Advanced Functional and Logic Programming

Lecture 5: Introduction to Functional Programming. The RACKET language

Mircea Marin
mircea.marin@e-uvt.ro

October 25, 2018

- Describe every computation as a request to **evaluate an expression**, and use the resulting value for something.

Main notions: function, value, expression.

- program: collection of function definitions in the *mathematical* sense
  - The value of a function call depends only on the values of the function arguments.
- expression = (typically) a combination of nested function calls.
- computation = evaluation of an expression $\Rightarrow$ a value.
- value = element of a datatype (string, integer, list, etc.) that can be
  - named
  - stored in a composite data (e.g., element of a list or vector)
  - passed as argument to a function call
  - returned as result of a function call

Characteristic features of FP

- **Variables are just names given to values**
- **There is no assignment** $\Rightarrow$ we can not change the value of a variable
  - $\Rightarrow$ we can not define repetitive computations by iteration.
  - $\Rightarrow$ we define repetitive computations by recursion
- **Functions are values** $\Rightarrow$ we can have
  - functions that take function arguments
  - functions that return functions as results
  - lists of functions, etc.

Example 1: Computation of $n!$ by recursion

We can not define `fact` by iteration

```
fact(int n)
  i=1; fact:=1;
  if(i<n)
    i:=i+1;
    fact:=fact*i;
 endif
 return fact
```

because there is no assignment in functional programming.
But we can define `fact` by recursion (pseudocode)

```
fact(int n)
  if (n==1)
     1
  else
     n*fact(n-1)
```

Example 2: A function with a function argument

`map(`f`,`L`)` takes as inputs a function $f : A \to B$ and a list

$$L = (a_1, a_2, \ldots, a_n) \quad \text{of elements from } A$$

and returns the list

$$(f(a_1), f(a_2), \ldots, f(a_n)) \quad \text{of elements from } B$$

Assume that

- `empty(L)` recognises if `L` is empty list
- `first(L)` returns the first element of a list
- `rest(L)` returns the list produced by removing the first element of `L`
- `prepend(e,L)` adds element `e` in front of list `L`

```
map(f,L)
  if (empty(L)) L
  else prepend(f(first(L)),map(f,rest(L)))
```

Why learn functional programming?

- It has a very simple model of computation:
  - Programs consist (mainly) of function definitions
  - Computation=evaluation of (nested) function calls
- We can define higher-order functions (functions that take functions as arguments and/or return functions as results)

  $\Rightarrow$ we can write highly modular and reusable code [Hughes:1989]
- According to [Thompson:1999]:
  - *"Functional languages provide a framework in which the crucial ideas of modern programming are presented in the clearest possible way. This accounts for their widespread use in teaching computing science and also for their influence on the design of other languages."*

How to use DrRacket to write and run functional programs

- DrRacket is an integrated development environment (IDE) for Racket, the current dialect of Scheme
- DrRacket is freely available to be installed on Windows, Linux, MacOS, etc.

  `https://racket-lang.org`

When started, DrRacket shows a window with two panels:

1. the definitions area, where you can start typing your own programs, save them for later use and run them.
2. the interactions area, where you can interact directly with the interpreter of RACKET.

- The first high-level programming language was Fortran (1957). Fortran is an imperative programming language.
- The second high-level programming language was Lisp (1958).
  - Designed by people interested in AI ("the science and engineering of making intelligent machines"); Lisp became the favoured language for AI research
  - Lisp is acronym for "**Lis**t **P**rocessing": Lists are used to represent both source code and composite data.

  Initial Lisp had no standard specification $\Rightarrow$ many dialects of Lisp appeared $\Rightarrow$ people were confused, and wanted a standardised and reliable version of Lisp
  - $\Rightarrow$ Common Lisp (ANSI 1994 standard): extensive libraries, ideal for writing commercial software
  - $\Rightarrow$ Scheme (IEEE 1990 standard): wonderful for educational purposes.

    The most recent dialect of Scheme is RACKET**.**

- They are strict programming languages
  - A language is strict if the evaluation of function calls proceeds as follows: First, we compute the values of the arguments, and next we call the function with the computed values.

Example

$4 + ((2 - 2) * (4 - 3))$ is the infix notation for the nested function call $+(4, *(0, -(4, 3)))$. The strict evaluation of this expression is:

$$+(4, *(\underline{-(2,2)}, -(4,3))) \rightarrow +(4, *(0, \underline{-(4,3)})) \rightarrow +(4, \underline{*(0,1)}) \rightarrow \underline{+(4,0)} \rightarrow \mathbf{4}$$

- All expressions are written in a peculiar syntax, called the parenthesised prefix notation (see next slide)

The parenthesised prefix notation

- Every function call $f(e_1, e_2, \ldots, e_n)$ is written as

  $(f \;\; pe_1 \;\; pe_2 \;\; \ldots \;\; pe_n)$

  where $pe_1, pe_2, \ldots, pe_n$ are the parenthesised prefix notations of $e_1, e_2, \ldots, e_n$
- Every other composite programming construct is of the form

  $(\textit{form-id} \;\; \ldots)$

  where *form-id* is the identifier of the programming construct.

Characteristics of the parenthesised prefix notation

- Every open parenthesis has a corresponding close perenthesis
- Instead of comma, we type whitespace (one or more blanks, newlines, tabs, etc.)

The parenthesised prefix notation
Examples

- `((2+7)/3-1)*(7-4)` is written as

  `(* (- (/ (+ 2 7) 3) 1) (- 7 4))`
- The parenthesised prefix notation of

  `if (n=1) 1 else n*fact(n-1)`

  is

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

Remark

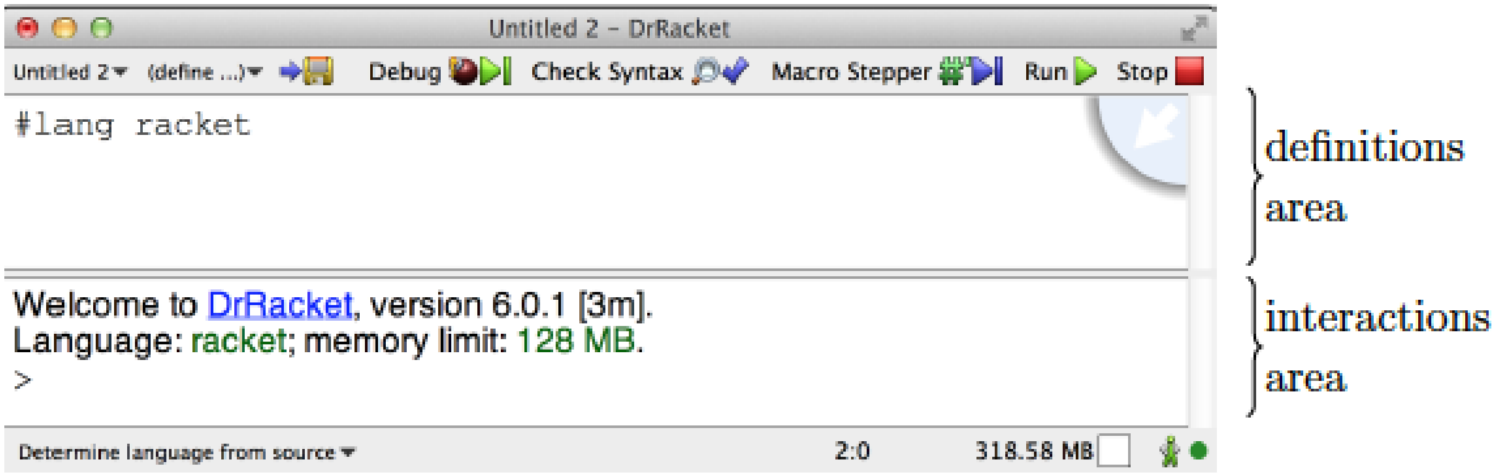
The parenthesised prefix notation of

`if` *cond* *expr*$_1$ `else` *expr*$_2$

is (`if` *cond-pe* *expr*$_1$*-pe* *expr*$_2$*-pe*)
where *cond-pe*, *expr*$_1$*-pe*, *expr*$_2$*-pe* are the parenthesised prefix notations of *cond*, *expr*$_1$, *expr*$_2$.

- Values are the simplest expressions: they evaluate to themself
- A value is either atomic or composite.
- Every value belongs to a datatype.
  - Datatypes with atomic values:
    - integer: `1` `-703` `12345678999999`
    - floating-point: `1.23` `3.14e+87`
    - string: `"abc"`
    - symbol: `'abc`
      (symbol values are preceded by the quote character')
    - boolean: `#t` (for truth) `#f` (for falsehood)
    - ...
- Some datatypes have composite values: pairs, lists, vectors, hash tables, etc.
  - A composite value is a collection of other values.
  - Composite values and datatypes will be described in Lecture 2.

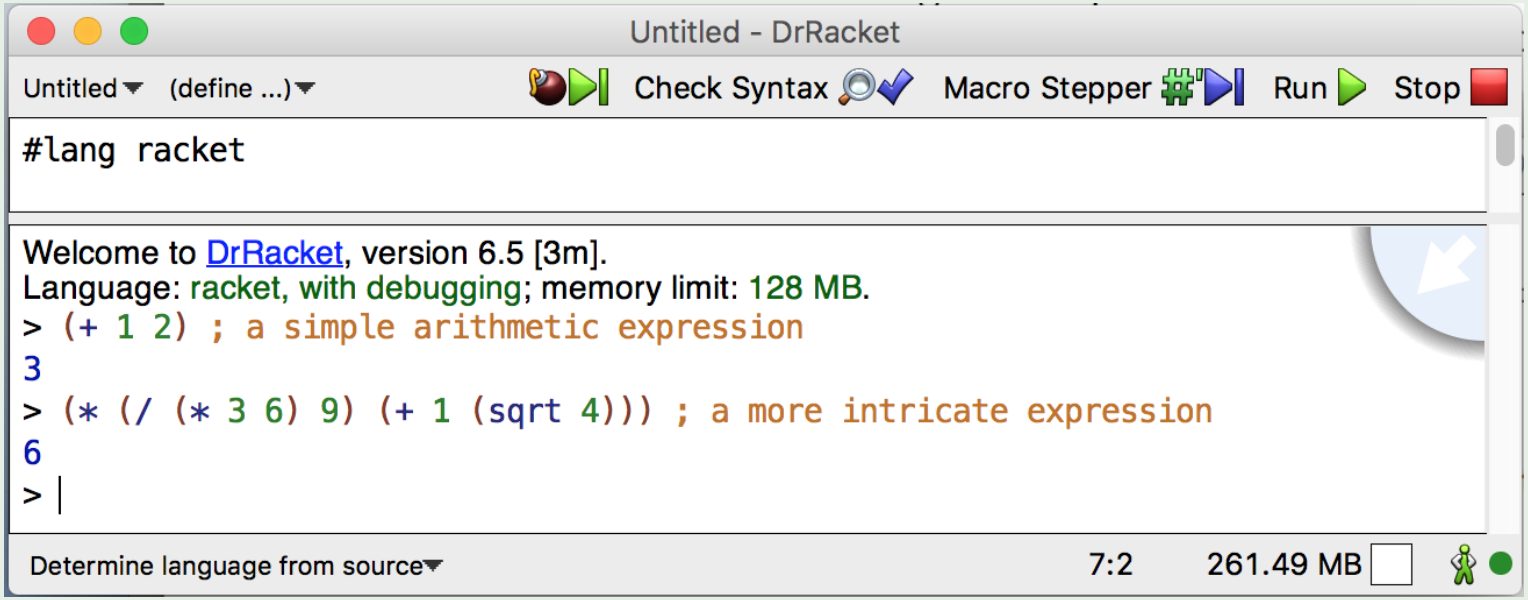
Interacting with DrRacket
The Read-Eval-Print loop (REPL)



In the interactions area, at the input prompt >

- Type in an expression *e* in parenthesised prefix notation, and press `Enter`
- *e* will be **read**, **evaluated**, and the resulted value will be **printed** on the next line in the interactions area.

Example



- the nesting of parentheses clarifies the order in which the function applications should be performed
- the semicolon ; starts a comment (highlighted with brown) that extends to the end of the line
  - comments are ignored by the interpreter

Expressions and definitions

The interpreter of RACKET recognises two kinds of input forms:

Expressions: An expression *expr* is read, evaluated, and its value is printed in the interactions area.

Definitions: A definition is of the form

`(define` *var* *expr*`)`

Definitions are interpreted as follows:

- *expr* is evaluated, and its value is assigned to variable *var*.
- Afterwards, *var* can be used to refer to the value of *expr*.

Example (Compute $\sqrt{x^2 + y^2}$ for $x = 2$ and $y = 3$)

```
> (define x 2)
> (define y 3)
> (sqrt (+ (* x x) (* y y)))
3.605551275463989
```

Function definitions

The meaning of an expression

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

is "the function which, for the values of arguments $x_1, \ldots, x_n$, computes the value of *body*."

- The evaluation of this expression creates a function, which is a value that can be assigned to a variable.

Example (the factorial function)

```
> (define fact
    (lambda (n)
      (if (or (= n 0) (= n 1))
          1
          (* n (fact (- n 1))))))
> (fact 5)  ; compute 5!
120
```

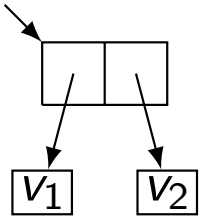
Composite datatypes

Every composite datatype has:

- recognizers = boolean functions that recognize values of that type.
- constructors = functions that build a composite value from component values
- selectors = functions that extract component values from a composite vulue
- utility functions = useful functions that operate on//with composite values
- A specific internal representation that affects the efficiency of the operations on them

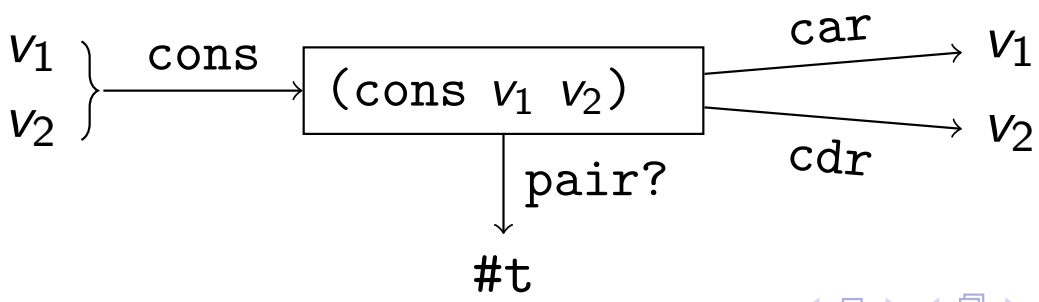
Pairs

- The simplest container of two values
  - constructor: `(cons` v_1 v_2`)`
  - internal representation: a `cons-cell` that stores pointers to the internal representations of v_1 and v_2



  - `(cons?` p`)`: returns `#t` if the value of p is a pair, and `#f` otherwise.
  - selectors
    - `(car` p`)`: returns the first component of pair p
    - `(cdr` p`)`: returns the second component of pair p

Diagrammatically, these operations behave as follows:



Operations on pairs
Examples

```
> (define p (cons 1 "a"))
> p
'(1 . "a")
> (pair? p)
#t
> (car p)
1
> (cdr p)
"a"
```

```
> (define q (cons 'a 'b))
> q
'(a . b)
> (car q)
'a
```

Remark

We can nest pairs to any depth to store many values in a single structure:

```
> (cons (cons (1 'a) "abc"))
'((1 . a) . "abc")
> (cons (cons 'a 1) (cons 'b (cons #t "c")))
'((a . 1) b #t . "c")
```

The printed form of pairs

RACKET applies repeatedly two rules to reduce the number of quote characters(') and parentheses in the printed forms:

rule 1: `(cons` v_1 v_2`)` is replaced by

`'(`w_1 `.` w_2`)`

where w_1, w_2 are the printed forms of v_1, v_2 from which we remove the preceding quote, if any. **There is space before and after the dot character in the printed form.**

rule 2: Whenever there is a dot character before a parenthesised expression, remove the dot character and the open/close parentheses.

Printed form of pairs
Example

```
> (cons (cons 'a 1) (cons 'b (cons #t (cons "c" 'd))))
'((a . 1) b #t "c" . d)
```

The printed form is obtained as follows:

- Apply rule 1 ro reduce the number of quote characters $\Rightarrow$ the form `'((a . 1) . (b . (#t . ("c" . d))))`
- Apply repeatedly rule 2 to eliminate dots and open/close parentheses:

```
'((a . 1) . (b . (#t . ("c" . d))))→'((a . 1) b . (#t . ("c" . d)))
'((a . 1) b . (#t . ("c" . d)))→'((a . 1) . (#t "c" . d))
'((a . 1) . (#t "c" . d))→'((a . 1) b #t "c" . d)
```

The final form is the printed form:

```
'((a . 1) b #t "c" . d)
```

We can input directly the printed forms, which are usually much shorter to write than combinations of nested `cons`-es:

Example

Instead of `(cons (cons 'v11 'v12) (cons 'v21 'v22))` we can type `'((v11 . v12) v21 . v22)`:

```
> (define p '((v11 . v12) v21 . v22))
> p
'((v11 . v12) v21 . v22)
> (car p)                    > (cdr p)
'(v11 . v12)                 '(v21 . v22)
> (car (car p))              > (cdr (car p))
'v11                         'v12
> (car (cdr p))              > (cdr (cdr p))
'v21                         'v22
```

Selectors for nested pairs

The selection of an element deep in a nested pair is cumbersome:

```
> (define p '(a ((x . y) . c) d))
```

To select the **second** of the **first** of the **first** of the **second** component of `p`, we must type

```
> (cdr (car (car (cdr p))))
'y
```

We can use the shorthand selector `cdaadr`:

```
> (cdaadr p)
'y
```

Other shorthand selectors: $\mathtt{c}x_1 \ldots x_p\mathtt{r}$ where $x_1, \ldots, x_p \in \{\mathtt{a}, \mathtt{d}\}$ and $1 \leq p \leq 4$ (max. 4 nestings)

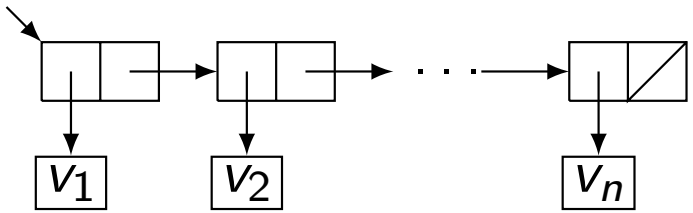
A **recursive** datatype with two constructors:

- `null`: the empty list
- `(cons` v l`)`: the list produced by placing the value v in front of list l.

If $n \geq 1$, the list of values $v_1, v_2, \ldots, v_n$ is

`(cons` v_1 `(cons` v_2 `...` `(cons` v_n `null)...))`

with the **internal representation**



REMARK: The internal representation of a list with n values $v_1, \ldots, v_n$ consists of n cons-cells linked by pointers.

Printed form of lists

```
> null
'()   ; the printed form of the empty list
```

All non-empty lists are pairs, and their printed form is computed like for pairs.

Example

```
> (cons 'a
    (cons 'b
        (cons 'c (cons (cons 'd null)
            null))))
'(a b c (d))
```

This printed form is obtained by applying repeatedly rule 2 to the form `'( a. (b . (c . ((d . ()) . ()))))`

A simpler constructor for the list of values $v_1, v_2, \ldots, v_n$:

> (list v_1 v_2 ... v_n)

Selectors:

- (car *lst*) selects the first element of the non-empty list *lst*
- (cdr *lst*) selects the tail of the non-empty list *lst*
- (list-ref *lst* *k*) selects the element at position *k* of *lst*

  NOTE: The elements are indexed starting from position 0.

Example

```
> (list 'a #t "bc" 'd)
'(a #t "bc" 'd)
> (list '() 'a '(b c))
'(() a (b c))
> (list-ref '(1 (2) 3) 1)
'(2)
> null
'()
> (list-ref '(1 2 3) 0)
1
```

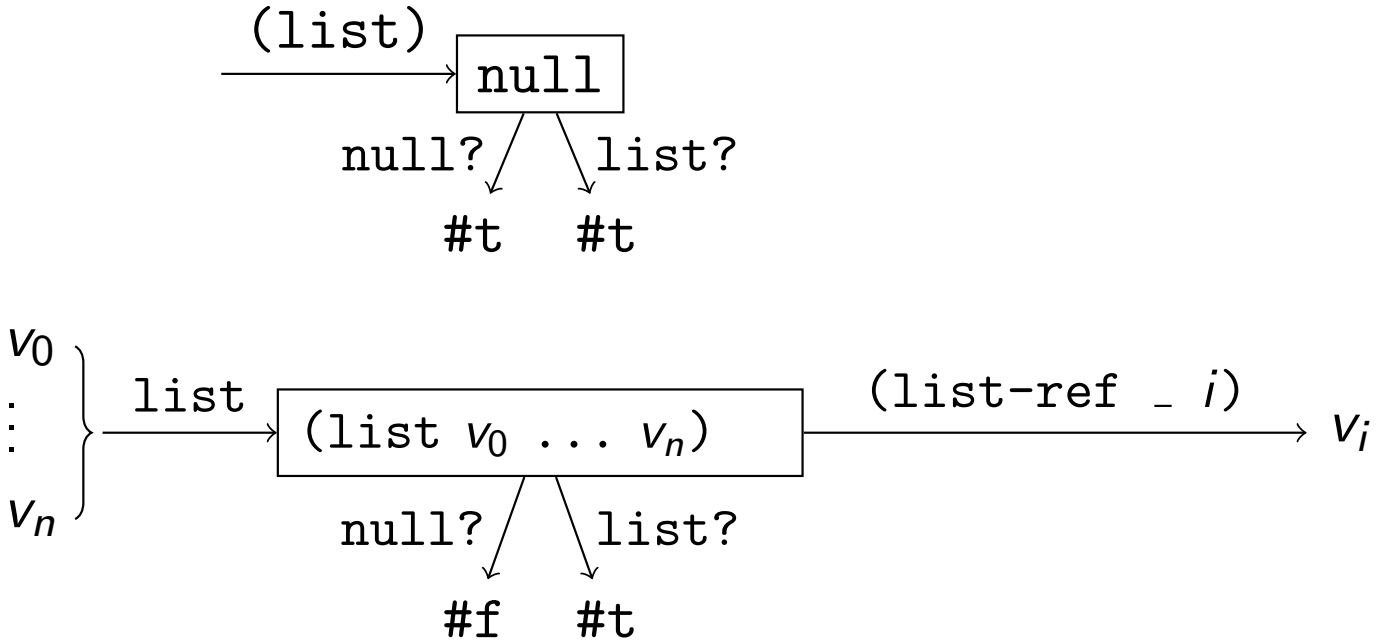
List recognizers

- (`list?` *lst*) recognizes if *lst* is a list.
- (`null?` *lst*) recognizes if *lst* is the empty list.

Example

```
> (define lst '(a b c d))
> (list? lst)
#t
> (car lst)
'a
> (cdr lst)
'(b c d)
> (list-ref lst 0)
'a
> (list-ref lst 1)
'b
```

List operations
Diagrammatic representation of their behavior



Utility functions on lists

(`length` *lst*) returns the length (=number of elements) of *lst*

```
> (length '(1 2 (3 4)))
3
```

```
> (length '())
0
```

(`append` lst_1 ... lst_n) returns the list produced by joining lists $lst_1, \ldots, lst_n$, one after another.

```
> (append '(1 2 3) '(a b c))
'(1 2 3 a b c)
```

(`reverse` *lst*) returns the list *lst* with the elements in reverse order:

```
> (reverse '(1 2 3))
'(3 2 1)
```

Operations on lists (1)

apply and filter

- If `f` is a function and `lst` is a list with component values $v_1, \ldots, v_n$ in this order, then
  `(apply f lst)`
  returns the value of the function call `(f` v_1 `...` v_n`)`.
- If `p` is a boolean function and `lst` is a list, then
  `(filter p lst)`
  returns the sublist of `lst` with elements `v` for which `(p v)` is true.

Examples

```
> (apply + '(4 5 6))     ; compute 4+5+6
15
> (filter symbol? '(1 2 a #t "abc" (3 4) b))
'(a b)
> (filter number? '(1 2 a #t "abc" (3 4) b))
'(1 2)
```

Operations on lists (2)

map

If `f` is a function and `lst` is a list with component values $v_1, \ldots, v_n$ in this order, then

```
(apply f lst)
```

returns the list of values $w_1, \ldots, w_n$ where every w_i is the value of `(f` v_i`)`

Example

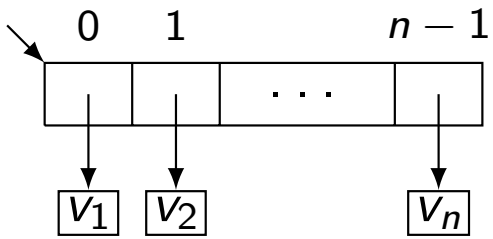
```
> (map (lambda (x) (+ x 1)) '(1 2 3 4))
'(2 3 4 5)
> (map list? '(1 2 () (3 4) (a . b)))
'(#f #f #t #t #f)
```

Vectors

A composite datatype of a fixed number of values.

Constructors:

- (`vector` v_0 v_1 ... v_{n-1})
  constructs a vector with n component values, indexed from 0 to $n-1$, and internal representation



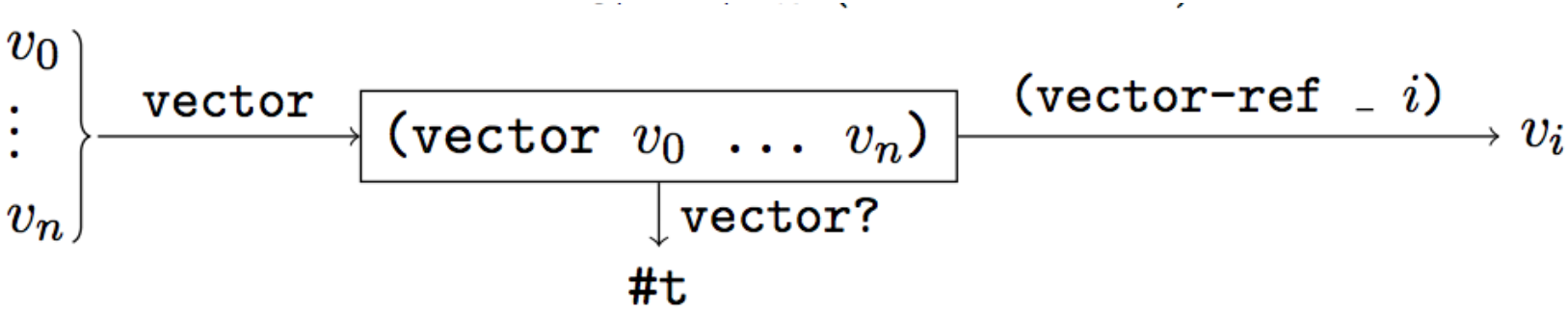
- (`make-vector` n v)
  returns a new vector with n elements, all equal to v.

Recognizer: `vector?`

Selectors: (`vector-ref` `vec` `i`)

returns the component value with index `i` of the vector `vec`.

Operations on vectors



Example

```
> (define vec (vector "a" '(1 . 2) '(a b)))
> (vector? vec)
#t
> (vector-ref vec 1)
'(1 . 2)
> (vector-ref vec 2)
'(a b)
> (vector-length vec) ; compute the length of vec
3
```

Printed form of vectors

The printed form of a vector with component values $v_0, v_1, \ldots, v_n$ is

'#(w_0 w_1 ... w_n)

where w_i is the printed form of v_i from which we remove the preceding quote character, if any.

Examples

```
> (vector 'a  #t  '(a . b)  '(1 2 3))
'#(a #t (a . b) (1 2 3))
> (vector 'a (vector 1 2) (vector) "abc")
'#(a #(1 2) #() "abc")
> (make-vector 3 '(1 2))
'#((1 2) (1 2) (1 2))
```

The printed forms of vectors are also valid input forms:

```
> '#(1 2 3)
'#(1 2 3)
```

```
> (vector? '#(1 2 3))
#t
```

The `void` datatype

Consists of only one value, `'#<void>`:

- The recognizer is `void?`
- Attempts to input `'#<void>` directly will raise a syntax error:

```
> '#<void>
read: bad syntax `#<'
```

- We can obtain `'#<void>` indirectly, as the value of the function call `(void)`:

```
> (list 1 (void) 'a)
'(1 #<void> a)
> (void? (void))
#t
```

- Usually, `'#void` is not printed

```
> (void)  ; nothing is printed
```

Block-structured programming
What is this?

- Block: sequence of **definitions**, **expressions** and **other blocks** that ends with an expression.
  - Representation:

    $comp_1$
    ...
    $comp_n$
    $expr$

    where the block components $comp_1, \ldots, comp_n$ are definitions, expressions, or blocks.
  - NOTE: blocks can be nested.
- A block-structured language interprets a block as a single expression:
  - Every variable declaration has a lexical scope: the textual portion of code where the name refers to that declaration.
    - The scope of a variable declaration is the block where the variable is declared. We also say that the variable is local to that block.
    - Block variables are visible only in the block where they are defined.
    - Nested block may shadow each other's declarations.

Block-structured programming with Racket

Racket is block-structured: We can evaluate the standalone block

$$\boxed{\begin{array}{l} comp_1 \\ \ldots \\ comp_n \\ expr \end{array}}$$

with the special form

(`local` `[ ]` $comp_1$ `...` $comp_n$ $expr$)

Remark

(`println` *expr*)

prints the value of *expr* on a new line, and returns the value `#<void>`.

- We will use `println` to illustrate how block-structured evaluation works (see next slide →)

Block-structured programming
Illustrated example

```
> (local [ ]
    (define x 1)
    (local [ ]
      (define x 2)          ; x=2 shadows x=1
      (define y 3)
      (println (+ x y)))    ; print the value of x+y for x=2,y=3
    (local [ ]
      (define y 4)
      (define z 5)
      (println (+ x y z))) ; print the value of x+y+z for x=1,y=4,z=5
    (+ x 2))                ; return the value of x+2 for x=1
5
10
3
```

Remark: This is a block with two sub-blocks

Special forms with blocks

- The `lambda`-form for function definitions:

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

- The `cond`-form

  `(cond [`$test_1$ $block_1$`]`
  ...
  `[`$test_n$ $block_n$`])`

  evaluates $test_1, test_2, \ldots$ in this order until it finds the first $test_i$ whose value is true, and returns the value of $block_i$.

  - If all expressions $test_i$ evaluate to `#f`, the value of the `cond`-form is `#<void>`

Remarks

In RACKET, any value different from `#f` acts as true. E.g.:

- `"abc"`, `'abc`, `null`, `0`, `'(1 2)`, `#t` are true values.
- `#f` is the only value which is not true.

Special forms with blocks

Example: the cond-form

Let's define the numeric function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$

$$f(x,y) = \begin{cases} \frac{(y-x^2)}{1+\sqrt{y-x^2}} & \text{if } x^2 < y, \\ (\sqrt{|x|}+y)+(\sqrt{|x|}+y)^2 & \text{if } x^2 \geq y. \end{cases}$$

```
(define (f x y)
 (define z (* x x))
 (cond [(< z y)
        (define (u (- y z)))
        (/ u (+ 1 (sqrt u)))]
       [(>= z y)
        (define u (+ (sqrt (abs x)) y))
        (+ u (* u u))]))
```

Advice: Avoid doing the same computation repeatedly! You can do so by computing the intermediary values $\mathtt{z} = \mathtt{x}^2$ and

- $\mathtt{u} = \mathtt{y} - \mathtt{z}$ if $\mathtt{z} < \mathtt{y}$,
- $\mathtt{u} = \sqrt{|\mathtt{x}|} + \mathtt{y}$ if $\mathtt{z} \geq \mathtt{y}$.

Other useful conditional forms

`when`, `unless`, `if`

- `(when` *test block*`)`
  behaves the same as

  `(cond [`*test block*`])`
- `(unless` *test block*`)`
  behaves the same as

  ```
  (cond [test (void)]
        [#t block])
  ```

Remark

The branches of the `if`-form can not be blocks: they must be expressions:

`(if` *test* $expr_1$ $expr_2$`)`

The `let` form

```
(let ([var_1 expr_1]
      ...
      [var_n expr_n])
    block)
```

is evaluated as follows:

1. $expr_1, \ldots, expr_n$ are evaluated to values $v_1, \ldots, v_n$.
2. The definitions $var_1 = v_1, \ldots, var_n = v_n$ are made local to *block*.
3. *block* is evaluated, and its value is returned as final result.

Remark

This special form is equivalent to

```
((lambda (var_1 ... var_n) body) expr_1 ... expr_n)
```

The `let` form

Examples

```
> (let ([x 5])
     (let ([x 2]      ; binds x to 2
           [y x])     ; binds y to the value of the outer x, which is 5
          (+ x y)))   ; computes the value of 2+5
7
> (let ([x 5])
     (let ([x 2]        ; binds x to 2
           [y x])       ; binds y to the value of the outer x, which is 5
          (define x 1)  ; this binding shadows the outer binding of x to 2
          (+ x y)))     ; computes the value of 1+5
6
```

□

The `let*` form

```
(let* ([var_1  expr_1]
       ...
       [var_n  expr_n])
      block)
```

- Similar with the `let` form, but with the following difference:
  - The scope of every local definition

    $[var_i \;\; expr_i]$

    is $expr_{i+1}, \ldots, expr_n$, and *block*.

Remark

This special form is equivalent to

```
(...(lambda (var_1)
      ...
       (lambda (var_n) body) expr_n) ... expr_1)
```

The `let*` form

Example

```
> (let* ([x 1]              ; binds x to 1
         [y (+ x 1)])       ; binds y to the value of (+ x 1), which is 2
        (+ y x))            ; computes the value of 2+1
3
```

Note that the following expression can not be evaluated

```
> (let ([x 1]               ; binds x to 1
        [y (+ x 1)])        ;x  is undefined here
       (+ y x))
x: unbound identifier in module in: x
```

The letrec form

```
(letrec ([var_1 expr_1]
         ...
         [var_n expr_n])
        block)
```

Similar with the `let*` form, but with the following difference:

- The scope of every local definition

  $(var_i \ \ expr_i)$

  is $expr_1, \ldots, expr_n$, and *block*.

Thus, $var_1, \ldots, var_n$ can depend on each other.

Example

```
> (letrec ([is-even
              (lambda (n) (if (= n 0) #t (is-odd  (- n 1))))]
           [is-odd
              (lambda (n)(if (= n 0)  #f (is-even (- n 1))))])
        (is-even 5))
#f
```

The special forms and and or

They are **not** functions, they are identifiers of special forms:

- (and e_1 e_2 ... e_n)
  evaluates $e_1, e_2, \ldots$ in this order, until it finds the first e_i whose value is #f
  - If all e_i have non-#f values, return the value of e_n.
  - Otherwise, return #f
- (or e_1 e_2 ... e_n)
  evaluates $e_1, e_2, \ldots$ in this order, until it finds the first e_i whose value is true.
  - If all e_i have value #f, return #f.
  - Otherwise, return the value of e_i.

```
> (and)
#t
> (and (= 1 2) (expt 2 50))
#t

> (or)
#f
> (or (= 1 2) (expt 2 50))
1125899906842624
```

```
> (and (< 1 2) 3 'abc)
'abc
> and
and: bad syntax in: and

> (or (< 1 2) 3 'abc)
#t
> or
or: bad syntax in: or
```

The evaluation process

Consider the expression

```
> (f x 3)
```

The evaluation process

Consider the expression

```
> (f x 3)
```

Question: How does RACKET know how to compute the value of `(f x 3)`?

The evaluation process

Consider the expression

```
> (f x 3)
```

Question: How does RACKET know how to compute the value of `(f x 3)`?

Answer: It must know the following:

1. where to find the values of the variables `f` and `x`.
   - The values of the variables are stored in a global data structure, called environment
2. the rules of evaluation.

The evaluation process

Consider the expression

```
> (f x 3)
```

Question: How does RACKET know how to compute the value of `(f x 3)`?

Answer: It must know the following:

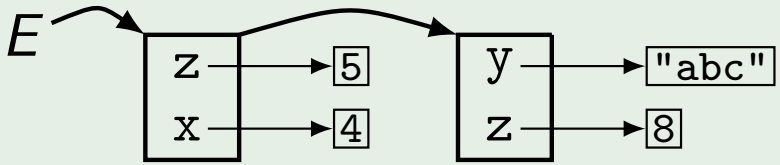
1. where to find the values of the variables `f` and `x`.
   - The values of the variables are stored in a global data structure, called environment
2. the rules of evaluation.

- The evaluation of an expression *expr* happens in the presence of an environment E which must provide values for the all the variables in *expr*

Environments

- An environment is a stack of frames.
- A frame is a table which stores values for variables.

Example (Environment E with two frames)



- The first frame is the top frame.
- **Variable lookup**: $E(var)$ is the value of *var* found in the first frame, from top to bottom (or left to right) which contains a value for *var*:

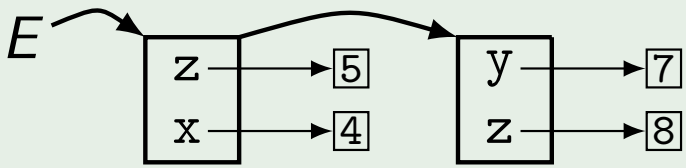
  $E(\mathtt{x}) = 4$, $E(\mathtt{y}) = \mathtt{"abc"}$, $E(\mathtt{z}) = 5$
  $E(\mathtt{t})$ is not defined.
  The binding $\mathtt{z} \mapsto 8$ is shadowed by the binding $\mathtt{z} \mapsto 5$ in the top frame.

The evaluation of expressions

The value of an expression *expr* in an environment E is computed in two steps:

1. All variables x in *expr* are replaced with $E(x)$
2. We compute the value of the new expression, using the rules of evaluation.

Example



The value of `(+ x (* y z))` in E is computed as follows:

`(+ x (* y z))` $\rightarrow$ `(+ 4 (* 7 5))` $\rightarrow$ `(+ 4 35)` $\rightarrow$ `39`

The interpretation of definitions

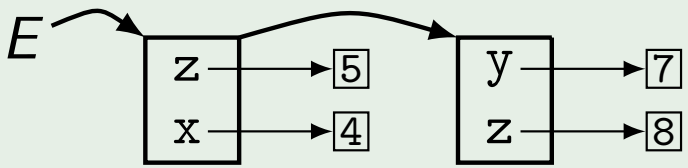
When the interpreter reads a definition

`(define` *var expr*`)`

in an environment E, it does the following:

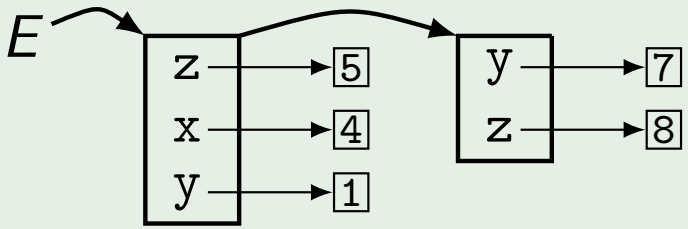
1. It computes the value v of *expr* in E
2. It adds the binding $var \mapsto v$ to the top frame of E.

Example




```
> (+ x (* y z))
39
```

The definition `(define y 1)` changes E to be




```
> (+ x (* y z))
9
```

User-defined functions

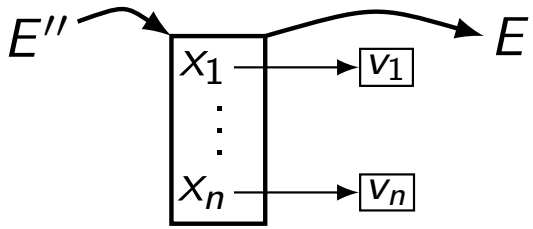
- The value of (`lambda` (x_1 ... x_n) *block*) in an environment E is the pair

  $\langle$(`lambda` (x_1 ... x_n) *block*), $E\rangle$

- User-defined function calls are evaluated as follows:
  If `f` is a user-defined function with value

  $\langle$(`lambda` (x_1 ... x_n) *block*), $E\rangle$

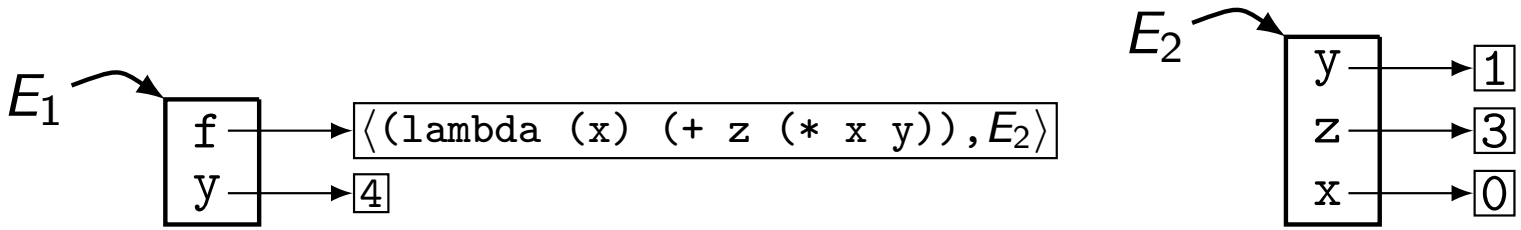
  the value of (`f` e_1 ... e_n) in E' is computed as follows:
  - compute the values $v_1, \ldots, v_n$ of $e_1, \ldots, e_n$ in E'
  - create the temporary environment

    and compute v =the value of *block* in E''
  - return v as the value of (`f` e_1 ... e_n) in E'.

The evaluation of function calls
Illustrated example

Consider the environments E_1 and E_2 where

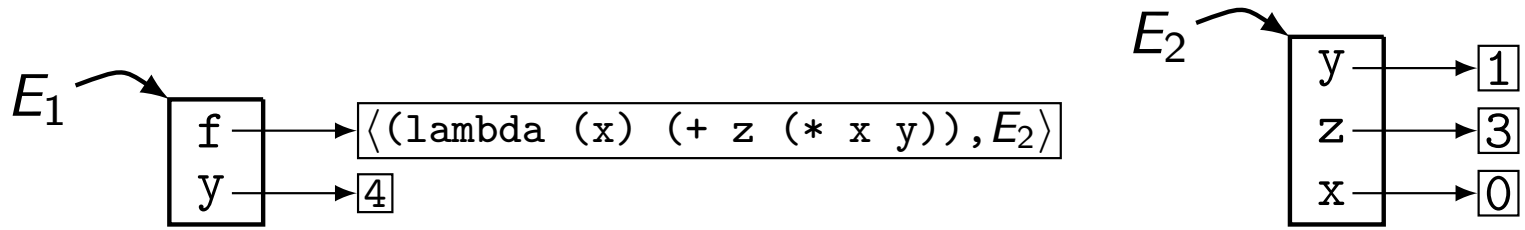


What is the value of `(f y)` in E_1?

The evaluation of function calls

Illustrated example

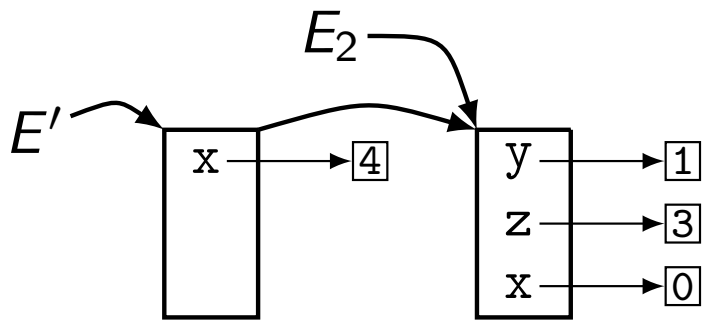
Consider the environments E_1 and E_2 where



What is the value of `(f y)` in E_1?

`(f y)` in E_1 $\rightarrow$ `(f 4)` in E_1 $\rightarrow$ `(+ z (* x y))` in E'

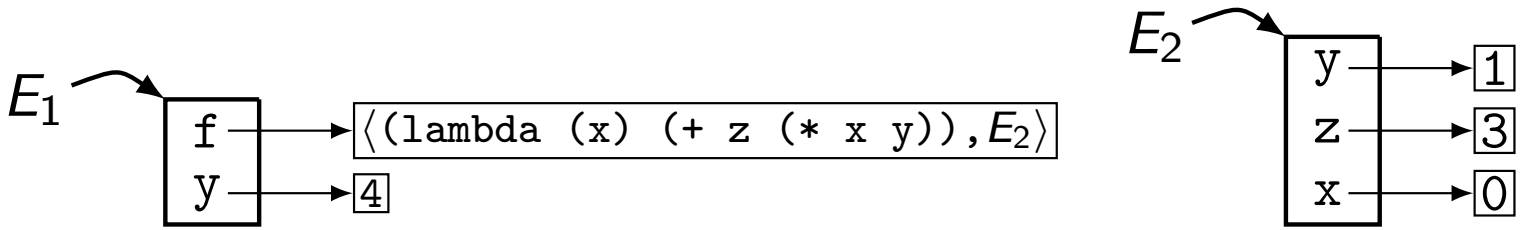
where




```
> (+ z (* x y)) in E'
12
```

The evaluation of function calls
Illustrated example

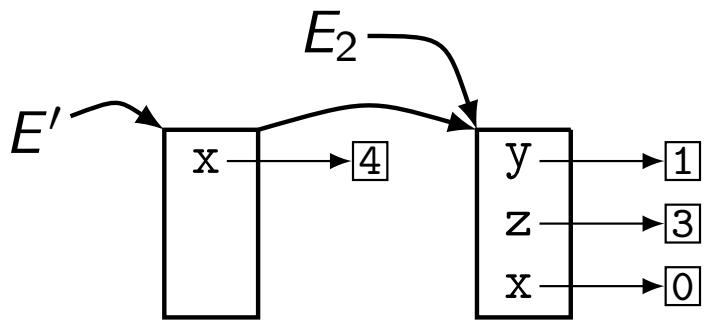
Consider the environments E_1 and E_2 where



What is the value of `(f y)` in E_1?

`(f y)` in E_1 $\rightarrow$ `(f 4)` in E_1 $\rightarrow$ `(+ z (* x y))` in E'

where

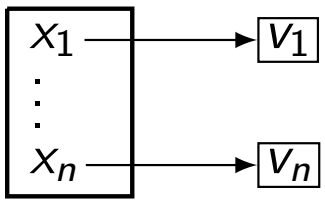



```
> (+ z (* x y)) in E'
12
```

$\Rightarrow$ the value of `(f y)` in E_1 is 12.

More about function values
Remarks

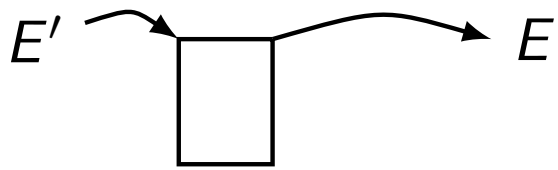
- A function value f created by the evaluation of
  `(lambda (`x_1 ... x_n`)` *block*`)`
  in an environment E is the pair
  $\langle$`(lambda (`x_1 ... x_n`)` *block*`)`$, E\rangle$
  - The first component is the textual definition of the function
  - The second component is the environment where f was created.
- The evaluation of $(f\ v_1\ \ldots\ v_n)$ in any environment E' is reduced to the evaluation of *block* in the environment E where f was defined, extended with the top frame



$\Rightarrow$ during the evaluation of the body of f, $x_1, \ldots, x_n$ have values $v_1, \ldots, v_n$.

The evaluation of blocks

The evaluation of *block* in E is reduced to the top-down evaluation of the content of *block* in the extension E' of E with an empty first frame:



- The top frame of E' will be filled with bindings for the local variables in *block*

Recursion

What is recursion?

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.
- In functional programming
  - A function is recursive when it calls itself directly or indirectly.
  - A data structure is recursive if it is defined in terms of itself.

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.
- In functional programming
  - A function is recursive when it calls itself directly or indirectly.
  - A data structure is recursive if it is defined in terms of itself.

Why learn recursion?

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.
- In functional programming
  - A function is recursive when it calls itself directly or indirectly.
  - A data structure is recursive if it is defined in terms of itself.

Why learn recursion?

- New way of thinking

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.
- In functional programming
  - A function is recursive when it calls itself directly or indirectly.
  - A data structure is recursive if it is defined in terms of itself.

Why learn recursion?

- New way of thinking
- Powerful programming tool

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.
- In functional programming
  - A function is recursive when it calls itself directly or indirectly.
  - A data structure is recursive if it is defined in terms of itself.

Why learn recursion?

- New way of thinking
- Powerful programming tool
- Divide-and-conquer paradigm

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.
- In functional programming
  - A function is recursive when it calls itself directly or indirectly.
  - A data structure is recursive if it is defined in terms of itself.

Why learn recursion?

- New way of thinking
- Powerful programming tool
- Divide-and-conquer paradigm

Many computations and data structures are naturally recursive

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.
- In functional programming
  - A function is recursive when it calls itself directly or indirectly.
  - A data structure is recursive if it is defined in terms of itself.

Why learn recursion?

- New way of thinking
- Powerful programming tool
- Divide-and-conquer paradigm

Many computations and data structures are naturally recursive

- Recursive computations: Euclid's gcd algorithm, quicksort, etc.

Recursion

What is recursion?

- Technique that allows us to break a problem into one or more subproblems that are similar to the initial problem.
- In functional programming
  - A function is recursive when it calls itself directly or indirectly.
  - A data structure is recursive if it is defined in terms of itself.

Why learn recursion?

- New way of thinking
- Powerful programming tool
- Divide-and-conquer paradigm

Many computations and data structures are naturally recursive

- Recursive computations: Euclid's gcd algorithm, quicksort, etc.
- Recursive data structures: linked lists, file directories, etc.

Recursive function definitions

General structure

- A simple base case (or base cases): a terminating scenario that does 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.
  - To ensure termination of the computation, the reduction process should eventually lead to base case computations.

- A simple base case (or base cases): a terminating scenario that does 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.
  - To ensure termination of the computation, the reduction process should eventually lead to base case computations.

Classic recursive functions:

1. Factorial function
2. Fibonacci function
3. Ackermann function
4. Euclid's Greatest Common Divisor (GCD) function

How to write a recursive definition?

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.

Example: The factorial function

$0! := 1$, and $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$ if $n > 0$.

- The recursive thinking approach:

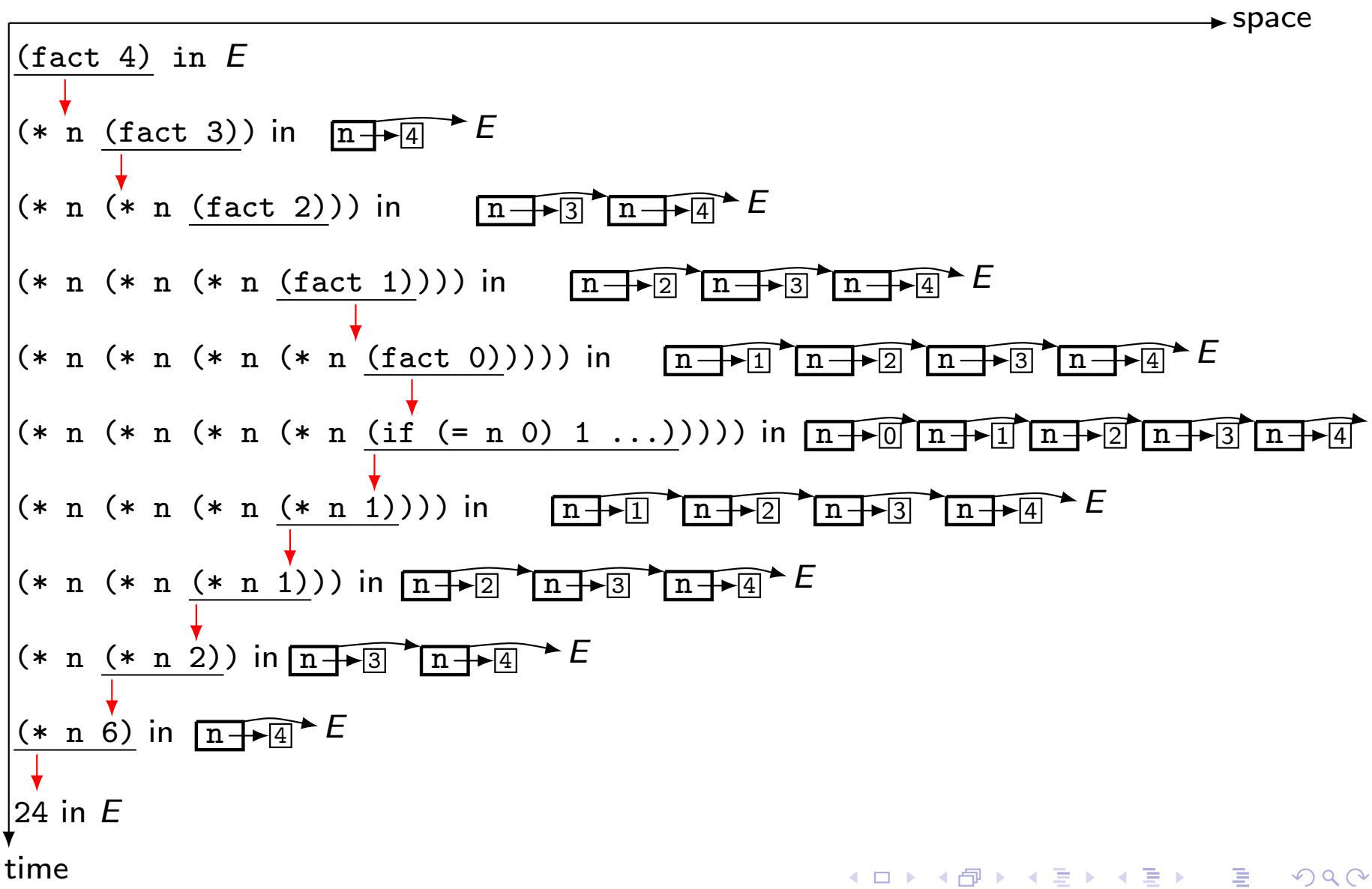
Mathematical definition	In Racket
$n! := \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$	`(define (fact n)` `  (if (= n 0) 1 (* n (fact (- n 1)))))`

  - The recursive case of `(fact n)` when `n` > 0 asks for a simpler computation: `(fact (- n 1))`
    In this case, "simpler" means "smaller value of the function call argument"

The factorial function

Time and space complexity of computation

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



The factorial function

Time and space complexity of computation

The computation of `(fact` n`)` has

time complexity $2 \cdot (n + 1) = O(n)$

space complexity $O(n)$: the maximum number of frames added to E is $n + 1$

The factorial function

Time and space complexity of computation

The computation of `(fact` n`)` has

time complexity $2 \cdot (n+1) = O(n)$

space complexity $O(n)$: the maximum number of frames added to E is $n+1$

Can we reduce the space complexity?

The factorial function

Time and space complexity of computation

The computation of `(fact` n`)` has

time complexity $2 \cdot (n+1) = O(n)$

space complexity $O(n)$: the maximum number of frames added to E is $n+1$

Can we reduce the space complexity?

Main idea: Add an extra argument to accumulate and propagate the result computed so far.
This extra argument is called **accumulator**.

The computation of (`fact` n) has

time complexity $2 \cdot (n+1) = O(n)$

space complexity $O(n)$: the maximum number of frames added to E is $n+1$

Can we reduce the space complexity?

Main idea: Add an extra argument to accumulate and propagate the result computed so far.
This extra argument is called **accumulator**.

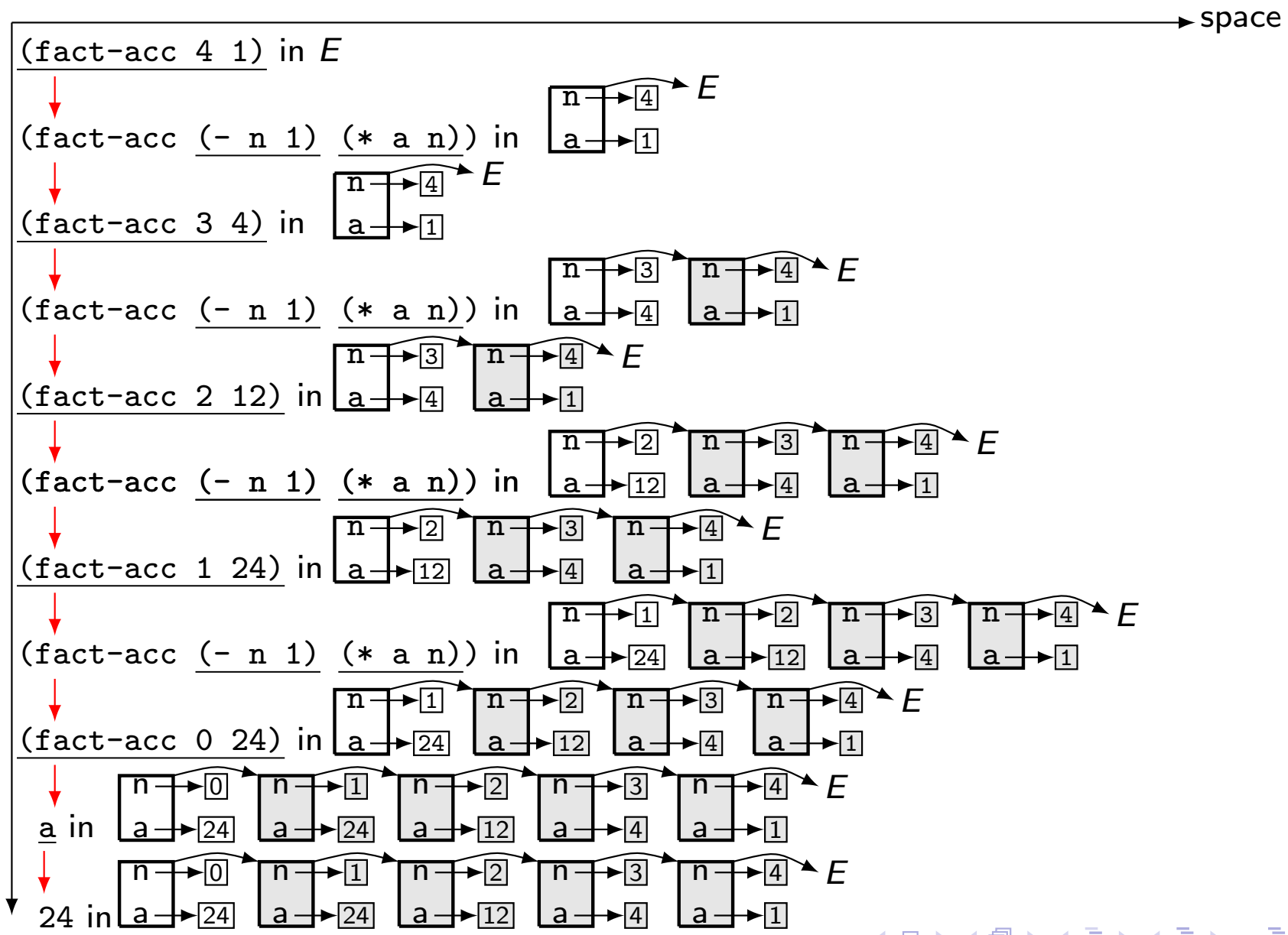
Implementation:

```
(define (fact n) (fact-acc n 1))
(define (fact-acc n a)
   (if (= n 0) a (fact-acc (- n 1) (* a n))))
```

- (`fact-acc` n a) computes $n! \cdot a$, therefore (`fact-acc` n `1`) computes $n!$

The factorial function

Towards a space-efficient implementation



A space-efficient implementation
Tail call optimization

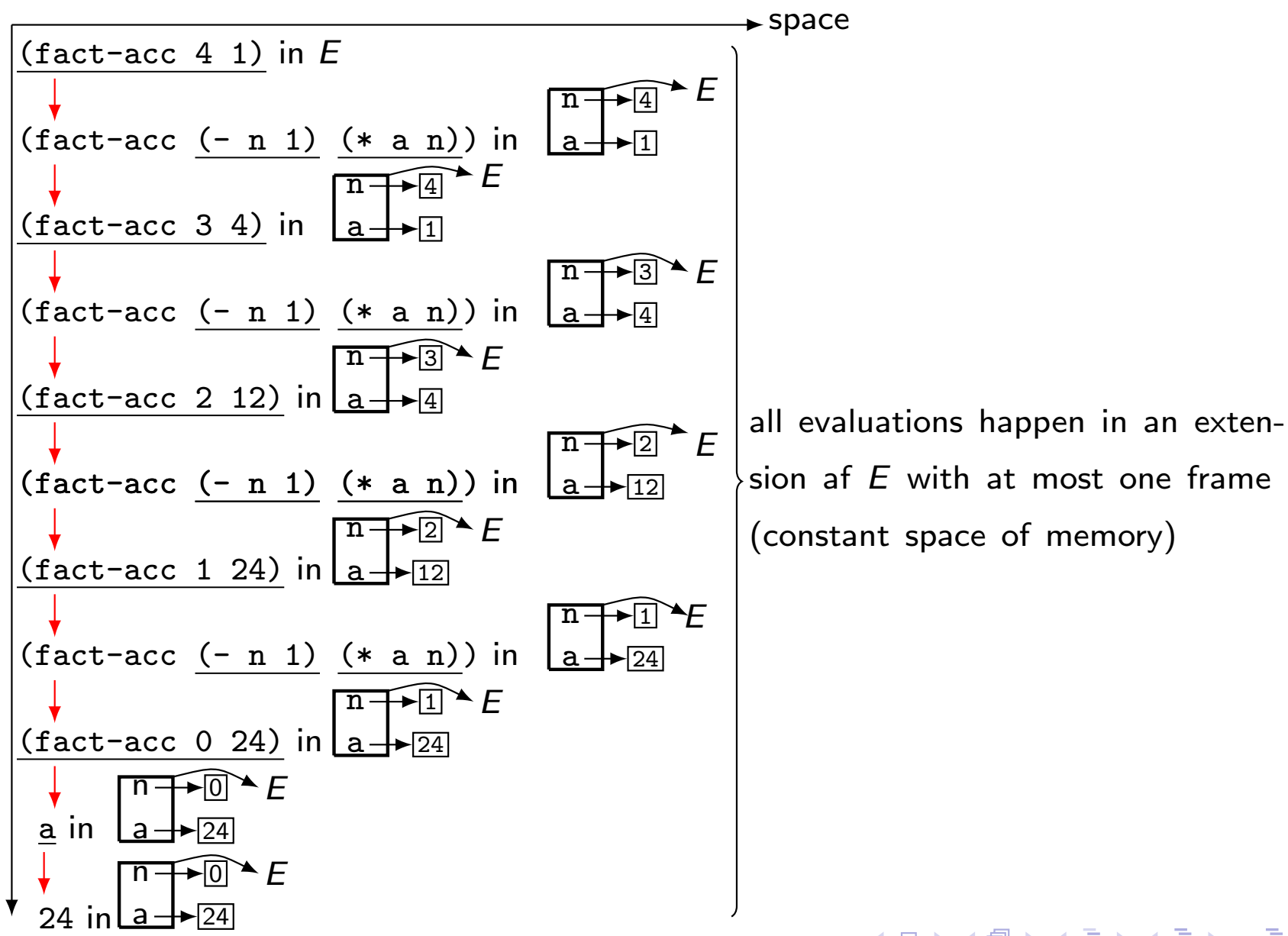
```
(define (fact n) (fact-acc n 1))
(define (fact-acc n a)
  (if (= n 0) 1 (fact-acc (- n 1) (* a n))))
```

Clever compilers and interpreters recognize the fact that the gray-colored frames are useless:

- The gray frames can be discarded by a garbage-collector
  $\Rightarrow$ the space complexity of computing (`fact-acc` n 1) becomes constant, $O(1)$ (see next slide).
- This technique of saving memory is called tail call optimization
  - Tail call optimization can be applied whenever the recursive call is **the last action** in the body of a recursive function.
  - Functions written in this way (including `fact-acc`) are called tail recursive.
- Most languages, including Racket, Java, C++ implement tail call optimization.

Tail call optimization

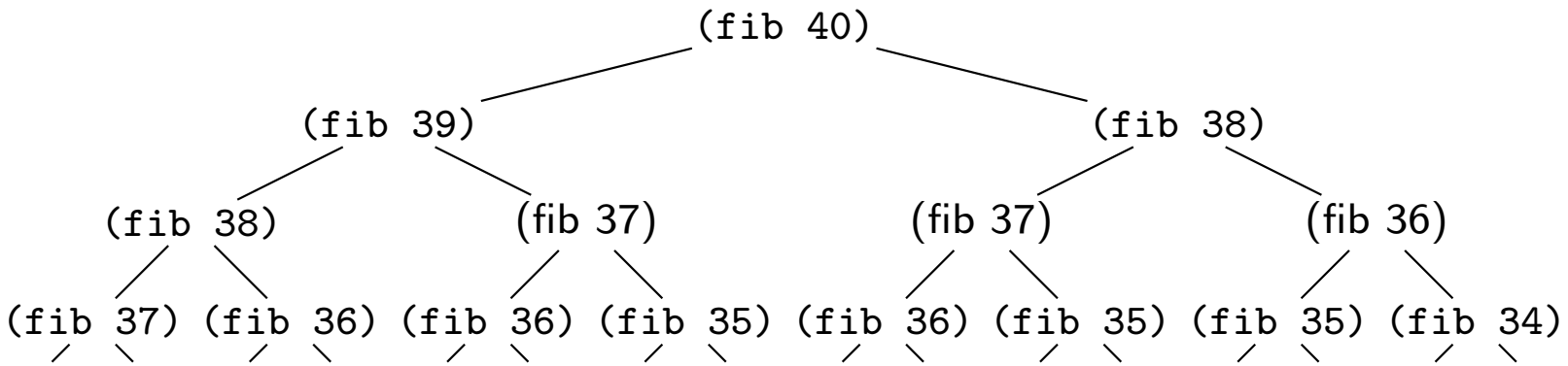
Example: computation of `(fact-acc 4 1)` with tail call optimization



More examples
The Fibonacci function

```
(define (fib n)
  (if (or (= n 0) (= n 1))
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

The computation of `(fib n)` for $n > 0$ has a tree-like structure.



- `(fib 40)` is computed once
- `(fib 38)` is computed 2 times

  ...
- `(fib 0)` is computed 165,580,141 times.

⇒ `(fib 40)` performs 331,160,281 function calls!

The Fibonacci function

A tail recursive definition

Add 2 extra arguments to accumulate and propagate the values of two successive Fibonacci numbers:

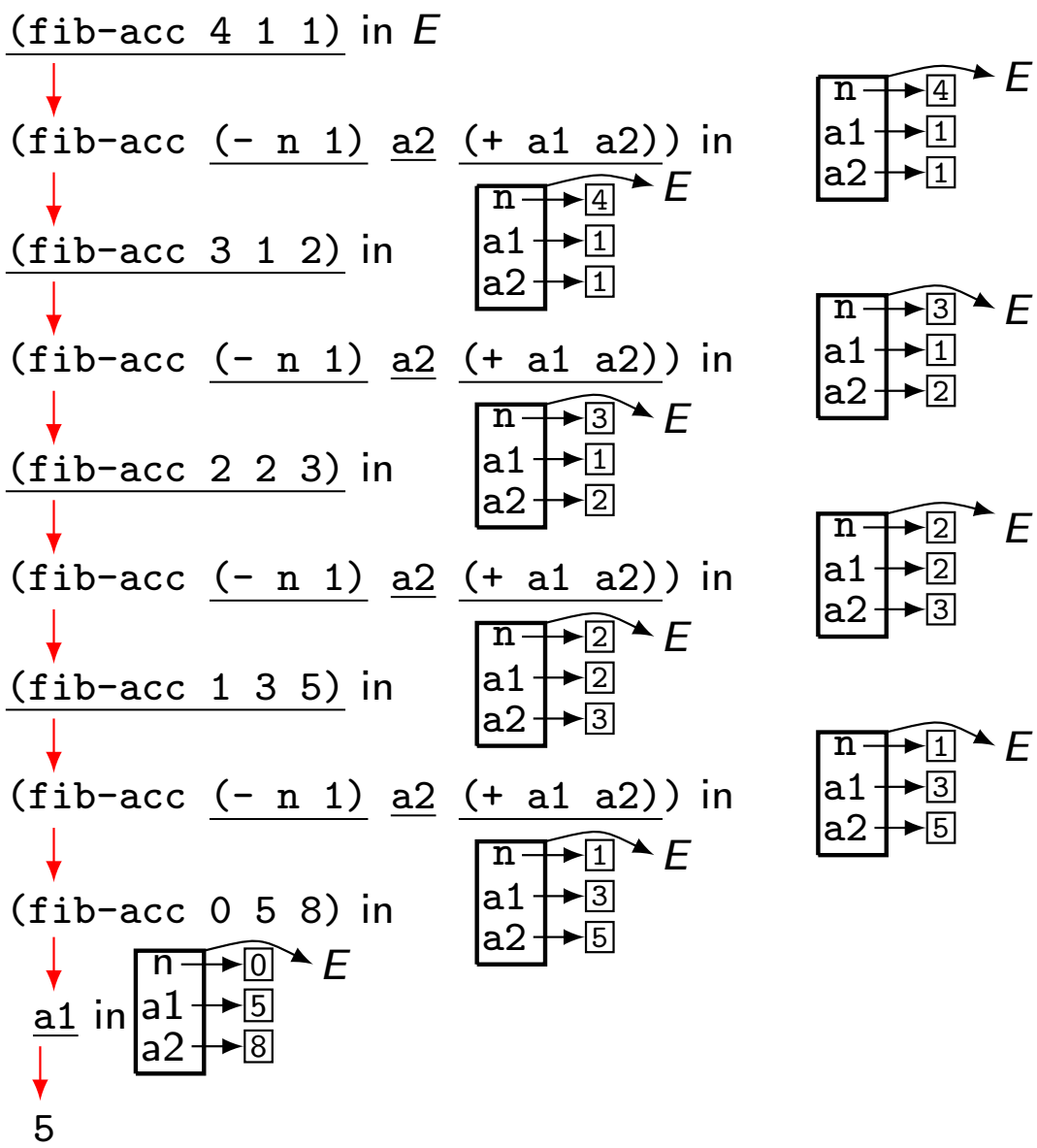
- Suppose f_n is the value of `(fib` n`)` for $n \geq 0$.
- To compute f_n, we call `(fib-acc` n f_0 f_1`)` whose computation evolves as follows:

$$\begin{aligned}
\texttt{(fib-acc}\ n\ f_0\ f_1\texttt{)} &\rightarrow \texttt{(fib-acc}\ n-1\ f_1\ f_2\texttt{)} \\
&\rightarrow \texttt{(fib-acc}\ n-2\ f_2\ f_3\texttt{)} \\
&\rightarrow \ldots \\
&\rightarrow \texttt{(fib-acc}\ k\ f_{n-k}\ f_{n-k+1}\texttt{)} \\
&\rightarrow \ldots \\
&\rightarrow \texttt{(fib-acc 0}\ f_n\ f_{n+1}\texttt{)} \\
&\rightarrow f_n
\end{aligned}$$

```
(define (fib-acc n a1 a2)
  (if (= n 0)
      a1
      (fib-acc (- n 1) a2 (+ a1 a2))))
```

Another example of tail call optimized computation

Computation of f_4 with `(fib-acc 4 1 1)`



Computation of Fibonacci numbers

Remarks

- `(fib n)` has time complexity $O(2^n)$ and space complexity $O(n)$
- `(fib-acc n 1 1)` has time complexity $O(n)$ and space complexity $O(1)$:
  - The cail call optimized computation of the Fibonacci number f_n with `(fib-acc n 1 1)` is similar to the computation of f_n with the imperative program:

```
a1=1; a2=1;
for (i = n;i>0;i--) { tmp=a1;
                      a1=a2;
                      a2=tmp+a2;
}
return a1;
```

SUGGESTION: Use RACKET to implement `fib` and `fib-acc`, and compare the runtimes of computing f_{39} with `(fib 39)` and `(fib-acc 39 1 1)`.

Possible pitfalls with recursion

Is recursive computation fast?

- **Yes**: some tail-recursive functions are remarkably efficient
- **No**: We can easily write elegant, but spectacularly inefficient recursive programs, e.g.

```
(define (fib n)
   (if (or (= n 0) (= n 1))
       1
       (+ (fib (- n 1)) (fib (-n 2)))))
```

Recursion can take a long time if it needs to repeatedly recompute intermediate results

General principle: Use tail recursion to make your functions efficient.

References

[Hughes:1989] John Hughes. *Why functional programming matters*. Computer Journal, 32(2), 1989.

[Thompson:1999] Simon Thompson. *The Craft of Functional Programming*, Second Edition. International Computer Science Series. Pearson Addison Wesley, 1999.