Problem Set – Bonus
Course **Safe and Secure Software**
(Winter Term 2016)

Bauhaus-Universität Weimar, Chair of Media Security
Prof. Dr. Stefan Lucks, Eik List
URL: `http://www.uni-weimar.de/de/medien/professuren/mediensicherheit/teaching/`
Due Date: 10 Mar 2017, 1:30 PM, via email to `eik.list(at)uni-weimar.de`.
If you reach $\geq 50\%$, all members of your group will receive a 1/3 bonus for the exam.

Task 1 – Mini-Project – Linked Lists (4 Credits)
Implement a linked list and write a sufficient test suite for your implementation with either **`AUnit`** or **`testgen`**.

```
generic
    type Item_Type is private;
package Lists is
    type List_Type is tagged private;
    type List_Iterator_Type is private;

    function First(List: List_Type) return List_Iterator_Type;
    -- Returns an Iterator that points to the first item of the list.
    -- Raises Iterator_Error if list is empty.
    function Last (List: List_Type) return List_Iterator_Type;
    -- Returns an Iterator that points to the last item of the list.
    -- Raises Iterator_Error if list is empty.
    function Succ (Iterator: List_Iterator_Type) return List_Iterator_Type;
    -- Returns an Iterator that points to the next item in the list.
    -- Returns Iterator_Error if Iterator is null or points to the last item.
    function Pred (Iterator: List_Iterator_Type) return List_Iterator_Type;
    -- Returns an Iterator that points to the previous item in the list.
    -- Returns Iterator_Error if Iterator is null or points to the first item.
    function Value(Iterator: List_Iterator_Type) return Item_Type;
    -- Returns the value at the current position of Iterator.
    -- Returns Iterator_Error if Iterator is null or list is empty.
    function Size (List: List_Type) return Natural;
    -- Returns the number of items in List.

    procedure Append(List: List_Type; Item: Item_Type);
    -- Appends Item to the last position of the list.
    procedure Prepend(List: List_Type; Item: Item_Type);
    -- Adds Item at the first position of the list.
    procedure Insert(Iterator: in out List_Iterator_Type; Item: Item_Type);
    -- Inserts Item before the current position of Iterator.
    -- Sets Iterator to the new item.
    -- Raises Iterator_Error if the list of Iterator is null.
    procedure Delete(Iterator: in out List_Iterator_Type);
    -- Deletes the current Item of Iterator.
    -- Sets Iterator to the Item that came after the deleted item or to
    -- the new last Item if the previous last Item was deleted.
    -- Raises Iterator_Error if Iterator is null or points to no item.

    Iterator_Error: exception;
end Lists;
```

Update: The error cases for the methods **`First`** until **`Value`** have been specified clearer.

Task 2 – Mini-Project – Cycle Finding (4 Credits)
Implement Brent's Cycle-Detection algorithm in the following package **`Parallel_Brent`** for searching a partial collision of SHA-1 in parallel. Your implementation shall use at least two tasks.

```
with Brent_Context;

generic
    with package Context is new Brent_Context(<>);
package Parallel_Brent is
    use Context;
    procedure Search_Collision(First: out Point_Type; Second: out Point_Type);
end Parallel_Brent;
```

```
generic
    Num_Colliding_Bits: Natural;
    Num_State_Bits:     Natural; -- 160 for SHA-1
    Num_Tasks:          Natural;
package Brent_Context is
    NUM_BITS_PER_CHAR:  constant Natural := 4;
    NUM_STATE_CHARS:    constant Natural := Num_State_Bits / NUM_BITS_PER_CHAR;

    subtype Hash_Type is String(1 .. NUM_STATE_CHARS);

    type Point_Type is record
        Preimage: Hash_Type;
        Hash:     Hash_Type;
    end record;

    function "="(Left: Point_Type; Right: Point_Type) return Boolean;
    function Compute_Hash(Input: Hash_Type) return Hash_Type;
end Brent_Context;
```

Implement an Ada program that calls the package and provides the following CLI:

`./<program_name> <num_tasks> <time>`

Your program shall stop when the maximal running time has been reached or earlier if a partial collision was found. The program – if not interrupted – shall output the two distinct inputs leading to the collision with their corresponding hash values. Pressing **`q`** shall stop the program immediately.

Update: The greater-than comparison **`"<"(Point_Type; Point_Type)`** has been removed.

Task 3 – Mini Project – Strassen's Matrix Multiplication (4 Credits)
Implement the following package to compute the matrix multiplication $C = A \times B$ using the algorithm by Strassen, where A, B, and C are matrices with $n \times n$ elements a_i, b_i, c_i from the ring $(\mathbb{Z}_p, *, +)$, where p is a user-specified prime number defining the set of the matrix elements.

```
generic
    Num_Tasks: Positive;
    type Item_Type is mod <>;
package Strassen is
    type Matrix_Type is array(Natural range <>, Natural range <>) of Item_Type;
    Invalid_Size_Exception: exception;

```

```
8     function Multiply(A: Matrix_Type; B: Matrix_Type) return Matrix_Type;
9     -- Returns the result of multiplying both given (n * n)-elements matrices.
10    -- Raises an Invalid_Size_Exception if A and B differ in size.
11 end Strassen;
```

Write a wrapping Ada programm which should be called as follows:

`./<program_name> <A> <B> <num_tasks> <time>`,

where **`A`** and **`B`** denote file paths of the two input matrices, **`num_tasks`** the number of tasks, and **`time`** the maximal running time of your program.

Update: You can hard-code the prime number **`p`**. It has been removed from the command-line interface.

Task 4 – Mini Project – Parallel Hofstadter Q Sequence (3 Credits)

Implement the following Ada package that computes the Hofstadter Q sequence with multiple tasks in parallel:

```
1  generic
2      Num_Tasks: Positive;
3  package Hofstadter is
4      type Positive_Array is array(Positive range <>) of Positive;
5      -- Implement either or both of the following functions:
6      function Compute_Q(N: Positive) return Positive;
7      function Compute_Q_Sequence(N: Positive) return Positive_Array;
8  end Hofstadter;
```

Write a wrapping Ada programm which should be called as follows:

`./<program_name> <length> <num_tasks> <time>`

where **`length`** is the length of the Hofstadter Q sequence, **`num_tasks`** the number of tasks, and **`time`** the maximal running time of your program. Furthermore, pressing **`q`** should stop the program immediately. **Example:** The output of **`# ./<program_name> 4 2 1`** should be **`1, 1, 2, 3`** or a real subset thereof if the timeout was triggered before the full sequence was computed.

Update: You can implement either or both of the functions in the package **`Hofstadter`**.

Task 5 – Mini Project – Password Cracking (4 Credits)

When performing password cracking, one considers a certain number of hash values (targets) for which the corresponding preimages (passwords) have to be found. Given a file containing one target per line, implement the following interface to perform a parallel brute-force search (testing all possible values) to find the corresponding passwords.

```
1  with Password_Cracking_Context;
2
3  generic
4      with package Context is new Password_Cracking_Context(<>);
5  package Password_Cracking is
6      use Context;
7      procedure Find_Passwords(Hashes:    Hash_Array_Type;
8                               Passwords: out Password_Array_Type;
9                               Success:   out Boolean);
10 end Password_Cracking;
```

```
generic
    Num_Password_Chars: Positive;
    Num_Tasks: Positive;
package Password_Cracking_Context is
    subtype Hash_Type is String(1 .. 40);
    subtype Password_Type is String(1 .. Num_Password_Chars);

    type Hash_Array_Type is array(Natural range <>) of Hash_Type;
    type Password_Array_Type is array(Natural range <>) of Password_Type;

    function  Compute_Hash(Password: Password_Type) return Hash_Type;
    procedure First(Password: out Password_Type);
    function  Has_Next(Password: Password_Type) return Boolean;
    procedure Next(Password: in out Password_Type);
end Password_Cracking_Context;
```

Implement an Ada program which stops either if all passwords are found, the maximal time is reached, the user presses **q**, or if all password candidates are tested. At the end, the program should output all found passwords with their corresponding hashes.

A call to your program should look as follows:

./<program_name> <target_file> <num_tasks> <time>,

where **target_file** contains all hashes for which the preimage (password) should be found (one hash per line), **num_tasks** the number of tasks, and **time** specifies the maximal running time of your program. The hashes given below have been generated with SHA-1 processing 4-character passwords consisting of characters from the range [**a-z A-Z**] (52 letters):

```
535b82c80fa2805d71958d81eba421def23d310c
d501d3ec02af893f83107482b06013fc919eeb0b
0f940a4107f563ea63488e174109a8cb2cd701af
33484cb8db19ba718734c71296e4ab1257699c09
54b06d3dbdfb8de4166ca371641e3303741d0e7d
93ad1edafafa81c496f7c2f3af2e1d5129b146ff
85fe0a692de667c2592f864ad72e006bfb2cc363
1a102ed8778e14deab89282d4a774e2caaf13516
bca32807e309c4163cbd95c86195351550e509f6
297a3be49756374f3848f7809c2341f31cb7a7f9
6ca0e02880af53c5afad9fff53ce73bbb5b700f8
2241e7ccfde7108f55c6c23bb32aca015e91b77f
```

Bonus (+1): If your program finds the 5-characters passwords (taken from the same alphabet) for the following targets (one per line).

```
82e1654b3e2b70029f3d4347c82f0987386196b3
7eb0443b62987568d843eadd92e5fdf618341050
e734b8fb04a3a62f74b39cdb08366b995053b327
3a01be17246d588caf9a649f8a04e3e5d629db94
b4282885803090c68bf6d07737d089350884880c
```

Update: One hash has been removed from the 5-character password list.