

**HONG KONG UNIVERSITY OF SCIENCE & TECHNOLOGY**  
**COMP3031 (Principles of Programming Languages)**

**Fall 2016**

**FINAL EXAMINATION**

08:30AM - 11:30AM  
Dec 8, 2016 Thursday  
LG5 Multi-function Room

|                   |                     |
|-------------------|---------------------|
| <b>Name</b>       |                     |
| <b>Student ID</b> | <b>ITSC Account</b> |

1. *About the exam:*
  - a. *This is a closed-book, closed-note examination.*
  - b. *You CANNOT use any electronic devices including calculators during the examination. Please TURN OFF all of your electronic devices (e.g., mobile phone) and put them into your bag.*
  - c. *You CANNOT leave during the last 15 minutes of the examination.*
2. *About this paper:*
  - a. *This paper contains 12 pages, including this title page.*
  - b. *The total number of points is 100, distributed to seven problems.*
3. *About your answers:*
  - a. *Write your answers in the designated space following each question.*
  - b. *Make sure your final answers are clearly recognizable.*
  - c. *Attempt all questions.*

|         |   |   |   |   |   |   |   |       |
|---------|---|---|---|---|---|---|---|-------|
| Problem | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Total |
| Marks   |   |   |   |   |   |   |   |       |

## Problem 1. SML Programming (20 points)

Given the following SML datatypes:

```
datatype node = Nd of int * int list; (*node ID and list of children*)
datatype tree = Tr of node list;
```

Example:

```
val t =
Tr [Nd(0, [3,5]), Nd(2, [0,1]), Nd(1, [6]), Nd(3, [4]), Nd(4, []),
Nd(5, []), Nd(6, [])];
```

In this tree `t`, we can see that the node labeled 2 is the root, with nodes labeled 0 and 1 as children. **Assume all node labels are unique.**

- a) Write a function `depth` to calculate the depth of a node, i.e., the number of edges to traverse from the root to this node. If the node does not exist in the tree, return `~1`.

```
val depth = fn : int * tree -> int
```

Examples:

```
- depth(0, t);
val it = 1 : int
- depth(2, t);
val it = 0 : int
- depth(1, t);
val it = 1 : int
- depth(3, t);
val it = 2 : int
- depth(4, t);
val it = 3 : int
- depth(7, t);
val it = ~1 : int
```

- b) Write a function `path` to detect whether there exists a path from a node labeled `n1` to a node labeled `n2`.

```
val path = fn : int * int * tree -> bool
```

Examples:

```
- path(0, 2, t);  
val it = false : bool  
- path(2, 0, t);  
val it = true : bool  
- path(0, 4, t);  
val it = true : bool  
- path(2, 4, t);  
val it = true : bool  
- path(0, 6, t);  
val it = false : bool  
- path(1, 0, t);  
val it = false : bool  
- path(6, 7, t);  
val it = false : bool  
- path(8, 7, t);  
val it = false : bool
```



## Problem 2. Prolog Programming (20 points)

Given a knowledge base of facts in the form of `edge(X, Y)`, where X, Y are the labels of two tree nodes. Each fact represents a directed edge in a binary tree and each node label is unique:

```
edge(a,b).  
edge(a,c).  
edge(b,d).  
edge(b,p).  
edge(c,e).  
edge(c,f).
```

- a) Define a predicate `hop(X, Y, H)` in which X, Y are the labels of two tree nodes, and H is the length of the path from node X to node Y. The code skeleton is given. You only need to fill in the missing predicates, one predicate per blank.

Examples:

```
?- hop(a,b,X).  
X = 1.  
?- hop(b,a,X).  
false.  
?- hop(a,f,X).  
X = 2.  
?- hop(a,X,1).  
X = b ;  
X = c ;  
false.
```

```
hop(X,Y,1):- _____ .
```

```
hop(X,Y,H):- _____ ,
```

```
_____ ,
```

```
_____ .
```

- b) Define a predicate `lca(X, Y, A)` such that for a given pair of nodes X and Y, A is the lowest common ancestor of the two nodes. A common ancestor means there is a path from A to X as well as a path from A to Y. The lowest common ancestor means except A, there is no other common ancestor of X and Y on the path A->X and path A->Y. The code skeleton is given. You only need to fill in the missing predicates, one predicate per blank.

Examples:

```
?- lca(b,b,X) .
X = a.
?- lca(b,d,X) .
X = b.
?- lca(e,f,X) .
X = c.
?- lca(d,p,X) .
X = b.
?- lca(e,X,a) .
X = a ;
X = b ;
X = d ;
X = p ;
false.
```

```
ancestor(A, B) :- edge(A, B) .

ancestor(A, B) :- _____ ,
                    _____ .

lca(A, B, A) :- ancestor(A,B) .

lca(A, B, B) :- _____ .

lca(A, B, X) :- _____ ,
                _____ ,
                not( (ancestor(X,Y) ,
                    ancestor(Y,A) ,
                    ancestor(Y,B)
                    ) ) ,
                not( (ancestor(A,B) ,
                    ancestor(B,A)
                    ) ) .
```

### Problem 3. Cuts and Negation in Prolog (10 points)

Given the following Prolog database, write *\*all\** the answers to each of the query a) – e):

```
teach(frank, physics).  
teach(fred, history).  
teach(fred, english).  
enroll(anne, english).  
enroll(alice, english).  
enroll(alice, physics).
```

a)

```
?- enroll(X, Y), !, teach(fred, Y).
```

b)

```
?- enroll(X, Y), \+teach(fred, Y), !.
```

c)

```
?- enroll(X, Y), !, \+teach(fred, Y).
```

d)

```
?- \+enroll(_, X), teach(_, X), !.
```

e)

```
?- teach(_, X), \+enroll(_, X), !.
```

#### Problem 4. Prolog Search Tree (10 points)

Consider the following program:

```
/*R1*/ t(iris).  
/*R2*/ t(aloe).  
/*R3*/ i(rose).  
/*R4*/ j(lily).  
/*R5*/ j(tulip).  
/*R6*/ f(X,Y) :- p(X,Y).  
/*R7*/ f(X,Y) :- t(Y), !, j(X).  
/*R8*/ p(X,Y) :- i(X), j(Y), !.
```

Draw the complete Prolog search tree for the query  $f(X, Y)$ , giving **all** answers. At each **tree edge**, whenever applicable, **i)** indicate the rule number  $R_i$ ,  $i=1,\dots,8$ , of the rule being applied, and **ii)** the unification(s) being made. At each **tree node** indicate the goal to be satisfied. At each leaf node indicate “succeed” or “fail”. The initial step has been done for you.

$f(X, Y)$   
R6:  $\{\_1/X, \_2/Y\}/$   
 $p(X, Y)$

### Problem 5. Flex and Bison (20 points)

Given the following grammar for bit expressions:

```
<S> ::= <S> and <S1> | <S> or <S1> | <S> xor <S1>
<S1> ::= ( <S> ) | <NUM>
<NUM> ::= 0 | 1
```

Complete the following Flex and Bison files to evaluate the bit expressions. Some examples:

Input : 1 or 0

Output : 1

Input : 1 and 0

Output : 0

Input : 1 xor 0

Output : 1

Input : 1 xor 1

Output : 0

Input : 1 xor 0 and 1

Output : 1

Input : 1 xor ( 0 or 1 )

Output : 0

Flex file “logic.lex”:

```
%option noyywrap
%{
#define YYSTYPE int
#include "logic.tab.h"
#include <stdio.h>
#include <stdlib.h>
%}
num    0|1
lb     [ (]
rb     [ ) ]
ws     [ \t ]+
%%
{num} { _____; return NUM; }
{lb}  return LB;
{rb}  return RB;
"and" _____;
"xor"  _____;
"or"   _____;
{ws}
"\n"  return *yytext;
%%
```



### Bison file "logic.y":

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
int yyerror(const char*);
%}

%token      NUM AND OR XOR LB RB

%%
/* Fill in the blanks in the grammar rules and actions*/
input: input line
    | line
    ;
line: '\n'
    | expr '\n' { printf("res : %d\n", $1); };
expr:
    expr AND exprh { _____ }
    | expr OR exprh { _____ }
    | expr XOR exprh { _____ }
    | exprh { _____ }
    ;
exprh:
    NUM { _____ }
    | LB expr RB { _____ }
    ;
%%
int main() { return yyparse(); }
int yyerror(const char* s) {
printf("error \n");
return 0;
}
```

### Problem 6. Parameter Passing Methods (10 Points)

The following program is in an imaginary D language, which has a syntax similar to the C language, but can apply static or dynamic scoping as well as various parameter passing methods as we specify. Determine the output of the following D program with each specified scoping and parameter passing method:

```
int i, j;  
i = j = 1;  
void calc(int x, int y)  
{  
    int i = 5;  
    y = y + (x++);  
    x = i + y;  
    printf("%d ", i + j);  
}  
int main(int argc, char **argv)  
{  
    calc(i, j);  
    int j = 6;  
    calc(i, j);  
    printf("%d ", i + j);  
}
```

Static scoping, call by value:

Static scoping, call by reference:

Static scoping, call by value-result:

Dynamic scoping, call by name:

### Problem 7. Activation Records (10 points)

Recall that the C language by default uses static scoping on variable names and passing-by-value for parameter passing in procedure calls. Complete the activation records, including the variables and their values if known, the parameters and their values if known, and the control links for the following C program at the specified point in time:

- (i) right before calling main;
- (ii) right before calling conv;
- (iii) right before exiting the call of conv;
- (iv) right before exiting main;

```
#include <stdio.h>

int x[3] = {1,3,5};
int y[2] = {-2,-3};

void conv(int a[], int len, int b[]){
    int i,j,temp;
    for(i=0;i<4 - len;i++){
        temp = 0;
        for(j=i;j<i+len;j++){
            temp += a[j-i] * x[j];
        }
        b[i] = y[i] + temp;
    }
    return;
}

int main(int argc, char const *argv[])
{
    int x[2] = {1,2}, z[2] = {10,20};
    conv(x,2,z);
    printf("z[0] : %d, z[1] : %d\n", z[0],z[1]);
    return 0;
}
```

Activation Records: