NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Computer and Information Science

Examination paper for **TD4165 Programming languages**

Grading aid

Academic contact during examination: Øystein Nytrø

Phone: 91897606

Examination date: 2015-12-04

Examination time (from–to): 09.00 – 13.00

Permitted examination support material: C: Specified (nothing) printed and hand-written support material is allowed. A specific basic calculator is allowed.

Other information:
This exam set is for a course taught in English only: In order to avoid inconsistency and interpretation problems, only one version exists (this one).

The concurrency problems can be solved in both Scala and Oz. There are NO alternate language-specific problems. (Risk of misunderstandings.)

The exam set has 4/14 parts/tasks. Every task has the same weight towards the final score.

There is only one correct answer in the multiple choice problems.

Short, succinct answers preferred. Explain necessary assumptions.

Language: English

Number of pages: 9

Number pages enclosed: 0

Checked by:

Date Signature

Grading aid *The grading aid is not a complete solution! Code answers with many solutions are not included, only lists of criteria for reviewing the code.* △

Part I Oz and semantics

1) How many of the expressions below are NOT lists?

```
'|'(one nil)
'|'(1:one 2:nil)
'|'(2:one 1:nil)
'|'(nil one)
'|'(nil nil)
'|'(one '|'(two nil))
'|'('|'(one nil) nil)
```

(a) 1 (c) 3 (e) 5
(b) 2 (d) 4 (f) 0

Grading aid *b, ie. 2! There was a superfluous ")" in 6), but it was still a list. To be fair, answer "c)" scores 4 of 10 p.*

```
 {Browse '#'(
'|'(one nil) =  [one]
'|'(1:one 2:nil) = [one]
'|'(2:one 1:nil) = (nil | one)
'|'(nil one) = (nil | one)
'|'(nil nil) = [nil]
'|'(one '|'(two nil)) = [one two]
'|'('|'(one nil) nil) )  = [[nil]]    )
 }
```

△

2) Explain the necessary conditions for a variable identifier occurrence to be bound in the scope of a procedure body. Explain what it means that a logical variable is not bound to a value. Using the following code, explain for all variable occurrences what representations it gives rise to in the single assignment store and the statement stack environment.

```
local X in
  local Y F in
    fun {F X} X * Y end
```

```
    Y=3
    Z=X+Y
    {F Z}
  end
end
```

Grading aid *The essential point in this task is to distinguish between identifier name binding (in a environment/scope) versus value binding (in SAS). An identifier is bound (occurs in the environment) in a procedure body if occurs in the scope of a declaration (ie. "local") or is a formal parameter. An identifier corresponding to a logical variable is not bound if the logical variable is not bound to a value (or to another unbound variable). Ie. in* `local X Y in X = Y end` *X and Y are considered bound (together). Consult section 2.2 for good illustrations!*

It was not required to evaluate the expressions in the program and show the stack, environment and SAS in detail. Many detected use of unbound variables, and commented on program behaviour. This was not asked for. Below are thus commenting on the statical interpretation of identifiers.

```
local X in
  % Xvar free, Xid bound
  local Y F in
    % Yvar, Fvar free, Yid,
    fun {F X} X * Y end
       % newXid bound, Yid free in fun, Fvar bound (to "fun {...")
       Y=3
       % Yvar bound
       Z=X+Y
      % (Zid free, from outer space?),
      % Yvar bound,
      % Xvar unbound .... will suspend ...
      {F Z}
      % Fvar bound, Zvar would have been bound if reached, but not possible
  end
end
```

△

3) Given

```
fun lazy {LMap Xs F}
  case Xs of nil then nil
  [] X|Xr then {F X}|{LMap Xr F} end
end
```

Does this code work on (i) infinite streams, (ii) only on finite lists, or (iii) on both finite and infinite streams? Briefly explain. Is the function LMap defined by this code incremental?

Grading aid *It will work on both infinite streams, because evaluation is incremental, producing a result value (only partially bound) on each invocation, and finite lists, since the end is handled properly. Incremental means that computation is stepwise, not accumulated and monolithic. Incremental does not preclude recursiveness. Incremental is not the same as iterative (looping, ie. non-recursive). (Since the list is recursively built, with an unbound variable at the end, it is a tail recursive function. But that was not an issue.)* $\triangle$

4) Which statement is correct?

(a) Oz uses dynamic scoping for identifiers, which is why Oz also uses dynamic type checking.

(b) Oz uses dynamic scoping for identifiers, which makes it easy to predict the values of identifiers.

(c) Oz uses static scoping for identifiers, which makes it impossible to do static type checking.

(d) Oz uses static scoping for identifiers, which makes it possible to statically predict the types of identifiers.

(e) Oz uses static scoping for identifiers, which requires it to make closures for procedure values.

(f) None of the above.

Grading aid *e)* △

5) What is the output, if any, of the following code in Oz? Explain!

```
local P Unk W in
  P = person(height: 62 weight: 190 age: Unk) W = 55
  case P of building(height: H weight: W age: A)
      then {Browse first#H#W#A}
    [] person(height: H weight: 190)
      then {Browse second#H}
    [] person(height: H weight: W age: A)
      then {Browse third#H#W#A}
    [] person(height: H weight: X age: A)
      then {Browse fourth#H#X#A}
    else {Browse none}
  end
end
```

Grading aid *Running the program prints* `third#62#190#_` *in the browser window. In the first case, person and building is not not unifiable. In the second case, the pattern has different number of positions, ie. age is missing. In the third case, unification succeeds. W is a local variable, and not the same as declared as local and bound to 55. The rest of the cases are not reached.* △

Part II Language and syntax

Given a partial grammar G_E for exponentiation expressions in a programming language:

```
<Expr> ::= <Expr> '**' <Expr> | <Integer>
<Integer> ::= ...
```

Assume that integers are Oz integers

1) Is the grammar G_E ambiguous? Explain why (not). Define associativity and precedence using exponentiation expressions. Can you rewrite the grammar into G_U so that it becomes easier to parse (and interpret)?

Grading aid *The grammar is ambigious,* `1 ** 2 ** 3` *will give rise to two different parse trees. Associativity for an operator (or set of operators with the same precedence) determines which of two (or more) productions should be applied. Selecting the leftmost operator yields left-associativity, and rightmost right-asssociativity. Precedence determines and application order among production rules (usually for operators).*

A left-associative grammar G_LA for exponentiation expressions:

```
<Expr> ::= <Mexpr> '**' <Expr> | <Mexpr>
<Mexpr> ::= <Integer>
<Integer> ::= ...
```

Precedence is not meaningful with only one operator. A left-associative, and right-recursive, grammar like G_LA lends itself to recursive descent parsing. Factoring

```
<Expr> ::= <Integer> <Etail>
<Etail> ::= '**' <Expr>  | epsilon
<Integer> ::= ...
```

and simplifying yields G_U:

```
<Expr> ::= <Integer> '**' <Expr>  | <Integer>
<Integer> ::= ...
```

which is straightforward to implement as a predictive recursive descent parser.

The grammar transformations was intended to be language-preserving. Introducing parentheses changes the language, and was not a solution to this task. $\triangle$

2) Define a grammar G_T for Oz-expressions representing the abstract syntax tree of expressions in G_U. Use BNF or EBNF. The trees should be valid record expressions in Oz.

Grading aid *There are many possible ways to represent different versions of G_U. We can do away with the (unique) operator, and just represent the operands in a binary tree:* `exp(2 exp(3 4))` *represents $2^{(3^4)}$. The corresponding G_T would be:*

```
<Expr> ::= 'exp(' <Oz integer> <Expr> ')' | <Oz integer>
```

△

3) Write an interpreter in Oz for exponential expressions according to G_U.

Grading aid *Using G'_U, we can simply write:*

```
declare fun {Evalexp E}
   case E of exp(N M)
    then {Number.pow N {Evalexp M}}
% Using the infix '**' would count as correct also.
   [] E then E
   end
end

{Browse {Evalexp exp(2 exp(3 4))}}
```

However, useful exponential expressions should be able to handle parentheses and subexpressions. But that was not included in G_E, so not expected! △

Part III Concurrency

All the tasks can be solved using either Oz or Scala.

1) Compare sequential and concurrent program execution. What are the key differences? What are the benefits of concurrency? What are the drawbacks of concurrency?

Grading aid *Review:*

1. *Differences: Independent processes are most naturally regarded as concurrent. Sequentiality introduces arbitrary constraints.*
2. *Benefits are easier decomposition, clear separation of independent tasks, distribution, failure tolerance, ease of analysis and run-time control and not least possibility of parallel execution.*
3. *Drawbacks are need for synchronisation, arbitrarily difficult timing problems, race conditions, resource starvation, possible non-determinism, deadlocks (detection and resolution) and much less ability to "design" interaction. Testing, debugging and program correctness proofs generally becomes (very) much more difficult.*

△

2) What is a deadlock? Explain and give a short code example that may lead to deadlock. How can you avoid deadlock?

Grading aid *Deadlock occurs when two or more processes or activities are mutually waiting for the other to finish or release resources, in a way that prevents all from finishing. The result of a deadlock is that all processes or activities are suspended indefinitely. For a deadlock to occur, the following conditions must be fulfilled: 1) at least one resource is mutually exclusive or not shareable. 2) It is possible to hold a resource while waiting (indefinitely) for another resource. 3) A resource can only be released voluntarily by the actor holding it. 4) Actors may acquire resources and wait for others forming, a circular interdependency.*

The code example should exemplify this.

Deadlock can be avoided by avoiding any of these conditions, ie.: Forcing release of resources if waiting for others, or requiring block allocation of resources. Refine access to resources according to real need (read/write etc.). Non-blocking process analysis and scheduling. Deadlock detection and preemptive resource release. Avoidance of circular dependencies by partially ordered hierarchies of communicating processes, or fixed order resource allocation. Scala: Use of actors and futures/promises. △

3) Implement a concurrent stack (First-in, Last-out) abstract data type. Explain your underlying representation of the stack. Define and implement functions `push`, `pop` and `clear`. Explain how you want to handle operations on empty stacks. What does it mean for an object to always be in a consistent state? How do you make sure your stack is always consistent?

Grading aid

1. *Look for complete functionality and consistency between representation and implementation.*
2. *Ensure proper handling of empty stack*
3. *Look for consistency of internal state.*

An object is in a consistent state if all operations on the object assumes that the object is in a consistent state, operations always result in a consistent state, and operations are atomic and mutually exclusive.

An implementation of a concurrent stack using LnkedList with synchronized operators could look like:

```
Class ConStack[T] {
var  s = LinkedList()
def clear = synchronized {s.clear()}
def pop: Option[T] = synchronized {s.pop()}
```

```
def push(v: T) = sycnhronized {s.push(v)}
}
```

△

4) Wikipedia says:

> In computer science, *future*, *promise*, and *delay* refer to constructs used for synchronization in concurrent programming languages. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete. ... Specifically, when usage is distinguished, a future is a read-only placeholder view of a variable, while a promise is a writable, single assignment container which sets the value of the future.

How are futures and promises realized in Oz or Scala? Explain the drawbacks and benefits of concurrent code using futures and promises versus code using explicit synchronization.

Grading aid *Futures are realized in Oz by means of logical variables acting as dataflow variables in combination with Lazy execution. Futures in Scala are asynchronous operations spawned from another process and expecting some value, and made available either by the spawning process blocking itself, or reacting upon various callbacks. A future is realized as a wrapper for an integer. A promise is a single-assignment variable that encapsulates a future object. See eg. verb!docs.scala-lang.org/overviews/core/futures.html!*

The benefit of using futures, or dataflow variables, is that it succinctly represents an often occurring pattern of asynchronous concurrency (expect a result, but continue until result is needed). It is often possible to implement very efficiently with low-level synchronization. △

Part IV Relational programming

1) Which answer is false?

(a) Relational programming realizes logic programming.

(b) Relational programming extends functional programming.

(c) Relational programming requires concurrency.

(d) Logic variables can be used in functional programming.

(e) Logic variables support declarativeness.

Grading aid *c) is false* △

2) Define a relation `{Consec Ls A B S}` that succeeds when in the list Ls, A and B are adjacent in that order, and the rest of the list is S. See examples:

```
{Browse {SolveAll proc {$ S} {Consec nil 1 2 S} end}}
> nil
{Browse {SolveAll proc {$ S} {Consec 1|2|nil 1 2 S} end}}
> [nil]
{Browse {SolveAll proc {$ S} {Consec [1 2 3] 1 2 S} end}}
> [[3]]
{Browse {SolveAll proc {$ S} {Consec [7 1 2 3 2 0 1 2 5] 1 2 S} end}}
> [[3 2 0 1 2 5] [5]]
{Browse {SolveAll proc {$ S} {Consec [7 1 2 3 2 0 1 2 5] 1 3 S} end}}
> nil
{Browse  {SolveAll proc {$ S} {Consec [3 1 3 1 1] 3 1 S} end}}
> [[3 1 1] [1]]
```

Grading aid *One simple solution, satisfying the above examples, uses the previously defined relational* `Append`*:*

```
proc {Append L1 L2 L3}
   choice
      L1=nil L3=L2
   [] X M1 M3 in
      L1=X|M1 L3=X|M3 {Append M1 L2 M3}
   end
end

proc {Consec List E1 E2 Rest}
   Head in {Append Head E1|E2|Rest List} %Head is just thrown away
end
```

△