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

Exam in TDT4165 Programming Languages

(with solutions)
Wednesday, December 3., 2008

Prepared by: Wacek Kuśnierczyk

Reviewed by: Ole Edsberg

General Comments and Hints

Read the following general comments and the rest of the exam text **before** you begin to answer.

- Throughout the text, the acronym 'CTMCP' is used to refer to the pensum book (P. van Roy and S. Haridi: Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004).
- The exam is composed of 5 tasks, each task contains one or more questions. To achieve the maximal score, you need to correctly answer all questions in all tasks. If you skip or answer incorrectly a question, your score will be reduced correspondingly. Different tasks contribute differently to the total score; see each task for details.
- All code examples are given in Oz. Where you are asked to write code, the code should be in Oz as well.
- Your answers should be concise, without text that is irrelevant or does not contribute to the answer.
- Each answer should include a brief explanation; rather than simply 'yes', you should answer 'yes, because ...'. Correct answers with no justification will score less than answers with justifications. Incorrect answers with coherent explanations may still give you some points. You may also disagree with what is stated in CTMCP or with what was explained during the lectures, but you should give convincing arguments in such cases.

Task 1 (15 points)

This task focuses on object-oriented programming (OOP).

Subtask 1-1. Some programming languages are called ‘object-oriented programming languages’.

(a) What is object-oriented *programming*?

(b) What are the features of an object-oriented programming *language*?

(c) Objects in OOP are a specific kind of data structures. Characterize OOP objects with respect to these three distinctions: declarative–nondeclarative, bundled–unbundled, secure–insecure.

Subtask 1-2. Some programming languages support *multiple inheritance*, while others don’t.

(a) What is inheritance?

(b) What are the benefits of inheritance?

(c) What are the drawbacks of inheritance?

(d) What is multiple inheritance?

(e) What are the benefits of multiple inheritance?

(f) What are the drawbacks of multiple inheritance?

A counter is an object that stores a value (typically an integer) and exposes methods for increasing, decreasing, resetting, and/or displaying the value. Consider the following partial code, written using the OOP part of the Oz syntax.

```
class Counter
   attr value
   meth init(Value) ... end
   meth increase(Value) ... end
   meth value($) ... end
end
```

The method `init(Value)` initializes the attribute `value` with the value of `Value`. The method `increase(Value)` updates the value of the attribute `value` to the sum of its current value and the value of `Value`. The method `value($)` returns the current value of the attribute `value`.

Subtask 1-3. Complete the implementation according to the specifications above by replacing the ellipses (...) with missing code.

Subtask 1-4. Show how to implement counters as objects *without* using the OOP syntax of Oz. You may find helpful the following template:

```
Counter =
local
   Attributes = ...
   Methods = ...
   proc {Init Message State}
      init(Value) = ... in
      (State.value) := ...
   end
   proc {Increase ...} ... end
   proc {Value ...} ... end
in
   'class'(...)
end
```

Your implementation should work with the function New implemented as follows:

```
fun {New Class Init}
   State = {Record.make state Class.attributes}
   {Record.forAll State
    proc {$ Attribute}
       Attribute = {NewCell _}
    end}
   proc {Object Message}
      {Class.methods.{Label Message} Message State}
   end
in
   {Object Init}
   Object
end
```

(source file: code/new.oz)

The function Record.make takes as arguments an atom and a list and returns a record with the atom as its label and the items of the list as its fields. The fields have no values (i.e., they name unbound variables). The function Record.forAll takes as arguments a record and a unary procedure and applies the procedure to every value in the record:

```
R = {Record.make record [some fields]}
% R is record(some:_ fields:_)

{Record.forAll R proc {$ Value} Value = value end}
% R is record(some:value fields:value)
```

The example below illustrates instantiation of the class Counter; the code should work with both implementations:

```
C = {New Counter init(1)}
{C increase(3)}
{Browse {C value($)}}
% the browser displays 4
```

Solutions

Subtask 1-1. (a) According to CTMCP, object-oriented programming is a programming paradigm in which programs are collections of interacting data abstractions. These data abstractions are called 'objects' and are bundled, typically stateful (non-declarative), and possibly secure.

(b) An object-oriented programming language is a language that has syntactic and semantic features enabling easy development of programs in the OOP style.

(c) Objects are typically non-declarative, but this is not an essential feature of OOP programming. For example, in many programming languages (e.g., Java, Python) strings are immutable objects. Objects are by definition bundled—the encapsulate both data and methods for accessing the data. Objects can be secure, but this is not an essential feature of OOP programming. In most OOP languages you can implement objects so that the encapsulated data is directly available (no method calls are needed to access it), while in some (e.g., Python) all data members of an object are accessible directly (Python objects are insecure by the language design).

Subtask 1-2. (a) Inheritance is an OOP feature which allows the definition of new classes (types) by subclassing previously defined ones, which are then called their 'ancestors' or 'superclasses'. The new classes inherit from the ancestors their members—attributes and methods—which thus do not have to be defined anew.

(b) The major benefit of using inheritance is increase in modularity and reuse of code. Operations common to objects of different classes that have shared ancestors can be coded just once. Inheritance is also essential for subtype polymorphism, a feature that enables the definition of polymorphic procedures, procedures that can take as arguments objects of different, though related classes.

(c) Using inheritance can lead to problems such as breaking a class invariant (e.g., when a successor class incorectly reimplements some functionality of an ancestor class) or breaking security barriers (e.g., when a successor class provides, intentionally or not, aditional access to members hidden in an ancestor class).

(d) Multiple inheritance (MI) is an inheritance scheme in which a class can be defined as a direct successor (a child) of more than one ancestor (parent) class.

(e) Multiple inheritance allows one to create classes that inherit from more than one parent, and thus may increase code reuse and enhance subtype polymorphism.

(f) Multiple inheritance may lead to problems with class member resolution, e.g., when more than one parent provides a method with the same signature (name and argument types). Some languages that allow MI provide a recovery scheme for such situations, while in others the compiler or interpreter will complain about call ambiguity.

Subtask 1-3. The class `Counter` can be implemented using the OOP syntax as follows:

```
class Counter
   attr value
   meth init(Value) @value = Value end
   meth increase(Value) value := @value + Value end
   meth value($) @value end
end
```

(source file: `code/counter-class.oz`)

Subtask 1-4. The class `Counter` can be implemented without using the OOP syntax as follows:

```
Counter =
local
   Attributes = [value]
   Methods = methods(init:Init increase:Increase value:Value)
   proc {Init init(Value) State}
      (State.value) := Value
   end
   proc {Increase increase(Value) State}
      (State.value) := @(State.value) + Value
   end
   proc {Value value(Value) State}
      Value = @(State.value)
   end
in
   'class'(attributes:Attributes methods:Methods)
end
```

(source file: `code/counter.oz`)

You can test the implementation by running `code/test-counter.oz`.

Task 2 (10 points)

Subtask 2-1. In Oz, as in many other programming languages, it is possible to execute programs in a *lazy* computation model.

(a) What does the term ‘lazy computation’ mean?

(b) What are the benefits of using lazy computation?

(c) What are the drawbacks of using lazy computation?

Subtask 2-2. Show how to build an infinite list (a stream) of *all* positive integers. You can find the following template helpful:

```
Integers =
local
   <...>
in
   {<..>}
end
```

You should be able to retrieve the first n integers for any possible n (at least in theory):

```
{List.take Integers 3}
% returns [1 2 3]

{List.take Integers 10}
% returns [1 2 3 4 5 6 7 8 9 10]

{List.take Integers 1000000}
% returns a list of the first 1 million integers
```

Solutions

Subtask 2-1. (a) Lazy computation is a computation scheme in which the evaluation of an expression or the execution of a statement can be postponed until it is actually needed.

(b) The major benefit of using the lazy computation model is the ability to postpone costly operations until they have to be done. Operations that do not have to be done are never executed. Besides, lazy computation provides a means for defining infinite data structures, such as infinite lists (streams).

(c) Lazy computation was originally developed in the context of declarative computing, where the result of a call to a function with particular arguments is always the same, irrespectively of when the call is actually executed. In a non-declarative context, lazy computation can lead to unexpected results; the model has to be implemented and used with care.

Subtask 2-2. The infinite list `Integers` can be implemented as follows:

```
Integers =
local
   fun lazy {Generate From}
      From|{Generate From+1}
   end
in
   {Generate 1}
end
```

(source file: `code/integers.oz`)

You can test the implementation by running `code/test-integers.oz`.

Task 3 (25 points)

According to Wikipedia,[1] "***dc** is a reverse-polish desk calculator which supports unlimited precision arithmetics. It is one of the oldest Unix utilities, predating even the invention of the C programming language.*" Here we will

[1] http://en.wikipedia.org/wiki/Dc_(Unix) .

consider only a small subset of dc, which we shall call 'μdc' ('micro dc').[2] You will formally define the syntax and implement an interpreter for this tiny language.

The purpose of μdc is to perform arithmetic operations on integers, and to display their results. μdc is a *stack-based* language: input values (integers) as well as intermediate results are kept on a stack; arithmetic operators cause values to be *popped* from the stack, and the results of the calculations are *pushed* back onto the stack. Additional commands are used to inspect the content of the stack. μdc operates on arbitrary integers,[3] and supports four arithmetic operations: addition, subtraction, multiplication, and (integer) division. In addition, it has three commands: p (for 'peek'), s (for 'state' or 'stack'), and r (for 'reset') that allow to print the top element or all elements from the stack, or clear the stack, respectively.

Consider the following examples, executed on a hypothetical interpreter:[4]

```
# push 1 and 2 onto the stack, add them, print the result
mdc> 1 2 + p
3

# push 1, reset, push 2 and 3, print the whole content
mdc> 1 r 2 3 s
3
2

# subtract 2 from 8, print the result, divide the result by 3,
# push 4 onto the stack and print the whole content
mdc> 8 2 - p 3 / 4 s
6
4
2
```

Note the order in which arguments are passed to the arithmetic operators: the code `8 2 -` results in 2 being subtracted from 8, not 8 from 2; the code `6 3 /` results in 6 being divided by 3, not 3 by 6, etc. Note also the order of values in the output: the code `1 r 2 3 s` results in 3 and then 2 being printed, in this and not the inverse order (so that the first value printed by s corresponds to the top of the stack, and the last value printed by s coresponds to the bottom of the stack).

Subtask 3-1. Use Backus-Naur Form (BNF) or Extended BNF (EBNF) to write a grammar that formally specifies the syntax of μdc. You can use the following template:

$$\begin{array}{rcl} \langle \textit{program} \rangle & ::= & \{\ \langle \textit{instruction} \rangle\ \} \\ \langle \textit{instruction} \rangle & ::= & \ldots \\ \ldots & ::= & \ldots \end{array}$$

Assume that only non-negative integers are allowed. You are free to use regular expressions or plain text comments where you can't enumerate all possible lexemes.

Subtask 3-2. Examine the following inputs. For each of them show, using your grammar, whether it is or is not a valid program in μdc. Justify your answers.

(a) `""`

(b) `"1 2 +"`

(c) `"1 p 1 p + r +"`

(d) `"123 + p s"`

(e) `"p / p 1 2"`

[2]`mdc` was used as a motivating example during the first few lectures.

[3]In practice, constraints on the range of integers will depend on the underlying implementation language, the operating system, the hardware, etc.; we shall ignore this issue here.

[4]Lines beginning with a '`#`' are comments, and are not part of the input or output; lines beginning with a '`mdc>`' show the input; lines with no prefix show the output.

Subtask 3-3. Read the following specification, and implement an interpreter for μdc.

- Programs are passed to the interpreter already parsed, in the form of a flat list of tokens; you do *not* implement the parser. The interpreter iteratively consumes a token at a time, performing an appropriate action, and stopping when no more tokens are available.
- Each token is an Oz record containing one value. Tokens with the record label '`int`' contain an integer; tokens with the label '`op`' contain a value that makes the interpreter perform one of the four arithmetic operations; tokens with the label '`cmd`' contain a value that makes the interpreter apply one of the two commands.
- Internally, the interpreter uses a stack for storing values. When the token consumed by the interpreter is an '`int`' record, the integer value is pushed onto the stack. When the consumed token is an '`op`' record, two values are popped from the stack, the corresponding arithmetic operation is applied to them in the appropriate order (see above), and the result is pushed onto the stack. When the consumed token is a '`cmd`' record, the appropriate stack-printing operation is executed, and the stack is left unchanged.

You should use the following template; fill in the '`<...>`' parts:

```
% the interpreter procedure
proc {MicroDC Program}
   <...>
in
   <...>
end
```

Your code must *not* exceed 50 lines, excluding comments. Below is an annotated example of application of the interpreter to a simple program:

```
% interpretation of a simple program
% the definitions of Add, Div, and S are not shown
Program = [int(6) int(4) int(2) cmd(S) op(Add) op(Div) cmd(S)]
{MicroDC Program}
% the browser displays, in separate lines, from top to bottom:
% 2, 4, 6, and 1
```

Additional requirements and comments:

- You can implement the internal stack in any way you like; a flat list is good enough for the purpose.
- You are free to decide whether your implementation should take a pure functional approach, or use explicit mutable state.
- The values included in `op` and `cmd` records (e.g., captured in the variables `Add` and `S` above) can be atoms (e.g., `'+'`, `plus`, `s`, `stack`, etc.), functions and procedures (e.g., `fun {Add N M} ... end` or `proc {S Stack} ... end`), or whatever else you find convenient.

Solutions

Subtask 3-1. Using EBNF and Perl-style regular expressions with POSIX classes, the grammar can be written as follows:

$$\begin{array}{rcl}
\langle \textit{program} \rangle & ::= & \{\ \langle \textit{instruction} \rangle\ \} \\
\langle \textit{instruction} \rangle & ::= & \langle \textit{integer} \rangle \mid \langle \textit{operator} \rangle \mid \langle \textit{command} \rangle \\
\langle \textit{integer} \rangle & ::= & \text{regex}(\texttt{[:digit:]+}) \\
\langle \textit{operator} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\langle \textit{command} \rangle & ::= & \texttt{p} \mid \texttt{s} \mid \texttt{r}
\end{array}$$

Here, ⟨*program*⟩ is a (possibly empty) list of tokens. ⟨*tokens*⟩ is defined as a token optionally followed by a list of tokens. ⟨*integer*⟩ is defined as one or more digits (here, 'regex(...)' denotes a regular expression).

Subtask 3-2. According to the grammar above, all four sequences are syntactically valid; the answer may differ for your grammar. Note that the syntactic validity of input sequences has nothing to do with their semantics; all the examples above are syntactically valid, even though a dc interpreter would complain during an evaluation of some of them.

Subtask 3-3. In a pure functional approach,[5] μdc can be implemented as follows:

```
proc {MicroDC Program}
   Ops = ops('+':Number.'+' '-':Number.'-' '*':Number.'*' '/':Int.'div')
   Cmds = cmds('p':fun {$ Stack} {Browse Stack.1} Stack end
               's':fun {$ Stack} {ForAll Stack Browse} Stack end
               'r':fun {$ _} nil end)
   proc {Iterate Tokens Stack}
      case Tokens
      of int(Int)|Tokens then
         {Iterate Tokens Int|Stack}
      [] op(Op)|Tokens then
         Int1|Int2|Rest = Stack in
         {Iterate Tokens {Ops.Op Int2 Int1}|Rest}
      [] cmd(Cmd)|Tokens then
         UpdatedStack = {Cmds.Cmd Stack} in
         {Iterate Tokens UpdatedStack}
      else skip end
   end
in {Iterate Program nil}
end
```

(source file: `code/microdc-stateless.oz`)

A stateful μdc can be implemented as follows:

```
proc {MicroDC Program}
   Ops = ops('+':Number.'+' '-':Number.'-' '*':Number.'*' '/':Int.'div')
   Cmds = cmds('p':proc {$ Stack} {Browse (@Stack).1} end
               's':proc {$ Stack} {ForAll @Stack Browse} end
               'r':proc {$ Stack} Stack := nil end)
   Stack = {NewCell nil}
   proc {Iterate Tokens}
      case Tokens
      of int(Int)|Tokens then
         Stack := Int|@Stack
         {Iterate Tokens}
      [] op(Op)|Tokens then
         Int1|Int2|Rest = @Stack in
         Stack := {Ops.Op Int2 Int1}|Rest
         {Iterate Tokens}
      [] cmd(Cmd)|Tokens then
         {Cmds.Cmd Stack}
         {Iterate Tokens}
      else skip end
   end
in {Iterate Program}
end
```

(source file: `code/microdc-stateful.oz`)

[5] If not counting the use of `Browse`.

You can test the implementation by running `code/test-microdc.oz`.

Task 4 (25 points)

A *circular list* (CL) is a list in which the last element is followed by the first one; effectively, the list is infinite. (Effectively, it is nonsensical to speak of the last element on the list; what is meant by 'last' is the last element in the underlying implementation, which typically is a plain list.) In this task you will define the abstract data type (ADT) for *one-way* circular lists—circular lists that can be traversed in only one direction. You also need to provide two *bundled secure* implementations: one *declarative*, and one *non-declarative*.

Subtask 4-1. Explain the terms 'declarative', 'bundled', and 'secure' in the context of data abstraction:

(a) What is a declarative data structure?

(b) What is a bundled data structure?

(c) What is a secure data structure?

Subtask 4-2. What is *observational declarativeness*?

A circular list contains zero, one, or more elements; the first of them (if there are any) is the *head* of the list. Internally, the content of a circular list can be represented in many ways; here, we will use linked lists (i.e., plain lists in Oz). When a circular list is accessed, the last element in the underlying list is followed, recursively, by its first element, as if they were linked. For example, a circular list might have `[1 2 3]` as the underlying representation. Then, conceptually, the circular list is `[1 2 3 1 2 3 1 ...]`, its head is `1`, and its tail is `[2 3 1 2 3 1 2 ...]` (another circular list).

Using the notation of Sec. 3.7 in CTMCP,[6] we can specify the interface of a *non-declarative* implementation as follows:

- create a new empty list:
  $\langle$`fun {New}`: $\langle$CL T$\rangle\rangle$
- check whether a list is empty:
  $\langle$`fun {Empty` $\langle$CL T$\rangle$`}`: Bool$\rangle$
- get the head of a list:
  $\langle$`fun {Head` $\langle$CL T$\rangle$`}`: T$\rangle$
- get the tail of a list:
  $\langle$`fun {Tail` $\langle$CL T$\rangle$`}`: $\langle$CL T$\rangle\rangle$
- insert an element at the head of a list:
  $\langle$`proc {Insert` $\langle$CL T$\rangle$ T`}`$\rangle$
- drop the element at the head of a list:
  $\langle$`proc {Drop` $\langle$CL T$\rangle$`}`$\rangle$
- rotate a list, i.e., set its next-to-head element as the head:
  $\langle$`proc {Rotate` $\langle$CL T$\rangle$`}`$\rangle$

We make the following assumptions:

- The tail of a circular list is another circular list; the tail of the circular list `[1 2 3 1 2 3 ...]` is the circular list `[2 3 1 2 3 1 ...]`.

[6] Here, T denotes some type of values, and $\langle$CL T$\rangle$ denotes a circular list of elements whose type is T. In Oz, there is no explicit typing and T will effectively mean that the values can be of any type. Int denotes the type of integers.

- Inserting or dropping an element into or from a circular list will effectively insert or drop it recursively; inserting `0` into the circular list `[1 2 3 1 2 3 ...]` will modify it to `[0 1 2 3 0 1 2 3 ...]`, while dropping `1` from the circular list `[1 2 3 1 2 3 ...]` will modify it to `[2 3 2 3 ...]`.

Subtask 4-3. Show which parts of the above interface specification need to be modified for the case of an implementation that is *observationally declarative*.

Subtask 4-4. Compare the declarative and the non-declarative versions of circular lists.

(a) What are the benefits of using an observationally declarative circular list?

(b) What are the benefits of using a non-declarative circular list?

Subtask 4-5. Provide a non-declarative implementation of the CL ADT. You may, though you are not obliged to, use the following template:

```
class CL
   attr ...
   meth ... end
   ...
end
```

Below is an annotated example of how your implementation should work:

```
% create a new circular list:
CL1 = {New CL init}     % the internal representation is nil

% try to display its head (there is none!)
try {Browse {CL1 head($)}}
catch Exception then {Browse Exception} end % displays an exception

% insert three elements, display the head
{CL1 insert(1)}         % the internal representation is [1]
{CL1 insert(2)}         % the internal representation is [2 1]
{CL1 insert(3)}         % the internal representation is [3 2 1]
{Browse {CL1 head($)}}  % displays 3

% get the tail, rotate it, drop an element, rotate, display the head
CL2 = {CL1 tail($)}     % the internal representation is [2 1 3]
{CL2 rotate}            % the internal representation is [1 3 2]
{CL2 drop}              % the internal representation is [3 2]
{CL2 rotate}            % the internal representation is [2 3]
{Browse {CL2 head($)}}  % displays 2
```

Note: `{CL1 tail($)}` returns a new circular list. You may consider implementing an additional initializer used internally in a call to `tail($)` (but other solutions are possible).

Subtask 4-6. Provide an observationally declarative implementation of the CL ADT. Below is an annotated example of how your implementation should work:

```
CL1 = {New CL init}     % the internal representation is nil
CL2 = {CL1 insert(1)}   % internally, CL1 is nil, CL2 is [1]
CL3 = {CL2 insert(2)}   % internally, CL3 is [2 1]
CL4 = {CL3 rotate}      % internally, CL4 is [1 2]
{Browse {CL4 head($)}}  % displays 1
```

Solutions

Subtask 4-1. (a) According to CTMCP, a declarative data structure is one that cannot change its state, i.e., its components are fixed and immutable. (Unbound dataflow variables are not considered mutable variables.)

(b) According to CTMCP, a bundled data structure is one that contains both data and procedures that can access the data.

(c) According to CTMCP, a secure data structure is one such that the data it contains can be accessed only with the use of dedicated procedures, which may be bundled or not within the structure.

Subtask 4-2. According to CTMCP, observational declarativeness is a property of a component that "behaves declaratively, i.e., as if it were independent, stateless, and deterministic, without necessarily being written in a declarative computation model." This means that an observably declarative component may in fact be implemented using mutable state (for example, for the purpose of caching the results of costly computations, as in the case of memoized functions), but the mutability is not observable through the component's interface.

Subtask 4-3. The following modifications are needed for the declarative case:

- insert an element at the head of a list:
  ⟨`fun {Insert` ⟨`CL T`⟩ `T}:` ⟨`CL T`⟩⟩
- drop the element at the head of a list:
  ⟨`fun {Drop` ⟨`CL T`⟩`}:` ⟨`CL T`⟩⟩
- rotate a list, i.e., set its next-to-head element as the head:
  ⟨`fun {Rotate` ⟨`CL T`⟩`}:` ⟨`CL T`⟩⟩

Subtask 4-4. (a) A declarative circular list has fixed content, which is the same on any occasion the list is accessed. This makes writing programs using such lists slightly awkward, but greatly simplifies their analysis.

(b) A non-declarative circular list has a mutable content, and thus it is not necessary to create a new object each time the content has to be modifed. This makes writing programs using such lists much easier, but complicates their analysis.

Subtask 4-5. Non-declarative circular lists can be implemented as follows:

```
class CL
   attr state
   meth init @state = nil end
   meth Init(State) @state = State end
   meth empty($) @state == nil end
   meth head($) (@state).1 end
   meth tail($) {New CL Init({Append (@state).2 [(@state).1]})} end
   meth insert(Item) state := Item|@state end
   meth drop state := (@state).2 end
   meth rotate state := {Append (@state).2 [(@state).1]} end
end
```

(source file: `code/cl-nondeclarative.oz`)

Subtask 4-6. Declarative circular lists can be implemented as follows:

```
class CL
   attr state
   meth init @state = nil end
   meth Init(State) @state = State end
   meth empty($) @state == nil end
   meth head($) (@state).1 end
   meth tail($) {New CL Init({Append (@state).2 [(@state).1]})} end
```

```
      meth insert(Item $) {New CL Init(Item|@state)} end
      meth drop($) {New CL Init((@state).2)} end
      meth rotate($) {New CL Init({Append (@state).2 [(@state).1]})} end
   end
```

(source file: `code/cl-declarative.oz`)

You can test both implementations by running `code/test-cl.oz`.

Task 5 (15 points)

Consider the following program:

```
local X P Q in
   X = 1
   P = proc {$} {Browse X} end
   Q = proc {$ X} local X in X = 2 {P} end end
   {Q X}
end
```

Subtask 5-1. Show the state of abstract machine during an execution of the program just before the statement `{Q X}` is executed. (Ignore the identifier `Browse` when you show the content of environments.)

Subtask 5-2. Provide formal semantics for the procedure application statement. The semantic statement is:

$(\{\langle id\rangle_0\ \langle id\rangle_1 \ldots \langle id\rangle_n\}, E)$

where E is an environment.

Subtask 5-3. Using the definition from **Subtask 5-2**, show how the execution of the above program proceeds from the state discussed in **Subtask 5-1**. At each step the single assignment store should contain only those variables that are reachable from the code to be executed. What will be printed in the browser window?

Subtask 5-4. Oz is a lexically scoped language. Explain the terms *lexical scoping* and *dynamic scoping*.

Subtask 5-5. Pretending that Oz is dynamically scoped, suggest an appropriate formal semantics for procedure application statements. (It suffices to show how this semantics differs from the one in **Subtask 5-2**.)

Subtask 5-6. Using the definition from **Subtask 5-5**, show how the execution of the above program proceeds from the state discussed in **Subtask 5-1**. At each step the single assignment store should contain only those variables that are reachable from the code to be executed. (It suffices to show how the execution will differ from the one in **Subtask 5-3**.) What will be printed in the browser window?

Solutions

Subtask 5-1. The state of the abstract machine during an execution of the above program (translated into the kernel language) just before the statement `{Q X}` is executed is as follows:

([(`{Q X}`, {`Q` $\to v_3$, `X` $\to v_1$})],
{ $v_1 = 1$, $v_2 =$ (`proc {$} {Browse X} end`, {`X` $\to v_1$}),
$v_3 =$ (`proc {$ X} local X in X = 2 {P} end end`, {`P` $\to v_2$}) })

where v_i are variables in the single assignment store.

Subtask 5-2. The semantic statement is:

$(\{\langle id\rangle_0\ \langle id\rangle_1 \ldots \langle id\rangle_n\}, \mathrm{E})$

The rule of execution is:

- If $\langle id\rangle_0$ is not declared in E, raise an error.
- Otherwise, if $\mathrm{E}(\langle id\rangle_0)$ is unbound, suspend the execution.
- Otherwise, if $\mathrm{E}(\langle id\rangle_0)$ is a closure of the form

  $(\texttt{proc \{\$}\ \langle id\rangle'_1 \ldots \langle id\rangle'_n\texttt{\}}\ \langle statement\rangle\ \texttt{end}, \mathrm{CE})$

  where CE is a closure environment, push onto the stack the semantic statement

  $(\langle statement\rangle, \mathrm{CE}')$

  where $\mathrm{CE}' = \mathrm{CE} + \{\langle id\rangle'_1 \to \mathrm{E}(\langle id\rangle_1), \ldots, \langle id\rangle'_n \to \mathrm{E}(\langle id\rangle_n)\}$
- Otherwise, raise an error.

Subtask 5-3. The execution of the above program, starting at the statement `{Q X}`, is as follows:

(a) $([(\texttt{\{Q X\}}, \{\texttt{X} \to v_1, \texttt{P} \to v_2, \texttt{Q} \to v_3\})],$
$\{ v_1 = 1, v_2 = (\texttt{proc \{\$\} \{Browse X\} end}, \{\texttt{X} \to v_1\}),$
$v_3 = (\texttt{proc \{\$ X\} local X in X = 2 \{P\} end end}, \{\texttt{P} \to v_2\}) \})$

(b) $([(\texttt{local X in X = 2 \{P\} end}, \{\texttt{X} \to v_1, \texttt{P} \to v_2\})],$
$\{ v_1 = 1, v_2 = (\texttt{proc \{\$\} \{Browse X\} end}, \{\texttt{X} \to v_1\}) \})$

(c) $([(\texttt{X = 2 \{P\}}, \{\texttt{X} \to v_4, \texttt{P} \to v_2\})],$
$\{v_1 = 1, v_2 = (\texttt{proc \{\$\} \{Browse X\} end}, \{\texttt{X} \to v_1\}), v_4\})$

(d) $([(\texttt{X = 2}, \{\texttt{X} \to v_4, \texttt{P} \to v_2\}), (\texttt{\{P\}}, \{\texttt{X} \to v_4, \texttt{P} \to v_2\})],$
$\{v_1 = 1, v_2 = (\texttt{proc \{\$\} \{Browse X\} end}, \{\texttt{X} \to v_1\}), v_4\})$

(e) $([(\texttt{\{P\}}, \{\texttt{X} \to v_4, \texttt{P} \to v_2\})],$
$\{v_1 = 1, v_2 = (\texttt{proc \{\$\} \{Browse X\} end}, \{\texttt{X} \to v_1\}), v_4 = 2\})$

(f) $([(\texttt{\{Browse X\}}, \{\texttt{X} \to v_1\})],$
$\{v_1 = 1\})$

(g) $([\,],$
$\{\,\})$

In the browser window, 1 will be printed.

Subtask 5-4. Lexical and dynamic scoping are two different approaches to resolving the value of a variable (or, as in Oz, to find which variable an identifier is mapped to) in a procedure call.

In lexical scoping, variables are looked-up using the closure environment of the closure object, which is an extension of the environment present at the time the closure was defined.

In dynamic scoping, variables are looked-up using the current evaluation environment, which is an extension of the environment in which the call was made.

Subtask 5-5. The semantic statement is:

$(\{\langle id\rangle_0\ \langle id\rangle_1 \ldots \langle id\rangle_n\}, \mathrm{E})$

The rule of execution is:

- If $\langle id\rangle_0$ is not declared in E, raise an error.
- Otherwise, if $\mathrm{E}(\langle id\rangle_0)$ is unbound, suspend the execution.
- Otherwise, if $\mathrm{E}(\langle id\rangle_0)$ is a procedure object of the form

  $\texttt{proc \{\$}\ \langle id\rangle'_1 \ldots \langle id\rangle'_n\texttt{\}}\ \langle statement\rangle\ \texttt{end}$

  push onto the stack the semantic statement

  $(\langle statement\rangle, \mathrm{E}')$

where $E' = E + \{\langle id\rangle'_1 \rightarrow E(\langle id\rangle_1), \ldots, \langle id\rangle'_n \rightarrow E(\langle id\rangle_n)\}$

- Otherwise, raise an error.

Note that the closure environment is no longer used when a procedure is called, and can thus be removed from the closure. In fact, closures are no longer needed, since procedure objects without an accompanying environment are sufficient for computing with dynamic scoping. (Retaining closure environments in your definition was not considered an error.)

Subtask 5-6. The execution of the above program, starting at the statement `{Q X}`, is as follows:

(a) ([(`{Q X}`, $\{X \rightarrow v_1, P \rightarrow v_2, Q \rightarrow v_3\}$)],
  $\{ v_1 = 1, v_2 =$ `proc {$} {Browse X} end`,
  $v_3 =$ `proc {$ X} local X in X = 2 {P} end end` $\}$)

(b) ([(`local X in X = 2 {P} end`, $\{X \rightarrow v_1, P \rightarrow v_2, Q \rightarrow v_3\}$)],
  $\{ v_1 = 1, v_2 =$ `proc {$} {Browse X} end`,
  $v_3 =$ `proc {$ X} local X in X = 2 {P} end end` $\}$)

(c) ([(`X = 2 {P}`, $\{X \rightarrow v_4, P \rightarrow v_2, Q \rightarrow v_3\}$)],
  $\{ v_2 =$ `proc {$} {Browse X} end`,
  $v_3 =$ `proc {$ X} local X in X = 2 {P} end end`, v_4 $\}$)

(d) ([(`X = 2`, $\{X \rightarrow v_4, P \rightarrow v_2, Q \rightarrow v_3\}$), (`{P}`, $\{X \rightarrow v_4, P \rightarrow v_2, Q \rightarrow v_3\}$)],
  $\{ v_2 =$ `proc {$} {Browse X} end`,
  $v_3 =$ `proc {$ X} local X in X = 2 {P} end end`, v_4 $\}$)

(e) ([(`{P}`, $\{X \rightarrow v_4, P \rightarrow v_2, Q \rightarrow v_3\}$)],
  $\{ v_2 =$ `proc {$} {Browse X} end`,
  $v_3 =$ `proc {$ X} local X in X = 2 {P} end end`, $v_4 = 2$ $\}$)

(f) ([(`{Browse X}`, $\{X \rightarrow v_4, P \rightarrow v_2, Q \rightarrow v_3\}$)],
  $\{ v_2 =$ `proc {$} {Browse X} end`,
  $v_3 =$ `proc {$ X} local X in X = 2 {P} end end`, $v_4 = 2$ $\}$)

(g) ([ ],
  { })

In the browser window, 2 will be printed.

Note that given the semantics for dynamically scoped procedure application given above, environments preserve mappings even if they are no longer needed. (You may have defined a different semantics.)

Task 6 (10 points)

Consider the following procedure that takes as an argument a cell and is supposed to update its content by increasing it by 1:

```
proc {Increase Cell} Old in
   {Exchange Cell Old Old+1}
end
```

Subtask 6-1. Unfortunately, `Increase` does not work as expected. When the following code is executed, no output appears in the browser window:

```
local C = {NewCell 0} in
   {Increase C}
   {Browse @C}
end
```

Explain why there will be no output printed. Hint: translate the definition of `Increase` to the kernel language.

Subtask 6-2. Provide a reimplementation of `Increase` (call it `PlusPlus`) that fixes the problem, so that if the new version were used in the code above, 1 would be printed.

Subtask 6-3. Consider the following program:

```
local C = {NewCell 0} in
   thread {PlusPlus C} end
   thread {PlusPlus C} end
   {Browse @C}
end
```

What are the values that could be printed in the browser window during an execution of the above program? Use your definition of `PlusPlus` to justify the answer.

Solutions

Subtask 6-1. The definition of `Increase` translates to the (semi-)kernel language as follows:

```
Increase =
proc {$ Cell}
   local Old New in
      New = Old + 1
      {Exchange Cell Old New}
   end
end
```

Since `Old` is unbound at the time `New = Old + 1` is executed, a call to `Increase` will freeze the calling thread.

Subtask 6-2. Here is one way to implement `PlusPlus`:

```
proc {PlusPlus Cell} Old in
   {Exchange Cell Old thread Old+1 end} end
```

(source file: `code/plusplus.oz`)

Subtask 6-3. `Exchange` performs an atomic swap. Irrespectively of the statement `New = Old + 1` running in a separate thread in each call to `PlusPlus`, the content of `C` is first swapped from 0 to 1 (by whichever of the two threads), and then from 1 to 2 (by the other of the two threads). At the end of the program, the content of `C` will always be 2.

However, depending on the scheduler, the `Browse` statement may be executed before the cell is updated for the first time, between the two updates, or after the two updates. Thus, different executions of the program may result in 0, 1, or 2 printed in the browser window.