NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Computer and Information Science

Examination paper for **TDT4165 Programming Languages**

Grading aid

Academic contact during examination: Lars Bungum

Phone: 9204 6135

Examination date: December 11th, 2014

Examination time (from–to): 09:00–13:00

Permitted examination support material: C: No (hand)written aids allowed. Only specified, simple calculators.

Other information:
This exam set was approved by Øystein Nytrø.

Language: English

Number of pages: 12

Number pages enclosed: 0

Checked by:

Date Signature

Part 1: Programming Language Fundamentals (30%)

Problem 1 The difference between *declarative* and other computational models is central in the textbook Concepts, Techniques and Models of Computer Programming (CTMCP) by Seif Haridi and Peter van Roy.

a) What does it mean that a program component is *declarative*?

$\boxed{\textit{Grading aid}}$ A program component is *declarative* iff it always produces the same output for the same input. △

b) What does it mean that a computational model is declarative?

$\boxed{\textit{Grading aid}}$ A model of computation is declarative iff all programs executed in this model are guaranteed to be declarative. △

c) What is a concurrent model of computation?

$\boxed{\textit{Grading aid}}$ Concurrency lets a program be organized into parts that execute independently and interact only when needed. △

d) Is it possible for a concurrent model of computation to be declarative? If so, how is this possible? Explain.

$\boxed{\textit{Grading aid}}$ For a concurrent model to be declarative, it must fulfill exactly one of the following conditions:

- All executions of the program must have the same result (i.e., logically equivalent)
- All executions must lead to a failure

△

e) What are *syntactic sugar* and *linguistic abstractions*? Give examples of them in Oz showing the difference between code in the practical and the kernel language.

Grading aid A *linguistic abstraction* is an abstraction that introduces an addition to the language syntax, such as a *for* or a *fun* construct.

syntactic sugar are shorthand notations for frequently occurring idioms, such as implicit variable declarations that are not possible in the kernel language. △

f) What is the difference between a *statement* and an *expression* in Oz? Give illustrative examples.

Grading aid An expression is syntactic sugar for a sequence of operations that returns a value. It is different from a statement, which is also a sequence of operations, but does not return value. An expression can be used inside a statement whenever a value is needed. For example, $22 * 22$ is an expression and $X = 22 * 22$ is a statement. △

g) Why do we use *syntactic sugar* and *linguistic abstractions*? Explain.

Grading aid It makes programming more easy. Hence the practical language is called that, because it is more practical. △

h) What is *last call optimization* and in what situations is it beneficial?

Grading aid A technique that allows a tail-recursive procedure to return immediately instead of going back through a sequence of stacked call frames. I.e., the reason tail-recursion can save space. △

i) You are contacted by a mathematician specializing in combinatorics whose calculator is no longer able to compute factorials. Implement a recursive procedure that computes the factorial of n, i.e., the factorial of $5 = 5 * 4 * 3 * 2 * 1 = 120$.

Grading aid

```
declare
  fun {Factorial N}
     if N < 2 then 1
       else N * {Factorial N-1}
     end
  end
{Browse {Factorial 5}}
```

△

j) The mathematician returns. While she is enthused by the ability to calculate large factorials she is exhausted at manually calculating the formula $\binom{n}{k}$ (pronounced: *n choose k*) that select members from a grouping without taking the order of choosing into account and returns their number. The mathematician generously reminds you of this nice way of calculating the formula: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, so if you have three fruits, apples oranges and pears, and can choose only two for a snack, you are left with the set of choices $\{\{apple, orange\}, \{apple, pear\}, \{orange, pear\}\}$ which can be calculated with the above formula, i.e., $\frac{3*2*1}{2*1*(3-2)} = 3$.

Implement a function that calculates this number for any $n \geq k$ where $n > 1$.

Grading aid

```
declare
fun {NoverK N K}
   local Denom = {Factorial K} * {Factorial N-K} in
      {Int.'div' {Factorial N} Denom}
   end
end
{Browse {NoverK 10 8}}
```

△

k) You realize that the operator / expects floats while you are dealing with integers. To demonstrate your programming prowess to the mathematician you implement integer division for use in the previous question. I.e., implement integer division in Oz.

(In the event that this is unsuccessful you are allowed to assume there exists such a function/procedure in the previous question).

l) The mathematician computes larger and larger numbers. You are worried about the strain this puts on your hardware resources, and realize you need to implement an *iterative* version of the factorial function to mitigate this concern. Implement an iterative version of *factorial.*

Grading aid

```
declare
fun {Factorial N}
   fun {Iterate Iterator Result}
```

```
      if Iterator > N then Result
      else {Iterate Iterator+1 Iterator*Result}
      end
   end in
   {Iterate 2 1}
end
{Browse {Factorial 5}}
```

△

m) How are computer programs processed? Sketch the steps that are taken from a string of characters are input.

Grading aid

- The initial input is linear – it is a sequence of symbols from the alphabet of characters.
- A lexical analyzer (scanner, lexer, tokenizer) reads the sequence of characters and outputs a sequence of tokens.
- A parser reads a sequence of tokens and outputs a structured (typically non–linear) internal representation of the program a syntax tree (parsetree).
- The syntax tree is further processed, e.g., by an interpreter or by a compiler.

△

n) Explain the difference between *left-folding* and *right-folding*. Use a higher-order procedure that does right-folding to compute the sum of the squared elements of a list.

Grading aid *right-folding* traverses the lists from left to right and does the combination backwards, i.e., from right to left.

```
   {Sum [1 2 3]}
== 1  + {Sum [2 3]}
== 1  + (2 + {Sum [3]})
== 1  + (2 + (3))
```

an implementation of the pattern:

```
 fun {FoldRight List Null Transform Combine}
  case List of nil then Null
  [] Head|Tail then
    {Combine {Transform Head}
      {FoldRight Tail Null Transform Combine}} end end
```

FoldLeft combines elements as it strips them off from the input list, using Iterate to implement an iterative process.

```
 fun {FoldLeft List Null Transform Combine}
{Iterate List#Null
fun {$ List#_} List == nil end
fun {$ (Head|Tail)#Result}
Tail#{Combine Result {Tranform Head}} end}.2 end
```

△

Part 2: Formal Grammars and Parsing (20%)

Problem 2

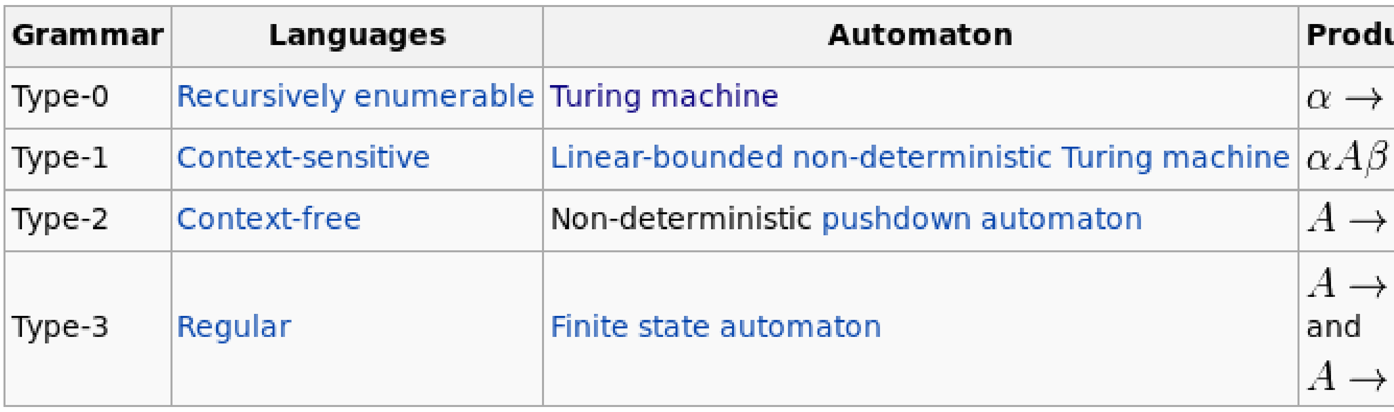
a) What is the Chomsky hierarchy of formal grammars? List the types of grammars, and show for each type what constraints there are on the production rules.

Grading aid

Noam Chomsky defined four classes of languages:

- Type 0: Unconstrained Languages
- Type 1: Context-Sensitive Languages
- Type 2: Context-Free Languages
- Type 3: Regular Languages

△



Grammar	Languages	Automaton	Produ
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow$
Type-3	Regular	Finite state automaton	$A \rightarrow$ and $A \rightarrow$

Figure 1: Chomsky Hierarchy

b) Write a small, context-free grammar in EBNF format that is ambiguous. Show by means of syntax trees that your grammar is indeed ambiguous for the same input string.

Grading aid

△

c) How does a recursive-descent parser for a grammar work? What is meant by *descent*?

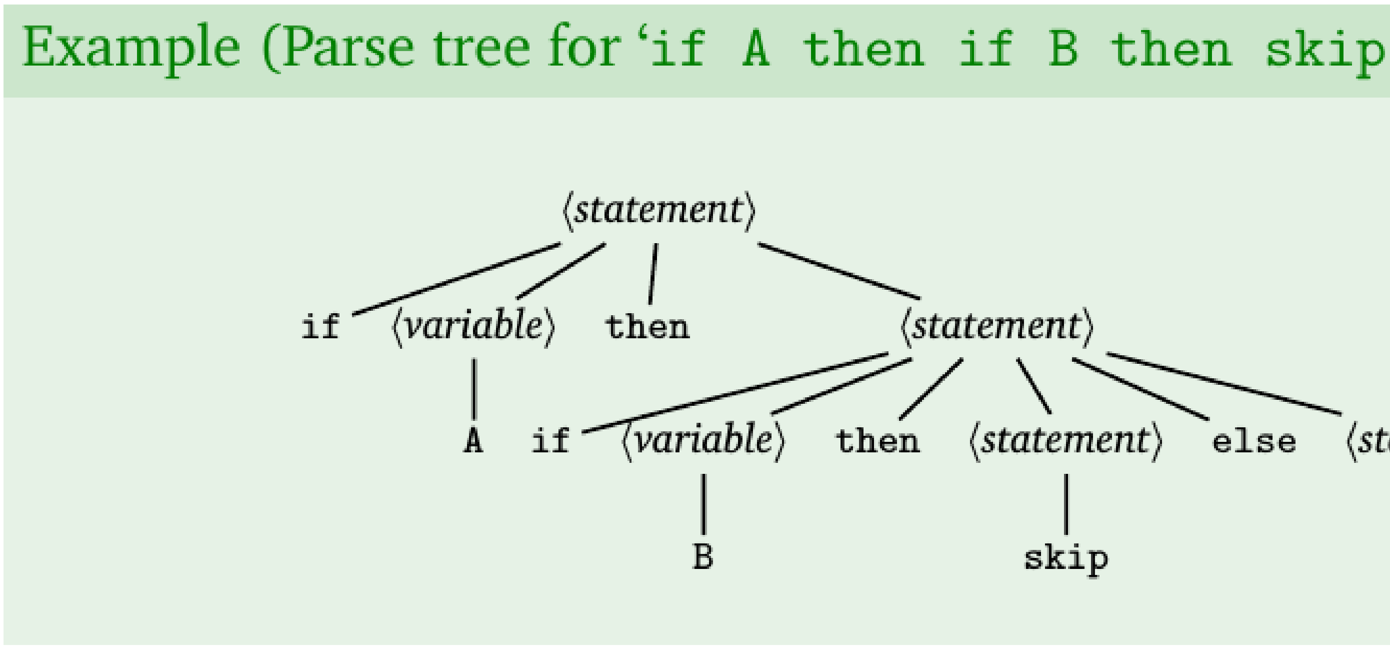
Grading aid In computer science, a recursive descent parser is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes. △

d) Consider this simple grammar:

```
<A> ::= a<A>y | g<B>g<B>
<B> ::= p<B>q | epsilon
```

Write a recursive-descent *recognizer* for this grammar, i.e., a program at either accepts or does not accept the input string as part of the language defined by the grammar.

Grading aid



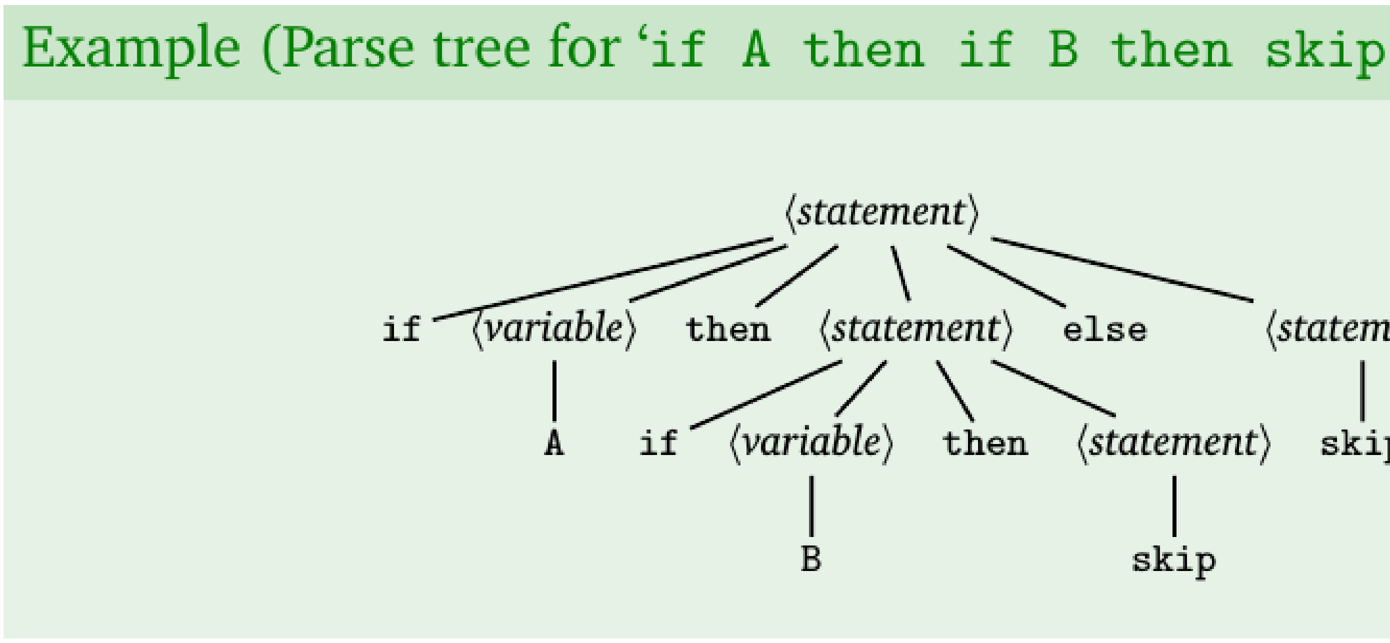


Figure 2: Ambiguous grammar examples

```
local
   fun {InLanguage? Tokens}
      try {A Tokens nil} catch _ then false end
   end
   fun {A Tok RemTok}
      case Tok
      of a|RTok then {A RTok y|RemTok}
      [] g|RTok then RTok2 in {B RTok g|RTok2} andthen {B RTok2 RemTok}
      else false
      end
   end
   fun {B Tok RemTok}
      case Tok
      of p|RTok then {B RTok q|RemTok}
      else RemTok = Tok true
      end
   end
in
   {Show {InLanguage? [a g p q g y] }} %% Insert own string here!
```

△

end

Part 3: Extensions to the Declarative Model(30%)

Problem 3

a) What non-declarative models of computation are available in Oz? Why are they not declarative?

Grading aid

- N Names
- Exception handling and threads
- Explicit state

Explicit state prevents declarativity. △

b) What extensions to the Oz's abstract machine are necessary to facilitate non-declarative message-passing concurrency with ports? What problems may arise if we don't have these extensions to our model?

Grading aid In order to use ports, the abstract machine must be extended with a mutable store to keep track of the ports. Using in this non-declarative way allows us to send more messages to the same port, without having to return a new port object (it is mutable).

To program everything declaratively, we would have to return new objects all the time in order to have many threads produce content for the same stream. Having more threads write to the same stream would lead to unification errors in that case. △

c) What are *implicit* and *explicit* state? Implement `quicksort` using both of these methods.

A quick referesher on `quicksort`. The description from the lecture:

> Given a collection of items to be sorted, pick an item (the "pivot") and divide the remaining items into those smaller and those larger than the pivot, quick-sort the two sub-collections, and concatenate them with the pivot in the middle.

An illustration of the algorithm from Wikipedia is shown in Figure 3.

Grading aid Implicit state:

```
local QS in
  fun {QS List}
      fun {Partition List}
local P B A T in
   P|T = List
   B = {Left T P}
   A = {Right T P}
   B#P#A
end
      end
  in
      case List of nil then nil else
local B P A in
```

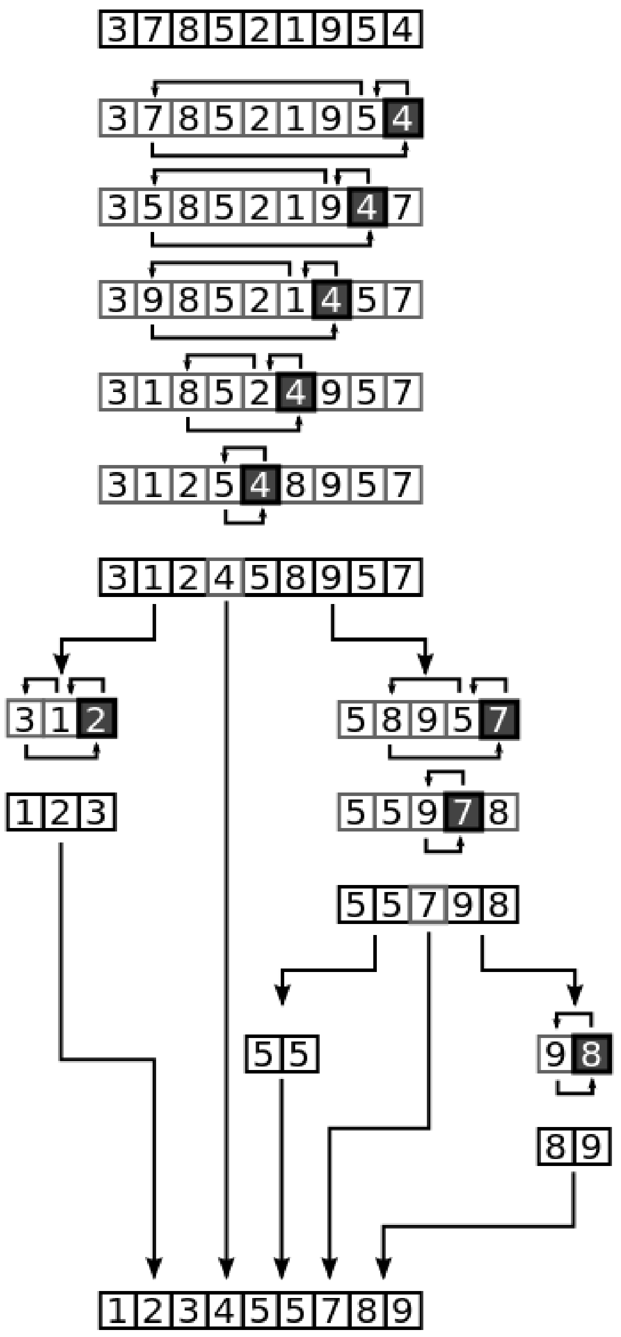


Figure 3: Example run of `quicksort`

```
      B#P#A = {Partition List}
      {Append {QS B} P|{QS A}}
  end
        end
    end
    {Browse {QS {Randoms 1000}}}
end
```

△

d) Why would we want to program with a) *mutable* and b *immutable* state? What are the advantages of both methods?

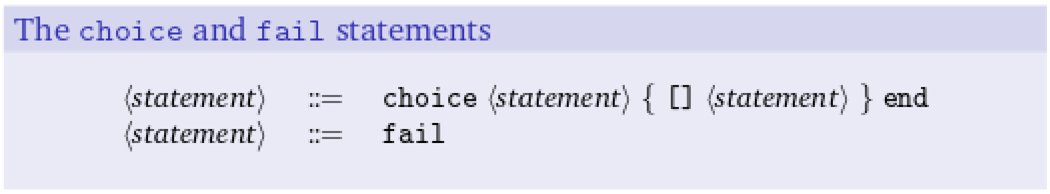
Grading aid Why would we prefer programming with mutable state to programming with immutable state?

- Because the underlying machine performs computations with mutable state.
- Because non-declarative implementations tend to be faster than declarative implementations.
- Because most algorithms are explained in terms of updating the state of a data structure (e.g., an array).
- Because the reality is stateful and modeling with mutable state is often more appropriate.

Why would we prefer programming with immutable state to programming with mutable state?

- Because solutions to many problems are more intuitive in the declarative form than in the non-declarative form; the translation from an abstract algorithm to an implementation is usually easier in the declarative approach.
- Because declarative programming is safer than non-declarative programming, particularly in a concurrent environment.
- Because in languages with a high level of abstraction declarative techniques are often more efficient than explicit operations on mutable state. Of course, the declarative techniques do use mutable state, but that's invisible to the programmer in the end, declarative programming on a stateful machine boils down to obs

△



The `choice` and `fail` statements

```
⟨statement⟩  ::=  choice ⟨statement⟩ { [] ⟨statement⟩ } end
⟨statement⟩  ::=  fail
```

Figure 4: Relantional Programming Extensions

Part 4: Relational Programming (20%)

Problem 4

a) What is relational programming, and how does it work? Explain the main characteristics in a few paragraphs at most.

Grading aid Relational programming is an approach to programming where procedures are thought of as relations between values, and where no clear distinction between input and output is made. △

b) What extensions to the syntax of the declarative model in Oz are needed to do relational programming?

Grading aid

△

c) Given that you have a *solver* readily available, write a relational program that outputs two vectors with three elements, whose dot product sums up to the parameter x that you give to the function. The dot product is the sum of the three elements with the same index multiplied with each other, i.e., $3 * 3 + 3 * 3 + 3 * 3$ in the sample below.

Remember that you have to wrap the arguments like in this example:

```
{Browse {SolveAll fun {$} {InnerProd 27} end}}
  [[3 3 3]#[3 3 3] ]
```