Functional and Logic Programming.
Lectures 1-7

Contents

1 Programming styles

The main programming styles of software engineering are

1. procedural programming
2. object-oriented programming (OOP)
3. functional programming (FP)
4. logic programming (LP)

The first two are **imperative** programming styles, and the last two are **declarative** programming styles.

Imperative programming styles perform computations by executing instructions (or commands) which change the state of a program (variables, ports, etc.). In imperative programming, programmers write programs that describe **how** to solve a problem by changing the program state.

Declarative programming styles perform computations according to a fixed and predictable strategy. In declarative programming, programmers write programs that describe **what** they know about the problem, and trust the built-in strategy which known how to compute the desired answer(s).

In FP, programs consists of definitions that describe what users know. Most definitions are definitions of functions, but some functional programming languages allows us to define other things as well, such as type classes and data types. Computation consists in computing the value of an expression with an evaluation strategy. The most popular evaluation strategies are

1. strict (or call by value) evaluation. Languages based on this evaluation strategy are called strict languages. Racket is a strict language.
2. lazy (or call by need) evaluation. Languages based on this evaluation strategy are called lazy languages. Haskell is a lazy language.

In LP, programs consist of definitions of relations that describe what users know. The most popular language is Prolog. Prolog programs consists of **rules** and **facts**. Computation consists in finding the answers to questions expressed in a way, with a search strategy called SLDNF resolution.

2 Characteristics of functional programming

- No assignment. Variables are just names given to expressions, and their values can not be changed.
- Data is immutable – it can not be changed. This implies that we can not change the content of a list, array, or any other composite data structure.

- Functions are values. In general, values are objects that can be passed as named, passed as arguments to function calls, returned as results of function calls, or stored as components of composite values.

  In FP, a value is an expression that evaluates to itself – it can not be reduced anymore.

- Repetitive computation is simulated by recursion.

3 A crash course to Racket

Racket is a language for FP. It is a dialect of Lisp.

> HISTORICAL NOTE: Lisp was the first high-level language designed for functional programming. It was developed by John McCarthy in 1958 and quickly became the favored programming language for AI research. Lisp derives from "**LIS**t **P**rocessor" because linked lists are one of Lisp's major data structures, and the source code of Lisp is made of lists. It is the second oldest high-level programming language, and is still in widespread use today.
>
> Lisp has changed since its early days, and many dialects have existed over its history. Today, the best-known general-purpose Lisp dialects are Racket, Common Lisp, Scheme and Clojure.

Like all dialects of Lisp, Racket has a peculiar syntax for writing expressions, called *fully perenthesised notation* (See below).

- Racket is a **strict** language. This means that
  - When we give a name x to an expression $expr$: first, we compute the value v of $expr$; next, we give the name x to value v.
  - When we call a function f with arguments $arg_1, \ldots, arg_n$: first, we compute the values $v_1, \ldots, v_n$; next, we call f with arguments $v_1, \ldots, v_n$.
- It is dynamically typed: types are not associated to variables, but only to values. There is no type checker to detect type errors before runtime.

Racket an be downloaded freely from https://racket-lang.org, for every major operating system, including Windows, Mac OS X, and Linux. The Racket Guide is a gentle quide for programmers who are new to Racket.

3.1 Syntax

Racket has a peculiar syntax, called **fully parenthesised notation**. This notation is used to write both definitions and expressions.

Like many other programming languages, Racket is *block-structured*. A block is a group of definitions and expressions which ends with an expression. Blocks can be used instead of expressions in many programming constructs, including the special forms `lambda` (for function definitions), `cond`, etc.

3.1.1 The syntax of expressions

An expression is either a literal for the value of a predefined data type, a function call, or a special form. The following are built-in datatypes in Racket:

Booleans, with literals `#t` (for boolean true) and `#f` (for boolean false). These constants are recognized by the function `boolean?`

Numbers, recognized by the function `number?`

Thare are many kinds of numbers. The most important ones are

- Integers, such as `6`, `99999999999999999` or `-12`. These constants are recognized by the function `integer?`
- IEEE floating-point representations of a number, such as `-3.0` or `3.14e+87`. These constants are recognized by the function `real?`

Strings, which are fixed-length array of characters. They are written as usual: delimited with double quotes. Typical examples are `"abc"` and `"I am Sam"`

Strings are recognized by the function `string?`

Symbols, which are Racket identifiers preceded with a quote (`'`). Typical examples are `'a` and `'X12`

Symbols are recognized by the function `symbol?`

Pairs are a composite datatype which joins together two arbitrary values with the `cons` constructor, and breaks them apart with the `car` and `cdr` selectors.

Pairs are recognized by the function `cons?`

Lists are a composite datatype which is recursively defined: it is either the constant `null`, or it is a pair whose second value is a list.

A list with component values $v_0, v_1, \ldots, v_n$ is built with the constructor `(list` v_0 v_1 ... v_n`)` and the i-th component of a list `lst` (starting from 0) is selected with

`(list-ref lst` i`)`

Lists are recognized by the function `list?`, and the empty list is recognized by the function `null?`

Vectors are fixed-length arrays with constant-time access of the values stored in their slots. A vector of length $n + 1$ with values $v_0, v_1, \ldots, v_n$ stored in its slots is built with the constructor `(vector` v_0 v_1 ... v_n`)` and the i-th component of a vector `vec` (starting from 0) is selected with

`(vector-ref vec` i`)`

Lists are recognized by the function `vector?`

The other kinds of expressions are function calls and special forms.

Function calls: instead of $f(arg_1, \ldots, arg_n)$ we write

(*f* arg_1 ... arg_n)

where f is either a function name, or an expression that describes a function.

- We use whitespace (instead of comma) to separate the arguments of a function call.
- Every open parenthesis must have a corresponding close parenthesis.

Special forms have special rules of evaluation. They are of the form

(*id* ...)

where *id* is the identifier of the special form. Examples of special forms are: `lambda`, `if`, `let`, `let*`, `cond`.

- **Abstractions** (also known as *lambda expressions*) are expressions that describe functions. They have the form

  (`lambda` (*x*$_1$ ... *x*$_n$) *block*)

  and describe an anonymous function which, for inputs $x_1, \ldots, x_n$, computes the value of *block*.

  - $x_1, \ldots, x_n$ are the **formal parameters** of the abstraction.
  - *block* is the **body** of the abstraction.

- **Conditional expressions** allow to choose what to compute depending on the result of a boolean test. The most important special forms for conditional computation are `if` and `cond`.

  (`if` *test* *expr*$_1$ *expr*$_2$)

  computes the value of $expr_1$ if the value of *test* is true[1]. Otherwise, it returns the value of $expr_2$.

  (`cond`
  [*test*$_1$ *block*$_1$]
  [*test*$_2$ *block*$_2$]
  ...
  [*test*$_n$ *block*$_n$])

  evaluates the boolean expressions $test_1, \ldots, test_n$ in this order.

  - As soon as it finds $expr_i$ whose value is true, it stops evaluating the other tests and evaluates $block_i$, whose value is returned as result.

[1] In Racket, a value is true if it is not `#f`.

  - If all $expr_i$ are `#f`, the evaluation returns the value `#void`

- The special forms `let` and `let*` allow to define local variables in the evaluation of an expression.

  (let ([x_1 $expr_1$]
        [x_2 $expr_2$]
        ...
        [x_n $expr_n$])
    $expr$)

  is an abbreviation of

  ((lambda (x_1 ... x_n) $expr$) $expr_1$ ... $expr_n$)

  The `let` special form behaves as follows:

  - ▶ It computes the values $v_1, \ldots, v_n$ of expressions $expr_1, \ldots, expr_n$, and assigns them to local variables $x_1, \ldots, x_n$
  - ▶ It uses $x_1, \ldots, x_n$ with these values in the computation of the value of $expr$.

  (let* ([x_1 $expr_1$]
         [x_2 $expr_2$]
         ...
         [x_n $expr_n$])
    $expr$)

  is an abbreviation of

  ((lambda (x_1)
     (lambda (x_2)
       ...
         (lambda (x_n) $expr$) ...))
   $expr_1$) $expr_2$) ...) $expr_n$)

  It behaves similar to `let`, with the only difference that we can use $x_1, \ldots, x_{i-1}$ in the expression $expr_i$ which gives the value of x_i.

3.1.2 The syntax of definitions

(define *name expr*)

assigns the value of *expr* to *name*. A function definition

(define *name* (lambda (x_1 ... x_n) *body*))

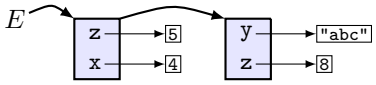
can be written using the simplified syntax

(define (*name* x_1 ... x_n) *body*)

3.2 Environment-based computations

The evaluation of any expression is performed in an environment which provides the values of the variables used in the expression.

An **environment** is a linked list of **frames**. A frame is a table which binds variables to values. For example



is an environment with two frames:

> The first (top) frame has the bindings `z` → `5` and `x` → `4`.
> The second frame has the bindings `y` → `"abc"` and `z` → `8`.

3.2.1 Variable lookup

Variable lookup is the operation of finding the value of a variable in an environment. If E is an environment and x a variable, then the value of x in E, written $E(x)$, is determined as follows:

- The frames of the environment are traversed until a frame is found with a binding $x \to v$. In this case, $E(x) = v$.
- If no frame of E has a binding for x, x is undefined in E and the variable lookup raises an error.

For example, $E($`x`$) =$ `4`, $E($`y`$) =$ `"abc"`, $E($`z`$) =$ `5`, and $E($`t`$)$ is undefined.

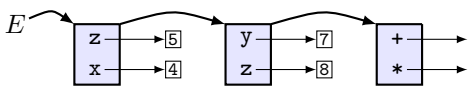
> Note that variable lookup ignores the binding `z` → `8` in the second frame because there is the binding `z` → `5` in the top frame. Such a situation, when a binding is ignored because a previous binding of the same variable in a previous frame, is called **shadowing**.

3.2.2 Evaluation of expressions

The environment is initialized with bindings for predefined variables when we start Racket. The built-in functions are just predefined variables with functions as values.

The value of an expression *expr* in an environment E is computed in two steps: (1) all variables x in *expr* are replaced with $E(x)$; and (2) the new expression is evaluated using the rules of evaluation.

For example, the value of `(+ x (* y z))` in environment E where



is `(+ x (* y z))`→`(+ 4 (* 7 5))`→`(+ 4 35)`→`39`

The underlined parts are the subexpressions that get replaced in every reduction step, until we obtain a value.

3.2.3 The interpretation of definitions

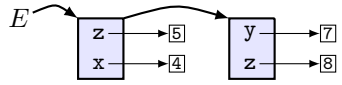
When the interpreter reads a definition

`(define` *var expr*`)`

in an environment E, it does the following:

1. It computes the value v of *expr* in E
2. It adds the binding $var \to v$ to the top frame of E.

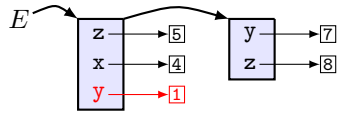
For example, suppose E is the environment



The value of `(+ x (* y z))` in E is `39` because

`(+ x (* y z))`→`(+ 4 (* 7 5))`→`(+ 4 35)`→`39`

The definition `(define y (- z 4))` computes the value `1` of `(- z 4)` în E, and adds the binding `y` → `1` to the top frame of E:



Afterwards, the value of `(+ x (* y z))` in E is different:

`(+ x (* y z))`→`(+ 4 (* 1 5))`→`(+ 4 5)`→`9`

This happens because the new binding `y` → `1` in the top frame shadows the binding `y` → `7` in the second frame.

3.2.4 Functions as values

Remember that the abstraction

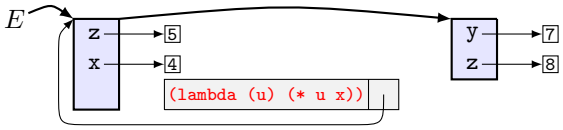
`(lambda (`x_1 ... x_n`)` *block*`)`

is an expression for the anonymous function which, for inputs $x_1, \ldots, x_n$ returns the value produced by the evaluation of *block*.

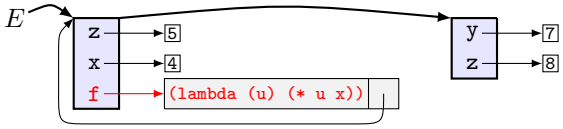
The evaluation of this expression in an environment E produces a **function value**, whose representation in the memory of the computer is the pair $\langle code, E \rangle$, where *code* is the expression `(lambda (`x_1 ... x_n`)` *block*`)` and E refers to the environment where the function was created. Such a pair is called a **closure**.

For example, the evaluation of `(lambda (u) (* u x))` in the environment

produces the function value



and the definition `(define (f u) (* u x))` in E extends the top frame of E with the binding depicted below.



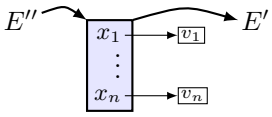
3.2.5 The evaluation of function calls

The evaluation of a function call

(*f* $expr_1$ ... $expr_n$)

in an environment E proceeds as follows:

1. $E(f)$ is looked up, which must be a closure $\langle$(`lambda` (*x*$_1$ ... *x*$_n$) *block*), $E'\rangle$.
2. The values $v_1, \ldots, v_n$ of $expr_1, \ldots, expr_n$ in E are computed.
3. The value v of *block* is computed in the environment

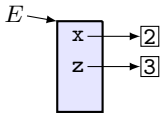
   The top frame of E'' is temporary and stores the values of $x_1, \ldots, x_n$ curing the evaluation of the function call.
4. The top frame of E'' is garbage-collected, and v is returned as result of the function call.

3.2.6 The evaluation of blocks

The evaluation of a block *block* in an environment E proceeds as follows:

1. E is extended with a temporary, initially empty, top frame
2. The components of *block* are evaluated one-by-one in this extended environment.
   - The definitions in the block will add bindings to the top frame.
   - The value of the last expression in *block* is returned as result, and the initial environment is restored, by garbage collecting the (temporarily created) top frame.

Here are some examples of how environment-based computation works. Let



The interpretation of the definition

```
(define f
  (let ([u (* x z)])
    (lambda (x y)
      (define v (- u x))
      (- v y))))
```

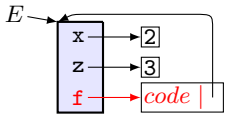
in E has the following effect:

1. We evaluate the `let` form in E. This is equivalent with the evaluation in E of the function call

   ```
   ((lambda (u)
      (lambda (x y)
         (define v (- u x))
         (- v y)))
    (* x z))                              ; E(x) = 2, E(z) = 3
   →((lambda (u) (lambda (x y) ...)) 6)
   →(lambda (x y) (define v (- 6 x)) (- v y))
   ```

   which produces the function value $\langle code, E\rangle$ where
   $code =$ `(lambda (x y) (define v (- 6 x)) (- v y))`

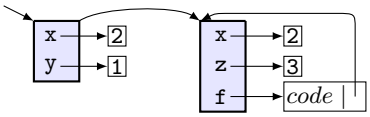
2. We add the binding `f` $\rightarrow \langle code, E\rangle$ to the top frame of E:

The evaluation of the function call `(f x (- z x))` in E proceeds as follows:

1. We compute the values `2` and `1` of `x` and `(- z x)` in E.

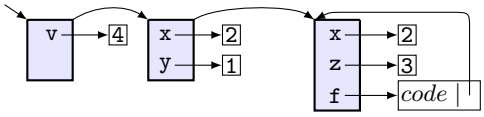
2. We extend E with a temporary frame that binds the formal parameters `x`, `y` of `f` to the values `2` and `1`:

   and evaluate the body of `f` in this environment. Since the body is a block, it's components will be evaluated in an extension of this environment with an empty top frame. The block definition

```
(define v (- 6 x))
```

computes the value `4` of `(- 6 x)` and adds the binding `v` → `4` to the top frame:



The last expression in the body of `f` is `(- v y)`. Its value in this environment is

`(- v y)`→`(- 4 1)`→`3`

Value `3` is returned as result of the function call, the temporary frames are garbage-collected, and the environment E is restored.

3.3 Recursion

In computer science, recursion is a method to define a data structure or solve a problem by decomposing it into smaller instances of the same kind.

In functional programming

- A function is recursive when it calls itself from within its code, directly or indirectly.
- A data structure is recursive if it is defined in terms of itself.
- All repetitive computations can be performed only by recursion.

The structure of a recursive function definition consists of

- one or terminating scenarios, called **base cases**, that do not use recursion to produce an answer.
- one or more **recursive cases** that reduce the computation, directly or indirectly, to simpler computations of the same kind.

For example, the following is a recursive definition of a function that computes the length of a list:

```
(define (length lst)
  (cond [(null? lst) 0]                              ; base case
        [(list? lst) (+ 1 (length (cdr lst)))]))     ; recursive case
```

To write a recursive function definition, use the following guidelines:

1. Try to break a problem into subparts, at least one of which is similar to the original problem. There may be many ways to do so. For example, if $m, n \in \mathbb{N}$ and $m > n > 0$ then
   $gcd(m, n) = gcd(m - n, n)$, or $gcd(m, n) = gcd(n, m \mod n)$

2. Make sure that recursion will operate correctly:
   - there should be at least one base case and one recursive case (it's OK to have more)
   - The test for the base case must be performed before the recursive calls.
   - The problem must be broken down such that a base case is always reached in a finite number of recursive calls.
   - The recursive call must not skip over the base case.
   - The non-recursive portions of the subprogram must operate correctly.
3. Ensure termination of the computation, by verifying that the reduction process will eventually lead to base case computation.

   For example, `length` is a terminating function because the recursive call is on a shorter list, therefore we will eventually reach the base case.

The rest of this section presents techniques to define recursive functions.

3.3.1 Functions defined by structural induction

Usually, a recursive datatype has a grammar that describes the structure of its data values. When functions operate on data from recursive datatypes, we can try to define them by structural induction:

> PRINCIPLE OF STRUCTURAL INDUCTION. When defining a function based on structural induction, the structure of the function should be patterned after the structure of the data.

Typically, the grammar for a composite datatype T is of the form

T `::=` `(`C_1 $T_{1,1}$ ... T_{1,k_1}`)` `|` ... `|` `(`C_n $T_{n,1}$ ... T_{n,k_n}`)`

where $C_1, \ldots, C_n$ are data constructors and $T_{i,j}$ are the types of its components. For example, the datatype of lists of integers is defined by the grammar

intList `::= null | (cons` *integer intList*`)`

which indicates that such a list is either the empty list `null`, or `(cons` n *lst*`)` where n is an integer and *lst* is a list of integers. The data constructors of *intList* are `null` and `cons`. A function f defined by structural induction on an argument x_i of this type has the structure

```
(define (f x_1 ... x_i ... x_n)
  (cond
     [(null? x_i) non-recursive computation]                ; base case
     [#t computation involving (f ... (cdr x_i) ...)]))     ; recursive case
```

Examples:

1. ; recognizer of intList
```
(define (intList? lst)
  (cond [(null? ls)] #t]
        [#t (and (list? lst) (intList? (cdr lst)))]))
```
or, even simpler:
```
(define (intList? lst)
  (or (null? lst) (and (list? lst) (intList? (cdr lst)))))
```

2. ; compute the sum of numbers in an intList
```
(define (n-sum lst)
  (cond [(null? lst) 0]
        [#t (+ 1 (n-sum (cdr lst)))]))
```

3. ; join two list by structural induction on the first list
```
(define (joinLists lst1 lst2)
  (cond [(null? lst1) lst2]
        [#t (cons (car lst1) (joinLists (cdr lst1) lst2))]))
```

Structural induction can also be used simultaneously on more than one argument. For example

```
; merge two lists of numbers sorted in increasing order
(define (mergeLists lst1 lst2)
  (cond [(null? lst1) lst2]                ; base case 1
        [(null? lst2) lst1]                ; base case 2
        [(< (car lst1) (car lst2))         ; recursive case 1
         (cons (car lst1) (mergeLists (cdr lst1) lst2))]
        [(> (car lst1) (car lst2))         ; recursive case 2
         (cons (car lst2) (mergeLists lst1 (cdr lst2)))]
        [(= (car lst1) (car lst2))         ; recursive case 3
         (cons (car lst1) (mergeLists (cdr lst1) (cdr lst2)))]))
```

3.3.2 Structural induction over the natural numbers

For some data values, such as numbers $n \in \mathbb{N}$, we there are many ways to choose a recursive structure. The choice should depend on the problem we want to solve. Here are some examples:

1. $n \in \mathbb{N}$ is either 0 (base case) or the successor of a natural number (recursive case). We can use this recursive structure to define the sum of the first n elements of a vector v with length at least n.

```
(define (vector-sum v n)
   (cond [(= n 0) 0]
         [(> n 0) (+ (vector-ref v (- n 1))
                     (vector-sum v (- n 1)))]))
```

Note that `(vector-ref v (- n 1))` is the `n`-th element of `v` because the elements of `v` are indexed starting from 0.

2. $n \in \mathbb{N}$ is either

   - 0 (case 1), or
   - $2 \cdot m$ with m a smaller natural number (case 2), or
   - $2 \cdot m + 1$ with m a smaller natural number (case 3).

   We can use this recursive structure to define a function that computes a^n for $a \in \mathbb{R}$ and $n \in \mathbb{N}$:

```
; (pow a n) computes a^n for a ∈ ℝ and n ∈ ℕ
(define (pow a n)
  (cond [(= n 0) 1]                                    ; case 1
        [(= (remainder n 2) 0) (pow (* a a) (/ n 2))]  ; case 2
        [#t (* a (pow (* a a) (/ (- n 1) 2)))]))       ; case 3
```

When a function f depends on two parameters $a, b \in \mathbb{N}$, we can try to define $(f \ \ldots \ a \ \ldots \ b \ \ldots)$ in terms of $(f \ \ldots \ c \ \ldots \ d \ \ldots)$ where $(c, d) \in \mathbb{N}^2$ is somehow "smaller" than $(a, b) \in \mathbb{N}^2$.

For example, we can consider $(c, d) < (a, b)$ if $c < a$, and define the gcd of $a, b \in \mathbb{N}^2$ as follows:

```
(define (gcd a b)
  (cond [(= b 0) a]                        ; base case 1
        [(> a b) (gcd b a)]                ; recursive case 1
        [#t (gcd b (remainder a b))]))     ; recursive case 2
```

This definition describes a terminating computation because every recursive call in the body of `(gcd a b)` is of the form `(gcd c d)` where `(c, d)` $<$ `(a, b)`.

3.3.3 Other kinds of recursion

Sometimes, we need more complicated arguments to prove that the reduction process of recursive calls will terminate.

To illustrate, consider the problem of flattening nested lists of symbols defined by the grammar

NSL `::= null | (cons` *symbol* *NSL*`) | (cons` *NSL* *NSL*`)`

For example, `'(((a (b c))) d)` is a nested list of symbols, and its flattened form is `'(a b c d)`.

To compute the flattened form of a nested list *lst*, we can reason as follows:

1. If *lst* is null, the flattened form is `null`.

2. Otherwise, it is `(cons` e_1 e_2`)` where $e_2 \in NSL$ and e_1 is either a symbol or $e_1 \in NSL$. We proceed by case analysis on the structure of e_1:

(a) If e_1 is symbol, we should compute `(cons` e_1 $lst2$`)` where $lst2$ is the flattened form of e_2 = `(cdr` lst`)`.

(b) If e_1 = `null`, we should compute the flattened form of e_2 = `(cdr` lst`)`.

(c) Otherwise, e_1 = `(cons` e_{11} e_{12}`)` and the flattened form of lst coincides with the flattened form of `(cons` e_{11} `(cons` e_{12} e_2`))`, which is `(cons (caar` lst`) (cons (cdar` lst`) (cdr` lst`)))`.

The implementation of this method in Racket is

```
(define (flatten lst)
  (cond [(null? lst) null]                                    ; case 1
        [(symbol? (car lst)) (cons (car lst)                  ; case 2
                                   (flatten (cdr lst)))]
        [(null? (car lst)) (flatten (cdr lst))]               ; case 3
        [#t (flatten (cons (caar lst)                         ; case 4
                           (cons (cdar lst) (cdr lst))))]))
```

Does this definition describe a terminating computation? The first two recursive calls in the body of `flatten` are

```
(flatten (cdr lst))
```

and they describe a terminating process by structural induction, but it is not obvious why the third recursive call

```
(flatten (cons (caar lst) (cons (cdar lst) (cdr lst))))
```

describes a terminating process. The reason is more subtle: if `lst` is of the form `(cons (cons` e_1 e_2`)` e_3`)` then `lst1` = `(cons` e_1 `(cons` e_2 e_3`))` is a nested list of symbols with the same flattened form as `lst`, but simpler in the sense that its first element has smaller depth. This simplification process is terminating because it will lead eventually to a case when the first element is either a symbol (case 2) or the empty list (case 3).

3.4 Tail recursion

Tail recursion is a programming technique to define functions which run fast and consume small space of memory. It is effective only in programming language whose compilers/interpreters implement a technique called *tail-call optimization*. Tail-call optimization is explained in the Course 3 of this lecture.

> Racket and Haskell perform tail-call optimization, therefore it is useful to learn how to write tail recursive definitions in Racket and Haskell.

A function is **tail recursive** if it is defined such that the recursive calls are the last things executed in the body of the function.

Examples of tail recursive function definitions:

1. ```
   ; (fact n) computes n!
   (define (fact n [result 1])
     (cond [(= n 0) result]                          ; base case
           [#t (fact (- n 1) (* n result))]))        ; recursive case
   ```
   Runtime: $O(n)$. Space complexity: $O(1)$.

2. ```
   ; (revList lst) computes the reverse of list lst
   (define (revList lst [result null])
     (cond [(null? lst) result]
           [#t (revList (cdr lst)
               (cons (car lst) result))]))
   ```
   Runtime: $O(n)$ where n is the length of `lst`. Space complexity: $O(1)$.

3. ```
   ; (fib n) computes the n-th Fibonacci number
   (define (fib n [A1 1] [A2 1])
       (if (= n 1)
           A1
           (fib (- n 1) A2 (+ A1 A2))))
   ```
   Runtime: $O(n)$. Space complexity: $O(1)$.

The following recursive definitions are **not** tail recursive:

1. ```
   (define (fact n)
      (if (= n 0)
          1
          (* n (fact (- n 1)))))
   ```
   Runtime: $O(n)$. Space complexity: $O(n)$.

2. ```
   (define (revList lst)
     (if (null? lst)
         lst
         (append (revList (cdr lst))
                 (list (car lst)))))
   ```
   Runtime: $O(n^2)$ where n is the length of `lst`. Space complexity: $O(n)$.

3. ```
   (define (fib n)
     (if (or (= n 1) (= n 2))
         1
         (+ (fib (- n 1) (fib (- n 2))))))
   ```
   Runtime: $O(2^n)$. Space complexity: $O(n)$.

3.5 Techniques to write tail-recursive definitions

3.5.1 Convert an iterative solution into a tail-recursive solution

Every iterative program with `for` or `while` loops can be transformed into an equivalent tail-recursive program:

- The variables that change in the loop become parameters of the recursive function that change in the recursive calls.

For example, $fact(N)$ has the iterative definition

$result = 1$
$n = N$
`while` $n > 0$
  $result = n * result$
  $n = n - 1$
`return` $result$

A tail-recursive definition is obtained by defining a function which has the argument *result* initialized with 1. In Racket:

```
(define (fact n [result 1])
  (if (= n 0) result (fact (- n 1) (* n result))))
```

3.5.2 Defining a more general recursive function

Some computations are special cases of more general computations which are easy to define by tail recursion. Here are some examples:

1. The computation of a^n for $a \in \mathbb{R}$ and $n \in \mathbb{N}$ is a special case of the computation of $c \cdot a^n$ where $a, c \in \mathbb{R}$ and $n \in \mathbb{N}$. A tail recursive definition for the computation of $c \cdot a^n$ is easy to derive from the observation that

$$c \cdot a^n = \begin{cases} c & \text{if } n = 0, \\ c \cdot (a^2)^{n/2} & \text{if } n > 0 \text{ is even}, \\ (c \cdot a) \cdot (a^2)^{(n-1)/2} & \text{if } n \text{ is odd}. \end{cases}$$

   In Racket:

```
; (pow a n c) computes  c · a^n for a, b ∈ ℝ and n ∈ ℕ.
(define (pow a n
             [c 1]) ; optional argument c initilized with 1
  (cond [(= n 0) c]
        [(even? n) (pow (* a a) (/ n 2) c)]
        [#t (pow (* a a) (/ (- n 1) 2) (* c a))]))
```

3.5.3 Tail recursion with accumulators

Some computations depend on the results of a finite number of previous recursive calls. We can use extra parameters, called **accumulators**, to refer to these results and pass them from one recursive call to another.

This technique works well for definitions of *linear recursive functions*, whose mathematical definition is of the following form: $f : \mathbb{N} \to \mathbb{R}$,

$$f(n) = \begin{cases} f_n & \text{if } 1 \leq n \leq k, \\ c_1 \cdot f(n-1) + \ldots + c_k \cdot f(n-k) & \text{if } n > k \end{cases}$$

where $f_1, \ldots, f_k, c_1, \ldots, c_k \in \mathbb{R}$ are given constants.

An iterative, accumulator-based computation, of this function is

$a_1 = f_1$; ...; $a_k = f_k$
for $i = 1$ to n
  if $i = n$
    return a_1
  else
    $a_1 = a_2; a_2 = a_3; \ldots; a_{k-1} = a_k$;
    $a_k = c_1 \cdot a_k + c_2 \cdot a_{k-1} + \ldots + c_k \cdot a_1$;

Note that, when $i = n$, we have $a_1 = f(n), a_2 = f(n+1), \ldots, a_k = f(n+k)$. The conversion of this iterative code into a tail recursive code is straightforward. In Racket:

```
(define (f n [i 1] [a1 f1] ... [ak fk])
  (cond [(= i n) a1]
        [#t (f n (+ i 1) a2 ... ak (+ (* c1 ak) ... (* ck a1)))]))
```

The accumulator-based technique can se used to write tail recursive definitions of any primitive-recursive function (See Course 3).

3.5.4 Limitations of tail recursion

It is well known that some recursive definitions can not be defined by tail-recursion, and their computation can become highly inefficient and slow. The classical example is the Ackermann function $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0, \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

```
; Definition of Ackermann function in Racket
(define (A m n)
  (cond [(= m 0) (+ n 1)]
        [(= n 0) (A (- m 1) 1)]
        [#t (A (- m 1) (A m (- n 1)))]))
```

To appreciate the inefficiency of this definition, try to compute `(A 5 0)`.

Another operation that can be defined by recursion, but not by tail recursion, is the computation of the disjunctive normal form (DNF) of a propositional formula. We consider representations of propositional formulas defined by

```
F ::= symbol                ; representation of atomic formula
    | (list 'and F1 F2)     ; representation of F1 ∧ F2
    | (list 'or  F1 F2)     ; representation of F1 ∨ F2
    | (list 'not F1)        ; representation of ¬F1
    | (list 'impl F1 F2)    ; representation of F1 → F2
```

For example, the representation of the propositional formula $(\neg A \vee B) \rightarrow (B \wedge C)$ is `'(impl (or (not A) B) (and B C))`.

A disjunctive normal form is a propositional formula whose representation is defined by the grammar

DNF ::= *conj* | `(list 'or` *DNF DNF*`)`
conj ::= *literal* | `(list 'and` *conj conj*`)`
literal ::= *atom* | `(list 'not` *atom*`)`
atom ::= *symbol*

and the representation of a formula into an equivalent disjunctive normal form can be obtained in 3 steps:

1. Eliminate implications by the top-down propagation of the transformation

   Replace $P \rightarrow Q$ with $(\neg P) \vee Q$.

2. Eliminate negation by the top-down propagation of the following transformations:

   Replace $\neg(P \vee Q)$ with $(\neg P) \wedge (\neg Q)$,
   Replace $\neg(P \wedge Q)$ with $(\neg P) \vee (\neg Q)$,
   Replace $\neg(\neg P)$ with P.

3. Distribute conjunctions inwards over disjunctions by the bottom-up propagation of the following transformations:

   Replace $P \wedge (Q_1 \vee Q_2)$ with $(P \wedge Q_1) \vee (P \wedge Q_2)$,
   Replace $(P_1 \vee P_2) \wedge Q$ with $(P_1 \wedge Q) \vee (P_2 \wedge Q)$.

For example, this method computes the DNF of $\neg(\neg A \vee B) \wedge (B \rightarrow C)$ as follows:

$$\neg(\neg A \vee B) \wedge \underline{(B \rightarrow C)} = \underline{\neg(\neg A \vee B)} \wedge (\neg B \vee C) = (\underline{\neg(\neg A)} \wedge \neg B) \wedge (\neg B \vee C)$$
$$= \underline{(A \wedge \neg B) \wedge (\neg B \vee C)} = ((A \wedge \neg B) \wedge \neg B) \vee ((A \wedge \neg B) \wedge C)$$

We will implement the function `(DNF F)` which computes the representation of the DNF of a propositional formula F. First, we define some auxiliary recognizers for propositional formulas:

```
; (conj? F) holds if F represents a conjunction F1 ∧ F2
(define (conj? F) (and (list? F) (eq? (car F) 'and)))
; (disj? F) holds if F represents a disjunction F1 ∨ F2
(define (disj? F) (and (list? F) (eq? (car F) 'or)))
; (impl? F) holds if F represents an implication F1 → F2
(define (impl? F) (and (list? F) (eq? (car F) 'impl)))
; (atom? F) holds if F represents an atom
(define atom? symbol?)
; (neg? F) holds if F represents a negated formula ¬F
(define (neg? F) (and (list? F) (eq? (car F) 'not)))
; (mk-and F1 F2) makes the representation of F1 ∧ F2
(define (mk-and F1 F2) (list 'and F1 F2))
; (mk-or F1 F2) makes the representation of F1 ∨ F2
```

```
(define (mk-or F1 F2) (list 'or F1 F2))
; (mk-neg F) makes the representation of ¬F
(define (mk-neg F) (list 'not F))
```

Next, we define the transformations corresponding to the three top-down steps mentioned before.

```
;  Step 1.
; (elim-impl F) eliminates '→' from F by top-down propagation
(define (elim-impl F)
  (cond [(atom? F) F]
        [(neg? F) (mk-neg (elim-impl (cadr F)))]
        [#t (let ([op (car F)] [F1 (cadr F)] [F2 (caddr F)])
              (cond [(impl? F) (mk-or (mk-neg (elim-impl F1))
                                      (elim-impl F2))]
                    [#t (list op (elim-impl F1) (elim-impl F2))]))]))

; Step 2.
; (elim-not F) eliminates negations from F by top-down propagation
(define (elim-not F)
  (cond [(atom? F) F]
        [(neg? F)
         (define F1 (cadr F))
         (cond [(neg? F1) (elim-not (cadr F1))] ; ¬(¬P) = P
               [(atom? F1) F]
               [#t (let ([F11 (cadr F1)]
                         [F12 (caddr F1)])
                     (cond [(disj? F1)          ; ¬(P ∨ Q) = (¬P) ∧ (¬Q)
                            (mk-and (elim-not (mk-neg F11))
                                    (elim-not (mk-neg F12)))]
                           [(conj? F1)          ; ¬(P ∧ Q) = (¬P) ∨ (¬Q)
                            (mk-or (elim-not (mk-neg F11))
                                   (elim-not (mk-neg F12)))]))])]
        [#t (let ([op (car F)]
                  [F1 (cadr F)]
                  [F2 (caddr F)])
              (list op (elim-not F1) (elim-not F2)))]))

; Step 3.
; (distr F) distributes '∧' inwards over '∨' in F by bottom-up propagation
(define (distr F)
  (cond [(or (atom? F) (neg? F)) F]
        [#t (let ([P (distr (cadr F))]
                  [Q (distr (caddr F))])
              (cond [(disj? F) (mk-or P Q)]
                    [(conj? F)
                     (cond [(disj? Q) ; P ∧ (Q1 ∨ Q2) = (P ∧ Q1) ∨ (P ∧ Q2)
                            (mk-or (mk-and P (cadr Q))
                                   (mk-and P (caddr Q)))]
                           [(disj? P) ; (P1 ∨ P2) ∧ Q = (P1 ∧ Q) ∨ (P1 ∧ Q)
                            (mk-or (mk-and (cadr P) Q)
```

```
                    (mk-and (caddr P) Q))]
          [#t (mk-and P Q)])))]))
```

Finally, we can define the function DNF by composing the operations that perform the three steps:

```
(define (DNF F) (distr (elim-not (elim-impl F))))
```

Note that `(elim-impl F)` and `(distr F)` are defined by structural induction because the recursive calls are on some parts of `F`, but `(elim-not F)` is not defined by structural induction. Therefore, this recursive definition of `DNF` is not by primitive recursion.

3.6 Problem solving strategies

There are two well known problem-solving strategies: top-down and bottom-up.

The **top-down approach**, also known as decomposition, solves a problem by starting from the most abstract specification that can describe succintly the solution, and decomposing it incrementally into simpler sub-problems until we reach situations with straightforward implementations. This approach is suitable for procedural programming and functional programming, because most procedures and functions have a compositional structure that can be identified by top-down refinement. For instance, defining functions by structural induction is based on the top-down approach.

The **bottom-up approach** is based on the use of a small predefined set of operations that can be combined to solve a large variety of problems.

- In Racket, such a set consists of the predefined functions `map`, `apply`, `filter`, `foldl` and `foldr`.
- In Haskell, such a set consists of the predefined functions `map`, `filter`, `foldl` and `foldr`. (Note that `apply` does not exist in Haskell).

These two strategies (top-down and bottom-up) can be combined in creative ways to solve a particular problem.

Below is a summary of what these predefined functions do.

3.6.1 `map` (Haskell and Racket)

In Haskell, `map` is a polymorphic function of type `(a->b)->[a]->[b]`. The syntax call is

```
map f lst
```

where `f::a->b` is a function, and `lst` is a list of elements of type `a`, say $[a_1, \ldots, a_n]$. The result is the list $[b_1, \ldots, b_n]$ where $b_i =$ (`f` a_i) for $1 \leq i \leq n$. For example:

```
> map (+1) [1,2,3]
[2,3,4]
```

```
> map (\(x,y)->x*y) [(2,5) (3,4)]
[10,12]
```

In Racket, the syntax call is

`(map f lst`$_1$ `...` `lst`$_k$`)`

where `f` is a function that can take k input arguments, and `lst`$_1$, ..., `lst`$_k$ are k lists of the same length. If all of them have length n and

$$\texttt{lst}_i = \texttt{(list } v_{i1} \texttt{ ...} v_{in}\texttt{)} \text{ for } 1 \leq i \leq k$$

then the result is the list `(list` v_1 `...`v_n`)` where $v_j =$ `(f` v_{1j} `...` v_{kj}`)` for $1 \leq j \leq n$. For example:

```
> ; same as map (+1) [1,2,3] in Haskell
  (map (lambda (x) (+ x 1)) '(1 2 3))
'(2 3 4)
> ; same as map (\(x,y)->x*y) [(2,5) (3,4)] in Haskell
  (map (lambda (p) (* (car p) (cadr p))) '((2 . 5) (3 . 4)))
'(10 12)
> ; has no obvious Haskell counterpart
  (map cons '(a b c) '(6 0 5))
'((a . 6) (b . 0) (c . 5))
> ; has no obvious Haskell counterpart
  (map (lambda (x) (if (symbol? x) "symbol" #f)) '(a #t (1 2) b))
'("symbol" #f #f "symbol")
```

3.6.2 `apply` (only in Racket)

In Racket, the function call

```
(apply f lst)
```

takes as inputs a list `lst` and a function `f` that can take as inputs the elements of `lst`. If `lst` = `(list` a_1 `...` a_n`)` the result is the value of `(f` a_1 `...` a_n`)`. For example:

```
> (apply + '(1 2 3 4))
10
```

```
> (map (lambda (l) (apply * l))
       '((1 2) (5 3) (-1 2)))
'(2 15 -2)
```

3.6.3 `filter` (Haskell and Racket)

In Haskell, `filter` is a polymorphic function of type `(a->Bool)->[a]->[a]`. The syntax call is

```
filter p lst
```

where `p` is a predicate (that is, a function of type `a->[Bool]`), and `lst` is a list of elements of type `a`, say `[`$a_1, \ldots, a_n$`]`. The result is the list of elements $a_i \in$ `lst` for which `(p` a_i`)` is true. For example:

```
> filter (>2) [3,1,4]   > filter (\(x,y)->x<y) [(2,5),(4,3),(1,6)]
[3,4]                   [(2,5),(1,6)]
```

In Racket, the syntax call is

```
(filter p lst)
```

and has the same effect as in Haskell. For example:

```
> ; same as filter (>2) [3,1,4] in Haskell
  (filter (lambda (x) (> x 2)) '(3 1 4))
'(3 4)
> ; same as filter (\(x,y)->x<y) [(2,5) (4,3),(1,6)] in Haskell
  (filter (lambda (p) (< (car p) (cadr p)))
          '((2 . 5) (4 . 3) (1 . 6)))
'((2 . 5) (1 . 6))
```

3.6.4 `foldl` and `foldr` (Haskell and Racket)

In Haskell, `foldl` and `foldr` are polymorphic functions:

> `foldl` has type `(b->a->b)->b->[a]->b`
> `foldr` has type `(a->b->b)->b->[a]->b`

This implies that their syntax calls are `(foldl f v lst)` and `(foldr g v lst)` where `f::b->a->b`, `g::a->b->b`, `v::b` and `lst::[a]`. If `lst=`$[a_1, \ldots, a_n]$ then

> `(foldl f v lst)` returns the value of `f (... (f (f v` a_1`)` a_2`)...)` a_n
> `(foldr g v lst)` returns the value of `f` a_1 `(f` a_2 `( ... (f` a_n `v)...))`

Both function calls return the value of `v` when `lst` is the empty list.

In Racket

- `foldr` behaves like in Haskell
- `(foldl f v lst)` behaves slightly different: it returns the value of `(f` a_n `(... (f` a_2 `(f` a_1 `v))...))`.

  This means that `(foldl f v lst)` in Racket is the same as `foldl (\b a->f a b) v lst` in Haskell.

4 A crash course to Haskell

Haskell is a language for FP. It emerged in 1997 from an effort to standardize lazy programming languages.

HISTORICAL NOTE: People became interested in lazy functional languages after the release of the proprietary software Miranda in 1985. In 1987, a committee was formed to define an open standard for lazy functional languages. The committee's efforts culminated in 1997 with the release of *Haskell 98*, which specifies such a minimal and portable language, and as a base for future extensions. The language continues to evolve rapidly, with the Glasgow Haskell Compiler (GHC) implementation representing the current de facto standard.

It is a **lazy** language, based on call-by-need evaluation. This means that

- When we give a name x to an expression $expr$, we don't compute the value of $expr$ but bind name x to the unevaluated expression $expr$.
- When we call a function, we evaluate its arguments only as much as it is needed to proceed with the computation of the result of the function call.

Haskell is statically typed. Types are associated to variables at compile time. A type checker checks the types specified by the programmer and detects type errors before runtime. Moreover, a type inference system can compute the types omitted by the programmer. Haskell has one of the most advanced type systems developed so far, with polymorphic types and type classes.

Haskell can be downloaded freely from https://www.haskell.org/downloads/ for every major operating system, including Windows, Mac OS X, and Linux.

4.1 The syntax of Haskell

A **function call** $f(expr_1, \ldots, expr_n)$ is written `(`f $expr_1$ $\ldots$ $expr_n$`)`. Like in Racket, we use whitespace instead of comma to separate the arguments of a function call. Parentheses are necessary only when we want to disambiguate the parsing of expressions (see Remark below).

Some binary functions are written between their arguments. These kind of functions are called *operators*. The names of operators do not start with a letter, but the names of all other functions must start with a letter. Typical examples are the operators `+` for addition, and `*` for multiplication.

REMARKS. Haskell uses the following rules of disambiguation:

- Function application has higher priority than operator application. Example: $f\ x + g\ y$ is parsed as $(f\ x) + (g\ y)$
- Function application is left-associative: $f\ x\ y\ z$ is parsed as $(((f\ x)\ y)\ z)$

Operator application x *op* y can be converted into function application, by writing (*op*) x y. For example

```
3 + 4 → 7
(+) 3 4 → 7
```

Binary function application f x y can be converted into operator application, by writing x ‘f‘ y. For example

```
mod 8 3 → 2
8 ‘mod‘ 3 → 2
```

A **lambda expression** $\lambda x.expr$ is written \$x$->$expr$, and $\lambda x_1.\cdots.\lambda x_n.expr$ is written \$x_1$ ... x_n->$expr$. It is also possible to write lambda expressions of the form \$patt.expr$ where *patt* is a pattern. For example

```
(\(x,_,(y:_))->x+y) (1,True,[4,2,5]) → 5
```

pentru că potrivire pattern-ului `(x,_,(y:_))` cu `(1,True,[4,2,5])` produce substituţia `[1/x,4/y]`, iar `[1/x,4/y](x+y)=1+4` → `5`.

Predefined datatypes

The following types are predefined:

- `Bool` for the type of boolean values `True` and `False`.
- `Integer` for integers of arbitrary size.
- `Int` for integers of fixed size.
- `Char` for characters, like `'a'`, `'A'` and `'Z'`.
- `Float` for floating-point numbers with single-word precision
- `Double` for floating-point numbers with double-word precision
- If $T, T_1, T_2, \ldots, T_n$ are types then
  - `[`T`]` is the type of lists `[`$v_1, v_1, \ldots$`]` of elements of type T. The empty list is `[]`.
  - T_1`->`T_2 is the type of functions which map inputs of type T_1 to outputs of type T_2.
  - `(`$T_1, \ldots, T_n$`)` is the type of tuples `(`$v_1, \ldots, v_n$`)` with components of type $T_1, \ldots, T_n$.

Strings are lists of characters. For example, `"abc"` is an abbreviation of the list `['a','b','c']`. Thus, the type of strings is `[Char]`.

Definitions

A definition gives a name (or identifier) to an expression of a particular type.

```
name::type      -- declare name of type type
name = expr     -- creates a binding of name to expr
```

For example:

```
x,y :: Int          -- declare x,y of type Int
x = 12 + 13
y = y + 1           -- example of a recursive binding
```

If we omit type declarations, Haskell tries to infer the type of *name* from the type of *expr*. There are very few cases when this is impossible.

expr is **not** evaluated: the environment stores a binding of *name* to *expr*.

> More details about function definitions, type classes and type declarations can be found in the lecture notes (Lectures 5, 6, 7).

4.2 Working with Haskell

To work with Haskell, we will use `ghci`, the interactive environment of the Glasgow Haskell Compiler. This tool can be started from a shell with the command `ghci`:

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

The tool recognizes definitions, expressions, and commands that are typed by the user at the prompt. Haskell recognizes comments too: they start with '`--`' and extend to the end of line. Comments are used to document the written code, and are ignored (not read) by `ghci`.

Commands have the form

`:command args`

The most important `ghci` commands are:

`:load` *Name* reads definitions of module *Name* from file *Name*`.hs`
`:quit` quits the current working session with Haskell.
`:type` *expr* prints the type of *expr* without evaluating it.
`:set +t` sets ghci to show the type of each variable bound by a statement.

We will typically use the following workflow:

1. Write a Haskell program *Name*`.hs` and save it in your home directory. This is a text file with the following structure

```
module Name where
name1 ::type1          -- type declaration
definition1            -- definition
...
```

Definitions assign names to expressions, or define type classes or datatypes.

2. Start `ghci` and load the definitions form your program with the command

   `:load` *Name*

   After loading the program, the prompt will change to

   `*`*Name*`>`

3. Perform computations by evaluating expressions that make use of the definitions in your program.

If the program is modified or extended with new definitions, it should be reloaded.

5 Exercises

5.1 Labworks 1 (Racket) – Homework

HW1. A list is *good* if it is either empty, or it is of the form

`(list` s_1 n_1 ... s_m n_m`)`

where $s_1, \ldots, s_m$ are symbols, and $n_1, \ldots, n_m$ are numbers. Define recursively a predicate `(good-list? l)` which returns `#t` if `l` is a good list, and `#f` otherwise. For example:

```
> (good-list? null)     > (good-list? '(a 1 b 2 c 3/4))
#t                      #t
> (good-list? "abc")    > (good-list? '(1 a b))
#f                      #f
```

Remember that `list?` recognises lists, `null?` recognises the empty list, `symbol?` recognises symbols, and `number?` recognises numbers.

HW2. Define recursively a function `(symb-value l s)` which takes as input a *good* list `l` and a symbol `s` which occurs in `l`, and returns the number that appears immediately after `s` in `l`.

For example

```
> (symb-value '(x 2 y 3 z 4 t 5) 'z)
4
> (symb-value '(a 3.14) 'a)
3.14
```

HW3. Define recursively the predicate `(mem l v)` which returns `#t` if `v` is an element of the list `l`, and `#f` otherwise. Use the predicate `equal?` to check if two elements are equal.

For example:

```
> (mem? '(1 a b a c) 'a)          > (mem? '(1 a b a c) 'd)
#t                                #f
> (mem? '(1 (2 3) 4) '(2 3))      > (mem? '() '())
#t                                #f
```

HW4. Define recursively a function `(add l)` which takes as input a list of numbers, and computes the sum of its elements. If `l` is `null`, the function should return `0`.

HW5. Define recursively a function `(mult l)` which takes as input a list of numbers, and computes the sum of its elements. If `l` is `null`, the function should return `1`.

HW6. A nested list of numbers is either the empty list, or a list whose elements are either numbers, or nested lists of numbers. Define recursively a predicate `(nlist? l)` which returns `#t` if `l` is a nested list of numbers, and `#f` otherwise. For example:

```
> (nlist? null)     > (nlist? '(((1) 2) 3.2 ((4))))
#t                  #t
> (nlist? 1)        > (nlist? '(4 ((-5) a)))
#f                  #f
```

HW7. Define recursively the following functions that take as input a nested list of numbers `l`:

(a) `(add-all l)` which computes the sum of all elements in list `l`. If `l` contains no number, this function should return 0.

(b) `(max-elem l)` which returns the maximum number that occurs in list `l`. If If `l` contains no number, this function should return `0`.

(c) `(max-depth l)` which returns the maximum number of nested parentheses in list `l`. For example:

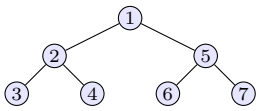
```
> (max-depth '())                  > (max-depth '(1 2 3))
1                                  1
> (max-depth '((1 2) (3 ((0)))))   > (max-depth '(1 ((()))
3                                  3
```

You can make use of the predefined function `(max` m n`)` which returns the maximum of numbers m and n.

5.2 Labworks 2 (Racket) – Recommended

LW1. Consider binary trees of integers defined by the BNF

`BTI ::=` *integer* `|` `(list` n `BTI BTI)`

For example, the binary trees of integers 

is represented by the list `'(1 (2 3 4) (5 6 7))`. A recognizer of this representation of binary trees of integers is

```
(define (BTI? bt)
   (or (integer? bt)
       (and (list? bt)
            (= (length bt) 3)
            (integer? (car bt))
            (BTI? (cadr bt))
            (BTI? (caddr bt)))))
```

Also, consider the following tree traversal strategies:

preorder: visit root, then left subtree, then right subtree.

inorder: visit left subtree, then root, then right subtree.

postorder: visit left subtree, then right subtree, then root.

Define recursively the functions `(preorder bti)`, `(inorder bti)`, and `(postorder bti)` which return the list of nodes in the binary treee of integers `bti` in the order in which they are visited. For example:

```
> (preorder '(1 (2 3 4) (5 6 7)))
'(1 2 3 4 5 6 7)
> (inorder '(1 (2 3 4) (5 6 7)))
'(3 2 4 1 6 5 7)
> (postorder '(1 (2 3 4) (5 6 7)))
'(3 4 2 6 7 5 1)
```

LW2. A nested list of numbers is either the empty list, or a list whose elements are either numbers, or nested lists of numbers.

(a) Write down the BNF definition for the recursive type `nlist` of nested lists of numbers.

(b) Define recursively the recogniser `(nlist? l)` for values of type `nlist`. For example:

```
> (nlist? null)     > (nlist? '(((1) 2) 3.2 ((4))))
#t                  #t
> (nlist? 1)        > (nlist? '(4 ((-5) a)))
#f                  #f
```

LW3. The decimal representation of a non-negative integer `N` is $d_n d_{n-1} \ldots d_1 d_0$ where $n \geq 0$, and the sum of its digits is $d_0 + d_1 + \ldots + d_{n-1} + d_n$.

Suppose we wish to define the function

```
(digit-sum N)
```

which computes the sum of digits of the non-negative integer `N`.

Note, again, that `N` does not have an explicitly defined recursive structure. However, we observe that non-negative integers **do have** a recursively defined structure, but we have to define our own recognisers and selectors for it:

Base case: `N` consists of a single decimal digit. In this case `(digit-sum N)` coincides with `N`.

Recursive case: `N` is of the form $\texttt{10} \cdot \texttt{M} + \texttt{D}$ where $\texttt{M} < \texttt{N}$ is a positive integer, and `D` is the last decimal digit of `N`. In this case, we must add `D` with the value of `(digit-sum M)`.

To take advantage of this structure of non-negative integers, we must define the recogniser

- `(is-digit? N)` which recognises if `N` is a decimal digit

and the selectors

- `(drop-last-digit N)` which returns the number `M` obtained by dropping the last digit of `N`, and
- `(last-digit N)` which returns the value of last digit `D` of `N`.

when $\texttt{N} > \texttt{10}$.

LW4. Define the function `(flatten sl)` which takes as input a nested list of symbols, and returns the list of symbols contained in `sl` in the order in which they occur when `sl` is printed. Intuitively, `flatten` removes all the inner parentheses form its argument. For example:

```
> (flatten '(a b c))
'(a b c)
> (flatten '((a b) c (((d)) e)))
'(a b c d e)
> (flatten '((a) () (b ()) () (c)))
'(a b c)
```

Suggestion: First, write a recursive definition (BNF) for the nested lists of symbols.

LW5. Define the function `(swapper s1 s2 sl)` which takes as input the symbols `s1` and `s2` and the list of symbols `sl`, and returns the list of symbols which is the same as `sl`, but with all occurrences of `s1` replaced with `s2` and all occurrences of `s2` replaced by `s1`.

For example, `(swapper 'a 'b '(a b r a c a d a b r a))` should produce the list `'(b a r b c b d b a r b)`

5.3 Labworks 3 (Racket) – Homework

Exercise set 1

1. Define the function `(joinLists lst1 lst2)` which returns the results of joining lists `lst1` and `lst2`. For example, `(joinLists '(1 2) '(a b))` should return `'(1 2 a b)`.

2. (LW3 from labwork 2) Define the numeric function `(digit-sum N)` which computes the sum of digits of the decimal representation of `N`. For example, `(digit-sum 726)` should return `15` because $7 + 2 + 6 = 15$. Racket has the following predefined functions for $\mathtt{a}, \mathtt{b} \in \mathbb{N}$:

   ```
   (quotient a b)      ; returns the quotient of dividing a by b
   (remainder a b)     ; returns the remainder of dividing a by b
   ```

3. (LW4 from labwork 2) Define `(flatten sl)` which takes as input a nested list of symbols, and returns the list of symbols contained in `sl` in the order in which they occur when `sl` is printed. Intuitively, `flatten` removes all the inner parentheses form its argument. For example:

   ```
   > (flatten '(a b c))
   '(a b c)
   > (flatten '((a b) c (((d)) e)))
   '(a b c d e)
   > (flatten '((a) () (b ()) () (c)))
   '(a b c)
   ```

   Note that nested lists of symbols are defined by the grammar

   `SL ::= null | (cons` s `SL) | (cons SL SL)`

   where s is a symbol.

Exercise set 2

1. Let `(rev2 lst1 lst2)` be the function which returns the result of joining the reverse of list `lst1` with list `lst2`. For example, `(rev2 '(3 4) '(2 1))` yields `'(4 3 2 1)`.

   Note that, if `lst1` is not empty, then `(rev2 lst1 lst2)` returns the same result as

   ```
   (rev2 (cdr lst1) (cons (car lst1) lst2))
   ```

   (a) Write a tail recursive definition of `rev2`.

   (b) Define the function `(revList lst)` which computes the reverse of list `lst` as a special case of using the function `rev2`.

2. Let `(f2 a b c)` the function which computes $\mathtt{c} \cdot \mathtt{a}^{\mathtt{b}}$ when $\mathtt{a}, \mathtt{b}, \mathtt{c}$ are non-negative integers. Note that

$$\mathtt{c} \cdot \mathtt{a}^{\mathtt{b}} = \begin{cases} \mathtt{c} & \text{if } \mathtt{b} = 0, \\ \mathtt{c} \cdot (\mathtt{a}^2)^{\mathtt{b}/2} & \text{if } \mathtt{b} > 0 \text{ is even}, \\ (\mathtt{c} \cdot \mathtt{a}) \cdot (\mathtt{a}^2)^{(\mathtt{b}-1)/2} & \text{if } \mathtt{b} \text{ is odd}. \end{cases}$$

   (a) Write a tail recursive definition of `f2`.

   (b) Define the function `(power a b)` which computes $\mathtt{a}^{\mathtt{b}}$ for $\mathtt{a}, \mathtt{b} \in \mathbb{N}$ as a special case of using the function `f2`.

3. Newton discovered the following method to compute $\sqrt{a}$ when a is a non-negative number: $\sqrt{a}$ is the limit of the sequence of numbers $(x_n)_{n\in\mathbb{N}}$ where

$$x_0 = 1.0, \quad x_{n+1} = (x_n + a/x_n)/2 \quad \text{for all } n \in \mathbb{N}.$$

   (a) Define the function `(improve xn a)` which takes as inputs the values of of x_n and a and returns the value of x_{n+1}.

   (b) x a *good enough* approximation of $\sqrt{a}$ if $|x^2 - a| \leq 0.000001$. Define the boolean function `(good? x a)` which returns `#t` if the value of `x` is a good enough approximation of $\sqrt{a}$, and `#f` otherwise.

   (c) Newton's method finds a good approximation of $\sqrt{a}$ starting from a number `x`, as follows: If `x` is a good approximation of $\sqrt{a}$ it returns `x`, otherwise it returns a good approximation od $\sqrt{a}$ starting from `(improve x a)`.

   Write a tail recursive definition of the function `(newton2 a x)` which uses Newton's method to compute a good approximation of $\sqrt{\mathtt{a}}$ starting from `x`.

4. Newton discovered the following method to compute $\sqrt[3]{a}$ when $a \in \mathbb{R}$: $\sqrt[3]{a}$ is the limit of the sequence of numbers $(x_n)_{n\in\mathbb{N}}$ where

$$x_0 = 1.0, \quad x_{n+1} = (2 \cdot x_n + a/x_n^2)/3 \quad \text{for all } n \in \mathbb{N}.$$

(a) Define the function `(improve3 xn a)` which takes as inputs the values of of x_n and a and returns the value of x_{n+1}.

(b) x a *good enough* approximation of $\sqrt[3]{a}$ if $|x^3 - a| \leq 0.000001$. Define the boolean function `(good3? x a)` which returns `#t` if the value of `x` is a good enough approximation of $\sqrt[3]{a}$, and `#f` otherwise.

(c) Newton's method finds a good approximation of $\sqrt{a}$ starting from a number `x`, as follows: If `x` is a good approximation of $\sqrt[3]{a}$ it returns `x`, otherwise it returns a good approximation of $\sqrt[3]{a}$ starting from `(improve3 x a)`.

Write a tail recursive definition of the function `(newton3 a x)` which uses Newton's method to compute a good approximation of $\sqrt[3]{\texttt{a}}$ starting from `x`.

5. Write a tail-recursive definition of the function `(eval v lst)` which takes as input a list of numbers

`lst = '(`a_0 a_1 ... a_n`)`

and computes the value of $a_0 + a_1 \cdot \texttt{v} + \ldots + a_n \cdot \texttt{v}^n$.

6. Consider lists of symbols defined by the grammar

`SL ::= null | (cons` *symbol* `SL) | (cons SL SL)`

Write a tail recursive definition of `(flattenTR sl)` which computes the reverse of the flattened form of a list of symbols `sl`.

7. A rational number is a number whose value coincides with $\frac{\texttt{a}}{\texttt{b}}$ where $\texttt{a}, \texttt{b} \in \mathbb{Z}$ and $\texttt{b} \neq 0$. Suppose we choose to represent every rational number $\frac{\texttt{a}}{\texttt{b}}$ as a pair `(cons a b)` where $\texttt{a}, \texttt{b} \in \mathbb{Z}$. Thus, we consider the following BNF for rational numbers

⟨rat⟩`::=(cons` ⟨integer⟩ ⟨integer⟩`)`

Define the following operations:

(a) `(rat? q)`, which recognizes if `q` $\in$ ⟨rat⟩.

(b) `(qsum q r)`, `(qdif q r)`, `(qmul q r)`, and `(qdiv q r)`, which take as inputs two values `q`, `r` $\in$ ⟨rat⟩ and compute their rational sum, difference, product, and division.

(c) `(rat-eq? q1 q2)`, which returns `#t` if `q` and `r` represent the same rational number, and `#f` otherwise.

(d) `(simplify q)`, which returns `r` $\in$ ⟨rat⟩ such that `q` and `r` represent the same rational number, and `r=(cons a b)` where `a`, `b` are relatiely prime. To define this function, you can use the predefined function `(gcd u v)` which returns the greatest common divisor of two integers `u`, `v` $\in \mathbb{Z}$.

8. A complex number is a number whose value coincides with $\texttt{a} + \texttt{i} \cdot \texttt{b}$ where $\texttt{a}, \texttt{b}$ are floating-point numbers and $\texttt{i} \in \mathbb{C}$ is the imaginary unit that satisfies the equation $\texttt{i}^2 = -1$. Suppose we choose to represent such a complex number as a pair `(cons a b)`. Thus, we consider the following BNF for complex numbers

   ⟨cplx⟩`::=(cons` ⟨real⟩ ⟨real⟩`)`

   Define the following operations:

   (a) `(cplx? c)`, which recognizes if $\texttt{c} \in \langle\text{cplx}\rangle$.

   (b) `(abs-value c)`, which computes the absolute value of $\texttt{c} \in \langle\text{cplx}\rangle$. Remember that the absolute value of a complex number $z = a + \texttt{i} \cdot b$ is $|z| = \sqrt{a^2 + b^2}$.

   (c) `(rdiv c r)` which takes as inputs $\texttt{c} \in \langle\text{cplx}\rangle$ and $\texttt{r} \in \langle\texttt{real}\rangle$, $\texttt{r} \neq \texttt{0}$, and returns the value from ⟨cplx⟩ corresponding to the division of `c` by `r`.

   (d) `(csum q r)`, `(cdif q r)`, `(cmul q r)`, and `(cdiv q r)`, which take as inputs two values $\texttt{q}, \texttt{r} \in \langle\text{cplx}\rangle$ and compute their complex sum, difference, product, and division.

9. Consider sets represented by lists without repeating elements, and let $\texttt{A}, \texttt{B}$ be two such sets. Define the following operations:

   (a) `(union A B)`, which computes the set union of `A` and `B`.

   (b) `(difference A B)`, which computes the set difference of `A` and `B`.

   You can use the predefined function `(member v lst)` which returns `#f` if `v` is not equal to any element of list `lst`, and true otherwise.

5.4 Labworks 4 (Racket) – Homework

1. Define `foldr` with `foldl` and `reverse`, and indicate the runtime complexity of this definition.

2. Define `filter` with `foldr`.

3. Define `length` with `foldl`.

4. Define the following higher-order functions:

   (a) `(nest` f n`)` which takes as input a function $f : A \to A$ and $n \in \mathbb{N}$, and returns the function that maps $x \in A$ to the value of $\underbrace{f(\ldots f}_{n \text{ times}}(x)\cdots)$.

   If $n = 0$ then `(nest f 0)` should return the identity function `(lambda (x) x)`.

   (b) `(nestwhile` f v p`)` which takes as inputs a function $f : A \to A$, a predicate $p : A \to \mathit{bool}$ and a value $v \in A$, and returns the value $w = f^n(v)$ for the smallest $n \in \mathbb{N}$ such that `(`p w`)` is `#f`.

5. Use `foldr` to define the variadic function

   `(comp` f_1 ... f_n`)`

   which takes as inputs $n \geq 0$ unary functions $f_1, \ldots, f_n$ and returns the function that maps x to the value of

   `(`f_1 `...` `(`f_n x`)...)`

6. Define the function `(list->set lst)`, which drops the duplicate occurrences of elements from a list `lst`.

   Suggestion: express the computation of `(list->set lst)` as

   `(foldr` f `null lst)`

   with a suitable function f. You can use the built-in function `(member e l)` which is true if `e` is an element of list `l` and `#f` otherwise.

7. Consider the problem of counting the number of occurrences of every word in a document `d`. More precisely, let `d` be a list of symbols (the words of document `d`). We wish to define `(count-words d)` which returns the list of pairs `(cons` w n`)` where w is a string in `d`, and n is the number of occurrences of w in `d`. For example

```
> (count-words '(a b a b b c x z z x))
'((a . 2) (b . 3) (c . 1) (z . 2) (x . 2))
```

8. Give recursive definitions to the following variadic functions:

   (a) `(fmap-2` a b f_1 ... f_n`)` which computes `(list` w_1 ... w_n`)` where w_i is the value of `(`f_i a b`)` for every $1 \leq i \leq n$. For example:

```
> (fmap-2 4 2 + * /)    > (fmap-2 4 2)
'(6 8 2)                '()
```

   (b) `(inc?` a_1 ... a_n`)` which takes as inputs $n \geq 0$ integers and returns `#t` if and only if $a_1 < a_2 < \ldots < a_n$. For example:

```
> (inc? 1 4 3)  > (inc?)  > (inc? 1)   > (inc? 4 7 8 9)
#f              #t        #t            #t
```

   (c) `(dec?` a_1 ... a_n`)` which takes as inputs $n \geq 0$ integers and returns `#t` if and only if $a_1 > a_2 > \ldots > a_n$. For example:

```
> (inc? 1 4 3)  > (dec?)  > (dec? 1)   > (dec? 9 7 5 0)
#f              #t        #t           #t
```

   (d) Find the common pattern of computation of `inc?` and `dec?` and define

   `(sorted?` *cmp* a_1 ... a_n`)`

   which returns `#t` if and only if `(`*cmp* a_i a_{i+1}`)` is true for all $1 \leq i < n$.

(e) `(monotone?` a_1 ... a_n`)` which takes as inputs $n \geq 0$ integers and returns `#t` iff $a_1 < a_2 < \ldots < a_n$ or $a_1 > a_2 > \ldots > a_n$.

9. Define `(f-inc` n`)` which takes an input $n \in \mathbb{N}$ and computes the list of functions `(list` f_1 ... f_n`)` where, for all $k \geq 1$, `(`f_k x`)` returns the value of $x + k$.

5.5 Labworks 5 (Haskell) – Homework

1. Define recursively the function `map2::(a->b->c)->[a]->[b]->[c]`

   such that, if

   - `f` is a binary function that takes inputs of types `a` and `b`, and computes result of type `c`,
   - `lst1=`$[a_1, \ldots, a_n]$ is a list of n elements of type `a`,
   - `lst2=`$[b_1, \ldots, b_n]$ is a list of n elements of type `b`

   then `(map2 f lst1 lst2)` computes the list $[c_1, \ldots, c_n]$ where c_i is the value of `(f` a_i b_i`)` for all $1 \leq i \leq n$. For example:

```
> map2 (+) [1,2,3] [4,5,6]       > map2 (*) [1,2,3] [4,5,6]
[5,7,9]                          [4.10,18]
```

2. Use `map` and `addLists` to define the infinite list of `Integer`s

   `yList =` $[y_1, y_2, y_3, \ldots]$

   where $y_1 = y_2 = 1, y_3 = 2$ and $y_n = y_{n-1}^2 - 2 \cdot y_{n-2}^2 + 3 \cdot y_{n-3}$ for all $n > 3$.

3. Define the function `nestList::(Double->Double)->Double->[Double]` such that

   `nestList f v`

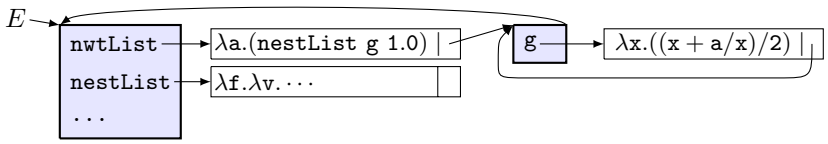
   computes the infinite list

   `[v, f v, f (f v),f (f (f v)),...]`

4. Consider the function `g` defined by

```
nwtList a = let g x = (x+a/x)/2 -- same as g=\x.((x+a/x)/2)
            in nestList g 1.0
```

   This definition extends the top frame of the evaluation environment E with the binding for `nwtList`, as shown below:



(a) What is the value of `nwtList 5.0`? Does the computation terminate?

(b) What is the value of

```
head (drop 5 (nwtList 7.0))
```

Does the termination terminate?

Suggestion: remember Newton's method to compute $\sqrt{x}$ when x is a positive real number.

5. Consider the function definition

```
triples::Int->[(Int,Int,Int)]
triples n = [(a,b,n-a-b)|a<-[1..n-2],b<-[1..n-1-a]]
```

What is the value of the function call `(triples n)` when `n` > 2?

Suggestion: use `ghci` to compute the values of `(triples n)` for some small values `n` > 2, to see what you get.

6. Define recursively the function `triplesFrom::Int->[(Int,Int,Int)]` such that, if `n` > 0, then `(triplesFrom n)` returns the list of all triples (a, b, c) with $a > 0, b > 0, c > 0$ and $a + b + c \geq n$.

7. Define the list `t3List` of all triples (a, b, c) of type `(Int,Int,Int)` with $a > 0, b > 0, c > 0$.

8. A Pythagorean triple is a triple (a, b, c) of strict positive integers such that $a^2 + b^2 = c^2$. Define the function `(pythTriples n)` which returns the first `n` Pythagorean triples from `t3List`.

5.6 Miniproject: Working with power series

Power series have many applications in sciences and engineering. We consider power series of the form $\sum_{n=0}^{\infty} a_n\, x^n$ with $a_n \in \mathbb{R}$ for all $n \geq 0$, and represent them by infinite lists of `Double`s. For example

1) $\displaystyle\sum_{n=0}^{\infty} x^n$ is represented by the list comprehension `[1|i<-[0..]]`

2) $e^x = \displaystyle\sum_{n=0}^{\infty} \frac{x^n}{n!}$ can be represented by

```
eRepr = map (\x->1/fromInteger x) (coeffList [1..] 1 (*))
```

where

```
coeffList::[Integer]->a->(Integer->a->a)->[a]
-- coeffList [1..] x f computes the list [c_0, c_1, c_2, ...] where
-- c_0 = x and c_n = (f n c_{n-1}) for all n > 0.
coeffList (n:ns) x f = x:coeffList ns (f n x) f
```

3) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2\,n+1}}{(2\,n+1)!}$ can be represented by

```
sinRepr = map c plist where
 c (x,y) = x/fromInteger y
 f n (_,y)
   | mod n 2 == 0 = (0,y*n)
   | mod n 4 == 1 = (1,y*n)
   | otherwise    = (-1,y*n)
 plist = coeffList [1..] (0,1) f
```

Note that `(coeffList [1..] (0,1) f)` generates on demand the infinite list of tuples

```
[(0,1),(1,1),(0,2!),(-1,3!),(0,4!),(1,5!),(0,6!),(-1,7!),...]
```

and `map c plist` generates on demand the list representation of $\sin x$. Another, less efficient, lazy implementation of the representation of e^x is the list comprehension

```
sinRepr = [(coeff n) | n<-[0..]] where
 coeff n = if (mod n 2) == 0
    then 0
    else ((-1)^(div (n+1) 2))/fromInteger (foldr (*) 1 [1..n])
```

4) A polynomial $\sum_{k=0}^{n} a_k\, x^k$ is a power series too: $\sum_{k=0}^{\infty} a_k\, x^k$ where $a_i = 0$ for all $i > n$. A lazy implementation of the representation of the power series of the polynomial $\sum_{k=0}^{n} a_k\, x^k$ is

```
[a_0, a_1 ..., a_n] ++ [0|<-i<-[1..]]
```

Power series can be added, multiplied, scaled and divided as follows: if $r \in \mathbb{R}$, $a(x) = \sum_{n=0}^{\infty} a_n\, x^n$ and $b(x) = \sum_{n=0}^{\infty} b_n\, x_n$ are power series, then

$$
\begin{aligned}
a(x) + b(x) &= \sum_{n=0}^{\infty} (a_n + b_n)\, x^n \\
r \cdot a(x) &= \sum_{n=0}^{\infty} (r \cdot a_n)\, x^n \\
x \cdot a(x) &= \sum_{n=0}^{\infty} a_n\, x^{n+1} = \sum_{n=0}^{\infty} c_n\, x^n \text{ where } c_n = \begin{cases} 0 & \text{if } n = 0, \\ a_{n-1} & \text{if } n > 0. \end{cases} \\
a(x) \cdot b(x) &= \left(a_0 + x \cdot \sum_{n=0}^{\infty} a_{n+1}\, x^n \right) \cdot b(x) = a_0 \cdot b(x) + x \cdot \left(b(x) \cdot \sum_{n=0}^{\infty} a_{n+1} x^n \right)
\end{aligned}
$$

Also, if $b_0 \neq 0$ then

$$
\begin{aligned}
\frac{a(x)}{b(x)} &= \frac{a_0}{b_0} + x \cdot \frac{\sum_{n=1}^{\infty} (a_n - \frac{a_0}{b_0} \cdot b_n)\, x^{n-1}}{b(x)} \\
&= \frac{a_0}{b_0} + x \cdot \frac{\sum_{n=0}^{\infty} a_{n+1}\, x^n + (-\frac{a_0}{b_0} \cdot \sum_{n=0}^{\infty} b_{n+1}\, x^n)}{b(x)}
\end{aligned}
$$

Suppose `ra`, `rb` are representations of the power series $a(x) = \sum_{n=0}^{\infty} a_n\, x^n$ and $b(x) = \sum_{n=0}^{\infty} b_n\, x^n$. Define lazy implementations of the following functions:

```
eval::[Double] -> Double -> Int -> Double
sumS,prodS,divideS::[Double]->[Double]->[Double]
xprodS::[Double]->[Double]
rprodS::Double->[Double]->[Double]
```

such that

- `(eval ra v m)` computes $\sum_{n=0}^{\mathtt{m}} a_n\, \mathtt{v}^{\mathtt{n}}$
- `(sumS ra rb)` computes the representation of $a(x) + b(x)$
- `(rprodS r ra)` computes the representation of $r \cdot a(x)$ where r is the value of `r`
- `(xprodS a)` computes the representation of $x \cdot a(x)$
- `(prodS ra rb)` computes the representation of $a(x) \cdot b(x)$
- if $b_0 \neq 0$ then `(divS ra rb)` computes the representation of $\dfrac{a(x)}{b(x)}$

Note that

- The list a`:[0|i<=1..]]` represents the constant polynomial a.

- If `ra` represents the power series $\sum_{n=0}^{\infty} a_n\, x^n$ then

  `(0:ra)` represents the power series $\sum_{n=0}^{\infty} a_n\, x^{n+1}$

  `(tail ra)` represents the power series $\sum_{n=0}^{\infty} a_{n+1}\, x^n$

- You can make use of the function `map2` and of predefined functions `foldl`, `foldr`, `map`.

To test if your implementation is correct, try this:

```
> :{
ar,br,cr::[Double]
ar = [1,2,3]++[0|i<-[1..]]  -- 1 + 2 x + 3 x^2
br = 1:[0|i<-[1..]]         -- constant polynomial 1
cr = [1,-1]++[0|i<-[1..]]   -- 1 - x
:}
> take 4 (prodS ar cr)      -- (1+2x+3x^2)·(1-x) = 1+x+x^2-3x^3
[1.0,1.0,1.0,-3.0]
> take 6 (divideS br cr)    -- 1/(1-x) = 1 + x + x^2 + x^3 + ...
[1.0,1.0,1.0,1.0,1.0,1.0]
> eval sinRepr (pi/4) 16    -- compute an approximation of sin (pi/4)
0.7071067811865474
> eval eRepr 1 16           -- compute an approximation of e^1 = e
2.718281828459042
```

5.7 Labworks 7 (Racket) – Recommended

Bottom-up problem solving

B1. Define by recursion on `m` the function `(range m n)` which takes as inputs two integers `m`, `n` and returns:

- ▷ The list of integers from `m` to `n`, if $\texttt{m} \leq \texttt{n}$,
- ▷ The empty list, if $\texttt{m} > \texttt{n}$.

For example:

```
> (range 2 2)     > (range 3 2)     > (range 5 8)
'(2)              '()               '(5 6 7 8)
```

B2. For $a, b \in \mathbb{N}$ we write $a|b$ to indicate that b is divisible with a, and $a \nmid b$ to indicate that b is not divisible with a. Use `range`, `map`, and `filter` to compute the lists of elements of the following sets:

(a) $\{(n+1)(n+2)/2 \mid n \in \mathbb{N}, 1 \le n \le 100 \text{ and } 11|n\}$.

(b) $\{2 \cdot n \mid n \in \mathbb{N}, 1 \le n \le 20,\ 2|n, 3 \nmid n\}$.

(c) $\{n \mid n \in \mathbb{N}, 1 \le n \le 50,\ n \text{ is divisible with 5 or with 13}\}$.

B3. Use `map` to define the function `(cprod1 a lst)` which takes as inputs a value `a` and a list `lst`, and returns the list of all pairs `(cons a b)` where `b` is an element of `lst`. For example:

```
> (cprod1 'x '(a b c))
'((x . a) (x . b) (x . c))
```

B4. Use `map`, `apply`, `append`, and `cprod1` to define `(cprod lst1 lst2)` which returns the cartesian product of lists `lst1` and `lst2`, which is the list of all pairs `(cons a b)` with `a` from `lst1` and `b` from `lst2`. For example:

```
> (cprod '(x y) '(a b c))
'((x . a) (x . b) (x . c) (y . a) (y . b) (y . c))
```

B5. Let `lst` be a list of pairs of the form `(cons x y)`. Define the following operations on such lists of pairs:

(a) `(filter2 p lst)`, which returns the list of pairs `(cons x y)` from `lst` for which `(p x y)` is true.

(b) `(map2 f lst)`, which returns the list of values of `(f x y)` for the pairs `(cons x y)` from `lst`.

B6. Use the previously defined functions `cprod`, `map2` and `filter2` to compute the lists of elements of the following sets of pairs:

(a) $\{$`(cons a b)` $\mid$ `a`, `b` $\in \{1, 2, 3\}$ and `a` $+$ `b` is even$\}$.

(b) $\{$`(cons a b)` $\mid$ `a`, `b` $\in \{1, 2, 3, 4\}$, `a` $+$ `b` is odd and $2|$`b`$\}$.

(c) $\{$`(cons a b)` $\mid$ `a`, `b` $\in \{1, 2, 3, 4\}$, `a` is odd or `b` is odd$\}$.

6 Answers to some exercises

6.1 Answers to labworks 1

HW1.
```
(define (good-list? l)
  (or (null? l)
      (and (list? l)
           (> (length l) 1)
           (symbol? (car l))
           (number? (cdr l))
           (good-list? (cddr l)))))
```

HW2.
```
(define (symb-value l s)
  (cons [(eq? s (car l)) (cadr l)]
        [#t (symb-value (cddr l) s)]))
```

HW3.
```
(define (mem l v)
   (cond [(null? l) #f]
         [(equal? v (car l)) #t]
         [#t (mem (cdr l) v)]))
```

or

```
(define (mem l v)
  (and (list? l)
       (> (length l) 0)
       (or (equal? v (car l)) (mem (cdr l) v))))
```

HW4.
```
(define (add l)
  (if (null? l) 0 (+ (car l) (add (cdr l)))))
```

or

```
(define (add l) (foldr + 0 l))
```

HW5.
```
(define (mult l)
  (if (null? l) 1 (* (car l) (mult (cdr l)))))
```

or

```
(define (mult l) (foldr * 1 l))
```

HW6. Note that nested lists of numbers are defined by the grammar

nlist ::= null | (cons *number nlist*) | (cons *nlist nlist*)

```
(define (nlist? l)
  (or (null? l) (and (list? l)
                     (or (number? (car l))
                         (nlist? (car l)))
                     (nlist? (cdr l)))))
```

HW7.
```
(define (add-all l)
  (cond [(null? l) 0]
      [(number? (car l)) (+ (car l) (add-all (cdr l)))]
      [#t  (+ (add-all (car l)) (add-all (cdr l)))]))

; This implementation works well only if
; l contains at least one positive number
(define (max-elem l)
    (cond [(null? l) 0]
          [(number? (car l)) (max (car l) (max-elem (cdr l)))]
          [#t (max (max-elem (car l)) (max-elem (cdr l)))]))

(define (max-depth l)
 (cond [(null? l) 1]
       [(number? (car l)) (max-depth (cdr l))]
       [#t (max (+ 1 (max-depth (car l))
                (max-depth (cdr l))))]))
```

6.2 Answers to recommended labworks 2

LW1.
```
(define (preorder bt)
  (cond [(number? bt) list bt)]
        [#t
         (let ([n (car bt)] [bt1 (cadr bt)] [bt2 (caddr bt)])
           (cons n (append (preorder bt1)
                           (preorder bt2))))]))

(define (inorder bt)
  (cond [(number? bt) (list bt)]
        [#t
         (let ([n (car bt)] [bt1 (cadr bt)] [bt2 (caddr bt)])
           (append (inorder bt1)
                   (list n)
                   (inorder bt2)))]))

(define (postorder bt)
  (cond [(number? bt) (list bt)]
        [#t
         (let ([n (car bt)] [bt1 (cadr bt)] [bt2 (caddr bt)])
           (append (postorder bt1)
                   (postorder bt2)
                   (list n)))]))
```

LW2. (a)
```
nlist ::= null
        | (cons number nlist)
        | (cons nlist nlist)
```

(b)
```
(define (nlist? l)
     (or (null? l) (and (list? l)
                        (or (number? (car l))
                            (nlist? (car l)))
                        (nlist? (cdr l)))))
```

LW5. This function is easy to define by structural induction on `sl`:

```
(define (swapper s1 s2 sl)
  (if (null? sl)
      null
      (let ([sl1 (swapper s1 s2 (cdr sl))])
        (cond [(eq? s1 (car sl)) (cons s2 sl1)]
              [(eq? s2 (car sl)) (cons s1 sl1)]
              [#t (cons (car sl) sl1)]))))
```

6.3 Answers to labworks 3

[Download from here]

Exercise set 1

1.
```
(define (joinLists lst1 lst2)
  (cond [(null? lst1) lst2]
        [#t (cons (car lst1)
                  (joinLists (cdr lst1) lst2))]))
```

2.
```
(define (is-digit? N)
  (and (integer? N) (>= N 0) (<= N 9)))
(define (drop-last-digit N) (quotient N 10))
(define (last-digit N) (remainder N 10))

(define (digit-sum N) ; recursive but not tail recursive
  (if (is-digit? N)
     N
     (+ (last-digit N)
        (digit-sum (drop-last-digit N)))))
```

   It is possible to define this function by tail recursion too:

```
; A tail recursive definition
(define (digit-sum-tr N [result 0])
  (if (is-digit? N)
     (+ N r)
     (digit-sum-tr (drop-last-digit N)
                   (+ result (last-digit N)))))
```

3. A nested list of symbols is $sl \in NSL$ where NSL is defined by the grammar

NSL `::= null | (cons` *symbol NSL*`) | (cons` *NSL NSL*`)`

We can use this grammar to define `flatten` by structural induction:

```
(define (flatten sl)
   (cond [(null? sl) null]
       [(symbol? (car sl)) (cons (car sl)
                                 (flatten (cdr sl)))]
       [#t (append (flatten (car sl)))])
                   (flatten (cdr sl)))]))
```

Exercise set 2

1. ```
   (define (rev2 lst1 lst2)
     (cond [(null? lst1) lst2]
           [#t (rev2 (cdr lst1) (cons (car lst1) lst2))]))
   (define (revList lst) (rev2 lst null))
   ```

2. ```
   (define (f2 a b c)
     (cond [(= b 0) c]
           [(= 0 (remainder b 2)) (f2 (* a a) (/ b 2) c)]
           [#t (f2 (* a a) (/ (- b 1) 2) (* a c))]))
   (define (power a b) (f2 a b 1))
   ```

3. ```
   (define (improve xn a) (/ (+ xn (/ a xn)) 2))
   (define (good? x a) (< (abs (- (* x x) a)) 0.000001))

   (define (newton2 a [x 1.0])
     (if (good? x a)
         x
         (newton2 a (improve x a))))
   ```

4. ```
   (define (improve3 xn a) (/ (+ (* 2 xn) (/ a (* xn xn))) 3))
   (define (good3? x a) (< (abs (- (expt x 3) a)) 0.000001))

   (define (newton3 a [x 1.0])
     (if (good3? x a)
         x
         (newton3 a (improve3 x a))))
   ```

5. This computation is a special case of `(eval lst v s p)` which takes the extra arguments $\mathtt{s}, \mathtt{p} \in \mathbb{R}$ and computes

$$\mathtt{s} + \mathtt{p} \cdot (a_0 + a_1 \cdot \mathtt{v} + \ldots + a_n \cdot \mathtt{v}^n) = \mathtt{s} + \mathtt{p} \cdot \left(\sum_{i=0}^{n} a_i \cdot \mathtt{v}^i \right)$$

A tail recursive definition of this more general operation is easy to derive from the observation that

$$\mathtt{s}+\mathtt{p}\cdot\left(\sum_{i=0}^{n} a_i\cdot \mathtt{v}^i\right) = \begin{cases} s & \text{if } \texttt{lst} \text{ is empty,} \\ (\mathtt{s}+\mathtt{p}\cdot a_0)+(\mathtt{p}\cdot\mathtt{v})\cdot\left(\sum_{i=0}^{n-1} a_{i+1}\cdot \mathtt{v}^i\right) & \text{otherwise} \end{cases}$$

```
(define (eval lst v
              [s 0] [p 1]) ; optional arguments with default values 0 and 1
  (cond [(null? lst) s]
        [#t (eval (cdr lst) v (+ s (* p (car lst))) (* p v))]))
```

Runtime: $O(n)$ where n is the length of `lst`. Space complexity: $O(1)$.

A much simpler implementation is with `foldr`:

```
(define (eval v lst) (foldr (lambda (a b) (+ a (* v b))) 0 lst))
```

6. `SL ::= null | (cons` *symbol* `SL) | (cons SL SL)`

We saw on page 16 that the flattened form of `lst` can be computed with

```
(define (flatten lst)
  (cond [(null? lst) null]
        [(symbol? (car lst)) (cons (car lst)
                                   (flatten (cdr lst)))]
        [(null? (car lst)) (flatten (cdr lst))]
        [#t (flatten (cons (caar lst)
                           (cons (cdar lst) (cdr lst))))]))
```

The last two recursive calls are tail recursive, but the first one is not. This definition can be transformed easily into a tail recursive definition if we extend it with an accumulator:

```
(define (flattenTR lst [A null])
  (cond [(null? lst) (reverse A)]
        [(symbol? (car lst)) (flattenTR (cdr lst)
                                        (cons (car lst) A))]
        [(null? (car lst)) (flattenTR (cdr lst) A)]
        [#t (flattenTR (cons (caar lst)
                             (cons (cdar lst) (cdr lst)))
                       A)]))
```

The additional argument `A` is an accumulator: it accumulates the symbols in the reverse order of their occurrence in `lst`. Remember that the operation `(reverse A)` is predefined and returns the reverse of list `A`.

7.
```
(define (rat? r)
  (and (list? r)
       (= (length r) 2)
       (integer? (car r))
       (integer? (cadr r))
       (not (= (cadr r) 0))))

(define (simplify r)
  (let* ([a (car r)] [b (cadr r)] [d (gcd a b)])
    (list (/ a d) (/ b d))))

(define (qsum q r)
  (let* ([a1 (car q)]
         [b1 (cadr q)]
         [a2 (car r)]
         [b2 (cadr r)]
         [num (+ (* a1 b2) (* a2 b1))]
         [denum (* b1 b2)])
    (simplify (list num denum))))

define (qdif q r)
  (let* ([a1 (car q)]
         [b1 (cadr q)]
         [a2 (car r)]
         [b2 (cadr r)]
         [num (- (* a1 b2) (* a2 b1))]
         [denum (* b1 b2)])
    (simplify (list num denum))))

(define (qmul q r)
  (let* ([a1 (car q)]
         [b1 (cadr q)]
         [a2 (car r)]
         [b2 (cadr r)]
         [num (* a1 a2)]
         [denum (* b1 b2)])
    (simplify (list num denum))))

(define (qdiv q r)
  (let* ([a1 (car q)]
         [b1 (cadr q)]
         [a2 (car r)]
         [b2 (cadr r)]
         [num (* a1 b2)]
         [denum (* b1 a2)])
    (simplify (list num denum))))
```

```
(define (rat-eq? q1 q2)
  (let ([a1 (car q1)] [b1 (cadr q1)]
        [a2 (car q2)] [b2 (cadr q2)])
      (= (* a1 b2) (* a2 b1))))
```

8.
```
(define (cplx? c)
  (and (list? c)
       (= (length c) 2)
       (real? (car c))
       (real? (cadr c))))

(define (abs-value c)
  (let ([a (car c)]
        [b (cadr c)])
    (sqrt (+ (* a a) (* b b)))))

(define (rdiv c r)
  (list (/ (car c) r) (/ (cadr c) r)))

(define (csum q r)
  (list (+ (car q) (car r)) (+ (cadr q) (cadr r))))

(define (cdiff q r)
  (list (- (car q) (car r)) (- (cadr q) (cadr r))))

(define (cmul q r)
  (let ([a (car q)]
        [b (cadr q)]
        [c (car r)]
        [d (cadr r)])
    (list (- (* a c) (* b d)) (+ (* a d) (* b c)))))

(define (cdiv q r)
  (let ([c (car r)] [d (cadr r)])
    (rdiv (cmul q (list c (- d))) (+ (* c c) (* d d)))))
```

9.
```
(define (union A B)
  (cond [(null? A) B]
        [(member (car A) B) (union (cdr A) B)]
        [#t (cons (car A) (union (cdr A) B))]))

(define (difference A B)
  (cond [(null? A) null]
        [(member (car A) B) (difference (cdr A) B)]
        [#t (cons (car A) (difference (cdr A) B))]))
```

Another possibility is to define these functions with `foldr`:

```
(define (union A B)
  (foldr
   (lambda (v lst) (if (member v lst) lst (cons v lst)))
   B A))
(define (difference A B)
  (foldr
   (lambda (v lst) (if (member v B) lst (cons v lst)))
   null A))
```

6.4 Answers to labworks 4

1. `(define (foldr f v lst) (foldl f v (reverse lst)))`

   Runtime complexity: $O(n)$ where n is the length of `lst`.

2. ```
   (define (filter p lst)
     (foldr (lambda (v l) (if (p v) (cons v l) l)) null lst))
   ```

3. `(define (length lst) (foldl (lambda (v l) (+ l 1)) 0 lst))`

4. ```
   (define (nest f n)
     (if (= n 0)
         (lambda (x) x)
         (lambda (x) ((nest f (- n 1)) (f x)))))

   (define (nestwhile f v p)
     (if (p v) v (nestwhile f (f v) p)))
   ```

5. ```
   (define (comp . flst)
     (foldr (lambda (f g) (lambda (x) (f (g x))))
            (lambda (x) x)
            flst))
   ```

6. ```
   (define (list->set lst)
     (foldr (lambda (v l) (if (member v l) l (cons v l)))
            null
            lst))
   ```

7. ```
   (define (count-words lst)
     (define (count-word w)
       (length (filter (lambda (x) (eq? w x)) lst)))
     (map (lambda (w) (cons w (count-word w)))
          (list->set lst)))
   ```

   Note that `count-word` is an auxiliary function which is local to the body of function `count-words`; it is used to count the number of occurrences of a word `w` in list `lst`.
   ```

8.
```
(define (fmap-2 a b . flst)
  (map (lambda (f) (f a b)) flst)

(define (inc? . lst)
  (or (< (length lst) 2)
      (and (< (car lst) (cadr lst)) (inc? (cdr lst)))))

(define (dec? . lst)
  (or (< (length lst) 2)
      (and (> (car lst) (cadr lst)) (dec? (cdr lst)))))

(define (sorted? cmp . lst)
  (sAux cmp lst))

(define (sAux cmp lst)
  (or (< (length lst) 2)
      (and (cmp (car lst) (cadr lst))
           (sAux cmp (cdr lst)))))

(define (monotone? . lst) (or (sAux < lst) (sAux > lst)))
```

9. Note that `(f-inc` n`)` is the list `(list` f_1 $\ldots$ f_n`)` where f_k = `(nest inc` k`)` for all $1 \leq k \leq n$. Based on this observation, we define

```
(define (f-inc n)
  (define (inc x) (+ x 1))
  (define (listTo n)
    (if (= n 0) null (cons 1 (map inc (listTo (- n 1))))))
  (map (lambda (k) (nest inc k)) (listTo n)))
```

The auxiliary function `(listTo n)` is defined to return the list of numbers from 1 to `n`.

6.5 Answers to labworks 5

[Download from here]

Preliminary exercises

```
intsFrom x = x:intsFrom (x+1)
sieve1,sieveAll:[Integer]->[Integer]
sieve1 (x:xs) = x:filter (\y->(mod y x) > 0) xs
sieveAll (x:xs) = x:sieveAll (filter (\y->(mod y x) > 0) xs)
-- the list of all integers, starting from 1
nats = intsFrom 1
-- the list of all prime numbers
primes = sieveAll (intsFrom 2)
```

1. ```
   map2::(a->b->c)->[a]->[b]->[c]
   map2 _ [] [] = []
   map2 f (x:xs) (y:ys) = (f x y):map2 f xs ys
   ```

   `(addLists lst1 lst2)` performs the componentwise addition of two lists of numbers. The lists must have same length, or both can be infinite.

   ```
   addLists [] [] = []
   addLists (x:xs) (y:ys) = (x+y):addLists xs ys
   ```

   This function can also be defined with `map2`:

   ```
   addLists lst1 lst2 = map2 (+) lst1 lst2
   ```

2. ```
   yList =
      let
         lst1 = map (\y->y*y) (tail (tail yList))
         lst2 = map (\y->(-2)*y^2) (tail yList)
         lst3 = map (*3) yList
      in 1:1:2:addLists lst1 (addLists lst2 lst3)

   > take 6 yList
   [1,1,2,5,20,356]
   ```

3. ```
   nestList:: (a->a)->a->[a]
   nestList f v = v:(nestList f (f v))
   ```

4. ```
   nwtList a = let f x = (x+a/x)/2 in (nestList f 1.0)
   ```

   (a) `(nwtList 5.0)` is the infinite list of successive approximations of $\sqrt{5.0}$, computed by Newton's method, starting with the initial approximation `1.0`. This computation does not terminate.

   (b) ```
   > head (drop 5 (nwtList 7.0))
   2.6457513111113693
   ```
   `(drop 5 (nwtList 7.0))` drops the first five approximations of $\sqrt{7.0}$ and `(head (drop 5 (nwtList 7.0)))` returns the sixth approximation. If we compare this approximation with the the floating point value of $\sqrt{7.0}$ returned by Haskell:

   ```
   > sqrt 7.0
   2.6457513110645907
   ```

   we notice that they coincide up to nine decimals.

5. `(triples n)` for $\mathtt{n} > 2$ returns the list of all triples $(a, b, c) \in \mathbb{N}^3$ with $a > 0, b > 0, c > 0$ and $a + b + c = \mathtt{n}$.

6. ```
   triplesFrom n = (triples n) ++ triplesFrom (n+1)
   ```

7. -- the list of all triples (a,b,c) of type `(Int,Int,Int)`
   -- with $a>0, b>0, c>0$
   `allTriples = triplesFrom 3`

8. ```
   pythTriples n =
     take n (filter (\(a,b,c)->(a^2+b^2==c^2)) allTriples)

   > pythTriples 5
   [(3,4,5),(4,3,5),(6,8,10),(8,6,10),(5,12,13)]
   ```

Answer to miniproject

[Download from here]

```
module PowerSeries where

eval::[Double] -> Double -> Int -> Double
eval ra v m = foldr (\a r->a+r*v) 0 (take (m+1) ra)

rprodS::Double->[Double]->[Double]
rprodS r as = map (r*) as

xprodS::[Double]->[Double]
xprodS ra = (0:ra)

sumS,prodS,divideS::[Double]->[Double]->[Double]
sumS (a:as) (b:bs) = (a+b):sumS as bs
prodS (a:as) bs = sumS (rprodS a bs) (xprodS (prodS bs as))
divideS (a:as) (b:bs) = (a/b):divideS (rprodS (-a/b) bs) (b:bs)

-- (coeffList [1..] c0 f) computes the list [c0,c1,c2,...] where
-- cn = (f n c(n-1)) for all n > 0.
coeffList::[Integer]->a->(Integer->a->a)->[a]
coeffList (n:ns) c f = c:coeffList ns (f n c) f
-- sinRepr = [1,0.-1/3!,0,1/5!,0,-1/7!,0,...] because sin(x) = sum_{n=0}^{inf} ((-1)^n * x^(2n+1)) / (2n+1)!)
sinRepr = map c plist where
 c (x,y) = x/fromInteger y
 f n (_,y)
   | mod n 2 == 0 = (0,y*n)
   | mod n 4 == 1 = (1,y*n)
   | otherwise    = (-1,y*n)
 plist = coeffList [1..] (0,1) f
 -- eRepr = [1/0!, 1/1!, 1/2!, 1/3!,...] because e^x = sum_{n=0}^{inf} x^n/n!
eRepr = map (\x->1/fromInteger x) (coeffList [1..] 1 (*))
```
```

6.6 Labworks 7: Bottom-up problem solving

[Download from here]

B1.
```
(define (range m n)
  (if (> m n) null (cons m (range (+ m 1) n))))
```

This recursive definition is not tail recursive, but a tail recursive definition is easy to find:

```
(define (rangeTR m n [result null])
  (if (> m n) result (rangeTR m (- n 1) (cons n result))))
```

B2.
```
; (a) {(n+1)(n+2)/2 | n ∈ ℕ, 1 ≤ n ≤ 100 and 11|n}.
(map
  (lambda (n) (/ (* (+ n 1) (+ n 2)) 2))
  (filter (lambda (n) (= (remainder n 11) 0)) (range 1 100)))

; (b) {(2 · n | n ∈ ℕ, 1 ≤ n ≤ 20, 2 | n, 3 ∤ n}.
(map (lambda (n) (* 2 n))
  (filter (lambda (n) (and (even? n) (> (remainder n 3) 0)))
        (range 1 20)))

; (c) {(n | n ∈ ℕ, 1 ≤ n ≤ 50, 5 | n or 13 | n}.
(filter (lambda (n) (or (= (remainder n 5) 0)
                        (= (remainder n 13) 0)))
        (range 1 50))
```

B3.
```
(define (cprod1 a lst) (map (lambda (x) (cons a x)) lst))
```

B4.
```
(define (cprod lst1 lst2)
  (apply append (map (lambda (x) (cprod1 x lst2)) lst1)))
```

B5.
```
(define (filter2 p lst)
  (filter (lambda (xy) (p (car xy) (cdr xy))) lst))

(define (map2 f lst)
  (map (lambda (xy) (f (car xy) (cdr xy))) lst))
```

B6.
```
; (a)
(let* ([A (range 1 3)]
       [A2 (cprod A A)])
  (filter2 (lambda (a b) (even? (+ a b))) A2))

; (b)
(let* ([A (range 1 4)]
       [A2 (cprod A A)])
  (filter2 (lambda (a b) (and (odd? (+ a b)) (even? b))) A2))
```

```
; (c)
(let* ([A (range 1 4)]
       [A2 (cprod A A)])
  (filter2 (lambda (a b) (or (odd? a) (odd? b))) A2))
```