Appendix A

How to test it

Jar file

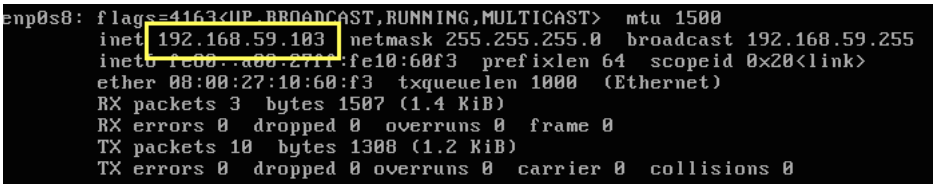
You can find our source code on the following bitbucket git: `https://paquota@bitbucket.org/paquota/correctoz.git`. All the classes you need are regrouped in the *CorrectOz.jar* file, you only have to execute it with the corresponding arguments. This command is the simplest one to run our tool on the Oz code written in the [FileName]. You can obviously use all the others arguments presented previously. A README is provided on the git with all the necessary information. You will also find on the git an example of the current configuration on INGInious and some inputs file containing codes in Oz.

```
java -jar CorrectOz.jar [FileName] -d false
```

Virtual machine

If you prefer to use our tool via INGInious, we also provide you a VM with the exercises of the first MOOC. You can download the VM (3GB) via this link : `https://drive.google.com/open?id=0B5SqZj-xejjrWVRmNkRWR04zdXM` All you have to do is to submit a code as a student would do. The procedure to use the virtual machine is the following one:

1. Open it with virtual box
2. Username and password are 'root'
3. Go to the INGInious folder: **cd INGInious**
4. Find your IP address: **ifconfig**



5. Launch the webapp: **inginious-webapp –host [IpAddress] –port 80**
6. Open your browser and go to the [IpAddress]
7. User and password are 'test'
8. Click on the unique course : [LTEST0000] Test tasks : H2G2 - List of exercises
9. Choose your exercise among the 15 and submit your code.

Appendix B

Examples

0.0.1 Edx1

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx0 -nVar 1 -nProc 0 -nFun 0 -nCall 1**, corresponding to the command line for the first exercise of the course.

Example 1 :

```
1 local X Y Z in
2    Y = (6*5)
3    Z = 9-7
4    X = Y+Z
5    {Browse X}
6 end
```

```
HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 1 variables instead of
 3.
```

Example 2 :

```
1 local X in
2    Y = (6*5)+(9-7)
3    {Browse Y}
4 end
```

```
ERROR :  The variable is not declared. The variable(s) concerned :[Y].
Error found in " Y = ( 6 * 5 ) + ( 9 - 7 )".
```

```
HINT : The variable X is not used. It is probably useless.
```

Example 3 :

```
1  local X Useless in
2
3     fun{Useless R}
4        R
5     end
6
7     X = {Useless 6*5} + {Useless 9-7}
8     {Browse X}
9
10 end
```

```
HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 1 variables instead of
 2.


HINT :  You seem to have declare too many functions for the exercice.
Maybe some functions are not needed.We expected 0 functions instead of 1.


HINT :  You seem to have performed too many procedure calls for the
exercice.
Try to delete some by storing the result in a variable or by deleting
some useless
procedure/function. We expected 1 instead of 3.
```

Example 4 :

```
1  local x in
2     x= (6*5)+(9-7)
3     {Browse x}
4  end
```

```
ERROR :  A variable should always start with an upper case.
The variable(s) concerned :[x]. Error found in " x".

ERROR :  The variable is not declared. The variable(s) concerned :[x].
Error found in " x = ( 6 * 5 ) + ( 9 - 7 )".
```

0.0.2 Edx2

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx1 -d false -nVar 6 -nFun2 -nCall 6**, corresponding to the command line for the second exercise of the course.

Example 5 :

```
1  local P Q T X=1 Y=2 Z=3 S in
2
3    fun {P X}
4      X*Y+Z
5  end
6
7    fun {Q Z}
8      X*Y+Z
9    end
10
11   T = {S 4}
12
13   {Browse T*3}
14   {Browse {Q 4} == 6}
15   {Browse {P 5} == 2}
16 end
```

```
ERROR :  The procedure you want to call is not declared.
 Maybe did you mean one of those : {P, Q}.
The variable(s) concerned :[S]. Error found in " { S 4 }".

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread.
The variable(s) concerned :[T]. Error found in " T * 3".

ERROR :  One of the term is not a number.
The variable(s) concerned :[T (t0) , 3 (3) ]. Error found in " T * 3".


HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 6 variables instead of
 7.
```

We can see here that an error can unfortunately lead to false positive. Since the student calls a function that does not exist, the variable T will not be assigned and therefore the call to browse with the argument "T*3" will also raise errors. We could stop the analyse after the first error found, but most of the times, it is useful to continue to find out others ones.

Example 6 :

```
local Q T X=1 Y=2 Z=3 in

   fun {P X}
     X*Y+Z
end

   fun {Q Z}
     X*Y+Z
   end

   T = {P 4}

   {Browse T*3}
   {Browse {Q 4} == 6}
   {Browse {P 5} == 2}
end
```

```
ERROR :  The variable is not declared. The variable(s) concerned :[P].
Error found in " fun { P X } X * Y + Z end".
```

Note that, in this case, even if the function P is not declared, the program will execute and give the correct result. When a student forgets to declare a variable, we declare it for him to let the program execute as it should.

Example 7 :

```
local P Q T X=1 Y=2 Z=3 in

    fun {P X
       X*Y+Z
    end

    fun {Q Z}
       X*Y+Z
    end

    T = {P 4}

    {Browse T*3}
    {Browse {Q 4} == 6}
    {Browse {P 5} == 2}
end
```

```
ERROR :  A " } " is missing in your procedure/function declaration.
Error found in " fun { P X X * Y + Z end".
```

```
ERROR :  Too many argument for the procedure.
The variable(s) concerned :[P]. Error found in " { P 4 }".

ERROR :  Too many argument for the procedure.
The variable(s) concerned :[P]. Error found in " { P 5 }".
```

Once again, the first error leads to two other non correct feedback. But once again, the result is the correct one as we try to correct as much errors as we can while giving feedback to the student.

Example 8 :

```
1  local P Q T X=1 Y=2 Z=3 in
2
3     fun {P X}
4        X*Y+Z
5     end
6
7     fun Q Z}
8        X*Y+Z
9     end
10
11    T = {P 4}
12
13    {Browse T*3}
14    {Browse {Q 4} == 6}
15    {Browse {P 5} == 2}
16 end
```

```
ERROR :  A " { " is missing in your procedure/function declaration.
Error found in " fun Q Z } X * Y + Z end".

ERROR : The procedure/variable already exists.
Maybe the procedure you want to implement is already implemented.
Otherwise, change the name of your procedure.
The variable(s) concerned :[Z]. Error found in " fun Q Z } X * Y + Z end
".

ERROR :  The variable is not a number. The variable(s) concerned :[Z (
OzFunction :
    body = funQZ}X*Y+Zend
    Env = {P=p0, Q=q0, T=t0, X=x0, Y=y0, Z=z0}
)]. Error found in " X * Y + Z".

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread.
The variable(s) concerned :[T]. Error found in " T * 3".
```

```
ERROR :  One of the term is not a number.
The variable(s) concerned :[T (t0) , 3 (3) ]. Error found in " T * 3".

ERROR :  The procedure you want to call is not declared.
 Maybe did you mean one of these : {P, Z}.The variable(s) concerned :[Q].
 Error found in " { Q 4 }".

ERROR :  You have an error due to one of the variables Error found in " {
 Q 4 } == 6".

ERROR :  You have an error due to one of the variables Error found in " {
 P 5 } == 2".
```

The first error is responsible of all the others. It is a consequence of continuing to look after errors even if we already found one.

0.0.3 Edx3

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx3 -d false -nVar 2 -nFun1 -nCall 5**, corresponding to the command line for the third exercise of the course.

Example 9 :

```
1  local X SlowAdd in
2    fun {SlowAdd Z Y}
3           {Delay 1000}
4           Z+Y
5     end
6
7     X = {SlowAdd 1000 1} + {SlowAdd 1000 1} + {SlowAdd 1000 1}
8      {Browse X}
9  end
```

```
HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function.
Or maybe did you forget to update the argument ?
The arguments are : [Argument "1000" with value "1000" at
position "1" in method "SlowAdd", Argument "1" with value "1" at position
 "2"
in method "SlowAdd"]The variable(s) concerned :[SlowAdd].
Error found in " { SlowAdd 1000 1 }".


HINT :  You seem to have performed too many procedure calls for the
exercice.
```

```
Try to delete some by storing the result in a variable or by deleting
some useless
procedure/function. We expected 5 instead of 7.
```

For the exercise, the fact that the student does not store the value in the variable should be considered as an error, but this is too specific to the exercise to be implemented as such. The unit tests should be in charge to report this error.

Example 10 :

```
1  local X SlowAdd in
2
3     fun {SlowAdd Z Y}
4           {Delay 1000}
5           Z+Y
6     end
7
8      X = {SlowAdd}
9
10 end
```

```
ERROR :  Not enough argument for the procedure.
The variable(s) concerned :[SlowAdd]. Error found in " { SlowAdd }".

ERROR :  The variable is not declared.
The variable(s) concerned :[Z]. Error found in " Z".

ERROR :  The variable is not declared.
The variable(s) concerned :[Y]. Error found in " Y".

ERROR :  One of the term is not a number.
The variable(s) concerned :[Z (null) , Y (null) ]. Error found in " Z + Y
".
```

Some false positive due to the first error.

Example 11 :

```
1  local X SlowAdd in
2
3     fun {SlowAdd Z Y}
4           {Delay 1000}
5           Z+Y
6     end
7
8
9      fun {SlowAdd X Y}
10          {Delay 1000}
11             X + Y
12    end
13
14     X={SloawAdd 3*1000 3*1}
15
16     {Browse X}
17
18 end
```

```
ERROR :  The procedure/variable already exists.
Maybe the procedure you want to implement is already implemented.
Otherwise, change the name of your procedure.
The variable(s) concerned :[SlowAdd]. Error found in " fun { SlowAdd X Y
} { Delay 1000 } X + Y end".

ERROR :  The variable is not declared.
The variable(s) concerned :[SloawAdd]. Error found in " SloawAdd".

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of those : {SlowAdd}.
The variable(s) concerned :[SloawAdd]. Error found in " { SloawAdd 3 *
1000 3 * 1 }".

HINT : The variable SloawAdd is not used. It is probably useless.
```

Example 12 :

```
local X SlowAdd in

   fun {SlowAdd Z Y}
         {Delay 1000}
         Z+Y
   end


    fun {SlowAdd X Y}
         {Delay 1000}
            X + Y
   end

      fun {SlowAdd X Y}
         {Delay 1000}
        X+Y
   end

   local X in
         fun {SlowAdd X }
            {Delay 3000}
               X = (X+Y)+(X+Y)+(X+Y)

         end

    end
    {Browse {SlowAdd 3003}}

end
```

```
ERROR :  The procedure/variable already exists.
Maybe the procedure you want to implement is already implemented.
Otherwise, change the name of your procedure.
The variable(s) concerned :[SlowAdd].
Error found in " fun { SlowAdd X Y } { Delay 1000 } X + Y end".

ERROR :  The procedure/variable already exists.
Maybe the procedure you want to implement is already implemented.
Otherwise, change the name of your procedure.
The variable(s) concerned :[SlowAdd].
Error found in " fun { SlowAdd X } { Delay 3000 } X = ( X + Y ) + ( X + Y
 ) + ( X + Y ) end".

ERROR :  The variable is not declared.
The variable(s) concerned :[Y]. Error found in " Y".

ERROR :  The variable is not a number.
The variable(s) concerned :[Y (null)]. Error found in " X + Y".
```

```
ERROR :  One of the term is not a number.
The variable(s) concerned :[(X+Y) (null) , (X+Y) (null) ]. Error found in
 " ( X + Y ) + ( X + Y )".

ERROR :  The variable is not a number.
The variable(s) concerned :[(X+Y) (null)]. Error found in " ( X + Y ) + (
 X + Y ) + ( X + Y )".

ERROR :  The variable is not declared.
The variable(s) concerned :[X]. Error found in " X = ( X + Y ) + ( X + Y
) + ( X + Y )".


HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 2 variables instead of
 3.
```

0.0.4 Edx4

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx4 -d false -nVar 3 -nFun 2 -nIf 1 -tailRec true**, corresponding to the command line for the exercise number 4 of the course.

Example 13 :

```
1  local MainSum R in
2     fun {MainSum N}
3           local Sum in
4                 fun {Sum N Acc}
5                   if N==0 then 0
6              else N*N+{Sum N-1 Acc}
7           end
8                 end
9               {Sum N 0}
10          end
11       end
12
13    R = {MainSum 8}
14 end
```

```
ERROR : These functions are not tail recursive : [Sum]
```

Example 14 :

```
local MainSum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N=0 then Acc
         else {Sum N-1 Acc+N*N}
         end
      end
      in {Sum N 0}
   end

   R = {MainSum 8}

end
```

```
ERROR : Your program contains some errors

ERROR :  The variable is not declared.
The variable(s) concerned :[N]. Error found in " N = 0".

ERROR :  The Condition is not a boolean.
The value is null.  Error found in " if N = 0 then Acc else { Sum N - 1
Acc + N * N } end".
```

Example 15

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N==0 then {Browse {MainSum n}}
         else Acc=(N)^2 N=N-1
         end
       end
    {Sum N 0}
   end

   R = {MainSum 8}

end
```

```
ERROR :  You cannot change the value of the variable because it was
already assigned to one.
 A variable can only be assigned to one value in Oz. Use cell with " :=\
 " for multiple assignments.
 The variable(s) concerned :[Acc]. Error found in " Acc = ( N ) ^ 2".
```

```
ERROR :  This operator does not exist in Oz. The variable(s) concerned
:[^]. Error found in " ( N ) ^ 2".

ERROR :  The variable is not declared. The variable(s) concerned :[N].
Error found in " N = N - 1".
```

Example 16

```
1  local MainSum Sum R in
2     fun {MainSum N}
3        fun {Sum N Acc}
4           fun {Sum N Acc}
5              if (N==0) then Acc
6              else {MainSum N+1 Acc*N} end
7           end
8         end
9      {Sum N 0}
10    end
11
12    R = {MainSum 8}
13 end
```

```
ERROR :  The procedure/variable already exists. Maybe the procedure you
want to implement
is already implemented. Otherwise, change the name of your procedure.
The variable(s) concerned :[Sum].
Error found in " fun { Sum N Acc } if ( N == 0 ) then Acc else { MainSum
N + 1 Acc * N } end end".

HINT : You seem to have declared too many functions for the exercise.
Maybe some functions
are not needed. We expected 2 functions instead of 3.
```

Example 17

```
1  local MainSum Sum R in
2     fun {MainSum N}
3        fun {Sum N Acc}
4           if (N==0) Acc
5           else Acc= Acc + N ^2
6         end
7      {Sum N 0}
8     end
9     R = {MainSum 8}
10 end
```

```
ERROR :  A " then " statement is missing in the if statement.
[if <condition> then <statement> (else <statement>) end].
Error found in " if ( N == 0 ) Acc else Acc = Acc + N ^ 2".

ERROR :  A " end " statement is missing in the if statement.
 [if <condition> then <statement> (else <statement>) end].
 Error found in " if ( N == 0 ) Acc else Acc = Acc + N ^ 2".


HINT :  An " else " statement is missing in your if structure.
it is better to have a default case.
Error found in " if ( N == 0 ) Acc else Acc = Acc + N ^ 2".


HINT :  The case/if statement does not match any pattern for the
expression.
You should have an else statement.
The variable(s) concerned :[(N==0)( false)].
Error found in " if ( N == 0 ) Acc else Acc = Acc + N ^ 2".
```

The two hints are false positives due to the two first errors.

Example 18

```
1  local MainSum Sum R in
2     fun {MainSum N}
3        fun {Sum N Acc}
4           if (N==0) then Acc
5           else Acc= Acc + N ^2 N--
6           end
7         end
8      {Sum N 0}
9     end
10    R = {MainSum 8}
11 end
```

```
ERROR :  You cannot change the value of the variable because it was
already assigned to one. A variable can only be assigned to one value in
Oz.
Use cell with " :=\ " for multiple assignments.
The variable(s) concerned :[Acc]. Error found in " Acc = Acc + N ^ 2".

ERROR :  This operator does not exist in Oz.
The variable(s) concerned :[^]. Error found in " N ^ 2".

ERROR :  The variable is not a number.
The variable(s) concerned :[N^2 (null)]. Error found in " Acc + N ^ 2".

ERROR :  This operator does not exist in Oz.
The variable(s) concerned :[--]. Error found in " N --".
```

Example 19

```
1  local MainSum Sum R in
2     fun {MainSum N}
3        fun {Sum N Acc}
4           if N == 0 then Acc
5           else {SumAcc N-1 Acc+N*N}
6         end
7      {Sum N 0}
8     end
9     R = {MainSum 8}
10 end
```

```
ERROR :  A " end " statement is missing in the if statement.
[if <condition> then <statement> (else <statement>) end].
Error found in " if N == 0 then Acc else { SumAcc N - 1 Acc + N * N }".

ERROR :  The variable is not declared.
The variable(s) concerned :[SumAcc]. Error found in " SumAcc".

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of those : {MainSum, Sum}.
The variable(s) concerned :[SumAcc]. Error found in " { SumAcc N - 1 Acc
+ N * N }".

HINT :  An " else " statement is missing in your if structure.
it is better to have a default case.
 Error found in " if N == 0 then Acc else { SumAcc N - 1 Acc + N * N }".

HINT : The variable SumAcc is not used. It is probably useless.
```

Example 20

```
1  local MainSum Sum R in
2     fun {MainSum N}
3        fun {Sum N Acc}
4           while N >0 then
5              Acc=Acc+N*N
6              N-1
7           end
8        end
9      {Sum N 0}
10    end
11
12    R = {MainSum 8}
13
14 end
```

```
ERROR : Your program contains some errors

ERROR :  A " do " statement is missing in the for/while construction.
[for/while <loopDec> do <statement>].
Error found in " while N > 0 then Acc = Acc + N * N N - 1 end".

ERROR :  You cannot change the value of the variable because it was
 already assigned to one. A variable can only be assigned to one value in
  Oz.
 Use cell with " :=\ " for multiple assignments.
 The variable(s) concerned :[Acc]. Error found in " Acc = Acc + N * N".

HINT :  You do not use enough if statement for the exercise.
We expect you to use at least 1 if statement(s).
```

Example 21

```
1  local MainSum Sum R in
2     fun {MainSum N}
3        fun {Sum N Acc}
4           if N==0 then Acc
5           else{Sum N-1 AccN*N}
6           end
7        end
8      {Sum N 0}
9     end
10
11    R = {MainSum 8}
12
13 end
```

```
ERROR :  The variable is not declared. The variable(s) concerned :[AccN].
 Error found in " AccN".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (8) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (7) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (6) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (5) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (4) ]. Error found in " AccN * N".
```

```
ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (3) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (2) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (1) ]. Error found in " AccN * N".
```

Example 22

```
1  local MainSum Sum R in
2     fun {MainSum N}
3        fun {Sum N Acc}
4           N*N
5           end
6      {Sum N 0}
7     end
8
9     R = {MainSum 8}
10
11 end
```

```
HINT : The variable Acc is not used. It is probably useless.

HINT :  You do not use enough if statement for the exercise. We expect
you to use at least 1 if statement(s).
```

We can see here the utility of simple check such as the number of if structure used. In this case, the student is warned that he should use at least one if structure for the exercise. The exercise is therefore more complicated that he thought and he is redirected in the good direction with the two hints.

Example 23

```
1  local MainSum Sum R in
2     fun {MainSum N}
3        fun {Sum N Acc}
4           if N==0 then Acc
5           else {Acc=N^2} {N=N-1} end
6
7           end
8      {Sum N 0}
9     end
10
11    R = {MainSum 8}
12
13 end
```

```
ERROR :  This operator does not exist in Oz.
The variable(s) concerned :[^]. Error found in " N ^ 2".

ERROR :  The variable is not declared.
The variable(s) concerned :[Acc]. Error found in " Acc = N ^ 2".

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of those : {MainSum, Sum}.
The variable(s) concerned :[Acc=N^2]. Error found in " { Acc = N ^ 2 }".

ERROR :  The variable is not declared.
The variable(s) concerned :[N]. Error found in " N = N - 1".

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of these : {MainSum, Sum}.
The variable(s) concerned :[N=N-1]. Error found in " { N = N - 1 }".
```

In this case, the student has not understood the notation for the call of procedure. Even if the feedback is not as precise as we would, it can lead the student to understand that the notation he used is for procedure call.

0.0.5 Edx5

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx5 -d false -nVar 4 -nFun 2 -nCall 7 -nIf 1 -tailRec true**, corresponding to the command line for the exercise number 5 of the course.

Example 24 :

```
1  local R Prime in
2     fun {Prime N}
3           fun {PrimeAcc N Acc}
4              if N==1 then false
5                 else if Acc==1 then true
6                    else if (N mod Acc)==0 then false
7                       else {PrimeAcc N Acc}
8                       end
9                    end
10                end
11             end
12          in {PrimeAcc N N-1}
13    end
14    R = {Prime 7}
15 end
```

```
ERROR :  Your program runs infinitely, it can be from one of
these parts of the program :
[ fun { PrimeAcc N Acc } if N == 1 then false else if Acc == 1 then
true else if ( N mod Acc ) == 0 then false else { PrimeAcc N Acc }
end end end end]
You call Argument "N" with value "7" at position "1" in method "PrimeAcc"
 187 times.
Maybe you forgot to update it.
You call Argument "Acc" with value "6" at position "2" in method "
PrimeAcc" 186 times.
 Maybe you forgot to update it.

HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function.
Or maybe did you forget to update the argument ?
The arguments are : [Argument "N" with value "7" at position "1" in
method "PrimeAcc",
Argument "Acc" with value "6" at position "2" in method "PrimeAcc"].
The variable(s) concerned :[PrimeAcc]. Error found in " { PrimeAcc N Acc
}".
```

Example 25 :

```
1  local R Prime in
2     fun {Prime N}
3        local isPrime in
4            fun {isPrime N Acc}
5                 if N == Acc then true
6               else  if (N mod 2) == 0 then false
7                      else {IsPrime N Acc + 1 }
8                      end
9                end
10            end
11       {IsPrime N 1}
12       end
13    end
14 R = {Prime 7}
15 end
```

```
ERROR :  A variable should always start with an upper case.
The variable(s) concerned :[isPrime]. Error found in " isPrime".

ERROR :  A variable should always start with an upper case.
Error found in " fun { isPrime N Acc } if N == Acc then true else
if ( N mod 2 ) == 0 then false else { IsPrime N Acc + 1 } end end end".

ERROR :  The variable is not declared. The variable(s) concerned :[
```

```
isPrime].
Error found in " fun { isPrime N Acc } if N == Acc then true else if ( N
mod 2 ) == 0
then false else { IsPrime N Acc + 1 } end end end".

ERROR :  The variable is not declared.
The variable(s) concerned :[IsPrime]. Error found in " IsPrime".

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of these : {isPrime, Prime}.
The variable(s) concerned :[IsPrime]. Error found in " { IsPrime N 1 }".
```

Example 26 :

```
1  local R Prime in
2     fun {Prime N}
3        fun {While N S}
4           if N = 1 then false
5           elseif N = S then true
6           elseif (N mod S) = 0 then false
7           elseif 1 = 1 then {While N (S+1)}
8           end
9     end
10    {While N 1}
11 end
12 R = {Prime 7}
13 end
```

```
ERROR : Your program contains some errors

ERROR :  The variable is not declared. The variable(s) concerned :[While
].
Error found in " fun { While N S } if N = 1 then false elseif N = S then
true
elseif ( N mod S ) = 0 then false elseif 1 = 1 then { While N ( S + 1 ) }
 end end".

ERROR :  The variable is not declared. The variable(s) concerned :[N].
Error found in " N = 1".

ERROR :  The Condition is not a boolean. The value is null.
Error found in " if N = 1 then false elseif N = S then true elseif
( N mod S ) = 0 then false elseif 1 = 1 then { While N ( S + 1 ) } end".

HINT :  An " else " statement is missing in your if structure. it is
better
to have a default case. Error found in " if N = 1 then false elseif N = S
then true elseif ( N mod S ) = 0 then false elseif 1 = 1
 then { While N ( S + 1 ) } end".
```

0.0.6 Edx6

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx6 -d false -nVar 3 -nFun 2 -nIf 1 -tailRec true**, corresponding to the command line for the exercise number 6 of the course.

Example 27 :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
                if N == 0 then 0
               else if N == 1 then 1
                  else Acc1+{FibAux N Acc2 Acc1+Acc2}
                  end
               end
            end
            {FibAux N 0 1}
      end
    end
    R = {Fib 9}
end
```

```
ERROR : These functions are not tail recursive : [FibAux]

ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { FibAux N Acc1 Acc2 } if N == 0 then 0 else if N == 1 then 1
else Acc1 + { FibAux N Acc2 Acc1 + Acc2 } end end end]
You call Argument "Acc2" with value "1" at position "2" in
method "FibAux" 2 times. Maybe you forgot to update it.
You call Argument "N" with value "9" at position "1" in
method "FibAux" 223 times. Maybe you forgot to update it.
```

Example 28 :

```
1  local Fib R in
2      fun {Fib N}
3          local FibAux in
4              fun {FibAux N Acc1 Acc2}
5                 if N == 0 then 0
6                 else  if N==1 then 1
7                 else  {FibAux N-1 0 1}+ {FibAux N-2 0 1}
8                 end
9           end
10           {FibAux N 0 1}
11       end
12       end
13    R = {Fib 9}
14 end
```

```
ERROR : These functions are not tail recursive : [FibAux]

ERROR :  A " end " statement is missing in the if statement. [if <
condition> then <statement> (else <statement>) end]. Error found in " if
N == 0 then 0 else if N == 1 then 1 else { FibAux N - 1 0 1 } + { FibAux
N - 2 0 1 } end".

===== HINTS =====

HINT :  An " else " statement is missing in your if structure. it is
better to have a default case. Error found in " if N == 0 then 0 else if
N == 1 then 1 else { FibAux N - 1 0 1 } + { FibAux N - 2 0 1 } end".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-1" with value "1" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-2" with value "0" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
```

```
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-1" with value "2" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-2" with value "1" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-1" with value "3" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-2" with value "2" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-1" with value "4" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-2" with value "3" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
```

```
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-1" with value "5" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-2" with value "4" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-1" with value "6" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "N-2" with value "5" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


HINT : The variable Acc2 is not used. It is probably useless.


HINT : The variable Acc1 is not used. It is probably useless.
```

The problem here is that the student does not use the accumulator, resulting in calling the function several times with the same arguments leading to this big feedback. Since he calls the functions twice at every call with the same accumulator that he does not update, CorrectOz think it could store the result in a variable. Even if this feedback is not the correct improvement for the mistake, the student is warned that he calls several times the same functions with the same arguments and that the two accumulators are not used. This feedback could make him find the right answer.

Example 29 :

```
1  local Fib R in
2      fun {Fib N}
3          local FibAux in
4              fun {FibAux N Acc1 Acc2}
5              if N==0 then Acc1
6              else
7               {FibAux Acc2 Acc2+Acc1}
8              end
9           end
10           {FibAux N 0 1}
11       end
12       end
13     R = {Fib 9}
14     {Browse R}
15 end
```

```
ERROR :  Not enough argument for the procedure.
The variable(s) concerned :[FibAux].
Error found in " { FibAux Acc2 Acc2 + Acc1 }".

ERROR :  The variable is not declared.
The variable(s) concerned :[Acc2]. Error found in " Acc2".

ERROR :  One of the term is not a number.
The variable(s) concerned :[Acc2 (null) , Acc1 (1) ].
Error found in " Acc2 + Acc1".

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread. The variable(s) concerned :[N].
Error found in " N == 0".

ERROR :  One of the term is not a number.
The variable(s) concerned :[Acc2 (null) , Acc1 (-43) ].
Error found in " Acc2 + Acc1".

ERROR :  Your program runs infinitely, it can be from
one of these parts of the program :
[ fun { FibAux N Acc1 Acc2 } if N == 0 then Acc1 else { FibAux Acc2 Acc2
+ Acc1 } end end]
You call Argument "Acc2" with value "null" at position "1" in method "
FibAux" 368 times. Maybe you forgot to update it.
You call Argument "Acc2+Acc1" with value "-43" at position "2" in method
"FibAux" 368 times. Maybe you forgot to update it.

HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function.
Or maybe did you forget to update the argument ?
The arguments are : [Argument "Acc2" with value "null" at position "1" in
```

```
 method
"FibAux", Argument "Acc2+Acc1" with value "-43" at position "2" in method
 "FibAux"].
The variable(s) concerned :[FibAux]. Error found in " { FibAux Acc2 Acc2
+ Acc1 }".
```

We have some false positive in the feedback of this exercise. Since the student forgot one argument when calling the function, the program thinks that the missing argument is not declared and has therefore the null value. This leads to a wrong feedback about the declaration of the variable and the deadlock. However, since the real error is explained (the number of argument is not the right one), the student can recover easily.

Example 30 :

```
1  local Fib R in
2      fun {Fib N}
3          local FibAux in
4              fun {FibAux N Acc1 Acc2}
5              if N == 0 then 0 else 1 end
6           end
7            {FibAux N 0 1}
8        end
9        end
10     R = {Fib 9}
11     {Browse R}
12 end
```

```
HINT : The variable Acc2 is not used. It is probably useless.

HINT : The variable Acc1 is not used. It is probably useless.
```

In this case, even if the syntax is correct and without unit test, the feedback shows to the student that his program does not behave as it should do since it does not use the two accumulators. If we couple it with some unit tests, CorrectOz would be really powerful.

Example 31 :

```
1  local Fib R in
2      fun {Fib N}
3          local FibAux in
4              fun {FibAux N Acc1 Acc2}
5              if N==0 then Acc2
6              else {FibAux N-1 Acc1 Acc1+Acc2} + {FibAux N-2 Acc1+Acc2 Acc2
               }
7              end
```

```
8           end
9            {FibAux N 0 1}
10       end
11       end
12    R = {Fib 9}
13    {Browse R}
14 end
```

```
ERROR: These functions are not tail recursive: [FibAux]

ERROR: Your program runs infinitely, it can be from one of those parts of
 the program: [ fun { FibAux N Acc1 Acc2 } if N == 0 then Acc2 else {
FibAux N - 1 Acc1 Acc1 + Acc2 } + { FibAux N - 2 Acc1 + Acc2 Acc2 } end
end]
You call Argument "Acc1+Acc2" with value "1" at position "3" in method "
FibAux" 9 times. Maybe you forgot to update it.

You call Argument "Acc1" with value "0" at position "2" in method "FibAux
" 9 times. Maybe you forgot to update it.

You call Argument "Acc1" with value "1" at position "2" in method "FibAux
" 320 times. Maybe you forgot to update it.
```

Example 32 :

```
1  local Fib R in
2      fun {Fib N}
3          local FibAux in
4              fun {FibAux N Acc1 Acc2}
5              if (N==1) then 1
6              elseif (N==0) then 0
7              else {FibAux N-1 } + {FibAux N-2 } end
8       end
9           {FibAux N 0 1}
10      end
11      end
12    R = {Fib 9}
13 end
```

```
ERROR: These functions are not tail recursive: [FibAux]

ERROR: Not enough argument for the procedure. The variable(s) concerned
:[FibAux]. Error found in " { FibAux N - 1 }".

ERROR: Not enough argument for the procedure. The variable(s) concerned
:[FibAux]. Error found in " { FibAux N - 2 }".
```

```
HINT: You call the procedure several times with the same argument(s). You
 could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ?

The arguments are: [Argument "N-1" with value "1" at position "1" in
method "FibAux"]

The variable(s) concerned:[FibAux]. Error found in " { FibAux N - 1 }".

HINT: You call the procedure several times with the same argument(s). You
 could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ?

The arguments are: [Argument "N-2" with value "0" at position "1" in
method "FibAux"]

The variable(s) concerned:[FibAux]. Error found in " { FibAux N - 2 }".

HINT: You call the procedure several times with the same argument(s). You
 could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ?

The arguments are: [Argument "N-1" with value "2" at position "1" in
method "FibAux"]
The variable(s) concerned:[FibAux]. Error found in " { FibAux N - 1 }".

HINT: You call the procedure several times with the same argument(s). You
 could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ?

The arguments are: [Argument "N-2" with value "1" at position "1" in
method "FibAux"]
The variable(s) concerned:[FibAux]. Error found in " { FibAux N - 2 }".

HINT: The variable Acc2 is not used. It is probably useless.

HINT: The variable Acc1 is not used. It is probably useless.
```

Example 34 :

```
1  local Fib R in
2      fun {Fib N}
3          local FibAux in
4              fun {FibAux N Acc1 Acc2}
5              if N==0 then Acc1+Acc2
6              else {FibAux N Acc1+Acc2 Acc1}
7              end
8            {FibAux N 0 1}
9        end
10       end
```

```
11     R = {Fib 9}
12
13 end
```

```
 ERROR: A " end " statement is missing in the if statement. [if <
 condition> then <statement> (else <statement>) end]. Error found in " if
  N == 0 then Acc1 + Acc2 else { FibAux N Acc1 + Acc2 Acc1 }".

ERROR: Your program runs infinitely, it can be from one of these parts of
 the program:
[ fun { FibAux N Acc1 Acc2 } if N == 0 then Acc1 + Acc2 else { FibAux N
Acc1 + Acc2 Acc1 } end]

You call Argument "Acc1+Acc2" with value "1" at position "2" in method "
FibAux" 2 times. Maybe you forgot to update it.

You call Argument "Acc1" with value "1" at position "3" in method "FibAux
" 2 times. Maybe you forgot to update it.

You call Argument "N" with value "9" at position "1" in method "FibAux"
363 times. Maybe you forgot to update it.

HINT: An " else " statement is missing in your if structure. it is better
 to have a default case. Error found in " if N == 0 then Acc1 + Acc2 else
 { FibAux N Acc1 + Acc2 Acc1 }".
```

Example 35 :

```
1  local Fib R in
2      fun {Fib N}
3          local FibAux in
4              fun {FibAux N Acc1 Acc2}
5              if N == 0 then Acc1
6              else {FibIter N-1 Acc2 Acc1+Acc2} end
7              end
8            {FibAux N 0 1}
9        end
10       end
11     R = {Fib 9}
12     {Browse R}
13 end
```

```
ERROR :  The variable is not declared. The variable(s) concerned :[
FibIter]. Error found in " FibIter".

ERROR :  The procedure you want to call is not declared. Maybe did you
```

```
mean one of these : {FibAux, Fib}.
The variable(s) concerned :[FibIter]. Error found in " { FibIter N - 1
Acc2 Acc1 + Acc2 }".

HINT : The variable FibIter is not used. It is probably useless.
```

0.0.7 Edx7

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx7 -d false -nVar 2 -nFun1 -nCase 1 -nilCase true -list true -extProc [Append]**, corresponding to the command line for the exercise number 7 of the course.

Example 36 :

```
1  local AppendLists R in
2
3     fun {AppendLists L1 L2}
4        case L1 of nil then L2
5         H|T then H|{AppendLists T L2}
6        end
7     end
8
9  R={AppendLists [1] [2]}
10 end
```

```
HINT :  A " [] " or an " else " statement is missing in the case
structure. [case <expression> of <pattern> then <statement> { [] <pattern
> then <statement> } || { else <statement> } end]. Error found in " case
L1 of nil then L2 H | T then H | { AppendLists T L2 } end".

HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[L1 ([ 1 ] )]. Error found in " case L1 of nil then L2 H | T then H | {
AppendLists T L2 } end".


HINT :  Some functions are not tail recursive.[AppendLists]
```

Example 37 :

```
1  local AppendLists R in
2
3     fun {AppendLists L1 L2}
4        case L1 of nil then L2
```

```
5        [] H|T then H|{AppendLists L1 L2}
6        end
7     end
8
9  R={AppendLists [1] [2]}
10 end
```

```
ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { AppendLists L1 L2 } case L1 of nil then L2 [] H | T then H | {
AppendLists L1 L2 } end end]
You call Argument "L2" with value "[ 2 ]" at position "2" in method "
AppendLists" 308 times. Maybe you forgot to update it.
You call Argument "L1" with value "[ 1 ]" at position "1" in method "
AppendLists" 308 times. Maybe you forgot to update it.


===== HINTS =====

HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ?
The arguments are : [Argument "L1" with value "[ 1 ]" at position "1" in
method "AppendLists", Argument "L2" with value "[ 2 ]" at position "2" in
method "AppendLists"]
The variable(s) concerned :[AppendLists]. Error found in " { AppendLists
L1 L2 }".


HINT : The variable T is not used. It is probably useless.


HINT :  Some functions are not tail recursive.[AppendLists]
```

Example 38 :

```
1  local AppendLists R in
2
3     fun {AppendLists L1 L2}
4        {Append L1 L2}
5     end
6
7  R={AppendLists [1] [2]}
8  end
```

```
ERROR :  You cannot use this predifined function.
The variable(s) concerned :[Append]. Error found in " { Append L1 L2 }".
```

```
HINT :  You do not use enough case statements for the exercise.
We expect you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement.
You should at least have one to succeed this exercise.
```

Example 39 :

```
1  local AppendLists R in
2
3     fun {AppendLists L1 L2}
4           L1|L2
5     end
6
7  R={AppendLists [1] [2]}
8  end
```

```
HINT :  You do not use enough case statements for the exercise.
We expect you to use at least 1 case statement(s).


HINT :  You do not have a " case <var> of nil " statement.
You should at least have one to succeed this exercise.
```

Our feedback, even if they seem silly at the first sight can help the student to understand that he is not in the right direction.

Example 40 :

```
1  local AppendLists R in
2
3     fun {AppendLists L1 L2}
4        if {L1==L2==nil} then nil
5        elseif {L1 == nil} then L2
6        elseif {L2 == nil} then L1
7        else L1 | L2| nil
8        end
9
10    end
11
12 R={AppendLists [1] [2]}
13 end
```

```
ERROR : Your program contains some errors

ERROR :  One of the term is not a number nor a boolean. Error found in "
L1 == L2 == nil".

ERROR :  The procedure you want to call is not declared. Maybe did you
mean one of these : {AppendLists}.
The variable(s) concerned :[L1==L2==nil]. Error found in " { L1 == L2 ==
nil }".

ERROR :  The Condition is not a boolean. The value is null.  Error found
in " if { L1 == L2 == nil } then nil elseif { L1 == nil } then L2 elseif
{ L2 == nil } then L1 else L1 | L2 | nil end".

HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

Example 41 :

```
1  local AppendLists R in
2
3     fun {AppendLists L1 L2}
4        local Addelement List Newlist in
5            fun {Addelement List Newlist}
6                if List == nil then Newlist|L2
7                 else {Addelement List.2 Newlist|List.1}
8                 end
9             end
10            {Addelement  L1 L2}
11       end
12    end
13
14 R={AppendLists [1] [2]}
15 end
```

```
ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { Addelement List Newlist } if List == nil then Newlist | L2 else {
 Addelement List . 2 Newlist | List . 1 } end end,

The list you build does not contain a nil : Newlist|List.1]

HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 2 variables instead of
 5.
```

```
HINT :  You do not use enough case statements for the exercise.
We expect you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement.
You should at least have one to succeed this exercise.
```

Example 42 :

```
1  local AppendLists R in
2
3     fun {AppendLists L1 L2}
4        if L1==nil then L2
5        else L1|L2.1 {AppendLists L1 L2.2}
6        end
7     end
8
9  R={AppendLists [123] [3 4 5]}
10 end
```

```
ERROR : Your program contains some errors

ERROR :  You are trying to access the field of an element which is not a
record. The variable(s) concerned :[L2 (nil]. Error found in " L1 | L2 .
1".

ERROR :  You are trying to access the field of an element which is not a
record. The variable(s) concerned :[L2 (nil)]. Error found in " {
AppendLists L1 L2 . 2 }".

HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

Example 43 :

```
1  local AppendLists R in
2
3  fun{AppendLists L1 L2}
4  L1 | L2 | nil;
5  end
6
7  R = {AppendLists [1 2 3] [4 5 6]}
8  end
```

```
ERROR :  The end of a statement is not marked by a ';' in Oz. Error found
 in " L1 | L2 | nil ;".

HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

0.0.8 Edx8

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx8 -d false -nVar 3 -nFun2 -nIf 1 -list true -tailRec true**, corresponding to the command line for the exercise number 8 of the course.

Example 44 :

```
1  local Fact S in
2     fun {Fact N}
3        fun {Fact N Acc}
4              if N<2 then Acc
5              else {Fact N-1 N*Acc} end
6         end
7        in
8           case N of nil then nil
9                 [] H|T then {Fact H 1}|{Fact T}
10           end
11    S = {Fact 12}
12 end
```

```
ERROR : These functions are not tail recursive : [Fact]

ERROR :  A " end " statement is missing in the case structure. [case <
expression> of <pattern> then <statement> end]. Error found in " case N
of nil then nil [] H | T then { Fact H 1 } | { Fact T }".

ERROR :  The procedure/variable already exists. Maybe the procedure you
want to implement is already implemented. Otherwise, change the name of
your procedure. The variable(s) concerned :[Fact]. Error found in " fun {
 Fact N Acc } if N < 2 then Acc else { Fact N - 1 N * Acc } end end".

HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[N (12 )]. Error found in " case N of nil then nil [] H | T then { Fact
H 1 } | { Fact T }".
```

Example 45 :

```
1
2  local Fact S in
3     fun {Fact N}
4        fun {Fact2 N A}
5           if N<2 then Acc
6            else {Fact2 N-1 N*A} end
7        end
8        in
9           case L of nil then nil
10                   [] H|T then {Fact2 H 1}|{Fact T}
11          end
12    end
13    S = {Fact 3}
14 end
```

```
ERROR : These functions are not tail recursive : [Fact]

ERROR : Your program contains some errors

ERROR :  The variable is not declared. The variable(s) concerned :[L].
Error found in " L".

HINT : The variable Acc is not used. It is probably useless.

HINT : The variable L is not used. It is probably useless.
```

Example 46 :

```
1
2  local Fact S in
3     fun {Fact N}
4        local fct in
5           fun {fct A B Acc}
6               if A == N then Acc
7               else {fct A+1 A*B {Append A*B Acc}}
8               end
9           end
10          {fct 1 1 nil}
11       end
12    S = {Fact 3}
13 end
```

```
ERROR :  A " end " statement is missing in the local declaration. [local
X1.....XN in <statement> end]. Error found in " local fct".
```

```
ERROR : A " in " statement is missing in the local declaration. [local
X1.....XN in <statement> end]. Error found in " local fct".

ERROR : A variable should always start with an upper case. Error found
in " fun { fct A B Acc } if A == N then Acc else { fct A + 1 A * B {
Append A * B Acc } } end end".

ERROR : The variable is not declared. The variable(s) concerned :[fct].
Error found in " fun { fct A B Acc } if A == N then Acc else { fct A + 1
A * B { Append A * B Acc } } end end".

ERROR : One of the two arguments given to the function Append is not a
list. The variable(s) concerned :[A*B (1 ), Acc (nil) ]. Error found in "
 { Append A * B Acc }".

ERROR : One of the two arguments given to the function Append is not a
list. The variable(s) concerned :[A*B (2 ), Acc (arg6) ]. Error found in
" { Append A * B Acc }".

ERROR : You have a deadlock, your variable is not initialized and you
only have one thread. The variable(s) concerned :[Acc]. Error found in "
Acc".
```

Example 47 :

```
1
2  local Fact S in
3     fun {Fact N}
4        local
5            fun {Factorial N A}
6                if N == 1 then A
7                 else {Factorial N - 1 A * N}
8                end
9            end
10
11       fun {FactorialAux A L}
12         if A == 1 then {Append L 1}
13         else {Factorial A-1 {Append L Factorial{A 1}}}
14         end
15     end
16 in
17      {List.reverse {Fact N nil}}
18 end
19       end
20    S = {Fact 8}
21 end
```

```
ERROR :  The variable is not declared. The variable(s) concerned :[
Factorial]. Error found in " fun { Factorial N A } if N == 1 then A else
{ Factorial N - 1 A * N } end end".

ERROR :  The variable is not declared. The variable(s) concerned :[
FactorialAux]. Error found in " fun { FactorialAux A L } if A == 1 then {
 Append L 1 } else { Factorial A - 1 { Append L Factorial { A 1 } } } end
 end".

ERROR :  Too many arguments for the procedure. The variable(s) concerned
:[Fact]. Error found in " { Fact N nil }".

ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { Factorial N A } if N == 1 then A else { Factorial N - 1 A * N }
end end,  fun { Fact N } local fun { Factorial N A } if N == 1 then A
else { Factorial N - 1 A * N } end end fun { FactorialAux A L } if A == 1
 then { Append L 1 } else { Factorial A - 1 { Append L Factorial { A 1 }
} } end end in { List . reverse { Fact N nil } } end end]
You call Argument "N" with value "8" at position "1" in method "Fact" 338
 times. Maybe you forgot to update it.

HINT : The variable List is not used. It is probably useless.

HINT : The variable FactorialAux is not used. It is probably useless.
```

Example 48 :

```
1
2  local Fact S in
3     fun {Fact N}
4        local FactN L J in
5              fun {FactN I A}
6                if I==0 then A
7                else {FactN I-1 I*A}
8                end
9              end
10           if N==0 then L
11             else
12               J={FactN N 1}
13             end
14           end
15       end
16    S = {Fact 8}
17 end
```

```
HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 3 variables instead of
```

```
5.

HINT :  You have to use some list structure in this exercise.
```

0.0.9 Edx9

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx9 -d false -nVar 3 -nFun2 -nIf 1 -list true -tailRec true**, corresponding to the command line for the exercise number 9 of the course.

Example 50 :

```
1
2  local Prefix FindString R in
3
4  declare
5  fun {Prefix L1 L2}
6     case L1#L2
7     of nil#Ys then true
8     [] Xs#nil then false
9     [] nil#nil then true
10    [] (X|Xs)#(Y|Ys) then
11       if X==Y then {Prefix Xs Ys}
12       else false end
13    end
14 end
15
16
17 declare
18 fun {FindString L1 L2}
19    if {Prefix L1 L2} then true
20    else
21       case L2
22       of nil then false
23       [] (X|Xs) then {FindString L1 Xs}
24       end
25    end
26 end
27 R = {FindString [1 4] [2 1 4 6]}
28 end
```

```
ERROR : Your program contains some errors

ERROR :  You should use a local instead of declare.  Error found in "
declare fun { Prefix L1 L2 } case L1 # L2 of nil # Ys then true [] Xs #
nil then false [] nil # nil then true [] ( X | Xs ) # ( Y | Ys ) then if
```

```
X == Y then { Prefix Xs Ys } else false end end end declare fun {
FindString L1 L2 } if { Prefix L1 L2 } then true else case L2 of nil then
 false [] ( X | Xs ) then { FindString L1 Xs } end end end".

ERROR :  You should use a local instead of declare.  Error found in "
declare fun { FindString L1 L2 } if { Prefix L1 L2 } then true else case
L2 of nil then false [] ( X | Xs ) then { FindString L1 Xs } end end end
".

ERROR :  The Condition is not a boolean. The value is null.  Error found
in " if { Prefix L1 L2 } then true else case L2 of nil then false [] ( X
| Xs ) then { FindString L1 Xs } end end".

HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[L1#L2 (nil#[ 6 ] )]. Error found in " case L1 # L2 of nil # Ys then
true [] Xs # nil then false [] nil # nil then true [] ( X | Xs ) # ( Y |
Ys ) then if X == Y then { Prefix Xs Ys } else false end end".
```

Example 51 :

```
1
2  local Prefix FindString R in
3
4  fun{Prefix L1 L2}
5      case L1#L2 of nil#L2 then true
6      [](H1|T1)#(H2|T2) then
7          if H1==H2 then {Prefix T1 T2}
8          else false
9          end
10     end
11 end
12 fun{FindString L1 L2}
13     if {Prefix L1 L2} then true
14     else
15         case L1#L2 of nil#nil then true
16         [] L1#(H2|T2) then {FindString L1 T2}
17         else false
18         end
19     end
20 end
21 R = {FindString [1 4] [2 1 4 6]}
22 end
```

```
ERROR : Your program contains some errors

ERROR :  The Condition is not a boolean. The value is null.  Error found
in " if { Prefix L1 L2 } then true else case L1 # L2 of nil # nil then
true [] L1 # ( H2 | T2 ) then { FindString L1 T2 } else false end end".
```

```
HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[L1#L2 (nil#[ 6 ] )]. Error found in " case L1 # L2 of nil # L2 then
true [] ( H1 | T1 ) # ( H2 | T2 ) then if H1 == H2 then { Prefix T1 T2 }
else false end end".
```

0.0.10 Edx10

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx10 -d false -nFun 2 -nCase 1 -nilCase true -list true -tailRec true -extProc [Flatten]** , corresponding to the command line for the exercise number 10 of the course.

Example 52 :

```
1  local FlattenList S in
2     fun {FlattenList L}
3        local FlattenListBis in
4           fun {FlattenListBis L Acc}
5                 case L of M then M|Acc
6                 [] nil then Acc
7                 [] H|T then {FlattenListBis T {Append
8  {FlattenListBis H nil} Acc}}
9                 end
10          end
11             {FlattenListBis L nil}
12       end
13    end
14    S= {FlattenList [[1 2] [5 6]]}
15 end
```

```
ERROR :  You should use the general pattern matching as the last case of
your case statement otherwise it will match every time and the rest of
the case is a dead code. The case statement is 'case L of M'. Error found
 in " case L of M then M | Acc [] nil then Acc [] H | T then {
FlattenListBis T { Append { FlattenListBis H nil } Acc } } end".

ERROR :  The variable is not declared. The variable(s) concerned :[L].
Error found in " L".

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread. The variable(s) concerned :[L]. Error found in "
case L of M then M | Acc [] nil then Acc [] H | T then { FlattenListBis T
 { Append { FlattenListBis H nil } Acc } } end".
```

Example 53 :

```
local FlattenList S in
   fun {FlattenList L}
      case L of nil then L
         [] H|T then {Append H {FlattenList T}}
      end
   end
   S= {FlattenList [ 1 [a2 3] [4] [[5 6]]]}
end
```

```
ERROR: One of the two arguments given to the function Append is not a
list.
The variable(s) concerned:[H (1 ), {FlattenListT} ([ 2 3 4 [ 5 6 ] ]) ].
Error found in " { Append H { FlattenList T } }".
```

Example 54 :

```
local FlattenList S in
   fun {FlattenList L}
      case L of H|T then H|{FlattenList T}
            [] H|nil then L
      end
   end
   S= {FlattenList [ 1 [a2 3] [4] [[5 6]]]}
end
```

```
ERROR : These functions are not tail recursive : [FlattenList]

ERROR : Your program contains some errors

HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[L (nil )]. Error found in " case L of H | T then H | { FlattenList T }
[] H | nil then L end".
```

Example 55 :

```
local FlattenList S in
   fun {FlattenList L}
      case L of nil then L
               []H|T then {Append {Flatten H}{Flatten
T}}
            else H|{Flatten T}
      end
   end
   S= {FlattenList [ 1 [a2 3] [4] [[5 6]]]}
end
```

```
ERROR: You cannot use this predifined function. The variable(s) concerned
 :[Flatten]. Error found in " { Flatten H }".
```

0.0.11 Edx11

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx11 -d false -nFun 2 -nCase 1 -list true**, corresponding to the command line for the exercise number 11 of the course.

Example 56 :

```
local FunnyFunc R in
   fun {FunnyFunc FunL L}
       case L of H|T then {FunL.1 H}|{FunnyFunc FunL.2 T}
       else nil
       end
   end

   R = {FunnyFunc [fun {$ X} X+1 end fun {$ X} X*2 end] [2 2 2]}
end
```

```
ERROR : Your program contains some errors

ERROR :  You are trying to access the field of an element which is not a
record.
The variable(s) concerned :[FunL (nil) ]. Error found in " { FunL . 1 H
}".

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of these : {FunnyFunc}.The variable(s) concerned
:[FunL.1].
Error found in " { FunL . 1 H }".
```

```
ERROR :  You are trying to access the field of an element which is not a
record.
 The variable(s) concerned :[FunL (nil) ]. Error found in " { FunnyFunc
 FunL . 2 T }".


HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times
the function. Or maybe did you forget to update the argument ?
The arguments are : [Argument "H" with value "2" at position "1" in
method "$"]
The variable(s) concerned :[FunL.1]. Error found in " { FunL . 1 H }".


HINT :  You seem to have declare to many functions for the exercice.
Maybe some functions are not needed.We expected 2 functions instead of 3.


HINT :  Some functions are not tail recursive.[FunnyFunc]
```

Example 57 :

```
1  local FunnyFunc Test in
2     fun {FunnyFunc FunL L}
3         case L of H|T then {FunL.1 H}|{FunnyFunc FunL.2 T}
4         else nil
5         end
6     end
7
8      proc {Test FunL L SolL}
9      {Browse {FunnyFunc FunL L} == SolL}
10    end
11
12 {Test [fun($ X) X*X end fun($ X) X*X*X end fun($ X) X*3 end ] [4 2 3] [16
    8 9]}
13 end
```

```
ERROR: A " { " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * X end".

ERROR: A " } " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * X end".

ERROR: A " { " is missing in your procedure/function declaration. Error
```

```
found
in " fun ( $ X ) X * X * X end".

ERROR: A " } " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * X * X end".

ERROR: A " { " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * 3 end".

ERROR: A " } " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * 3 end".
```

0.0.12 Edx12

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx12 -d false -nFun3 -nIf 1 -nCase 1 -list true -bottom true -tailRec true**, corresponding to the command line for the exercise number 12 of the course.

Example 58 :

```
1  local F Build D C in
2     fun {Build D C}
3        fun {$ X}
4           fun {FindX X D C}
5               case D of H|T then
6                 if H==X then C.1
7                 else {FindX X D C.2}
8                 end
9              else bottom
10             end
11          end
12       in {FindX X D C}
13       end
14    end
15    D = [1 2 3]
16    C = [4 5 6]
17    F = {Build D C}
18    {Browse {F 1}}
19    {Browse {F 2}}
20    {Browse {F 3}}
21    {Browse {F 4}}
22 end
```

```
ERROR :  You are trying to access the field of an element which is not a
record. The variable(s) concerned :[C (nil) ]. Error found in " { FindX X
 D C . 2 }".

HINT : The variable T is not used. It is probably useless.
```

0.0.13 Edx13

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx13 -d false -nFun2 -nCase 1 -nilCase true -list true -tailRec true**, corresponding to the command line for the exercise number 13 of the course.

Example 59 :

```
1  local Transform R in
2     fun{Transform L}
3        local TransformValues in
4           fun {TransformValues R F V}
5                 case V of nil then R
6                 [] H|T then
7                     R.(F.1) = {Transform H}
8                     {TransformValues R F V}
9                 end
10          end
11          case L of [G F V] then {TransformValues {Record.make G
12 F} F V}
13          else L
14          end
15       end
16    end
17 R = {Transform [record [3 5 6] [b c d]]}
18 end
```

```
ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { TransformValues R F V } case V of nil then R [] H | T then R . (
F . 1 ) = { Transform H } { TransformValues R F V } end end]
You call Argument "F" with value "[ 3 5 6 ]" at position "2" in method "
TransformValues" 367 times. Maybe you forgot to update it.
You call Argument "V" with value "[ b c d ]" at position "3" in method "
TransformValues" 367 times. Maybe you forgot to update it.
You call Argument "R" with value "record({3=b, 5=_, 6=_})" at position
"1" in method "TransformValues" 366 times. Maybe you forgot to update it.
You call Argument "H" with value "b" at position "1" in method "Transform
" 367 times. Maybe you forgot to update it.
```

```
HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "H" with value "b" at position "1" in method "
Transform"]The variable(s) concerned :[Transform]. Error found in " {
Transform H }".
```

```
HINT :  You call the procedure several times with the same argument(s).
You could maybe store the result in a variable to avoid to call several
times the function. Or maybe did you forget to update the argument ? The
arguments are : [Argument "R" with value "record({3=b, 5=_, 6=_})" at
position "1" in method "TransformValues", Argument "F" with value "[ 3 5
6 ]" at position "2" in method "TransformValues", Argument "V" with value
 "[ b c d ]" at position "3" in method "TransformValues"]The variable(s)
concerned :[TransformValues]. Error found in " { TransformValues R F V
}".
```

```
HINT : The variable T is not used. It is probably useless.
```

Example 60 :

```
1  local Transform R in
2     fun{Transform L}
3        local R Recordness Add Acc
4        in
5            fun {Add R Lol Acc}
6                 case Lol of H|T then
7                    if Acc<{Width R} then R.Acc=Lol.Acc
8                    else R
9                       end
10                end
11             end
12
13           fun {Recordness Lis}
14              R = {Record.make L.1 L.2.1}
15              case L.2.2 of H|T then {Recordness T}
16               else {Add R T 1}
17                 end
18           end
19           {Recordness L}
20    end
21    end
22 R = {Transform [record [3 5 6] [b c d]]}
23 end
```

```
ERROR :  You cannot change the value of the variable because it was
already assigned to one. A variable can only be assigned to one value in
```

```
Oz. Use cell with " :=\ " for multiple assignments. The variable(s)
concerned :[R]. Error found in " R = { Record . make L . 1 L . 2 . 1 }".

ERROR :  Your program runs infinitely, it can be from one of these parts
of the program : [ fun { Recordness Lis } R = { Record . make L . 1 L . 2
 . 1 } case L . 2 . 2 of H | T then { Recordness T } else { Add R T 1 }
end end]

HINT :  A " [] " or an " else " statement is missing in the case
structure. [case <expression> of <pattern> then <statement> { [] <pattern
> then <statement> } || { else <statement> } end]. Error found in " case
Lol of H | T then if Acc < { Width R } then R . Acc = Lol . Acc else R
end end".

HINT : The variable Lis is not used. It is probably useless.

HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

0.0.14 Edx14

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx14 -d false -nVar 5 -nFun3 -nIf 1 -nCase 3 -nilCase true -leafCase true -list true -tailRec true**, corresponding to the command line for the exercise number 14 of the course.

Example 61 :

```
1  local Infix R Tree in
2     fun {Infix Tree}
3        case Tree of nil then nil
4        [] btree(T left:L right:R) then {Append {Infix L} T|{Infix R}}
5        end
6     end
7
8     Tree = btree(4 left:btree(2 left:btree(1 left:leaf right:leaf)
9                                right:btree(3 left:leaf right:leaf))
10                right:btree(5 left:leaf right:leaf))
11
12    R = {Infix Tree}
13 end
```

```
HINT :  The case/if statement does not match any pattern for the
expression.
You should have an else statement. The variable(s) concerned :[Tree (leaf
 )].
Error found in " case Tree of nil then nil [] btree ( T left : L right :
R ) then { Append { Infix L } T | { Infix R } } end".
```

```
HINT :  You do not use enough if statement for the exercise. We expect
you to use at least 1 if statement(s).


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 3 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

more informations

Example 62 :

```
1  local Infix R Tree in
2     fun {Infix Tree}
3
4  declare
5  fun{Infix T Acc}
6     if(T.left\=leaf) then
7        {Infix T.left Acc}
8        Acc|T.1
9     elseif(T.right \= leaf) then
10       {Infix T.right Acc}
11    else
12       Acc
13    end
14 end
15
16 declare
17 T = btree(4 left:btree(2 left:btree(1 left:leaf right:leaf)
18                                    right:btree(3 left:leaf right:leaf))
19       right:btree(5 left:leaf right:leaf))
20 {Browse {Infix T nil}}  end
21
22
23    Tree = btree(1:42 left:btree(1:12 left:btree(1:9 left:leaf right:leaf)
24 right:btree(1:21 left:leaf right:btree(1:24 left:leaf right:btree(1:31
25 left:leaf right:leaf)))) right:leaf)
26
27    R = {Infix Tree}
28 end
```

```
RROR :  You should use a local instead of declare.
Error found in " declare fun { Infix T Acc } if ( T . left \= leaf ) then
 { Infix T . left Acc } Acc | T . 1 elseif ( T . right \= leaf ) then {
Infix T . right Acc } else Acc end end declare T = btree ( 4 left : btree
 ( 2 left : btree ( 1 left : leaf right : leaf ) right : btree ( 3 left :
```

```
 leaf right : leaf ) ) right : btree ( 5 left : leaf right : leaf ) )".

ERROR :  You should use a local instead of declare.  Error found in "
declare T = btree ( 4 left : btree ( 2 left : btree ( 1 left : leaf right
 : leaf ) right : btree ( 3 left : leaf right : leaf ) ) right : btree (
5 left : leaf right : leaf ) )".

ERROR :  The procedure/variable already exists. Maybe the procedure you
want to implement is already implemented. Otherwise, change the name of
your procedure. The variable(s) concerned :[Infix]. Error found in " fun
{ Infix T Acc } if ( T . left \= leaf ) then { Infix T . left Acc } Acc |
 T . 1 elseif ( T . right \= leaf ) then { Infix T . right Acc } else Acc
 end end".

ERROR :  The variable is not declared. The variable(s) concerned :[T].
Error found in " T = btree ( 4 left : btree ( 2 left : btree ( 1 left :
leaf right : leaf ) right : btree ( 3 left : leaf right : leaf ) ) right
: btree ( 5 left : leaf right : leaf ) )".


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 3 case statement(s).


HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

0.0.15 Edx15

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx15 -d false -nFun3 -nIf 1 -nCase 3 -nilCase true -leafCase true -list true -tailRec true**, corresponding to the command line for the exercise number 15 of the course.

Example 63 :

```
1  local FromListToTree R T FromTreeToList S in
2
3
4  fun{FromListToTree L}
5      local AddE AddAllL in
6          fun{AddE T E}
7              case T
8              of leaf then btree(E left:leaf right:leaf)
9              [] btree(K left:T1 right:T2) andthen K==E then
10                 T
```

```
11              [] btree(K left:T1 right:T2) andthen K>E then
12                  btree(K left:{AddE T1 E} right:T2)
13              [] btree(K left:T1 right:T2) andthen K<E then
14                  btree(K left:T1 right:{AddE T2 E})
15              end
16          end
17          fun{AddAllL L T}
18              if L==nil then T
19              else {AddAllL L.2 {AddE T L.1}} end
20          end
21          {AddAllL L leaf}
22      end
23  end
24
25  fun{FromTreeToList T}
26      local RemoveSmallest DoAllT in
27          fun{RemoveSmallest T}
28              case T
29              of leaf then none
30              [] btree(K left:T1 right:T2) then
31                  case {RemoveSmallest T1}
32                  of none then double(T2 K)
33                  [] double(Ts Xs) then
34                      double(btree(K left:Ts right:T2) Xs)
35                  end
36              end
37          end
38          fun{DoAllT T}
39              case {RemoveSmallest T}
40              of none then nil
41              [] double(Ts Xs)
42                  Xs|{DoAllT Ts}
43              end
44          end
45          {DoAllT T}
46      end
47  end
48
49      R = {FromListToTree [42 21 12 34 9]}
50      T = btree(1:42 left:btree(1:9 left:leaf right:leaf) right:leaf)
51      S = {FromTreeToList T}
52
53
54  end
```

```
ERROR : These functions are not tail recursive : [DoAllT]

ERROR :  A " then " statement is missing in the case structure. [case <
expression> of <pattern> then <statement> end]. Error found in " case {
RemoveSmallest T } of none then nil [] double ( Ts Xs ) Xs | { DoAllT Ts
} end".
```

```
HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[{RemoveSmallestT} (double({1=btree({1=42, left=leaf, right=leaf}),
2=9}) )]. Error found in " case { RemoveSmallest T } of none then nil []
double ( Ts Xs ) Xs | { DoAllT Ts } end".


HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 5 variables instead of
 9.


HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

0.0.16 Edx2_1

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_1 -d false -nFun3 -nIf 1 -nCase 2 -leafCase true -tailRec true**, corresponding to the command line for the exercise number 1 of the second course.

Example 65 :

```
1  local Reverse L  Y Z R T S in
2     fun{Reverse L}
3        local X in
4            {NewCell nil X}
5           for E in L do
6              X:=E|@X
7              end
8              X
9        end
10    end
11
12 L ={Reverse [1 2 3]}
13 {Browse L.1}
14 end
```

```
ERROR :  You are trying to access the field of an element which is a cell
.
You probably want to access to the value of the cell which is a record.
Try with the " @ " operator. The variable(s) concerned :[L (OzCell : [ 3
2 1 ] )].
Error found in " { Browse L . 1 }".
```

Example 66 :

```
local Reverse Z in
   fun{Reverse L}
      local X in
          {NewCell nil X}
         for E in L do
            X:=E|X
            end
            @X
      end
   end

Z= {Reverse [1 2 3]}
{Browse Z}

end
```

```
ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[The list you build does not contain a nil : E|X]


HINT :  You do not use enough if statement for the exercise. We expect
you to use at least 1 if statement(s).


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 2 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

Example 67 :

```
local Reverse Y in
   Y = {NewCell 128}
   Y:= Y+12

end
```

```
ERROR :  One of the term of you operation is a cell. Use the " @ "
operator to access its content.
The variable(s) concerned :[Y]. Error found in " Y + 12".
```

```
HINT :  You do not use enough if statement for the exercise. We expect
you to use at least 1 if statement(s).


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 2 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

Example 68

```
1  local Reverse L in
2     fun{Reverse L}
3        for E H T in L do
4            if L==nil then nil
5            elseif L==H|T then E={Reverse T}|H
6            end
7        end
8     end
9  L ={Reverse [1 2 3]}
10 end
```

```
ERROR : These functions are not tail recursive : [Reverse]

ERROR :  You can only use one variable in the for loop. The correct
syntax is "for <var> in <var> do <statement> end". Error found in " E H T
 in L".

ERROR :  The variable is not declared. The variable(s) concerned :[E].
Error found in " E".

ERROR :  The variable is not declared. The variable(s) concerned :[H].
Error found in " H".

ERROR :  The variable is not declared. The variable(s) concerned :[T].
Error found in " T".


HINT :  An " else " statement is missing in your if structure.
it is better to have a default case. Error found in " if L == nil then
nil elseif L == H | T then E = { Reverse T } | H end".


HINT :  You do not use enough case statements for the exercise.
We expect you to use at least 2 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement.
```

```
You should at least have one to succeed this exercise.
```

Example 69

```
1  local Reverse L in
2     fun{Reverse L}
3
4  L = {NewCell nil}
5  for E in L do
6      if (E==nil) then @L
7      L=@L|E
8  end
9     end
10
11 L ={Reverse [1 2 3]}
12 end
```

```
ERROR : Your program contains some errors

ERROR :  A " end " statement is missing in the if statement.
[if <condition> then <statement> (else <statement>) end].
Error found in " if ( E == nil ) then @ L".

ERROR :  You cannot change the value of the variable because
it was already assigned to one. A variable can only be assigned
to one value in Oz. Use cell with " :=\ " for multiple assignments.
The variable(s) concerned :[L]. Error found in " L = { NewCell nil }".

ERROR :  You want to iterate over the value of a cell. Are you sure you
did not forget to use the " @ " operator ? Error found in " E in L".

ERROR :  The variable is not declared. The variable(s) concerned :[E].
Error found in " E".

HINT :  An " else " statement is missing in your if structure. it is
better to have a default case. Error found in " if ( E == nil ) then @ L
".


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 2 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

0.0.17 Edx2_2

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_2 -d false -nFun 1 -tailRec true -cell true**, corresponding to the command line for the exercise number 2 of the second course.

Example 70 :

```
local Shuffle R in
    fun {Shuffle L}
      local
            Size = {Length L}
            Array = {NewArray 1 Size 0}
            Output = {NewCell nil}
        in
            for I in 1..Size do
            Array.I:= {Nth L I}
         end
            local
                Max = {NewCell Size}
            in
                for I in 1..Size do
                    local RandomNumber = {OS.rand} mod @Max + 1 in
                    Output:= Array.RandomNumber | @Output
                    Array.RandomNumber:= Array.@Max
                    Max:= @Max - 1
                    end
                end
            end
            @Output
      end
   end

R:= {Shuffle [1 2 3 4 5]}
{Browse R}
end
```

```
ERROR :  The variable is not a cell/attribute. You cannot use the " := "
operator.
The variable(s) concerned :[R]. Error found in " R := { Shuffle [ 1 2 3 4
 5 ] }".
```

Example 71 :

```
local Shuffle R in
    fun {Shuffle L}
      local
            Size = {Length L}
            Array = {NewArray 1 Size 0}
            Output = {NewCell nil}
        in
            for I in 1..Size do
            Array.I:= {Nth L I}
         end
            local
                Max = {NewCell Size}
            in
                for I in 1..Size
                    local RandomNumber = {OS.rand} mod @Max + 1 in
                    Output:= Array.RandomNumber | @Output
                    Array.RandomNumber:= Array.@Max
                    Max:= @Max - 1
                    end
                end
            end
            @Output
       end
   end

R:= {Shuffle 1}
{Browse R}
end
```

```
ERROR : Your program contains some errors

ERROR :  A " do " statement is missing in the for/while construction. [
for/while <loopDec> do <statement>]. Error found in " for I in 1 .. Size
local RandomNumber = { OS . rand } mod @ Max + 1 in Output := Array .
RandomNumber | @ Output Array . RandomNumber := Array . @ Max Max := @
Max - 1 end end".

ERROR :  The variable is not a cell/attribute. You cannot use the " := "
operator. The variable(s) concerned :[R]. Error found in " R := { Shuffle
 1 }".

ERROR :  The argument of Length is not a list nor a string. The variable(
s) concerned :[L (1) ]. Error found in " { Length L }".

HINT :  You have to use some cell structure in this exercise.
```

0.0.18 Edx2_3

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_3 -d false -nFun1 -nFun 4 -nilCase true -list true -cell true**, corresponding to the command line for the exercise number 3 of the second course.

Example 72 :

```
1  local NewStack IsEmpty Push Pop Eval R in
2     fun {NewStack}
3        {NewCell nil}
4     end
5
6        fun {IsEmpty S}
7        case S of nil then true
8        else false
9        end
10       end
11
12    proc {Push S X}
13       S:= X|@S
14    end
15
16    fun {Pop S}
17       if {IsEmpty S} then empty
18       else
19          case @S of H|T then
20             S:= T
21             H
22          else
23             empty
24             end
25          end
26       end
27
28    fun {Eval L}
29       local S={NewStack} in
30          for E in L do
31             case E
32             of '+' then {Push S {Pop S}+{Pop S}}
33             [] '-' then {Push S ~{Pop S}+{Pop S}}
34             [] '*' then {Push S {Pop S}*{Pop S}}
35             [] '/' then
36                local Tmp = {Pop S} in
37                   {Push S {Pop S} div Tmp}
38                   end
39             else {Push S E}
40             end
41          end
42          {Pop S}
43       end
```

```
44      end
45         R = {Eval [1 8 '+' 4 '*' 10 '/'] }
46      {Browse R}
47   end
```

```
ERROR :  You are trying to perform a pattern matching on a cell. Are you
sure you did not forget to use the operator to perform the pattern
matching on the value of the cell ? Error found in " case S of nil then
true else false end".
```

0.0.19 Edx2_4

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_4 -d false -nClass 1 -list true**, corresponding to the command line for the exercise number 4 of the second course.

Example 73 :

```
1   local Collection C C2 X in
2      class Collection
3         attr list
4
5         meth init
6            list:= nil
7         end
8
9         meth put(X)
10           list:=X|@list
11        end
12
13        meth isEmpty($)
14           @list==nil
15        end
16
17     meth get($)
18        case @list of H|T then list:=T
19                               H
20        [] nil then notfound
21        end
22     end
23
24     meth browse
25        {Browse @list}
26     end
27
28
```

```
29      meth union(C)
30        if {C isEmpty($)}==false then
31           {self put({C get($)})}
32           {self union(C)}
33        end
34     end
35
36     end
37
38     C = {New Collection init}
39     {C put 3}
40
41
42
43  end
```

```
===== ERRORS FOUND =====
ERROR: You have to put the argument of the method between parenthesis: {C
put(3)}. Error found in " { C put 3 }".

ERROR: Not the right number of arguments for the method. Error found in "
 { C
put 3 }".
```

Example 74

```
1  local Collection C C2 X in
2     class Collection
3        attr list
4
5        meth init
6           list:= nil
7        end
8
9        meth put(X)
10          list:=X|@list
11       end
12
13       meth isEmpty($)
14          @list==nil
15       end
16
17    meth get($)
18       case @list of H|T then list:=T
19                             H
20       [] nil then notfound
21       end
22    end
23
24    meth browse
```

```
25        {Browse @list}
26     end
27
28
29      meth union(C)
30        if {C isEmpty($)}==false then
31           {self put({C get($)})}
32           {self union(C)}
33        end
34     end
35
36     end
37
38     C = {New Collection init}
39     {C put}
40
41  end
```

```
===== ERRORS FOUND =====
ERROR: Not the right number of arguments for the method. Error found in "
 { C
put }".
```

0.0.20 Edx2_5

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_5 -d false -nClass 1 -list true**, corresponding to the command line for the exercise number 5 of the second course.

Example 75 :

```
1  local Constant Variable VarX VarY Addition Substraction Multiplication
   Division
2  in
3        class Constant
4           attr c
5           meth init(X) c:=X end
6           meth evaluate($) @c end
7        end
8
9        class Variable
10          attr v
11          meth init(X) c:=X end
12          meth set(X) c:=X end
13          meth evaluate($) @c end
14       end
15
```

```
16        class Addition
17           attr a b
18           meth init(X Y) a:=X b:=Y end
19           meth evaluate($) {@a evaluate($)}+{@b evaluate($)} end
20        end
21
22        class Substraction
23           attr a b
24           meth init(X Y) a:=X b:=Y end
25           meth evaluate($) {@a evaluate($)}-{@b evaluate($)} end
26        end
27
28        class Multiplication
29           attr a b
30           meth init(X Y) a:=X b:=Y end
31           meth evaluate($) {@a evaluate($)}*{@b evaluate($)} end
32        end
33
34        class Division
35           attr a b
36           meth init(X Y) a:=X b:=Y end
37           meth evaluate($) {@a evaluate($)} div {@b evaluate($)}
38    end
39        end
40
41        VarX = {New Variable init(5)}
42          VarY = {New Variable init(6)}
43          local
44              Result
45              C = {New Constant init(6)}
46              Expr1 = {New Addition init(VarX VarY)}
47              Expr2 = {New Division init(Expr1 C)}
48          in
49              {VarX set(6)}
50              {VarY set(2)}
51              {Expr2 evaluate(Result)}
52              {Browse Result}
53          end
54
55
56    end
```

```
===== ERRORS FOUND =====
ERROR: The cell/attribute c is not declared. Error found in " c:= X".

ERROR: One of the term is not a number Error found in " { @ a evaluate (
$ ) }
+ { @ b evaluate ( $ ) }".

ERROR: One of the two terms is not a number. Error found in " { @ a
evaluate (
```

```
$ ) } div { @ b evaluate ( $ ) }".

ERROR: the variable Result is not declared. Error found in " { Browse
Result
}".
```

Example 76 :

```
1  local Constant Variable VarX VarY Addition Substraction Multiplication
   Division
2  in
3         class Constant
4            attr c
5            meth init(X) c:=X end
6            meth evaluate($) @c end
7         end
8
9         class Variable
10           attr v
11           meth init(X) v:=X end
12           meth set(X) v:=X end
13           meth evaluate($) @v end
14        end
15
16        class Addition
17           attr a b
18           meth init(X Y) a:=X b:=Y end
19           meth evaluate($) {a evaluate($)}+{@b evaluate($)} end
20        end
21
22        class Substraction
23           attr a b
24           meth init(X Y) a:=X b:=Y end
25           meth evaluate($) {@a evaluate($)}-{@b evaluate($)} end
26        end
27
28        class Multiplication
29           attr a b
30           meth init(X Y) a:=X b:=Y end
31           meth evaluate($) {@a evaluate($)}*{@b evaluate($)} end
32        end
33
34        class Division
35           attr a b
36           meth init(X Y) a:=X b:=Y end
37           meth evaluate($) {@a evaluate($)} div {@b evaluate($)}
38 end
39        end
40
41        VarX = {New Variable init(0)}
42          VarY = {New Variable init(0)}
```

```
43          local
44              Result
45              C = {New Constant init(6)}
46              Expr1 = {New Addition init(VarX VarY)}
47              Expr2 = {New Division init(Expr1 C)}
48          in
49              {VarX set(1)}
50              {VarY set(4)}
51              {Expr2 evaluate(Result)}
52              {Browse Result}
53          end
54
55
56  end
```

```
===== ERRORS FOUND =====
ERROR: You are trying to call a method on a cell. did not you forget to
access
the value of the cell thanks to the "@" operator ? Error found in " { a
evaluate
( $ ) }".
```

0.0.21 Edx2_7

Here follow some examples of code and feedback given when executing the CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_7 -d false -nFun 1 -nCall 4 -list true -cell true**, corresponding to the command line for the exercise number 7 of the second course.

Example 77 :

```
1   local Reverse in
2     fun {Reverse S}
3
4        local I={NewCell S} in
5           local Temp={NewCell nil} in
6              for C in I do
7                  Temp:=C|@Temp
8              end
9              @Temp
10          end
11       end
12
13    end
14
15    {Browse {Reverse [1 2 3 4 5]}}
16  end
```

```
===== ERRORS FOUND =====
ERROR: You want to iterate over the value of a cell. Are you sure you did
 not
forget to use the "@" operator ? Error found in " C in I".
```

Even if the student made a mistake by iterating on the cell instead of the value of the cell, we do it for him and the program continues to execute normally, searching for other mistakes.

0.0.22 Edx2_8

Here follow some examples of code and feedback given when executing the CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_8 -d false -nFun 1 -nCall 4 -list true -cell true**, corresponding to the command line for the exercise number 8 of the second course.

```
1  local Stack Queue MyQ MyS Size in
2
3     class Stack
4        attr s
5        meth init s:=nil end
6        meth size($) {Length @s} end
7        meth isEmpty($) @s==nil end
8        meth browse {Browse @s} end
9        meth top($)
10           case @s
11           of nil then raise emptyStackException() end
12           else @s.1
13           end
14       end
15       meth push(X) s:=X|@s end
16       meth pop($)
17           case @s
18           of nil then raise emptyStackException() end
19           [] H|T then s:=T H
20           end
21       end
22    end
23
24
25
26    class Queue
27       attr q
28       meth init q:=nil end
29       meth size($) {Length @q} end
30       meth isEmpty($) @q==nil end
31       meth front($)
32           case @q
33           of nil then raise emptyQueueException() end
34           else {Reverse @q}.1
```

```
35              end
36          end
37          meth enqueue(X) q:=X|@q end
38          meth browse {Browse @q} end
39          meth dequeue($)
40              case @q of nil then raise emptyQueueException() end
41              else
42                  local Temp={Reverse @q} in
43                      q:={Reverse Temp.2}
44                      Temp.1
45                  end
46              end
47          end
48      end
49
50      MyQ = {New Queue init}
51      {MyQ size(Size)}
52      {Browse Size}
53      {Browse {MyQ isEmpty($)}}
54      {MyQ enqueue(3)}
55      {Browse {MyQ size($)}}
56      {Browse {MyQ isEmpty($)}}
57      {MyQ enqueue(5)}
58      {Browse {MyQ size($)}}
59      {Browse {MyQ isEmpty($)}}
60      {Browse {MyQ front($)}}
61      {Browse {MyQ dequeue($)}}
62      {MyQ browse}
63      {Browse {MyQ dequeue($)}}
64      {MyQ browse}
65      {Browse {MyQ front($)}}
66      {Browse {MyQ dequeue($)}}
67
68      MyS = {New Stack init}
69      {Browse {MyS size($)}}
70      {Browse {MyS isEmpty($)}}
71      {MyS push(5)}
72      {MyS push(6)}
73      {MyS push(7)}
74      {Browse {MyS pop($)}}
75      {Browse {MyS size($)}}
76      {Browse {MyS isEmpty($)}}
77      {Browse {MyS top($)}}
78
79  end
```

```
===== ERRORS FOUND =====
ERROR: Your code raise an "emptyQueueException()" in the code:" case @ q
of
nil then raise emptyQueueException ( ) end else local Temp = { Reverse @
q } in
```

```
q:= { Reverse Temp . 2 } Temp . 1 end end"
```

Limitations

Here are different examples where CorrectOz shows its limitations. Some errors cannot unfortunately be detected as we would like.

Limitation - 1

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N == 0 then Acc
         else {Sum N-1 Acc + (N*2)}
         end
      end
    {Sum N 0}
   end

   R = {MainSum 8}

end
```

Here an example of a submission where the code is correct but the result is not the expected one. Since the error is specific to the exercise, we cannot provide a relevant feedback, we should have implemented a tool for each exercise to detect those errors, but it is not the purpose of CorrectOz.

Limitation - 2

```
fun {Infix Tree}
local F A in
    fun{F A}
        if(Tree.left==leaf)
        then nil
        else
            {Append Tree.left A}
            {Indix Tree.left A}
        end
        if(Tree.right==leaf)
        then nil
        else
            {Append A Tree.right}
            {Indix Tree.right A}
            if(Tree.right==leaf)
                if(Tree.right==leaf)
                then nil
                end
            end
        end
```

```
21      end
22      {F Tree.1}
23  end
```

When the code is too messy and full of syntax errors, CorrectOz does not succeed to parse and to identify the origin of the errors.

Limitation - 3

```
1  local X SlowAdd in
2
3     fun {SlowAdd X Y}
4           {Delay 1000}
5           X+Y
6     end
7
8      X = SlowAdd{3000 + 3}
9      Browse(X)
10
11 end
```

Some errors of understanding are not identified by CorrectOz, the students are really creative and it is almost impossible to cover all the possible errors.

Limitation - 4

```
1  local X SlowAdd in
2
3     fun {SlowAdd X Y}
4           {Delay 1000}
5           X+Y
6     end
7
8      local X in
9      X = {SlowAdd 1000 1}
10     {Browse X} + {Browse X} + {Browse X}
11 end
12
13 end
```

The tool does not understand what the student is trying to do and does not succeed to parse the code.

Limitation - 5

```
1  fun {Prime}
2     local Prime1 in
3         fun{Prime1 N Acc}
4               if N==1 then false
5              elseif Acc==N then false
6              elseif length(N div Acc)==length(N) then false
```

```
7             else {Prime1 N (Acc+1)}
8               end
9         end
10    end
11 end
```

The auxiliary function is only called inside itself and is therefore useless. CorrectOz does not take into account this kind of error.

Limitation - 6

```
1 local X in
2     X == 2 ;
3     fun {Prime1 N X }
4         if X == N then true ;
5         elseif N mod X ==0 then false
6             else {Prime1 N X+1 }
7         end
8     end
9 end
```

In some case, when the student (used to java) ends the line with a ";", CorrectOz does not succeed to parse the code. Note that it depends where the semi colon is put.

Limitation - 7

```
1 fun {PrimePart Acc}
2     if N % 2 == 0 then false
3     else if Acc > N / 2 then true
4         else PrimePart Acc + 2
5         end
6     end
7 end
```

Here the student used the modulo operator "%" of other language than Oz. Unfortunately, "%" is used in OZ for comments, therefore CorrectOz does not parse this code correctly and does not report the error.

Limitation - 8

```
1 locale Pr in
2    fun {Pr N A}
3        if A == 1 then true
4        else
5          if N mod A == 0 then false
6          else
7             {Pr N A-1}
8          end
9        end
10    end
11    {Pr N N div 2}
```

```
12 end
```

We tried to include some typo faults in our interpreter, but the students always find a way to surprise us. In this case, the locale is not reported by CorrectOz.

Limitation - 9

```
1  local Fib R in
2     fun {Fib N}
3         local FibAux in
4             fun {FibAux N Acc1 Acc2}
5              if N==0 then Acc1+Acc2
6             else FibAux N-1 Acc1+Acc2 Acc1 end
7                end
8             end
9             {FibAux N 0 1}
10       end
11    end
12    R = {Fib 9}
13   {Browse R}
14 end
```

CorrectOz does not parse the code correctly when the student forgets to use the bracket to perform a procedure call.

Limitation - 10

```
1  local Fib R in
2     fun {Fib N}
3         local FibAux in
4             fun {FibAux N Acc1 Acc2}
5          if N == 1 then Acc1 end
6          if N == 1 then Acc2
7          else  {FibAux N-1 Acc1 Acc1+Acc2}
8          end
9             end
10            {FibAux N 0 1}
11       end
12    end
13    R = {Fib 9}
14   {Browse R}
15 end
```

In this case the student use the same condition for two consecutive if statement. CorrectOz does not detect that error.

Limitation - 11

```
1
2  local Fib R in
3     fun {Fib N}
```

```
4           local FibAux in
5               fun {FibAux N Acc1 Acc2}
6               if N==0 then Acc2
7               else
8                   {FibAux N-1 Acc2 Acc2*Acc1}
9               end
10           end
11              {FibAux N 0 1}
12       end
13     end
14     R = {Fib 9}
15    {Browse R}
16 end
```

The student multiplies his accumulator by 0 at the first iteration, the accumulator will therefore always be equal to zero. it is difficult for us to detect those errors since we cannot know if it is the desired behaviour of the program or not.

Appendix C

Statistics

Exercise	**# Students**	**# Submissions**	**Submissions/Students**
FromImplicitToExplicit	515	2,067	4
Shuffle	438	1,585	3.61
Calculator	440	2,095	4.76
Collections	446	2,510	5.62
Expressions	420	1,843	4.38
Palindrome	371	1,428	3.84
FromJavaToOz	399	2,094	5.24
Exceptions	397	1,654	4.16
ProducerConsumerFilter	419	2,259	5.39
TrackingInfo	376	1,824	4.85
ForCollectA	433	1,755	4.05
IDontWantPrimeNumbers	399	1,547	3.87
NFullAdder	359	1,939	5.4
InformationExchanges	178	628	3.52
TreesAsObjects	367	2,082	5.67
WeNeedExceptions	367	1,696	4.62
SharedStream	393	700	1.78
ObjectsConsumer	384	853	2.2

Table C.1: Statistical table concerning Louv1.2x exercises

Exercise	**# Students**	**# Submissions**	**Average submissions/student**
HelloWorld	1,292	2,443	1.89
BrowseX	1,270	2,152	1.69
Scope	1,198	2,821	2.34
CalledOnlyOnce	1,162	4,179	3.59
Sum	1,002	4,895	5.88
Mirror	949	3,721	3.92
Prime	883	5,525	5.8
Fib	831	3,055	3.67
Append	823	4,688	5.69
Fact	743	3,770	5.07
FindString	668	3,393	5.07
Flatten	344	1,273	3.7
FunAsInput	624	2,350	3.76
BuildMyFunction	587	2,558	4.35
BuildMyRecord	554	2,532	4.57
Infix	562	2,344	4.17
SortWithTree	493	2,448	4.96
IsBalanced	513	2,630	5.12
Complexity	463	2,033	4.39
Midterm	518	1,574	3.03

Table C.2: Statistical table concerning Louv1.1x exercises

Appendix D

Louv1.1x - Paradigms of Computer Programmings: Fundamentals

1 Introduction

1.1 Why and how should we learn about paradigms ?

A PROGRAM is basically a set of instructions solving a specific problem. These instructions can be written in many different LANGUAGES with the help of many different PARADIGMS.

Programming paradigm An approach to programming a computer is based on a coherent set of principles or a mathematical theory. A programming paradigm corresponds to a "*way of thinking*" in order to solve some kinds of problems.

The key idea to visualize here is the 3-level hierarchy programmers are facing everyday:

- Programmers can learn hundreds of LANGUAGES to write diverse programmes.
- Even if many languages **seem** completely different in terms of syntax, they often solve the same kind of problems. We can say these languages realize the same PARADIGM. That is why there are many fewer paradigms than languages.
- Finally, as said above, each paradigm is based on a set of CONCEPTS. In fact, there exists only a few concepts that can be combined to create diverse paradigms. So, we have once again many fewer concepts than paradigms.

As you can see, mastering many languages can be reduced to the understanding of a few paradigms - and even a fewer number of concepts behind these paradigms. Therefore, this course is based on one RESEARCH LANGUAGE called OZ which can express many paradigms at once. To combine many different paradigms, Oz is using a KERNEL LANGUAGE to gather the core concepts behind these many different paradigms. Indeed, each paradigm has its KERNEL LANGUAGE as its simple core language.

Kernel language A simple language containing the essential concepts - the primitive elements - of a paradigm.

Louv1.1x covers a first fundamental paradigm to understand a lot more afterwards (cfr. Louv1.2x). This paradigm is called FUNCTIONAL PROGRAMMING. The objective of the course is to build and understand its kernel language concept by concept.

1.2 Practical Organization

The course is divided into 6 lessons of one week. Each lesson is explained in one or more videos focusing on one subject at a time. Students can also purchase the course textbook called "*Concepts, Techniques, and Models of Computer Paradigms*, PETER VAN ROY, SEIF HARIDI, MIT Press".

Weekly exercises have to be performed by students on the INGINIOUS paltform in order to improve their practical understanding of the theoretical lessons. These weekly exercises are accounting for 50% of the final grade and students have an infinite number of tries per exercise. This means they can submit unlimited number of submissions on the INGINIOUS platform.

At the end of the course, a final exam is organized to evaluate the remaining 50% of the final grade. During this final exam, the students are limited to 2 submissions only on INGINIOUS. Once the course successfully completed, edX can deliver 2 kinds of certificate to prove the student's achievements:

Honor Code Certificate for free registrations.

Verified Certificate for registrations performed with a donation.

2 Basic programming concepts

2.1 A first paradigm

There are 2 fundamental ways of programming which are respectively called IMPERATIVE PROGRAMMING and DECLARATIVE PROGRAMMING.

Declarative programming *What* result a programmer wants in terms of properties and the computer figures out *how* to get there. Declarative programming is a good vision to follow in the future of computing. It is for example used in machine learning where computer learns how to solve some problems. Speaking about properties involve speaking in terms of mathematical functions and relations. Therefore, a declarative program that works for one day will work forever. Indeed, mathematical functions do not change. Finally, declarative programming is stateless.

Imperative programming It is about telling commands to be computed directly by the computer with a stateful programming in mind.

As stated previously, the objective of the course is to build a fundamental KERNEL LANGUAGE such that its core concepts are equivalent for a lot of others paradigms (and so, for many many more languages). This kernel language realizes a paradigm called FUNCTIONAL PROGRAMMING.

Functional programming A form of DECLARATIVE PROGRAMMING which is one of the simplest paradigm and the foundation of many many other ones.

2.2 Single and local assignment

Let's first introduce how to DECLARE, ASSIGN and print a variable - we say BROWSE in Oz. Four concepts are essential to understand how these ideas are handled inside the memory:

Identifier A character string starting with an uppercase letter which is bound to a variable in memory. This variable is accessible by refering to its given identifier until we redefine it.

Environment The binding, the link between a given identifier and a variable in memory. Each point in a program's execution has its own environment. We can consider this environment as a function that takes an identifier as input and returns a variable in memory as output.

Variable in memory A programmer does not see this variable but it exists *behind the curtain*. He can access a variable inside the current environment by using its identifier.

Single assignment A variable in memory can only be bound to one value.

Single assignment is a fundamental concept of the functional paradigm. Even if it could seem like a big handicap, it is actually *not* the case. In fact, single assignment allows programmers to reduce the risk of breaking a correct program. Indeed, it avoids programmers to mistakenly change the value of a variable.

Even if variables can only be bound to one value, identifiers can refer to multiple values at different parts of the execution. Indeed, at two points in time, the same identifier could refer to different variables in memory. We say they belong to different environments or in other words, they are bound LOCALLY to different variables in memory, each with its own value. The DECLARE instruction allows programmers to instantiate different environments. Indeed, each DECLARE corresponds to one environment at a time. Another way of instantiating an environment is to use a LOCAL instruction in which the environment starts right after the **in** keyword and ends just before the **end** keyword.

A significant concept underlying environments is the SCOPE of an identifier occurence. It is the fifth concept of functional programming.

Scope the part of the program code in which that same identifier refers to the same variable in memory.

If you ever wrote any code in Oz, you may have noticed that variables are not staticly typed. Indeed, Oz supports DYNAMIC TYPING which means that the variable type is only defined at assignment and not at declaration. Speaking of types, Oz introduces 2 kinds of numbers: exact numbers called INTEGERS and approximate numbers also called FLOATING POINTS. Both sets of numbers are completely independent and no automatic conversion can be achieved between them.

According to its syntax, Oz is definitely not a C-like language. Its syntax is more inspired by languages like Prolog, SmallTalk and so on. In fact, Oz syntax was carefully designed so that many paradigms do not interfere with each others. Peter Van Roy insists on 4 main differences in Oz syntax:

Identifiers always start with a uppercase letter.

Procedure and function calls are surrounded by braces: { }.

Local identifiers are introduced by: local <identifiers> in <statement> end.

Variables must respect the SINGLE-ASSIGNMENT RULE.

According to early studies, most programming bugs in Oz are introduced because of syntax errors. Therefore, it is important for students to understand and assimilate these 4 differences.

2.3 Functions

FUNCTION is a computing key structure. Indeed, programmers can use functions to avoid writing the same things over and over, maybe for different values. So, 4 last concepts can be introduced to complete the current functional paradigm:

Function It defines a program code to execute. It corresponds to another kind of value in memory. Indeed, the program code of a function is stored as a variable in memory. Therefore, each function has its own identifier.

Function composition Ability of a function to call another one such that it is possible to define a function in terms of other functions. Composition is a key ability for building large systems: each function is a layer built on top of another one.

Function recursion Ability of a function to call itself. Recursion is a key ability to build complex systems: it is possible to divide complex problems into smaller subproblems.

Conditional statement Ability to change the execution depending on the value of a calculation.

These final concepts complete the functional paradigm. Indeed, students can now calculate with numbers, define functions, apply function composition and recursion, and use conditional statement. **These concepts are the key basis for every programming language.** We can say the functional paradigm allows programmers to build a TURING COMPLETE MACHINE because these concepts have the ability to do anything other computer can do.

Starting with a fundamental paradigm like functional programming is a very efficient way to approach many more paradigms. Indeed, we just need to add one or a few concepts to enter a new paradigm. This is the key idea behind the course.

3 Recursion, loops and invariant programming

3.1 Introduction to invariant programming

If a programmer combines both a RECURSIVE call and a CONDITIONAL STATEMENT, his function will act like a LOOP. In this case, the function body corresponds to one iteration of the loop. And loops are a special case of recursion called TAIL RECURSION where the recursive call is the last operation in the function body.

Loop A calculation repeated until a condition is satisfied to achieve a result.

In order to program correct and efficient loops, programmers may need to learn a general technique called INVARIANT PROGRAMMING. Indeed, loops can be difficult to get exactly right and a set of good practices can be useful to achieve correct loops. Invariant programming is a very important technique because it applies to both the declarative and imperative paradigms! So, this single technique allows to understand many paradigms, which is exactly the course purpose.

In the next sections, will be developped new notions related to invariant programming: SPECIFICATIONS of a function, INVARIANT of a loop, ACCUMULATOR and the PRINCIPLE OF COMMUNICATING VASES.

3.2 Principle of communicating vases

The goal behind the so called PRINCIPLE OF COMMUNICATING VASES is to present better ways to implement recursive functions. The concept of INVARIANT is very effective to find these *better ways*. Because an invariant is a mathematical formula, a new function can be derived from it. This new function will correspond to a new way to solve the initial problem.

Invariant A formula that is true at every stage of the execution.

In order to find this invariant, programmers should learn how to apply the principle of communicating vases. The principle does speak for its name: one variable should

decrease, while another one should increase. Then, the invariants should be made of 2 variables at least. The first variable should decrease at each iteration of the loop. And in order to keep the formula true at each stage of execution, the second variable should increase. So, the idea of communicating vases is properly applied.

The second variable which is increasing at each iteration is called an ACCUMULATOR. Indeed, the new function - the new and better way to solve a problem - consists of an extra variable in which the result is computed and improved after each iteration.

Accumulator An extra argument in which the result is calculated piece by piece - we say it is accumulated.

In conclusion, invariant programming relies on the PRINCIPLE OF COMMUNICATING VASES to develop better INVARIANTS. And this principle relies itself on the notion of ACCUMULATOR.

3.3 Tail recursion

Invariant programming introduces the concept of loops. Loops can be programmed with TAIL RECURSIVE functions. In fact, it is proved that tail recursion is the most efficient technique to develop loops.

Tail recursive function When the recursive call is the last operation in the function body.

- Let's assume 2 functions that perform the same recursive computation.
- The first function is **not tail recursive**. It requires to store the *incomplete* results on a stack. Indeed, each call makes a part of the computation and these temporary results must be stored in memory. Once all the calculations are complete, the Oz interpretor must come back to these stored results to complete the final calculation. **A loop without tail recursion keeps the stack growing while executing.**
- The second function is **tail recursive**. In this case, the recursive call does not need to come back to previous results. Indeed, each computation is done before the recursive call - remember: the recursive call is the last operation - and the intermediate result is simply added to this recursive call as an argument. In conclusion, **a tail recursive loop has a constant sized stack and is more efficient because of its constant memory usage.**

3.4 Conclusion on invariant programming

- A recursive function is equivalent to a loop if it is TAIL RECURSIVE.
- To write tail recursive functions (loops), programmers should use an ACCUMULATOR.

- The accumulator can be found starting from an INVARIANT in respect to the PRINCIPLE OF COMMUNICATING VASES.
- This general technique is called INVARIANT PROGRAMMING and it is the only reasonable way to program loops.
- Invariant programming is useful in all programming paradigms. Indeed, other paradigm are build on top of this one.

4 Lists and pattern matching

4.1 Lists

Handling integers only may seem limited for programming more complex calculation. Therefore, programmers introduced a first compound data type called a LIST while developing LISP.

List An ordered sequence of elements. These elements can be of any type : an integer, an atom, or even another list. Indeed, lists can be defined in terms of themselves. Therefore, lists are defined recursively. Finally, lists are also values in memory just like integers or functions.

Syntactically and according to the EBNF rules, lists support the following formal definition:

- <List T> ::= nil | T '|' <List T>, where:
- A list is either empty (nil) ...
- ... or is a pair of one element (T) followed ('|') by another list (<List T>)
- and <List T> represents the set of all syntactic representations of a list of elements of type T.

Oz support many representations for lists according to the principle of SYNTACTIC SUGAR.

Syntactic sugar Ability to define different textual representations of the same concept.

- Bracket notation: [1 2 3]
- EBNF notation: '|'(1 ('|'(2 ('|'(3 nil)))))
- Another equivalent notation: a | b | c | nil

4.2 Pattern matching

The most efficient way to program with lists is to apply the previous concept of TAIL RECURSION. The main idea when handling a list is to divide it in terms of its own structure. A programmer can divide a list into 2 parts: the HEAD which corresponds to the first element and the TAIL which corresponds to the list without its first element. Therefore, programmers can perform some computations with the head of the list, and then make a recursive call on the tail such that, at each call, the next element of the list is being handled.

Therefore, programmers developed a general technique to divide a variable in terms of its own structure. This technique is called PATTERN MATCHING. The idea is, starting from a list, to give a pattern the list should match. This pattern is more a *shape* the list should follow.

- **H|T** where H is the head of the list and T its tail. For example, the list is an integer H followed by another list T.
- **H1|H2|T** where H1 is the head of the initial list, H2 the head of the list (H2|T) and T its tail. For example, the list is an atom H1, followed by another atom H2, itself followed by an empty list T (nil).

Pattern engineering The ability to *play* around with different patterns. But pattern engineering can cause DEAD CODE: patterns that can never be reached. Students should stay aware of this issue.

4.3 Lists functions are tail recursive

Remember: the kernel language is the foundation of every programming paradigm because it contains the core concepts of computing. Therefore, programmers can translate their list functions into this kernel language. This translation gives them the ability to see all the intermediate results computed during the computation. Then, they can prove that lists functions are in fact tail recursive. Indeed, it can be proved that the output list is created **before** the recursive call. This call remaining the last operation of the function body proves that the list functions are TAIL RECURSIVE by definition.

In conclusion, **programmers should use tail recursion and pattern matching while handling lists** because they are very efficient techniques.

Why are lists functions tail recursive

Let's prove how the implementation of the Append function in Oz can be translated into kernel language:

- First, the function must become a procedure with an extra argument. Indeed, functions do not exist in kernel language.

- The extra argument will contain the result that should be returned by the procedure.
- Every intermediate result should become visible.

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

```
declare
proc{Append L1 L2 L3}
  case L1 of nil then L3=L2
  else
    case L1 of H|T then
 local T3 in
   L3=H|T3
   {Append T L2 T3}
 end
    end
  end
end
```

Indeed, the translation into kernel language proves that the output list L3 is generated before the recursive call to Append.

5 Higher-order programming and records

In this section are introduced two final concepts to complete the functional paradigm and its kernel language. These concepts are called HIGHER-ORDER PROGRAMMING and RECORDS.

Higher-order programming The ability to use functions and procedures as first-class entities in the language. For example, as inputs or outputs of other functions. This is a key ability to introduce DATA ABSTRACTION.

Record A data structure, a general compound data type that allows symbolic indexing - with the help of symbolic constants called atoms. Record is a useful concept for both symbolic programming and DATA ABSTRACTION.

5.1 Contextual environment and procedure value

Higher-order programming is based on 2 main concepts called CONTEXTUAL ENVIRONMENT and PROCEDURE VALUE.

Contextual environment A little data structure in memory related to a function (resp. procedure) that stores all the free identifiers used inside this function (resp. procedure).

Free identifiers An identifier used *inside* the function but declared *outside* the function.

We can easily see the need of having a contextual environment. Assume an identifier is declared twice in a program with different scope and a procedure makes use of this identifier in its body. Therefore, the identifier is said to be "free" for the procedure and the contextual environment allows to know which definition can be retrieved for the given identifier.

Procedure value Each procedure is a value in memory, just like integers.

When a programmer declares a procedure called "Append" in his program, the whole procedure (both its signature and body) is in fact bound to a variable called "Append" in memory.

5.2 Higher-order programming

Higher-order programming is a key concept in this course. Indeed, it is an important concept for building large systems based on data abstraction in layers or to achieve encapsulation. These are ideas that programmers hear all the time when programming. But why is higher-order programming so important ?

First, because **procedures (and functions) are values**, we can pass them as inputs to other functions and return them as outputs, just like integers! This introduces a concept called the ORDER of a function:

- A function whose inputs and outputs are not functions is said to be FIRST ORDER.
- A function is said to be ORDER N+1 if its inputs and outputs contain a function of maximum order N.
- For example, a function with a first order function as input is said to be a second order function.

Based on this definition of order, we can define a few more properties:

Genericity When a function is passed as an input.

Instantiation When a function is returned as an ouptut.

Function composition When a function takes 2 functions as input and return their composition.

Encapsulation The ability to hide a value inside a function. The most trivial example is to define a function called Zero which only returns the integer 0.

5.3 Records

Record is the last concept to study for completing the functional paradigm and its full kernel language. It is a key concept to make things such as data abstraction much more readable.

Atom A symbolic value - or constant - corresponding to a sequence of lowercase letters and digits that starts with a letter. It is also any sequence of any characters delimited by single quotes.

Record A record has a label - its name - and groups a set of values into a single compound value. Each record has a fixed number of values and each value can be accessed directly through the dot operation. Indeed, each value is stored as a pair of field name and field value separated by a colon.

Label An atom.

Field names Atoms or integers.

Field values Can be any value.

Width The width of a record is the number of values that it groups.

Arity The width of a record is a list of its field names.

In fact, it is *the* general compound data type used to build many other compound data types. Indeed, **it is the only compound data type in the kernel language**! It keeps the kernel language simple. Let's prove that the previous compound data types can build on top of our record definition:

Atom A record whose width is 0, a record with a label but without any grouped value.

Tuple A record whose field names are successive integers starting with 1. Note that fields declared without field names are automatically bound to successive integers starting with 1. it is another form of syntactic sugar.

List A recursive data type built on 2 kinds of records: nil and H|T. In fact, the following syntactic sugar is applied: $'|'(1{:}H\ 2{:}T)$.

6 Trees and computational complexities

6.1 Trees

In this section are introduced TREES. Next to lists, trees are the most important data structure in computing. Trees are usually used to make calculations while maintaining a certain property. Indeed, many tree data structures are based on a global property, a property of the whole tree that must be maintained during the calculations. Therefore, trees are a good example of goal-oriented programming which is a generalization of invariant programming.

Tree A recursive data structure: it is either an empty tree called a **leaf** or an element and a set of trees called **subtrees**.

- Lists and trees are similar in terms of recursivity.
- However, a list is a linear structure (there is only one single sublist) while a tree is a branching structure with several potential subtrees.

6.2 Ordered binary trees

A first kind of interesting special tree is the ORDERED BINARY TREE which is based on both properties called ORDERED and BINARY as its name states.

Binary Each non-leaf tree has 2 subtrees, respectively named left and right.

Ordered For each tree, including all subtrees: all the keys in the left subtree are smaller than the key of the root, and the key of the root is smaller than all the keys in the right subtree.

Ordered binary tree A tree that has a key and a value field at each node - called information fields - and maintains both ordered and binary property.

6.3 Search trees

A second kind of interesting tree is called a SEARCH TREE. Furthermore, experiences shows that these trees are usually BALANCED.

Search tree A tree that is used to organize information and with which we can perform various operations such as looking up, inserting and deleting information.

Balanced tree A binary tree with both subtrees that have the same amount of leaves (with a difference of maximum 1) and these subtrees balanced themselves.

6.4 Goal-oriented programming

As seen in the introduction section, many tree algorithms depend on global properties and most of the work they do is to maintain these properties. For example, ordered binary tree must satisfy the global ordering condition. Despite the ease of inserting a new element in these trees, it is way harder to delete information from them.

Goal-oriented programming is widely used in AI algorithms. But these algorithms can give unexpected results because they try to maintain the global properties. In fact, goal-oriented programming is characteristic of living organisms. Indeed, thanks to this unpredictability, it is an efficient way to give a spark of life to some algorithms!

6.5 Introduction to computational complexities

An interesting question for a programmer is: "how performant is my algorithm ?". To answer this question, programmers can try to analyse the execution time or the memory usage of their implementation. For example, it is easy to measure the number of seconds needed to execute an algorithm for a given input on a given platform. But this measure is not very useful because it is bound to a specific processor.

It is more interesting to see how the execution time depends on the input size in order to be able to predict this time on different plaftorms. So, programmers should look for a *function* and not a number. This function would allow them to predict the execution time despite the hardware composition. Even more interesting is to perform an ASYMPTOTIC ANALYSIS.

Asymptotic analysis The goal is to find a function that gives the execution time given an input size whose size *increases without bound*.

Computational complexity The use of asymptotic analysis to study the execution time and memory usage of programs.

By observing the computational complexities of some functions, programmers can highlight 2 common properties about asymptotic analysis:

- **Fast-growing functions are bad:**
  Once powers are included in a function, these functions quickly tend to take years to be computed for large input size. Moreover, they do not allow to solve big problems in short time.
- **Constant factors can be ignored:**
  In fact, 2 kinds of functions are part of the game: *polynomial* and *exponential* ones. The exponential functions are very bad because their performances do not improve very well in comparison to the hardware enhancements. So, in order to avoid the hardware influences in our functions, we should ignore constant factors.

6.6 Best case, average case & worst case

Let's assume a programmer does not want to perform a temporal analysis based on integers only. Indeed, a programmer could have coded a function to store a list for example. Basically, this function would work as follow:

- **If the list is already sorted:** return the list right away.
- **If the list needs to be sorted:** do some computations to sort the list, then return it.

So, the temporal complexity might be different for different input lists ! Therefore, there is a need to compute some kind of average. There are 3 standards ways to do this:

Best case Takes only inputs of size n with smallest time.

Worst case Takes only inputs of size n with largest time.

Average case Takes inputs of size n according to some probability distribution (which must be given).

Therefore, programmers can highlight 3 scenarios when proceeding to a temporal analysis: a best case (e.g. when the list is empty), a worst case (e.g. when the list is not ordered at all) and an average case. These scenarios correpond to computing a lower bound, an upper bound, or both lower and upper bound. Then, we can introduce 3 new mathematical notations:

Big O(n) For upper bound, or worst case.

Big Omega(n) For lower bound, or best case.

Big Theta(n) For both lower and upper bound, or average case

Asymptotic equivalence When a function has the same lower and upper bounds.

6.7 Spatial complexity

Temporal complexity - which is about measuring execution time - is not the only way to scale the performance of a program. In fact, SPATIAL COMPLEXITY is also important and can be measured by 2 means:

Spatial complexity Memory usage measurement for a given input size.

- **Active memory active(n,t):**
  A first way to measure spatial complexity is to count the total number of words in use by the program at time t, for a given input size n. It can be useful to know the memory usage at a particular time. Indeed, different functions do not need the same amount of memory during the program execution, and they need memory capacity at this exact time, and not later.

- **Memory consumption consume(n,t):**
  A second way to measure spatial complexity is to count the number of words allocated per second at time t, for a given input size n. It can be useful to see how much temporary data structures are used by a program during its execution.
  As an analogy, active memory corresponds to a human weight (in kg) whereas memory consumption corresponds to how much this human eats (in kg/day).

6.8 Intractable problems and the class NP

In the previous sections were developped some ways to apprehend performance. Let's think more about it: how can we improve performance ?

- **New processor hardware** in respect to Moore's law: the density of integrated circuits doubles around every 2 years.
- **Program optimization** by developing faster but more complex algorithms even-though optimization has its own limitations.
- **Intractable problems and P=NP question:** some problems seem to be inherently time consuming.

Basically, some problems seem to take a lot of time to be solved by a computer because there is no easy way to get around it. Therefore, the temporal complexity of the algorithms solving these problems are exponential. This introduces the class of NP PROBLEMS for NONDETERMINISTIC POLYNOMIAL TIME. Indeed, it is easy to *verify* a solution in a polynomial time, but it is way harder to *find* a solution from scratch !

The fundamental question then becomes: "Is P=NP ?", or in other words : "Is finding a solution as easy as verifying a solution ?". Note that some NP problems have the property that if an efficient algorithm can be found for them, then it is possible to derive an efficient algorithm for all problems in NP. These *hardest problems* are said to be NP-COMPLETE. They are often encountered in practice.

6.9 Conclusion on performance

Both execution time and memory usage are very important properties of a program to measure its performance. Asymptotic analysis should be performed to evaluate these measurements. Moreover, practical performances are influenced by 3 aspects : hardware performances, program optimization and fundamental properties of the problem (NP).

7 Correctness and semantics

7.1 Introduction to semantics

In order to write correct programmes and to understand other people's programmes, a good programmer should have a deep understanding about how a language works. The goal of this section is to learn language SEMANTICS. There are many things a programmer can do by using semantics: making sure a program is well-designed, explaining the program of other ones, understands how a program manages memory, or even ensures that a program is correct. In this section, it is actually the verification, the correctness of a program that will be interesting. A programmer can build his thinking on 3 pillars to verify the correctness of a program:

1. **Specification** Describe what we want, a mathematical formula for example.
2. **Program** What we have written in a specific programming language.
3. **Semantics** Connect the specification and the program to prove that the code executes according to what we want. Then, we can prove our program is satisfying our specification.

   Semantics of a programming language Or formal semantic, or mathematical semantic, is a completely precise explanation of how programs execute. It can be used to reason about program design and correctness.

Both programmation and specification have been studied previously. Therefore, this section approaches semantics only. There are 4 fundamental ways to approach semantics:

1. **Operational semantics:** Explains a program in terms of its *execution*, its operations on a rigorously defined abstract machine.
2. **Axiomatic semantics:** Explains a program in terms of properties, as an *implication*: if certain properties called preconditions hold before the execution - they are axiomatic -, then some other properties called postconditions will hold after the execution.
3. **Denotational semantics:** Explains a program as a *function* over an abstract domain, which simplifies certain kinds of mathematical analysis of the program.
4. **Logical semantics:** Explains a program as a *logical model* of a set of logical axioms, so program execution is actually deduction: the result of a program is a true property derived from the given axioms.

This section develops the OPERATIONAL SEMANTICS because it has the advantage to work for all paradigms. Indeed, programs have to run one day. This is the most general technique to approach semantics. Operational semantics can be divided in two parts: the KERNEL LANGUAGE - in which the program can be translated - and the ABSTRACT MACHINE - to execute the translated program -. The first part was already studied in the previous sections. Then, this section will mainly focus on the abstract machine operation.

7.2 Abstract machine

The abstract machine brings several new concepts to our study:

Single-assignment memory Set of variables and the values they are bound to.

Environment Set of link between identifiers and variables in memory.

Semantics instruction A pair of instruction and its environment.

Semantic stack A stack of semantic instructions to be executed.

Execution state A pair of semantic stack and the single-assignment memory.

Execution A sequence of execution states.

In practice, both the initial memory and the initial environment start empty. The semantic stack is filled with the program to be executed. While the semantic stack is not empty, the abstract machine will pop the instruction at the top of it and execute it according to its semantics rule. Both the memory and the environment will be modified if necessary. That is how the abstract machine does work in practice.

7.3 Semantics rule

A programmer must first study the SEMANTICS RULE bound to every kernel language instruction in order to know how to execute a piece of code in the abstract machine.

Semantics rule Describes how a kernel language instruction should be executed in the abstract machine. e.g.: how an instruction modifies the memory, when to push/pop an environment,...

Variable creation How a variable is created in a local instruction.

Sequential composition How sequential instructions are split into the semantic stack to be executed one after the other.

Conditional How a if ... else is managed in the abstract machine.

Assignment How a variable is bound to a value in memory.

Procedure definition/call How a procedure is defined and called in terms of memory.

8 Conclusion on Louv1.1x

The course was organized in 2 paths. The first one was about operations that developpers can do while programming such as invariant programming, higher-order programming and a full kernel language overview. The second path was about data structure on which we can do these operations such as lists, trees and records. Finally, both paths met up to introduce a brief overview of semantics and performance analysis.

Appendix E

Louv1.2x - Paradigms of Computer Programmings: Abstraction and Concurrency

1 Introduction

1.1 Motivations and roadmap

Programmers need to introduce new essential concepts for building large problems that are part of the real world. Indeed, the real world is *complex* and *changing*, but also made of *independent activities*. Based on the FUNCTIONAL PROGRAMMING and its full kernel language studied during Louv1.1x, this course develops 4 new programming paradigms:

1. **Object-Oriented Programming** based on STATE (CELLS), DATA ABSTRACTION, POLYMORPHISM and INHERITANCE. Louv1.2x also introduces Java in order to better understand OOP. However, this master thesis only focuses on Oz and the Java part is willingly skipped.

2. **Deterministic Dataflow** based on CONCURRENCY (THREADS), STREAMS, AGENTS and the lack of RACE CONDITIONS.

3. **Multi-agent Dataflow** based on DETERMINISTIC DATAFLOW with PORTS and NON-DETERMINISM where it is truly needed.

4. **Active objects** combining both OBJECT-ORIENTED PROGRAMMING and MULTI-AGENT DATAFLOW.

To understand and manipulate these new paradigms, programmers should study a set of new concepts. Louv1.2x is about studying these new principles.

1.2 Practical organization

Louv1.2x is organized in 7 consecutive lessons. Each lesson is divided in one or more video produced by the professor PETER VAN ROY. However, the 6th lesson is taught by the professor SEIF HARIDI that developped the Oz language along with PETER VAN ROY.

The lessons are interleaved with small exercises. Some exercises are graded by the INGINIOUS on-line grader. These exercises are accounting for 50% of the final grade. The remaining 50% must be completed thanks to the final exam. During the final exam, students have *only one* try per exercise on INGINIOUS.

The evaluation and certification commodities are similar to Louv1.1x.

2 State

2.1 Implicit state

In the FUNCTIONAL PARADIGM, there is no notion of *time*. Indeed, functions are defined in terms of mathematics and they cannot change once they are defined. Furthermore, a function cannot observe the execution of another one, but only its final result.

However, in the real world, there is both *time* and *change*. Living organisms do grow and learn. A first solution to model both time and change is to understand ASBTRACT TIME. This notion can result in the concept of IMPLICIT STATE. We say these states are *implicit* because the language does not need to be changed. Indeed, the notion of change is purely observed in/by the programmer's head.

Abstract time A simplified version of time that keeps the essential property that we need: modeling change.

State A sequence of values calculated progressively, which contains the intermediate results of a computation.

Implicit Because the language did not change, we are still in the functional paradigm, it is only our programmer's mind that started to observe the execution of the function step by step.

In fact, programmers would like the program *itself* to observe changes, and not only the programmers themselves. Therefore, they should leave the functional paradigm by adding some new concepts, new ways of thinking and programming.

2.2 Explicit state

The objective for programmers is to extend the kernel language to make state *explicit*. Explicit, because we will modify the current language to add the concept of state to it.

In practice, explicit state is represented by the concept of CELL by programmers. Cells do respect the token equality in opposition to the concept of structure equality (for lists, records,...).

Cell A cell is a box with an identity (a constant, its name or address) and a content. This content can be modified: we can observe explicit change.

Structure equality 2 structures with the same values created separately are equal.

Token equality 2 cells are equal if they are the same cell, therefore 2 cells created separately are always different.

2.3 Semantics of state

Even though the kernel language was extended to add the concept of cell, the abstract machine is still not aware of these changes. Therefore, it should be extended too. In order to achieve this, 2 stores must be added to the abstract machine. Students are already comfortable with the first one which corresponds to the concept of SINGLE-ASSIGNMENT. The second store contains cells which are related to MULTIPLE-ASSIGNMENT.

Single-assignment store Immutable store that contains variables.

Multiple-assignment store Mutable store that contains cells.

Cells in the store A pair of 2 variables. The pair itself (the link between both variables) is stored in the multiple-assignment store. The variables are stored independently in the single-assignment store. The first variable cannot be changed, it represents the name of the cell, a constant denoted by a Greek letter in the store. The second variable corresponds to the content of the cell and can be swapped with another one in order to modify the cell's content. Indeed, changing the cell's content means changing the second variable of the pair.

One of the key ideas behind Louv1.1x is that the functional paradigm can be easily extended to approach many paradigms. By adding the concept of CELL and MULTIPLE-ASSIGNMENT, programmers managed to get out of this basic paradigm. The resulting paradigm is called IMPERATIVE PARADIGM.

Imperative paradigm Combination of functional programming and the concept of cell such that programmers can explicitly add growth and change in their programs.

2.4 State and modularity

The goal of Louv1.2x is to study the tools for building large and complex software systems. EXPLICIT STATE (CELLS) is a key tool to enable MODULARITY which is an important property when building large and complex programs.

Modular A program is *modular* with respect to a given part if this part can be changed without changing the rest of the program.

But modularity is not always easy to deploy efficiently. Therefore, choosing between functional and imperative paradigm should be thought carefully.

- **Functional paradigm**:
  1. A component never changes its behaviour. If it works once, it works for all the time.
  2. Updating a component often means changing its interface. Therefore, many other components using it might become wrong.
- **Imperative paradigm**:
  1. Updating a component does not require to modify its interface thanks to explicit state.
  2. However, the behaviour of the component might change and break some other component using it.

3 Data abstraction

DATA ABSTRACTION is the main organizing principle for building large and complex software systems such that many parts can be organized in layers or in compartments and do not interfere with each others. It allows to program the *complexity* of the real world. The main idea behind DATA ABSTRACTION is called ENCAPSULATION.

Data abstraction A part of a program that has an inside, an outside and an interface in between.

Inside Is hidden from the outside such that all the operations on the inside must pass through the interface.

Interface A set of operations that can be used according to certain rules.

Encapsulation Fact of enforcing the separation between inside and outside.

Encapsulation brings many assets to programmers:

- It *guarantees* that programmers can interact with the inside only through well-defined operations - remember the notion of INTERFACE.
- It *reduces the complexity*. Indeed, programmers do not have to know the internal implementation but only the interface in order to make use of a component.

- It *enables programming with large teams* in which each developer is responsible for implementing, maintaining and guaranteeing the internal behaviour of its own components. The other programmers only interact with the interface of these components and do not modify anything to their internal implementation.

There exist 4 kinds of data abstraction in practice:

1. **Object** Represents both a value and a set of operations grouped together in a single entity.
2. **Functional object** Immutable object such that invoking an object returns another object with the new value.
3. **Abstract data type** Consists of a set of values (constants) and a set of stateless operations.
4. **Stateful ADT** Uses both a cell and a secure wrapper. In practice in C or for static attributes in Java.

4 Object-oriented programming

4.1 Introduction to OOP

OBJECT-ORIENTED PROGRAMMING - also abbreviated OOP - is the second paradigm studied in Louv1.2x. It relies on 3 new principles. The first one is DATA ABSTRACTION as seen previously. The others one are POLYMORPHISM and INHERITANCE.

4.2 Polymorphism

POLYMORPHISM is an important principle in OOP. In fact, both objects and all kinds of data abstractions do support polymorphism in OOP. Polymorphism allows to comportamentalize responsibilities such that a design decision only imply a small part of the program.

Polymorphism An operation is polymorphic if it works correctly for arguments of different types.

Responsibility principle A responsibility should be concentrated in one part of the program such that a polymorphic message is understandable by many objects and each object knows how to react upon its own responsibility.

4.3 Inheritance

In practice, many data abstractions are very similar. For example, a multiset is a collection with no defined order, and a sequence is a multiset with a total order. Because

of these similarities, programmers are often tempted to duplicate code accross their programs. But INHERITANCE was invented to avoid these inconvenient duplications. It is an important idea to avoid redundancy by building data abstractions incrementally. For example, a sequence can *inherit* from the basic properties of a multiset, then adds the total order property to it. Therefore, inheritance is very useful to avoid code duplication and to improve code reusage.

However, inheritance should be used carefully and programmers should be aware of 2 general rules:

- **Alway prefer composition to inheritance**:
  By default, a programmer should not want his classes to be extensible by inheritance.
- **Always follow the substitution principle when using inheritance.**

Composition When an object refers to another object in one of its attributes such that the object referred is accessed through its usual interface.

Substitution principle When a class A extends a class B, A cannot modify the ground basis of B. We say that A is a *conservative extension* of B.

4.4 OOP objects

Despite some important principles like inheritance and polymorphism, the main feature about object-oriented programming... is OBJECT indeed. OOP objects bring some interesting properties for programmers.

Object A data abstraction encaspulating both values and operations. Values are called *attributes* and are hidden from the outside. Operations are called *methods* and are visible from the outside as an interface. Objects are based on a simple data abstraction extended with dispatching and instantiation.

Dispatching Instead of having access to multiple procedures directly, objects are reduced to a single entry point which receives a message such that, when we call the object, the entry point reads the message and chooses the right method to execute.

Instantiation Ability to create multiple instances of an object such that each object has the same methods, but has its own states (it has its own cells as values). Indeed, each object instance is independent of others.

4.5 Class

DATA ABSTRACTION, DISPATCHING and INSTANTIATION are important properties of OOP objects. But the most interesting one is surely the concept of CLASS. Classes are very useful data abstractions because they bring many improvements to objects:

- A class *guarantees the objects are constructed without error*. Indeed, their syntax is verified.
- A class *improves the readability* by using concepts at the level of objects. It does not use concepts proper to procedures of cells that are lower level concepts, more basic and generalistic.
- A class *improves overall performance* if the system can find a better way to implement the object.

Class A specialized syntax to define many objects with the same behaviour. These objects are said to be "*of the same class*". In memory, classes are values. They are stored as a record that groups both attributes and methods definition.

5 Deterministic dataflow programming

5.1 Concurrency

The world is naturally *concurrent* because it is made of independent activities. Many examples shows computing is also highly concurrent: distributed systems, operating systems, processes,...etc. However, the paradigms studied until now do not support CONCURRENCY. There is no way of *communication* from one concurrent activity to another and there is no way of *synchronization* between concurrent activities.

Concurrency A property of systems that are made of activities that progress independently. It allows to model independent activities in a program.

To support concurrency, a new paradigm should be introduced. Many paradigms do support it but concurrency is quite complex to program. Therefore, Louv1.2x only introduces one of them called DETERMINISTIC DATAFLOW. Indeed, this paradigm is quite close to the functional paradigm that was already studied in Louv1.1x - even though it is truly concurrent in opposition to functional programming.

Deterministic dataflow A simplified form of concurrency that always gives the same outputs for the same inputs. Activities are running independently and support unbound variables. Indeed, if they encounter an unbound variable, they enter in a waiting state such that another activity can bound this variable later which would make the waiting activity to resume its execution

5.2 Threads

Speaking of *independent activities*, let's define them and enumerate their properties. In computing, independent activities are also called THREADS. For example, each call to the Oz function *Browse* instantiates in fact a new thread in execution.

Activities (Thread) A sequence of executing instructions.

1. **Sequential**: each thread is executed sequentially as any other parts of a program.

2. **Independent of other threads**: there is no order of execution defined between different threads, they're all executed using an INTERLEAVING SEMANTICS such that one thread is in execution at a time and the system guarantees that each thread will have a fair-share of the computational capacity of the only processor.

3. **Shared variables** are used for communicating between threads.

Interleaving semantics Assuming all the threads share the same processor, the semantic rule can be formulated as follows: "each thread has its own semantic stack with its own set of instructions and shares the memory with every other existing stacks". Therefore, a semantic stack with the current environment should be instantiated every time the abstract machine does encounter a new thread.

5.3 Non-determinism

In a *sequential* program, execution states respect a TOTAL ORDER property. However, in a *concurrent* program, only the execution states of the same thread respect this total order property. The execution states of the complete program respect in fact a PARTIAL ORDER property such that many executions are compatible with this partial order and the SCHEDULER chooses one of these potential scenarios during the actual execution. In fact, there is some kind of NON-DETERMINISM in the deterministic data flow paradigm that was introduced. Let's define these new terms:

Total order When comparing any 2 execution states, one must always happen before the other.

Partial order When comparing any 2 execution states, there might be no order between them (either may happen first).

Determinism A deterministic program always gives the same output for the same input.

Non-determinism Ability of a system to make decisions that are visible when running a programme such that these decisions can vary from one execution to another.

Scheduler The part of the system that decides at each moment which thread to execute.

And NON-DETERMINISM brings of course its set of issues. For example, using both THREADS and CELLS at the same time in a program can lead to severe dysfunctions. Therefore, non-determinism must always be managed carefully in this case.

However, **deterministic dataflow supports only threads and not cells**. Therefore, threads can only synchronize themselves with single-assignment variables. So it is its main advantage: the non-determinism of the scheduler does not affect its final result ! In fact, one can say that deterministic dataflow has "NO OBSERVABLE NON-DETERMINISM"

because it has no race condition. But that does not mean the scheduler is deterministic ! Indeed, **it is non-deterministic**, but we can't observe it directly because the results are always the same, whatever non-deterministic choices are made by the scheduler.

5.4 Concurrency transparency

The main reason to learn about deterministic dataflow in order to introduce a concurrent paradigm is the fact that it is very close to the functional paradigm. Indeed, the CONCURRENCY TRANSPARENCY of the deterministic dataflow is a key property for programming concurrent programs that do work all the time.

Concurrency transparency Adding threads to a functional program does not change its final result.

In fact, including threads in a functional program makes it more incremental: it will calculate things piece by piece. But its final result will never change because of its inherent properties:

1. **A functional program**: By definition, always outputs the same result for the same inputs.
2. **No observable non-determinism**: Including threads but no cell only changes the order of execution, so when the result is calculated but not how.

6 Multi-agent dataflow programming

6.1 Streams

In order to achieve concurrency in deterministic dataflow, threads share the variables in memory. Therefore, they can communicate between them by using these shared variables. An efficient way of communication between 2 threads is to make use of a STREAM as a communication channel. Therefore, it is possible to implement a PRODUCER/CONSUMER programme in which the threads are communicating through a stream in a *pipeline* fashion.

Stream A list that ends with an unbound variable such that this unbound variable can be used to extend the list as long as necessary. If the unbound variable is bound to "nil", then the stream will be closed.

Producer A thread that generates a stream of data.

Consumer/Transformer A thread that reads this stream of data and performs some action with it (it can *transform* the stream for example).

Pipeline One kind of multi-agent program where each agent receives a stream, makes a part of the work and outputs a new stream for the next agent. The first agent is usally called PRODUCER, the intermediate ones are called TRANSFORMERS or CONSUMERS.

6.2 Agents

In the previous section, was introduced a first kind of multi-agent program called a PIPELINE. There are in fact 2 main multi-agent architectures encountered in practice:

Client-Server Where a server and many clients are running independently. The server provides some services - it receives messages and replies to them - and the clients know the server address such that they are able to ask for some of its services.

Peer-2-Peer A very similar architecture to client-server, except that every client is also a server - we say it is a peer. Both can communicate by receiving, sending and replying to messages.

In fact, clients, servers and peers are called AGENTS, or actors. Each of these agents has an identity, receives and processes messages and replies to them. These abilities have to be casted in our programming paradigm. Let's introduce/recall some concepts:

Identity or address, corresponds to the notion of PORT.

Messages are related to any DATA STRUCTURE. A message can be a record or even a procedure.

Receiving messages can be achieved by using a STREAM of messages such that new messages can be appended one by one at the end of the stream.

Replying to messages can be achieved by using a DATAFLOW VARIABLE in the received message such that the receiving agent can compute his answer and bind the unbound variable afterwards.

Programmers already know many data structures and how to use streams and dataflow variables. One remaining concept should be studied in order to make a correct implementation of a multi-agent program. This concept is called PORT.

6.3 Ports

In order to represent the identity of an agent, developpers made up a new notion called PORT which is in fact a "named streams". 2 kinds of agents can be built using ports.

Port A pair consisting of a name and the tail of a *message stream* such that we can send messages to the stream by using the port's name. The tail of the message stream is an unbound variable that can be bound to the next message.

Stateless agents out of ports Agents that will receive messages and will respond to these messages, but that do not have any state.

Agents with state Agents that maintain a state such that they have a state-transition function which models the evolution of the state: the current state and the received message both influence the next state.

6.4 Sending messages

There exist 3 properties about sending messages with ports:

Asynchronous sending The sender continues immediately after sending his message. Indeed, the sender does not know when the message will be processed by the receiver. Very useful when the sender does not need an answer to continue his execution.

Ordered per thread Multiple sends in a thread are executed in the sequential order in which they are defined.

Not ordered from multiple threads Multiple sends in different threads can be executed in different orders depending of the scheduler choices. **For the first time, non-determinism can be observed!**

6.5 Replying to messages

There are 2 common ways to reply to a given message when using ports:

Traditional view An agent includes his own port when sending a message such that the receiver can send back his answer to this port.

Asynchronous reply An agent includes a dataflow variable when sending a message such that the receiver can bind this variable with his answer. The sender will stop his execution when he encounters this unbound variable later in his program. Finally, he will resume his execution when the receiver will have bound the variable with a final answer.

6.6 Agent protocols

This subsection goal is to create higher-level abstractions for communication by observing the existing patterns of message exchange. These patterns are defining PROTOCOLS for exchanging messages. There are 2 main protocols that programmers can observe with multi-agent programming: BROADCAST PROTOCOL and CONTRACT NET.

Protocol Set of rules for sending and receiving messages.

1. **Broadcast protocol**: Rule is simple, each agent just send a message M to all the agents in the system. For this purpose, a BROADACASTER agent is implemented such that any agent that wants to broadcast a message sends a broadcast request to this broadcaster agent. This agent is also responsible for keeping up to date the list of known agents in the system.

2. **Contract net**: The goal of contract net is to choose one agent satisfying a certain criteria among a group of agents. Then, the idea is to query a set of providers such that all providers send their information back to the client and this client can choose the most suitable provider for his needs.

- **Coordinator**: The agent that is responsible for receiving requests from the users and broadcasting them in response to all slaves. This coordinator is then responsible to pick the most suitable slaves among all, inform them of his choice and finally complete the whole protocol by sending his choice to the client.
- **Slave**: An agent that receives a request message from the coordinator, sends his status to this coordinator and waits until a decision is made.

7 Active Objects

Louv1.2x introduces 4 new paradigms. In the previous section were already studied OOP, deterministic dataflow and multi-agent programming. Therefore, there is one remaining paradigm to study. This one is called ACTIVE OBJECTS PROGRAMMING. It is at the encounter of OOP and multi-agent programming. Indeed, active objects gathers both concepts of CLASSES and OBJECTS, and CONCURRENCY into the same paradigm. In fact, agents are not only defined in terms of PORTS that receive messages and STATE-TRANSITION FUNCTIONS that define their behaviour but also in terms of classes and objects.

In active objects programming, classes define the agent behaviour. The agent local state corresponds to the class local attributes, the possible messages correspond to the method signatures and the message handlers (the state-transition function) correspond to the method definitions. Therefore, each instantiated object of a class will have its own thread that reads a message from the stream and invokes the corresponding method to answer the received messages.

Appendix F

Templates INGInious

0.1 Mirror

In this exercise, the student is asked to implement a reverse function of an integer.

template.py

```
local MainMirror R in
   fun {MainMirror Int}
      local Mirror in
         fun {Mirror Int Acc}
@  @Mirror@@
            end
           {Mirror Int 0}
      end
   end

R = {MainMirror 10}
end
```

0.2 Prime

This exercise focuses on the use of accumulators. The recursion is therefore important.

template.py

```
local R Prime in
   fun {Prime N}
@  @Prime@@
   end
   R = {Prime 7}
end
```

0.3 Fibonacci

This exercise also focuses on the use of accumulators by making the student to use two accumulators.

template.py

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
@  @Fibonacci@@
            end
            {FibAux N 0 1}
      end
    end
    R = {Fib 9}
   {Browse R}
end
```

0.4 Flatten

The students are asked to implement their own flatten procedure. We therefore check that they do not call the official flatten in their code.

```
java -jar CorrectOz.jar input.oz -d false -nFun 2 -nCase 1 -nilCase true
-list true -tailRec true -extProc [Flatten]
```

template.py

```
local FlattenList S in
   fun {FlattenList L}
@  @Flatten@@
   end
   S= {FlattenList [ 1 [2 3] [4] [[5 6]]]}

end
```

0.5 FunAsInput

In this exercise, the student has to call a test procedure with a list of function as input. The input asked to the students is too small for us to make relevant checks.

template.py

```
local FunnyFunc Test in
   fun {FunnyFunc FunL L}
       case L of H|T then {FunL.1 H}|{FunnyFunc FunL.2 T}
       else nil
       end
   end

    proc {Test FunL L SolL}
    {Browse {FunnyFunc FunL L} == SolL}
   end

@  @FunAsInput@@
end
```

0.6 BuildMyRecord

The students are asked to complete the body of a function that builds records using the oz procedure Record.make.

template.py

```
local Transform R in
   fun{Transform L}
@  @BuildMyRecord@@
   end
R = {Transform [record [3 5 6] [b c d]]}
end
```

0.7 InfixTree

First exercise with records representing trees.

```
java -jar CorrectOz.jar input.oz -d false -nCase 1 -leafCase true
```

template.py

```
local Infix R Tree in
   fun {Infix Tree}
@  @InfixTree@@
   end

```

```
6     Tree = btree(1:42 left:btree(1:12 left:btree(1:9 left:leaf right:leaf)
7  right:btree(1:21 left:leaf right:btree(1:24 left:leaf right:btree(1:31
8  left:leaf right:leaf)))) right:leaf)
9
10    R = {Infix Tree}
11 end
```

0.8 SortWithTree

The student has to provide two functions with the signature and the entire body. One function has to transform a tree into a list and the other does the opposite.

```
java -jar CorrectOz.jar input.oz -d false -nIf 1 -nCase 3 -nilCase true
-leafCase true -list true
```

template.py

```
1  local FromListToTree R T FromTreeToList S in
2
3  @  @SortWithTree@@
4     R = {FromListToTree [42 21 12 34 9]}
5     T = btree(1:42 left:btree(1:9 left:leaf right:leaf) right:leaf)
6     S = {FromTreeToList T}
7  end
```

0.9 IsBalanced

The student has to provide an entire function that checks if a tree is balanced.

```
java -jar CorrectOz.jar input.oz -d false -nIf 1 -nCase 2 -leafCase true
```

template.py

```
1  local NumLeaves IsBalanced R in
2  @  @IsBalanced@@
3
4     R = {IsBalanced btree(1:42 left:btree(1:12 left:leaf right:leaf)
5  right:leaf)}
6  end
```