A	`grep "(" | grep switch *.c *.cpp *.c++ | sort`
B	`grep "(" *.c* | sort | grep switch`
C	`grep switch *.c* | grep "(" | sort`
D	`grep switch *.c *.cpp *.c++ | sort | grep "("`

Problem 2

Consider the following predicate definition and query in Prolog:

```
p2(X, L) :- append(_ ,[_, X, _, _ | _], L).
?- p2(X, [2, 3, 5, 7, 11, 13]).
```

If we keep pressing the semicolon, the query (before printing "false") will produce:

A	2 results
B	3 results
C	4 results
D	5 results

Problem 3

Consider the following predicate definition and query in Prolog[1]:

```
age(jean, 10).    age(bill, 23).    age(sam, 17).    age(paul, 5).
?- findall(X, age(X, Y), L).
```

The result of this query will be:

A	`L = [bill, jean, paul, sam].`

B	`Y = 5,`
	`L = [paul] ;`
	`Y = 10,`
	`L = [jean] ;`
	`Y = 17,`
	`L = [sam] ;`
	`Y = 23,`
	`L = [bill].`

C	`L = [jean, bill, sam, paul].`
D	`L = [10, 23, 17, 5].`

[1] **findall(Template, Goal, Bag)**: Creates a list of the instantiations which Template gets successively on backtracking over Goal, and unifies the result with Bag.

Problem 4

Only one of the four predicates in Prolog, defined below, correctly adds all integers between two given numbers, so that for example (given that 2+3+4+5=14), we get

```
?- addUp(2, 5, Result).
Result = 14 .
```

The correct definition is:

A	`addUp(N, N, N).`
	`addUp(L, H, T) :-`
	`  L < H,`
	`  LL is L + 1,`
	`  addUp(LL, H, TT),`
	`  T is L + TT.`
		B	`addUp(N, N, 0).`
			`addUp(L, H, T) :-`
			`  L < H,`
			`  LL is L + 1,`
			`  addUp(LL, H, TT),`
			`  T is L + TT.`

C	`addUp(N, N, N).`
	`addUp(L, H, T) :-`
	`  L < H,`
	`  LL is L + 1,`
	`  addUp(LL, H, TT),`
	`  TT is L + T.`
		D	`addUp(N, N, 0).`
			`addUp(L, H, T) :-`
			`  L < H,`
			`  LL is L + 1,`
			`  addUp(LL, H, TT),`
			`  TT is L + T.`

Problem 5

We wish to implement in Prolog the following conditional assignment, and to do it *most accurately and efficiently*:

```
if (X < 10) Y = 10;
else if (X < 100) Y = 100;
else Y = 0;
```

The way to do it is:

A	`f(X,  0) :- !.`
	`f(X,  10) :- X <  10, !.`
	`f(X, 100) :- X < 100.`
		B	`f(X,  10) :- X <  10.`
			`f(X, 100) :- X < 100.`
			`f(X,  0).`

C	`f(X,  10) :- X <  10, !.`
	`f(X, 100) :- X < 100, !.`
	`f(X,  0).`
		D	`f(X,  10) :- !, X <  10.`
			`f(X, 100) :- !, X < 100.`
			`f(X,  0).`

Problem 6

Consider the following procedure definition in Scheme:

```
(define sch6
  (lambda (l1 l2)
    (cond
      ((and (null? l1) (null? l2)) ())
      ((null? l1) (if (number? (car l2))
                      (cons (car l2) (sch6 l1 (cdr l2)))
                      (sch6 l1 (cdr l2))
      )           )
      ((number? (car l1)) (cons (car l1) (sch6 (cdr l1) l2)))
      (else (sch6 (cdr l1) l2))
) ) )
```

Now, consider the following expression:

```
(sch6 '(1 5 a 7 g) '(3 b 3 c))
```

The value of this expression is:

A	(a g b c)
B	(a b c g)
C	(1 3 3 5 7)
D	(1 5 7 3 3)

Problem 7

Assume a sorting procedure in Scheme, such as `sort` from lab 10, which takes a comparison predicate and a list, for example like this:

```
(sort char>? '(#\c #\a #\n #\t #\h #\i #\s #\b #\e))
```

Now, consider this expression:

```
(let
  ( (q (lambda (x) (* (car x) (cdr x)))) )
  (sort
    (lambda (x y) (< (q y) (q x)))
    '((1 . 9)(2 . 7)(3 . 5)(4 . 3)(5 . 1))
) )
```

The value of this expression is:

A	((3 . 5) (2 . 7) (4 . 3) (1 . 9) (5 . 1))
B	((5 . 1) (1 . 9) (4 . 3) (2 . 7) (3 . 5))
C	((1 . 9) (2 . 7) (3 . 5) (4 . 3) (5 . 1))
D	((5 . 1) (4 . 3) (3 . 5) (2 . 7) (1 . 9))

Problem 8

We need a definition in Scheme of a procedure which inserts an integer into a *strictly ordered* list of integers, without repetitions. Examples:

```
> (sch8 0 '(1 3 5 7))
(0 1 3 5 7)
> (sch8 4 '(1 3 5 7))
(1 3 4 5 7)
> (sch8 5 '(1 3 5 7))
(1 3 5 7)
> (sch8 8 '(1 3 5 7))
(1 3 5 7 8)
```

The only **incorrect** definition of procedure `sch8` is:

A	`(define sch8`
	`  (lambda (x l)`
	`  (cond`
	`  ((null? l) (cons x ()))`
	`  ((> x (car l)) (cons x l))`
	`  ((< x (car l)) (cons (car l) (sch8 x (cdr l))))`
	`  (else l)`
	`) ) )`

B	`(define sch8`
	`  (lambda (x l)`
	`  (cond`
	`  ((null? l) (cons x ()))`
	`  ((< x (car l)) (cons x l))`
	`  ((> x (car l)) (cons (car l) (sch8 x (cdr l))))`
	`  (else (sch8 x (cdr l)))`
	`) ) )`

C	`(define sch8`
	`  (lambda (x l)`
	`  (cond`
	`  ((null? l) (cons x ()))`
	`  ((< x (car l)) (cons x l))`
	`  ((> x (car l)) (cons (car l) (sch8 x (cdr l))))`
	`  (else l)`
	`) ) )`

D	`(define sch8`
	`  (lambda (x l)`
	`  (cond`
	`  ((null? l) (cons x ()))`
	`  ((= x (car l)) l)`
	`  ((> x (car l)) (cons (car l) (sch8 x (cdr l))))`
	`  (else (cons x l))`
	`) ) )`

Problem 9

We need a definition in Scheme of a procedure which calculates the dot product of two vectors **x** and **y** ($x_1y_1 + x_2y_2 + ... + x_ny_n$), represented as lists. Here is an example:

```
> (sch9 '(1 -3 0 5 3) '(3 1 8 0 7))
21
```

There is only one correct definition of procedure `sch9` below (two cause execution errors, one returns a wrong result).[2] The correct definition is:

A	`(define sch9`
	`  (lambda (v1 v2)`
	`  (eval '+ (map * v1 v2)`
	`) ) )`

B	`(define sch9`
	`  (lambda (v1 v2)`
	`  (apply + (map * v1 v2))`
	`) )`

C	`(define sch9`
	`  (lambda (v1 v2)`
	`  (+ (map * v1 v2))`
	`) )`

D	`(define sch9`
	`  (lambda (v1 v2)`
	`  (map + (map * v1 v2)`
	`) ) )`

Problem 10

Consider the following definition in Scheme:

```
(define sch10?
  (lambda (x)
    (cond
      ((null? x) #f)
      ((null? (cdr x)) #f)
      ((< (car x) (cadr x)) (sch10 (cdr x)))
      (else #t)
) ) )
```

Procedure `sch10?` checks that a given list of numbers:

A	has at least two elements and is not in strictly ascending order
B	has at least two elements and is in strictly ascending order
C	is not empty and no two adjacent elements are identical
D	is not empty and has two adjacent identical elements

[2] Hint: determine first the value of `(map * v1 v2)` for the two given lists.

Problem 11

We have a list of triples `#(item lowscore highscore)`. We need a procedure in Scheme which finds a triple with the minimal amplitude, for example like this:

```
> (sch11  '(#(one 4 21) #(two 5 19) #(ten 11 25)))
#(two 5 19)
```

Procedure `sch11` uses `my-neat-extr` from the Scheme notes (see Problem 14 for its definition). The only correct definition of procedure `sch11` is:

A	`(define sch11 (lambda (lst)`
	`  (my-neat-extr (lambda (x y)`
	`  (< (- (vector-ref x 1) (vector-ref x 2))`
	`  (- (vector-ref y 1) (vector-ref y 2))))`
	`  lst`
	`) ) )`

B	`(define sch11 (lambda (lst)`
	`  (my-neat-extr (lambda (x y)`
	`  (< (- (vector-ref x 2) (vector-ref x 1))`
	`  (- (vector-ref y 2) (vector-ref y 1))))`
	`  lst`
	`) ) )`

C	`(define sch11 (lambda (lst)`
	`  (my-neat-extr (lambda (x y)`
	`  (< (- (vector-ref y 2) (vector-ref y 1))`
	`  (- (vector-ref x 2) (vector-ref x 1))))`
	`  lst`
	`) ) )`

D	`(define sch11 (lambda (lst)`
	`  (my-neat-extr (lambda (x y)`
	`  (< (- (vector-ref x 1) (vector-ref y 1))`
	`  (- (vector-ref x 2) (vector-ref y 2))))`
	`  lst`
	`) ) )`

Problem 12

Consider the following procedure definition in Scheme:

```
(define sch12
  (lambda (test l1 l2)
    (cond
      ((null? l1) l2)
      ((null? l2) l1)
      ((equal? (car l1) (car l2)) (sch12 test (cdr l1) l2))
      ((test (car l1) (car l2))
        (cons (car l1) (sch12 test (cdr l1) l2)))
      (else
        (cons (car l2) (sch12 test l1 (cdr l2))))
) ) )
```

The evaluation of the expression

```
(sch12 char<? '(#\g #\c #\b #\a) '(#\g #\e #\d #\c))
```

gives:

A	(#\g #\c #\b #\a #\g #\e #\d #\c)
B	(#\c #\b #\a #\g #\e #\d #\c)
C	(#\g #\e #\d #\c #\b #\a)
D	(#\a #\b #\c #\d #\e #\g)

Problem 13

Consider the following expression in Scheme:

```
(
 (lambda (x y) (if (x y) y (cons (cdr y) (car y))))
 (lambda (z) (< (car z) (cdr z)))
 '(17 . 6)
)
```

The value of this expression is:

A	(17 6)
B	(17 . 6)
C	(6 17)
D	(6 . 17)

Problem 14

Here is a procedure from the Scheme notes (*find an extremum given a comparison and a list*):

```
(define my-neat-extr
  (lambda (c x)
    (define my-extr-help
      (lambda (curr-extr tail)
        (cond
          ((null? tail) curr-extr)
          ((c (car tail) curr-extr)
            (my-extr-help (car tail) (cdr tail)))
          (else (my-extr-help curr-extr (cdr tail)))
    ) ) )
    (if
      (null? x)
      x
      (my-extr-help (car x) (cdr x))
) ) )
```

Now, consider the following expression:

```
(my-neat-extr (eval (read)) '("this" "is" "not" "what" "it" "seems"))
```

We evaluate it four times, with different input. The value `"this"` is calculated if the input is:

A	`string<?`
B	`string>=?`
C	`equal?`
D	`(lambda (x y) (not (equal? x y)))`

Problem 15

Assume the tree representation from the course handout on Pascal (file `treePackage.p`):

```
type infoType = integer;
     tree = ^treeNode;
     treeNode =
       record
         info: infoType;
         left, right: tree
       end;
```

Now consider the following procedure:

```
procedure pas15(t: tree);
begin
  if t = nil then
    writeln
  else
  if (t^.left = nil) and (t^.right = nil) then
    write(t^.info, ' ')
  else
  begin
    pas15(t^.left);
    pas15(t^.right)
  end
end;
```

For a given tree **T**, procedure `pas15` prints:

A	the internal nodes of **T**
B	the crown of **T** (its leaves)
C	the whole **T** by pre-order traversal
D	the information in the root of **T**

Problem 16

We need programs in Pascal to compute the norm of a complex number represented as two real numbers. One program does not do it. The program which does **not** compute the norm is:

A

```
var n, r, i: real;
procedure norm(r, i: real;
                var n: real);
begin
  n := sqrt(sqr(r) + sqr(i))
end;
begin
  readln(r, i);
  norm(r, i, n);
  writeln(n)
end.
```

B

```
var r, i: real;
function norm(r, i: real): real;
begin
  norm := sqrt(sqr(r) + sqr(i))
end;
begin
  readln(r, i);
  writeln(norm(r, i))
end.
```

C

```
var n, r, i: real;
procedure norm(r, i: real);
var n: real;
begin
  n := sqrt(sqr(r) + sqr(i))
end;
begin
  readln(r, i);
  norm(r, i);
  writeln(n)
end.
```

D

```
type realptr = ^real;
var r, i: real;
    n: realptr;
procedure  norm(r, i: real;
    n: realptr);
begin
  n^ := sqrt(sqr(r) + sqr(i))
end;
begin
  readln(r, i);
  new(n);
  norm(r, i, n);
  writeln(n^)
end.
```

Problem 17

Consider the following program in Pascal:

```
const
  maxindex = 10;
type
  VEC = array [1 .. maxindex] of integer;
var
  A: VEC;
  i: integer;

procedure pas17(n: integer; var T: VEC);
var i, aux: integer;
begin
  aux := T[maxindex];
  for i:= maxindex downto 2 do T[i] := T[i - 1];
  T[1] := aux;
  if n > 1 then pas17(n - 1, T);
end;

begin
  for i := 1 to maxindex do A[i] := maxindex - i;
  pas17(6, A);
  for i:= 1 to maxindex do write(A[i])
end.
```

The program prints:

A	4567890123
B	0123456789
C	5432109876
D	9876543210

Open questions—answer on the same page where you see the question.

Problem 18

Give two examples of things[3] which are more easily programmed in Prolog than in Java. Next, give two examples of things which are more easily programmed in Scheme than in Java. *Explain* your answer.

[3] I have left it deliberately vague. A "thing" may be an application, an algorithm. an operation – you get to choose.

Problem 19

Discuss the idea of procedures as first-class objects (in Scheme, and elsewhere if you know other languages which allow it). You may like it or dislike it, but either way you have to *explain* why.

Problem 20

Recall what you know about procedures in Scheme and predicates in Prolog. Discuss similarities and differences. Have a few examples to support your discussion, and in general *explain*.

Problem 21

Rank C++, Java, Pascal, Prolog and Scheme by *your* preference from the most to the least liked. I will not mark your ranking: all marks will be awarded for an *explanation*.