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 suggested solutions for grading)
0900-1300. August 5, 2013

Language: English

Contact during the exam:

- Lars Bungum (92046135), Øystein Nytrø (91897606)

Exam aids: C. Only approved calculator. No written or printed material permitted.

There is only an *english* version of this exam set.

The exam has 14 subtasks with equal weight. Short and succinct answers are preferred. Do not risk diluting a correct answer by adding exam-bloat! When in doubt about interpretation, state your assumptions. All programs should be written in Oz.

This exam set was approved by Lars Bungum _____

Part 1 Programming paradigms

Task 1.1. What is a programming paradigm? What are the relationships between programming paradigms and programming language semantics?

Grading aid *A paradigm is a preferred view of the world taken by the software designer, or a programming language designer. Bringing some features, concepts, things, doings into the forefront. Programming language semantics is the formal, or computational, definition of what happens when a program (and its sentences) are executed. The relationships between paradigm and semantics concepts, things and doings:*

> *A program* **represents** *a model of a solution to a problem described according to a paradigm, and the program* **has a meaning** *according to the programming language semantics.*

A close correspondence between paradigm, language and semantics makes modelling and programming simple. Domain-specific and paradigm-specific programming languages may improve efficiency and quality manyfold. Some possible programming paradigms, or ways of solving a problem by means of a computer are:

- *Constraint satisfaction*
- *Search*
- *Computation (number crunching...)*
- *Procedural*
- *State-transition*
- *Simulation*
- *Logical/deductive*
- *Data-compositional*
- *Data-stream*
- *Scheduling*

and so on, using different constructs, concepts and execution modes.

One language may support many problem views. The language semantics is designed so that expressions and statements in a language is interpreted and executed according to the relevant paradigm. For example will a state-transition and procedural paradigm require an underlying memory state semantics, a constraint, logic or search paradigm will require a semantics with embodying search, consistency, truth, etc, while a data-stream, simulation or scheduling paradigm will require parallelism, message-passing, state exclusion, mutuality, deadlock management (. . . not further elaborated here). △

Task 1.2. Explain and compare the main features of logic, functional and imperative programming paradigms.

Grading aid *Some features:*

logic *Proposition, facts, rules, queries. Logic variables. Computation as query answering. Logic variables. Resolution. Unification of data structures. Meta-logic programs.*

functional *Function-composition. Functions (and continuations) as data ("first-class citizens"). Anonymous functions. Meta-programming.*

imperative *(Memory-)State. State-change. Execution sequence as computational regime. Sequence manipulation. Branching, loop, procedures. Structured types (records).*

No explanation or comparison here. . . △

Task 1.3. Why should a competent information system architect or designer worry about programming paradigms? Give examples of designs requiring different paradigms.

Grading aid . . . △

Task 1.4. Explain how variables in Oz is different from eg. `C++`. How is a variable implemented in the abstract machine?

Grading aid *Main points of difference:*

- *C++ requires static type declaration and checking (in general). Oz does not.*
- *Oz has logical variables, that are either bound or not, and when fully grounded, retains that value for the rest of its lifetime. C++ variables may change their value.*
- *Oz variables may be bound to each other without being instantiated.*
- *All Oz variables have the same semantics. In C++, reference, value and "constant" variables behave very differently.*

△

Task 1.5. Using the underlying abstract machine, explain in detail how variable equality is implemented.

Grading aid *Equality is not unification or unifiability. That distinction is intended and important. However, unification may yield equality. Main points:*

- *An equality test will not perform an unification.*
- *Equality tests fail for ground, different values or values that can never become equal.*
- *Equality tests suspend for unifiable, non-ground values*
- *Equality tests succeed for ground, equal values*

This can be explained by traversing term structures in SAS, similarly to unification. △

Task 1.6. What are the main features of relational programming as realized in Oz?

Grading aid *Main points worth mentioning are:*

- *The* `choice`*-statement, which introduces alternative subspaces of the single-assignment store, thus allowing retractable bindings and computations. The effect of a succeeding computation subspace is that it is unfolded into the overlying computation space, and the bindings get the same lifespan as the overlying (which may of course be a subspace generated by another* `choice`*-sentence.)*
- *The*
- *!fail!-statement, which emulates a failed sentence (eg. unification) which is used to terminate a computation and retract the spaces created by the choice-statement.*

- *The fact that a statement with a* `choice`*-statement has to be passed (as a parameterless function) to a procedure for traversing the alternatives in a systematic way. Eg.* `SolveAll ...`

△

Task 1.7. Is relational programming declarative? Explain.

Grading aid *Relational programming in Oz is declarative. However, it is (easily) possible to state problems that are indeterministic, or will not terminate, depending on the implementation of .* △

Part 2 Programming in Oz

Sudoku is a one-person number puzzle with the following (approximate) rules/constraints:

1. The board is 9×9 squares, each square empty, or with a digit 1-9.
2. Each single row or column, when completely filled with digits, should contain all the digits from 1-9.
3. The board is subdivided into 9 non-overlapping sub-boards of 3×3 squares.
4. Each single sub-board, when completely filled with digits, should contain all the digits from 1-9.
5. From a state of a partially filled board, fill all empty squares, obeying the constraints 1-4.

Grading aid *Some complained that this quite sizable task had low weight. However, that was intended. The solution does not present detailed code. There are many too many solutions, and the intention is to get an impression of what Oz-features the students know, what elegance they can muster in the design, how literate they are in terms of employing lazy execution, structure-sharing, threads etc.* △

Task 2.1. Make an Oz-program that tests if a sudoku-board is (partially) filled according to the constraints 1-4.

Grading aid

- *Design of data model (binding cells between each 9-digit structure (3×3 ("house" btw.), rows and colums) is a good idea. Keeping a list of unused digits with each 9-structure.*
- *Testing: Possibly using laziness, and freezing threads.*
- *One procedure for each of the 9-digit structures.*

△

Task 2.2. Explain a design for solving a sudoku-puzzle, taking care to utilize relevant Oz-features and taking into consideration computational complexity.

Grading aid

- *Simple: Test and generate. More involved/more efficient: heuristic search based on freedom of cells.*
- *Relational programming for traversing the search space (using heuristic or not.)*
- *Utilizing structure-sharing*
- *Precursor to constraint programming*

△

Task 2.3. Implement your design in Oz.

Grading aid *Evaluation based on actual code quality.* △

Part 3 Kernel language and semantics

Task 3.1. What is a closure? Give an example.

Grading aid *A closure is a mapping (function) between non-local variables occurring in an environment, sentence, procedure body and their values or bindings at the time of definition, such that these bindings are available even when the variables are not occurring in the lexical scope. In particular, this is necessary for languages with function values, because these may be returned as values and evaluated outside the scope of their definition. Without the closure-mechanism, the bindings of the defining environment would get lost. A closure for a procedure definition can be implemented by a unique lambda expression. The lambda expression is invoked (to bind the free local variables) when the procedure is called.* △

Task 3.2. Given the following program with a `case`-statement.

```
local X in
   local V in
      X = x(f:V)
      case X of x(f:X) then
         V = X
      else skip
      end
   end
end
```

Make a step-by-step listing of execution stack and single-assignment store. Is the `V = X` statement executed at all? What is the content of `X`?

Grading aid

1 ([(`local X in ... end`,{})], {})

2 ([(`local V in ... end`,{X → v_1})], {v_1})

3 ([(`X=x(f:V) case ... end`,{X → v_1, V → v_2})], {v_1, v_2})

4 ([(`X=x(f:V)`,{X → v_1, V → v_2}), ...], {v_1, v_2})

5 ([(`case X of x(f:X) ... end`,{X → v_1, V → v_2})], {v_1 = x(f:v_2), v_2})

6 ([(`V=X`,{X → v_2, V → v_2})], {v_1 = x(f:v_2), v_2})

7 ([], {v_1 = x(f:v_2), v_2})

The inner `X` *is distinct from the outer one, so the unification is executed and both* `V` *and* `X` *are both unbound. (See line 6.)* △

Task 3.3. Given the following program with a `try`-statement.

```
local A B C in
   try
      A = 1
      B = 2
      raise B end
      C = 3
   catch E then
      {Browse E}
   end
   {Browse A#B#C}
end
```

What will be displayed? Explain in detail (using execution state listing as needed.)

Grading aid *The explicit and long version (not expected as a solution...) The point is of course that bindings from before the exception is raised remain after the catch.*

```
1 [ (local A in ... end, {}) ]
  { }
```

2
```
[ (local B in ... end, {A:v1}) ]
{ v1 }
```

3
```
[ (local C in ... end, {A:v1, B:v2}) ]
{ v1, v2 }
```

4
```
[ (try ... end {Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1, v2, v3 }
```

5
```
[ (try ... end, {A:v1, B:v2, C:v3}),
  ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1, v2, v3 }
```

6
```
[ (A=1 B=2 raise B end C=3, {A:v1, B:v2, C:v3}),
  (catch E ... end, {A:v1, B:v2, C:v3}),
  ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1, v2, v3 }
```

7
```
[ (A=1, {A:v1, B:v2, C:v3}),
  (B=2 raise B end C=3, {A:v1, B:v2, C:v3}),
  (catch E ... end, {A:v1, B:v2, C:v3}),
  ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1, v2, v3 }
```

8
```
[ (B=2 raise B end C=3, {A:v1, B:v2, C:v3}),
  (catch E ... end, {A:v1, B:v2, C:v3}),
  ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1=1, v2, v3 }
```

9
```
[ (B=2, {A:v1, B:v2, C:v3}),
  (raise B end C=3, {A:v1, B:v2, C:v3}),
  (catch E ... end, {A:v1, B:v2, C:v3}),
  ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1=1, v2, v3 }
```

10
```
[ (raise B end C=3, {A:v1, B:v2, C:v3}),
  (catch E ... end, {A:v1, B:v2, C:v3}),
  ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1=1, v2=2, v3 }
```

11
```
[ (raise B end, {A:v1, B:v2, C:v3}),
  (C=3, {A:v1, B:v2, C:v3}),
  (catch E {Browse E} end, {A:v1, B:v2, C:v3}),
  ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1=1, v2=2, v3 }
```

12
```
[ ({Browse E}, {A:v1, B:v2, C:v3, E:v2}),
  ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1=1, v2=2, v3 }
```

13 *'2' displayed in the browser window*

```
[ ({Browse A#B#C}, {A:v1, B:v2, C:v3}) ]
{ v1=1, v2=2, v3 }
```

14 *'1#2#_' displayed in the browser window*

```
[ ]
{ v1=1, v2=2, v3 }
```

△

Task 3.4. Explain the implementation of ports in Oz. Why do ports need an extension of the kernel declarative kernel model with mutable store?

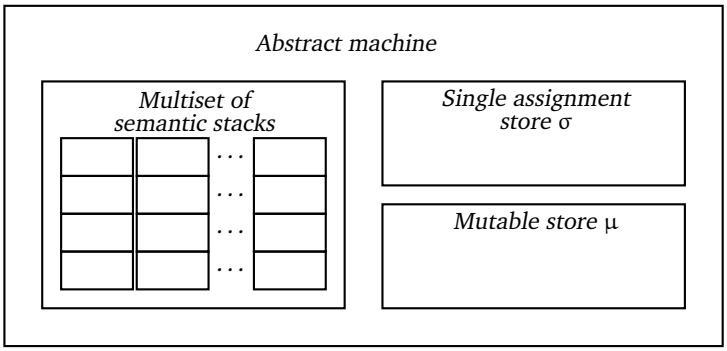
Grading aid *Grabbed directly from lecture 12: Ports have mutable state which cannot be accommodated for with dataflow variables.*

- *We need to extend the abstract machine—besides the single-assignment store we need a mutable store.*

The mutable store μ *contains pairs* (p, s)*, where:*

- p *is a port variable,*
- s *is the current end of the stream associated with the port* p*, accessible only through the port.*
- p *and* s *are regular dataflow variables—their values, once created in the single assignment store* σ*, cannot change.*
- *Port-stream associations* (p, s) *can be added to and removed from* μ*.*
- *The mutable state of the port* p *is realized by means of replacing* (p, s) *with* (p, s') *in* μ*, where* s' *is the new end of the stream.*

The abstract machine is extended with a mutable store μ *as shown underneath:*



The machine may, but does not have to, include multiple semantic stacks (for concurrency) and a trigger store τ *(for lazy execution)*

$\triangle$