Functional Programming
Lecture 1

Rostislav Horčík
Niklas Heim

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz
heimnikl@fel.cvut.cz

Introduction

What is functional programming?

- Functional programming is a programming style that prefers to structure computer programs as compositions of pure functions.

What is functional programming?

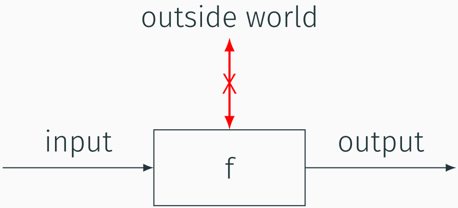
- Functional programming is a programming style that prefers to structure computer programs as compositions of pure functions.
- It does not depend on a programming language but some languages are more suitable for functional programming than others.

What is functional programming?

- Functional programming is a programming style that prefers to structure computer programs as compositions of pure functions.
- It does not depend on a programming language but some languages are more suitable for functional programming than others.
- Functional programming languages are languages encouraging usage of pure functions.

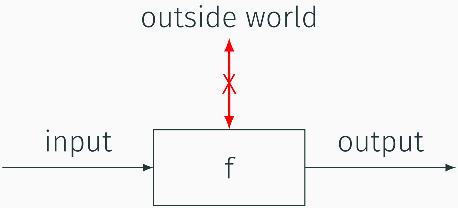
Pure functions

A pure function is a function that, given the same input, will always return the same output and has no observable side effect.



Pure functions

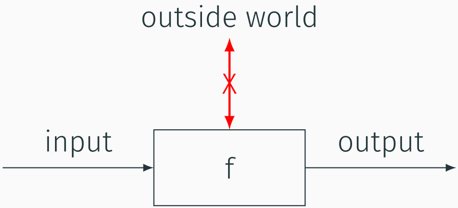
A pure function is a function that, given the same input, will always return the same output and has no observable side effect.



No side effects = pure functions cannot modify and don't depend on any existing data structures

Pure functions

A pure function is a function that, given the same input, will always return the same output and has no observable side effect.



No side effects = pure functions cannot modify and don't depend on any existing data structures

A pure functional program = a composition of pure functions

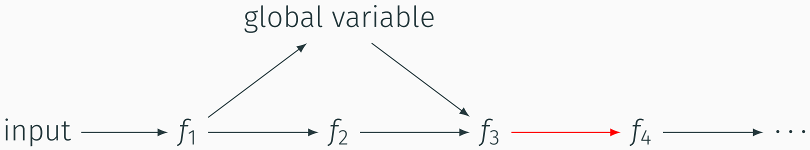
Examples of (im)pure functions

```python
counter = 0

def pure(x, y):
  return (x + y)/2

def do_other(x):
  global counter
  counter += 1
  return x**2

def depends_on_other(x):
  return counter + x**2
```

- unit testing and debugging

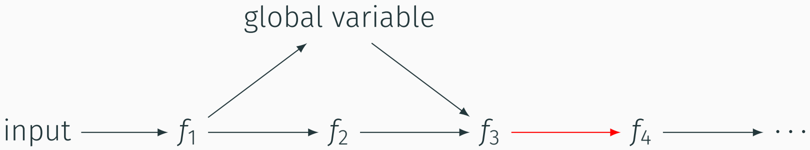
Advantages

- unit testing and debugging



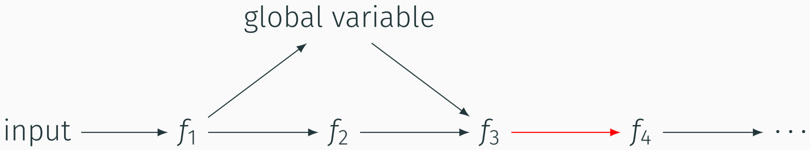
Advantages

- unit testing and debugging
- simpler refactoring



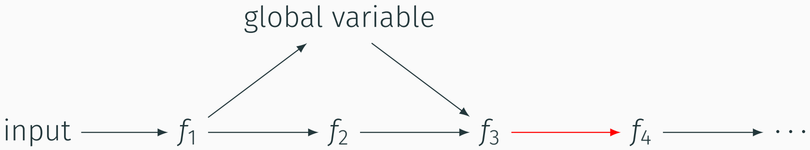
Advantages

- unit testing and debugging
- simpler refactoring
- concurrency and parallelism – $f(g_1, \ldots, g_n)$



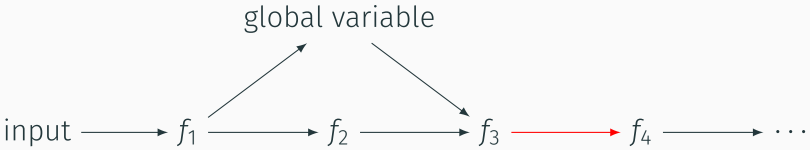
Advantages

- unit testing and debugging
- simpler refactoring
- concurrency and parallelism — $f(g_1, \ldots, g_n)$
- formal verification — mathematical induction, algebraic reasoning



Advantages

- unit testing and debugging
- simpler refactoring
- concurrency and parallelism — $f(g_1, \ldots, g_n)$
- formal verification — mathematical induction, algebraic reasoning
- compiler optimization, pure functions are cachable



Consequences

- Imperative loops updates a state in each iteration. FP uses recursion instead (stack holds the state).

Consequences

- Imperative loops updates a state in each iteration. FP uses recursion instead (stack holds the state).
- Data structures in pure functional programs are immutable.

Consequences

- Imperative loops updates a state in each iteration. FP uses recursion instead (stack holds the state).
- Data structures in pure functional programs are immutable.
- To modify a data structure, we need to copy it and do the desired modification.

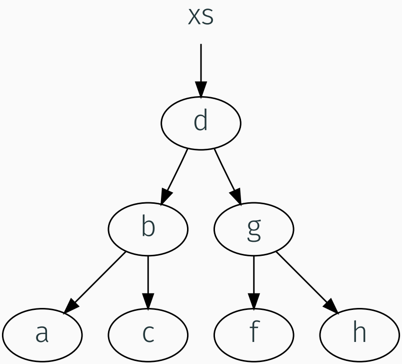
Consequences

- Imperative loops updates a state in each iteration. FP uses recursion instead (stack holds the state).
- Data structures in pure functional programs are immutable.
- To modify a data structure, we need to copy it and do the desired modification.
- The code generated by functional programming languages is typically less efficient.

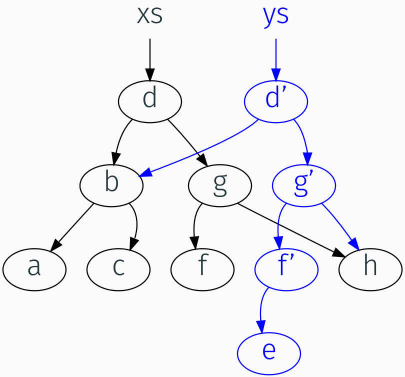
Consequences

- Imperative loops updates a state in each iteration. FP uses recursion instead (stack holds the state).
- Data structures in pure functional programs are immutable.
- To modify a data structure, we need to copy it and do the desired modification.
- The code generated by functional programming languages is typically less efficient.
- To reduce the number of copying, persistent data structures are used.

Persistent data structures



```
ys = insert ("e", xs)
```



Necessary side effects

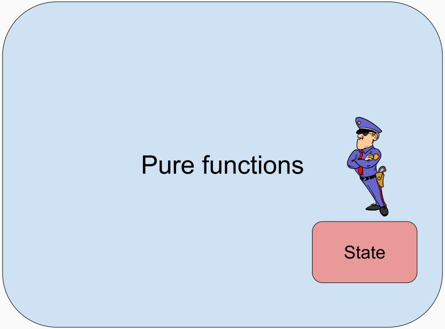
- A pure functional program behaves like a calculator.

Necessary side effects

- A pure functional program behaves like a calculator.
- Real applications need side effects. In FP, we tend to make the pure part of an app as large as possible, keeping the "unsafe" effectful code to the bare minimum.

Necessary side effects

- A pure functional program behaves like a calculator.
- Real applications need side effects. In FP, we tend to make the pure part of an app as large as possible, keeping the “unsafe” effectful code to the bare minimum.



A bit of history

Alonzo Church	Alan Turing

A bit of history

Alonzo Church	Alan Turing
λ-calculus	Turing machine

A bit of history

Alonzo Church	Alan Turing
λ-calculus	Turing machine
Functional programming	Imperative programming

A bit of history

Alonzo Church	Alan Turing
λ-calculus	Turing machine
Functional programming	Imperative programming
Composition of functions	Seq. of instructions changing state

A bit of history

Alonzo Church	Alan Turing
λ-calculus	Turing machine
Functional programming	Imperative programming
Composition of functions	Seq. of instructions changing state
Function application	Instruction execution

A bit of history

Alonzo Church	Alan Turing
λ-calculus	Turing machine
Functional programming	Imperative programming
Composition of functions	Seq. of instructions changing state
Function application	Instruction execution
Recursion	Loops

Alonzo Church	Alan Turing
λ-calculus	Turing machine
Functional programming	Imperative programming
Composition of functions	Seq. of instructions changing state
Function application	Instruction execution
Recursion	Loops

Theorem

Turing machines and λ-calculus are equally strong regarding computing functions.

Organization

Course organization

- Web: `https://aicenter.github.io/FUP/`

Course organization

- Web: `https://aicenter.github.io/FUP/`
- Lectures + Labs

Course organization

- Web: `https://aicenter.github.io/FUP/`
- Lectures + Labs
- BRUTE Homework assignments (50 points) ≥ 25

Course organization

- Web: `https://aicenter.github.io/FUP/`
- Lectures + Labs
- BRUTE Homework assignments (50 points) ≥ 25
    - 2x Racket

Course organization

- Web: `https://aicenter.github.io/FUP/`
- Lectures + Labs
- BRUTE Homework assignments (50 points) ≥ 25
    - 2x Racket
    - 2x Haskell

Course organization

- Web: `https://aicenter.github.io/FUP/`
- Lectures + Labs
- BRUTE Homework assignments (50 points) ≥ 25
    - 2x Racket
    - 2x Haskell
    - must have at least 1 point from each

Course organization

- Web: `https://aicenter.github.io/FUP/`
- Lectures + Labs
- BRUTE Homework assignments (50 points) ≥ 25
    - 2x Racket
    - 2x Haskell
    - must have at least 1 point from each
    - Deadlines: -1 per day until +1 is left

Course organization

- Web: `https://aicenter.github.io/FUP/`
- Lectures + Labs
- BRUTE Homework assignments (50 points) ≥ 25
  - 2x Racket
  - 2x Haskell
  - must have at least 1 point from each
  - Deadlines: -1 per day until +1 is left
- Programming exam (30 points) ≥ 16

Course organization

- Web: `https://aicenter.github.io/FUP/`
- Lectures + Labs
- BRUTE Homework assignments (50 points) ≥ 25
    - 2x Racket
    - 2x Haskell
    - must have at least 1 point from each
    - Deadlines: -1 per day until +1 is left
- Programming exam (30 points) ≥ 16
- Theoretical oral exam (20 points) ≥ 0

What will we learn?

- Lisp/Scheme/Racket

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed
  - code-as-data (easy to write interpreters,...)

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed
  - code-as-data (easy to write interpreters,...)
  - allows mutable data

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed
  - code-as-data (easy to write interpreters,...)
  - allows mutable data
- λ-calculus

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed
  - code-as-data (easy to write interpreters,...)
  - allows mutable data
- λ-calculus
- Haskell

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed
  - code-as-data (easy to write interpreters,...)
  - allows mutable data
- λ-calculus
- Haskell
  - pure functional language

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed
  - code-as-data (easy to write interpreters,...)
  - allows mutable data
- λ-calculus
- Haskell
  - pure functional language
  - statically typed

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed
  - code-as-data (easy to write interpreters,...)
  - allows mutable data
- λ-calculus
- Haskell
  - pure functional language
  - statically typed
  - rich type system

What will we learn?

- Lisp/Scheme/Racket
  - simple syntax (directly matches λ-calculus)
  - dynamically typed
  - code-as-data (easy to write interpreters,...)
  - allows mutable data
- λ-calculus
- Haskell
  - pure functional language
  - statically typed
  - rich type system
  - strictly separates the pure core from the mutable shell

Suggested literature

[1] Harold Abelson and Gerald Jay Sussman and Julie Sussman: Structure and Interpretation of Computer Programs, MIT Press, 1996. `https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html`

Suggested literature

[1] Harold Abelson and Gerald Jay Sussman and Julie Sussman: Structure and Interpretation of Computer Programs, MIT Press, 1996. `https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html`

[2] Raul Rojas. A Tutorial Introduction to the Lambda Calculus. `http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`

Suggested literature

[1] Harold Abelson and Gerald Jay Sussman and Julie Sussman: Structure and Interpretation of Computer Programs, MIT Press, 1996. `https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html`

[2] Raul Rojas. A Tutorial Introduction to the Lambda Calculus. `http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`

[3] Graham Hutton: Programming in Haskell, Cambridge University Press, 2016.

Suggested literature

[1] Harold Abelson and Gerald Jay Sussman and Julie Sussman: Structure and Interpretation of Computer Programs, MIT Press, 1996. `https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html`

[2] Raul Rojas. A Tutorial Introduction to the Lambda Calculus. `http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`

[3] Graham Hutton: Programming in Haskell, Cambridge University Press, 2016.

Course webpage: `https://aicenter.github.io/FUP/`

Lisp/Scheme/Racket

Lisp/Scheme/Racket

- Lisp = List processor

Lisp/Scheme/Racket

- Lisp = List processor
- Scheme is a dialect of Lisp (such as Common Lisp, Clojure)

Lisp/Scheme/Racket

- Lisp = List processor
- Scheme is a dialect of Lisp (such as Common Lisp, Clojure)
- Scheme last standard from 2007 — The Revised6 Report on the Algorithmic Language Scheme (R6RS)

Lisp/Scheme/Racket

- Lisp = List processor
- Scheme is a dialect of Lisp (such as Common Lisp, Clojure)
- Scheme last standard from 2007 — The Revised6 Report on the Algorithmic Language Scheme (R6RS)
- Racket is another dialect based on R5RS (Scheme with batteries)

Lisp/Scheme/Racket

- Lisp = List processor
- Scheme is a dialect of Lisp (such as Common Lisp, Clojure)
- Scheme last standard from 2007 — The Revised6 Report on the Algorithmic Language Scheme (R6RS)
- Racket is another dialect based on R5RS (Scheme with batteries)
- DrRacket: `racket-lang.org`
  text editor + REPL (read-evaluate-print loop)

Racket's syntax

Racket program is a collection of expressions

Racket's syntax

Racket program is a collection of expressions

- Primitive expressions (literals, built-in functions)

```
"Hello World!"
```

Racket's syntax

Racket program is a collection of expressions

- Primitive expressions (literals, built-in functions)
  ```
  "Hello World!"
  ```
- Compound expressions (built by function composition)
  ```
  (cos (+ 1 2))
  ```

Racket's syntax

Racket program is a collection of expressions

- Primitive expressions (literals, built-in functions)

  ```
  "Hello World!"
  ```

- Compound expressions (built by function composition)

  ```
  (cos (+ 1 2))
  ```

- Definitions (introduce new names and functions)

  ```
  (define (square x) (* x x))
  ```

Racket's syntax

Racket program is a collection of expressions

- Primitive expressions (literals, built-in functions)
  ```
  "Hello World!"
  ```
- Compound expressions (built by function composition)
  ```
  (cos (+ 1 2))
  ```
- Definitions (introduce new names and functions)
  ```
  (define (square x) (* x x))
  ```
- Comments
  ```
  ; This is a one-line comment
  #|
      This is
      a block comment
  |#
  ```

Compound expressions

Compound expressions are built from primitive expressions by function composition.

Compound expressions

Compound expressions are built from primitive expressions by function composition.

Racket uses prefix notation. E.g.

$$\frac{xy^2 + 3}{x - 1}$$

Compound expressions

Compound expressions are built from primitive expressions by function composition.

Racket uses prefix notation. E.g.

$$\frac{xy^2 + 3}{x - 1}$$

```
(/ (+ (* x y y) 3)
   (- x 1))
```

Compound expressions

Compound expressions are built from primitive expressions by function composition.

Racket uses prefix notation. E.g.

$$\frac{xy^2 + 3}{x - 1}$$

```
(/ (+ (* x y y) 3)
   (- x 1))
```

Note how we don't have to worry about operator precedence!

Compound expressions

Compound expressions are built from primitive expressions by function composition.

Racket uses prefix notation. E.g.

$$\frac{xy^2 + 3}{x - 1}$$

```
(/ (+ (* x y y) 3)
   (- x 1))
```

Note how we don't have to worry about operator precedence!

S-expression

```
(fn arg1 arg2 ... argN)
```

Definitions

- Naming expressions

  `(define id exp)`

Definitions

- Naming expressions

```
(define id exp)
```

- Defining functions

```
(define (name arg1 ... argN)
   exp1
   ...
   expM)
```

Definitions

- Naming expressions

```
(define id exp)
```

- Defining functions

```
(define (name arg1 ... argN)
   exp1
   ...
   expM)
```

- Nested definitions

```
(define (name a1 ... aN)
   (define (fn b1 ... bM) <body-fn>)
   <body-using-fn>)
```

Racket's semantics

Racket program is, in fact, an expression.

Racket's semantics

Racket program is, in fact, an expression.

Its evaluation is the computation process represented by the program.

Racket's semantics

Racket program is, in fact, an expression.

Its evaluation is the computation process represented by the program.

The evaluation resembles simplifying expressions we know from math.

Racket's semantics

Racket program is, in fact, an expression.

Its evaluation is the computation process represented by the program.

The evaluation resembles simplifying expressions we know from math.

More precisely, we subsequently evaluate subexpressions until we end up with the expression's value.

Evaluation strategy

```
(define (square x) (* x x))
(square (+ 3 4))
```

Evaluation strategy

```
(define (square x) (* x x))
(square (+ 3 4))
```

```
(square (+ 3 4)) => (square 7) => (* 7 7) => 49
```

Evaluation strategy

```
(define (square x) (* x x))
(square (+ 3 4))
```

```
(square (+ 3 4)) => (square 7) => (* 7 7) => 49
```

```
(square (+ 3 4)) => (* (+ 3 4) (+ 3 4))
                 => (* 7 7) => 49
```

Evaluation strategy

```
(define (square x) (* x x))
(square (+ 3 4))
```

```
(square (+ 3 4)) => (square 7) => (* 7 7) => 49
```

```
(square (+ 3 4)) => (* (+ 3 4) (+ 3 4))
                 => (* 7 7) => 49
```

- Evaluation strategy defines the order of evaluating the expressions, influences program termination, not the result

Evaluation strategy

```
(define (square x) (* x x))
(square (+ 3 4))
```

```
(square (+ 3 4)) => (square 7) => (* 7 7) => 49
```

```
(square (+ 3 4)) => (* (+ 3 4) (+ 3 4))
                 => (* 7 7) => 49
```

- Evaluation strategy defines the order of evaluating the expressions, influences program termination, not the result
- Racket's strategy is strict (or eager) evaluates all arguments (left to right) before evaluating the function

Evaluation strategy

```
(define (square x) (* x x))
(square (+ 3 4))
```

```
(square (+ 3 4)) => (square 7) => (* 7 7) => 49
```

```
(square (+ 3 4)) => (* (+ 3 4) (+ 3 4))
                 => (* 7 7) => 49
```

- Evaluation strategy defines the order of evaluating the expressions, influences program termination, not the result
- Racket's strategy is strict (or eager) evaluates all arguments (left to right) before evaluating the function
- Evaluation of some syntactic forms is lazy if, cond, and, or

Conditional expressions

```
(if test-exp then-exp else-exp)
```

Conditional expressions

```
(if test-exp then-exp else-exp)
```

```
(if (> 0 1) 1 2) => 2
```

Conditional expressions

```
(if test-exp then-exp else-exp)

(if (> 0 1) 1 2) => 2

(if (< 0 1) 1 (+ 3 "a")) => 1
```

Conditional expressions

```
(if test-exp then-exp else-exp)

(if (> 0 1) 1 2) => 2

(if (< 0 1) 1 (+ 3 "a")) => 1

(cond [test-exp1 exp]
      [test-exp2 exp]
      ...
      [else exp])
```

Conditional expressions

```
(if test-exp then-exp else-exp)

(if (> 0 1) 1 2) => 2

(if (< 0 1) 1 (+ 3 "a")) => 1

(cond [test-exp1 exp]
      [test-exp2 exp]
      ...
      [else exp])

(cond [(odd? 12) 1]
      [(even? 12) 2]
      [else 3])     => 2
```

Basic data types

- Numbers: exact ½, inexact 3.14, complex $2 + 3i$
  `+, -, *, /, abs, sqrt, number?, <, >, =`

Basic data types

- Numbers: exact ½, inexact 3.14, complex $2 + 3i$
  `+, -, *, /, abs, sqrt, number?, <, >, =`
- Logical values: `#t, #f`
  `and, or, not, boolean?`

Basic data types

- **Numbers**: exact ½, inexact 3.14, complex $2 + 3i$
  `+, -, *, /, abs, sqrt, number?, <, >, =`
- **Logical values**: `#t, #f`
  `and, or, not, boolean?`
- **Strings**: `"abc"`
  `string?, substring, string-append`

Basic data types

- **Numbers**: exact ½, inexact 3.14, complex $2 + 3i$
  `+, -, *, /, abs, sqrt, number?, <, >, =`
- **Logical values**: `#t, #f`
  `and, or, not, boolean?`
- **Strings**: `"abc"`
  `string?, substring, string-append`
- **Characters**: `#\A, #\@`
  `char?, char->integer, integer->char, list->string, string->list`

Basic data types

- Numbers: exact ½, inexact 3.14, complex $2 + 3i$
  `+, -, *, /, abs, sqrt, number?, <, >, =`
- Logical values: `#t, #f`
  `and, or, not, boolean?`
- Strings: `"abc"`
  `string?, substring, string-append`
- Characters: `#\A, #\@`
  `char?, char->integer, integer->char,`
  `list->string, string->list`
- Other types:
  `symbol?, pair?, procedure?, vector?, port?`

Simple debugging

- Helper print-outs

```
(begin (displayln x)
       <do-work>)
```

Simple debugging

- Helper print-outs

```
(begin (displayln x)
       <do-work>)
```

- Tracing function calls and returns

```
(require racket/trace)
(trace fn)
(untrace fn)
```

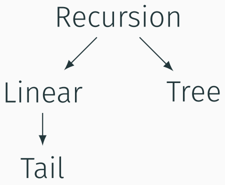
Recursion

Recursion

Recursive function calls itself in its body.

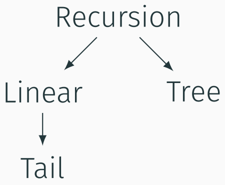
Recursion

Recursive function calls itself in its body.



Recursion

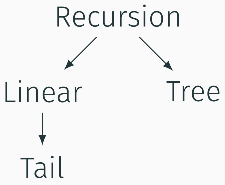
Recursive function calls itself in its body.



- **Linear**: makes one recursive call

Recursion

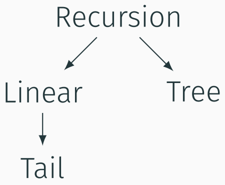
Recursive function calls itself in its body.



- **Linear**: makes one recursive call
- **Tree**: makes several recursive calls

Recursion

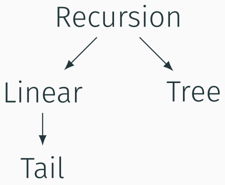
Recursive function calls itself in its body.

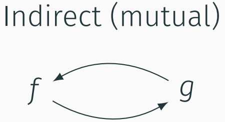


- **Linear**: makes one recursive call
- **Tree**: makes several recursive calls
- **Tail**: the result of the recursive call is the final result of the function

Recursion

Recursive function calls itself in its body.





- **Linear**: makes one recursive call
- **Tree**: makes several recursive calls
- **Tail**: the result of the recursive call is the final result of the function

Examples

```
(define (loop) (loop))
```

Examples

```
(define (loop) (loop))

(define (fact n)
    (if (<= n 1)
        1
        (* n (fact (- n 1)))))
```

Examples

```
(define (loop) (loop))

(define (fact n)
    (if (<= n 1)
        1
        (* n (fact (- n 1)))))

(fact 4) => (* 4 (fact 3))
         => (* 4 (* 3 (fact 2)))
         => (* 4 (* 3 (* 2 (fact 1))))
         => (* 4 (* 3 (* 2 (* 1 1)))) => 24
```

Examples

```
(define (loop) (loop))

(define (fact n)
    (if (<= n 1)
        1
        (* n (fact (- n 1)))))

(fact 4) => (* 4 (fact 3))
         => (* 4 (* 3 (fact 2)))
         => (* 4 (* 3 (* 2 (fact 1))))
         => (* 4 (* 3 (* 2 (* 1 1)))) => 24
```

Not space efficient. It needs $O(n)$ memory.

Example — Tail recursion

```
(define (fact n [acc 1])
    (if (<= n 1)
        acc
        (fact (- n 1) (* n acc))))
```

Example — Tail recursion

```
(define (fact n [acc 1])
    (if (<= n 1)
        acc
        (fact (- n 1) (* n acc))))
```

```
(fact 4) =  (fact 4 1)
         => (fact 3 4)
         => (fact 2 12)
         => (fact 1 24)
         => 24
```

Example — Tail recursion

```
(define (fact n [acc 1])
    (if (<= n 1)
        acc
        (fact (- n 1) (* n acc))))
```

```
(fact 4) =  (fact 4 1)
         => (fact 3 4)
         => (fact 2 12)
         => (fact 1 24)
         => 24
```

This needs $O(1)$ memory due to tail elimination.

Example – Tree recursion

Consider a tree-like fractal of a given size n and direction d in degrees generated by:

1. Draw a stick of size n in the direction d.

Example – Tree recursion

Consider a tree-like fractal of a given size n and direction d in degrees generated by:

1. Draw a stick of size n in the direction d.
2. Draw the fractal of size $n/2$ in the direction $d + 60$.

Example – Tree recursion

Consider a tree-like fractal of a given size n and direction d in degrees generated by:

1. Draw a stick of size n in the direction d.
2. Draw the fractal of size $n/2$ in the direction $d + 60$.
3. Draw the fractal of size $n/2$ in the direction $d - 60$.

Example – Tree recursion

Consider a tree-like fractal of a given size n and direction d in degrees generated by:

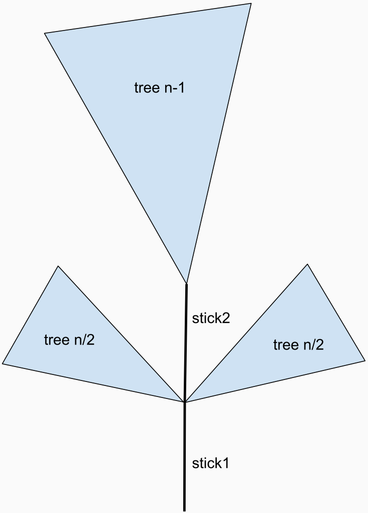
1. Draw a stick of size n in the direction d.
2. Draw the fractal of size $n/2$ in the direction $d + 60$.
3. Draw the fractal of size $n/2$ in the direction $d - 60$.
4. Draw a stick of size n in the direction d.

Example – Tree recursion

Consider a tree-like fractal of a given size n and direction d in degrees generated by:

1. Draw a stick of size n in the direction d.
2. Draw the fractal of size $n/2$ in the direction $d + 60$.
3. Draw the fractal of size $n/2$ in the direction $d - 60$.
4. Draw a stick of size n in the direction d.
5. Draw the fractal of size $n - 1$ in the direction $d + 5$.

Fractal example



Fractal example

To draw a picture, we use the library value-turtles.

Fractal example

To draw a picture, we use the library value-turtles.

Its functions operates on an image together with a position and direction of a turtle.

Fractal example

To draw a picture, we use the library value-turtles.

Its functions operates on an image together with a position and direction of a turtle.

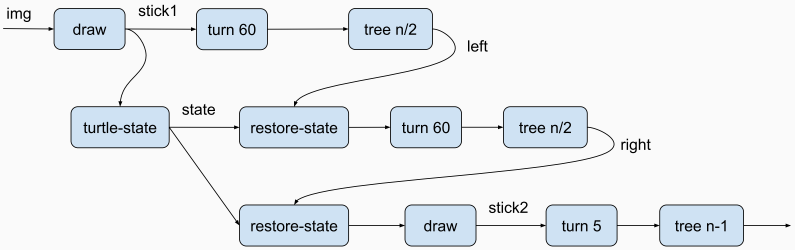
E.g. `(draw 100 img)`

Fractal example

To draw a picture, we use the library value-turtles.

Its functions operates on an image together with a position and direction of a turtle.

E.g. `(draw 100 img)`



What have we learned?

- A pure function always returns the same output on a fixed input and has no side effects.

What have we learned?

- A pure function always returns the same output on a fixed input and has no side effects.
- Make the pure part of a program as large as possible, keeping the code handling the state transparent and small.

What have we learned?

- A pure function always returns the same output on a fixed input and has no side effects.
- Make the pure part of a program as large as possible, keeping the code handling the state transparent and small.
- Functional languages handle iterative computations by recursion.

What have we learned?

- A pure function always returns the same output on a fixed input and has no side effects.
- Make the pure part of a program as large as possible, keeping the code handling the state transparent and small.
- Functional languages handle iterative computations by recursion.
- We classify recursive functions according to the number of recursive calls they make on linear-recursive and tree-recursive functions.

What have we learned?

- A pure function always returns the same output on a fixed input and has no side effects.
- Make the pure part of a program as large as possible, keeping the code handling the state transparent and small.
- Functional languages handle iterative computations by recursion.
- We classify recursive functions according to the number of recursive calls they make on linear-recursive and tree-recursive functions.
- Tail recursive functions are space efficient as they do not consume memory by making recursive calls.