Exam in TDT4165 Programming Languages: Solutions with Explanations

Authors: Wacław Kuśnierczyk, Ole Edsberg*

Exam Date: November 30, 2006

Solutions

Problem 1 (10%)

Question 1a (50%)

Answer. `Map` is defined as a function that takes two arguments, a list of n arguments and a one-parameter function, and returns a list of the results of an application of that function to the elements of the list:

$$\texttt{\{Map [}e_1\ e_2\ \ldots\ e_n\texttt{]}\ f\texttt{\}} \mapsto \texttt{[}f(e_1)\ f(e_2)\ \ldots\ f(e_n)\texttt{]}$$

A correct definition of `Map` shown in Fig. 1; it uses recursion, and is perhaps the most intuitive.

```
fun {Map List Function}
  case List of
     Head|Tail then
     {Function Head}|{Map Tail Function}
  [] nil then
     nil
  end
end
```

Figure 1: A recursive definition of `Map`.

Discussion. The order of patterns in the definition above may be inversed with no influence on the correctness, while possibly slowing down the process of mapping over longer lists—two patterns rather than one are matched at each recursive step (except for the final step).

`Map` can also be defined in terms of the the procedure `FoldR`, as in Fig. 2.

`Map` can also be defined as in Fig. 3, using an accumulator, so that the process of mapping is iterative (but the use of `Reverse` spoils the gain in efficiency).

*Contact info: {waku,edsberg}(at)idi.ntnu.no

```
fun {Map List Function}
  {FoldR List
         fun {$ Element Rest} {Function Element}|Rest end
         nil}
end
```

Figure 2: A definition of `Map` in terms of `FoldR`.

```
fun {Map List Function}
  fun {ReverseMap List Accumulator}
    case List of
       Head|Tail then
       {ReverseMap Tail {Function Head}|Accumulator}
    [] nil then
       Accumulator
    end
  end
in
  {Reverse {ReverseMap List nil}}
  % alternatively {ReverseMap {Reverse List} nil}
end
```

Figure 3: A definition of `Map` using an accumulator.

Question 1b (50%)

Answer. The order in which `Map` applies `Function` to the elements of `List` has no influence on the order of the values in the list returned by `Map`, but it may have influence on those values themselves. In a declarative model of computation, the value of an application of a function to a particular input must be the same on every execution, and so `Map` will always return the same result, given a particular input. In a non-declarative model of computation, `Map` may return different results in different calls, even if given the same input, since the function it applies to the elements of a list may give different results on different executions.

Discussion. Following the first definition above (answer to Question 1a), `Map` applies `Function` to the elements of `List` in the left-to-right order, i.e., first to the first element of the list, then to the second element of the list, and so on. This order can be easily reversed to be right-to-left, as in Fig. 2. Nevertheless, for a given input `[`e_1 e_2 ... e_n`]` and f, the returned list is always `[`$f(e_1)$ $f(e_2)$ ... $f(e_n)$`]`—the order of applications of f has no influence on the order of the results. (All these variants are correct wrt. to the specification in Question 1a.) But the order of application of f may have influence on the values $f(e_i)$, e.g., when explicit state (or other side effects) is used, or when both threads and exceptions are allowed—that is, when the model of computation is non-declarative. Given the definitions shown in Fig. 4, we observe that:

- the expression `{Map [1 2 3] F1} == {Map [1 2 3] F1}` will always evaluate to `false`;
- the expression `{Map [1 2 3] F2} == {Map [1 2 3] F2}` will likely evaluate to `false` on most executions.

```
F1 = local State in
  State = {NewCell 0}
  fun {$ Argument}
    State := @State + 1
    Argument * @State
  end
end
fun {F2 Argument}
  {OS.rand}
end
```

Figure 4: Two non-declarative functions.

Problem 2 (10%)

Question 2a (50%)

Answer. The apparently random sequence `3`, `45`, `8`, `12`, ... may have been observed only if the hidden code was as in the alternatives 5 and 6 (though it is highly unlikely that we would actually observe such a sequence).

Discussion. See below.

Question 2b (50%)

Answer. The strictly increasing sequence `1`, `2`, `3`, ... can be guaranteed only by the code in the alternative 2.

Discussion. The observed output apparently does not correspond to the order of elements in the stream. `Producer` generates an endless stream `[`p_1 p_2 ... `]` of no-parameter (nullary) procedures, such that each procedure p_i, when applied to no arguments, prints (displays) its number, i, in the browser window. That is, for any natural number $i > 0$, `{`p_i`}` is guaranteed to print i.

The stream is infinite: `Producer` calls itself recursively, and there is no stop condition in its definition. In the hidden code, for the call `{Consume Numbers}` to be reached the call `{Produce Numbers}` must therefore be run in a separate thread. If the call `{Produce Numbers}` is not enclosed in a `thread ... end` statement, there will be no output in the browser window. Whether `{Consume Numbers}` is enclosed in a `thread ... end` statement or not, has no practical meaning in this case. (However, if two `{Consume Numbers}` calls are present, then the first one must be run in a new thread for the other to be reachable.)

When the (a) consumer executes the call `{Head}`, further execution of the consumer is stopped until `{Head}` returns. If the `{Display Number}` call executed within `{Head}` does not start a new thread, `Consumer` can proceed only when `{Display Number}`, and thus `{Head}`, has returned (i.e., only when `Number` has already been displayed). Therefeore, if `Display` starts a thread, then the output is likely to be a sequence of numbers displayed in the order in which the scheduler choses particular `Display`-threads. If `Display` does not start a new thread, the sequence is guaranteed to be strictly increasing (in the case of one consumer thread). If more consuments are running, each of them outputs an increasing sequence of numbers, but ther outputs are interleaved, and so that smaller numbers may follow larger numbers, but the output can never start with `3` (the first number in the output is the first number output by the first consumer chosen by the scheduler,

and it must be `1`).

The six alternative pieces of hidden code would give the following behaviour:

1. No output (`{Consume Numbers}` is unreachable).
2. The sequence `1`, `2`, `3`, ... (guaranteed by unthreaded `Display`).[1]
3. A sequence such as `1`, `2`, `1`, `3`, ..., where each number occurs twice,[2] and a number n output by one of the consumers can never precede any number $m < n$ output by the same consumer.[3]
4. No output (`{Consume Numbers}` is unreachable).
5. A more or less random sequence, with each number occurring once (cf. footnote 2), and with larger numbers more likely to appear later in the sequence.
6. As above, but with each number occurring twice.

Note: The delay between the production of two consecutive elements of `Numbers` is 100 ms, while the delay between the execution of a `{Display Number}` and the actual printing of `Number` is 0–1000 ms. It is therefore highly unlikely that `45` would appear as second in the output sequence. (With a fair scheduler, by the time the 45^{th} element of `Numbers` is generated at least 35 numbers should have already been displayed.) However, `{Delay` n`}` suspends the current thread for n ms, but the thread may remain suspended for an arbitrarily longer time. It is not *impossible* that `45` appears as second in the output (in the case of the threaded `Display`). (If your answer to Question 2a is that it is impossible to observe such a sequence, it is considered correct, provided that you gave an appropriate argumentation.)

Problem 3 (20%)

Question 3a (50%)

Answer. `Function` can be implemented using higher-order programming, with or without explicit state, as in Fig. 5.

Discussion. Since the result of a call `{Function` a_1`}` must be applicable to another single argument, e.g., as in the call `{{Function` a_1`}` a_2`}`, `Function`, when applied to an argument, must return a one-parameter (unary) function. Furthermore, the result of `{{Function` a_1`}` a_2`}` must again be applicable to a single argument, i.e., it must again be a unary function. In general, the result of a call `{...{{Function` a_1`}` a_2`}` ... a_n`}` must be a unary function, for any $n > 0$; let us denote it by f_n. Then `{`f_n a_{n+1}`}` prints a_n in the browser window, and returns f_{n+1}, which follows the specification above. Furthermore, there is no a_0 to be printed by f_1. There is no requirement that $f_i = f_j$ for all i, j.

In the first solution, the first call to `Function`, `{Function` a_1`}`, does not display anything, but returns a new function, f_2. In all subsequent calls `{`f_i a_i`}`, $i > 1$, the function f_i will display a_{i-1} and return a new function, f_{i+1}. The one-call delay in printing is realized by each f_i having

[1] The sequence is such that if the number n is printed, then every number $m < n$ must have already been printed exactly once, and no number $m > n$ can have been printed yet.

[2] If we could wait infinitely, we would see all natural numbers printed, each exactly twice.

[3] We are guaranteed that when the number n is printed, then each number $m < n$ must have already been printed at least once; if the number n has been printed twice, then every number $m < n$ must have already been printed twice.

```
% without explicit state
fun {Function Argument}
  fun {$ NextArgument}
    {Browse Argument}
    {Function NextArgument}
  end
end

% with explicit state
fun {Function Argument}
  State = {NewCell Argument}
  fun {Function Argument}
    {Browse {Exchange State $ Argument}}
    Function
  end
in
  Function
end
```

Figure 5: Definitions of `Function` using higher-order programming and explicit state.

the argument a_{i-1} of the preceding call to f_{i-1} in its closure. Note: At each step i, f_i is a new function, i.e., $f_i \neq f_j$ for any $i \neq j$, while `Function` (f_1) does not change (it is stateless).

In the second solution, a cell is used to keep the argument of a call to be printed in the next call. In the first call to `Function`, `{Function` a_1`}`, a cell, c, is created with a_1 as its content, and a new function, f_2, that has c in its closure is returned. There is no printout at this step. In every subsequent call `{`f_i a_i`}`, the content of c, i.e., a_{i-1}, is printed and replaced by the next argument, a_i. Note: At each step i, f_i is the same stateful function f_2 (i.e., $f_i = f_j$ for any $i, j > 1$), though each time with a new content of the cell c (i.e., the content of c as seen by f_i is not necessarily the same value as that seen by f_j, for any $i \neq j$). `Function` (f_1) does not change (it is stateless).

An alternative solution using explicit state is given in Fig. 6. The disadvantage is that i) the cell is external to `Function`, and thus its content can be changed without making a call to `Function`; ii) a particular value, e.g., `nil`, must be used to initially fill the cell, and this value cannot then be used as an argument to `Function` (it would not be displayed in the next call); iii) if an unbound variable is passed to `Function`, the process suspends; iv) a call to `Function` will not display the argument only on the first occasion, otherwise `Function` will always display a value (see Fig. 6; the version with explicit state as in Fig. 5 creates a new state cell on each call). (Nevertheless, this solution is considered correct.)

Question 3b (50%)

Answer. `Apply` can be implemented as in Fig. 7.

Discussion. `Apply` is not the same as `Map`. While `Map` takes as arguments a list and a unary function and returns a list of the results of application of the function to each element of the list, `Apply` takes as arguments a list and a unary function[4] and returns a function. This can be clearly seen in the example call `{Apply {Apply Function [1 2 3]]} [4 5]}`, where the return value of the inner call `{Apply Function [1 2 3]}` is `Apply`-ed to `[4 5]` in the outer call. Furthermore, `Apply` must iteratively (or recursively, see Fig. 8) perform a sequence of nested applications,

[4]The order of arguments of `Apply` is actually the inverse, but this plays here no role.

```
State = {NewCell nil}
fun {Function Argument}
  if @State \= nil then {Browse @State} end
  State := Argument
  Function
end

% undesirable behaviour
{{Function nil} 2}    % -> no printout
{{Function 1} 2}      % -> displays 1
{{Function 1} 2}      % -> displays 2, 1
```

Figure 6: An alternative definiton of a stateful `Function`.

```
fun {Apply Function Arguments}
  case Arguments of
     Head|Tail then
     {Apply {Function Head} Tail}
  [] nil then
     Function
  end
end
```

Figure 7: A definition of `Apply` with iterative behaviour

rather than a sequence of independent applications. The call `{Apply Function [1 2 3]}` must give the same output as the call `{{{Function 1} 2} 3}`, rather than as the sequence of calls `{Function 1} {Function 2} {Function 3}`.

The definition above implements an iterative process. It is also possible to implement a recursive process—see Fig. 8. It is much less effective: The stack grows linearly with the depth of recursion, and it is necessary to first reverse the list of arguments (a recursive process, again). (This definition was considered correct.)

```
fun {Apply Function Arguments}
  fun {Apply Function ReversedArguments}
    case ReversedArguments of
       Head|Tail then
       {{Apply Function Tail} Head}
    [] nil then
       Function
    end
  end
in
  {Apply Function {Reverse Arguments}}
end
```

Figure 8: A definition of `Apply` with recursive behaviour.

Problem 4 (20%)

Question 4a (40%)

Answer. The data structures for representing cards, hands, and decks may be defined as in Fig. 9.

$$
\begin{array}{rcl}
\langle \mathsf{deck} \rangle & ::= & \texttt{[} \{\langle \mathsf{card} \rangle_i\} \texttt{]} \\
\langle \mathsf{hand} \rangle & ::= & \texttt{[} \langle \mathsf{card} \rangle_1 \ \langle \mathsf{card} \rangle_2 \ \langle \mathsf{card} \rangle_3 \ \langle \mathsf{card} \rangle_4 \ \langle \mathsf{card} \rangle_5 \texttt{]} \\
\langle \mathsf{card} \rangle & ::= & \texttt{card(suit:}\langle \mathsf{suit} \rangle \texttt{ rank:}\langle \mathsf{rank} \rangle\texttt{)} \\
\langle \mathsf{suit} \rangle & ::= & \texttt{heart} \mid \texttt{diamond} \mid \texttt{spade} \mid \texttt{club} \\
\langle \mathsf{rank} \rangle & ::= & \texttt{ace} \mid \texttt{two} \mid \texttt{...} \mid \texttt{ten} \mid \texttt{jack} \mid \texttt{queen} \mid \texttt{king}
\end{array}
$$

Additional conditions:

1. In $\langle \mathsf{deck} \rangle$, $0 < i \leq n$ is a subscript, where $0 < n \leq 52$ is the count of cards in a deck; $\langle \mathsf{card} \rangle_i$ appears exactly once for each i; $\langle \mathsf{card} \rangle_i \neq \langle \mathsf{card} \rangle_j$ for any $i \neq j$.
2. In $\langle \mathsf{hand} \rangle$, $\langle \mathsf{card} \rangle_i \neq \langle \mathsf{card} \rangle_j$ for any $i \neq j$.

Figure 9: Syntax for representing cards, hands, and decks, in EBNF. Subscripts are used to distinguish between different occurences of the same nonterminal on the right-hand side of a production rule. Additional conditions are needed since the syntax cannot be completely specified by a context-free grammar. In the rule for $\langle \mathsf{rank} \rangle$, ellipsis ('...') is not a part of the syntax, its meaning is that some terminals to which $\langle \mathsf{rank} \rangle$ may evaluate are not visible.

Discussion. A data structure is an object that stores data. In programming, the definition of a data structure involves (roughly) three elements: i) definition of the behaviour of the data structure as an abstract data type (ADT); ii) representation, in a particular language, of the ADT in terms of the data structres already available in that language, or implementation of the ADT as a new structre (requires recompiling of the language's interpreter/compiler); iii) (optional) definition of a special syntax, for convenient coding. Strictly speaking, BNF (and EBNF) is used to specify the *syntax* of programs valid in a programming language, but not to specify the data structures as objects. We can specify, as part of the syntax, how data structures are to be represented in the code, but it does not specify their internal layout.

In Question 4a, you were asked to define data structures using BNF/EBNF; this (over)simplification was made to reduce the amount of your work, under the assumption that using a particular Oz syntax you will think of the underlying data structures, such as lists and records. For completeness, here we will follow all three steps mentioned above, though we will not discuss operations on the defined ADTs.

A card can be abstractly represented as a pair of a suit and a rank:

$$card = \langle suit, rank \rangle$$

where *suit* is an instance of the type $Suit = \{\mathsf{heart}, \mathsf{diamond}, \mathsf{spade}, \mathsf{club}\}$, *rank* is an instance of the type $Rank = \{\mathsf{ace}, \mathsf{two}, \ldots, \mathsf{ten}, \mathsf{jack}, \mathsf{queen}, \mathsf{king}\}$, *card* is an instance of the type $Card = \{\langle suit, rank \rangle \mid suit \in Suit, rank \in Rank\}$, and heart, ace, etc., are distinct objects. (Here we define types extensionally, i.e., by means of listing all their instances, and use the symbol '$\in$' to denote that an object is an instance of a particular type.)

An n-card deck can be represented as a set of n cards:

$$deck = \{card_1, card_2, \ldots, card_n\}$$

Note that if a deck is represented as a set rather than as a tuple, it is guaranteed not to include the same card twice. A 5-card hand can be represented in the same way. (A hand is actually a deck of a particular size.)

To implement this in Oz, we may use records, tuples (which are records), lists (which are nested records), literals, numbers, names, etc. Here, we make this particular choice:

1. Decks and hands are implemented as lists of cards.
2. Cards are represented as records labelled 'card', with two features labelled 'suit' and 'rank', with the values of the corresponding fields being a suit and a rank, respectively.
3. Suits are represented as the literals 'heart', 'diamond', 'spade', 'club'.
4. Ranks are represented as the literals 'ace', 'two', ..., 'ten', 'jack', 'queen', 'king'.

In an Oz program, a representation of these structures and elements might follow the syntax given in Fig. 9.

We might decide to introduce a completely new syntax:

$$\begin{aligned} \langle \mathsf{deck} \rangle &::= \texttt{deck}\ \{\langle \mathsf{card} \rangle_i\}\ \texttt{end} \\ \langle \mathsf{hand} \rangle &::= \texttt{hand}\ \{\langle \mathsf{card} \rangle_i\}\ \texttt{end} \\ \langle \mathsf{card} \rangle &::= \texttt{card}\ \langle \mathsf{suit} \rangle\ \langle \mathsf{rank} \rangle\ \texttt{end} \end{aligned}$$

with suits and ranks as before. We would need to define the semantics of these new syntactic constructs, and perhaps recompile the language with new basic data structures.

Question 4b (60%)

Answer. The function `CountHands` may be implemented as in Fig. 10.

```
fun {CountHands Deck HandType}
  {Length {Filter {SolveAll fun {$} {Hand Deck} end}
                  fun {$ Hand} {HandType Hand} end}}
end
```

Figure 10: An implementation of `CountHands`.

Discussion. To count hands of a particular type that can be drawn from a particular deck, we employ the following generate-and-test algorithm: i) generate a list of all hands that can be drawn from the deck; ii) remove from this list those hands that are not of the demanded type; iii) return the count of the remaining elements. To do this, we need a function that generates a list of possible hands, given a deck. Question 4b explicitly requires us to use the relational model, and we realize this by implementing a function `Hand` that nondeterministically picks one hand from the given deck. `SolveAll` is then used to return a list of all solutions, i.e., a list of all possible hands, given a deck. This list is then filtered using `Filter` and `HandType` (`Filter` is a built-in in Oz, but may be easily implemented, as in Fig. 11). Finally, `Length` is applied to return the length of the filtered list.

An implementation generating all possible hands form a deck must take care of the following conditions: i) no hand contains the same card twice; ii) if a hand is represented as a list of cards, then those lists of cards that are permutations of each other are effectively representations of the

```
fun {Filter List Predicate}
  case List of Head|Tail then
    if {Predicate Head} then Head|{Filter Tail Predicate}
    else {Filter Tail Predicate} end
  else nil end
end
```

Figure 11: An implementation of `Filter`.

same hand; iii) no hand should be represented twice in a list of hands. Given a deck of n cards, we can realize these requirements using several different strategies:

1. Generate all possible combinations of 5 cards drawn separately from the deck (with repetitions, i.e., each card is drawn independently from a full deck), and then filter out all hands containing a particular card more than once, and remove redundant representations of the same hand.
2. Generate all possible combinations of 5 cards drawn sequentially from the deck (without repetitions, i.e., when a card is drawn, it is drawn from a deck that contains all cards of the original deck except for those already drawn), and then remove redundant representations of the same hand.
3. Generate only non-redundant, non-repetitive combinations of 5 cards from the deck.

Observe that there is a huge difference in how these algorithms consume time and space. From a deck of n cards, the first will generate n^5 hands (you pick 1 out of n cards, then you pick another 1 card out of n, and so on). The second will generate $n!/(n-5)!$ hands (you pick 1 out of n cards, then another out of $n-1$ cards, and so on). The last will generate $n!/5!(n-5)!$ cards (see Fig. 13 for how it can be done). For a deck of 52 cards, these numbers are 380204032, 311875200, and 2598960, respectively. Furthermore, in the first and second cases you need to go through the whole list, making a number of tests and comparisons, to filter out incorrect and redundant representations of hands.

Figure 12 shows a function `Hand` that implements the third strategy. (Implementations of the first and the second were not uncommon in your solutions. They were not considered incorrect, unless you forgot to address redundancy.) `Hand` nondeterministically picks five cards from one deck. Note that it is guaranteed that no card will appear more than once in a hand, and that a particular set of cards will be represented by exactly one permutation, without any testing and filtering.

```
fun {Hand Deck}
  fun {Pick N List}
    case List of Head|Tail then
       choice
          Head|{Pick N-1 Tail}
       [] {Pick N Tail} end
    else N=0 nil end
  end
in
  {Pick 5 Deck}
end
```

Figure 12: A nondeterministic implementation of `Hand`.

Figure 13 shows how the rest of the solution may be implemented. To generate a full 52-card

deck, call `FullDeck = {SolveAll Card}`. To calculate how many hands of the type *flush*[5] can be drawn from the full deck, you should call `Count = {CountHands FullDeck Flush}` (do not even try). Figure 14 shows how `Flush` can be implemented within this framework.

```
fun {Card}
  card(suit:{Suit} rank:{Rank})
end

fun {Suit}
  choice heart [] diamond [] spade [] club end
end

fun {Rank}
  choice
     ace [] two [] three [] four [] five [] six [] seven [] eight
  [] nine [] ten [] jack [] queen [] king end
end
```

Figure 13: Support code for `CountHands`.

```
fun {Flush Hand}
  case {Suits Hand}
  of [Suit Suit Suit Suit Suit] then true
  else false end
end

fun {Suits Hand}
   {Map Hand
        fun {$ Card}
          case Card of card(suit:Suit rank:_)
          then Suit end
        end}
end
```

Figure 14: An implementation of `Flush`.

Problem 5 (20%)

Question 5a (20%)

Answer. No, the specifications do not guarantee that the end of a stream associated to a port can never be extended (bound) otherwise than by applying `Send` to the port.

Discussion. The stream cannot be extended directly by binding its tail, since its tail (s in the specifications) is a variable bound to a read-only view of an unbound variable (s' in the specifications) visible only to the port. However, the specifications do not guarantee that the port cannot be asked to bind s' (and thus changing the content visible through s). The specifications do not specify the port's internal structure either, so that there is no guarantee that the 'hidden' variable s' is actually hidden.

[5] Flush is a hand containing five cards of any rank, but all of the same suit.

The expression 'p is a port associated with s' is underspcecified: It makes no claim about the nature of this association. In fact, a port can be represented as a record with the unbound variable s' as one of its fields (some of you have proposed such a solution, contrarily to the code template of Question 5c), making this variable directly accessible from outside of the port. In Oz, ports are implemented in a way that guarantees that only `{Send` p`}` can be used to extend the stream of the port p, but this is an implementational detail not given in the specifications above. Figure 15 shows an implementation of ports that strictly follows the specifications, yet does not guarantee security of the stream. (The answer 'yes' was considered correct provided that you gave a sufficient argumentation based on the fact that the end of a port's stream is a read-only view.)

Question 5b (20%)

Answer. Ports can be implemented in the sequential and concurrent models with explicit state, but not in the declarative models, whether with or without concurrency.

Discussion. To implement ports following the specifications, we clearly need i) dataflow variables, ii) read-only views, iii) records (tuples). Furthermore, since a port is an object that changes its state (the unbound variable read-only visible at the end of the associated stream) while keeping its identity, we need explicit state. To implement ports, concurrency (threads) is not needed. Ports are of no use in a sequential model, but they are implementable in such a model. (We can trivially implement ports in terms of ports, but pre-defined ports are not necessary to implement ports.)

Dataflow variables are an essential part of every version of the kernel language. Read-only views are introduced in Sec. 3.7.5 of the pensum book as a part of the sequential declarative kernel language. (An answer in which you argued that read-only views are not a constitutive part of the kernel language and thus ports are not implementable according to the specifications was also considered correct.) Records are an essential part of every version of the kernel language. Eksplicit state is present only in non-declarative versions of the kernel language. Therefore, ports:

- cannot be implemented in the sequential declarative kernel language, since it lacks expressivity for explicit state;
- cannot be implemented in the concurrent declarative kernel language, for the same reason (threads are of no help);
- can be implemented in the sequential kernel language with explicit state (the lack of threads is no obstacle);
- can be implemented in the concurrent kernel language with explicit state (the shared-state concurrency model), since it is an extension of the one listed immediately above.

Question 5c (40%)

Answer. Ports can be implemented, e.g., as in Fig. 15.

Discussion. A port needs explicit state to keep trace of the shifting end of the associated stream. This is realized by means of the cell `EndPointer`. When a new port is created, a new cell is made and initialized with an unbound variable. Then the stream to be associated with the port, which must initially be an unbound variable, becomes bound to a read-only view of the content of the port's cell. The cell is visible only inside the procedure `Send` defined within `NewPort` (the cell is inside the procedure's closure). A port is then represented as a record with one field called `send` whose value is the procedure `Send`.

When the value of the `send` field of a port is applied to any object, `Message`, (including an unbound

```
proc {NewPort Stream Port}
   proc {Send Message}
      HiddenEnd
   in
      {Exchange EndPointer $ HiddenEnd}
         = Message|!!HiddenEnd
   end
   EndPointer = {NewCell _}
in
   Stream = !!@EndPointer
   Port = port(send:Send)
end

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

Figure 15: An implementation of ports.

variable), a new unbound variable, `HiddenEnd`, is created, and the unbound variable kept inside the cell `EndPointer` is bound to a new tuple made of `Message` and a read-only view of `HiddenEnd` and named `|`, thus extending the stream. The content of `EndPointer` is then replaced by `HiddenEnd`, so that the port is able to extend the stream again.

The procedure `Send` is defined so that when it is applied to a port and a message, it simply forwards the message to the port's `Send`, i.e., it calls the port's internal procedure `Send` accessible through the port's field `send`.

(The picture is somewhat more complicated when you consider the semantics of read-only views, but this has no practical influence on how the implementation presented here works. As an exercise, you may study the details on your own.)

Question 5d (20%)

Answer. No, the implementation given in the answer to Question 5c does not guarantee this security.

Discussion. The question was fomulated as follows: 'Does the implementation guarantee that the stream associated to a port can never be extended in any way other than by applying `Send` to the port?' The (global) procedure `Send` is defined in terms of the (internal) procedure `Send` of the port passed to the former as an argument. These are two distinct procedures, and only the former can be applied to a port (and a message). The question was thus about the global `Send`. Observe that *any* procedure defined in the same way as the global `Send` is able to send a message to a port without calling the global `Send`. In fact, it is not even necessary to define such a procedure, as the port's `send` can be used directly, as illustrated in Fig. 16. Note also that in this simple program several messages are sent to a single port without the use of threads (`Browse` does start a separate thread, but we use it only for the purpose of visualization, to convince you that the program works in a sequential model; it is not a part of the program).

Since the implementation follows the specifications, and it does not guarantee that applying `Send` to a port is the only way to extend the port's stream, it is clear the the specifications cannot guarantee this either—otherwise there would be a contradiction. (Some of you answered 'yes'

```
proc {AlternativeSend Port Message}
  {Port.send Message}
end
Stream
Port = {NewPort _}

{AlternativeSend Port hello}
{proc {$ Port} {Port.send hello} end Port}
{Port.send hello}

{Browse Stream}
% displays: hello|hello|hello|!!_
```

Figure 16: An example of sending to a port without the use of `Send`.

to Question 5a and 'no' to Question 5d, without discussing this apparent clash.) We could, still fulfilling the specifications, implement ports so that `NewPort` binds `Port` not to `port(send:Send)`, but rather to `port(send:Send end:EndPointer)` (as some of you did, despite the template), making it possible to extend the stream even without using the port's internal `Send`.

To secure ports, some of you suggested to use names and the wrapper/unwrapper approach of Sec. 3.7.5 of the pensum book. Figure 17 shows an implementation of ports that guarantees that only an appplication of the global `Send` to a port can send a message to the port (and thus extend its stream). To send a message to a port, the port's `SecureSend` has to be passed a message and a key. The key must be identical to the unique name hidden inside the port; if it is, the message is sent in the same way as in the implementation in Fig. 15. If the key is not correct, the port ignores the call (you may prefer it to raise an exception).

When a port is created, it calls `Send` with its own key as the message. `Send`, when called with a new port (a port it has never seen before),[6] registers this port in a dictionary, where the port is the index, and the port's key (passed to `Send` as `Message`) is the value; `Send` does not send any message to the port at that time. When `Send` is passed an already registered port, it retrieves the port's key from the dictionary and uses it in a call to the port's `SecureSend`. Note that, according to the semantics of Oz, it is impossible to compromise this solution: A port's name is visible only to the port (which has no way to report it), and to `Send` which keeps it in a dictionary visible only to `Send` itself (and has no way to report it).

Problem 6 (20%)

Question 6a (70%)

Answer. Figure 18 shows how the syntax validator for the given grammar can be implemented using relational programming.

Discussion. The syntax validator takes a list of tokens (`Tokens`) and tries to parse from it a valid program, so that no tokens are left. It calls the procedure `ParseProgram` that parses a valid program from `Tokens`, returning (in the parameter `Rest`) the rest of the tokens that remain after the program has been parsed. If `ParseProgram` cannot parse a valid program, it fails. If it does not fail, the anonymous function passed to `SolveAll` in `ValidateSyntax` returns `true`.

[6] `Send` knows that it has not seen a port before from that there is no entry for this port in its dictionary; we can safely make the dictionary return `nil` in case it finds no entry, since the key to a port must be a name, and thus `nil` can never be a key to a port.

```
proc {NewPort Stream Port}
   proc {SecureSend Message Key}
      if Key == HiddenKey then HiddenEnd in
         {Exchange EndPointer $ HiddenEnd}
            = Message|!!HiddenEnd end
   end
   EndPointer = {NewCell _}
   HiddenKey = {NewName}
in
   Port = port(send:SecureSend)
   {Send Port HiddenKey}
end

Send = local Keys in
   Keys = {NewDictionary}
   proc {$ Port Message}
      Key == {Keys.get Port} in
      if Key == nil then {Keys.put Port Message}
      else {Port.send Message Key} end
   end
end
```

Figure 17: An implementation of secure ports using names and stateful dictionaries (described in Sec. 6.5.1 of the pensum book).

`SolveAll` returns a list of as many `true`s as there are possible derivations of a valid program from `Tokens`. If this list is empty, the list of tokens does not correspond to any valid program, and `ValidateSyntax` returns `false`. Otherwise, it returns `true`.

`ParseProgram` tests whether it is possible to parse from its input an expression enclosed between `{` and `Browse` on the left and `}` on the right, according to the grammar. It calls `ParseExpression` to actually parse the expression. `ParseExpression` nondeterministically tries to parse its input according to one of the rules specified by the grammar; for each rule, it calls a procedure dedicated to this rule. These procedures are shown in Fig. Question 6a; all of them are based on the principles of relational programming. (Question 6a did not require you to use the relational model.)

Question 6b (30%)

Figure 20 shows a valid program in `f` that computes and displays the number 1000! Observe that in `f` there is no way to define a named function: Neither `fun {Factorial N} ... end` nor `Factorial = fun {$ N} ... end` are syntactically valid. A function only becomes (locally) named if it is passed as an argument to another function. The way of programming shown in Fig. 20 is sometimes called the 'continuation-passing programming style'.

```
fun {ValidateSyntax Tokens}
   {SolveAll fun {$}
                {ParseProgram Tokens nil} true end}
   \= nil
end

proc {ParseProgram Tokens Rest}
   Expression in
   Tokens = '{'|ident('Browse')|Expression
   {ParseExpression Expression '}'|Rest}
end

proc {ParseExpression Tokens Rest}
   choice
      {ParseApplication Tokens Rest}
   [] {ParseFunction Tokens Rest}
   [] {ParseIf Tokens Rest}
   [] {ParseOperation Tokens Rest}
   [] {ParseIdentifier Tokens Rest}
   [] {ParseInteger Tokens Rest}
   end
end
```

Figure 18: An implementation of `ValidateSyntax` using relational programming.

```
proc {ParseApplication Tokens Rest}
   Expression Expressions in
   Tokens = '{'|Expression
   {ParseExpression Expression Expressions}
   {ParseExpressions Expressions '}'|Rest}
end

proc {ParseExpressions Tokens Rest}
   Expression Expressions in
   Tokens = Expression|Expressions
   choice
      {ParseExpression Expression nil}
      {ParseExpressions Expressions Rest}
   [] Rest = Expressions
   end
end

proc {ParseFunction Tokens Rest}
   Identifier Identifiers Expression in
   Tokens = 'fun'|'{'|'$'|Identifier
   {ParseIdentifier Identifier Identifiers}
   {ParseIdentifiers Identifiers '}'|Expression}
   {ParseExpression Expression 'end'|Rest}
end

proc {ParseIf Tokens Rest}
   Identifier Condition Consequent Alternative in
   Tokens = 'if'|Identifier
   {ParseIdentifier Identifier '=='|Condition}
   {ParseExpression Condition 'then'|Consequent}
   {ParseExpression Consequent 'else'|Alternative}
   {ParseExpression Alternative 'end'|Rest}
end

proc {ParseOperation Tokens Rest}
   Operator Expression in
   {ParseIdentifier Tokens Operator|Expression}
   Operator = choice '*' [] '-' end
   {ParseExpression Expression Rest}
end

proc {ParseIdentifiers Tokens Rest}
   Identifier Identifiers in
   choice
      Tokens = Identifier|Identifiers
      {ParseIdentifier [Identifier] nil}
      {ParseIdentifiers Identifiers Rest}
   [] Rest = Tokens
   end
end
```

Figure 19: Definitions supporting the implementation of a syntax validator given in Fig. 18.

```
proc {ParseIdentifier Tokens Rest}
   Tokens = ident(_)|Rest
end

proc {ParseInteger Tokens Rest}
   Tokens = int(_)|Rest
end
```

Figure 19 (contd): Definitions supporting the implementation of a syntax validator given in Fig. 18.

```
{Browse
  {fun {$ Function Argument}
     {Function Argument Function}
   end
   fun {$ N Factorial}
     if N==0 then 1
     else N*{Factorial N-1 Factorial}
     end
   end
   1000}}
```

Figure 20: An implementation of the computation of 1000! in `f`.