Norwegian University of Science and Technology
Department of Computer and Information Science

Contact during exam:
Lars Bungum (92046135)

This exam set was approved by Øystein Nytrø

Programming Languages (TDT4165)

December 10th 2013
Tid: 15:00 – 19:00

Exam aids:
Exam aid code C. No (hand)written aids allowed. Only specified, simple calculators.

Short answers are preferred. If find assumptions necessary to be made to give your answer, state these assumptions before you go ahead. All implementations should be done in Oz. Good luck!

Part 1: Computational Models and Programming Language Fundamentals (30%)

Problem 1 Accordig to the textbook Concepts, Techniques, and Models of Computer Programming (CTMCP), programming encompasses the elements

- a model of computation
- a set of programming techniques and design principles
- a set of reasoning techniques

a) What is a computational model?

A computational model is a formal system that defines a language and how the sentences of the language are executed by the abstract machine. This means that both the syntax and the semantics of the language are defined by the computational model.

b) What does it mean that a computational model is declarative? Is the real world declarative? A computational model as explained above, is declarative, if it defines a language in such a way that the programming is done declaratively.

Declarative programs are *compositional*, the components can be tested and proved to correct independent of eachother. It is therefore easier to reason about them.

A declarative operation is independent, stateless and deterministic.

c) Explain how the syntax of a programming language can be defined.

The syntax can be defined by means of a formal grammar, specifying valid expressions in the language, programs that can be executed.

d) What does a 'kernel language approach' mean? What other translation approaches are there to define the semantics of a programming language?

A kernel language approach means that a practical programming language is translated into a kernel language. This language should be easy to reason in. A translation scheme from the practical language into the kernel language is needed.

Other translation schemes are translation into a foundational calculus or an abstract machine (p. 41).

e) The kernel language syntax is given in Figure 1

⟨s⟩ ::=

1 **skip**
2 | ⟨s⟩$_1$ ⟨s⟩$_2$
3 | **local** ⟨x⟩ **in** ⟨s⟩ **end**
4 | ⟨x⟩$_1$=⟨x⟩$_2$
5 | ⟨x⟩=⟨v⟩
6 | **if** ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**
7 | **case** ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**
8 | {⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$}

Figure 1: The declarative kernel language

In this figure there are 8 lines, and the function in each element is (in random order):

a Variable-variable binding

b Empty statement

c Value creation

d Variable creation

e Pattern matching

f Conditional

g Statement sequence

h Procedure application

make the correct mapping between the line from the kernel language syntax and the letters describing the functions.

Correct mapping given in Figure 2. In tabular form:

1. b
2. g
3. d
4. a
5. c
6. f
7. e
8. h

⟨s⟩ ::=	
skip	Empty statement
⟨s⟩$_1$ ⟨s⟩$_2$	Statement sequence
local ⟨x⟩ **in** ⟨s⟩ **end**	Variable creation
⟨x⟩$_1$=⟨x⟩$_2$	Variable-variable binding
⟨x⟩=⟨v⟩	Value creation
if ⟨x⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Conditional
case ⟨x⟩ **of** ⟨pattern⟩ **then** ⟨s⟩$_1$ **else** ⟨s⟩$_2$ **end**	Pattern matching
{⟨x⟩ ⟨y⟩$_1$ ⋯ ⟨y⟩$_n$}	Procedure application

Table 2.1: The declarative kernel language.

158

Figure 2: The declarative kernel language

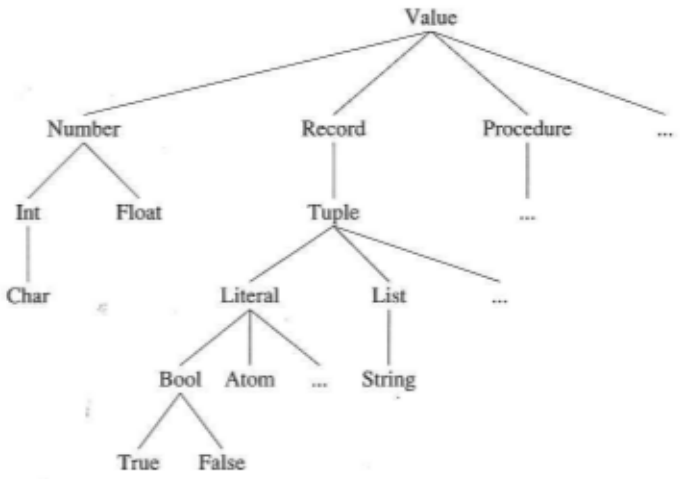

Figure 3: The type hierarchy of the declarative model

f) The type hierarchy of the declarative model in CTMCP is depicted in Figure 3

Explain the difference between static and dynamic typing. Which type is used in the declarative model in Oz?

In static typing all variable types are known at compile time. In dynamic typing, the variable type is known only when the variable is bound. The declarative model in Oz is dynamically typed.

g) Explain the relationship between tuples, lists and records in Oz. Illustrate your answer with code examples.

A record is a data structure consisting of a label followed by a set of pairs of features and variable identifiers. Features can be atoms, integers or booleans.

As is seen in Figure 3, lists and tuples are special cases of records. A record can look like

```
car(windows:3 doors:4)
```

A tuple is a record whose features are consecutive integeres starting from 1, for example

```
car(1:X1 2:X)
car(X1 X2)
```

A list is either the atom nil or the tuple

```
'|'(H T)
```

where T is either unbound or bound to a list.

This means that the same list can be expressed in three ways.

```
L1=[X Y]
L2=X|Y|nil
L3='|'(X '|'(Y nil))
```

(records with the vertical bar as label).

h) What is the abstract machine? Sketch the components of the abstract machine for the declarative sequential model, and explain their function.

The abstract machine for the declarative model consists of a stack of semantic operations and a single-assigment store as illustrated in Figure 4

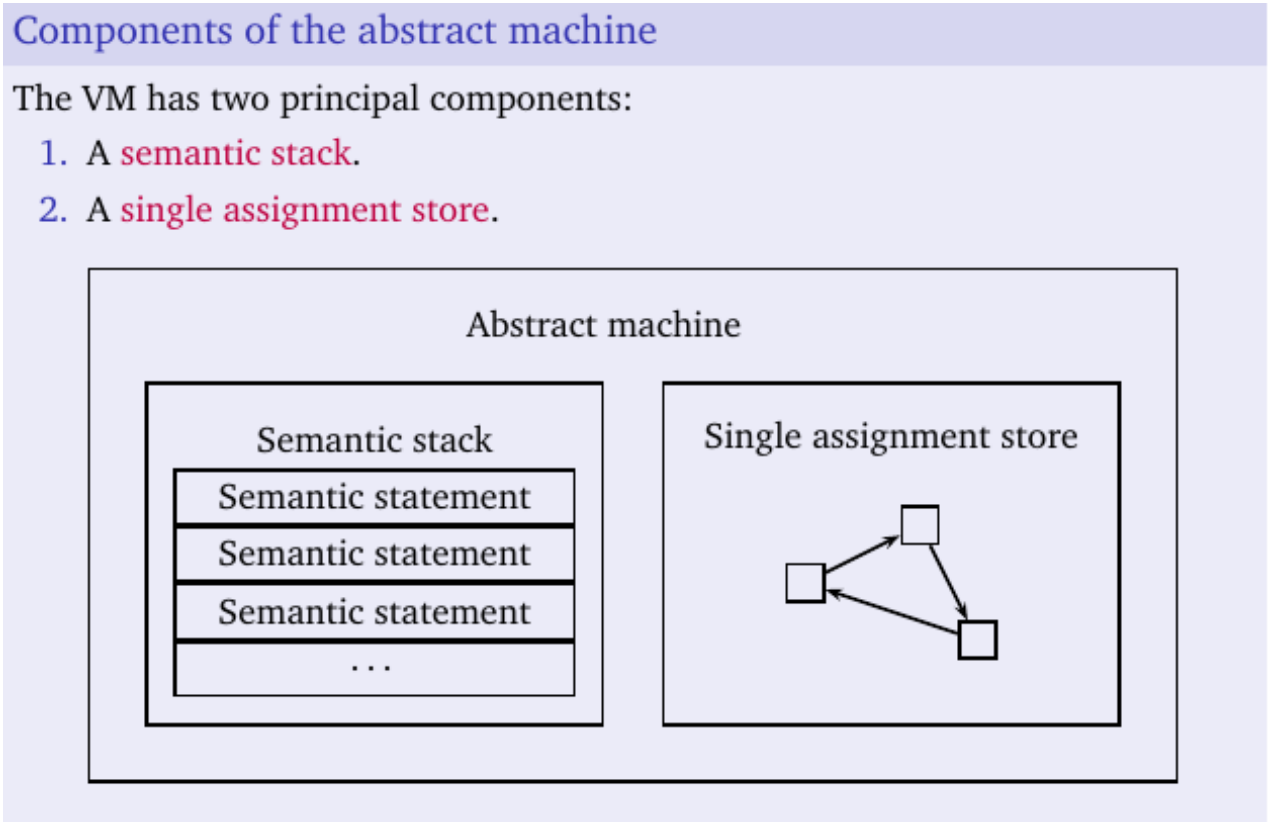

Figure 4: Components of the abstract machine

i) What happens when the following Oz code is processed by the abstract machine?

```
local X in
   local Y in
      local Z in
         X = 2
```

```
            Y = 4
            Z = X * Y
            {Browse Z}
         end
      end
   end
```

Show the execution states until completion. What does the code output?

This code will render the following sequence of execution states:

1. $([(< localX... >)], \phi)$
2. $([(< localY... >), \{X \rightarrow x\}], x)$
3. $([(< localZ... >), \{X \rightarrow x, Y \rightarrow y\}], x, y)$
4. $([(< X = 2.... >), \{X \rightarrow x, Y \rightarrow y, Z \rightarrow z\}], x, y, z)$
5. $([(< X = 2 >), \{X \rightarrow x, Y \rightarrow y, Z \rightarrow z\}), (< Y = 2 >), \{X \rightarrow x, Y \rightarrow y, Z \rightarrow z\}), (< Z = X * Y >), \{X \rightarrow x, Y \rightarrow y, Z \rightarrow z\})]x, y, z)$
6. $([(< BrowseY >), \{X \rightarrow x\, Y \rightarrow y, Z \rightarrow z\}], x = 2, y = 4, z = 8)$

all three variable bindings are done in one step between step 5 and step 6.

j) In what way is an iterative implementation of an algorithm more efficient than a recursive implementation? Use the components of the abstract machine in your explanation.

Recursive implementation will put more and more operations on the stack, requiering more variables to be created and use space in the storage that the itetrative (tail-recursive) implementations do not

k) The Fibonacci sequence is the integer sequence 0 1 1 2 3 5 8 13 21 34 55 89 144... i.e. the integer in place n is the sum of the two previous integers where $n > 2$ where the two first numbers are 0 1.

Write a program that implements an iterative Fibonacci function (a function that returns the nth Fibonacci number) in Oz and output the 7th number in your program.

Tail-recursive implementation:

```
fun {IterativeFib N}
   fun {Fib N F1 F2}
      if N < 1 then F1 else {Fib (N - 1) F2 (F1 + F2)} end
   end
in
   {Fib N 0 1}
end
```

Using cells:

```
fun{FibI N}
   Temp = {NewCell 0}
   A = {NewCell 0}
   B = {NewCell 1}
in
   for I in 1..N do
      Temp := @A + @B
      A := @B
      B := @Temp
   end
   @A
end
```

Part 2: Formal Grammars and Parsing (10%)

Problem 2 The Chomsky Hierarchy of formal grammars consists of regular, context-free, context-sensitive and unrestricted languages. A formal grammar consists of

- A finite set of production rules
- A finite set of non-terminal symbols
- A finite set of terminal symbols
- A start symbol

a) Give an example of a small formal grammar, with the categories above. Use Extended Backus-Naur Form (EBNF). Can a grammar be a member of more categories in the Chomsky Hierarchy? If so, why?

Grammars can be members of more categories, as the the categories are subsets of eachother.

$regular \subset context - free \subset context - sensitive \subset unrestricted$

b) What is the difference between regular and context-free grammars? What is the relation between parsing and a formal grammars place in the hierarchy? What technology is sufficient to parse regular grammars?

The difference is in the right side in the production rules, where the context-free grammars can have rules that go from a non-terminal to a string consisting of the union of non-terminals and terminals.

Regular grammars either right-regular or left-regular have production rules that can only go from one non-terminal to one non-terminal and a terminal, or from a non-terminal to a terminal symbol. (See lecture 3).

Part 3: Extensions to the Declarative Model(30%)

Problem 3 According to CTMCP, all computational models have their place, summarized in the following rule:

a) What does it mean that a declarative program is (a) *natural* and (b) *efficient*?

A declarative program is efficient if it differs just by a constant factor from the performance of an assembly language program to solve the same problem.

A program is natural if very little code is needed for technical reasons unrelated to the problem you are solving.

b) What are some limitations to the declarative model? Give examples of programs that would benefit from being implemented in an extended model.

Programs that do incremental modifications of large data structures, can not be compiled efficiently.

A program that does memoization can not be programmed without changing its interface. An accumulator is needed.

A function that implements intricate code (complex algorithms). A stateful implementation can be easier to write.

c) The *Stateful computational model* is an extension to the declarative model with explicit state. Why is this model no longer declarative? Explain what changes are needed to the abstract machine you sketched above, if any.

A mutable state is needed. The model is no longer declarative, because the explicit states can have side-effects that affect how other components of the program are behaving, and they can not be reasoned about alone.

d) Explain the notion of *memoization*.

Write a program that consists of a function that memoizes another function and create a memoized Finbonacci function using this function. Calculate the 7th Fibonacci number from the memoized Fibonacci function twice. Which calculation will be faster?

You are free to use the Fibonacci function you implemented above in this program if you like.

Memoization is retaining earlier computations, so that they are looked up in memory rather than computed again.

A program that memoizes a given function:

```
declare
fun {Memoize Function}
   Memo = {Dictionary.new} in
   fun {$ Argument}
      try {Dictionary.get Memo Argument}
      catch _ then
         Result = {Function Argument} in
         {Dictionary.put Memo Argument Result}
         Result end end end

local MemFib = {Memoize Fib} in
   {Browse {MemFib 666}}
   {Browse {MemFib 666}}
end
```

The second computation is quicker as the value is simply drawn from the dictionary (see lecture 18).

e) Explain the notion of *streams*

Write a program that consists of two functions, one *generator* function that generates (produces) a stream of integers up to a threshold given as input, and another function that *sums* (consumes) the elements in a list. Invoke the two functions in separate threads like this:

```
{Browse thread {Sum thread {Generator 50000} end } end}
```

```
declare
  fun {Generator N}
   if N > 0 then N|{Generator N-1}
   else nil end
end
```

```
local
    fun {Sum1 L A}
      case L
      of nil then A
      [] X|Xs then {Sum1 Xs A+X}
      end
    end
in fun {Sum L} {Sum1 L 0} end
end
```

Part 4: Data Abstraction and Object-Oriented Programming (30%)

Problem 4 A data type is a set of values together with a set of operations on these values. The basic types in Oz were shown above in Figure 3. A data type is abstract (ADT) if it is defined by its set of operations, regardless of the implementation.

a) Is it possible to change the implementation of an ADT without changing its use? Yes, this is possible (follows from the above definition).

b) Data abstractions can be either *open* or *secure*, *bundled* or not, or *stateful* or *stateless*, opening for 8 ways of organizing data abstraction.

Briefly explain what the three dimensions mean.

open and secure refers to whether the data can be manipulated directly. If something is implemented as a list, and this list be touched directly? or only by the use of operations (secure)'

bundled means that the data type is an entity that combines the notion of value and operation

stateful if it uses explicit state.

c) The queue data abstraction is outlined in Figure 5

Write a program that implements a bundled, secure and stateful *queue* in Oz. Use this queue in this program to enqueue three numbers (of choice), and dequeue 1.

Example (The $\langle queue\ T \rangle$ ADT)

operations:
- new: $\emptyset \rightarrow \langle queue\ T \rangle$
- enqueue: $\langle queue\ T \rangle \times T \rightarrow \langle queue\ T \rangle$
- dequeue: $\langle queue\ T \rangle \rightarrow (\langle queue\ T \rangle, T)$
- empty?: $\langle queue\ T \rangle \rightarrow \{\text{true, false}\}$

axioms:
- empty?(new()) = true
- q_2 = enqueue(enqueue(q_1, e_1), e_2) iff dequeue(q_2) = (q_3, e_1) and dequeue(q_3) = (q_1, e_2)
- dequeue(q) is illegal if empty?(q) = true

Figure 5: The queue abstraction enqueues elements in a FIFO manner, the element first to be enqueued is the first to be dequeued

```
fun {Queue}
   Content
   Front = {NewCell Content}
   Rear = {NewCell Content} in
   queue(enqueue:proc {$ Item} NewEnd in
                    {Exchange Rear Item|NewEnd NewEnd} end
         dequeue:fun {$} NewStart in
                    {Exchange Front $|NewStart NewStart} end) end
```

(see lecture 18).

d) Is it possible to do object-oriented programming in a programming language that is not object-oriented? What is required to do this?

It is possible to implement if the language has necessary elements, like closures.