Exam for 2G1512 DatalogiII, Aug 16th 2002 14.00–19.00

Dept. of Microelectronics and Information Technology

December 2, 2002

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.

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–n) 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 Head Tail

Head = a
Tail = b|c|nil
A = Head|Tail
B = [a b c]
R = A==B
{Show R}
```

Code fragment B:

```
declare A B R Not

A = monday
B = tuesday
fun {Not Boolean}
   if Boolean then false
   else true end
end

if A==B then R=true
elseif {IsNumber A} andthen {Not {IsNumber B}} then R=false
else skip
end
{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:	true
l:	false
m:	R<optimized>
n:	None of the above

Question 2 (2 points):

Given the following code fragment, what will the call to `{Show Y.1.2.1}` print?

```
local X Y Z in
   X = a(b X)
   Y = c(X Z)
   Z = d(e f g h)
   {Show Y.1.2.1}
end
```

Question 3 (3 points):

Translate the following code fragment into the kernel language syntax.

```
local
   X1
   proc {X2 X3 X4 X5}
      if X3==2 andthen X4==b then
         X5=d
      else
         X5=t
      end
   end
in
   {X2 1+1 b X1}
   {Show X1}
end
```

Question 4 (2 points):

The function `{OddProduct L}` is given below. It takes one argument `L`, which is a list of positive integers. The function multiplies all odd integers in `L` and returns result. A cell `C` is used to save partial results. When the end of `L` is encountered, the cell's content is returned.

Your assignment is to write a **tail recursive** (i.e. *last call optimization*) function `{OddProductDec L}`. This function should perform the same operation as the function `{OddProduct L}`, (i.e. multiplies all odd integers in a given list and returns the result). You must use the **declarative** programming model.

```
declare

fun {OddProduct L}
   C = {NewCell 1}
   fun {Prod L}
      case L of nil then {Access C}
      else
         if {IsOdd L.1} then
            {Assign C {Access C}*L.1}
         end
         {Prod L.2}
      end
```

```
   end
in
   {Prod L}
end
```

Example

```
{Show {OddProduct [1 2 3 4 5]}} -> 15
{Show {OddProductDec [1 2 3 4 5]}} -> 15
```

Declarative Programming Techniques

Question 5 (2 points):

In this assignment you have to write the recursive function `{ApplyToEach L F}`. The function takes two arguments: a list `L` and a function `F`, and returns the new list created by applying the function `F` to every element of the list `L`. You must use the declarative programming model.

Example

```
{Show {ApplyToEach [1 2 3 4] MyltiplyByFive}} -> [5 10 15 20]
```

Where MultiplyByFive, for example, would be as follows:

```
fun{MultiplyByFive X}
   X * 5
end
```

Question 6 (2 points):

The following function Foo, is defined as follows:

$$Foo(x) = \begin{cases} "a" & \text{if } x = 0 \\ "b" & \text{if } x = 1 \\ Append(Foo(x-1),\ Foo(x-2)) & \text{otherwise} \end{cases}$$

For example, $Foo(2) = "ba"$, $Foo(3) = "bab"$ and $Foo(4) = "babba"$

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

```
fun {RecFoo  X}
   case X of 0 then "a"
   [] 1 then "b"
   else {Append {RecFoo X-1} {RecFoo X-2}}
   end
end
```

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

Question 7 (6 points):

Write the function `{GenBaseConverter InBase OutBase}` which returns a function which takes a string as its only argument. The returned function should interpret the string as a number in base `InBase` and return a string representing the same number in base `OutBase`. You can study some run-time examples below:

```
{{GenBaseConverter 16 8} "F"} -> "17"
{{GenBaseConverter 10 8} "4711"} -> "11147"
{{GenBaseConverter 10 16} "17"} -> "11"
```

You can assume that `Inbase` and `Outbase` ≤ 16. To help you we have provided the following functions:

```
%%
%% Convert a string to a list of values. The string should
%% represent a number in a base less than or equal to 16. Each
%% character is converted to its numeric value. i.e.
%% {StringToList "14AF"} -> [1 4 10 15]
%%
fun{StringToList Str}
   {List.map Str fun{$ V}
    if V >= ("0".1) andthen V =< ("9".1) then
       V - ("0".1)
    elseif V >= ("A".1) andthen V =< ("F".1) then
       V - ("A".1) + 10
    end
 end}
end

%%
%% Convert a list of values to a string. The list should
%% represent a number in a base less than or equal to 16.
%% Each value in the list is converted to its ascii value.
%% i.e. {ListToString [1 4 10 15]} -> "14AF"
%%
fun{ListToString Lst}
   {List.map Lst fun{$ V}
    if V >= 0 andthen V =< 9 then
       V + ("0".1)
    elseif V >= 10 andthen V =< 15 then
       V - 10 + ("A".1)
    end
 end}
end
```

Declarative Concurrency

Question 8 (4 points):

Part I (2 points)

In this assignment you have to write two procedures, `{Producer File Word}`, and `{Consumer ReadStream}`.

The `Producer` creates a stream of strings which the `Consumer` will process. The producer is supposed to be demand- driven (Remember that demand-driven means that it will not add new strings to the stream unless the consumer needs them). To accomplish this you **are not allowed** to use `lazy` notation, i.e. you must explicitly express demand-driven execution.

The procedure `{Producer File Word}` takes two arguments. The first argument `File` is a text file (i.e. a list of strings) that the `Producer` updates to point to the next string in the file each time the `Consumer` asks for a new word. The second argument is an unbound variable that the `Producer` and the `Consumer` use for synchronization. The `Producer` waits for the `Consumer` to bind the variable `Stream` to a pair `X|Xs`, where `X` and `Xs` are unbound variables as well. Then the `Producer` binds the `X` to `File.1`, and recursively calls itself with `File.2` and `Xs` as arguments.

The procedure `{Consumer ReadStream}` takes an unbound variable as an argument. The same variable is used as argument when calling the `Producer` for the first time. The `Consumer` creates two new variables (e.g. `X1` and `X2`) and binds the variable `Stream` to the pair `X1|X2`. The variable `X1` has to be bound to an integer by the `Producer`. Then the `Consumer` calls the procedure `Show` to display the variable `X1`. After that it calls the procedure `Delay` with an argument of `1000` and then recursively calls itself with the `X2` as the argument.

Hint

You can assume that the reading file is infinitely long, (i.e. it will never end).

Example

```
declare
Word
thread {Producer MozartDoc Word} end
{Consumer Word}
```

Part II (2 points)

In this assignment you have to write one function, `{Producer Book}`, and one procedure, `{Consumer StreamOfWords}`.

The `Producer` function creates a stream of strings which the `Consumer` processes. The `Producer` is supposed to be demand- driven, (i.e. it will not add new string to the stream unless the consumer needs it). To accomplish that, you **must** use **lazy** notation.

The function, `{Producer Book}`, takes one argument (a file, `Book`) and produces a stream of strings (i.e. `0 1 2 3 4 5 ...`)

The procedure `{Consumer Stream}` takes a stream as an argument. The stream `Stream` is the stream that is produced by the `Producer`.

The procedure `Consumer` does the following:

- Fetches the string from the stream.
- Prints the string using the procedure `Show`.
- Delays execution for one second by calling `{Delay 1000}`.
- Repeat.

Example

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

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: $A > 0$, $B > 0$, $B > A$. *Be aware that both subquestions below are graded as a whole.*

a

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

b

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):

Write the output of each of the code fragments a-d
Each question gives 0.5 points.

```
%(a)-----------------
declare
A={NewCell 5}                  %What will be printed,
{Show A}                       % <cell>, 5, Binding Error?

%(b)-----------------
declare
A={NewCell 5}
{Assign A 1}                   %What will be printed,
{Show {Access A}}              % <cell>, 1, 5, Binding Error?

%(c)-----------------
declare
B={NewCell 0}
A=1                            %What will be printed here,
{Show A}                       % <cell>, 1, 0, Binding Error?

%(d)-----------------
declare
A={NewCell 0}
B=A
{Assign A 5}                  %What will be printed here,
{Show {Access B}}             % <cell>, 0, 5, Binding Error?
```

Object-Oriented Programming

Question 13 (10 points):

Our goal is to do symbolic programming with objects. That is to say our object oriented programming system does not support symbolic data structures such as lists and pattern matching on them. For this problem consider the following task: Given the following function for removing an element from a list in Oz, with the results shown below, we would like to write the same function, but using objects to represent lists.

```
fun {Delete El Ls}
   case Ls
   of nil then nil
   [] !El|Xs then
      {Delete El Xs}
   [] X|Xs then
      X|{Delete El Xs}
   end
end


{Browse {Delete 2 [1 2 3]}} %%% shows [1 3]
```

To do this in an object-oriented system we are going to take a number of steps:

Step 1 We have to define three classes, one to represent the general class of a list, the second represents the nil (empty) list and the third represents the cons cell (i.e. the non-empty list cell).

```
declare
class List
       meth init skip end
     ...
end

class NilClass from List
       meth isNil(T) T = true end
end

class ConsClass from List
```

```
        attr element next
        ...
end
```

Step 2 We have to convert the above definition of the Append function to one that does not use pattern matching, nil, nor the cons operator ?|?. To do this we need to define the following functions that work on List objects:

```
% Nil creates a NilClass object
fun {Nil} ... end

% Cons takes an element X of any type and a List object,
% and creates a ConsClass object
fun {Cons X Xr} ... end

% Hd (i.e. head) returns the first element of the List object
% Xs; it assumes Xs is a ConsClass object
fun {Hd Xs} ... end

% Tl (i.e. tail) returns the remaining list of the List object
% Xs; it assumes Xs is a ConsClass object
fun {Tl Xs} ... end

% IsNil is a Boolean function that returns true if its argument
% is a NilClass object; otherwise false
fun {IsNil Xs} ... end
```

Step 3 After writing the above classes and the functions including delete, we can at last do list manipulation, for example the following:

```
declare
R = {Delete
        2
        {Cons 1 {Cons 2 {Cons 3 {Nil}}}}}
```

However we would also like to be able to print the result in the usual fashion using Show or Browse. Therefore we need to add a new method to the List class to convert lists as objects to list as symbolic structures. The new method is called toList(?Xs). It returns the symbolic list in the variable Xs.

```
class List ...
...
meth toList(?Xs) ... end
end
```

Now we can print the list!

```
declare Ls
{R toList(Ls)} {Browse Ls}
```

Question 13 a (2 points):

We need to define the method `isNil(?T)` to check whether an object is of class `NilClass` or not. This method is then used to define the function IsNil as follows:

```
fun {IsNil Xs} T in {Xs isNil(T)} T end
```

When implementing the method isNil(?T) one has to consider two different design alternatives:

1 The first design alternative is to define the method in List to return with some default, say always false, and then override it in the class NilClass to return true.

2 The second design alternative is to define it only in the classes NilClass (to return true) and ConsClass (to return false).

Which is the better design alternative? Please motivate!

Question 13 b (2 points):

Define the Delete function in terms of the auxiliary functions defined in step 2 above.

Question 13 c (4 points):

To implement the functions defined in step 2, you need to complete the definition of the three classes: List, NilClass and ConsClass. Here you are asked to complete the definition of the class ConsClass, described with its methods below:

```
class ConsClass from List
   attr element next

   meth isNil(?T) ... end          % returns T = false

   meth hd(?X) ... end             % returns X as the
                                  % element of the cons object

   meth tl(?Xr) ... end             % returns Xr as the
                                  % remaining list

   meth cons(X Xr) ...   end       % fills the fields of the
                                  % cons object
end
```

You should also complete the definitions of the functions Nil, Cons, Hd, Tl and IsNil.

Question 13 d (2 points):

Augment the class List with a definition of the method toList as described in step 3 above.

```
class List ...
...
meth toList(?Xs) ? end
end
```