Exam for 2G1512 DatalogiII, Dec 17th 2001 08.00–13.00

Dept. of Microelectronics and Information Technology

December 14, 2001

Allowed materials

You are allowed to bring the course book and all handouts to the exam, except old (or example) exams and solutions to old (or example) exams. English to "your favourite language" dictionaries are also allowed. Self-made notes are also allowed as long as they do not consist of copies of old (or example) exams or solutions to old (or example) exams.

Laptops or other turing complete devices are not allowed.

Instructions

Write each one of your answers on a separate sheet of paper (it's OK to let a single answer span several pages). Use only one side of the sheet. Write your full name and "personnummer" on each of the sheets. Number each sheet. Before you hand in your answers, make sure that your solutions are sorted in ascending order.

Please write "Bonuspoäng: X" on front of the envelope, where X is the number of bonus points you think you have earned.

Write your answers in English, or in Swedish.(We prefer English.)

The questions are not arranged according to difficulty, questions we consider hard are marked with ♦.

Grading: The preliminary limits are as follows: If the sum of exam and bonus points are: < 25 fail (U), ≥ 25 grade 3, ≥ 36 grade 4 and ≥ 44 grade 5.

Good luck!

Declarative Computation Model

Question 1 (4 points):

For each of the following code fragments (A, B, C and D), select one of the **possible outputs** (a–k) as described below. Each correct answer gives +1 point, unanswered 0 and erroneous answers -1. You cannot get less than zero points.

Code fragment A:

```
declare
A B R
A = fun{$} {Show A} a end
B = fun{$} {Show B} b end
R = if 1 > 2 then {A} else {B} end
{Show R}
```

Code fragment B:

```
declare
A B R MyIf
A = fun{$} {Show a} a end
B = fun{$} {Show b} b end
MyIf = fun{$ Cond TrueClause ElseClause}
          if Cond then
             TrueClause
          else
             ElseClause
          end
       end
R = {MyIf 1 > 2 A B}
{Show R}
```

Code fragment C:

```
declare
A B R
A = fun{$} {Show a} A end
B = fun{$} {Show b} B end
R = {if 1 > 2 then A else B end}
{Show R}
```

Code fragment D:

```
declare
A B R
A = fun{$}
       A = fun {$}
              B
           end
    in
       {Show {A}}
       a
    end
B = fun{$} {Show B} b end
R = if 1 > 2 then {A} else {B} end
{Show R}
```

Possible output:

a:	<P/1 A>
b:	<P/1 B>
c:	<P/1 A> b
d:	<P/1 B> a
e:	<P/1 B> b
f:	<P/1 B> a
g:	b <P/1 A>
h:	a <P/1 B>
i:	b <P/1 B>
j:	a <P/1 B>
k:	None of the above

Question 2 (2 points):

Given the following code fragment, what will the call to `{Show X.2.2.1}` print?

```
local X Y in
   X = 1|2|Y
   X = Y
   {Show X.2.2.1}
end
```

Question 3 (3 points):

Translate the following code fragment into the kernel language syntax.

```
local
   R
   fun{A B}
      case B of
         1 then 11
      [] 2 then 22
      else
         3
      end
   end
in
   R = {A 1 + 2}
end
```

Question 4 (2 points):

Below is a function written in pseudo-code. As arguments it takes two positive integers: X, Y. It iterates through all the integers in the interval (X, Y) and calculates a result. `:=` is the update (assignment) operation on the mutable variable `Res`.

Rewrite the function in the declarative (functional) model of Mozart in such a way that the computation is iterative. Hint: The function should be *tail recursive*.

```
int HellsKitchen(int X, int Y){
    int Res;
    for I from X to Y do:
        If I modulo 7 == 0 then Res := Res-I;
        If I modulo 13 == 0 then Res := Res div I;
        If I modulo 17 == 0 then Res := Res * I;
        Otherwise then Res := Res + I;
    end
```

```
    return Res;
}
```

Declarative Programming Techniques

Question 5 (2 points):

Consider the problem of computing the exponential of a given number. We would like to have a procedure that takes as its arguments a base `B` and a positive integer exponent `N`, and it computes B^N. The exponential can be defined as follows:

$$B^N = \begin{cases} 1 & \text{if } N = 0 \\ B \times B^{N-1} & \text{if } n > 0 \end{cases}$$

Your assigment is to write the recursive function `{RecExp B N}` corresponding to this definition. You should only use the declarative model.

Example

```
declare
Result = {RecExp 2 10}
{Browse Result} -> 1024
```

Question 6 (2 points):

In general, the Fibonacci numbers can be defined by the rule

$$Fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{otherwise} \end{cases}$$

We can translate this definition into a recursive function for computing Fibonacci numbers:

```
fun {RecFib  N}
   case N of 0 then 0
   [] 1 then 1
   else {RecFib N-1} + {RecFib N-2}
   end
end
```

After some testing we have seen that this is very inefficient way to compute Fibonacci numbers. We want you to write an iterative version of the Fibonacci function. The function should accept one integer argument and be named `IterFib`.

Question 7 (6 points) ♦:

A grammar for a small language describing the contents and structure of lists can be defined as follows:

```
<pattern> ::= ’[’ <list of patterns> ’]’ | <element>

<list of patterns> ::= <pattern> <list of patterns> | <pattern>

<element> ::= ’a’ | ’b’ | ’c’ | ’d’ | ’*’
```

Patterns in this language can be seen as a template defining lists containing the atoms `a`, `b`, `c`, `d` or sublists of the same construction.

The special atom `’*’` does not specify itself, it is a wildcard matching one of the atoms `a`, `b`, `c`, `d` or any pattern (note that this allows any list i.e. its elements are not limited to `<element>`).

Example 1

The pattern `[a ’*’ a]` matches all three-element lists starting with `a` and ending with `a`.

Example 2

The pattern `[a [c ’*’] a]` matches all three-element lists starting with `a` and ending with `a`, having as its middle element a sublist of two elements the first of which is `c`.

Write a function, `{GenerateMatcher APattern}`, that takes one argument, a pattern. As its result it returns a function taking one argument, a list to check against APattern. The returned function should return true if the list given as its argument matches the pattern supplied to generate matcher, otherwise it should return false.

Examples

```
A = {GenerateMatcher [a ’*’ a]}
```

```
{A [a b a]} -> true
{A [a b d]} -> false
{A [a [a b] a]} -> true

B = {GenerateMatcher [a [c '*'] a]}
{B [a [b c] a]} -> false
{B [a [c b] a]} -> true
```

Declarative Concurrency

Question 8 (4 points):

In this assigment you have to write one function, `Producer`, and one procedure, `Consumer`.

The Producer creates a stream of atoms which the Consumer will process. The producer is supposed to be demand driven, it means that it will not add new atoms to the stream unless the consumer needs them. To acomplish that you have to use *lazy* evaluation.

The function, `{Producer N}`, takes one argument (an integer, `N`) and produces a stream which elements are choosen from the set of atoms {'tick', 'tack', 'minute'}. These three atoms are generated as follows:

The Producer starts with `N` equal to zero. If `N` is even it produces the atom 'tic'. If `N` is odd it produces the atom 'tac'. In both cases `N` is updated to `N` + 1 and the production continues. If `N` is equal to 59 then the atom 'minute' is produced, `N` is set to zero and the production continues.

The procedure Consumer takes one argument, the stream the producer produces, and does the following:

The consumer first calls `{Delay 1000}`. This will force consumer to suspend for one second. When the consumer is woken up it will try to fetch an atom from the stream and print it using `{Show ...}`. Then it repeats the process by calling `{Delay 1000}` and so forth.

Your assignment is to write the function `Producer` and `Consumer`.

Example

```
declare
Str={Producer 0}
{Consumer Str}
```

The example above will produce the following output in the Emulator window:

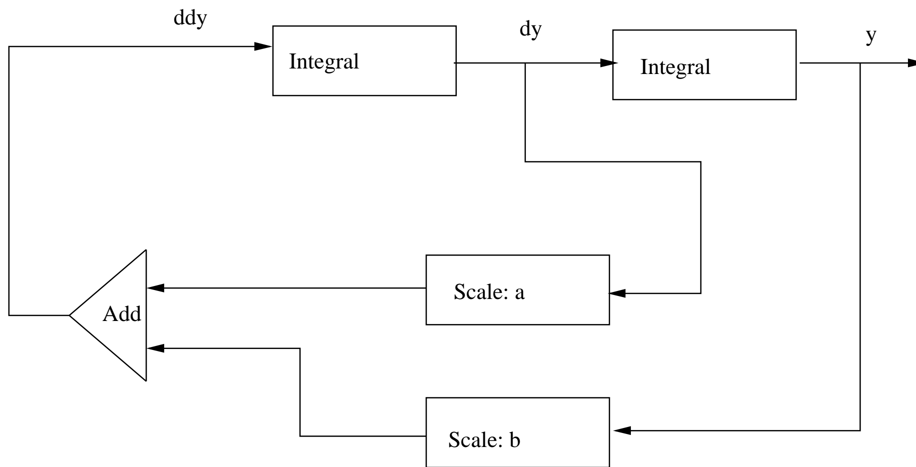
```
tic
tac
tic
tac
tic
...
...
...
minute
tic
tac
...
...
```

Question 9 (4 points):

Consider the problem of designing a signal-processing system to study the homogeneous second-order linear differential equation:

$$\frac{d^2y}{dt^2} - a\frac{dy}{dt} - by = 0$$

$y(t)$ can be calculated using streams according to the following data-flow diagram:



Write a function `{Second A B Dt Y0 Dy0}` that takes as arguments the constants a, b, dt and the initial values y_0 and dy_0 for y and dy/dt and generates the stream of successive values of y according to the data-flow diagram above.

You can assume that you have access to the function `{Integral Integrand InitialValue Dt}` that implements the Integral function-block.

Hint: Start by implementing the function-blocks Add and Scale as separate functions. You may use either eager concurrent execution or lazy execution.

Encapsulated State

Question 10 (6 points):

The function `fun{Q A B} ... end` iteratively calculates the sum $\sum_{i=A}^{B} i$. You can assume that initially: $A > 0$, $B > 0$, $B > A$.

a (3 points)

Implement `Q` in such a way that it only uses *implicit state* to calculate the sum. (An analytical solution is not allowed)

b (3 points)

Implement `Q` in such a way that it only uses *explicit state* to calculate the sum. An analytical solution is not allowed, the state has to be encapsulated.

Question 11 (3 points):

At Itsey Bitsey Machine Corporation they use the following system to keep track of their employees' salaries:

```
fun{MakeEmployee Name Amount}
% This function is only known to Eben Scrooge
   salary(name: Name amount: {NewCell Amount})
end

proc{ChangeSalary Salary NewAmount}
% This procedure is only known to Eben Scrooge
   {Assign Salary.amount NewAmount}
end

fun{CheckSalary Salary}
% A reference to this function is given to employees
```

```
   {Access Salary.amount}
end
```

When an employee is hired the head of the accounting department, Eben Scrooge, creates a new employee instance (by calling `MakeEmployee`) which they store in their files. A reference to this instance as well as `CheckSalary` is given to the employee as he may want to check his salary in the future. This is deemed safe as only the accounting department has access to `ChangeSalary` and `MakeEmployee`.

Lately, Olivier Warbucks, the managing director of Itsey Bitsey Machine Corporation has started to increase his salary by manipulating the record representing the salary information directly. This has come to Eben Scrooge's attention and now he want's IBMC's computer programmer, Alyssa P Hacker, to prevent such changes in the future. Alyssa analyzes the problem and decides that she needs some way to prevent an employee from changing his salary even though he has seen the source code to `MakeEmployee` or `CheckSalary`.

Help Alyssa by modifying `MakeEmployee`, `CheckSalary` and `ChangeSalary`.

Question 12 (2 points):

Consider the following code fragment. Answer the questions (a–d) in the comments on the lines starting with `{Show ...`. Each question gives 0.5 points.

```
declare
A={NewCell 0}
B={NewCell 0}
T1={Access A}
T2={Access B}
{Show A==B}       % a: What will be printed here,
                  % true, false, A, B or 0?
{Show T1==T2}     % b: What will be printed here,
                  % true, false, A, B or 0?
{Show T1=T2}      % c: What will be printed here,
                  % true, false, A, B or 0?
{Assign A {Access B}}
{Show A==B}       % d: What will be printed here,
                  % true, false, A, B or 0?
```

Object-Oriented Programming

Question 13 (2 points):

a (1 point)

We have defined four classes `Person`, `Student`, `Employee`, and `PhDStudent`:

```
declare
class Person
   attr
      name
      lastName
      workingStatus

   meth init(N LN)
      name<-N
      lastname<-LN
      workingStatus<-unkown
   end
end
class Student from Person
   meth init(N M)
      Person,init(N M)
      workingStatus<-student
   end
end
class Employee from Person
   meth init(N M)
      Person,init(N M)
      workingStatus<-engineer
   end
end
class PhDStudent from Student Employee
   meth init(N M)
      Student,init(N M)
      Employee,init(N M)
   end
end
```

As you can see class `PhDStudent` has two super-classes `Student`, and `Employee`. Is this use of multiple inheritance correct? Motivate your answer.

b (1 point)

Given four classes `A`, `B`, `C`, and `D`:

```
declare A B C D
class A end
class B from A D end
class C from A end
class D from C end
```

Is this use of inheritance correct? Motivate your answer.

Question 14 (4 points):

Higher order Object Oriented Programming

Write a class HOP with the method `for(A M)`, where A is an array of objects, and M is message. Calling the method `for(A M)` invokes the method M in all objects in array A.

Example of usage, where `{MakeArray L H F}` creates an array from lower bound L to higher bound H where each element A.I is `{F I}`.

```
declare
class C
   attr i
   meth init(I) i <- I end
   meth doit {Show @i} end
end
A1 = {MakeArray 1 10
      fun{$ I} {New C init(I)} end}

Hop = {New HOP init}
{Hop for(A1 doit)}

The result is
1
2
:
:
10
```

Question 15 (4 points):

Programming higher order patterns in Object-oriented languages

Assume that our programming language does not support higher-order functions. It is still possible to code higher order techniques in our Object-oriented language. Use the object-oriented support in Mozart to program the following higher order example

```
declare
fun {Filter Xs F}
    case Xs
    of nil then nil
    [] X|Xr then
       if {F X} then X|{Filter Xr F}
       else {Filter Xr F}
       end
    end
end

% Odd is a function which tests whether an integer is Odd or not
{Show {Filter [1 2 3 6 9] Odd}}
```

This means that you should define a class `ListC` with the method `filter(Xs O ?Ys)` that corresponds to `Filter Xs F ?Ys`, where `O` is an object with an `apply(X ?Y)` method. Here is an example:

```
declare R
List = {New ListC init}
class OddC from BaseObject
   meth apply(X ?Y) Y = {Odd X} end
end
Odd = {New OddC init}
{List filter([1 2 3 6 9] Odd R)}
{Show R}
```