NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Computer and Information Science

Examination paper for **TD4165 Programming languages**

Advisory contact during examination

Teacher: Waclaw Kusnierczyk
Mobile: +47 94814820

Examination

Date: 6th December, 2016
Time: 09:00–13:00

Permitted examination support material:

- C: Specified (nothing) printed and hand-written support material is allowed.
- A specific basic calculator is allowed.

Language

- 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).

Content

The exam is composed of 11 tasks. Every task has the same weight towards the final score. Short, succinct answers preferred. Explain all assumptions you are making while providing answers.

This document contains 6 pages with questions and tasks (7 pages in total including this front page, not included in numbering).

Task 1

Write a complete BNF grammar defining the syntactic structure of an Oz expression specifying a record. Make sure your grammar covers the extended (practical) language, where the fields of a record may be given arbitrary values, not only variable identifiers.

For example,

```
some(cost:100
     name:Whoknows
     record:with(true:true
                 false:false))
```

Suggestions and constraints:

- Be careful about clearly distinguishing the syntactic elements of the defining language (BNF) and the defined language (Oz).
- Assume that values can be identifiers, atoms, numerals, and records.
- Use regular expressions to specify the microsyntax of identifiers, atoms, and numerals.
- Limit numerals to integers only.
- Regular expression-based specifications do not have to be complete relative to actual Oz syntax (i.e., you do not have to cover all possible forms; for example, it is enough if identifiers are composed of only letters and digits).
- If you are not familiar with regular expressions, make up your own notation and explain it in free text.
- You do not need to explicitly specify whitespace in your grammar rules.

Solution 1

Here is one way to specify the grammar for the structure of records:

```
    <record> ::= <atom> | <label> '(' <feature> ':' <value> {  <feature> ':' <value> }+ ')'
      <atom> ::= REGEX([a-z][a-zA-Z0-9]*)
     <label> ::= <identifier>
   <feature> ::= <identifier>
     <value> ::= <identifier> | <atom> | <integer> | <record>
<identifier> ::= REGEX([A-Z][a-zA-Z0-9]*)
   <integer> ::= REGEX([0-9]+)
```

In the grammar above:

- a record is either an atom, or has the structure of a label followed by parentheses enclosing one or more fields;
- a field is a semicolon-separated feature-value pair;
- an atom consists of a lower-case letter followed by any number of letters and digits;
- labels and features are identifiers;
- a value is an identifier, an atom, an integer, or (recursvely) a record;
- an identifier consists of an upper-case letter followed by any number of letters and digits;
- an integer is one or more digits;
- syntactic elements of the defined language are provided within single quotes (e.g., `'('`, `':'`);

- regular expressions are encapsulated within `REGEX(` and `)` (this is a custom extension—you could have done it in any other way that makes clear where regular expressions are used).

Note that the microsyntax of atoms and identifiers above is not complete wrt. the actual Oz syntax (e.g., in Oz atoms can be any sequences of non-single-quote characters embedded within single quotes).

An answer with `<label> ::= <atom>` and/or `<feature> ::= <atom>` is also taken as correct—you were not expected to remeber all specific details of Oz syntax.

Task 2

Consider the following Oz code snippets, one per line:

```
local X in X = 10 end
skip skip skip
if X == 0 then X else Y end
Record = record(field:value)
```

Draw the abstract syntax tree (AST) for each of them, based on your understanding of the grammar of the kernel Oz language.

For each snippet:

- decide whether it is valid kernel language code, and draw its AST;
- if you believe there is more than one AST, draw all of them;
- if you do not think it is valid kernel language code, explain why.

Suggestions:

- Instead of drawing an AST, you can represent it as a (potentially nested) record.

Solution 2

In this solution, all ASTs are presented as Oz records. The labels and features are arbitrary.

There is one AST for `local X in X = 10 end`:

```
local(identifier:identifier('X')
      statement:unification(left:identifier('X')
                            right:integer('10')))
```

The relevant grammar rules would be:

```
  <statement> ::= ... | <local> | ...
      <local> ::= 'local' <identifier> 'in' <statement> 'end'
<unification> ::= <identifier> '=' <value>
```

There are two ASTs for `skip skip skip`:

```
sequence(first:skip
         second:sequence(first:skip
                         second:skip))
sequence(first:sequence(first:skip
                        second:skip)
         second:skip)
```

The relevant BNF grammar rules would be:

```
<statement> ::= ... | 'skip' | <sequence> | ...
 <sequence> ::= <statement> <statement>
```

It is possible to write the grammar in such a way that `skip skip skip` would have only one AST. You were not expected to remember the exact syntax of Oz, so either answer was acceptable, given appropriate justification.

There is no AST for `if X == 0 then X else Y end`, assuming BNF grammar rule for conditional statements is:

```
  <statement> ::= ... | <conditional> | ...
<conditional> ::= 'if' <expression> 'then' <statement> 'else' <statement> 'end'
```

Neither `X` nor `Y` are statements (they are expressions).

Note that `if X == 0 then X else Y end` has an AST in the practical (extended) language, where the whole `if` construct can be either a statement, or an expression. In the latter case, both the consequent (here, `X`) and the alternative (here, `Y`) need to be expressions as well.

You are not expected to know the exact syntax of Oz, but the distinction between statements and expressions is not merely syntactic. However, if you provided a parse tree for the above with the explicit assumption of the grammar rule below, the answer is considered correct:

```
 <expression> ::= ... | <conditional> | ...
<conditional> ::= 'if' <expression> 'then' <expression> 'else' <expression> 'end'
```

There is one AST for `record(field:value)`:

```
unification(left:identifier('Record')
            right:record(label:atom('record')
                         fields:[field(feature:atom('field') value:atom('value'))])
```

The relevant BNF grammar rule would be:

```
  <statement> ::= ... | <unification> | ...
<unification> ::= <identifier> '=' <value>
      <value> ::= ... | <record> | ...
```

with rule for `<record>` as in Task 1.

Task 3

Describe the Chomsky hierarchy of languages.

- Name all four levels of formal grammars, and explain the generic structure of their rules.
- For each grammar level, write in BNF one specific example of a rule that cannot be stated in any of the less expressive grammar levels.
- What is the minimal (least complex) grammar level suitable for defining the microsyntax of a programming language?
- What is the minimal grammar level suitable for defining the (macro)syntax of a programming language?

Solution 3

The Chomsky hierarchy classifies languages according to their expressivity (in parallel with computational complexity). The form of their rules is given below, from least to most complex grammars, with `t` denoting a terminal, `n` denoting a non-terminal, and `s` denoting a (possibly empty) sequence of terminals and/or nonterminals:

- regular: `n -> t n’` (right-regular) or `n -> n’ t` (left-regular)
- context-free: `n -> s`
- context-sensitive: `s n s’ -> s s" s’`
- unrestricted: `s -> s’`

Hypothetical rule examples:

- regular: `<sequence> ::= ’skip’ <sequence>`
- context-free: `<sequence> ::= ’begin’ <sequence> ’end’`
- context-sensitive: `’begin’ <sequence> ’end’ ::= ’begin’ <statement> <sequence> ’end’`
- unrestricted: `<statement> ’skip’ <statement> ::= ’begin’ <sequence> ’end’`

The microsyntax of a programming language (the syntax of lexemes, such as identifiers, keywords, etc) can usually be expressed with a regular language (thus regular expressions are commonly used).

The macrosyntax of a programming language (the way tokens are composed into statements) can usually be expressed with a context-free grammar. All of the syntax in Oz we’ve seen at the course were expressed with context-free grammar.

Task 4

Processing lists is a central concept in functional programming.

- Implement the procedure/function `Zip` that takes as input two lists and builds a list of pairs of the respective elements of those lists.
- Implement the procedure/function `Unzip` that takes as input a list of pairs, and builds two lists of the first and second elements of those pairs, respectively.

For example,

```
{Zip [1 2 3] [4 5 6]}
% [1#4 2#5 3#6]

{Unzip [1#4 2#5 3#6]}
% [1 2 3]#[4 5 6]
```

Constraints:

- Implement `Zip` as a `fun` in the practical language.
- Implement `Unzip` as a `proc` in the kernel language.

Solution 4

Here is one possible implementation of `Zip` as a function in the practical (extended) language:

```
local Zip in
   fun {Zip List1 List2}
      case List1#List2 of (Head1|Tail1)#(Head2|Tail2) then
         (Head1#Head2)|{Zip Tail1 Tail2}
      [] nil#nil then nil
      else raise exception(cause:length) end
      end
   end
   % {Browse {Zip [1 2 3] [4 5 6]}} -> [1#4 2#5 3#6]
end
```

Here is one possible implementation of `Unzip` as a procedure in the kernel (impractical) language:

```
local Unzip in
   Unzip = proc {$ List ?Result}
              local Unzip in
                 Unzip = proc {$ List Result1 Result2}
                    case List of '|'(1:Head 2:Tail) then
                       case Head of '#'(1:Item1 2:Item2) then
                          local Rest1 in
                             local Rest2 in
                                Result1 = '|'(1:Item1 2:Rest1)
```

```
                              Result2 = '|'(1:Item2 2:Rest2)
                              {Unzip Tail Rest1 Rest2}
                           end
                        end
                     else
                        raise exception(cause:format) end
                     end
                  else
                     Result1 = nil
                     Result2 = nil
                  end
               end
            end
         end
   local Result1 in
      local Result2 in
         {Unzip List Result1 Result2}
         Result = Result1#Result2
      end
   end
   % local Result in {Unzip [1#4 2#5 3#6] Result} {Browse Result} -> [1 2 3]#[4 5 6]
end
```

Both implementations raise an exception if the input is not of the correct format (e.g., lists of unequal length). A solution based on the assumption that the input lists are of equal length is correct as well, provided you made clear the assumptions you make.

Task 5

Implement the function `Splitter` that takes as input a one-argument boolean function `Predicate` and returns a record with an arbitrary label and three fields:

- `put`, which is a procedure of one argument;
- `true` and `false`, which are streams.

For example,

```
Positive = {Splitter fun {$ Number} Number > 0 end}
% positive(put:<procedure> true:<stream> false:<stream>)
```

Calling `put` with a value results in the value being placed onto exactly one of the two streams. If an application of `Predicate` to the value given to `put` evaluates to true, the value is placed on the stream `true`, otherwise it is placed on the stream `false`.

For example,

```
{Positive.put -5}
{Positive.put 10}
{Positive.put 0}

Positive.true#Positive.false
% (10|_)#(-5|0|_)
```

Note that neither `true` nor `false` are terminated with `nil` (they are not proper lists).

Questions:

- Can this be done using the declarative subset of Oz only? Why?
- Consider using ports. Is it possible to provide an implementation using one port only? Why?
- Can this be done using the sequential subset of Oz only? Why?

Solution 5

To be able to use the same procedure (e.g., `Positive.put`) to append multiple items to one or more streams, some form of mutable state is needed to keep track of their unbound ends. The declarative subset of Oz offers no way to handle mutable state, hence is not sufficient to implement `Splitter`.

The limited mutable state provided by ports is sufficient to implement `Splitter`. It can be done with one, two, or three ports. (Or more than three, uselessly.)

Here is one possible implementation of `Splitter` with one port:

```
fun {Splitter Predicate}
   Input True False
   InputPort = {NewPort Input}
   proc {Process Head|Tail True False}
      NewTail in
```

```
      if {Predicate Head} then
         True = Head|!!NewTail
         {Process Tail NewTail False}
      else
         False = Head|!!NewTail
         {Process Tail True NewTail}
      end
   end
   proc {Put Message}
      {Send InputPort Message}
   end
in
   thread {Process Input True False} end
   splitter(put:Put true:!!True false:!!False}
end
```

Note that the unbound tails exposed to outsde of the object need to be write-protected (read-only).

Here is one possible implementation of `Splitter` with two ports:

```
fun {Splitter Predicate}
   True False
   TruePort = {NewPort True}
   FalsePort = {NewPort False}
   proc {Put Message}
      {Send if {Predicate Message} then TruePort else FalsePort end
            Message}
   end
in
   splitter(put:Put true:True false:False)
end
```

Note that here `True` and `False` are read-only because of the association with ports.

Here is one possible implementation of `Splitter` with three ports:

```
fun {Splitter Predicate}
   Input True False
   InputPort = {NewPort Input}
   TruePort = {NewPort True}
   FalsePort = {NewPort False}
   proc {Process Head|Tail}
      if {Predicate Head} then
         {Send TruePort Head}
      else
         {Send FalsePort Head}
      end
      {Process Tail}
   end
   proc {Put Message}
      {Send InputPort Message}
   end
```

```
in
   thread {Process Input} end
   splitter(put:Put true:True false:False)
end
```

Ports provide a bridge to the mutable state model by offering a very constrained form of mutable state; nevertheless, mutable state is in use.

It is possible to implement `Splitter` in the sequentual model of computation. Unlike the one-port and the three-port solutions above, the two-port solution, arguably the simplest, does not need a separate background thread to avoid freezing over the unbound end of the input stream.

With explicit mutable state (in Oz, cells; not part of pensum), it is possible to implement `Splitter` in the sequential model of computation (without ports). Here is one possible implementation:

```
fun {Splitter Predicate}
   True = {NewCell _}
   False = {NewCell _}
   proc {Put Message}
      Tail in
      if {Predicate Message} then
         @True = Message|!!Tail
         True := Tail
      else
         @False = Message|!!Tail
         False := Tail
      end
   end
in
   splitter(put:Put true:!!@True false:!!@False)
end
```

Consult Oz documentation for the meaning of `@` and `:=`.

Task 6

Declarativeness and concurency are two major concepts discussed in the course textbook.

- What is declarativeness?
- What are the benefits of writing programs in a declarative language?
- What are the drawbacks of writing programs in a declarative language?
- What is concurrency?
- What are the benefits of writing programs in a concurrent language?
- What are the drawbacks of writing programs in a concurrent language?
- Is it possible to combine declarativeness and concurrency (i.e., have a language that is both declarative and concurrent)?

Solution 6

Answers in this solution are based on the pensum book.

- Declarativeness is a property of a computational model such that a declarative operation always leads to the same result if executed with the same input. In a declarative model, operations are stateless (do not preserve state that might change between executions), independent of external state (their results depend only on the input), and deterministic (their results are always the same for specific input).
- The benefits of a declarative language follow from the above and include ease of reasoning about the execution, maintainability, modularity, and thread-safety in a concurrent environment.
- The drawbacks of a declarative language include impossibility of representing real-life objects (which typically are mutable) in a natural way, and increased use of resources (reproduction rather than modification of data structures).
- Concurrency is a feature of a computation model that allows more than one computation to happen at the same time, e.g., by means of threads (typically communicating through shared state) and processes (typically communicating by messages).
- The benefits of a concurrent language include potential to improve overall computation time, avoiding the risk of computation getting stale while waiting for input, and more efficient use of resources (e.g., multiple processors).
- The drawbacks of a concurrent language include the difficulty of writing, testing, and debugging, increased risk of errors (e.g., inconsistent state due to concurrent updates), race conditions and deadlocks.
- In Oz, it is possible to combine declarativeness and concurrency as these are independent, orthogonal features of the model of computation: concurrency by itself does not require, and does not imply, mutable state; threads can be synchronised through dataflow variables.

Task 7

Consider the following implementation of the function `Reverse` that takes a list as input and returns a list containing all the elements of the original list, but in the reverse order:

```
fun {Reverse List}
   case List of Head|Tail then
      {Append {Reverse Tail} [Head]}
   else
      nil
   end
end

fun {Append Front Back}
   case Front of Head|Tail then
      Head|{Append Tail Back}
   else
      Back
   end
end
```

Translate these definitions to the kernel language.

Consider an application of `Reverse` to a three-element list:

```
local Reverse in
   Reverse = proc {$ List ?Result}
                % <your kernel implementation>
             end
   local List in
      List = [1 2 3]
      local Result in
         {Reverse List Result}
         {Browse Result}
      end
   end
end
```

Think of an execution of the code above on the abstract kernel machine. The initial state (step 1) of the execution is as follows:

```
( [ ( local Reverse in ... end, {} ) ], {} )
```

- What will be the tenth step in the execution? You don't need to write the whole state of the machine; just explain in words what the content of the semantic stack and the store would be. Also, do not include all the preceding steps in your final answer.
- At the tenth step, what would be the environment included in the first (top-most) semantic statement on the stack?

Solution 7

Here is one way to implement `Reverse` in the kernel language:

```
local Reverse in
   Reverse = proc {$ List ?Result}
      local Append in
         Append = proc {$ Front Back ?Result}
            case Front of '|'(1:Head 2:Tail) then
               local Appended in
                  Result = '|'(1:Head 2:Appended)
                  {Append Tail Back Appended}
               end
            else
               Result = Back
            end
         end
         case List of '|'(1:Head 2:Tail) then
            local Reversed in
               local Last in
                  Last = [Head]
                  {Reverse Tail Reversed}
                  {Append Reversed Last Result}
               end
            end
         else
            Result = nil
         end
      end
   end
   % ...
end
```

Alternatively, `Reverse` might be implemented with `Append` included in the closure instead of as a procedure internal to `Reverse`. The execution depends on the implementation, hence the tenth step should correspond to the student's solution, not neessarily the solution above.

An execution of the code above with this implementation of `Reverse` would proceed as follows:

```
1. ( [ ( local Reverse in ... end, {} ) ],
     {} )

2. ( [ ( Reverse = proc ... end local List in ... end, {Reverse -> v1} ) ],
     {v1} )

3. ( [ ( Reverse = proc ... end, {Reverse -> v1} ),
       ( local List in ... end, {Reverse -> v1} ) ],
     {v1} )

4. ( [ ( local List in ... end, {Reverse -> v1} ) ],
     {v1 -> (proc {$ List ?Result} ... end, {Reverse -> v1})} )

5. ( [ ( List = [1 2 3] local Result in ... end, {Reverse -> v1, List -> v2} ) ],
     {v1 -> (...), v2} )

6. ( [ ( List = [1 2 3], {Reverse -> v1, List -> v2} ),
```

```
       ( local Result in ... end, {Reverse -> v1, List -> v2} ) ],
     {v1 -> (...), v2} )

7. ( [ ( local Result in ... end, {Reverse -> v1, List -> v2} ) ],
     {v1 -> (...), v2 = [1 2 3]} )

8. ( [ ( {Reverse List Result} {Browse Result}, {Reverse -> v1, List -> v2, Result -> v3} ) ],
     {v1 -> (...), v2 = [1 2 3], v3} )

9. ( [ ( {Reverse List Result}, {Reverse -> v1, List -> v2, Result -> v3} ),
       ( {Browse Result}, {Reverse -> v1, List -> v2, Result -> v3} ) ],
     {v1 -> (...), v2 = [1 2 3], v3} )

10. ( [ ( local Append in ... end, {Reverse -> v1, List -> v2, Result -> v3} ),
        ( {Browse Result}, {Reverse -> v1, List -> v2, Result -> v3} ) ],
      {v1 -> (...), v2 = [1 2 3], v3} )
```

In the tenth step, the procedure application statement `{Reverse List Result}` is replaced with the procedure body. The environment of this statement consists of the closure environment `{Reverse -> v1}` extened with mappings from the procedure parameters `List` and `Result` to the variables `v2` and `v3` mapped to the procedure application arguments `List` and `Result` in the application environment `{Reverse -> v1, List -> v2, Result -> v3}`. This environment happens to be identical to the environment of the call, but it is incidental.

Task 8

Implement the recursive function `Iterate` that takes three arguments:

- `State`, the initial state of computation;
- `Final`, a predicate function that applied to `State` returns `true` if the state is final, and `false` otherwise;
- `Next`, a function that applied to `State` returns the next state of computation.

`Iterate` should define an iterative, not recursive process.

Using `Iterate`, implement the following functions:

- `Generate` that takes three arguments, `From`, `Next`, and `Stop`, and produces a list; `From` is the first element of the list, `Next` is a function that given an element generates the next element, and `Stop` is a function that given en element decides whether next element should be generated or not;
- `FoldLeft`, that takes four arguments, `List`, `Nil`, `Transform`, and `Combine`, and performs left-folding of the list;
- `Map`, that takes two arguments, `List` and `Transform`, and produces a list of all elements of `List` after applying `Transform` to each.

After you're done, implement:

- `Enumerate`, a function that takes three arguments, `From`, `To`, and `By`, all three of them integers, and returns a list of numbers starting at `From` and not exceeding `To`, in increments `By`; implement `Enumerate` using `Generate`;
- `Map`, as above, but using `FoldLeft`.

Consider the following examples:

```
{Enumerate 1 5 2}
% [1 3 5]
{Map [1 2 3] fun {$ N} int(N) end}
% [int(1) int(2) int(3)]
{FoldLeft [1 2 3] 1 fun {$ N} N*N end fun {$ N M} N*M end}
% 36
```

Solution 8

Here are possible implementations of the above:

```
fun {Iterate State Final Next}
   if {Final State} then State
   else {Iterate {Next State} Final Next}
   end
end

fun {Generate From Next Stop}
   Result in
   {Iterate Result#From
```

```
            fun {$ _#Value} {Stop Value} end
            fun {$ List#Value} Tail in
               List = Value|Tail
               Tail#{Next Value} end} = nil#_
   Result
end

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

fun {Map List Transform}
   Result in
   {Iterate List#Result
            fun {$ List#_} List == nil end
            fun {$ (Head|Tail)#End} NewEnd in
               End = {Transform Head}|NewEnd
               Tail#NewEnd end} = _#nil
   Result
end

fun {Enumerate From To By}
   {Generate From
             fun {$ N} N + By end
             fun {$ N} N > To end}
end

fun {Map List Transform}
   Result in
   {FoldLeft List Result Transform
             fun {$ Result Value} Tail in
                Result = Value|Tail
                Tail
             end} = nil
   Result
end
```

Task 9

Specify the abstact kernel machine semantics of the following types of statements:

- `try ... catch ... then ... end`;
- `raise ... end`.

Answer these questions:

- If you extend the declarative sequential model with exceptions, is the extended language still declarative?
- If you extend the declarative concurrent model with exceptions, is the extended language still declarative?

Justify your answers.

Solution 9

The semantics of the `try` statement is as follows.
Given the semantic statement (`try <try-statement> catch <id> then <then-statement> end`, E) popped from the semantic stack:

1. Push onto the stack the semantic statement (`catch <id> then <then-statement> end`, E).
2. Push onto the stack the semantic statement (`<try-statement>`, E).

The semantics of the `raise` statement is as follows.
Given the semantic statement (`raise <raise-id> end`, Er) popped from the semantic stack:

1. If the semantic stack is empty, stop execution and signal an uncaught exception error.
2. Pop another statement from the semantic stack and check if it has the following form:
   (`catch <catch-id> then <statement> end`, Ec).
   - If yes, push onto the semantic stack the semantic statement:
     (`<statement>`, Ec + { `<catch-id>` -> Er(`raise-id`) }).
   - If not, return to step 1.

In this task you were not asked to provide the semantics of the `catch` statement.

According to the pensum book, extending the declarative sequential model with exceptions results in a model that still is declarative. The justification for this claim is that in a declarative sequential program, if an exception occurs, it will be the same exception at all executions of the program, and it will lead to the same result—either an uncaught exception on termination of the program, or a successful termination with the same result after catching the exception.

The concurrent model of computation combined with exception leads out of declarativity—one example being two or more threads competing to bind a variable to different values; without exceptions, there would always be failure.
With appropriate exception handling, at least one thread may succeed, but on different runs it may be a different thread, leading to different results.

Task 10

Central to how programs are executed is the notion of scope.

- Explain the difference between dynamic scope and lexical scope.
- What kind of scope does Oz support?
- Provide an example of code in Oz that would lead to a different result if Oz supported scoping other than it actually does.
- In your opinion, which type of scoping makes it easier to reason about the execution of programs? Why?

Solution 10

In the context of variable binding, scope corresponds to the limit of visibility of a variable and its value within a program.

With lexical (or static) scoping, the limitation refers to the lexical (syntactic) structure of a program; the binding of a variable depends on the syntactic context of the code where the variable was declared. With dynamic scoping, the limitation refers to the runtime execution of the program; the binding of a variable depends on the call context of the executing process.

Oz is a lexically scoped language. In the following example, the binding of `Variable` seen by the procedure `Show` at the time it is executed is that declared in the outer `local` block—it is the binding in the lexical context of `Show`'s definition:

```
local
   Variable = 0
   proc {Show} {Browse Variable} end
in
   local
      Variable = 1
   in
      {Show} % prints 0 with lexical scoping, 1 with dynamic scoping
   end
end
```

With dynamic scoping, the value of `Variable` seen by `Show` at the time it is executed would be that given to `Variable` within the inner `local` block, within which `Show` is called.

Roughly speaking, with lexical scoping, the the value of a variable depends on *where* is is defined, with dynamic scoping it depends on *when* it is used.

Task 11

Consider the following code written in Scala:

```
var state = 0
val threads = (1 to 3).map { int => new Thread { override def run() = state += int } }
threads.foreach(_.start())
threads.foreach(_.join())
```

In brief, this program

- defines an integer variable called `state` and initializes it with the value `0`;
- creates a range of integers from `1` to `3`, inclusive;
- for each of those integers, creates a thread that increments `state` by that integer;
- starts the threads and then blocks until all of them are complete with their computations.

Answer the following questions:

- What is the value of `state` after the program above is complete?
- Is `state` guaranteed to have the same value on each execution of this program?
- If yes, what is the value?
- If not, what are the values that `state` can possibly have?
- If you don't think this program is guaranteed to result in `state` always having the same value, explain why and propose an improvement that would guarantee that. (If you don't know Scala, sketch your solution in words.)
- If you do think this program is guaranteed to result in `state` always having the same value, explain why.

Solution 11

The operation `+=` has three components:

- looking up the value of the left-hand side variable;
- increasing this value by the right-hand side value;
- assigning the increased value back to the variable.

In a sequential model of computation, it does not matter whether these components are executed as separate operations, or as one atomic operation. In a concurrent model of computation, it does matter.

If the operation is atomic, then:

- each thread successfully increases the value of `state` so that the value just after the assignment equals the value just before the reading plus the increment;
- in total, the three threads increase `state` by `1 + 2 + 3 = 6`, hence the final value would always be `6`.

If the operation is not atomic, then:

- threads may interrupt each other, so that two threads read the same value of `state`, update it with their respective increments, and then write back the incremented value;
- thus, at least one thread's update is lost (its update is overwritten by the other thread's update);

- the final result will depend on the order of execution of the threads and overlap among them between the read and write operations;
- the possible values for `state` are: `1`, `2`, or `3` if only one thread correctly updates `state` and the other two updates are lost; `3`, `4`, and `5` if one thread's update is lost; `6` if all threads update `state` without interruption.

There are a few possibilities to prevent corruption of `state` in case of non-atomic `+=`:

- a thread may explicitly lock access (`synchronize`) to `state` for the duration of the whole update;
- instead of plain `int`, one may use a reference type (e.g., `java.util.concurrent.atomic.AtomicInteger`) with atomic update operations;
- use the actor model to have one thread only with direct access to `state`, and all other threads making updates to it by sending messages to that thread.