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

Wednesday, 15. October, 2008

Language: English

Contact during the exam:

- Wacek Kuśnierczyk, Tlf 94894894

Exam aids: C. No written material allowed. An officially approved calculator is allowed.

Prepared by: Wacek Kuśnierczyk

General Comments and Hints

Read the following general comments and the rest of the exam text **before** you begin to answer.

- The exam is composed of 9 tasks, each task contains one or more questions. To achieve the maximal score, you need to correctly answer all questions in all tasks. If you skip or answer a question incorrectly, your score will be reduced correspondingly.
- Different tasks contribute differently to the total score. For each task there is a suggestion on how much time you should spend on solving the task; the times add up to 100 minutes, and correspond to the partial score for each task.
- Where you are asked to write code, the code should be written in Oz.
- Your answers should be concise, without text that is irrelevant or does not contribute to the answer, and should fit the empty space provided after each question (you do not have to fill the spaces completely). Answer directly on the question sheets, as no additional material will be accepted at the delivery.
- Each answer should include a brief explanation; rather than simply 'yes', you should answer 'yes, because ...'. Correct answers with no justification will score less than answers with justifications. Incorrect answers with coherent explanations may still give you some points.
- You may also disagree with what is stated in the pensum book or with what was explained during the lectures, but you should give convincing arguments in such cases.

Task 1 (12 minutes)

The terms 'recursion' and 'iteration' are often using in the context of computation and programming languages.

Subtask 1-1. Explain what a *recursive procedure* is.

Subtask 1-2. Explain how a *recursive process* and an *iterative process* differ.

Subtask 1-3. Does it make sense to speak of an *iterative procedure*?

It is convenient to implement a generic control abstraction for iterative processes that can then be used to implement more specific procedures.

Subtask 1-4. Show how to do it by completing the following template.

```
proc {Iterate State Final Update}




end
```

Task 2 (18 minutes)

The function `Sum` takes as input a list of numbers and returns a single number that equals to the sum of all elements of the list.

The function `RunningSum` is somewhat different: it takes as input a list of numbers and returns a *list of numbers* of the same length as the input list, and such that the `i`-th number on the output list equals the sum of the first `i` elements from the input list.

```
{Sum [1 2 3]}
% evaluates to 1+2+3 = 6

{RunningSum [1 2 3]}
% evaluates to [1 1+2 1+2+3] = [1 3 6]
```

Subtask 2-1. Show how to implement `RunningSum` using an internal recursive helper function.

```
fun {RunningSum Numbers}
   fun {RunningSum                }
      case
      end
   end
in
   {RunningSum                    }
end
```

Subtask 2-2. Show how to implement `RunningSum` using `Iterate`.

```
fun {RunningSum Numbers}


   {Iterate
   }
in


end
```

Hint: you may use a difference list to maintain the sums computed so far.

Task 3 (8 minutes)

Procedural abstraction is one of the key ingredients of efficient programming, and is essential in higher-order programming.

Subtask 3-1. Name and characterise four major principles of higher-order programming.

(a)

(b)

(c)

(d)

Task 4 (10 minutes)

In the new, forthcoming standard for the programming language C++ (the so-called 'C++0x' specification) it will be possible to create anonymous function values (lambda functions, lambdas):

```
auto inc = [n](int x) { return x+n; }
auto inx = [&x](int n) { x += n; }
```

Here, `inc` is a function that takes an integer and returns the integer increased by `n` (which may be another integer, a float, or whatever else `n` happens to be when `inc` is defined); `inx` is a procedure that takes an integer and increases the variable `x` (whatever `x` happens to be when `incx` is defined) with the value of that integer. The two square brackets ('`[`' and '`]`') denote a set of variables to be included in the *closure environments* of `inc` and `inx`.

Subtask 4-1. What is a closure?

Subtask 4-2. What is the purpose of a closure environment?

In C++0x, it will be possible to explicitly specify what a closure environment will contain, and how the values can be accessed; in the example above, the closure environment of `inc` contains the value of the variable `n`, while the closure environment of `inx` contains a reference to the variable `x`.

Subtask 4-3. When are closure environments created in Oz? What is the rule for deciding what such an environment will contain?

Task 5 (10 minutes)

An abstract data type (ADT) specifies, in abstract terms, the operations that can be performed on instances of the ADT.

Subtask 5-1. Describe three major binary criteria used to classify the ways an ADT can be implemented.

(a)

(b)

(c)

It is possible to imagine eight ways of implementing an ADT, each characterised by a different combination of outcomes for the three criteria above (hopefully you got them right).

Subtask 5-2. During the course, we have discussed only four ways of implementing an ADT (we do not yet have the means for the other four versions). Characterize these four ways with respect to their *usefulness* in programming.

(a)

(b)

(c)

(d)

Task 6 (8 minutes)

Oz, as most programming languages today, is lexically scoped, while a few other languages are dynamically scoped.

Subtask 6-1. Explain the terms 'lexical scope' and 'dynamic scope'.

lexical scope:

dynamic scope:

Imagine that Oz were *dynamically scoped*. Consider the following piece of code:

```
local P in
   local X = 1 in
      P = proc {$} Y = 2 in {Browse X+Y} end
   end
   local X = 3 Y = 4 in
      {P}
   end
end
```

Subtask 6-2. What would you expect to be displayed if the code above were executed, and why?

Task 7 (12 minutes)

The following code is supposed to generate and display an infinite stream `s` of streams such that the `i`-th stream in `s` contains an infinite sequence of integers starting at `i` and increasing stepwise by `i`.

```
fun {Integers}
   fun {Integers Start Step}
      Start|{Integers Start+Step Step}
   end
in
   {Map {Integers 1 1}
    fun {$ Start}
       {Generate Start Start}
    end}
end

{Browse {Integers}}
% supposed to display [[1 2 3 ...] [2 4 6 ...] [3 6 9 ...] ...]
```

(The outer stream and its inner stream components are potentially infinite, and the actual output would rather be like `(1|2|3|,,,)|(2|4|6|,,,)|(3|6|9|,,,)|,,,`; this does not matter for the questions below.)

Unfortunately, this does not work: there will be no output printed, the browser window will not even be opened.

Subtask 7-1. Explain why the code does not work as desired.

Subtask 7-2. Suggest two ways in which the code can be improved so that there would be an output, as desired.

(a)

(b)

Subtask 7-3. Implement one of your suggestions using the template below. Make as few modifications to the original code as possible.

```
fun {Integers}

in

end
```

Subtask 7-4. What is the minimal number of modifications that must be made for the code to work as desired? Explain why fewer modifications won't help.

Task 8 (10 minutes)

Programming languages can be divided, roughly, into those *interpreted* and those *compiled* (there are also those which can be both interpreted and compiled, and there are many variations on the theme).

Subtask 8-1. What does the term 'interpretation' mean?

Subtask 8-2. What does the term 'compilation' mean?

Interpretation and compilation are processes composed of a number of steps; some of the steps are common to both these processes.

Subtask 8-3. Describe at least three steps common to interpretation and compilation of programs.

Task 9 (12 minutes)

Consider the following trivial EBNF grammar, which might be a part of a larger grammar for some unspecified programming language.

⟨*statement*⟩ ::= `NOOP` | ⟨*if*⟩
⟨*if*⟩ ::= `IF` ⟨*expression*⟩ `THEN` ⟨*statement*⟩ [⟨*else*⟩]
⟨*expression*⟩ ::= `YES` | `NO`
⟨*else*⟩ ::= `ELSE` ⟨*statement*⟩

The square brackets ('[' and ']') denote an optional element (one that may appear once or not at all), and are part of the EBNF syntax and not the syntax specified by the grammar.

Subtask 9-1. What does it mean for a grammar to be *context-free*?

Subtask 9-2. Is the grammar above context-free?

Subtask 9-3. What does it mean for a grammar to be *unambiguous*?

Subtask 9-4. Is the grammar above unambiguous? If not, give a counterexample.

— End —

If you have any comments, questions, etc., write them here.