Ada Exercise 2

Christian Tabe christian.tabe@rwth-aachen.de
Andreas Wüstenberg andreas.wuestenberg@rwth-aachen.de
Johannes Neuhaus johannes.neuhaus@rwth-aachen.de

Task 1

a

default value

Components might have a default value, of course. One could easily initialize a record and omit already defaulted values. Every record member having a default value will be initialized with this value.

```
type Car (wheelcount : positive := 4) is -- default value for wheels
    record
         wheels : Integer := wheelcount; -- default value for component
         axis : Integer := wheelcount/2;
         seats : Integer := 5; -- default value without dependency of discriminant.
    end record;
```

discriminant constraint

The discriminant can be used to distinguish between slightly different records depending on the discriminant. For example:

```
type PERSON(SEX :  GENDER  :=  F) is
  record
    AGE :  INTEGER range 0 .. 123;
    case SEX is
      when M =>  BEARDED :  BOOLEAN;
      when F =>  CHILDREN  :  INTEGER range 0 .. 20;
    end case;
  end record;
```

taken from http://archive.adaic.com/standards/83rat/html/ratl-04-07.html#4.7.2.

b

No. The discriminant of R is missing. The last line should look like: `A : array(1..10) of R(100)` where 100 is just an example value.

Task 2

a

valid integer type declaration

```
type Week_Number_Type is range 1..52;
```

Create a type from scratch. Using range it becomes an integer like type but is a type for its own.

derived type declaration

```
type Week_Number_Type is new Integer range 1..52;
```

Derived Type inherits primitive operations from the parent type but still is incompatible. The type is a “new” kind of integer.

subtype declaration

```
subtype Week_Number_Type is Integer range 1..52;
```

Subtype is compatible to parent and sibling types. Therefor it is not a “new” subtype.

b

One can assign type - subtype and the other way round, without having the compiler complaining about the assignment. If there are assignments of two different types (e.g. Hour... - Month... - they are declared as types, not subtypes) the compiler will complain about it.

Therefor legal assignments are:
1, 2, 4

Nevertheless, 2 can lead to an error during compile time or runtime if the range does not fit.

Task 3

a

Exceptions are automatically raised and propagated trough a program's structure. In case an exception ‘e’ is not handled in ‘U’ and ‘U’ is left to ‘V’, ‘e’ might be handled in ‘V’. If not, again ‘V’ is left and ‘e’ will be propagated trough the whole program until the program terminates.

This can be build with `goto` statements, too, but only in knowledge of the callstack.

If an exception is raised and is not handled, it occurs a runtime error containing the information where the first entrance into the callstack was and where the respective raise statement is located. This information is hard to rebuild with `goto` statements. It even gets harder considering exception messages, which can be specified, too. Then a simple `goto` is not enough as no information can be passed to the handler without using global data structures.

In a nutshell, exceptions are a much more modular solutions than `goto` statements.

b

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure Navigation is -- main procedure embedding Position Calculation
    Position_Failure : exception;
    Navigation_Failure : exception;

    --
    -- Begin Mock
    --
    subtype Position_Type is Integer range 0..100;

    procedure Get_Position1 (Position : out Position_Type) is

    begin
        Put_Line("Entered 1. Position is: " & Integer'Image(Position));
        raise Position_Failure;
    end Get_Position1;

    procedure Get_Position2 (Position : out Position_Type) is

    begin
        Put_Line("Entered 2. Position is: " & Integer'Image(Position));
        raise Position_Failure;
    end Get_Position2;

    procedure Get_Position3 (Position : out Position_Type) is

    begin
        Put_Line("Entered 3. Position is: " & Integer'Image(Position));
        raise Position_Failure;
    end Get_Position3;

    --
    -- End Mock
    --

    pos : Position_Type;

begin

    --
    -- Interesting Part!
    --
    begin
        Get_Position1(pos);
    exception
        when Position_Failure =>
            begin
                Get_Position2(pos);
            exception
                when Position_Failure =>
```

```
                    begin
                        Get_Position3(pos);
                    exception
                        when Position_Failure =>
                            Put_Line("Everything handled and we still failed. Raise Navigation_Failure"
                            raise Navigation_Failure;
                    end;
            end;
    end;

end Navigation;
```

Task 4

a

`Parser/parser.adb`

b

`Parser/parser.adb`
Error handling using exceptions is a good way for providing a centralized and clean way of error handling. By using simple return statements it is not clear, when which error hast to be handled. Exceptions provide a clean centralized solution by attaching error handlers to the procedure raising them.

On the other hand exception handling bloats the code. For smaller code fragments and non-production code snippets, exceptions are probably over-engineered if not really necessary for reasons like 'failing fast' or providing fall back solutions for some computation (e.g. nicely shown in task 3).

In case of a parser, exceptions are probably the better solution. Exceptions should be used if an real error, like an invalid character, occurs. This needs to be handled and if it is not recoverable, the program should fail with some helpful error message.

Error handling becomes a central feature of a parser and should therefore not be done by using return statements which needs further interpretations.