

**Trial exam questions  
for June 2002**

Version 1.0 of 2002-06-07

**Question 1 (25 %)**

Binary trees of integers can be represented as values of type `inttree`:

```
datatype inttree =
  Lf
  | Br of int * inttree * inttree;
```

The intention is that `Lf` represents an empty tree, and `Br(x, t1, t2)` represents a tree with root node value `x`, left subtree `t1` and right subtree `t2`.

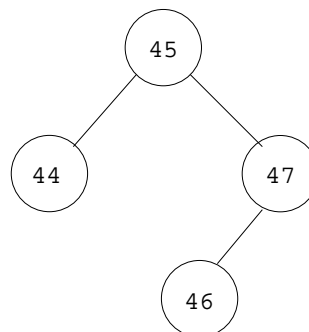
**Question 1.1**

Draw the trees represented by the SML values bound to variables `ta` and `tb`:

```
val ta = Br(23, Lf, Lf);
val tb = Br(42, ta, Br(56, Lf, Br(78, Lf, Lf)));
```

**Question 1.2**

Write an SML expression that represents this tree:



**Question 1.3**

The following function `doubl : inttree -> inttree` multiplies the root node value (if any) of the tree by 2:

```
fun doubl Lf = Lf
  | doubl (Br(y, t1, t2)) = Br(2 * y, t1, t2);
```

Modify the function so that it multiplies *all* node values in the tree by 2.

**Question 1.4**

A binary tree `Br(x, t1, t2)` is *ordered* if all node values in `t1` are strictly smaller than `x`, and all node values in `t2` are strictly greater than `x`, and the subtrees `t1` and `t2` themselves are ordered. The binary tree `Lf` is trivially ordered. For instance, all three example trees above are ordered.

Define a function `contains : int * inttree -> bool` so that `contains(x, t)` returns true if `x` appears in `t`, and false otherwise. The tree `t` can be assumed to be ordered.

In outline, the function should look like this:

```
fun contains (x, Lf) = false
  | contains (x, Br(y, t1, t2)) = ...
```

**Question 1.5**

Define a function `insert : int * inttree -> inttree` so that `insert(x, t)` returns an ordered tree in which `x` has been inserted. If `x` appears in `t` already, the tree is unchanged. The tree `t` can be assumed to be ordered.

Define a function `insertlist : int list * inttree -> inttree` so that `insertlist(xs, t)` returns an ordered tree in which all elements of `xs` have been inserted, except those that appear in `t` already.

**Question 1.6**

Define a function `ordered : inttree -> bool` so that `ordered(t)` returns true if the tree `t` is ordered, and false otherwise.

**Question 2 (20 %)****Question 2.1**

The following regular expression recognizes certain strings consisting of the letters  $a$ ,  $b$  and  $c$ :

$$a((ab)|(ac))^*c$$

Indicate which of these five strings are recognized by the above regular expression:

$aacc$ ,  $abac$ ,  $ac$ ,  $ababababacac$ ,  $aabacc$

Also, show three more strings that are recognized by the above regular expression.

Finally, show three more strings, consisting only of the letters  $a$ ,  $b$  and  $c$ , that are *not* recognized by the above regular expression.

**Question 2.2**

Construct a nondeterministic finite automaton that recognizes exactly the same strings as the above regular expression.

**Question 2.3**

Here is part of the lexer specification for micro-Java:

```
rule Token = parse
  [' ' '\t' '\n' '\r'] { Token lexbuf }
| ['0'-'9']+           { case Int.fromString (getLexeme lexbuf) of
                        NONE => lexerError lexbuf "internal error"
                        | SOME i => CSTINT i
                        }
| ...
```

The second clause recognizes integers in decimal format. However, in Java, there are three different kinds of integer constants:

- decimal (base 10) constants that do not start with a zero, and which consist of the digits 0–9, for instance 117;
- octal (base 8) constants that start with a zero, and which consist of the digits 0–7, for instance 0117;
- hexadecimal (base 16) constants that start with 0x, and which consist of the digits 0–9, a–f, A–F, for instance 0x02BFCD.

Modify the above lexer specification to accept and distinguish these three kinds of integer constants. You can ignore the conversion from lexeme (string) to token (integer) in the semantic actions { ... }.

### Question 3 (20 %)

In a text-based dungeons-and-dragons game, a player may issue simple commands such as these:

```
buy dagger
cross chasm
pick up axe
slay dragon
kill guard
open door
examine coffin
get coins
```

and so on. As can be seen, every simple command consists of a verb and a noun which is the object of the verb.

In addition, composite commands are allowed, for instance:

```
get horse, and then cross chasm
pick up axe, and then slay dragon
pick up axe, and then slay dragon, and then drop axe
```

So a composite command consists of a command followed by a comma, the word *and*, and the word *then*, and another command.

Finally, to help create robot players, conditional commands are permitted:

```
if door is closed, climb ladder
if dragon is awake, pick up axe, and then slay dragon
```

A conditional command consists of the word *if* followed by a noun, the word *is*, an adjective, a comma, and a command (which may itself be composite or conditional).

#### Question 3.1

Write an informal grammar for commands. It should permit a command to be simple commands, composite, and conditional. You may assume that nonterminals *Verb*, *Noun*, and *Adjective* for verbs, nouns and adjectives.

Show at least one way to derive the command *if dragon is awake, pick up axe, and then slay dragon* from your grammar. You should rewrite only one nonterminal in every step.

If possible (if your grammar is ambiguous), show two distinct derivations of that command.

#### Question 3.2

In an abstract syntax for commands we can consider verbs, nouns, and adjectives to be simply strings. For instance, in SML:

```
type verb = string
type noun = string
type adjective = string
```

Write an SML function `isKillable : noun -> bool` that returns true if applied to a noun that represents something that can be alive (you may restrict this to be a dragon, guard, or horse), and returns false otherwise.

Also, write an SML function `isOpenable : noun -> bool` that returns true if applied to a noun that represents something that can be opened or closed (you may restrict this to be a window, door, or coffin), and returns false otherwise.

**Question 3.3**

Given the above definitions of `verb`, `noun` and `adjective`, an abstract syntax for commands can be written like this in SML:

```
datatype command =  
  Simple of verb * noun  
  | Compos of command * command  
  | Condit of noun * adjective * command
```

Write an SML function `nouns : command -> noun list` that returns a list of all the nouns that appear in the given command (and its subcommands), whether it is simple, composite, or conditional.

**Question 3.4**

Write an SML function `isMeaningful : command -> bool` that checks whether all parts of a command are meaningful, and returns true if they all are.

For simplicity we define ‘meaningful’ as follows: A simple command is meaningful if the verbs `open` and `close` are applied only to openable things, and if the verb `kill` is applied only to killable things. A composite command is meaningful if both subcommands are. A conditional command is meaningful if the adjective `closed` is applied only to openable things, and the adjectives `alive` and `dead` are applied only to killable things; moreover, the subcommand must be meaningful.

**Question 4 (35 %)**

This question concerns a simple stack machine for building binary trees such as those discussed in Question 1. The stack machine has a program  $p$ , a program counter  $pc$ , and a stack  $s$  holding integers and trees.

The instructions of the abstract machine, and their effect on the stack, are described by this table:

Instruction	Stack before	Stack after	Effect
CST $i$	$s \Rightarrow$	$i : s$	Push integer constant $i$
ADD	$i_2 : i_1 : s \Rightarrow$	$(i_1 + i_2) : s$	Add integers
SUB	$i_2 : i_1 : s \Rightarrow$	$(i_1 - i_2) : s$	Subtract integers
DUP	$v : s \Rightarrow$	$v : v : s$	Duplicate
SWAP	$v_2 : v_1 : s \Rightarrow$	$v_1 : v_2 : s$	Swap
POP	$v : s \Rightarrow$	$s$	Pop
GOTO $a$	$s \Rightarrow$	$s$	Jump to $a$
IFNZRO $a$	$v : s \Rightarrow$	$s$	Jump to $a$ if $v \neq 0$
CALL $a$	$v : s \Rightarrow$	$v : r : s$	Call function at $a$ , pushing return address $r$
RET	$v : r : s \Rightarrow$	$v : s$	Return: jump to $r$
PRINT	$v : s \Rightarrow$	$v : s$	Print $v$
STOP	$s \Rightarrow$	—	Halt the machine
MKLEAF	$s \Rightarrow$	Lf : $s$	Make leaf node Lf
MKBRANCH	$t_2 : t_1 : i : s \Rightarrow$	Br( $i, t_1, t_2$ ) : $s$	Make branch node Br( $i, t_1, t_2$ )

The stack machine may be implemented in Java as shown on the next page. The array  $p$  contains the program code, and execution starts at  $p[pc]$ . The stack contains Integer objects as well as Tree objects. The initial stack  $s[0..k-1]$  contains the  $k$  command line arguments.

The interpreter uses a common Java representation of trees:

```

abstract class Tree { }

class Leaf extends Tree {
    public String toString() {
        return "Lf";
    }
}

class Branch extends Tree {
    public final Integer i;
    public final Tree t1, t2;

    public Branch(Integer i, Tree t1, Tree t2) {
        this.i = i; this.t1 = t1; this.t2 = t2;
    }

    public String toString() {
        return "Br(" + i + ", " + t1 + ", " + t2 + ")";
    }
}

```

**Question 4.1**

Declare an SML datatype `instr` to represent the instructions of CST  $i$ , ADD, etc. of this machine.

**Question 4.2**

Show the tree that would be built and printed by executing this program:

```
CST 42, MKLEAF, MKLEAF, MKBRANCH, STOP
```

Also, write code to create and print the tree `tb` in Question 1.1a.

```

static int execcode(int[] p, Object[] s, int pc, int sp) {
    for (;;) {
        switch (p[pc++]) {
            case CST:
                s[sp+1] = new Integer(p[pc++]); sp++; break;
            case ADD:
                s[sp-1] = new Integer(((Integer)s[sp-1]).intValue() + ((Integer)s[sp]).intValue());
                sp--; break;
            case SUB:
                s[sp-1] = new Integer(((Integer)s[sp-1]).intValue() - ((Integer)s[sp]).intValue());
                sp--; break;
            case DUP:
                s[sp+1] = s[sp]; sp++; break;
            case SWAP:
                { Object tmp = s[sp]; s[sp] = s[sp-1]; s[sp-1] = tmp; } break;
            case POP:
                sp--; break;
            case GOTO:
                pc = p[pc]; break;
            case IFNZRO:
                pc = (((Integer)s[sp--]).intValue() != 0 ? p[pc] : pc+1); break;
            case CALL:
                s[sp+1] = s[sp]; s[sp] = new Integer(pc+1); sp++;
                pc = p[pc]; break;
            case RET:
                pc = ((Integer)s[sp-1]).intValue();
                s[sp-1] = s[sp]; sp--; break;
            case PRINT:
                System.out.println(s[sp] + " "); break;
            case MKLEAF:
                s[sp+1] = new Leaf(); sp++; break;
            case MKBRANCH:
                s[sp-2] = new Branch((Integer)s[sp-2], (Tree)s[sp-1], (Tree)s[sp]);
                sp -= 2; break;
            case STOP:
                return sp;
            default:
                throw new RuntimeException("Illegal instruction " + p[pc-1] + " at " + (pc-1));
        }
    }
}

```

### Question 4.3

Write stack machine code that creates this right-linear tree:  $Br(3, Lf, Br(2, Lf, Br(1, Lf, Lf)))$   
 Then write an SML function `rl : int -> instr list` that for a given integer  $n \geq 0$  generates stack machine code to build and print a right-linear tree with  $n$  branch nodes, of this form:

$$Br(n, Lf, Br(n-1, Lf, \dots, Br(2, Br(1, Lf, Lf)) \dots))$$

### Question 4.4

The machine instructions `CALL a` and `RET` are used to implement simple functions that take one argument (on the stack top) and return one argument (on the stack top).

What does this stack machine program compute, given that the integer 42 is on the stack top when execution is started at instruction address 0 (`CALL`)? Note that 4 is the address of the first `MKLEAF` instruction:

```

CALL 4, PRINT, STOP
MKLEAF, MKLEAF, MKBREAK, RET

```

### Question 4.5

Write a stack machine program that, given that  $n \geq 0$  is on the stack top, builds a right-linear tree of the form indicated in Question 4.2 above. Hint: you will need to use functions, implemented using `CALL` and `RET`.