I. Language features

(a) [**20 marks**] Fill in the following table with either "yes" or "no" in each cell. When deciding whether a language has a particular feature, only consider the built-in (default) feature set, not features provided by extra modules, or added by the user.

	Haskell	**Java / C / C++**	**Prolog**
Backtracking			✓
Pattern recognition	✓		✓
Loops		✓	
Ad-hoc polymorphism		✓	✓
Mutable (changeable) variables		✓	
Recursion	✓	✓	✓
Higher order functions	✓		
Lambda expressions	✓		
Strong typing	✓	✓	
Output parameters		✓	✓

(b) [**1 mark**] Rewrite the following Haskell code in Prolog.

```
lst = [x | x <- [1..9], (p x)]
```

```
bagof( X, (for( X, 1, 9 ), p( X )), Lst ).
```

(c) [**4 marks**] Rewrite the following Java code in Prolog.

```
int f( int[] x ) {
    int l = 0;
    for (i = 0; i <= x.length; i++)
        switch (x[i]) {
            case 1:
            case 2:
                l += 2;
                break;
            case 3:
            case 4:
                l++;
                break;
            case 5:
                l--;
                break;
            default:
                break;
        }

    return l;
}
```

```
f( [], 0 ) .
f( [X|Xs], R ) :- f( Xs, L ),
                  ( ((X is 1; X is 2), R is L + 2, !);
                    ((X is 3; X is 4), R is L + 1, !);
                    ((X is 5),         R is L - 1, !);
                    (                  R is L)
                  ).
```

II. Consider the following Haskell code.

```
f x y = foldl (\k l -> (div k 2) + l) 0
              [a + b + c | a <- [2..x], b <- [y..a^2], c <- (reverse [a..b])]
```

(a) [**8 marks**] Rewrite this function in Prolog.

```
loopbody( X, Y, Sum ) :- for( A, 2, X ),
                         A2 is A * A,
                         for( B, Y, A2 ),
                         forrev( C, B, A ),
                         Sum is A + B + C.

forrev( X, Hi, Lo ) :- bagof( A, for( A, Lo, Hi ), Lst ),
                       reverse( Lst, R ),
                       member( X, R ).

f( X, Y, Res ) :- bagof( S, loopbody( X, Y, S ), Lst ),
                  fold( Lst, 0, Res ).

fold( [], K, K ) :- !.
fold( [L|Ls], K, Res ) :- Next is (K // 2) + L,
                          fold( Ls, Next, Res ).
```

(b) [**7 marks**] Rewrite this function in an imparative language of your choice (or pseudocode). YOU MAY NOT USE AN ARRAY OR A LIST IN THE IMPERATIVE VERSION.

```
int f( int x, int y ) {
    int total = 0;

    for (a = 2; a <= x; a++)
        for (b = y; b <= (a*a); b++)
            for (c = b; c >= a; c--)
                total = (total / 2) + (a + b + c);

    return total;
}
```

III. Prolog has a feature called *negation-by-failure*. That is, according to the predicate logic underpinning Prolog, the negation of a predicate occurs when the predicate fails.

(a) [**4 marks**] Write a function `not/1` that negates the truth value of its sole parameter. For example, given the below definition for `p/1`, then `not( p( a ) )` should always fail, and `not( p( c ) )` should always succeed. You may wish to make use of the built-in predicate `fail/0` which always fails. However you may **NOT** make use of the built-in `\+` operator.

```
p( a ).
p( b ).
```

```
not( P ) :- P, !, fail.
not( _ ).
```

(b) [**9 marks**] Java, C and C++ all have some version of a *string tokenizer*. A string tokenizer is a function that takes a string and (sometimes) some character to use as a delimiter (often defaults to whitespace), then returns, one-by-one, all the substrings of the original string that are set apart by that delimiter. Write a list tokenizer in Prolog called `tokenize/3` that can tokenize a list as follows. You must make use of the `append/3` predicate.

```
| ?- tokenize( X, [3,4,0,7,4,0,2,0,4,5,9,0,1], 0 ).

X = [3,4] ? a

X = [7,4]

X = [2]

X = [4,5,9]

X = [1]
```

```
listtok( X, Lst, T ) :- append( X, [T|_], Lst ),
                        not( member( T, X ) ).
listtok( X, Lst, T ) :- append( _, [T|L1], Lst ),
                        !,
                        (  listtok( X, L1, T );
                           append( _, [T|X], Lst ),
                           !,
                           not( member( T, X ) )
                        ).
```

(c) [**2 marks**] There is a function in Haskell that does this. What is it called? (alternatively: write a function in Haskell that will tokenize a string).

```
words
```

IV. Use the following database to answer both parts of this question.

```
a( 1, 2 ).
a( 2, 3 ).
a( 2, 4 ).
a( 6, 1 ).

b( 1, 2 ).
b( 1, 4 ).
b( 2, 4 ).
b( 3, 5 ).
b( 5, 6 ).
```

(a) [**9 marks**] Given the following definition for `c/2`, what is the result when the query `c( X, Y )` is made?

```
c( A, B ) :- a( A, B ).
c( A, B ) :- b( A, C ), c( C, B ).
```

```
X = 1       X = 1
Y = 2       Y = 3

X = 2       X = 1
Y = 3       Y = 4

X = 2       X = 3
Y = 4       Y = 1

X = 6       X = 5
Y = 1       Y = 1
```

(b) [**9 marks**] If we now change the definition of `c/2`, to the following, what is the result when the query `c( X, Y )` is made?

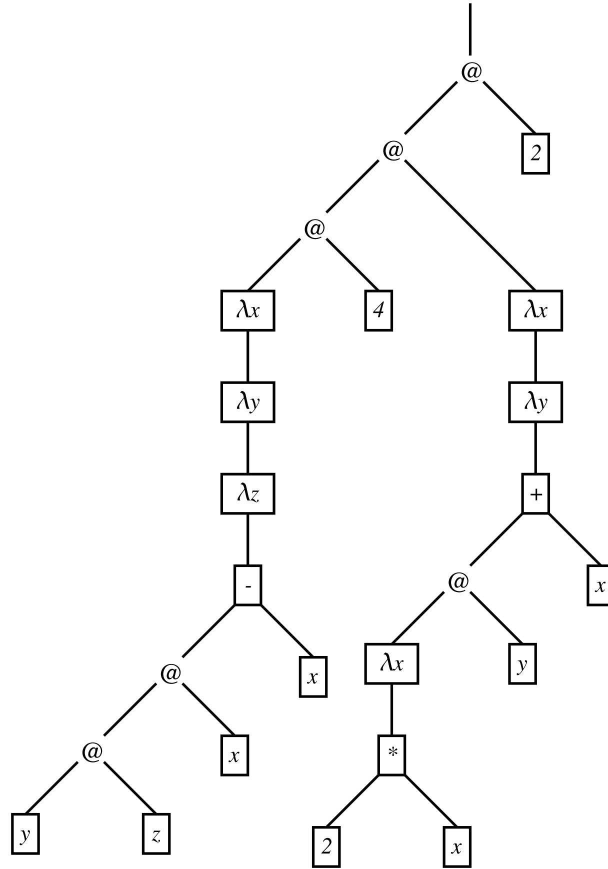
```
c( A, B ) :- a( A, B ).
c( A, B ) :- b( A, C ), !, c( C, B ).
```

```
X = 1       X = 1
Y = 2       Y = 3

X = 2       X = 1
Y = 3       Y = 4

X = 2
Y = 4

X = 6
Y = 1
```

V. Consider the following lambda expression.

$(\lambda xyz.(yzx) - x)\,4\,(\lambda xy.(\lambda x.2 * x)y + x)\,2$

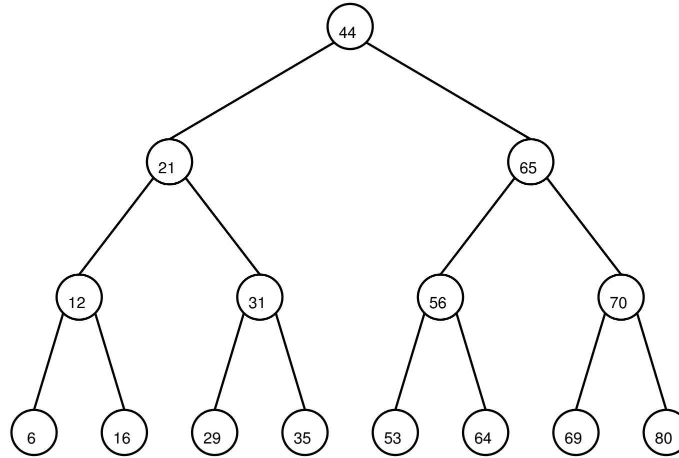
(a) [**7 marks**] Draw the tree resulting from this expression.



(b) [**5 marks**] Reduce this lambda expression as far as possible.

$$
\begin{aligned}
&(\lambda xyz.(yzx) - x)\ 4\ (\lambda xy.(\lambda x.2 * x)y + x)\ 2 \\
&\qquad \rightarrow_\beta (\lambda yz.(yz4) - 4)\ (\lambda xy.(\lambda x.2 * x)y + x)\ 2 \\
&\qquad \rightarrow_\beta (\lambda z.((\lambda xy.(\lambda x.2 * x)y + x)z4) - 4)\ 2 \\
&\qquad \rightarrow_\beta ((\lambda xy.(\lambda x.2 * x)y + x)\ 2\ 4) - 4 \\
&\qquad \rightarrow_\beta ((\lambda y.(\lambda x.2 * x)y + 2)4) - 4 \\
&\qquad \rightarrow_\beta ((\lambda x.2 * x)4 + 2) - 4 \\
&\qquad \rightarrow_\beta (2 * 4 + 2) - 4 \\
&\qquad = 6
\end{aligned}
$$

VI. A *binary search tree* (sometimes also called a *dictionary*) is a binary tree with the following property: for every node in the tree, all elements of the left subtree have a value less than that node, and all elements of the right subtree have a value greater than that node. See the following image for an example of a binary search tree (BST). The aim of this question is that you implement a working BST in Prolog. Note it is not required that a BST always be balanced. For this question, you should assume the recursive definition of a binary tree which is either `nil`, or `t(a,t1,t2)` where a is an integer and the left and right children $t1$ and $t2$ are themselves trees.



(a) [**5 marks**] Write a function `addnode/3` that takes a BST, and an integer, then adds that integer to the tree. If the integer is already in the tree, then the tree should just be returned unchanged.

```
addnode( nil, N, t( N, nil, nil ) ) :- !.
addnode( t( N, T1, T2 ), N, t( N, T1, T2 ) ) :- !.
addnode( t( A, T1, T2 ), N, t( A, New, T2 ) ) :- N < A,
                                                 addnode( T1, N, New ),
                                                 !.
addnode( t( A, T1, T2 ), N, t( A, T1, New ) ) :- N > A,
                                                 addnode( T2, N, New ),
                                                 !.
```

(b) [**4 marks**] Write a function `searchnode/2` that takes a BST, and an integer and succeeds if that integer is in the tree. `searchnode` should fail if the integer is not in the tree.

```
searchnode( t( N, _, _), N ) :- !.
searchnode( t( A, T1, _), N ) :- N < A,
                                 searchnode( T1, N ),
                                 !.
searchnode( t( A, _, T2), N ) :- N > A,
                                 searchnode( T2, N ),
                                 !.
```

(c) [**4 marks**] Write a function `listtotree/2` that takes a list of integers, and creates a BST out of them. That is, it should add each element of the list, one-by-one, to a tree which is initialized to `nil` (the empty tree).

```
listtotree( Lst, T ) :- listadd( Lst, Nil, T ).
listadd( [], T, T ) :- !.
listadd( [A|As], T, R ) :- addnode( T, A, T1 ),
                           listadd( As, T1, R ).
```

(d) [**1 mark**] Write a function `searchlist/2` that takes a list of integers, and an integer, and succeeds if that integer is in the list, failing otherwise.

```
searchlist( Lst, N ) :- listtotree( Lst, T ),
                        searchnode( T, N ).
```

VII. Unification in Prolog.

(a) [**7 marks**] Examine the following Prolog code:

```
{1} foo( A, B, C ) :-
{2}     D = A,
{3}     E =.. [C, A, 2],
{4}     G = a( B, E ),
{5}     a( 1, D ) = G.
```

If one were to pose the query: `foo( X, Y, b )`, would this succeed? If so, then what will be the value of `Y`? If it were to fail, why would it fail?

Occurs check fails in line 5 when comparing `X` with `b( X, 2 )`.

(b) [**1 mark**] Change the operator so the following query succeeds:
`2 = 1+1.`

`2 is 1+1.`

(b) [**1 mark**] Change the operator so the following query succeeds:
`1 = 3/3.`

`1 =:= 3/3.`

(b) [**1 mark**] Change the operator so the following query succeeds:
`'A' =< b.`

`'A' @=< b.`

VIII. Examine the following Prolog code:

```
f(X,Y) :- X > 0,
          f(X,Y,2).

f(1,[],_) :- !.
f(X,[Y|Ys],Y) :- Z is X // Y,
                 X =:= Z * Y,
                 !,
                 f(Z,Ys,Y).
f(X,Y,Z) :- g(X,Z,W),
            f(X,Y,W).

g(_,2,3) :- !.
g(X,Y,Z) :- Y * Y < X,
            !,
            Z is Y + 2.
g(X,_,X).
```

[**4 marks**] What is the value of `X` after posing the query `f( 3, X ).`?

```
X = [3]
```

[**4 marks**] What is the value of `X` after posing the query `f( 4, X ).`?

```
X = [2,2]
```

[4 marks] What is the value of `X` after posing the query `f( 5, X ).`?

```
X = [5]
```

[4 marks] What is the value of `X` after posing the query `f( 6, X ).`?

```
X = [2,3]
```