Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

EXAM IN COURSE TDT 4165
PROGRAMMING LANGUAGES

Thursday November 30, 2005, 9.00–13.00

ENGELSK

Contact during the exam:

Wacław Kusnierczyk, Tlf 948 94 894
Ole Edsberg, Tlf 952 81 586

Exam aid code: C

No written material is permitted.
The officially approved calculator is allowed.

The exam was created by Wacław Kuśnierczyk and Ole Edsberg.

The quality of the exam was approved by Per Kristian Lehre.

Read all of the following before you start giving your answers:

- Answer briefly and concisely. Unclear and unnecessarily long answers will receive lower grades.
- For tasks where we ask you to give an argument for you answer, a correct answer without an argument will result in close to zero score for that task. Remember to give arguments!
- The programming problems must be solved with `Oz`.
- You may use the following functions and procedures from the textbook, without defining them: `Append`, `Drop`, `FoldL`, `FoldR`, `ForAll`, `Length`, `Map`, `Max`, `Min`, `Member`, `Reverse`, `Take`, `Solve`, `SolveAll`.

Problem 1: (10%)

a)

Write a function `{Map List Function}` that applies the one-argument function `Function` to each element in `List` and returns a list with the results of the applications, i the same order as the corresponding elements in `List`.

For example, the application `{Map [true false true] fun{$ Element} Element == false end}` should return the list `[false true false]`.

b)

It is possible to make variants of `Map` that are correct with respect to a) but that perform the calls to `Function` in different orders. Can this difference between the variants affect the return value of a program calling `Map`? Give convincing arguments for your answers.

Problem 2: (10%)

Consider the following producer/consumer-situation:

```
proc {Produce Stream}
  fun {Enumerate Number}
    {Delay 100}
    proc {$} {Display Number} end | {Enumerate Number+1}
  end
in
  Stream = {Enumerate 1}
end

proc {Consume Stream}
  case Stream
  of Head|Tail then {Head} {Consume Tail}
  else skip
  end
end

<Skjult kode>
```

a)

During an execution of the code above, the following sequence of numbers was shown in the Browser window (one per line): 3, 45, 8, 12, ...

Some of the code is hidden. Which of the following alternatives for the hidden code makes the observed behaviour possible according to the semantics defined in chapter 4 of the textbook? Give a convincing argument for your answer.

1.
```
proc {Display Number}
   {Delay {OS.rand} mod 1000} {Browse Number}
end
local Numbers in
  {Produce Numbers}
  {Consume Numbers}
end
```

2.
```
proc {Display Number}
   {Delay {OS.rand} mod 1000} {Browse Number}
end
local Numbers in
  thread {Produce Numbers} end
  thread {Consume Numbers} end
end
```

3.
```
proc {Display Number}
   {Delay {OS.rand} mod 1000} {Browse Number}
end
local Numbers in
  thread {Produce Numbers} end
  thread {Consume Numbers} end
  thread {Consume Numbers} end
end
```

4.
```
proc {Display Number}
   thread {Delay {OS.rand} mod 1000} {Browse Number} end
end
local Numbers in
  {Produce Numbers}
  {Consume Numbers}
end
```

5.
```
proc {Display Number}
   thread {Delay {OS.rand} mod 1000} {Browse Number} end
end
local Numbers in
  thread {Produce Numbers} end
  thread {Consume Numbers} end
end
```

6.
```
proc {Display Number}
   thread {Delay {OS.rand} mod 1000} {Browse Number} end
end
local Numbers in
  thread {Produce Numbers} end
  thread {Consume Numbers} end
  thread {Consume Numbers} end
end
```

`{Delay N}` freezes the thread in `N` milliseconds. `{Delay {OS.rand} mod 1000}` freezes the thread in a random number of milliseconds (between 0 and 1000).

b)

Which of the alternatives for the hidden code guarantees that the sequence of numbers displayed in the Browser window will be 1, 2, 3, 4,... ? Give a convincing argument for your answer.

Problem 3: (20%)

Consider the function `Function` with the following behaviour:

- The call `{Function a}` returns a value, but displays nothing in the Browser window.
- The call `{{Function a} b}` returns a value and displays `a` in the Browser window.
- The call `{{{Function a} b} c}` returns a value and displays `a` and `b` on two separate lines in the Browser window.
- ...

Generally, a call `{`$\ldots$`{Function` a_1`}` a_2`}`$\ldots a_n$`}` will return a value and display the values of the arguments a_1, a_2, ..., a_{n-1}, in that order, each on a separate line in the Browser window, for an arbitrary natural number n.[1]

a)

Use the template below to make an implementation of `Function` that will give the behaviour specified above.

```
fun {Function Argument}
   ...
end
```

b)

Use the template below to implement a function `Apply` that takes two arguments and performs a sequence of nested applications of the function (the first argument) to the values in the list (the second argument), and returns the result of the last application.

```
fun {Apply Function Arguments}
   ...
end
```

For example, each of the following applications should display `1`, `2`, `3`, og `4` on four separate lines in the Browser window:

```
{{{{Function 1} 2} 3} 4} 5}
{Apply Function [1 2 3 4 5]}
{Apply {Apply Function [1 2 3]} [4 5]}
```

[1] If $n = 0$, there is no function call! We assume $n > 0$.

Problem 4: (20%)

You have a deck of ordinary playing cards that you intend to play poker with.[2] The deck is missing some cards, but contains no duplicates.

In poker there are types of hands with special significance. (A poker hand consists of five cards). For example, all hands consisting of three cards of one rank and two cards of another rank belong to the hand type *full house*. You want to use `Oz` to find out how many different hands of each hand type it is possible to draw from your deck.[3]

A natural solution would be to calculate the number of possible hands of each type mathematically. To avoid this work, we will instead try to use a generate-and-test strategy in the relational programming model.

a)

Define data structures that are suitable to represent cards, deck and hands in `Oz`. Use BNF or EBNF. Take a look at the next subproblem to find out how the data structures will be used.

b)

Write a function `{CountHands Deck HandType}`. The parameter `Deck` is a representation of the deck of the form that you defined in the previous subtask. The parameter `HandType` is a boolean function som defining a hand type. An example of such a hand type function is `FullHouse`; the call `{FullHouse Hand}` returns `true` if the hand `Hand` belongs to the type *full house*, and `false` otherwise. (You don't need to implement the hand type functions.) The call `{CountHands Deck HandType}` should return the number of possible hands from the deck `Deck` that belong to the hand type defined by `HandType`.

Example: The call `{CountHands Deck FourOfAKind}` should return the number 624 ($13 \times (52-4)$) if `Deck` is a complete deck and `FourOfAKind` is a function returning `true` for hands that contain for cards of the same rank and `false` otherwise.

Use the relational computation model. We will accept a naive generate-and-test solution. Explain the disadvantages of this kind of solution.

Problem 5: (20%)

The message pssing, concurrent computation model from chapter 5 of the textbook introduces ports. The syntax for port-statmenets is as follows:

Creating a port:	`{NewPort` ⟨Stream⟩ ⟨Port⟩`}`
Sending til port:	`{Send` ⟨Port⟩ ⟨Message⟩`}`

where ⟨Stream⟩, ⟨Port⟩, and ⟨Message⟩ stand for identifiers. If we consider ports as data abstractions, we can specify their behaviour with the following assertions:

{ `Stream` $\mapsto s \ \wedge \ s$ is unbound $\wedge$ `Port` $\mapsto p \ \wedge \ p$ is unbound }
`{NewPort Stream Port}`

[2] A complete deck contains one card for each possible combination of suit and rank. The suits are: clubs, diamonds, spades and hearts. The ranks are: ace, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king.

[3] Two hands are identical if they contain the same cards, irrespectively of the order in which they were drawn. If there are N cards in the deck, the total number of possible hands are $\binom{N}{5} = \frac{N!}{5!(N-5)!}$. Hint: You don't need to apply such calculations to solve this problem.

{ `Stream` $\mapsto s \ \wedge$ s is a read-only view of $s' \wedge$
  `Port` $\mapsto p \ \wedge$ p is a port associated with $s' \wedge$
  s' is a newly created unbound variable }

{ `Port` $\mapsto p \ \wedge$ p is a port associated with $s \ \wedge$ s is unbound $\wedge$
  `Message` $\mapsto m \wedge$ (m is unbound $\vee$ m has some value v) }
`{Send Port Message}`
{ `Port` $\mapsto p \ \wedge$ p is a port associated with $s' \wedge$
  `Message` $\mapsto m \wedge$ (m is unbound $\vee$ m has the value v)[4] $\wedge$
  s' is a newly created unbound variable $\wedge$
  s is bound to a newly created tuple with '|' as its label,
    and m and a read-only view of s' as its fields }

The arrow '$\mapsto$' symbolizes a binding, in the current environment, between an identifier and a variable.

Following the textbook, we do not distinguish between bound variables and the values they are bound to. (For example, we say 'p is a port' rather than 'p is a variable bound to a port value'.)

We assume that `Port`, `Stream`, and `Message` are declared before they are used, and that `NewPort` and `Send` are bound to procedures for creating a new port and sending to a port, respectively (see below).

For the sake of simplicity we let the behaviour of `NewPort` be undefined in those cases wheree one or both of the arguments are bound variables. We also let the behaviuor of `Send` be undefined in those cases where the first argument is an unbound variable or a value that is not a port.

a)

Do the assertions guarantee that the stream associated with a port can neverbe extended (that is, that its end — an unbound variable — can never be bound) in any way other than by applying `Send` to the port? Justify your answer.

b)

What kinds of language elements are necessary to make an implementation of ports that would satisfy the above specification?[5]

Can ports be implemented in:

- the declarative, sequential computation model?
- the declarative, concurrent computation model?
- the sequential computation model with explicit state?
- the computation model with shared-state concurrency?

Give convincing arguments for your answers.

c)

Use the template below to implement the procedures `NewPort` and `Send`. Assume that `NewPort` will always be applied to two unbound variables, and that the first argument to `Send` will always be a port (you do not need to verify these assumptions in your code).

[4] That is: if m is unbound before the execution, it is unbound after the execution; if m is bound to a value v before the execution, it is bound to the same value after the execution.

[5] Ports can trivially be implemented with ports. Please do not include this possibility in your answer.

```
proc {NewPort Stream Port}
   proc {Send Message}
      ...
   end
   ...
in
   ...
   port(send:Send)
end

proc {Send Port Message}
   ...
end
```

d)

Does your implementation guarantee that the stream associated to a port can never be extended in any way other than by applying `Send` to the port? Is it possible to make an implementation that guarantees this without modifying the implementation of the `Oz` interpreter/compiler? Give convincing arguments for your answers.

Problem 6: (20%)

In this task we will consider a programming language `f`. `f` is a subset of `Oz` with a lot in common with lambda calculus. The syntax of `f` is defined by the following EBNF grammar:

```
<Program> ::= '{' Browse <Expr> '}'
   <Expr> ::= '{' <Expr> { <Expr> } '}'
            | fun '{' '$' <Ident> { <Ident> } '}' <Expr> end
            | if <Ident> '==' <Expr> then <Expr> else <Expr> end
            | <Ident> '*' <Expr>
            | <Ident> '-' <Expr>
            | <Ident>
            | <Integer>
```

`<Ident>` is an arbitrary identifier in `Oz`, and `<Integer>` is an arbitrarty integer in `Oz`.

Programs in `f` can be executed by feeding them to `Oz` just like other `Oz` programs. In other words, we assume that the semantics of programs in `f` are the same as if these programs were run in `Oz`.

a)

Assume that we have created a tokenizer for `f`. The tokenizer takes a program text as input and splits it into a list of tokens. The possible tokens are the special symbols `'$'` `'{'`, `'}'`, `'=='`, `'-'` and `'*'`, the reserved keywords `'fun'`, `'if'`, `'then'`, `'else'`, `'end'`, identifiers and integers. Identifiers are represented by `Oz`-atoms wrapped in records with the label `ident`. Integers are represented by `Oz`-integers wrapped in records with the label `int`. Note that the occurence of `Browse` in the start of every program is counted as an identifier.

Given the following program in `f`:

```
{Browse {fun {$ Number} Number + 2 end 2}}
```

The resulting token list would be:

```
['{' ident('Browse') '{' 'fun' '{' '$' ident('Number') '}'
 ident('Number') '+' int(2) 'end' int(2) '}' '}']
```

Write a function `{ValidateSyntax Tokens}` that returns `true` if `Tokens` represents a valid program according to the grammar and `false` otherwise.

b)

Write a program in `f` that calculates 1000! ($1 \times 2 \times \ldots \times 1000$) and displays the result on the screen.

Hint: The difficult part of this task is to call the function recursively. The following might be useful as a part of the solution:

```
fun {$ Function Argument} {Function Argument Function} end
```

END OF EXAM