Exam Datalogi II (2G1512)
2002-12-14, 09:00-14:00

Firstname: ______________________________

Lastname: ______________________________

Personnummer: ______________________________

Rules

You are not allowed to bring any material or equipment (such as laptops, PDAs, or mobile phones) with you. The only exceptions are an English to "your favorite language" dictionary and pencils.

Instructions

- The exam has 300 points and takes 300 minutes. The points for each task should help you to judge how much time you use for each task.
- Please read the entire exam first!
- Write on these sheets of paper. Use the free space after each assignment for your answer.
- Write your name and "personnummer" on each page of the exam.

- If you need more space for your answers, use the additional sheets you have been provided with.

  Answers on additional sheets will only be considered if the sheets are marked with your name and "personnummer" and if you have noted in the space left after the question that part of your answer is on an additional sheet.

- You have to hand in the *complete* exam, you are not allowed to take home part of it (this also refers to the extra sheets).

- Write your answers in English or Swedish.

- Tables you might need are at the end of the exam.

Grading

The grades depend on the sum of exam and bonus points n:

$$\begin{array}{rcll} & n & < 150 & \text{fail (U)} \\ 150 \leq & n & < 200 & \text{grade 3} \\ 200 \leq & n & < 250 & \text{grade 4} \\ 250 \leq & n & & \text{grade 5} \end{array}$$

Points

Please do not write here, this is for correcting the exam.

Task	1	2	3	4	5	6	7	8	9	Σ
Max	45	25	20	40	45	20	40	40	25	300
Points										

Bonuspoints:

Totalpoints:

Grade:

Name:___________________________________ Personnummer:_______________

1 Questions (45 points)

1. Are environments shared among all statements of a single thread (3 points)?

   Solution. No. Threads (semantic stacks) consist of semantic statements being pairs of statements and environments.

2. Is the **case**-statement suspendable (3 points)?

   Solution. Yes.

3. Consider the environments

$$\begin{aligned} E_1 &= \{\mathsf{X} \mapsto x_1, \mathsf{Z} \mapsto x_2\} \\ E_2 &= \{\mathsf{X} \mapsto x_3, \mathsf{Y} \mapsto x_4, \mathsf{Z} \mapsto x_5\} \end{aligned}$$

   Give the environment $E_1 + E_2$ (3 points):

   Solution.

$$\{\mathsf{X} \mapsto x_3, \mathsf{Y} \mapsto x_4, \mathsf{Z} \mapsto x_5\}$$

   Give the environment $E_2 + E_1$ (3 points):

   Solution.

$$\{\mathsf{X} \mapsto x_1, \mathsf{Y} \mapsto x_4, \mathsf{Z} \mapsto x_2\}$$

4. Give the activation condition of the semantic statement

$$(\{\langle x\rangle\ \ \langle y\rangle_1\ \cdots\ \langle y\rangle_n\}, E)$$

   (3 points).

   Solution. $E(\langle x\rangle)$ is determined.

5. Do infinite computations (computations that never terminate such as an agent that receives an infinite number of messages) always require infinite stack space (3 points)?

Solution. No, due to tail recursion.

6. Assume a recursive procedure `P` (it is directly recursive, that is `P` calls `P`). Give one variable identifier that must be an external reference of the definition of `P` (3 points).

   Solution. `P` itself.

7. Has each semantic stack in the abstract machine a private store (3 points)?

   Solution. No. Semantic stacks (threads) actually communicate by using the single shared store.

8. Assume execution of the following statement (where ⟨*s*⟩ is also some statement):

```
local X in
   local Z in skip end
   local Y in
      local Y in
         ⟨s⟩
      end
   end
end
```

   Give the variable identifiers that occur in the environment E when the semantic statement (⟨*s*⟩ , E) is executed (3 points).

   Solution. `X`, `Y`.

9. Give the external references of the following procedure definition (3 points):

```
P = proc {$ X Y}
       local Z in {X {U Z} Y} end
    end
```

   Solution. `U`

10. Give the external references of the following procedure definition (3 points):

```
P = proc {$ X Y}
       local Y in {P {X Z} Y} end
    end
```

Solution. P, Z

11. Consider the following statement $\langle s \rangle$:

```
P = proc {$ X Y}
        if X==Y then Z=a else Z=P end
    end
```

Give the contextual environment created by execution of the semantic statement $(\langle s \rangle, \{Z \mapsto z, P \mapsto p, X \mapsto x\})$ (3 points).

Solution. $\{Z \mapsto z, P \mapsto p\}$

12. Give the value of U after execution of the following statement (3 points):

```
local X Y Z U in
   thread if X==1 then Z=2 else Y=3 end end
   thread if Z==3 then U=4 else U=5 end end
   X=2
end
```

Solution. U is unbound.

13. Give the value of Z after execution of the following statement (6 points):

```
local C X Y Z in
   fun {C A B}
      fun {$ F}
         {F A B}
      end
   end
   fun {X A B}
      A
   end
   fun {Y A B}
      B
   end
   Z={{{C {C a b} {C c d}} Y} X}
end
```

Solution. c

2 Accumulators (25 points)

The following function `{Longest Xs}` takes a list of lists `Xs` as input:

```
fun {Longest Xs}
   case Xs
   of nil  then 0
   [] X|Xr then
      N={Length X} M={Longest Xr}
   in
      if N>M then N else M end
   end
end
```

2.1 Examples (5 points)

What does `{Longest nil}` return?

Solution. `0`

What does `{Longest [[a] [b] [b c]]}` return?

Solution. `2`

2.2 Tail-Recursion (20 points)

Give an equivalent tail-recursive function `{LongestAcc Xs N}` that uses `N` as accumulator.

Solution.

```
fun {LongestAcc Xs N}
   case Xs
   of nil then N
   [] X|Xr then
      M={Length X}
      NM=if N>M then N else M end
   in
      {LongestAcc Xr NM}
   end
end
```

3 Higher-Order Programming (20 points)

Write a function `{SwitchMap Xs F G}` that takes a list `Xs` and two unary functions `F` and `G` as input. It returns a list where the elements at odd positions are obtained by applying `F` to the element at the same position in `Xs` and the elements at even positions are obtained by applying `G` to the element at the same position. For example, with the definitions

```
fun {Inc N} N+1 end
fun {Dec N} N-1 end
```

the call `{SwitchMap [1 1 2 2] Inc Dec}` returns `[2 0 3 1]`.
The function must be tail-recursive and you are not allowed to use other functions.

Solution.

```
fun {SwitchMap Xs F G}
   case Xs
   of nil then nil
   [] X|Xr then {F X}|{SwitchMap Xr G F}
   end
end
```

4 Runtime (40 points)

Consider the following function

```
fun {Last Xs}
   case Xs of X|Xr then
      if Xr==nil then X else {Last Xr} end
   end
end
```

Tables for execution times and asymptotic complexity are at the end of the exam.

4.1 Kernel Syntax (5 points)

Transform `Last` into kernel-syntax.

Solution.

```
proc {Last Xs Y}
   case Xs of X|Xr then B in
      B = (Xr==nil)
      if B then Y=X
      else {Last Xr Y}
```

```
        end
      end
    end
```

4.2 Input Argument and Size Function (5 points)

Give the input argument (that is, the I function) of `{Last Xs Y}` and an appropriate size function.

Solution. Input argument is `Xs`, size function is length of list.

4.3 Recurrence Equation (15 points)

Give a recurrence equation for the runtime $T(n)$ of `Last`.

Solution. $T(n) = c + T(n-1)$

4.4 Asymptotic Complexity (5 points)

Give the asymptotic complexity of `Last`.

Solution. $O(n)$.

4.5 `Last` With a Single `case`-Statement (10 points)

Give an equivalent definition of `Last` that only has a single **`case`**-statement but no **`if`**-statement. You can assume that `Last` is only called with lists that have at least one element.

Solution.

```
fun {Last Xs}
   case Xs
   of [X]  then X
   [] _|Xr then {Last Xr}
   end
end
```

5 Runtime (45 points)

Consider the following function:

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

Tables for execution times and asymptotic complexity are at the end of the exam.

5.1 Examples (6 points)

What does `{Pile [a]}` return?

Solution. `[a]`

What does `{Pile [a b]}` return?

Solution. `[a b b]`

What does `{Pile [a b c]}` return?

Solution. `[a b c b c c]`

5.2 Kernel Syntax (5 points)

Transform `Pile` to kernel syntax.

Solution.

```
proc {Pile Xs Ys}
   case Xs
   of nil then nil
   [] X|Xr then Yr in
      {Pile Xr Yr}
      {Append Xs Yr Ys}
   end
end
```

5.3 Input Argument and Size Function (5 points)

Give the input argument (that is, the I function) of `Pile` and an appropriate size function.

Solution. Input argument is `Xs`, size function is length of list `Xs`.

5.4 Recurrence Equation (20 points)

Give a recurrence equation for the runtime $T(n)$ of `Pile`.

Solution. $T(n) = c_1 + c_2 n + T(n-1)$

5.5 Asymptotic Complexity (5 points)

Give the asymptotic complexity of `Pile`.

Solution. $O(n^2)$

5.6 The Last Element (4 points)

Give an expression that computes `{Last {Pile Xs}}` in linear time. For the definition of `Last` see Question 4.

Solution.

```
{Last Xs}
```

6 Demand-driven Execution (20 points)

6.1 Generating Numbers (6 points)

Write a lazy function `{Double N}` that lazily computes the stream of numbers

```
N|2*N|4*N|8*N|···
```

Solution.

```
fun lazy {Double N}
   N|{Double 2*N}
end
```

6.2 Lazy Map (10 points)

Give a lazy `{LazyMap Xs F}` function, where `Xs` is a list and `F` a unary function.

Solution.

```
fun lazy {LazyMap Xs F}
   case Xs
   of nil then nil
   [] X|Xr then {F X}|{LazyMap Xr F}
   end
end
```

6.3 Scaling (4 points)

Consider the following two definitions which multiply lists of numbers `Ns` with a number `M`:

```
fun {EagerScale Ns M}
   {LazyMap Ns fun {$ N} N*M end}
end
fun lazy {LazyScale Ns M}
   {LazyMap Ns fun {$ N} N*M end}
end
```

Both return a list of numbers.
How many list elements are computed by

```
{EagerScale [1 2 3] 3}
```

Solution. `0`

How many list elements are computed by

```
{LazyScale [1 2 3] 3}
```

Solution. `0`

7 Abstract Datatypes (40 points)

In the following we are going to use and implement an abstract data type *stack*. The interface for stacks is defined by the following functions:

- `{NewStack}` returns an empty stack.
- `{IsEmpty S}` tests whether the stack is empty.
- `{Top S}` returns the topmost element of stack `S`.
- `{Push S X}` returns a stack with `X` pushed on top of stack `S`.
- `{Pop S}` returns a stack where the top-element of stack `S` has been popped.

7.1 Pushing List Elements (10 points)

Write a function `{PushAll Xs S}` that returns a stack according to the above definition with all elements of the list `Xs` pushed on the stack `S`. The first element of `Xs` must be pushed first. The function must be tail-recursive.
For example, for the stack `S` computed by

```
S={PushAll [a b] {NewStack}}
```

it holds that `{Top S}=b` and `{Top {Pop S}}=a`.
You are only allowed to use the functions defined by the interface above.

Solution.

```
fun {PushAll Xs S}
   case Xs
   of nil then S
   [] X|Xr then {PushAll Xr {Push S X}}
   end
end
```

7.2 Popping to a List (10 points)

Write a function `{PopAll S}` that returns a list of all elements of the stack `S` as obtained by popping them from the stack (in that order). The function must be tail-recursive.
For example, for the stack `S` computed by

```
S={Push {Push {NewStack} a} b}
```

it holds that `{PopAll S}=[b a]`.
You are only allowed to use the functions defined by the interface above.

Solution.

```
fun {PopAll S}
   if {IsEmpty S} then nil
   else {Top S}|{PopAll {Pop S}}
   end
end
```

7.3 Reversing Lists (5 points)

Implement a function `{Reverse Xs}` that returns a list with the elements of the list `Xs` in reverse order.
You must implement `Reverse` with the functions `PushAll` and `PopAll` from above.

Solution.

```
fun {Reverse Xs}
   {PopAll {PushAll Xs {NewStack}}}
end
```

7.4 Implementing Stacks (15 points)

Give an implementation of the *stack* abstract data type that uses lists.

Solution.

```
fun {NewStack}
   nil
end
fun {IsEmpty S}
   S==nil
end
fun {Top S}
   S.1
end
fun {Pop S}
   S.2
end
fun {Push S X}
   X|S
end
```

8 Agents with State: Group Multicast (40 points)

This assignment develops an agent that manages multicast groups. The agent understands the following messages:

- `subscribe(A)` The agent `A` is subscribed to the multicast group.
- `unsubscribe(A)` The agent `A` is unsubscribed from the multicast group.
- `multicast(M)` The message `M` is send to all members of the multicast group.

The state of the agent consists of a *multicast group*. A multicast group is implemented as a list of agents. Initially, the agent starts with an empty multicast group.
The function `NewAgent` to create agents is as follows:

```
fun {NewAgent Process InitState}
   Port Stream
   proc {Execute Stream State}
      case Stream of Message|Rest then
         {Execute Rest {Process State Message}}
      end
   end
in
   Port={NewPort Stream}
   thread {Execute Stream InitState} end
   Port
end
```

The function `FoldL` for left folding lists is as follows:

```
fun {FoldL Xs F S}
   case Xs
   of nil then S
   [] X|Xr then {FoldL Xr F {F S X}}
   end
end
```

8.1 `NewAgent` with `FoldL` (4 points)

Give a function `NewAgent` that is equivalent to the above definition but where `Execute` is implemented with `FoldL`.

Solution.

```
fun {NewAgent Process InitState}
   Port Stream
in
   Port={NewPort Stream}
   thread _={FoldL Stream Process InitState} end
   Port
end
```

8.2 Subscribing (8 points)

Implement a function `{Subscribe G A}` which returns a multicast group where the agent `A` is added to the multicast group `G` only if `A` is not yet a member of `G`.
As agents are represented by ports, testing whether two agents are the same can be done by the normal equality test. For example, if `A1` and `A2` are two agents, then `A1==A2` returns true, if and only if `A1` and `A2` are the same agent.
You are *not* allowed to use the built-in`Member` function.

Solution.

```
fun {Subscribe G A}
   case G
   of nil then [A]
   [] A1|R then
      if A1==A then G else A1|{Subscribe R A} end
   end
end
```

8.3 Unsubscribing (8 points)

Implement a function `{UnSubscribe G A}` which returns a multicast group where the agent `A` is removed from the multicast group `G`.
You are *not* allowed to use the built-in `Remove` function.

Solution.

```
fun {UnSubscribe G A}
   case G
   of nil then nil
   [] A1|R then
      if A1==A then R else A1|{UnSubscribe R A} end
   end
end
```

8.4 Multicasting (8 points)

Implement a procedure `{MultiCast G M}` where `G` is a multicast group and `M` is a message. The procedure sends `M` to all agents contained in `G`.
You are allowed to use `ForAll`.

Solution.

```
proc {MultiCast G M}
   {ForAll G proc {$ A}
                {Send A M}
             end}
end
```

Without `ForAll`:

```
proc {Multicast G M}
   case G
   of nil then skip
   [] A|R then {Send A M} {MultiCast R M}
```

```
end
```

8.5 Agent Processing Function (8 points)

Implement a function `{Process S M}` which takes a state `S` (a multicast group), a message `M`, and returns a new state. The function must implement processing of the messages as described above.

Solution.

```
fun {MCP S M}
   case M
   of subscribe(A) then
      {Subscribe S A}
   [] unsubscribe(A) then
      {UnSubscribe S A}
   [] multicast(M) then
      {MultiCast S M} S
   end
end
```

8.6 Agent Creation (4 points)

Define a function `{NewGroupAgent}` that returns an agent for multicast groups. The agent should be created with an empty multicast group.

Solution.

```
fun {NewGroupAgent}
   {NewAgent MCP nil}
end
```

9 Checking Runtime (25 points)

9.1 Result Sending (10 points)

Implement a procedure `{RunAndSend F P}` that executes the nullary function `F` in a newly created thread and sends its result to the port `P` *after* execution of `F` has terminated.

Solution.

```
proc {RunAndNotify F P}
   thread R={F} in {Send P R} end
end
```

9.2 Checking (10 points)

Implement a function `{Check P T}` which takes a nullary procedure `P` and a time value `T` in milliseconds. It returns **`true`**, if execution of the procedure `P` in a thread of its own has finished in less than or equal to `T` milliseconds. Otherwise, it returns **`false`**.

Hint: Use a port where the first message that arrives determines the function's result. The procedure `{Delay T}` suspends the executing thread for `T` milliseconds.

Solution.

```
fun {Check P T}
   Po Stream
in
   Po={NewPort Stream}
   {RunAndNotify fun {$} {P} true end        Po}
   {RunAndNotify fun {$} {Delay T} false end Po}
   Stream.1
end
```

9.3 Cheating (5 points)

Cheat the `Check` function from above! Give a nullary procedure `Cheat` such that `{Check Cheat 1000}` returns **`true`** even though there are still statements from `Cheat` being executed.

Solution.

```
local
   proc {Spin} {Spin} end
in
   proc {Cheat} thread {Spin} end end
end
```

A Execution Times for Statements

Statement	Execution Time
`skip`	0
$\langle \mathsf{x} \rangle = \langle \mathsf{y} \rangle$	c
$\langle \mathsf{x} \rangle = \langle \mathsf{v} \rangle$	c
$\langle \mathsf{s} \rangle_1 \ \langle 2 \rangle_2$	$T(\langle \mathsf{s} \rangle_1) + T(\langle \mathsf{s} \rangle_2)$
`local` $\langle \mathsf{x} \rangle$ **`in`** $\langle \mathsf{s} \rangle$ **`end`**	$c + T(\langle \mathsf{s} \rangle)$
`if` $\langle \mathsf{x} \rangle$ **`then`** $\langle \mathsf{s} \rangle_1$ **`else`** $\langle \mathsf{s} \rangle_2$ **`end`**	$c + \max(T(\langle \mathsf{s} \rangle_1), T(\langle \mathsf{s} \rangle_2))$
`case` $\langle \mathsf{x} \rangle$ **`of`** $\langle \mathsf{p} \rangle$ **`then`** $\langle \mathsf{s} \rangle_1$ **`else`** $\langle \mathsf{s} \rangle_2$ **`end`**	$c + \max(T(\langle \mathsf{s} \rangle_1), T(\langle \mathsf{s} \rangle_2))$
$\{\langle \mathsf{x} \rangle \ \langle \mathsf{y} \rangle_1 \ \cdots \ \langle \mathsf{y} \rangle_n\}$	$T_{\langle \mathsf{x} \rangle}(\mathrm{size}(I(\{\langle \mathsf{y} \rangle_1), \ldots, \langle \mathsf{y} \rangle_n\}))$

B Asymptotic Complexity for Recurrence Equations

Recurrence Equation	Asymptotic Complexity
$T(n) = c + T(n-1)$	$O(n)$
$T(n) = c_1 + c_2 n + T(n-1)$	$O(n^2)$
$T(n) = c + T(n/2)$	$O(\log n)$
$T(n) = c_1 + c_2 n + T(n/2)$	$O(n)$
$T(n) = c + 2T(n/2)$	$O(n)$
$T(n) = c_1 + c_2 n + 2T(n/2)$	$O(n \log n)$