Contact persons under the exam:
Per Holager, tlf. 99 61 78 36
Ole Edsberg tlf. 95 28 15 86

Language: English

EXAM IN COURSE TDT4165 PROGRAMMING LANGUAGES Saturday May 15th, 2004, 0900–1300

No printed or handwritten material allowed. Approved calculator is allowed.

Read the whole exam before you start solving it. Make your answers short and precise: if the answer is inaccurate or longer than necessary, it will count against you.

Examination results published June 5th 2004.

The exam is quality-assured by course teacher and an external controller.

Per Holager (course teacher):

Håvard Engum (external controller):

Comment/solution

Del 1 Multiple choice (each subtask count 2.86%)

- This task shall be answered on a enclosed form.
- Check off only one box in each task. Tasks with more than one box checked off will give you zero points. On the enclosed form there are more answer boxes than there answer alternatives in the task. Take care not to check off for answers that you can't find in the task.
- Use a black or blue pen when answering. Make sure the cross fills the whole check-box, but also that it is contained by it. If you check off the wrong check-box, you can correct this by filling the whole check-box with ink. Make sure no white region is visible in the corrected check-box.
- Don't fold the enclosed form in any way.
- Write your student number twice.
- Don't write anything in the field marked "Eventuell ekstra ID".
- For each subtask, choose the answer you think is the most correct one.

1) Which parts does a simple compiler consist off?

   A: An interpreter and a parser.
   B: A lexical analysator/tokenizer, a parser and a code-generator.og en kodegenerator.
   C: A precompiler and a parser.
   D: A precompiler, a parser, a lexical analysator/tokenizer, and a virtual machine.

   Comment/solution

   B is correct.

2) This function uses pattern matching to find the first element in a list. Which line is correct to insert?

```
fun {First Xs}
   % Insert correct line here. <---
end
```

A: `case Xs of Xs.1 then true end`

B: `case Xs of Xs.1 then Xs.1 end`

C: `case Xs of X|_ then X end`

D: `case X of Xs.1 then X end`

Comment/solution

C is correct.

3) What would be a consequence if you change a single-assignment store in the declarative kernel-language (DKL) so it allowed for more than one assignment to a variable?

A: Higher order programming would be impossible.

B: Object-oriented programming would be impossible.

C: Programming of declarative components would be impossible.

D: Programmed components would no longer guaranteed be declarative.

Comment/solution

D is correct.

4) What happens in the declarative calculation model in Oz when a semantic sentence `(raise <x> end, E)` is executed? (In this task we use the same notation for execution state as in the interpreter exercise.)

A: Nothing. The semantic sentence is illegal.

B: The execution consists of the following actions:

- If the stack is empty, stop the execution with an error message. Else, pop the next element of the stack. If this element isn't a `catch`-sentence, stop the execution with a error message.
- let `(catch <y> then <s> end Ec)` be the `catch`-sentence that was found.
- Push `(<s>,Ec+{<y>->E(<x>)})` on the stack.

C: The execution consists of the following actions:

- If the stack is empty, stop the execution with an error message. Else, pop the next element off the stack.
  - If this element isn't a `catch`-sentence, push `(raise( <y> end, E)`, where `<y>` is the element that was popped, on the stack.
  - If this element is a `catch`-sentence `(catch <z> then <s> end Ec)`, push `(<s>,Ec+{<z>->E(<x>)})` on the stack.

D: The execution consists of the following actions:

- Elements is popped off the stack in a search for a `catch`-sentence.
  - If a `catch`-sentence `(catch <y> then <s> end Ec)` was found, pop it from that stack. Then push `(<s>,Ec+{<y>->E(<x>)})` on the stack.
  - If the stack was emptied without a `catch`-sentence being found, stop the execution with an error message.

Comment/solution

There was supposed to be a comma after `end` in the answer alternatives. D was supposed to be the correct answer. Because of the missing commas, we had to accept A as a correct answer also.

5) Which of the programming languages Java, Prolog, the declarative kernel language in Oz (DKL), and the kernel language in Oz with explicit state (KLES) uses static type checking.

A: DKL and KLES.

B: All.

C: All except Prolog.

D: Java.

Comment/solution

D is correct.

6) Study the following program:

```
local X Y Z in
   fun {X Y} Y+Z end
   Y = 2
   Z = 3
   local Y Z in
      Y = 5
      Z = 7
      {Browse {X Y}}
   end
end
```

What would be displayed in the Browser window if Oz used dynamic scope rules? (Oz uses static scope rules.)

A: 5

B: 8

C: 9

D: 12

Comment/solution

D is correct.

7) What would be displayed in the browser window after the following buffer is fed into the emacs-interface of Oz?

```
declare

fun {FoldL L F U}
   case L
   of nil then U
   [] X|L2 then
      {FoldL L2 F {F U X}}
   end
end

{Browse {FoldL [1 2 3] fun{$ X Y} Y-X end 0}}
```

A: -6

B: 2

C: 6

D:
```
%************************* syntax error ***********************
%**
%** expression at statement position
%**
%** in file ‘‘Oz’’, line 11, column 34
%** ------------------ rejected (1 error)
```

Comment/solution

B is correct.

8) Which of the following two functions will run with constant stack-size?

```
fun {F1 A B}
   if B<0 then raise domainError end
   elseif B==0 then 1
   else A*{F1 A B-1}
   end
end

fun {F2 A B C}
   if B<0 then raise domainError end
   elseif B==0 then C
   else {F2 A B-1 A*C}
   end
end
```

A: None of them

B: F1 but not F2.

C: F2 but not F1.

D: Both.

Comment/solution

C is correct.

9) Which statement about the following producer/consumer program is correct?

```
declare
fun {Produce N Limit}
   if N<Limit then
      N|{Produce N+1 Limit}
   else nil end
end
fun {Consume Xs A}
   case Xs
   of X|Xr then {Consume Xr A+X}
   [] nil then A
   end
end
local Xs S in
   thread Xs={Produce 0 500} end
   thread S={Consume Xs 0} end
   {Browse S}
end
```

A: This is a synchronous program since one of the threads is executed after the other one.

B: The variable `Xs` is shared between the threads, which makes the program inpredictable.

C: The consumer thread is synchronized with the producer-thread because the case-sentence in the consumer blocks it's execution until the next stream-element arrives.

D: This program will never terminate.

Comment/solution

C is correct.

10) Race conditions can arise in the Oz concurrency model if the following is true:

A: Automatic garbage collection is used.

B: Automatic garbage collection isn't used.

C: Explicit state is used.

D: Explicit state isn't used and lazy execution is used.

Comment/solution

C is correct.

11) Which method for parameter passing is simulated in the following procedure?

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

A: "Call by variable"

B: "Call by value"

C: "Call by value-result"

D: "Call by need"

Comment/solution

B is correct.

12) Which of the following class-hierarchies is legal?

A:
```
class AX meth m(...) ... end end
class BX meth n(...) ... end end
class A from AX meth m(...) ... end end
class B from BX end
class C from A B end
```

B:
```
class AX meth m(...) ... end end
class BX meth m(...) ... end end
class A from AX meth m(...) ... end end
class B from BX end
class C from A B end
```

C:
```
class AX meth m(...) ... end end
class BX meth m(...) ... end end
class A from AX end
class B from BX end
class C from A B end
```

D:
```
class AX meth m(...) ... end end
class BX meth n(...) ... end end
class A from AX meth n(...) ... end end
class B from BX meth o(...) ... end end
class C from A B end
```

Comment/solution

A is correct.

13) Study the following statements about pure Prolog and the relational kernel language in Oz (RKL).

- i): All programs in pure Prolog can be translated to RKL.
- ii): All programs in RKL can be translated to pure Prolog.
- iii): Prolog and RKL contains the same programming concepts and are only different in the way these concepts appear syntactically.

Which of them are true?

- A: Only i).
- B: Only ii).
- C: Only iii).
- D: All three.

Comment/solution

A is correct.

14) Study the following statements about the relational calculation model in Oz.

- i) It is possible to run several searches concurrently.
- ii) It is possible to run a search within a different search.
- iii) It is possible to encapsulate a search so that the rest of the program is protected from it.

Which of them are true?

- A: Only i).
- B: Only ii).
- C: Only iii).
- D: All three.

Comment/solution

D is correct.

Del 2 Questions (Each subtask counts 6%)

Don't write more than one page in normal type-size as answer in this part.

a) What is lazy execution? Mention two possible advantages by using lazy execution.

Comment/solution

Lazy execution is that a calculation isn't performed before the result is needed. Two possible advantages are:

- You can manage potential large or infinite data-structures.
- You can easily create a producer/consumer-situation that is driven by the consumer.

4 points for correct, short and precise definition. (-1 for imprecise answer, -1 for length, -1 to -3 for incorrectness)

For each of the two advantages: 3 points for each advantage. (-1 for lack of relevance, -1 for length, -1 to -2 for incorrectness)

b) What is a stream? Mention two things that can easily be done with streams, but not without.

Comment/solution

Oz-specific definition: A stream is a data-structure that is like a list, except for the last tuple that have a tail that is unbounded. General definition: A stream is a data-structure that represents a sequence of elements and where you can place elements in one end and collect them in the other end.

Both Oz-specific and general definition is accepted.

Some examples that can be done with streams:

- Declaratively model state that change over time.
- Declarative communication between threads.
- In Oz: easily program queues.
- In Oz: easily program right-recursion without inverting the accumulator value.

It isn't easy to argue for why something is difficult to do with *all* other techniques than streams. Students therefore get full score by giving examples for sensible use of streams.

4 points for correct, short and precise definition. (-1 for imprecise answer, -1 for length, -1 to -3 for incorrectness)

For each of the two advantages: 3 points for each advantage. (-1 for lack of relevance, -1 for length, -1 to -2 for incorrectness)

c) What is syntactically sugar? Mention an advantage or a disadvantage by using syntactically sugar.

Comment/solution

Syntactically sugar is syntactically elements that can be defined by translating the elements to existing parts of the language. The syntactic sugar doesn't implement new functions or programming techniques, but is only used to make programs shorter and/or more easy to follow.

- Advantage: Can make the program more easy to follow.
- Advantage: Shorter programs.
- Disadvantage: Makes to syntax rules more complex, and can therefore obscure the connection between syntax and semantics.
- Disadvantage: More complex parser.

Note, the students where asked for an advantage *or* a disadvantage. Therefore you don't have to mention both to get full score.

5 points for correct, short and precise definition. (-1 or -2 for imprecise answer, -1 for length, -1 to -3 for incorrectness)

5 points for relevant advantage or disadvantage (-1 for lack of relevance, -1 for length, -1 to -2 for incorrectness)

d) What is the advantage by doing error handling using exceptions, compared to calling user-defined procedures when errors arise?

Comment/solution

Possible answers

- When we call a procedure, the execution will continue right after the call-place when the procedure is finished. This is often not wanted behavior in error handling since the call probably is because the things we were doing wasn't possible. By using exceptions, the execution will continue after the exception handling instead of continue where the code went wrong. We therefore have the possibility to start the code that caused the error from the beginning or do something else. This is the wanted answer.(Max 10 points).
- Exceptions let us handle errors in an uniform manner. This is correct, but user defined procedures allows for the same. Procedures actually lets us handle errors in a more uniform manner because use can reuse the same procedure several places. You will often call a error-handling procedure from a catch-sentence. It is additionally unclear what "uniform" means.(Max 3 points).
- Exceptions lets us catch many different exceptions with one catch-sentence. This can also be done with procedures by calling the same procedure in different error-situations.
- Saves work that have to be done in explicit error handling: The errors must in both cases be handled by the user. Exceptions saves work in the manual work of clearing the stack above the next catch-sentence. That you save work is a very general answer, and would be "correct" almost regardless of the advantages of exceptions were.(Max 1 point).
- Exceptions let us delegate responsibility for error handling to the program that calls the component that raised the exception. This is smart since it might be the caller that knows what have to be done in the case of an exception. This happens for instance when the Mozart-system raises a system-exception or when a library-function raises an exception. This is correct and a good point, but the same can be done with procedures by making the caller sending a procedure-value for error handling as parameter to the module or function that delegates to error handling. Max 5 points since the solution with user defined procedures isn't obvious and maybe a little difficult.
- Mozart throws exceptions in case of system error and we must use exception to handle these errors: This wasn't what we had in mind, and the task wasn't specific to Oz. We must assume that if user-defined procedures is used in stead of exceptions, then this is the case for the underlying system also. The user can for instance send a error-handling procedure to the system in the startup of the program.(0 points).
- Exceptions is already implemented in the language and it is easier to use these than creating things yourself: Procedures are also supported by the language.(0 points).
- Explanation on how exceptions work: This isn't a relevant answer if you don't also include the advantages compared to user-defined procedures.

Points in this task will be the max-score for the answer given above (If several answers are given, the highest scoring of these are chosen.) -1 to -2 points for length, -1 to -5 for imprecise answer, -1 to -5 for incorrectness.

Del 3 Higher order programming (counts 6%)

Write a function `Map` that takes as arguments a list `L` and a function `F` and returns a list where each element `E` in `L` is replaced by {`F E`}.

For instance shall {`Map [1 2 3] fun`{`$ X`} `X+1 end`} return the list `[2 3 4]`.

Comment/solution

Solution:

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

Step i	Environment E_i		Store σ_i
0			
1	P1	$\longrightarrow$	x_1
2	P1	$\longrightarrow$	$x_1 \rightarrow x_2 \rightarrow$ (`proc {$ T U V} T=U+V end`,$\emptyset$)
3	X	$\longrightarrow$	x_3
	P1	$\longrightarrow$	$x_1 \rightarrow x_2 \rightarrow$ (`proc {$ T U V} T=U+V end`,$\emptyset$)
⋮	⋮		⋮
5	X	$\longrightarrow$	x_3
	P1	$\longrightarrow$	$x_1 \rightarrow x_2 \rightarrow$ (`proc {$ T U V} T=U+V end`,$\emptyset$)
	P2	$\longrightarrow$	$x_4 \rightarrow x_5 \rightarrow$ (`proc {$ T U V} T=X+V end`,{X$\rightarrow x_3$})
⋮	⋮		⋮
10	X	$\longrightarrow$	$x_3 \rightarrow x_8 \rightarrow 1$
	Y	$\longrightarrow$	$x_6 \rightarrow x_9 \rightarrow 2$
	Z	$\longrightarrow$	x_7
	P1	$\longrightarrow$	$x_1 \rightarrow x_2 \rightarrow$ (`proc {$ T U V} T=U+V end`,$\emptyset$)
	P2	$\longrightarrow$	$x_4 \rightarrow x_5 \rightarrow$ (`proc {$ T U V} T=X+V end`,{X$\rightarrow x_3$})
11	X, U	$\longrightarrow$	$x_3 \rightarrow x_8 \rightarrow 1$
	T	$\longrightarrow$	x_7
	V	$\longrightarrow$	$x_6 \rightarrow x_9 \rightarrow 2$
12			$x_3 \rightarrow x_8 \rightarrow 1$
			$x_7 \rightarrow 3$
			$x_6 \rightarrow x_9 \rightarrow 2$

Tabell 1: Sequence of execution states for the secret program.

```
Correction comment:
10 points for a 100% working program. (-1 for Oz-spesifict problem, syntax), or
5 points for an almost correct, but not working program, or
3 points for explanation of structure of the program.
```

Del 4 Semantics (counts 6%)

There exists at least one program in the declarative kernel language (DKL) that will be executed on the abstract machine in 13 execution steps

$$(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow \cdots (ST_{13}, \sigma_{13}),$$

where the execution states after some of the steps are the same as in table 1. Write such a program. (Table 1 skips each single step. Steps that are results of division of sequential composition isn't shown i Table 1 or in the enumeration that gave the number 13.)

Comment/solution

There is an error in the table. The procedure values shall still be in the store in the two last steps.

Solution:

```
local P1 in
   proc {P1 T U V} T = U+V end
   local X in
      local P2 in
         proc {P2 T U V} T = X+V end
         local Y in
```

```
            local Z in
               X = 1
               Y = 2
               skip
               {P2 Z X Y}
            end
         end
      end
   end
end
```

On the exam it was announced that it wasn't that important that the execution consists of exactly 13 steps.

10 points for a executable program that gives the correct states and 13 (+/- 1) steps. (-1 for Oz-specific problem, syntax, -3 for small deviation in states, -5 for larger deviation in states, -1 for error in execution steps larger than (+/- 1))

Students that doesn't already have max score in this task gets one bonus point if they spot the missing procedure values in the last to steps.

Del 5 Grammars and parsing (each subtask counts 6%)

In this task you shall write in Oz a recursive top-down parser for a small subset of `SELECT`-sentences in SQL defined by the following grammar:

```
<select stmt> ::= 'select' <name list> 'from' <name list> 'where' <expr>
<name list>   ::= <name> | <name> ',' <name list>
<expr>        ::= <name> '=' <value>
```

Assume that `'select'`, `'from'`, `'where'`, `','`, and `'='` have been reduced to atoms of a lexical analysator/tokenizer that also have created the tuples `name(L)` for `<name>` and `val(V)` for `<value>`, where `L` is an atom the corresponds to the name the the user typed and `V` is an integer with the same value as the constant-value.

Don't use time on error-handling. The incorporated error-handling in Oz is sufficient for this task. Use unification and similar techniques where it can make the program shorter.

a) Is the structure in the grammar suitable for a recursive top-down parser? Explain. If it isn't suitable, describe equivalence-perservative transformations that can be done to create a suitable grammar. Describe the changed definitions in EBNF-notation.

Comment/solution

`<name list>` have two alternatives that both starts with `<name>`; this have to be left-factorized. It then becomes:

```
<name list> ::= <name> [',' <name list>]
```

b) Write in Oz a function `{Expr In ?Out}` that recognizes an instance of `<expr>` from the start of the token list `In`, for instance the list `[name(jon) '=' val(4) ...]`. The function have to bind the rest of the list to `Out` and return a tuple, for instance `equal(jon 4)`, that in a sensible way incorporates the information for the part of the list that was recognized.

Comment/solution

```
declare
fun {Expr In Ut}
```

```
        case In
        of name(N)|'='|val(V)|R then Ut=R erLik(N V)
        end
end

declare N
{Browse {Expr [name(jon) '=' val(4)] N}}
{Browse N}
```

c) Write in Oz a function `{NameList In ?Out}` that recognizes an instance of `<name list>` from the start of the token list `In`, for instance the list `[name(f1) ',' name(f2) ...]`, binds the rest of the list to `Out` and returns a list of tuples, for instance `[name(f1) name(F2)]`, that in a sensible way incorporates the information from the part of the list that was recognized.

Comment/solution

There is an error in the task. It should be `[name(f1) name(f2)]`, not `[name(f1) name(F2)]`. This was announced under the exam.

```
declare
fun {NamList In Ut}
        case In
        of name(N)|R then
                case R
                of ','|R2 then navn(N)|{NamList R2 Ut}
                else Ut=R navn(N)|nil
                end
        else Ut=In nil
        end
end

declare N
{Browse {NamList [name(olav) ',' name(jens) ',' name(helge)] N}}
{Browse N}
```

d) Write in Oz a function `{SelectStmt In ?Out}` that recognizes a complete instance of `<select stmt>` from the token list `In`, for instance

```
['select' name(f1) ',' name(f2)
 'from' name(tab)
 'where' name(f1) '=' val(4) ... ],
```

binds a parameter `Out` to the rest of the token list (mark by `...`) and returns a syntax-tree that incorporates the information from the part of the token list that was recognized, for instance `select([name(f1) name(f2)] [name(tab)] equal(f1 4))`.

Comment/solution

```
declare Lx=['select' name(f1) ',' name(f2)
                'from' name(tab)
                'where' name(f1) '=' val(4)
                ]

declare
fun {Select I1 Ut} S I2 F I3 B in
        if I1.1=='select' then
                S={NamList I1.2 'from'|I2}
                F={NamList I2 'where'|I3}
                B={Expr I3 Ut}
                select(S F B)
        end
end

declare N
{Browse {Select Lx N}}
{Browse N}
```