UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Exam in:	**INF3110 Programming Languages**
Day of exam:	**December 1, 2011**
Exam hours:	**14:30 – 18:30**
This examination paper consists of 7 pages.	
Appendices:	**No**
Permitted materials:	**All printed and written**
	Textbook:
	John C Mitchell:
	Concepts in Programming Languages, 2003. Cambridge University Press. ISBN: 0-521-78098-5.

Make sure that your copy of this examination paper is complete before answering.

This exam consists of 3 questions that may be answered independently. If you think the text of the questions is unclear, make your own assumptions/interpretations, but be sure to write these down as part of the answer.

Good luck!

Contents

Question 1 Runtime-systems, scoping, types (weight 40%) 2
Question 2 ML (weight 40%) 7
Question 3 Prolog (weight 20%) 8

Question 1. Runtime-systems, scoping, types (weight 40%)

Ordinary object-oriented languages do not support the return of more than one value from a method. The return type may be a predefined type like `int` or `bool`, or a user-defined class in which case a reference to an object of that class is returned.

Suppose that we want to extend such a language with the possibility of returning in general a list of values, and that these values may be assigned to a list of variables.

Given the following two definitions of `equals`, with the definition of return types by means of a list notation:

```
class Point {
  int x,y;
  (bool) equals(Point p){
     return (x=p.x and y=p.y)}
}

class ColorPoint extends Point {
  Color c;
  (bool, Color) equals(ColorPoint cp) {
     return ((x=cp.x and y=cp.y and c=cp.c),c)}
}
```

and given variables declared like this:

```
bool b;
Color c;
Point p1 p2;
ColorPoint cp1, cp2;
```

it would then be possible to call them like this:

```
(b) = p1.equals(p2);
(b, c) = cp1.equals(cp2);
```

A method that takes a `bool` and a `Color` as parameters may then be called with a call to `equals` of a `ColorPoint` as an actual parameter. Suppose that the `main` method looks like this:

```
void main() {
  bool verdict(bool b, Color c){
    return (b and (c = colorOfTheDay()))
  };
  bool b = verdict(cp1.equals(cp2))
}
```

with a `verdict` method that will only return true if the two points are equal and if the color is the color that is returned by `colorOfTheDay()`.

1a

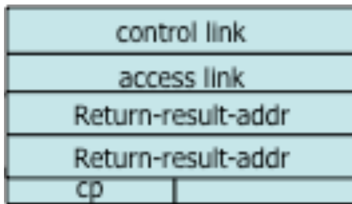
How must an activation record be extended in order to cope with this new mechanism?

Look at page 171 in Mitchell or slide 13 in the slides dated 9/14/11 (Runtime Organization I)

Sketch how an activation record will have to be.

Answer:

Two result addresses. E.g.

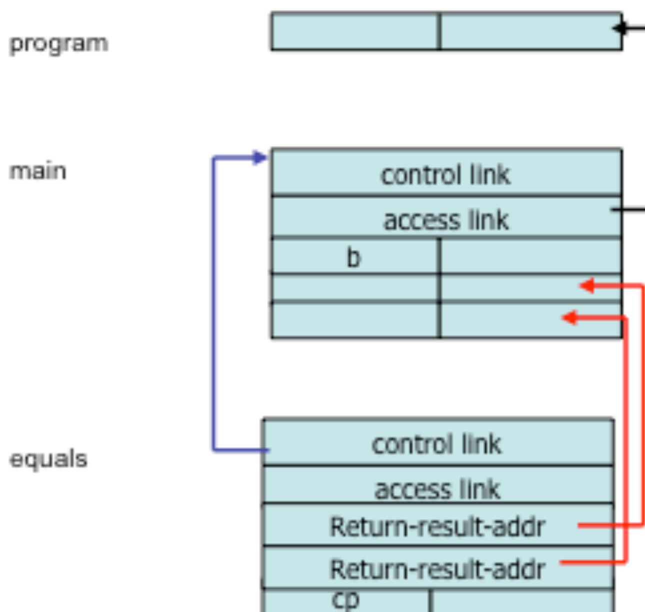


In general as many as there are return values.

1b

How will the stack of activation records be just before the `equals` method exit. Fill in all fields and links of the activation records.

Answer:



It is not important that they show that the two results have to taken care of in this way. The use of intermediate results has been covered, but not the use of registers (that is for the compiler course).

It will not be an error if they indicate that the access link of the equals refers a ColorPoint object denoted by cp1.

1c

We now assume that the `equals` method of `Point` is virtual so that may be redefined in `ColorPoint`, in the same way as in Java. The `equals` of `ColorPoint` will therefore have the same signature as `equals` of `Point`, i.e. is it has the same type `(Point)` of the parameter.

We allow covariant re-definition of the result to be a list `(bool, Color)`

```
class ColorPoint extends Point {
  Color c;
  (bool, Color) equals(Point p) {
    return if p instanceOf(Point)
               then ((x=p.x and y=p.y), white) else
           if p instanceOf(ColorPoint)
             then ((x=p.x and y=p.y and c=(ColorPoint)p.c),c)
  }
}
```

assuming that `(bool, Color)` is a subclass of `(bool)`, as if implicit classes were defined:

```
class BoolList {bool b}
class BoolColorList extends BoolList {Color c}
```

The list returned from `equals` has the same type as the list `(b,c)`:

```
(b,c) = cp1.equals(cp2)
```

where `cp1` and `cp2` are `ColorPoint` variables.

With this scheme, a `BoolList` can be assigned a `BoolColorList` (as `BoolColorList` is a subclass of `BoolList`):

```
(b) = (b1, c1)
```

with the obvious semantics that only the `b` is assigned a value (`b1`), as if the following assignment was performed:

```
b = implicitBoolColorListObject.b1
```

An obvious question would be: Why not allow the following:

```
(c) = (b1,c1)
```

However, given the above scheme of making implicit classes, the class `BoolColorList` is not a subclass of the implicit class `ColorList` for the list `(c)`. In order for this to be the case it would require multiple inheritance:

```
class ColorList {Color c}
class BoolList { bool b }
class BoolColorList extends (BoolList, ColorList) {}
```

Suppose that we do not want multiple inheritance, but rather do this by means of interfaces.

Sketch a solution, both how the interfaces should be, how the class `BoolColorList` should be, and what the semantics of

```
(c) = (b1,c1)
```

should be, in terms of using an operation in one of these interfaces. Assume that the list `(b1, c1)` is represented by an implicit object `implicitBoolColorListObject` of class `BoolColorList`.

Answer:

```
interface BoolListInf { bool getBool(); bool setBool(bool pb) }
interface ColorListInf { Color getColor(); Color setColor(Color pc)}
interface BoolColorListInf extends (BoolList, ColorList){}

class BoolColorList implements BoolColorListInf {
  bool b;
  Color c;

  bool getBool(){return b};
  bool setBool(bool pb) {p = pb}
  Color getColor(); {return c}
  Color setColor(Color pc) {c = pc}

}
```

Semantics of

```
(c) = (b1,c1)
```

is

c = implicitBoolColorListObject.getColor()

1d

Assume that we have following implementation of very simple expressions,

```
exp ::= const | exp + exp
```

in terms of an interface `Exp` with an `interpret` operation, and classes for the different kinds of expressions:

```
interface Exp {int interpret()}

class Const implements Exp {
  int value; // value of constant
int interpret() { return value; }
}
class Plus implements Exp {
  Exp first, second;
  int interpret() {
    return first.interpret() + second.interpret();
  }
}
```

We have deliberately omitted details like constructors that would be used to set he values of `value`, `first` and `second`.

When extending this implementation with a `formatter` operation, this is defined in a new interface that extends the `Exp` interface:

```
interface FExp extends Exp {
  String formatter();
}
class FConst extends Const implements FExp {
  String formatter() { return "" + value;}
}
class FPlus extends Plus implements FExp {
  String formatter() {
    return "(" +
```

```
        first.formatter() + " + " +       //   (a)
        second.formatter() + ")";         //   (b)
    }
  }
```

What is the problem in lines marked with `(a)` and `(b)`,

and how would you rather make these lines?

Answer:

```
    ((FExp)first).formatter() + " + " +
    ((FExp)second).formatter() + ")";
```

Question 2. ML (weight 40%)

2 (a)

Assume the following definitions. Recall that `#"G"` is ML's notation for single characters.

```
fun hd : 'a list -> a
fun hd (x::xs) = x ;
fun map : ('a -> 'b) -> 'a list -> 'b list
fun map f nil      = nil
  | map f (x::xs) = (f x) :: map f xs ;


fun h nil n          = nil
  | h ((x,c)::xs) n = if (x = n) then c :: h xs n else h xs n ;
```

Give the result of evaluating the following two expressions:

```
hd (map (fn x => x+1) [3*2,2]) ;
h [(0,#"y"),(3,#"G"),(5,#"r"),(2,#"u"),(1,#"r"),(3,#"O")] 3 ;
```

Answer:

```
1) 7
2) [#"G",#"O"]
```

2 (b+c)

The following data structure represents polymorphic binary trees that contain values only in special `Value` nodes that have a single successor:

```
datatype 'a tree = Leaf
                 | Node of ('a tree * 'a tree)
                 | Value of ('a*'a tree);
```

Consider the tree `t` of characters on the right-hand side. The representation of `t` as an object of type `char tree` in ML would be:

```
Node (Value (#"a", (Value (#"b", Leaf))),
      Node (Node (Leaf,Leaf), Value (#"c",Leaf)))
```

Implement the following functions in ML.

2b

The function `foldTree` of type

```
('a * 'b -> 'b) -> ('b * 'b -> 'b) -> 'b -> 'a tree -> 'b
```

works as follows: `foldTree f g h x` replaces all occurrences of the constructor `Value` in the tree `x` by `f`, it replaces all occurrences of the constructor `Node` in `x` by `g`, and it replaces all occurrences of the constructor `Leaf` in `x` by `h`. So for the tree `t` above,

```
foldTree (op ::) (op @) [] t
```

should compute

```
((@) (((::) (#"a",((::) (#"b",[])))),((@)  (((@) ([],[])),
                                            ((::) (#"c", []))))))
```

which in the end results in "abc" (i.e., in the list [#"a",#"b","c"]). Here, `Value` is replaced by `(::)`, `Node` is replaced by `(@)`, and `Leaf` is replaced by `[]`.

Answer:

```
fun foldTree f g h (Value (n,x)) = f (n,(foldTree f g h x))
  | foldTree f g h (Node (x,y))  = g ((foldTree f g h x),
                                     (foldTree f g h y))
  | foldTree _ _ h Leaf = h;
```

2c

Use the `foldTree` function from (**2a**) to implement the `average` function, which has the type `int tree→int` and returns the average of the values that are stored in the tree. This should be accomplished as follows:

- Use `foldTree` with suitable functions as arguments in order to compute the *sum* of the values stored in the trees.
- Use `foldTree` with suitable functions as arguments in order to compute the *number of Value-objects in the tree*.
- Perform integer division with the pre-defined function `div :: int → int → int` on these values to obtain the result.

Here your function is required to work correctly only on those trees that contain the constructor `Value` at least once.

Answer:

```
fun average t = (foldTree (op +) (op +) 0 t)
              div
               (foldTree (fn (x,y) => y+1) (op +) 0 t )
```

2d

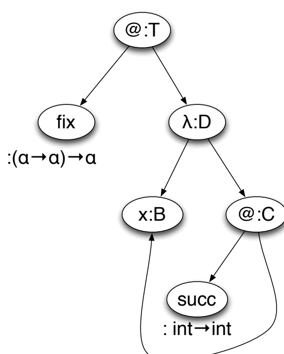
Draw the type inference parse tree of the following expression and describe the type inference steps of the expression according to this tree. Give the type of the expression, or in the case of a type error, explain how you detect this.

```
fix (λx. succ x)
```

Assume the following given types:

```
fix : :(α → α) → α
succ : int → int
```

Answer:



Question 3. Prolog (weight 20%)

3a

Write a Prolog predicate `sameHead(Xs,Ys)` that is true if and only if `Xs` and `Ys` have the same (unifiable) first elements

E.g.

```
| ?- sameHead([1,2,3],[1,4,5]).

yes
| ?- sameHead([X,1],[2,3]).

X = 2

| ?- sameHead([1,2],[3,4]).

no
```

Answer:

```
sameHead([X|Xs],[X|Ys]).
```

3b

Write a Prolog predicate `zip(Xs,Ys,Zs)` that is true if and only if `Xs` and `Ys` are lists of the same length, and `Zs` consists of the first element of `Xs`, followed by the first element of `Ys`, then the second of `Xs`, the second of `Ys`, etc.

E.g.

```
| ?- zip([1,2,3],[a,b,c],[1,a,2,b,3,c]).

yes

| ?- zip([1],[a,b],[1,a,b]).

no
```

Answer:

```
zip([],[],[]).
zip([X|Xs],[Y|Ys],[X,Y|Zs]) :- zip(Xs,Ys,Zs).
```

3c

Use `zip` to define a Prolog predicate `stutter(Xs)` that is true if and only if `Xs` is a list with an even number of elements, where the first element is equal to (i.e. is unified with) the second, the third is equal to (unified with) the fourth, etc. E.g.

```
| ?- stutter([a,a,b,b]).

yes

| ?- stutter([1,1,1,1,1,1]).

yes

| ?- stutter([1,a,a,1]).

no
```

Answer:

```
stutter(Xs) :- zip(Ys,Ys,Xs).
```

3d

What will the Prolog interpreter answer to the following queries:

```
| ?- zip([X],[Y],[a,b]).

| ?- zip([X],[Y],[Y,X]).

| ?- stutter([X,a]).

| ?- stutter([X,Y,Z,a,Z,Y,X,b]).
```

Answer:

Forgotten yes or question marks/semicolons in the responses don't matter, only the substitutions are important. And "no" or something like that for the last one.

| ?- zip([X],[Y],[a,b]).

```
X = a
Y = b
```

yes

| ?- zip([X],[Y],[Y,X]).

```
Y = X
```

yes

| ?- stutter([X,a]).

```
X = a ?
```

yes

| ?- stutter([X,Y,Z,a,Z,Y,X,b]).

no

3e

Briefly explain:

- what the effect of the cut predicate is
- why it is needed in Prolog
- why it is problematic

Answer:

20%, 5 for the effect, 5 for the why, 10 for the problem

- what the effect of the cut predicate is

It tells the interpreter to cut off branches of the search tree in two ways: no further ways of satisfying earlier literals in the same clause will be tried, and no further clauses will be tried to satisfy the head.

- why it is needed in Prolog

It is needed for efficiency. It is used to cut off parts of the search tree which the programmer knows will not give any new results.

- why it is problematic

Its effect is not related to the declarative semantics of Prolog programs. Its use prevents a declarative reading of the program. This makes programs with cut hard to write and hard to read. It is very difficult to find mistakes in programs that use the cut.