Lecture 3: Environment-based computations

Functions as values. Tail recursion

Mircea Marin
West University of Timișoara
mircea.marin@e-uvt.ro

The smallest language for FP. It consists of

1. A **language** to write expressions, also known as terms.

$$t ::= x \mid \lambda x.t_1 \mid t_1\ t_2$$

   where x is a variable and
   - $\lambda x.t$ is an abstraction with intended reading "the function which, for input x computes the value of t."
     - λx is the binder of the abstraction
     - t is the body (or scope) of the abstraction
   - $t_1\ t_2$ is an application: t_1 is applied to argument t_2.

2. Transformation rules

   α-conversion: $\lambda x.t \rightarrow_\alpha \lambda y.[y/x]t$
   if $[y/x]t$ is a capture-free substitution.

   β-reduction: $(\lambda x.t_1)\ t_2 \rightarrow_\beta [t_2/x]t_1$
   if $[t_2/x]t_1$ is a capture-free substitution.

Racket and the λ-calculus

The λ-calculus is the core language of Racket $\Rightarrow$ Racket recognizes the expressions of the λ-calculus, but we should write them in a slightly different way:

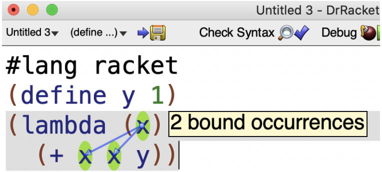
`(lambda (`x`) `t`)` instead of $\lambda x.t$
`(`t_1 t_2`)` instead of $t_1\ t_2$

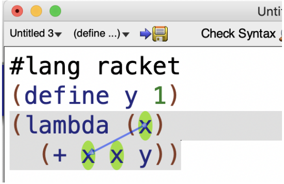
Remarks

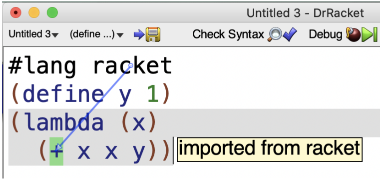
1. For efficiency reasons, Racket has built-in values for many useful datatypes including many predefined functions.
2. The editor of Racket allows us to view the referenced-based representation of λ-terms
   - If we hover the mouse over a binder, the editor highlights the occurrences bound to it.
   - If we hover the mouse over a variable occurrence, we see a reference to its corresponding binder.

Racket and the λ-calculus

Referenced-based representations of expressions (snapshot)







α-conversion allows us to do harmless renamings of parameters of functions.

Example

Suppose y is a global variable with a given value.

- $\lambda x.y$ is the function which, for every input x, returns the value of y.

  $$\lambda x.y \rightarrow_\alpha \lambda z.[z/x]y = \lambda z.y$$

  is harmless because $\lambda x.y$ and $\lambda z.y$ are describe the same function. But we are not allowed to perform the variable-capture substitution

  $$\lambda x.y \rightarrow \lambda y.[y/x]y = \lambda y.y$$

  because $\lambda x.y$ and $\lambda y.y$ describe different functions.

β-reduction simulates the first-step of evaluating a function call: We replace in the body of the function the formal parameters with the input arguments.

Example (Evaluation in Racket)

```
(define y 7)
> ((lambda (x) (+ x y)) 5)
  →β [7/y][5/x](+ x y)
  = (+ 5 7)
  → 12
> ((lambda (x) (lambda (y) (+ x y))) 6)
  →β [6/x](lambda (y) (+ x y))
  = (lambda (y) (+ 6 y))
```

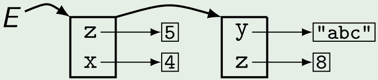
Remark: + and y have free occurrences ⇒ to use them, we need to know where to find their values.

Environment-based computations

Environment = data structure which stores the values of variables with free occurrences.

- Environment = a list of frames.
- Every frame is a table of values for some 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.

All evaluations are performed w.r.t. a **global environment** which stores the values of variables with free occurrences in expressions.

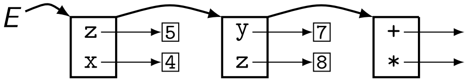
The global environment is initialized with bindings for predefined variables when we start the system

- Built-in functions names are predefined variables with functions as values

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

1. All variables x in *expr* are replaced with $E(x)$
2. The new expression is evaluated using the rules of evaluation.

Evaluation of expressions
Example



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

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

Remark

From now on we will always assume implicitly that the environment has a frame with bindings for all built-in operations and constants.

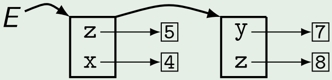
When the interpreter reads a definition

(`define` *var* *expr*)

in an environment *E*, it does the following:

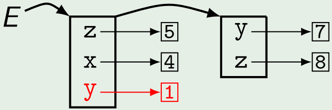
1. It computes the value *v* of *expr* in *E*
2. It adds the binding *var* $\mapsto$ *v* to the top frame of *E*.

Example




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

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




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

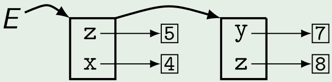
The new binding $y \mapsto 1$ shadows the binding $y \mapsto 8$.

Bindings can shadow each other, but they can not be overwritten

⇒ (`define` *var* *expr*)
is prohibited in an environment *E* which has a binding of *var* in the first frame.

Example

We can not redefine `x` and `z` in environment



but we can define `y`.

Blocks and their evaluation

Block = sequence of definitions and expressions, which ends with an expression.

- `(local [ ]` *comp*$_1$ `...` *comp*$_n$ *expr*`)`

  is a special form for the block made of the sequence of components *comp*$_1$,..., *comp*$_n$ followed by *expr*.

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

1. E is extended with a temporary top frame, initially empty.
2. The all components of the block are interpreted one by one:
   - the block definitions add bindings to the (initially empty) top frame
   - *expr* is evaluated and its value is returned as value of the block
3. E is restored by discarding its temporary top frame.

The evaluation of blocks
Example

Remark

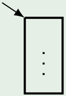
(**println** *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
```



Remark

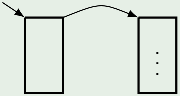
(`println` *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
```



The evaluation of blocks

Example

Remark

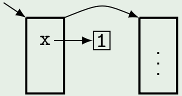
(println *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
```



Remark

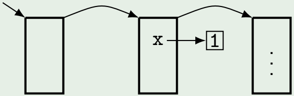
(`println` *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
```



Remark

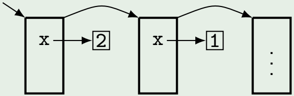
(`println` *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
```



Remark

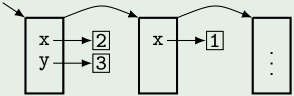
(println *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
```



The evaluation of blocks

Example

Remark

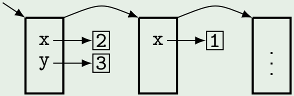
(println *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
5
```



Remark

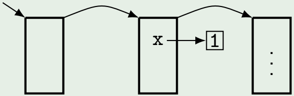
(`println` *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
5
```



Remark

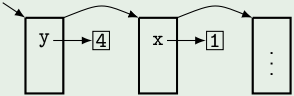
(`println` *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
5
```



Remark

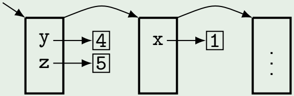
(`println` *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
5
```



Remark

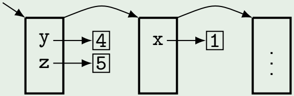
(`println` *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
5
10
```



Remark

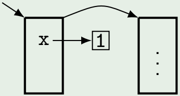
(println *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
5
10
3
```



The evaluation of blocks

Example

Remark

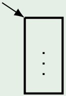
(`println` *expr*)

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

Example

```
> (local [ ]
    (define x 1)
  (local [ ]
    (define x 2)
    (define y 3)
    (println (+ x y)))
  (local [ ]
    (define y 4)
    (define z 5)
    (println (+ x y z)))
  (+ x 2))
5
10
3
```



Other special forms with blocks

1. The conditional form

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

where $test_1, \ldots, test_n$ are boolean expressions. The evaluation returns the value of the first block $block_i$ for which $test_i$ is true. If all tests are false, the evaluation returns value `#<void>`

2. Abstractions, which are used to define functions

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

3. `let` and `let*`:

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

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

The boolean operators and and or

`and` and `or` are special forms: **they are not functions!**

1. `(and` t_1 `...` t_n`)`
   evaluates expressions $t_1, \ldots, t_n$ from left to right.
   - if it finds t_i with value `#f`, it returns `#f`
   - otherwise, it returns the value of t_n.
2. `(or` t_1 `...` t_n`)`
   evaluates expressions $t_1, \ldots, t_n$ from left to right.
   - if it finds t_i whose value is not `#f`, it returns the value of t_i.
   - otherwise, it returns `#f`.

REMARK: In Racket, all non-`#f` values are true. This is similar to language C, where anything non-zero is interpreted as true.

```
> (and 1 (lambda (x) x) #f)
#f
> (and)
#t
> (and 1 "abc" 'abc)
'abc
```

```
> (or #f 'abc "abc")
'abc
> (or)
#f
```

The special forms `if` and `cond`

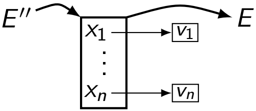
(`if` *test* $expr_1$ $expr_2$)

is equivalent with

```
(cond [test expr1]
      [#t expr2])
```

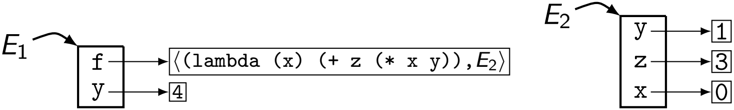
- `cond` is more general than `if`, also because its branches can be blocks.
- The branches of `if` must be expressions.

- The value of `(lambda (`x_1 ... x_n`)` *block*`)` in an environment E is the pair $\langle$`(lambda (`x_1 ... x_n`)` *block*`)`,$E\rangle$
  - Such a value is called lexical closure or function closure or closure: it is a pair made of (1) the textual definition of the function and (2) the environment where f was created.
- If f has value $\langle$`(lambda (`x_1 ... x_n`)` *block*`)`,$E\rangle$ then the value of `(`f t_1 ... t_n`)` in E' is computed as follows:
  - compute the values $v_1, \ldots, v_n$ of $t_1, \ldots, t_n$ in E'
  - create the temporary environment



    and compute v =the value of *block* in E''
  - return v as the value of `(f` t_1 ... t_n`)` in E'.

Consider the environments E_1 and E_2 where

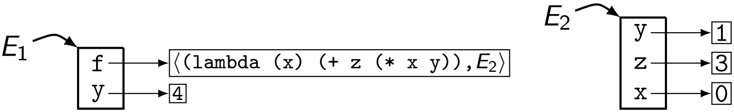


What is the value of `(f y)` in E_1?

The evaluation of function calls

Illustrated example

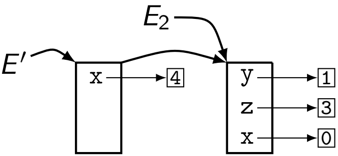
Consider the environments E_1 and E_2 where



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

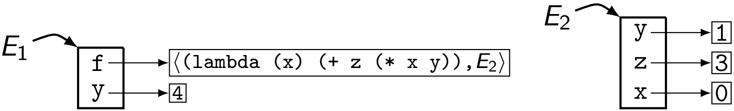



```
> (+ z (* x y)) in E'
12
```

The evaluation of function calls

Illustrated example

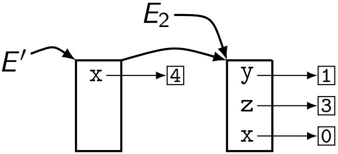
Consider the environments E_1 and E_2 where



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.

Tail recursion

Tail recursion = technique to implement efficient recursive computation.
Remember that:

Tail recursion

Tail recursion = technique to implement efficient recursive computation.

Remember that:

- Recursion = technique that allows us to break a problem into one or more subproblems similar to the initial problem.

Tail recursion

Tail recursion = technique to implement efficient recursive computation.

Remember that:

- Recursion = technique that allows us to break a problem into one or more subproblems 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.
  - All repetitive computations can be performed only by recursion.

Tail recursion

Tail recursion = technique to implement efficient recursive computation.
Remember that:

- Recursion = technique that allows us to break a problem into one or more subproblems 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.
  - All repetitive computations can be performed only by recursion.

Why learn recursion?

Tail recursion

Tail recursion = technique to implement efficient recursive computation.
Remember that:

- Recursion = technique that allows us to break a problem into one or more subproblems 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.
  - All repetitive computations can be performed only by recursion.

Why learn recursion?

- New way of thinking

Tail recursion

Tail recursion = technique to implement efficient recursive computation.

Remember that:

- Recursion = technique that allows us to break a problem into one or more subproblems 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.
  - All repetitive computations can be performed only by recursion.

Why learn recursion?

- New way of thinking
- Powerful programming tool

Tail recursion

Tail recursion = technique to implement efficient recursive computation.
Remember that:

- Recursion = technique that allows us to break a problem into one or more subproblems 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.
    - All repetitive computations can be performed only by recursion.

Why learn recursion?

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

Tail recursion

Tail recursion = technique to implement efficient recursive computation.

Remember that:

- Recursion = technique that allows us to break a problem into one or more subproblems 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.
  - All repetitive computations can be performed only by recursion.

Why learn recursion?

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

Many computations and data structures are naturally recursive

- 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

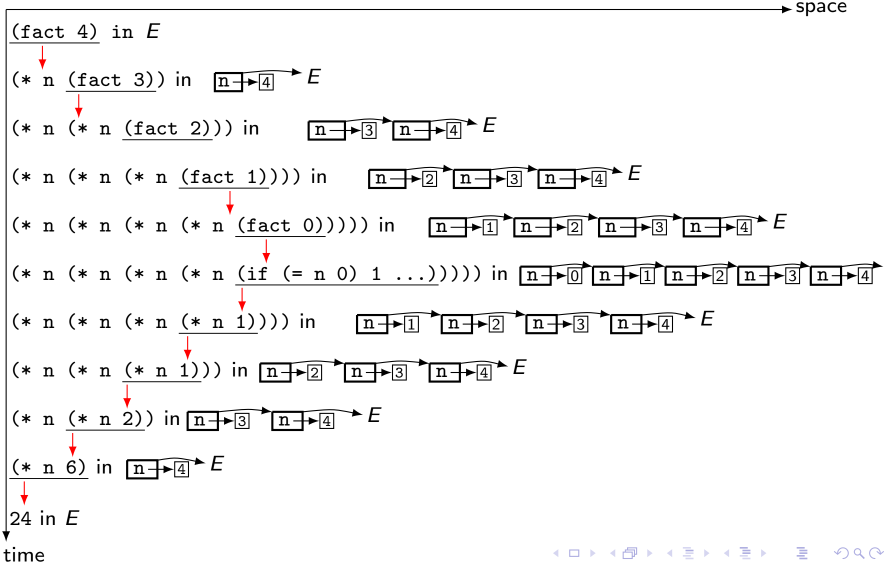
How to write a recursive definition?

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.

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

Q1: What is the space and time complexity of computing (fact n) when $n \in \mathbb{N}$?

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



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

Q1: What is the space and time complexity of computing `(fact n)` when $n \in \mathbb{N}$?

A1: 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 $n+1$

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

Q1: What is the space and time complexity of computing (fact n) when $n \in \mathbb{N}$?

A1: 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 $n+1$

Q2: Can we reduce the space complexity?

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

Q1: What is the space and time complexity of computing `(fact n)` when $n \in \mathbb{N}$?

A1: 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 $n+1$

Q2: Can we reduce the space complexity?

A2: Main idea: Add an extra argument to accumulate and propagate the result computed so far.

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

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

Q1: What is the space and time complexity of computing `(fact n)` when $n \in \mathbb{N}$?

A1: 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 $n+1$

Q2: Can we reduce the space complexity?

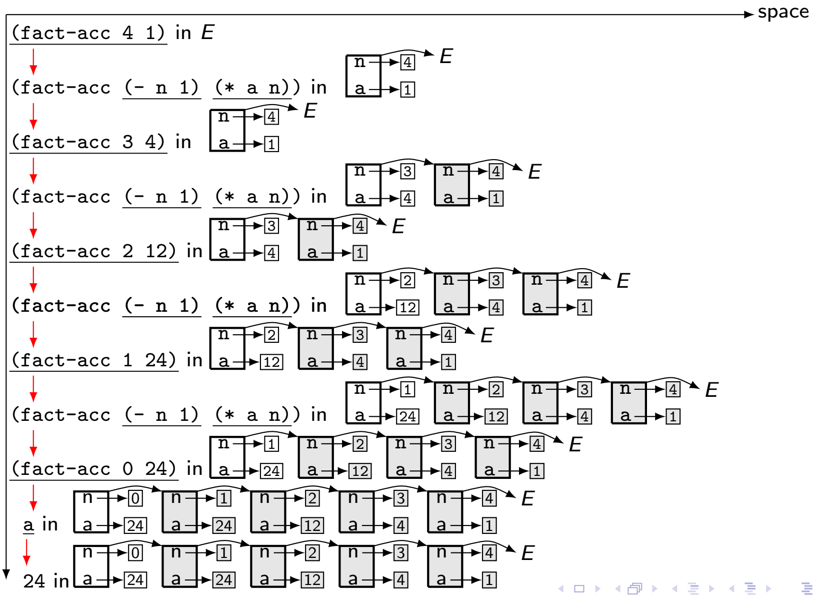
A2: Main idea: Add an extra argument to accumulate and propagate the result computed so far.

```
(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

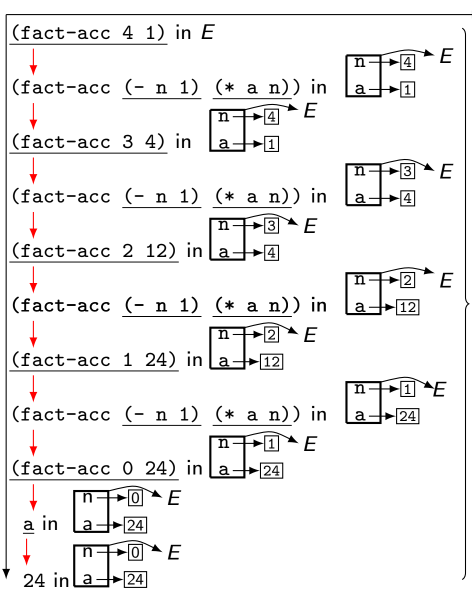
```
(define (fact n) (fact-acc n 1))
(define (fact-acc n a)
  (if (= n 0) 1 (fact-acc (- n 1) (* a n))))
```

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

- The gray frames can be discarded by a garbage-collector
  ⇒ 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



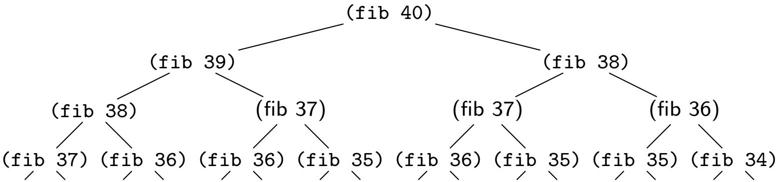
all evaluations happen in an extension af *E* with at most one frame (constant space of memory)

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.

$\Rightarrow$ (fib 40) performs 331,160,281 function calls!

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

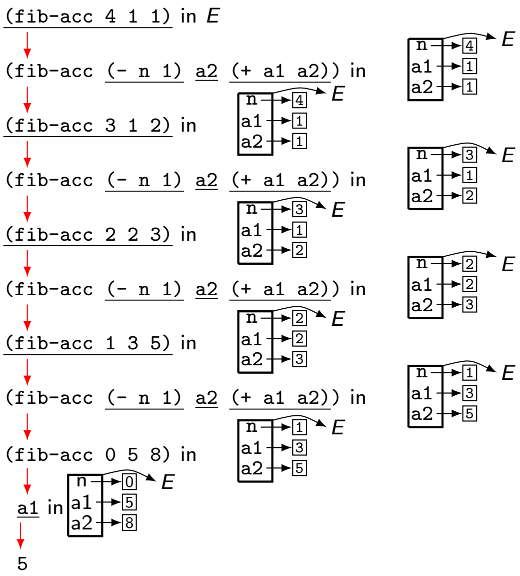
- Suppose f_n is the value of `(fib n)` for $n \geq 0$.
- To compute f_n, we call `(fib-acc n f0 f1)` whose computation evolves as follows:

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

```
(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 tail 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;
```

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: Whenever possible, use tail recursion to make your functions efficient.

Conclusion

Environment-based computation is a standard technique to keep track of the meaning of names in a program.

- Environment = list of frames; every frame is a table that maps distinct names to values.
- Definitions add bindings to the top (=first) frame of the environment
- Evaluation of blocks extends the environment with a temporary top frame, to store the bindings of local definitions. The top frame and its bindings are garbage collected when block evaluation ends.
- In FP, all recursive computations are performed by recursion.
  - Every recursive step extends environment with a new frame ⇒ deep recursive calls produce stack overflow
  - Tail recursion = compiler optimization technique which garbage collects frames and bindings that become inaccessible