NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Computer Science

Examination paper for **TD4165 Programming languages**

Grading aid

Academic contact during examination: Øystein Nytrø

Phone: 91897606

Examination date: 2017-12-12

Examination time (from–to): 15.00 – 19.00

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.

The exam set has 4 parts with 11 problems. All problem grades will contribute with the same weight towards the final grade.

Short, succinct answers are strongly preferred. When needed, explain necessary assumptions.

Language: English

Number of pages: 8

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.*

A good solution answers all questions for each task. △

Part I Oz and semantics

1) Define an iterative version of `{Length Xs}`:

```
fun {Length Xs}
 case Xs of nil then 0
 [] _|Xr then 1+{Length Xr}
 end
end
```

Grading aid *An example from CTMCP 3.4.2.4:*

```
local
 fun {IterLength I Ys}
  case Ys
   of nil then I
      [] _|Yr then {IterLength I+1 Yr}
    end
  end
in
  fun {Length Xs}
    {IterLength 0 Xs}
  end
end
```

△

2) An n-ary tree can be defined by the following grammar:

```
<Tree T> ::= tree(node:T branches:<List <Tree T>>)
```

(Assuming that the `<List ...>` represents the Oz list syntax.) All internal nodes have a value `T`, and a possibly empty list of branches, each branch a tree.

What is the minimal tree? Define a bottom-up accumulator of node values `{FoldTree NodeAcc BranchAcc Tree StartVal}` where `BranchAcc` accumulates a list of branches, while `NodeAcc` accumulates a node value and the

result of accumulating branches. `StartVal` is the value for an empty list of branches.

Grading aid *The minimal tree does not have any branches, and is* `tree(X nil)`*, where X could be a value or a variable. Ie. there is no non-tree or nil-value.*

A bottom-up accumulator could be realized by the following definitions:

```
local
  fun {FoldTreeR NodeAcc BranchAcc ListofTrees U}
    case ListofTrees
    of nil then U
    [] S|ListofTrees2 then
      {BranchAcc
        {FoldTree  NodeAcc BranchAcc S U}
        {FoldTreeR ListofTrees2 NodeAcc BranchAcc U}}
    end
  end
in
  fun {FoldTree NodeAcc BranchAcc Tree  U}
    case Tree of tree(node:N branches:ListofTrees) then
      {NodeAcc N {FoldTreeR NodeAcc BranchAcc ListofTrees  U}}
    end
  end
end
```

△

3) Explain what it means for a statement (in Oz) to be declarative. Explain, by example, why `Z = X` is declarative.

Grading aid *Following the CTMCP, section 3.1.3:*

> *Given any statement in the declarative model, partition the free variable identifiers in the statement into inputs and outputs. Then, given any binding of the input identifiers to partial values and the output identifiers to unbound variables, executing the statement will give one of three results: (1) some binding of the output variables, (2) suspension, or (3) an exception. If the statement is declarative, then for the same bindings of the inputs, the result is always the same.*
>
> *For example, consider the statement* `Z=X`*. Assume that X is the input and Z is the output. For any binding of X to a partial value, executing this statement will bind Z to the same partial value. Therefore the statement is declarative.*

Both Z and X may be regarded as inputs. Examples leading to suspension or exceptions should also be included for completeness. △

4) What are the two main features of declarative programming? What kind of declarativeness do we find in programming?

Grading aid *There are of course many features of declarative programming, but these two are central (and also mentioned in the CTMCP):*

1. *Declarative programs are compositional. A declarative program consists of components that can each be written, tested, and proved correct independently of other components and of its own past history (previous calls).*
2. *Reasoning about declarative programs is simple. Programs written in the declarative model are easier to reason about than programs written in more expressive models. Since declarative programs compute only values, simple algebraic and logical reasoning techniques can be used.*

Determinism is, if you think about it, an important feature of all programs. Determinism is thus not a sufficient requirement for declarativeness.

The kinds of declarativeness can be illustrated by the classes in figure 3.3 in the CTMCP-book:

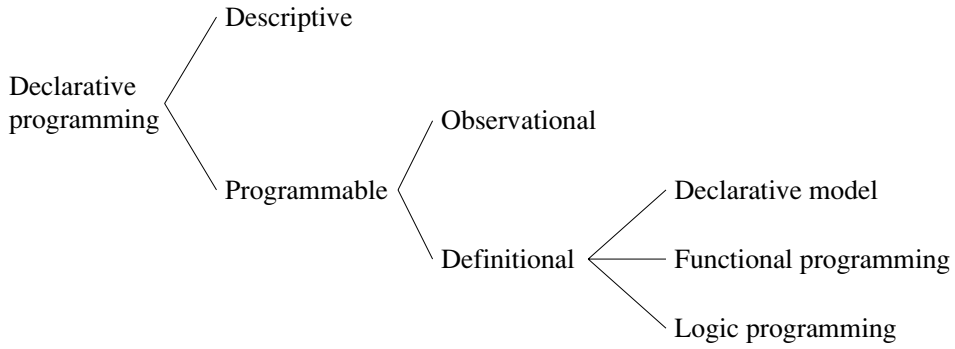


Figure 3.3: A classification of declarative programming.

△

5) Three types of sentences are suspendable in the kernel language:

```
<s> ::= ...
| if <x> then <s1> else <s2> end
| case <x> of <pattern> then <s1> else <s2> end
| {<x> <y1> ...<yn>}
```

What does suspendable mean in this context? What are their respective conditions for becoming activated again? In a sequential programming model (with a single semantic stack), when does activation occur?

Grading aid *Suspendable means that execution halts until the variable <x> is bound, respectively to a number, record or procedure value. In a sequential programming model, activation will never occur, so suspension simply stops execution.* △

Part II Language and syntax

1) Given several (more or less well-designed) grammars for binary numbers:

```
a) <number> ::= <digit> <number>
    <digit> ::= 0|1
b) <number> ::= <number> | <digit> | <epsilon>
    <digit> ::= 0|1
c) <number> ::= <digit> | <digit> <number> | <epsilon>
    <digit> ::= 0|1
d) <number> ::= <number> <digit> | <number>
    <digit> ::= 0|1
e) <number> ::= <digit> <number> | <digit>
    <digit> ::= 0|1
```

Where `<epsilon>` represents the empty string. Which grammar(s), if any, will produce the exact string `1010`? Show the stepwise productions.

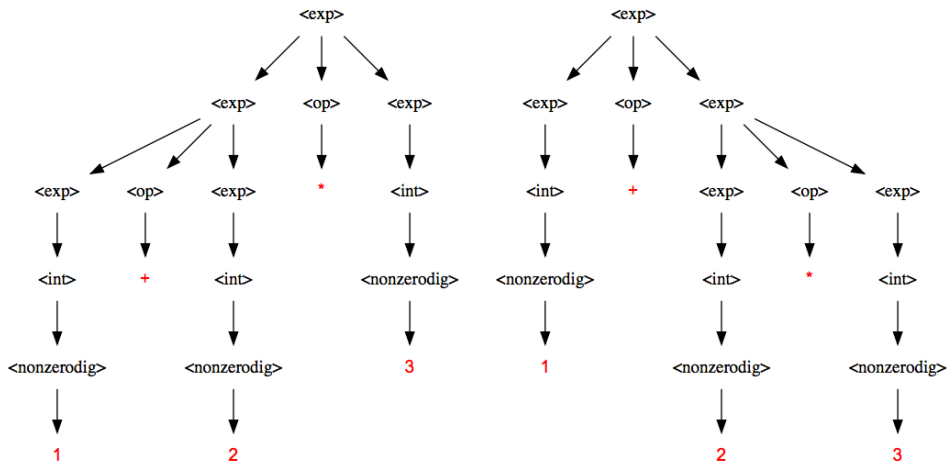
Grading aid *c) and e). Derivations required.* △

2) Given the grammar for arithmetic expressions over numbers:

```
<exp> ::= <int> | <exp> <op> <exp>
<op>  ::= +|-|*|/
<int> ::= <nonzerodig> { <dig> }
<dig> ::= 0 | <nonzerodig>
<nonzerodig> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Draw all parse trees for different derivations of the expression `1+2*3`.

Grading aid *A parse tree is an ordered, rooted tree that represents the syntactic structure of a string parsed according to a context-free grammar. All internal nodes represent non-terminals, the leaves represent terminals (or lexemes). The*



root for a parse tree of a complete sentence derived from the grammar is always the start symbol. An abstract syntax does only contain terminal symbols, ordered according to precedence (or convenience for further processing). Abstract syntax trees were not asked for.

△

Part III Concurrency

Use either Oz or Scala for examples or solutions.

1) Explain the language features that support concurrency (in either Oz or Scala). Give program examples. Explain the role of immutability of data.

Grading aid *Solution should mention, explain and exemplify features like:*

Scala *Threads, actors, immutable variables (val), "synchronized", atomicity, runnable*

Oz *Threads, logic variables, controlled mutable ports, lazy execution, relational programming (utilizing concurrency),*

Examples required! △

2) Explain and show how a memory model (of either Oz or Scala) supports concurrency.

Grading aid *This was intended to explain the underlying semantics, in contrast to the previous question that was about syntax and language features. In practice, slightly overlapping. Main points:*

Scala *Scala (before 2.11) employs the JVM memory model. It is not thread-safe, and eg. symbol initialisation and reflection may lead to race conditions. Process and thread memory models (are different).*

Oz *Single SAS, multiple samantic stacs. Communication by sharing. Lazyness implementation. Suspension, exceptions and triggering. Implementation of ports and cells.*

Examples required! △

Part IV Logic and constraint programming

1) NTNUI has an incredible network of tiny, small and large cabins (koier and kåter). Walking between them is great! The following set of Prolog facts describes a small part of the network, with access points (from public or private transport), cabins, and direct trips with estimated walking time between locations.

```
cabin(nico).
cabin(kraakli).
cabin(lynhoegen).
cabin(flaa).
cabin(morten).

access(fremo).
access(langlete).
access(broettem).
access(lundamo).

trip(fremo,nico,3).
trip(nico,kraakli,2).
trip(kraakli,lynhoegen,5).
trip(nico,lynhoegen,5).
trip(kraakli,morten,5).
trip(morten,langlete,4).
trip(flaa,nico,7).
trip(lundamo,flaa,2).
trip(broettem,lynhoegen,4).
trip(lundamo,kraakli,7).
```

Define a predicate `path(From,To)` if there is a contiguous path between `From` and `To` without repetitions, backtracking or cycles. Define a predicate `tour(From,To,Route,Hours)` which succeeds if there are trips connecting `From`

to `To`, with the list of places visited in `Route` and total walking time `Hours`. Finally, make a query that will give walking time between pairs of connected access points.

Grading aid

```
cabin(nico).
cabin(kraakli).
cabin(lynhoegen).
cabin(flaa).
cabin(morten).

access(fremo).
access(langlete).
access(broettem).
access(lundamo).

trip(fremo,nico,3).
trip(nico,kraakli,2).
trip(kraakli,lynhoegen,5).
trip(nico,lynhoegen,5).
trip(kraakli,morten,5).
trip(morten,langlete,4).
trip(flaa,nico,7).
trip(lundamo,flaa,2).
trip(broettem,lynhoegen,4).
trip(lundamo,kraakli,7).

% and, trips can be walked both directions
trip(X,Y,T) :- trip(Y,X,T).

% use a list as a set representation
% acceptable to use without definition
member(X,[X|_]).
member(X,[_|R]) :- member(X,R).

% nonmembership, and indeed negation as failure,
% was not in the curriculum, but any
% indication of nonmembership would be acceptable!
notmember(X, S) :- \+ member(X,S).

% introducing a helper relation with accumulator of visited places
path(From,To) :- path_acc(From,To,_P).
```

```
path_acc(From,To,[From,To]) :- trip(From,To,_).
path_acc(From,To,[From|P]) :- trip(From,Place,_),
path_acc(Place,To,P), notmember(From, P).

tour(From,To,[From,To],Hours) :- trip(From,To,Hours).
tour(From,To,[From|P],Hours) :- trip(From,Place,TH),
tour(Place, To, P, ToH), Hours is TH+ToH.
```

△

2) Explain what constraint (logic) programming is, and outline the basic principles for solving a constraint problem.

A list represents items in a knapsack. The list `[item(A,3,4),item(B,4,5)]` represents all collections with A number of items with value 3 and weight 4 and B number of items with value 4 and weight 5. The unbounded knapsack problem is to maximize a total value of a collection (the knapsack), given a maximal total weight (that you can carry) and given the specified kind of items that you may select to put in the sack. Define the predicate `knapsack(TotalValue, MaxWeight, ListofItems)` that succeeds with a sack packed as specified in the list, with no more than maximal weight and a computed total value. A sample run for maximal weight 5 is:

```
?- knapsack(V,5,[item(A,3,4),item(B,4,5),item(C,2,2)]),
   label([V,A,B,C]).

A = B, B = C, C = V, V = 0 ;
A = B, B = 0, C = 1, V = 2 ;
A = 1, B = C, C = 0, V = 3 ;
A = B, B = 0, C = 2, V = 4 ;
A = C, C = 0, B = 1, V = 4 ;
false
```

Grading aid *"Define the predicate" does of course imply a solution in Prolog using CLP(FD) as lectured and used in an assignment.*

```
knapsack(0, 0, []).
knapsack(TotVal, TotWeight, [item(Amount,Val,Weight)|RestItems]) :-
    Amount #>= 0,
    TotWeight #>= Amount*Weight + RestWeight,
    TotVal #= Amount*Val + RestVal,
    knapsack(RestVal, RestWeight, RestItems).
```

△