Exam for 2G1512 DatalogiII, Apr 12th 2002 09.00–14.00

Dept. of Microelectronics and Information Technology

April 11, 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–d) 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 Not
fun {Not Boolean}
   if Boolean then false
   else true end
end

A = fun{$} true end
B = fun{$ Boolean} {Not Boolean} end
R = if {B false} then {A} else {B {A}} end
{Show R}
```

Code fragment B:

```
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 C:

```
local R Y in
   Y = 1
   local X1 Y L in
      L = [X1 Y]
      L = [1 2]
      R = Y==1
   end
   {Show R}
end
```

Code fragment D:

```
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}
```

Possible output:

a:	true
b:	false
c:	R<optimized>
d:	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 = 1|X
   Y = X|Z
   Z = 2|3|4|nil
   {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==2 then X5=4
      else X5=0
      end
   end
in
   {X2 1+1 2 X1}
   {Show X1}
end
```

Question 4 (2 points):

The function `{SumList L}` is given below. It takes one argument, `L`, which is a list of positive integers. The function sums all integers in `L` and returns its sum. As you can see, one 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 `{SumListDec L}`. This function will perform the same operation as the function `{SumList L}`, (i.e. sum all integers in a given list and return the sum), but you must use the **declarative** programming model.

```
declare

fun {SumList L}
   C = {NewCell 0}
   fun {Sum L}
      case L of nil then {Access C}
      else
         {Assign C {Access C}+L.1}
         {Sum L.2}
      end
   end
in
   {Sum L}
end
```

Example

```
{Show {SumList [1 2 3]}} -> 6
{Show {SumListDec [1 2 3]}} -> 6
```

Declarative Programming Techniques

Question 5 (2 points):

In this assignment you have to write the recursive function `{DeleteNth L N}`. The function takes two arguments: a list `L` and an integer `N`, and it returns the new list created by removing the *nth* element from the list `L`. You must use the declarative programming model. You can assume that the index `N` is smaller than or equal to the length of the list.

Example

```
 {Show {DeleteNth [1 2 3 4] 2}} -> [1 3 4]
```

Question 6 (2 points):

The following function `ProdList` takes as argument a list of integers and returns the product of the integers in the list. e.g. `{ProdList [1 2 3]}` returns 6.

Here is an inefficient implementation of `ProdList` because it is written in a recursive style.

```
declare
fun {ProdList L}
   case L of nil then 1
   [] H|T then
      H * {ProdList T}
   end
end
```

Write an iterative version of `ProdList`. **You must use the declarative programming model**.

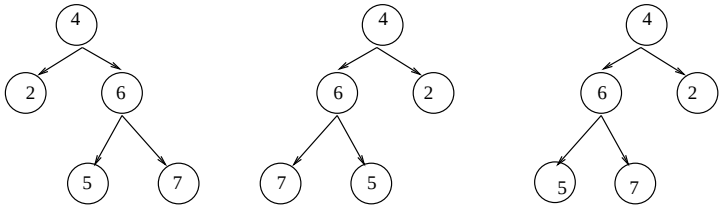
Question 7 (6 points):

Consider the following piece of code that defines an ADT (Abstract Data Type) for binary trees in Mozart:

```
fun{MakeTree Info LeftTree RightTree}
   tree(info: Info left: LeftTree right: RightTree)
end

fun{GetInfo ATree}
   ATree.info
end

fun{GetLeftTree ATree}
   ATree.left
end

fun{GetRightTree ATree}
   ATree.right
end

TheEmptyTree = nil

fun{IsEmptyTree ATree}
```

```
   ATree == TheEmptyTree
end
```

In an unordered tree the order of a node's children is unimportant, i.e. if the left and right subtrees are interchanged in zero or more of the nodes the resulting tree is equivalent to the original tree. An example of three equivalent trees are shown below:



Write a predicate `{TreeEqv TreeA TreeB}` that returns true if the trees `TreeA` and `TreeB` are equivalent according to the definition above. You can assume that only integers are stored in the `info` field of the nodes and that all nodes have a unique value. Informally describe how your solution works before writing the code.

Declarative Concurrency

Question 8 (4 points):

Part I (2 points)

In this assignment you have to write two procedures, `{Producer Val Stream}`, and `{Consumer Stream}`.

The `Producer` creates a stream of numbers which the `Consumer` will process. The producer is supposed to be demand driven (Remember that demand driven means that it will not add new integers 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 Val Stream}` takes two arguments. The first argument `Val` is an integer that the `Producer` updates to `Val+1` each time the `Consumer` asks for a new integer. 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 `Val`, and recursively calls its self with `Val+1` and `Xs` as arguments.

The procedure `{Consumer Stream}` 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.

Example

```
declare
Str
thread {Producer 0 Str} end
{Consumer Str}
```

Part II (2 points)

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

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

The function, `{Producer N}`, takes one argument (an integer, `N`) and produces a stream of numbers (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:

The `Consumer` fetches the first integer from the stream and prints it using the procedure `Show`. Then it calls `{Delay 1000}` (i.e. goes to sleep for one second). When the `Consumer` wakes up it repeats the process.

Example

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

Question 9 (4 points):

Part I (2 points)

The procedure `{PassingTheToken Id Tin Tout}`, given below, takes three arguments: `Id` is an integer (identification number), `Tin` and `Tout` are dataflow variables. The procedure `PassingTheToken` (see the code below) waits for the variable `Tin` to be bound to a pair `H|T`. Then it prints `Id#H`, suspends for a second, binds the variable `Tout` to the pair `H|X` (where `X` is the newly created local variable), and recursively calls itself (i.e. `{PassingTheToken Id T X}`).

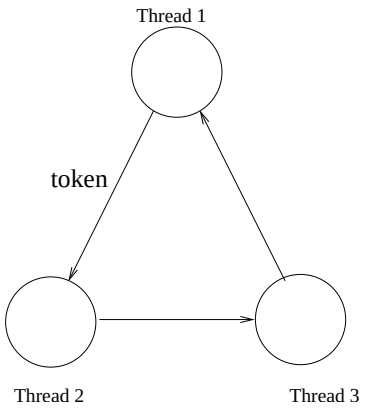
Function definition

```
declare
proc {PassingTheToken Id Tin Tout}
   case Tin of
      H|T then X in
      {Show Id#H}
      {Delay 1000}
      Tout=H|X
      {PassingToken Id T X}
   [] nil then
      skip
   end
end
```

Your assignment is to create three threads (see figure below), where each of the threads calls the `PassingTheToken` procedure. The first thread will pass the token to the second thread, the second thread will pass the token to the the third, and finally the third thread will pass the token back to the first thread, and so on...

Example

```
declare
T1
T2
T3
T=token

Your code goes here...
```



Hint 1: Variables `T1`, `T2`, `T3` should be used to connect the threads in a circle formation.

Hint 2: You must give the token to one of the threads at the start to make the system run.

Part II (2 points)

In the code given below, you have one cell `C`, one lock `L`, one function `{Read C}`, and two procedures: `{ReadAndUpdate Id C}`, and `{Update C NewVal}`.

```
declare
Counter = {NewCell 0}
L = {NewLock}
proc {Update C NewVal}
   {Delay 1000}
   {Assign C NewVal}
end

fun {Read C}
   {Delay 1000}
   {Access C}
end

proc {ReadAndUpdate Id C}
   Val NewVal
in
   Val = {Read C}
```

```
   {Show Id#' read the C value and the value is = '#Val}
   NewVal = Val + Id
   {Delay  {OS.rand} mod 1000}
   {Update C NewVal}
   {Show Id#' updated the C value and the new value is = '#{Read C}}
   {ReadAndUpdate Id C}
end

thread {ReadAndUpdate 10 Counter} end
thread {ReadAndUpdate 1 Counter} end
thread {ReadAndUpdate 100 Counter} end
```

Three threads are started and all three are evaluating the procedure `ReadAndUpdate`. The procedure `ReadAndUpdate` repeatedly does the following:

- It calls the function `Read` (i.e. reads the content of the cell `C`)
- Prints out a message
- Binds a variable `NewVal` to `Val+Id`
- Suspends for a random number of milliseconds (less than 1000)
- Calls the procedure `{Update C NewVal}` (i.e. updates the content of the cell `C` with `NewVal`)
- Prints out a message

We want each read operation (i.e. a call to the `Read` function) to be followed by a write operation (i.e. a call to `Update`) of the same thread.

This can be accomplished with the use of locks. Your assignment is to secure the critical section with the use of the lock `L`. You are only allowed to add one `lock L then ... end` statement to the code. Explain what a critical section is.

Encapsulated State

Question 10 (6 points):

The function `fun{Q A B} ... end` iteratively calculates the product:

$$A \times (A-1) \times (A-2) \times \ldots \times (B+1) \times B$$

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

10 a

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

10 b

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

Question 11 (3 points):

Consider the following piece of code that defines an ADT (Abstract Data Type) for binary trees in Mozart:

```
fun{MakeTree Info LeftTree RightTree}
   tree(info: Info left: LeftTree right: RightTree)
end

fun{GetInfo ATree}
   ATree.info
end

fun{GetLeftTree ATree}
   ATree.left
end

fun{GetRightTree ATree}
   ATree.right
end

TheEmptyTree = nil

fun{IsEmptyTree ATree}
   ATree == TheEmptyTree
end
```

The ADT is correct but it is not a secure ADT, i.e. it is possible to break the abstraction barrier by directly accessing a tree created by `MakeTree`.

Rewrite the ADT to make it into a secure ADT. Your rewritten ADT should have the same interface as the original ADT, i.e. it should be possible to replace

the original ADT with yours if the program using the ADT does not break the abstraction barrier.

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 0}
B
A=1                              %What will be printed,
{Show A}                         % <cell>, 1, Binding Error?

%(b)-----------------
declare
A={NewCell 0}
B
{Assign A 1}                    %What will be printed,
{Assign A 2}                    % <cell>, 1, 2, Binding Error?
{Show {Access A}}

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

%(d)-----------------
declare
A={NewCell 0}
B
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 appending lists in Oz, with the results shown below, we would like to write the same function, but using objects to represent lists.

```
fun {Append Xs Ys}
   case Xs
   of nil then Ys
   [] X|Xr then
      X|{Append Xr Ys}
   end
end

{Browse {Append [1 2 3] [4 5]}} %%% shows [1 2 3 4 5]
```

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 append, we can at last do list manipulation, for example the following:

```
declare
R = {Append
        {Cons 1 {Cons 2 {Cons 3 {Nil}}}}
        {Cons 4 {Cons 5 {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 Append 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
```