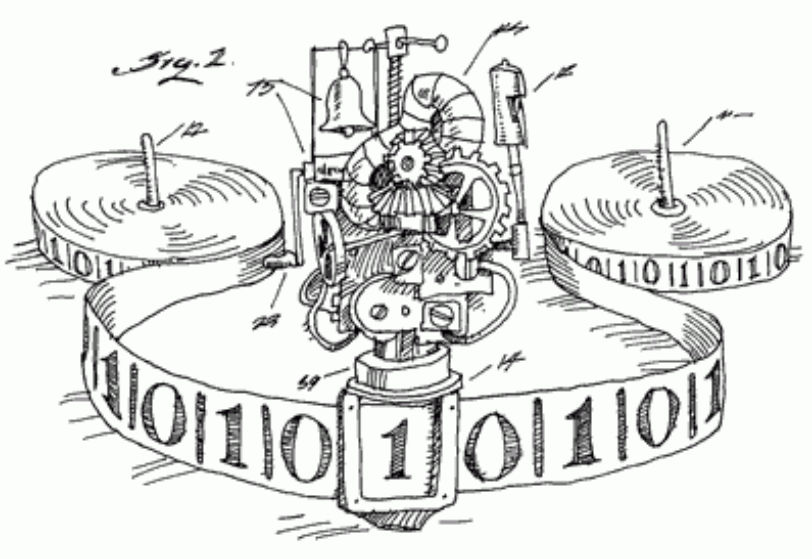
INFO 101 – Introduction to Computing and Security

[2020 - Week 5 / 1]



Prof. Dr. Rui Abreu

University of Porto, Portugal

rui@computer.org

@rmaranhao

Content

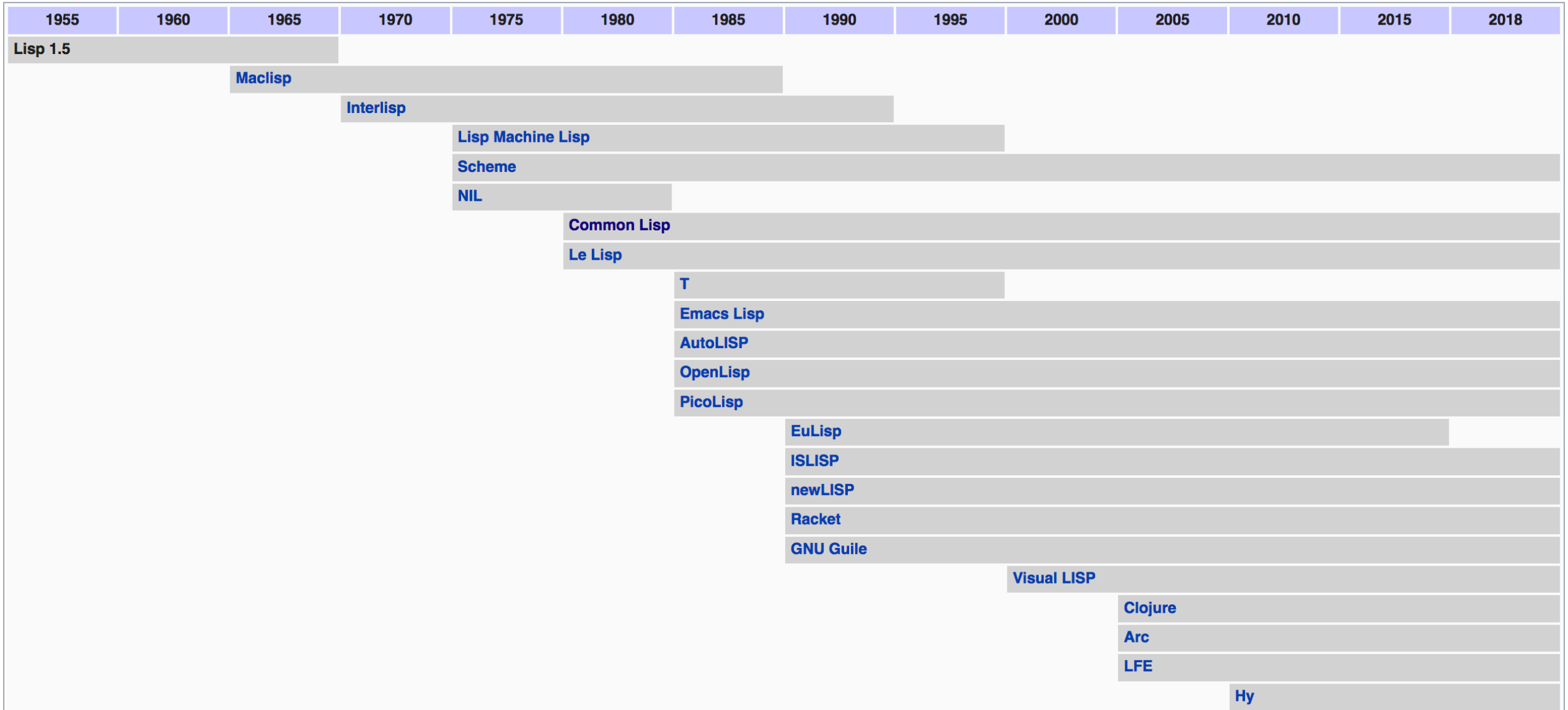
- Introduction into LISP
  - Simple expressions using operations with numbers
  - Simple expressions using lists
  - Conditionals
  - How to define variables and functions
  - Quote functions
  - How functions are called and call-by-value
  - Semantics behind LISP (simplified)
- Recursive LISP functions
- Proving correctness of recursive functions (induction)
- The big O notation using examples
- Functions calling themselves
- The halting and the equivalence problem (only overview)

What is LISP?

- LISP is a **family** of **programming languages** dating back from 1958 using fully parenthesized prefix notation.
- Invented by John McCarthy (co-founder of **Artificial Intelligence**)
- Dialects:
  - Scheme
  - **Common Lisp**

- Have been heavily used for Artificial Intelligence applications
- Based on **Alonzo Church's lambda calculus**
- Lisp derives from „**LISt Processor**“

Timeline of LISP



Some additional remarks

Edsger W. Dijkstra in his 1972 Turing Award lecture said,

"*With a few very basic principles at its foundation, it [LISP] has shown a remarkable stability. Besides that, LISP has been the carrier for a considerable number of in a sense our most sophisticated computer applications. LISP has jokingly been described as “the most intelligent way to misuse a computer”. I think that description a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts*."

A Lisp computer

So why are we learning LISP?

- LISP is still in use
- LISP is a functional programming language
- You can write very powerful programs in LISP with less effort compared to other programming languages
- LISP provides some features todays languages do not have
  - In LISP programs and data have the same form

- It is fun to use!
  - But we focus not on learning the whole language
  - We discuss the basic concepts only and a little bit more

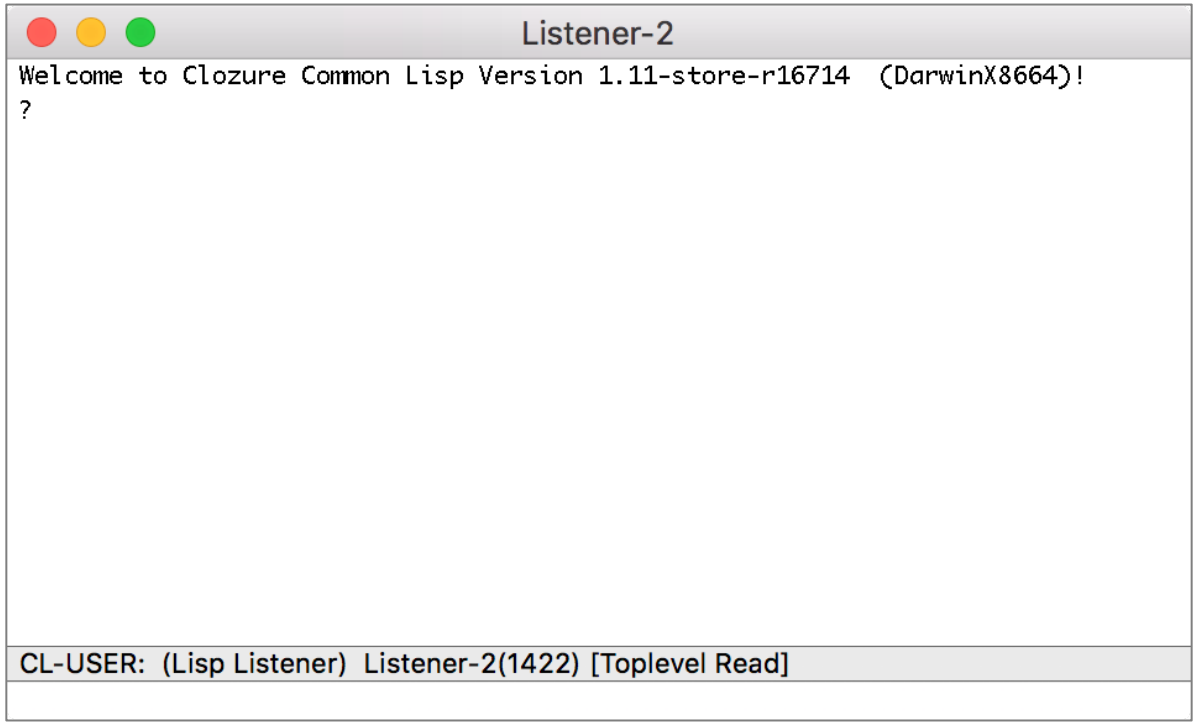
For more information about LISP

- Have a look at:
  - The Common Lisp Cookbook: https://lispcookbook.github.io/cl-cookbook/ (where you also find links to implementations)
  - Wikipedia: https://en.wikipedia.org/wiki/Common_Lisp (also including links to implementations)

- I am using Clozure Lisp: https://ccl.clozure.com/

Let us start with some examples…

- We have to install a LISP interpreter, e.g., **Clozure Common Lisp**
- After starting we get a window where we can enter a command



- Let us try `(+ 1.2 3.4)`
- And we receive `4.6000004` (the part after 4.6 is due to rounding errors)

… and some more …

```
? (+ 1 5)
6
? (+ 1)
1
? (+ 2 2 3)
7
? (+ (- 2 3) 4)
3
? (* (+ 2 4 5) (- 0 2))
-22
?
```

LISP makes use of prefix notation, i.e.:

$$\texttt{function argument}_1 \ldots \texttt{argument}_n$$

surrounded by parentheses ().

We see that LISP can handle integers as well as floating point numbers!

Prefix, postfix and infix notation

- Notations for writing **expressions** like ”1 + 2” where ”1” and “2” are **operands** and “+” is the **operator** (or **function**)
- Depending on the position of the operator we distinguish:
  - **Prefix**: operator at the beginning of an expression
  - **Infix**: operator between its operands
  - **Postfix**: operator at the end of an expression

- **Example**: “3 + (5 – 3)” in infix notation is
  - Prefix: “+ 3 (- 5 3)”
  - Postfix: “3 (5 3 -) +”

What else can we do with Lisp?

- Handling lists!

This is a quote (') can also be written as command `quote`
Used to say that we do not want to execute the list after the quote

```
? (car '(a b c d))
A
```

First element of a list

```
? (cdr '(a b c d))
(B C D)
```

All elements of a list without its first element

```
? (cons 'x '(a b c d))
(X A B C D)
```

Adding an element to a list at the first place

List processing and errors

```
? (car (quote (a b c d)))
A
? (car (a b c d))
> Error: Unbound variable: B
> While executing: CCL::CHEAP-EVAL-IN-ENVIRONMENT, in process Listener-2(1422).
> Type cmd-/ to continue, cmd-. to abort, cmd-\ for a list of available restarts.
> If continued: Retry getting the value of B.
> Type :? for other options.
1 > :q
?
```

Booleans

- Booleans are represented using `T` for true and `nil` for false

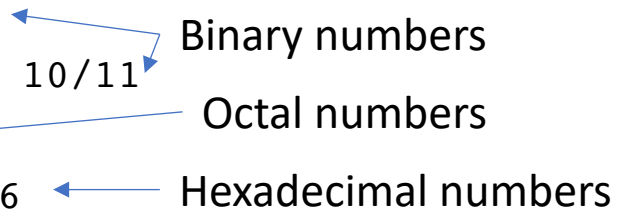
- Functions on Booleans: `and, or, not`

  (uses short-circuit evaluation = stop computing values if the result is already determined).

Numbers (integers, reals)

Integer representation

```
123 ==> 123
+123 ==> 123
-123 ==> -123
123. ==> 123
2/3 ==> 2/3
-2/3 ==> -2/3
4/6 ==> 2/3
6/3 ==> 2
#b10101 ==> 21
#b1010/1011 ==> 10/11
#o777 ==> 511
#xDADA ==> 56026
```



Floating point representation of reals

```
1.0 ==> 1.0
1e0 ==> 1.0
123.0 ==> 123.0
123e0 ==> 123.0
0.123 ==> 0.123
.123 ==> 0.123
123e-3 ==> 0.123
123E-3 ==> 0.123
0.123e20 ==> 1.23e+19
```

Numbers (integers, reals)

Functions

- **Basic functions**: `+`, `-` , `*`, `/` (can be also used with one argument)
- **Rounding**: `truncate`, `floor`, `ceiling`, `round`
- **Modulo** : `mod`
- **Remainder** : `rem`
- **Others:** `log, exp, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, min, max, sqrt, exp (=e`^{n}`), expt (=n`^{m}`), abs`
- `pi` – returns the value of π

Predicates

- = – equivalence not taking care of the type
- `EQL` – equivalence taking care of the type
- /= – the negation of = (unequal)
- <,>, <=, >= – smaller, greater and in combination with equivalence
- `zerop`, `minusp`, `plusp` – testing for 0, negative, or a positive real number
- `evenp`, `oddp` – testing integers for even or odd
- `numberp` – tests whether the argument is a number

Lists

- A list is the basic structure of programs and data in LISP.
- A list comprises elements where each element is either a list or an atom (e.g. a symbol, or number)
- **Empty list**: `()` or `nil`

- There are functions for adding elements, removing elements, and accessing elements of lists available.
- There are also predicates available for checking the status of a list.

Lists

Functions

- `car` – returns the first element of a list
- `cdr` – returns the rest of a list (all elements without the first one)
- `cons` – adds the given element at the first position in a list
- `append` – returns the union of all lists given as arguments
- `list` – returns a list comprising all given arguments as elements
- `length` – returns the number of elements (at its highest level)
- `reverse` – returns the given list with the elements reversed

Predicates

- `equal` – returns `T` if two lists are equivalent, and `nil` otherwise. Works also for all other data types.
- `atom` – returns `T` if the argument is an atom, i.e., not a list, and `nil` otherwise.
- `listp` – returns `T` if the argument is a list, and `nil` otherwise.
- `null` – returns `T` if the argument is an empty list, and `nil` otherwise.
- `member` – returns `T` if the first argument is element of the list, i.e., the second argument, and `nil` otherwise.

Lists

Functions

- `subst` – returns a list where all occurrences of an atom are replaced by the given expression.

  Form: `(subst` *newExpr oldatom list*`)`

```
? (subst '(a b) 'a '(c b a))
(C B (A B))
```

- `last` – returns a list comprising only the last element of the given list.

Further notes on lists

- `car` and `cdr` can be combined in one command, e.g. `cadr` delivers the second element in a list (if it exists), and `caddr` the third one. You can also use `cdar`, but in this case the first element has to be a list, where `cdr` can be applied.

Other important LISP functions

- **Conditionals / if statement:**

`(` `if` *condition then-branch else-branch* `)`

"if the *condition* evaluates to `T`, the *then-branch* is evaluated and otherwise, the *else-branch.*"

```
? (if (> 2 1) (+ 1 2) (- 2 1))
3
? (if (< 2 1) (+ 1 2) (- 2 1))
1
```

Other important LISP functions

- **Conditionals / `cond` statement:**

(`cond` ($condition_1$ $action_1$) … ($condition_n$ $action_n$))

"Each condition-action pair is called clause. Clauses are evaluated. The first clause *i* where $condition_i$ evaluates to `T` is taken and $action_i$ is evaluated. The result of $action_i$ becomes the result of `cond`."

```
? (cond ((> 2 1) (+ 1 2)) (T (- 2 1)))
3
```

Other important LISP functions

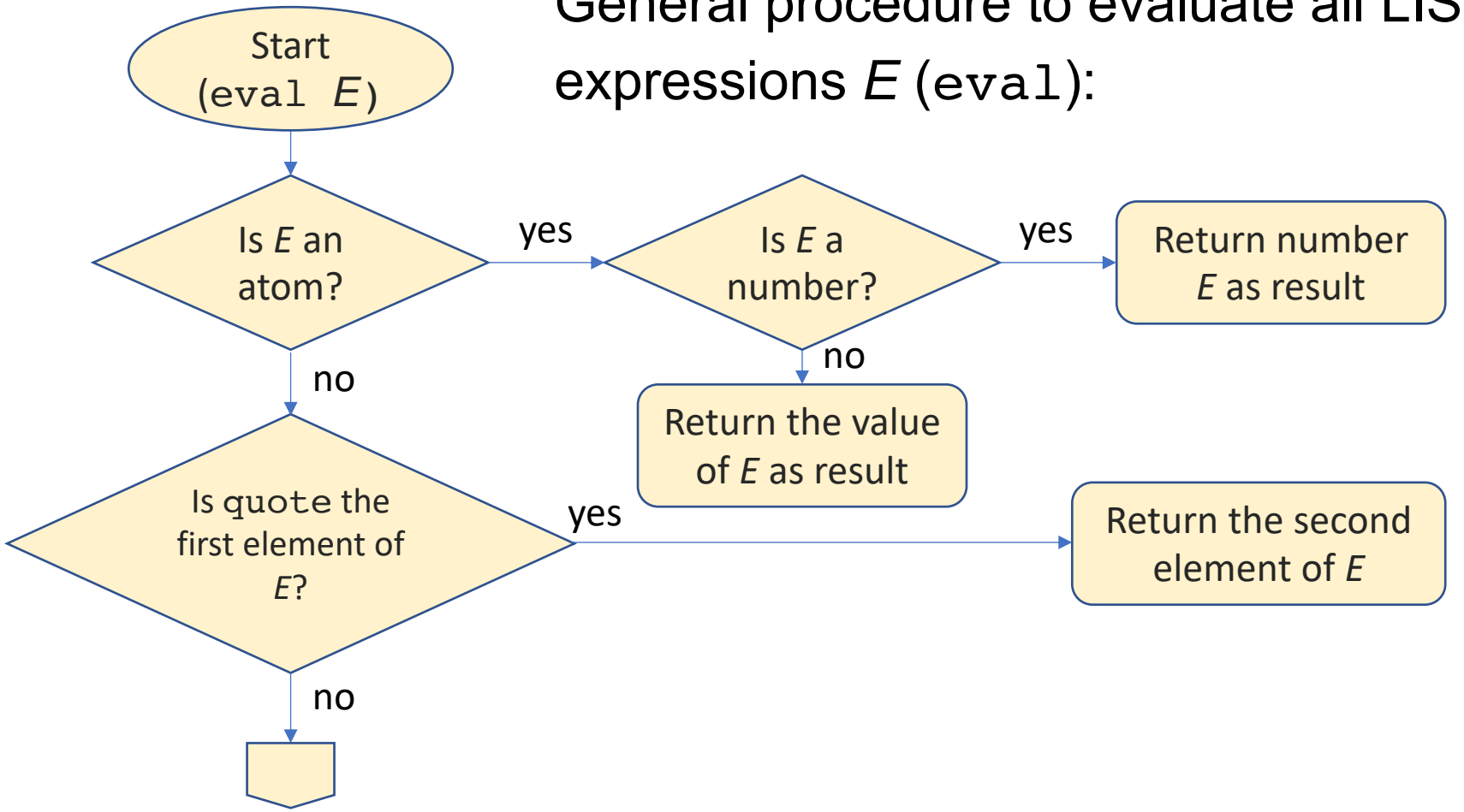
- `quote` – Delivers back its argument without evaluation, e.g., `(quote (+ 1 3))` returns the list `‘(+ 1 3)`. Note that `quote` is equivalent to `‘`, hence we can also write: `‘(+ 1 3)`.

- `eval` – Evaluates the given expression, e.g., `(eval (quote (+ 1 3)))` returns `4` as a result.

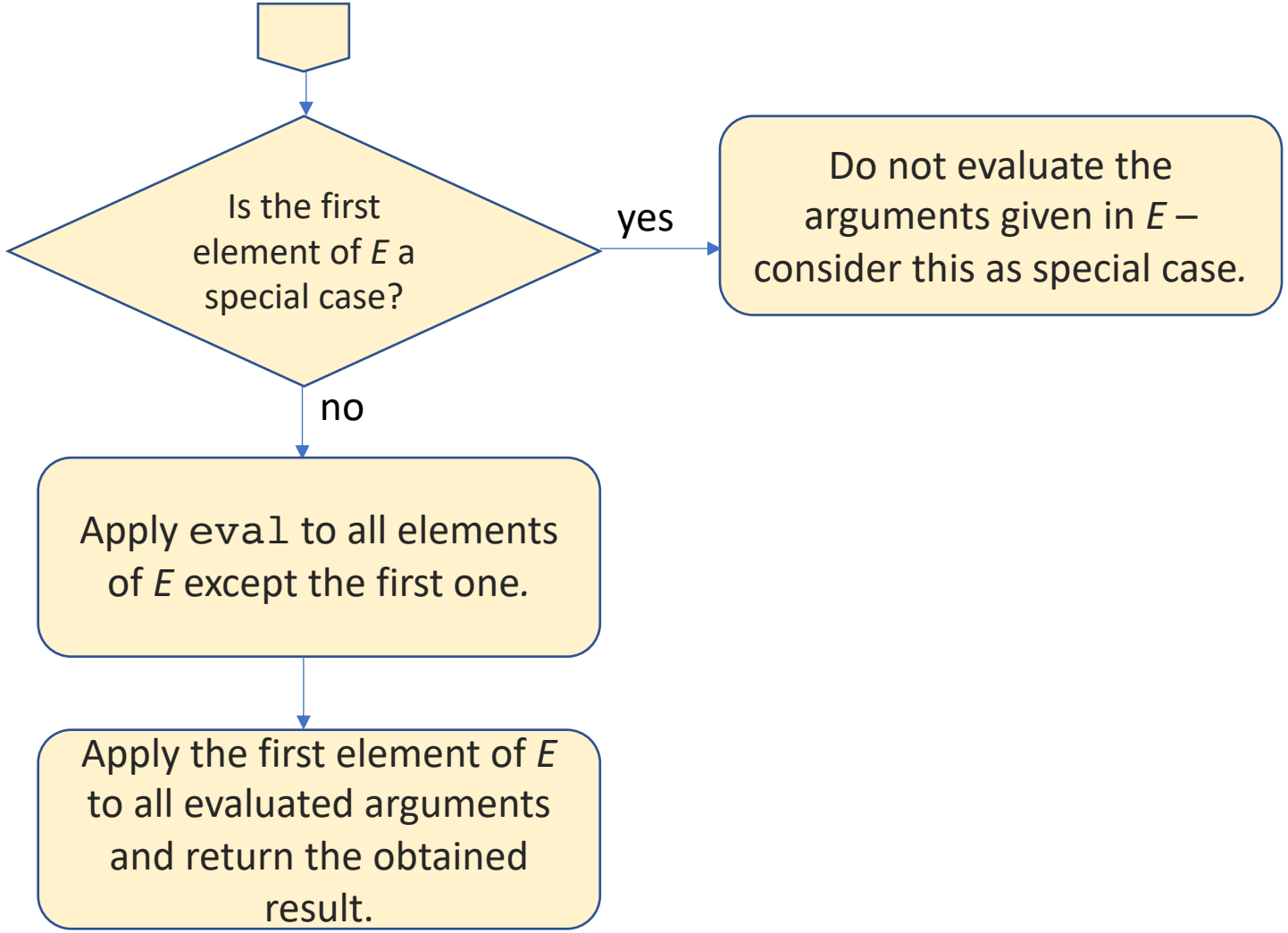
- `setq` – Sets the value of a variable, e.g. `(setq my-list ‘(a b c))` generates a variable `my-list`, which can be accessed in other expressions, e.g., `(car my-list)` returns `a`.

Evaluation of LISP expressions

General procedure to evaluate all LISP expressions *E* (`eval`):



Evaluation of LISP expressions



Example evaluation

- `eval: (+ (* 2 3) (- 2 5))`
  - `eval: (* 2 3)`
    - `eval: 2` → 2
    - `eval: 3` → 3
    - Apply * to return values → 6
  - `eval: (- 2 5)`
    - `eval: 2` → 2
    - `eval: 5` → 5
    - Apply – to return values → -3
  - Apply + to return values → 3

User defined functions

- We can also write functions that can be applied using different arguments

(`defun` *name* ($parameter_1$ … $parameter_n$) *body*)

- A function has a *name* and parameters written in a list (which can also be empty. The *body* of a function comprises its definition where we use the parameters as variables.

- **Example**: `(defun foo (x) (if (> x 2) (- x 2) x))`

Example (cont.)

- Function **definition**:

```
(defun foo (x)
   (if (> x 2)
      (- x 2)
      x))
```

- Function **call**: `(foo 3)` – What is going to happen?

  The value of expression `3`, i.e., the number 3 is assigned to variable `x` in the body of `foo`. Afterwards, the body of `foo` is evaluated (call by value).

Example (cont.)

- What happens in the following case?

```
(foo (+ 3 5))
```

Again the expression `(+ 3 5)` is evaluated and assigned to `x`. Afterwards, the body of `foo` is evaluated using the value of `x`.

Call-by-value evaluation

- When calling a user defined function or procedure first all arguments are evaluated and assigned to their corresponding parameters (i.e., variables) used in the function definition. Afterwards, the body of the function is evaluated using the variables and their assigned values.

- *Note: There is also call-by-name and call-by-reference.*

Let us discuss some examples

- Implementing the logic implication operator

  We have and, or, and not in LISP but nothing else!

```
;;;; Implementation of a implies b
(defun implies (a b)
  (if (not a) T b))
```

This implementation follows the definition of implies. If a is false, then the result must be true. If a is true, the result depends on b only.

Second version of `implies`

```
;;;; 2nd implementation of a implies b
(defun implies-2 (a b)
  (or (not a) b))
```

This implementation makes use of the following logical equivalence:

$$a \rightarrow b \models\models \neg a \vee b$$

What to take with you

- Both versions of imply can be tested in order to check correctness!
  - Try all possible inputs and have a look at the output!
- The different versions of imply are based on different properties.
  - `implies` makes use of the definition of an implication
  - `implies-2` makes use of a known logical equivalence
- Hence, we do not even need testing. We only have a look whether the properties are correctly represented in the source code.

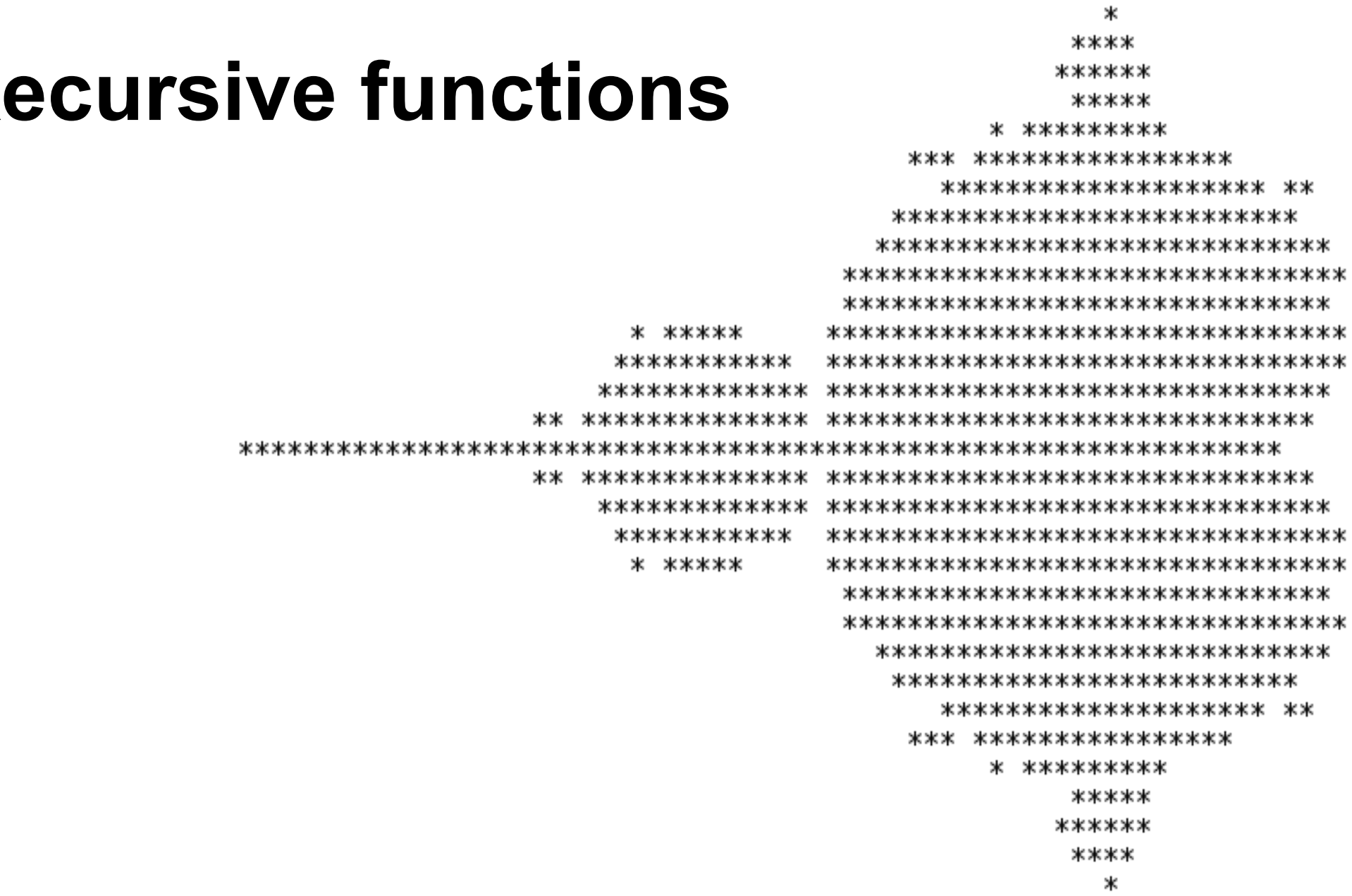
… and another example.

- A function converting the temperature from Fahrenheit into Celcius

$$T_C = \frac{T_F - 32}{1.8}$$

```
(defun F-to-C (temp-F)
    (/ (- temp-F 32.0) 1.8))
```

Recursive functions



Recursive function

- Is a function that calls itself!
- Used instead of iterators like for- or while-loops (not only) in case of functional languages.
- May not halt on all inputs.
- Similar to defining functions inductively in mathematics.

$$x * y = \begin{cases} 0 & if\ (x = 0) \\ y + ((x - 1) * y) & otherwise \end{cases}$$

First example – Add operator for integers larger or equal to zero

- Defining the add operation using only increment and decrement functions

```
(defun inc (x) (+ x 1))
(defun dec (x) (- x 1))

(defun plus (x y)
    (if (= x 0) y
        (plus (dec x) (inc y))))
```

Second example – Multiplication for positive integers including zero

```
(defun mult (x y)
    (if (= x 0) 0
        (plus y (mult (dec x) y))))
```

$$x * y = \begin{cases} 0 & if\ (x = 0) \\ y + ((x-1) * y) & otherwise \end{cases}$$

How can we be sure that the programs are correct?

- Let us prove their correctness!

- Mathematical induction over the parameter that determines the recursion!

- What is induction?
  - State a theorem, i.e., the implemented function works as expected, formally.
  - Prove that the theorem hold in case of the initial value where no recursion is happening (induction base or IB)
  - Assume that the theorem hold in induction step n (induction assumption or IA)
  - Prove that the theorem also holds in step n+1 (induction step or IS)

Proving the `plus` implementation

```
(defun plus (x y)
             (if (= x 0) y
                    (plus (dec x) (inc y))))
```

Theorem: `(plus n m)` returns `n+m`

Proof:

(IB) We have to show that `(plus 0 m)` returns `0+m`. When calling `(plus 0 m)`, the condition `(= x 0)` evaluates to `T` and the value of `y` is given back as result, which is equivalent to `m` and therefore, `0+m` holds. Hence, IB is fulfilled.

(IH) Let us assume that `(plus n m)` returns `n+m`

Proving the `plus` implementation

```
(defun plus (x y)
              (if (= x 0) y
                     (plus (dec x) (inc y))))
```

(IS) Show that `(plus n+1 m)` returns `n+m+1`.

We see that `x` must be larger than 0. Hence, we have to evaluate the expression `(plus (dec x) (inc y))`. From the definition of `dec` and the call `(dec n+1)` we see that the first parameter becomes `n`. From the definition of `inc` we obtain for the second parameter and its call `(inc y)`, the result `m+1`. From the (IA) we know that `(plus n m+1)` has to return `n+m+1`. Hence, the theorem holds. □

Proving the `mult` function

- Can be done in a similar way.
- We know now that `plus` is correct, which can be used for the proof.

```
(defun mult (x y)
              (if (= x 0) 0
                     (plus y (mult (dec x) y))))
```

Theorem: `(mult n m)` returns `n*m`

Verifying functions

- Proving their correctness (e.g. using induction)
- Testing functions
  - Trying functions on several inputs and looking whether the output confirms to the specification of the function. E.g. `plus` should deliver 14 when called with 2 and 12.

- Note that both proving correctness (also called formal verification) and testing are required in practice!

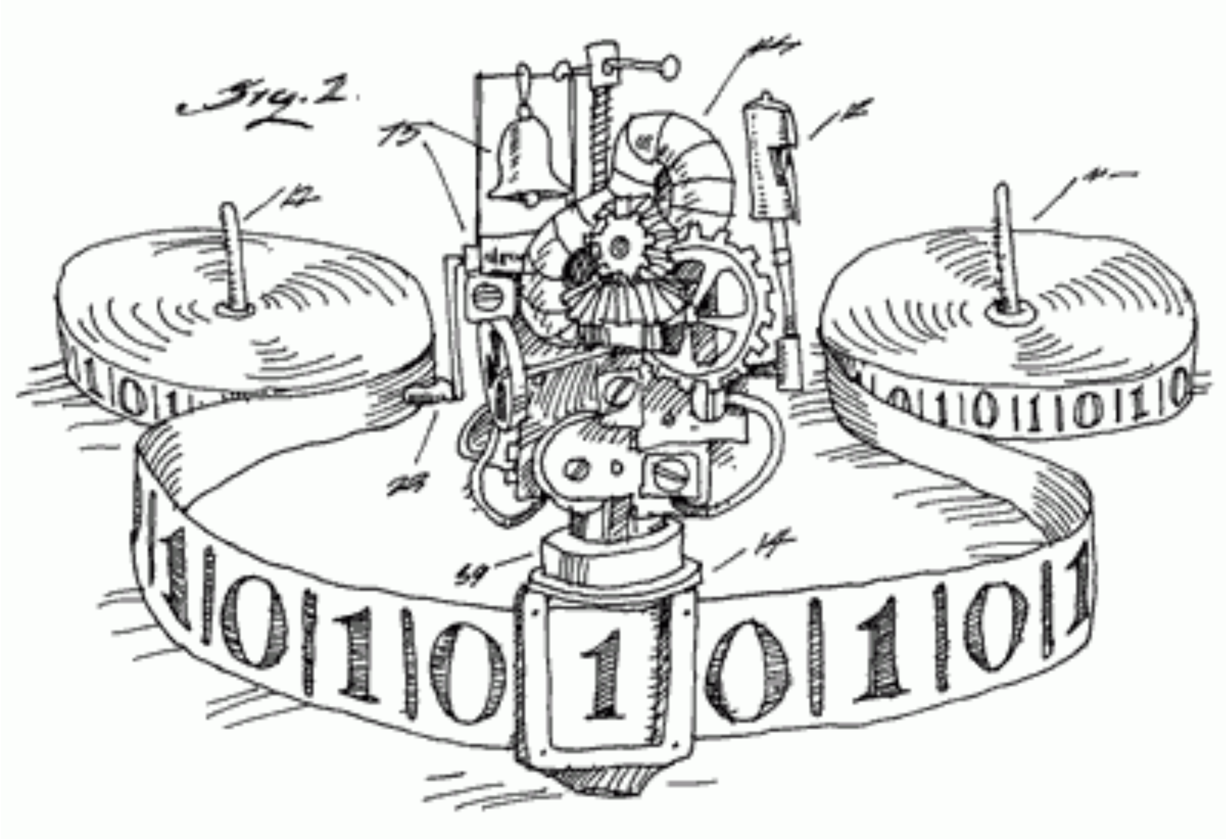
„*Testing shows the presence, not the absence of bugs.*“

[Edsger W. Dijkstra]

„*Beware of bugs in the above code; I have only proved it correct, not tried it.*“

[Donald Knuth]

Limits of computation

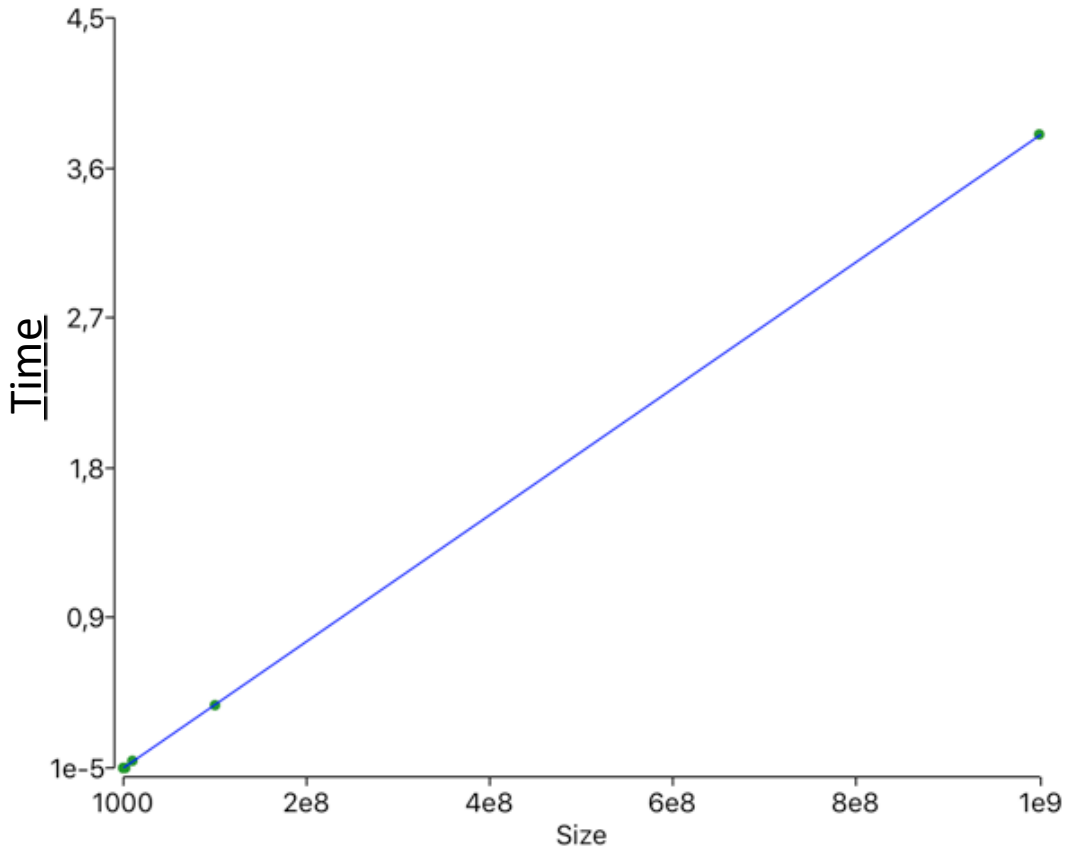


Runtime of functions or procedures

- Let us make an experiment!
- Call the function plus with different inputs for the first parameter ranging from 1,000 to 1,000,000,000 and maybe more.
- Using the available `time` function in common lisp, we are able to have a look at time depending on the input size.

Time as a function of the input size

- Calling `(plus n 1)` leads to a straight line.
- We can say that the runtime of `plus` depends linearly on the (size) of the first parameter.



Make the same experiment for `mult`

- Cannot be carried out:

Limitation of the runtime system (can only be detected using **testing**!)

```
? (time (mult 100000 1))
> Error: Stack overflow on value stack.
> While executing: MULT, in process Listener-2(81).
> Type cmd-. to abort, cmd-\ for a list of available restarts.
> Type :? for other options.
```

Let us try to estimate the runtime from the source code directly

```
(defun plus (x y)
              (if (= x 0) y
                     (plus (dec x) (inc y))))
```

- Observations:
  - The number of recursions for given parameters n and m only depend on n when calling `(plus n m)`.
  - We have n+1 calls. If n = 0 there is one, if it is 1 there are two, etc.
  - When assuming that the other commands do have a runtime we can ignore, the overall runtime of `plus` is a function of n only.
  - This is a worst-case scenario.

Runtime of `plus` is O(n)

Let's analyze `mult`

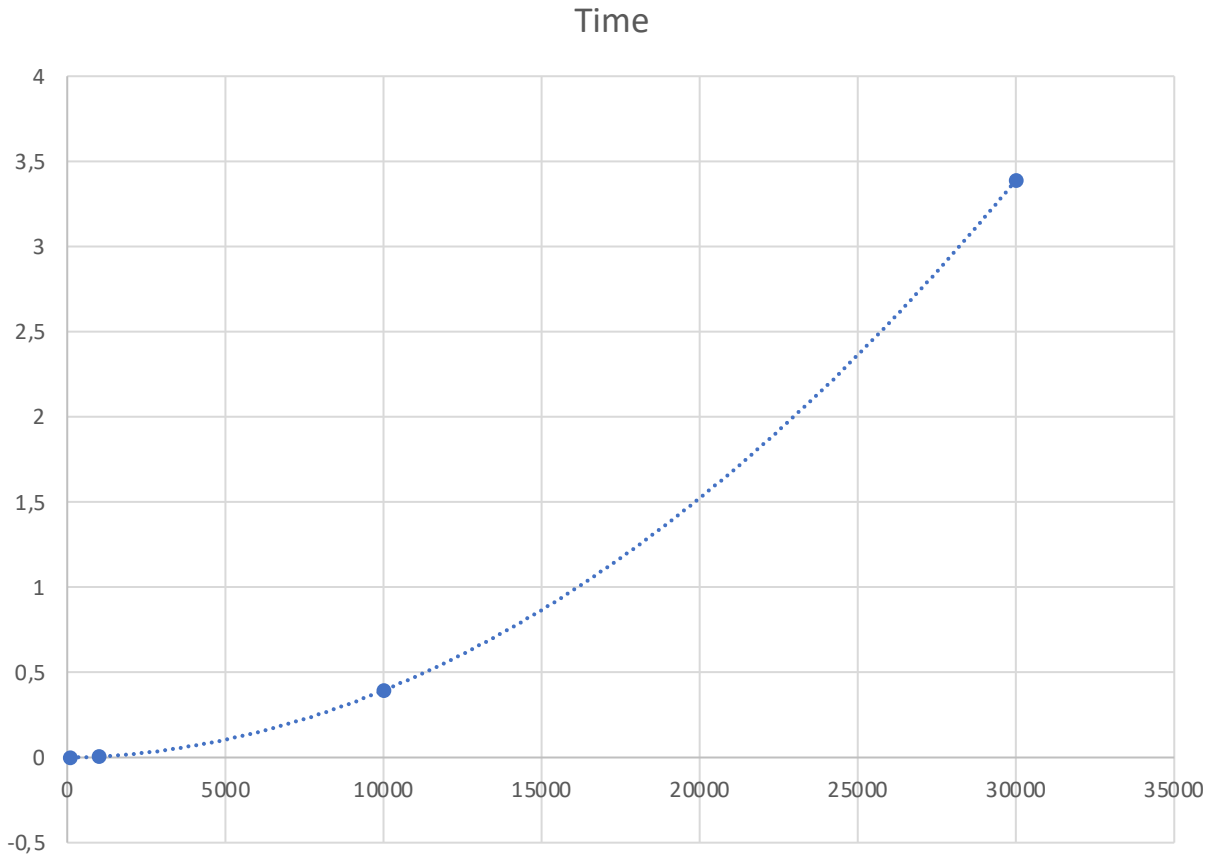
```
(defun mult (x y)
              (if (= x 0) 0
                     (plus y (mult (dec x) y))))
```

- The number of recursive calls is similar to plus when calling `(mult n m)`. We need n+1 recursive calls.
- In this case, we may not be able to ignore the runtime of all commands/functions used, because we rely on the `plus` function.
- In each call, `plus` is called using m. Hence, we see that the runtime must be a function of n * m.

Runtime of `mult` is O(n*m)

Let us try again the experiment

- This time we use the call `(mult n n)` for different values of `n`:



Big O notation

- Big O notation is a mathematical notation that describes the limiting behavior of a function
- Used to classify computers
- ”*Count the number of evaluated/executed statements for given parameters*”
- Used for the analysis (of the worst-case) runtime of algorithms

Big O notation

- Simplification rules:
  - If $f(x)$ is a sum of several terms, if there is one with largest growth rate, it can be kept, and all others omitted.
  - If $f(x)$ is a product of several factors, any constants (terms in the product that do not depend on x) can be omitted.

- O(n + c) where c is a constant is approx. O(n)
- $O(c*n^2)$ where c is a constant is approx. $O(n^2)$

Big O notation

- O(1) .. Constant runtime
- O(log n) .. Logarithmic runtime
- O(n) .. Linear runtime
- O(n*log n) .. Loglinear, or quasilinear runtime
- $O(n^2)$ .. Quadratic runtime
- $O(n^c)$ .. Polynomial runtime
- $O(c^n)$ .. Exponential runtime
- Etc.

Remarks

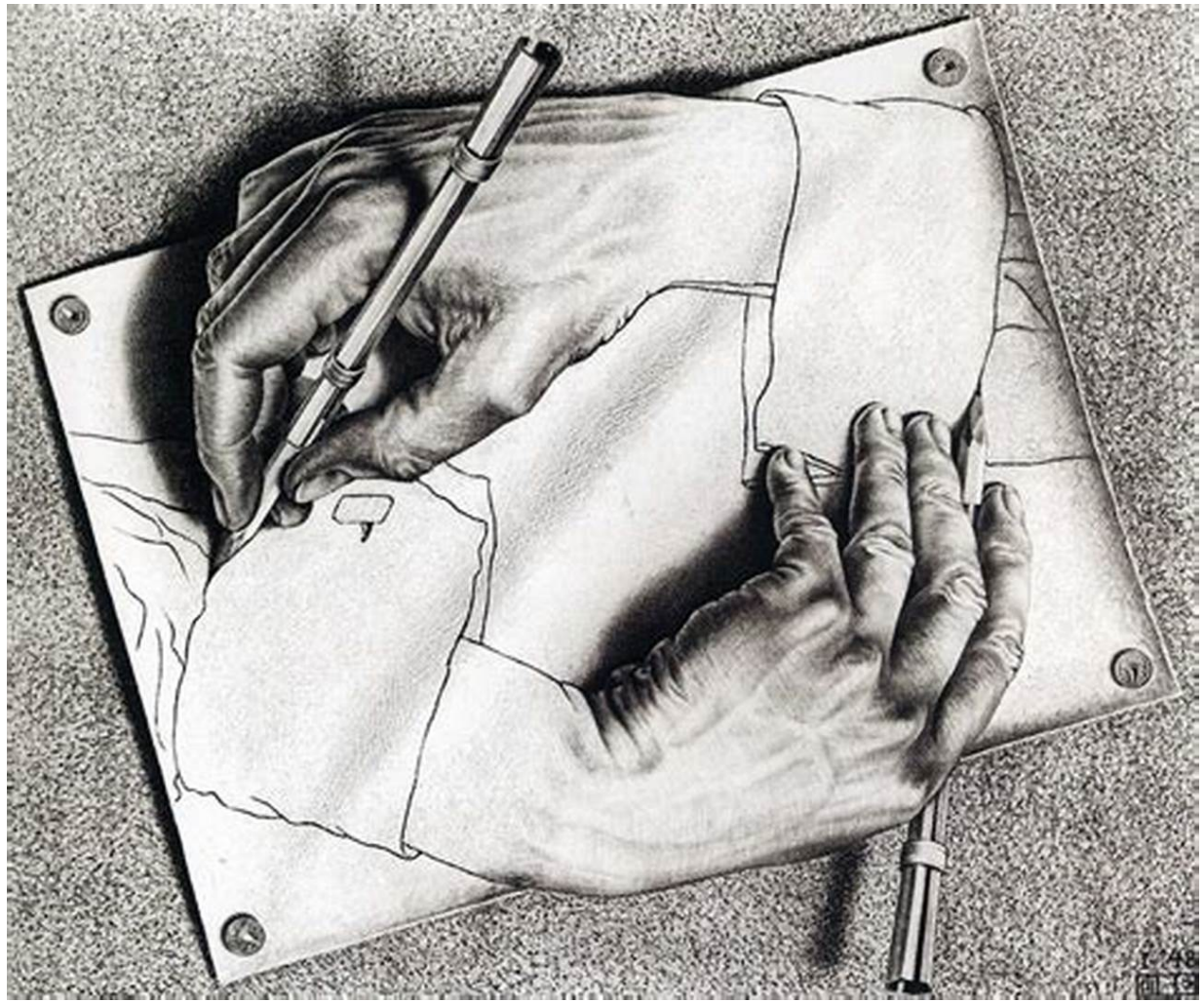
- Consider $O(2^n)$: for n=0, we have 1, but for n=64, we get $1.84467*10^{19}$. Even in case, one execution takes only one nano-second (10^{-9} seconds), the execution of a program would require 585 years to finish!

- Related area: Computational complexity theory
  - P, NP, and all other complexity classes
  - Not for an algorithm but stated for a problem
  - Used to classify problems
  - Example: checking for satisfiability in propositional logic is NP-complete (e.g. a difficult problem)

What to take with you?

- O(.) estimates the worst-case complexity of an algorithm
- There are algorithms, which may take far too long to terminate
- O(.) can be computed considering the number of statements to be executed for given parameters.
- In practice, algorithms should have at least polynomial runtime with a small constant power (1-3).

Can programs analyze themselves?



To answer this question…

- … we first have a look at programs processing lists.

- Let us write a function, that returns the length of a list like `length`.

```
(defun my-length (a-list)
    (if (null a-list) 0
        (+ 1 (my-length (cdr a-list)))))
```

Function `my-length`

```
(defun my-length (a-list)
     (if (null a-list) 0
          (+ 1 (my-length (cdr a-list)))))
```

- Let us test `my-length`:

```
? (my-length '(a b c d e))
5
? (my-length nil)
0
```

Function `my-length`

```
(defun my-length (a-list)
     (if (null a-list) 0
          (+ 1 (my-length (cdr a-list)))))
```

- But we can also apply `my-length` on itself!

- `? (my-length '(defun my-length (a-list) (if (null a-list) 0 (+ 1 (my-length (cdr a-list))))))`

  `4`

- So why 4? `my-length` comprises 4 atoms and lists at its top level.

Extending `my-length`

- What is about a function that returns the number of all atoms in a list (and not only those in the top level)?

```
(defun sum-of-all-atoms (a-list)
  (cond ((null a-list) 0)
        ((null (car a-list)) (sum-of-all-atoms (cdr a-list)))
        ((atom (car a-list)) (+ 1 (sum-of-all-atoms (cdr a-list))))
        (T (+ (sum-of-all-atoms (car a-list)) (sum-of-all-atoms (cdr a-list))))))
```

- When applying `sum-of-all-atoms` on itself we obtain 29!

What we have learned!

- Basic handling of lists
- We can make a function and apply it on its source code (i.e., on itself!)

- We can further write a LISP program that evaluates other LISP programs (and also itself).

- Hence, the answer to the question **“Can programs analyze themselves?”** has to be answered with **YES, THEY CAN!**

Some remarks

- Programs modifying themselves can be more or less easily written in LISP.
- The reason is that data and programs share the same structure, i.e., lists!
- In other programming languages this is not that simple and requires additional steps for parsing programs.

- There is much more functionality in LISP. We only scratched the surface.

Does a program terminates?

- This has been a big question in Computer Science!
- Let us have a look at the following function:

```
(defun non-terminating () (non-terminating))
```

- `non-terminating` obviously cannot terminate!

Can we find a function that checks for termination?

- We know that we can write programs analyzing other programs and itself.

- Can we write a program that solves the **halting problem?**

- **Halting problem**: Given a program `prog`. Can we write a program that answers the question, whether `prog` holds on all inputs in a finite amount of time?

Answering the halting problem

- Let us assume there is a such a function, say `is-terminating`, which returns `T` if the given function terminates on all inputs, and `nil` otherwise.

- Let us now construct the following program:

```
(defun bad (prog) (
  (if (is-terminating prog) (bad prog) nil)))
```

Answering the halting problem

- Let us now call `bad` on `bad` , e.g., `(bad 'bad)`
- Consider again the source code:

```
(defun bad (prog) (
  (if (is-terminating prog) (bad prog) nil)))
```

- We see that `bad` if it is terminating will not terminate, and otherwise would return `nil`. As a consequence, we cannot find a program that always answers questions about termination!

Answering the halting problem

- **The halting problem is undecidable!**

- You can never find a program that makes a decision about termination for all programs.

- There as a similar problem in Computer Science, e.g., the equivalence problem, which is also undecidable!
- **Equivalence problem**: Given two programs `prog1` and `prog2`. Can we find a program that can always decides whether the behavior of `prog1` is equivalent to the behavior of `prog2`.