Practical Exercises
Concepts of High-Level Programming Languages
2007-2008

cderoove@vub.ac.be

HIGH-LEVEL PROGRAMMING CONCEPTS IN PICO

The goal of these exercises is to make you acquainted with high-level concepts present in the programming language Pico. There are multiple exercises on the use and the syntax of each specific programming concept, of which we will solve a selection. The remainder of the exercises are meant to be solved individually in case you encountered any difficulties during the class.

A Quick Reminder: Various Pico Expressions

Exercise 1. Fill in the gaps from the following Pico-transcript. Pay attention to the difference between a value that is returned as the result of the evaluation of an expression (e.g. *b* below) and a text that might be displayed on the screen as a side effect of the evaluation of the same expression (e.g. *a* below).

```
> g(1,m) : 5
<function g>
> g(display(0), display(1))
   a b
> x : 1
   c
> y : 10
   d
> z : y + 3
   e
> f(x,y) :  x + y + z
<function f>
> f(2,4)
   f
> for(i:1, !(i=2),i:=i+1, {display(i); i-1})
   g h
> mystery(x) : if(x=0, display(x), {mystery(x-1); display(x); x})
<function mystery>
> mystery(5)
   i  j
> { x := 10;
make_multiplier(n) : multiplier(x) : n * x;
t : make_multiplier(3);
t(x)
}
   k
> { i : 0;
listing[x] : make_multiplier(i:=i+1);
t := listing[5];
t(10)
}
  l
>
```

Simple Functions and Operators

Exercise 2. Define a function `matrix(a1,a2,b1,b2)` that returns the following matrix consisting of the arguments the function was called with: $\begin{bmatrix} a1 & a2 \\ b1 & b2 \end{bmatrix}$.

Also write a second `determinant(m)` that calculates the determinant of 2x2-matrices originating from your `matrix(a1,a2,b1,b2)` function.

Exercise 3. Define a function `divides(number,divisor)` that indicates whether or not a given divisor divides a number exactly. Mind you that we can write this function without an if-construct.

Exercise 4. Define a less-than-or-equal operator `x <= y`.

Exercise 5. Define a binary operator `p !^ q` that implements the logical NAND operation.

In the logic truthtable, the result of this operation is always true except when `p` en `q` are both true. You are only allowed to use the built-in functions `true` and `false` and not the functions `and`, `or`, `not` or `if`.

It should be possible to use your operator in the following way:

```
[true !^ false, true !^ true,
 false !^ false, false !^ true] ⇒
[<native function true>, <native function false>,
<native function true>, <native function true>]
```

Repetition through Recursion and Iteration

Exercise 6. Predict the output of the expressions `count1(4)` and `count2(4)` when `count1` and `count2` are defined as follows:

```
count1(x) : if(x=0, display(x), {display(x); count1(x-1)});

count2(x) : if(x=0, display(x), {count2(x-1); display(x)});
```

Exercise 7. Define addition and substraction recursively using the functions `dec` and `inc` which respectively decrease and augment their arguments. You are only allowed to use the `+` and `-` operators in the definition of the `dec` and `inc` functions.

Exercise 8. Write a recursive function that prints a decimal number as a binary number. The rightmost bit is 1 if the number is odd while it is 0 when the number is even. To obtain the second least significannt bit, one has to divide the number by two and perform the same test. For the third least significant bit, one has to divide the previous quotient by 2 again and so on ... Make sure you don't print the binary number in reverse!

Exercise 9. Write a function `divisors(n)` that displays all divisors of `n` on a numbered axis. When a number has, for instance, no divisors

between 4 and 8, 3 spaces should be printed between 4 and 8. Use a `for`-loop for this exercise.

```
> delers(16)
1 2  4    8         16
>
```

Exercise 10. Define a function `sum_integers(a,b)` that calculates the sum of all integers between `a` and `b`. Do this recursively as well as iteratively.

Exercise 11. Define the functions `sin(x,n)` en `cos(x,n)` which approximate a sine and cosine using the first n terms of the following sums. Use a `while`-loop for this exercise.

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

Functions returning functions

Exercise 12. Define a function `return_multiplier(n)` that returns another function which will always multiply its given argument with the same number `n`:

```
> triple  : return_multiplier(3)
<function f>
> triple(2)
6
> double : return_multiplier(2)
<function f>
> double(2)
4
```

Functions as arguments for other functions

Exercise 13. Define a function returning the composition of two functions. More specifically, this function takes two functions `f(a)` and `g(b)` as arguments and returns the composition `compose(f,g) : f(g(x))`.

Exercise 14. Define analogously to the recursive definition of the function `sum_integers(a,b)` a recursive functie `sum_squares(a,b)` which will square every number between a and b before returning the sum of the results: `sum_squares(5,7) = 25 + 36 + 49` £

Exercise 15. Define the recursive function `sum_every_two_integers` in the same way such that: `sum_every_two_integers(5,9) = 5 +`

```
7 + 9
```

Exercise 16. By now you have probably discovered the general pattern in these exercises:

$$\sum_{i=a,next(a),\ldots}^{b} term(i)$$

It would be handy to express this pattern in a generic function `sum` that could be reused in all the previous situations. In order to be as generic as possible, this function will have to depend on two other functions:

- the function `next(a)` which will, given a number `a`, return the next number `i=next(a)` to be added, after application of `term` to `i`, to the sum accumulated so far
- the function `term(i)` which will perform an operation on a number whose result will make up a term in the sum

We can define this generic sum function as follows:

```
sum(term, next, a, b) :
  if(a > b,
     0,
     term(a) + sum(term,
                   next,
                   next(a), b))
```

We will also have to define a suitable `term` and `next` function. To calculate the sum of all the numbers in an interval, we use the identity function `id(x)` for `term` and an increment function `inc(x)` for `next`.

```
id(x) : x
inc(x) : x + 1
```

Subsequently, we can express `sum_integers` using the generic function `sum` in the following manner:

```
sum_integers(a,b) : sum(id, inc, a, b)
```

Exercise 17. Express the functions `sum_squares` and `sum_every_two_integers` using the generic function `sum`.

Exercise 18. Define a function `product(factor, next, a, b)` which calculates the following products:

$$\prod_{i=a,next(a),\ldots}^{b} factor(i) = factor(a) \times factor(next(a)) \times \ldots \times factor(b)$$

The arguments `a` en `b` are numbers (met b >= a) while the arguments `factor` and `next` are functions in 1 argument.

a/ Write this function in an iterative manner using a while, until or for-loop.

b/ Write this function in a recursive manner.

c/ Show how you can use this function to express the `factorial(n)` function. Assign an operator `!!(x)` to this function.

Exercise 19.
Consider the following expression approximating a k-term infinite fraction:

$$breuk = \cfrac{N_k}{\left(D_k + \cfrac{N_{k-1}}{\left(D_{k-1} + \cfrac{N_{k-2}}{\left(D_{k-2} + \cfrac{N_{k-3}}{D_{k-3} + \left(\ldots\ldots\ldots + \cfrac{N_1}{D_1 + 0} \right)} \right)} \right)} \right)}$$

Write a recursive Pico-function `f(n,d,k)` that takes a number k and the unary functions `n(i)` and `d(i)` (returning the numbers D_i en N_i respectively) and calculates the expression above.

The inverse of the golden ratio, $\frac{1}{\phi}$, can be approximated by putting all D_i and N_i in the fraction equal to 1. It should thus be possible to use your function in the following way:

```
{f_returning_1(i) : 1.0;
 f(f_returning_1, f_returning_1, 100) ^ -1} ⇒ 1.61803 ~ φ
```

Functional Parameters

The function `sum(term, next, a, b)` is somewhat tedious to use when we constantly need to define extremely small functions for the `term` parameter (e.g. `id(x) : x`) and the `next` parameter (e.g. `inc(x) : x + 1`). Pico offers a solution to this problem: functional parameters. These are parameters of a function that look themselves as functions. The following transcript should clarify things:

```
> f(a,g(x)) : g(1) + g(2) + g(a) + a
<function f>
> f(1,x*x)
7
```

Exercise 20. Study the following transcript and make sure you can explain the different ways in which the variable `x` is bound.

```
> x : 1000
1000
> f(x,a,g(x)) : x + a + g(a) + g(x)
<function f>
> f(3,50,x*x)
2562
> 3 + 50 + 2500 + 9
2562
```

Exercise 21. Alter the definition of the generic `sum` function so that it uses functional parameters and reimplement `sum_squares` and `sum_every_two_integers` in terms of your newly defined `sum` function.

Exercise 22. Do the same for the `product` and `factorial` functions.

Exercise 23. Alter the definition of your composition function `compose` so that it makes use of functional parameters. Pay attention to the names of the variables.

Exercise 24. Define a function (with functional parameter) `derivative(f(z))` which returns a function `derived(x)`. The function `derived(x)` is defines as follows:

$$derived(x) = \frac{f(x+dx) - f(x)}{dx}$$

```
> d : derivative(z^3 + z^2 + z + 10)
<function derived>
> d(5)
86.0002
>
```

Table generation

Remember that Pico has a special `t[idx]:exp` expression for generating tables. The `exp` expression is evaluated as many times as is indicated by the `idx` expression. The resulting table is bound to the `t` variable.

Exercise 25. Generate a table whose elements have a 1 in the even positions and a 0 in the odd positions.

Exercise 26. Generate a table whose elements are the factorial of each element's index in the table.

Exercise 27. Define a table `reverse_table(table)` that takes a table and returns a new table that is the reverse of the old one. You are not allowed to make use of recursion or iteration constructs.

Exercise 28. Define a function `numbers(b,e)` that returns a table containing the numbers between `b` and `e`.

Higher-order functions on tables

Exercise 29. Define a function that takes a table and multiplies each element by 2, returning a new table without changing the original table.

Exercise 30. Generalize the previous function to a `map(table, operation)` function which applies the `operation` function to each element of the given `table`, returning a new table while leaving the original one intact. Express the previous function using the generalized one.

Exercise 31. Define a function that takes a table and multiples each element by itself, destructively changing each element of the original table.

Exercise 32. Generalize the previous function to a `destructive_map(table, operation)` function which applies the `operation` function to each element of the given `table`, destructively updating each element of the original table. Express the previous function using the generalized one.

Exercise 33. Define a function that takes a table and displays each element of the given table.

Exercise 34. Generalize the previous function to a `for_each(table, operation)` function which applies the `operation` function to each element of the given table. Express the previous function using the generalized one.

Functions with a variable amount of arguments

Exercise 35. Implement a function `average@numbers` that calculates the average of the variable amount of `numbers` it is given.

Exercise 36. Implement a function `largest@numbers` that calculates the largest number of the variable amount of `numbers` it is given.

Exercise 37. Implement a function `accumulate@numbers` that calculates the sum of the variable amount of `numbers` it is given.

Exercise 38. Give the definitions Pico uses for its built-in `begin` and `tab` functions.

Function application through the @-operator

Exercise 39. Evaluate the expression `accumulate@[1,2,3,4].` With this knowledge, define a function `sumbetween(b,e)` using only `numbers(b, e)` en `accumulate@args.`