TRIAL EXAM
TDT4165 PROGRAMMING LANGUAGES SOLUTIONS

No printed or handwritten material is allowed.

Part 1 Multiple Choice

For each subtask, choose the one answer that you think is the most correct.

a) Which of the following functions is a correct `Append` function?

A:
```
fun {Append L1 L2}
      case L1
      of nil then L2
      [] H|T then H|{Append T L1}
      end
   end
```

B:
```
fun {Append L1 L2}
      case L2
      of nil then L1
      [] H|T then {Append T L1}|H
      end
   end
```

C:
```
fun {Append L1 L2}
      case L1
      of nil then L2
      [] H|T then H|{Append L1 L2}
      else raise notAListException end
      end
   end
```

D:
```
fun {Append L1 L2}
      case L1#L2
      of nil#L then L
      [] L#nil then L
      [] (H|T)#L then H|{Append T L}
      end
   end
```

Comment/solution

D is correct.

b) Consider the following execution state of the abstract machine in the computation model of the declarative kernel language:

```
( [ ( {X Y R}, {X->a, Y->b, Z->c, R->d} ) ],
  {a->( proc {$ Y R} R=Y+Z end, {Z->e} )
   b->5, c->7, d, e->3} )
```

What will be the next execution state?

A: There will be no next state.

B:
```
( [ ],
  {a->( proc {$ Y R} R=Y+Z end, {Z->e} )
   b->5, c->7, d->8, e->3} )
```

C:
```
( [ ],
  {a->( proc {$ Y R} R=Y+Z end, {Z->e} )
   b->5, c->7, d->10, e->3} )
```

D:
```
( [ ( R=Y+Z, {Y->b, Z->e, R->d} ) ],
  {a->( proc {$ Y R} R=Y+Z end, {Z->e} )
   b->5, c->7, d, e->3} )
```

Comment/solution

D is correct.

c) Which properties(s) of the following grammar would be problematic for parsing with a left-to-right recursive-descent parser?

```
<s> ::= <s> <s>
      | if <s> then <s> else <s> end
      | <t>
<t> ::= hip | hop
```

A: Left-recursiveness

B: Right-recursiveness

C: Left- and right-recursiveness

D: Ambiguity

Comment/solution

A is correct. See exercise 4. D is also correct. This was a mistake in the exercise.

d) Mozart uses lexical scoping in function calls. We wish to extend the semantics of the kernel language so that we can also perform function calls with dynamic scoping. For this we introduce the following syntax:

```
dyn {<x> <y>1 ... <y>n}
```

In addition, we must define the semantics of the expression by giving a rule for how the abstract machine should interpret the expression.

The semantic statement is:

(dyn {<x> <y>1 ... <y>n}, E)

Execution consists of the following actions:

- If the activation condition is true (E(⟨x⟩) is determined), then do the following actions:
  - If E(⟨x⟩) is not a procedure value or is a procedure with a number of arguments different from n, then raise an error condition.
  - If E(⟨x⟩) has the form (proc {$ ⟨z⟩1, ... ⟨z⟩n} ⟨s⟩ > end, CE) then push _____ on the stack.
- If the activation condition is false, then suspend execution.

Which of the following should go in the empty space in the rule above?

A: (<s>, CE + { <z>1->E(<y>1), ..., <z>n->E(<y>n) })

B: (<s>, E + { <z>1->CE(<y>1), ..., <z>n->CE(<y>n)})

C: (<s>, E + { <z>1->E(<y>1), ..., <z>n->E(<y>n)})

D: (<s>, E + { <y>1->E(<z>1), ..., <y>n->E(<z>n)})

Comment/solution

C is correct.

e) Consider the following type specification:

```
<Samling T> ::= null |
ord(tekst:<String> verdi:<T> ene:<Samling T> andre:<Samling T>)
```

Which of the following is a correct instance of `<Samling <Int>>`?

A:
```
ord(tekst:[101 110 101]  verdi:0
       ene:ord(tekst:''andre''  verdi:0  ene:null  andre:null)
       andre:ord(tekst:nil  verdi:0  ene:null  andre:null))
```

B:
```
ord(tekst:''ene''  verdi:null
       ene:ord(tekst:''andre''  verdi:null  ene:null  andre:null)
       andre:ord(tekst:''tredje''  verdi:null  ene:null  andre:null))
```

C:
```
ord(tekst:''ene''  verdi:0
       ene:ord(tekst:''andre''  verdi:0  ene:nil  andre:nil)
       andre:ord(tekst:nil  verdi:0  ene:nil  andre:nil))
```

D:
```
ord(tekst:'ene'  verdi:0
       ene:ord(tekst:'andre'  verdi:0  ene:null  andre:null)
       andre:ord(tekst:'tredje'  verdi:0  ene:null  andre:null))
```

E: None of the above.

Comment/solution

A is correct.

f) Which of the following functions will run with constant stack size?

```
fun {F1 A B}
   if A==0 then B
   else {F1 A-1 B+A}
   end
end

fun {F2 A B}
   if A\=0 then {F1 A-1 B+A}
   else B
   end
end
```

A: Neither.

B: F1 but not F2.

C: F2 but not F1.

D: Both.

Comment/solution

D is correct.

g) *Lazy execution ...*

A: is incompatible with functional programming.

B: is incompatible with relational programming.

C: is incompatible with object-oriented programming.

D: is compatible with functional, relational and object-oriented programming.

Comment/solution

D is correct.

h) *Dataflow execution* means that:

A: threads can execute in parallel without explicit scheduling.

B: threads must synchronize at all varible bindings.

C: an operation waits until all its arguments are bound before executing.

D: a function will always return the same value if called with the same arguments.

Comment/solution

C is correct.

i) In the following procedure, which parameter passing method is simulated?

```
proc {Increment A}
   B={NewCell @A}
in
   B:=@B+1
   A:=@B
end
```

A: Call by reference

B: Call by variable

C: Call by value

D: Call by value-result

Comment/solution

D is correct.

j) Which statement about explicit state is wrong?

A: Explicit state removes some limitations of declarative programming

B: An explicit state in a procedure is a state whose lifetime extends over more than one procedure call without being present in the procedure's arguments

C: Oz does not directly support explicit state

D: A component programmed with explicit state gives the component a sense of long time memory

E: Abstract Data Types can be written without explicit state, but these ADTs can not be modified after thay have been created

Comment/solution

C is the right answer.

k) Consider the following buffer.

```
declare

class A
   meth init skip end
   meth which
      {Browse a}
   end
end

class B from A
   meth which
      {Browse b}
   end
end

X = {New B init}
{X which}
```

If the buffer was fed, what would show in the browser and what type of binding is used for the method invocation?

A: a, dynamic binding

B: a, static binding

C: b, dynamic binding

D: b, static binding

Comment/solution

As the task is written we think none of the answers are correct, for the types of binding don't come into play when the method calls are recursive within the object. The code should have been like this:

```
declare

class A
   meth init skip end
   meth callme {self which} end
   meth which {Browse a}
   end
end

class B from A
   meth which {Browse b} end
end

X = {New B init}
{X callme}
```

For this code, C is correct.

l) Which statement about multiple inheritance is wrong?

A: Multiple inheritance may be problematic if the superclasses have a common ancestor class with attributes.

B: A programming language with direct multiple inheritance must be specially adapted to handle inheritance conflicts.

C: Multiple inheritance works better when highly different abstractions are combined than when abstractions with concepts in common are combined.

D: Multiple inheritance can only be done from generic superclasses.

Comment/solution

D is the right answer.

m) *Ambiguous* grammmars ...

A: are impossible to parse with.

B: are often used in natural language processsing.

C: does not define languages but language families.

D: cannot be written with BNF syntax.

Comment/solution

B is correct.

n) Which of the following features are supported in the relational computation model but not in pure Prolog?

A: Higher-order programming.

B: Non-determinism.

C: Operational semantics of logic programs.

D: Explicit state.

Comment/solution

A is correct.

o) How does the textbook classify the Java programming language in terms of computation models?

A: Eager, stateful, non-concurrent.

B: Eager, non-stateful, concurrent.

C: Eager, stateful, concurrent.

D: Lazy, stateful, concurrent.

Comment/solution

C is correct.

Part 2 Loops

a) Write a procedure {`For I1 I2 I3 P`} that applies the unary procedure P to integers from I1 to I2 proceeding in steps of size I3. For example,

```
{For 1 11 3 Browse}
```

should display the numbers 1, 4, 7, and 10 in the browser window, whereas

```
{For 11 1 ~3 Browse}
```

should display the numbers 11, 8, 5, and 2. (The description above is taken from the documentation of the Oz loop module. You may of course not use that module in your implementation.)

Comment/solution

```
declare

proc {For X Y Z F}
        proc {For2 X Comp Op}
                if {Comp X Y} then skip
                else {F X} {For2 {Op X Z} Comp Op}
                end
        end
        Comp = if X<Y then Value.'>' else Value.'<' end
        Op = if X<Y then Number.'+' else Number.'-' end in
        if Z<1then raise domainError end
        else {For2 X Comp Op}
        end
end
```

De som ikke vet om Value- og Numbermodulen kan skrive sine egne enkle funskjoner for pluss, minus, etc.

b) Suppose we wanted to introduce this kind of `for` loops in the declarative kernel language. Invent a suitable syntax for this and define it using BNF.

Comment/solution

```
<for-stmt> ::= for (ident) (ident) (ident) (ident) end
```

where (ident) follows the variable identifier syntax of Oz.

c) Write structural operational rules defining the behaviour of these `for` loops using the notation of chapter 13 of the textbook.

Comment/solution

```
Et forslag er:

Regel 1 (avslutting av løkka):

for x y z p end || skip
-----------------------
phi             || phi

if phi |= x=a ^ y=b ^ z=c where a,b and c are integers
and abs(a-b)<c

Regel 2 (fortsettelse av løkka):

for x y z p end || for x' y z p end
-----------------------------------
phi             || phi'
```

```
if phi |= x=a ^ y=b ^ z=c ^ x'=a' where a,b and c are integers
and abs(a-b)>=c and

{p x} || skip
------------- and
phi   || phi'

a' = a - c*abs(a-b)/(a-b)
```

Task 3

Part 3 Word lists

a) Write a function `HasWord` in Oz that takes a word and a list of words as input and returns `true` if the word is in the list and `false` otherwise. The words are represented by atoms. For example, the call {`HasWord [gabi hans fritz anna] anna`} should return `true`.

Comment/solution

```
declare
fun {Finn W L}
        case L
        of nil then false
        [] X|Lr then
                if X==W then true else {Finn W Lr} end
        end
end
Ln=[hans fritz gabi anna]
{Browse {Finn hvrt L1}}
```

b) We are going to make something that might resemble a search engine: Using the function you wrote in a), write a function `Search` that takes a word and two word lists as input, starts two threads, one that searches for the word in the first list and one that searches for it in the second list, and returns 1 or 2 depending on which list has the word. You may assume that the word always exists in one but not both lists.

Comment/solution

```
declare
fun {Srch Q L1 L2}  W R in
        thread
                if {Finn W L1} then R=1 end
        end
        thread
                if {Finn W L2} then R=2 end
        end
        W=Q
        R
end
Lv=[geht steht fahrt hvrt]
{Browse {Srch steht Ln Lv}}
```

c) We are going to change the search function so that we don't have to give the lists as input each time, but without having the lists built anew for each function call. Bind the identifier `Search` to a `local` statement that defines the word lists and returns the search function.

The search function will thus be bound to local word lists. The search function must use threads as in b) above.

Comment/solution

```
declare
Srch=local
                  L1=[hans fritz gabi anna]
                  L2=[geht steht fahrt hvrt]
          in
                  fun {$ Q}  W R in
                          thread
                                  if {Finn W L1} then R=1 end
                          end
                          thread
                                  if {Finn W L2} then R=2 end
                          end
                          W=Q
                          R
                  end
          end
{Browse {Srch steht}}
{Browse {Srch anna}}
```

Part 4 Paradigms

Bondesjakk is a game where two players take turns in putting tokens on a grid. A player wins when he manages to get five consecutive tokens on the grid. Suppose you were given the task of creating a program that can play Bondesjakk against a human player. The program must have a graphical user interface for communicating with the player, and it must be able to select relatively good moves so that the game won't be too easy for the human.

Comment/solution

It is hard to determine the correctness of any any answers to these subtasks. The important thing is that the argumentation makes sense and is not obviously wrong.

a) Write a short and concise evaluation of the usefulness of the following paradigms for this problem. Do not write more than one page.

- Imperative programming
- Functional programming
- Object oriented programming
- Logic programming

Comment/solution

One starting point might be:

- The absence of high-level abstractions in IP could make performance slightly better.
- It is difficult to write a GUI in FP because of the lack of explicit state. Declarativeness of functions makes is easy to modularize and incrementally develop the program.
- As explained by the textbook, OO is well-suited for modelling GUIs. In addition, encapsulation in classes could make modularization easier should the program become large.
- The search feature of NDLP makes it possible to solve the problem of selecting moves with very little code. This could be wasteful performance-wise, but, depending on the requirements, that may not be imortant. Lack of explicit state makes GUI programming difficult.

b) It is possible to combine elements of many paradigms in a single program. Do you think this is a good or a bad idea in general? Give reasons for your answer.

Comment/solution

One way to argue that it is a good idea is to claim that there is no big disadvantage of combining paradigms (especially when using a language that supports multiple paradigms), and that combining paradigms can make it possible to cover up the weaknesses and combine the strenghts of the individual paradigms. For example one can put an object-oriented GUI in an otherwise functional programs.

Part 5 Parsing

You are going to write a recursive descent parser in Oz that recognizes the following grammar:

```
<prog>      ::= (nothing) | <prog> '->' <stmt>
<stmt>      ::= <asg> | <loop>
<asg>       ::= <name> ':=' <value>
<loop>      ::= 'where' <bool expr> 'do' <stmt>
<bool expr> ::= <name> '<' <value>
```

`(nothing)` is a special token that stands for the empty string. Assume that `'->'`, `':='`, `'where'`, `'do'` and `'<'` are reduced to atoms by a lexical analysator that has also made tuples `name(L)` for `<name>` and `val(V)` for `<value>`, where `L` is a list of the characters in the name and `V` is an integer equal to the constant-value. Don't spend time on error handling. The built-in exception-raising and error handling of Mozart is sufficient. Use unification etc. when this can make the code shorter.

a) Does the grammer have a structure that is suitable for parsing by recursive-descent? Explain?

If is is not suitable, show the equivalency-transformations that must be made to obtain a suitable grammar. It is recommended that you write in EBNF notation.

Comment/solution

Grammatikken bruker venstre-rekursjon, dette me transformeres til hxyre-rekursjon. Vi kan greie oss med en transformert definisjon:

```
<prog> ::=  | '->' <stmt> <prog>
```

b) Write a function {`BoolExpr In ?Out`} that recognizes a ⟨ `bool expr` ⟩ given as the start of the first parameter, e.g. the list `[name([j o n]) '<' val(4) ...]`. The function should bind '...', i.e. the rest of the list, to the second parameter and return a tuple containing the available information in what was recognized.

Comment/solution

```
declare
fun {BoolExpr Inn ?Ut} N V in
        Inn = name(N)|'<'|val(V)|Ut
        mindreEnn(N V)
end

declare N
{Browse {BoolExpr [name([a n n]) '<' val(4711)] N}}
{Browse N}
```

c) Write corresponding functions for the rest of the grammar. Build a tuple `prog(...)` that holds the information for a whole ⟨`prog`⟩,`settLik(...)` for a whole ⟨`asg`⟩ and `lokke(...)` for a ⟨`loop`⟩ The following call:

```
{Prog
 ['->' name([o l e]) ':=' val(3)
  '->' 'where' name([j o n]) '<' val(4)
  'do' name([j o n]) ':=' val(7)]
 nil}
```

Could for example give the following result:

```
prog(settLik([o l e] 3)
     prog(lokke(mindreEnn([j o n] 4) settLik([j o n] 7)) nil))
```

Comment/solution

```
declare Loop
fun {Asg Inn ?Ut} N V in
        Inn = name(N)|':='|val(V)|Ut
        settLik(N V)
end
fun {Stmt Inn ?Ut}
        case Inn
        of name(_)|_ then {Asg Inn Ut}
        [] 'where'|R  then {Loop R Ut}
        end
end

declare
fun {Loop Inn ?Ut} B I1 S in
        B={BoolExpr Inn  'do'|I1}       % <--NB: gir komplisert unifisering
                                        %           ner Ut brukes i Bool Expr
        S={Stmt I1 Ut}
        lxkke(B S)
end

declare
fun {Prog Inn ?Ut}
        case Inn
        of '->'|I2 then S I3 R in
                S={Stmt I2 I3}
                R={Prog I3 Ut}
                prog(S R)
        else Ut=Inn nil
        end
end

declare
InnX=['->' name([o l e]) ':=' val(3)
               '->' 'where' name([j o n]) '<' val(4)
                       'do' name([j o n]) ':=' val(7)]

declare N
{Browse {Prog InnX N}}
{Browse N}
```