

# **Рекурсивни и итеративни процеси**

15 октомври, 2019

# **Линейни рекурсивни процеси**

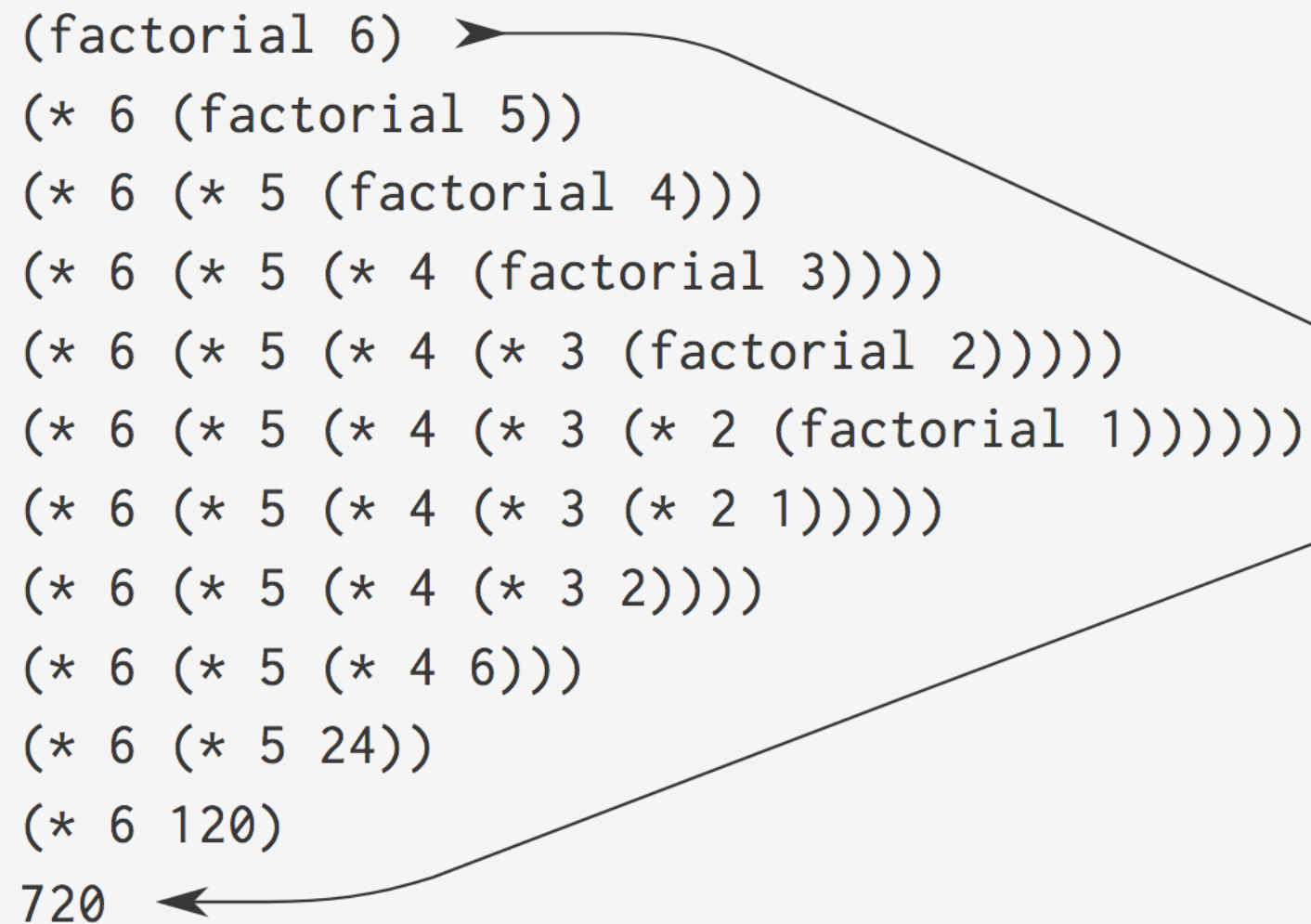
# Да разгледаме процедурата **factorial**

*; Linear recursive process*

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

- **factorial** поражда линеен рекурсивен процес
- Използваме индуктивната дефиниция на **n!**:
  - $0! = 1$
  - $n! = n * (n - 1)!$

```
(factorial 6) ➤  
(* 6 (factorial 5))  
(* 6 (* 5 (factorial 4)))  
(* 6 (* 5 (* 4 (factorial 3))))  
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))  
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))  
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))  
(* 6 (* 5 (* 4 (* 3 2))))  
(* 6 (* 5 (* 4 6)))  
(* 6 (* 5 24))  
(* 6 120)  
720 ←
```



# Два основни етапа

- разгръщане:
  - извършва се поредица от субституции докато не се стигне до базата
  - получава се верига от отложени операции
- свиване — извършване на операциите

# Сложность

- время —  $O(n)$
- память —  $O(n)$

# **Линейни итеративни процеси**



# Да разгледаме процедурите **factorial** и **fact-iter**

*; Linear iterative process*

```
(define (factorial n)  
  (fact-iter 1 1 n) )
```

```
(define (fact-iter product counter max-count)  
  (if (> counter max-count)  
    product  
    (fact-iter (* counter product)  
                (+ counter 1)  
                max-count) ) )
```

- **factorial** поражда линеен итеративен процес
- Дефинираме състояние, което се изменя на всяка стъпка (итерация):
  - начално състояние:
    - `counter <- 1`
    - `product <- 1`
    - `max-count <- n`
  - правила за преминаване към следващо състояние:
    - `product <- counter * product`
    - `counter <- counter + 1`
    - `max-count <- max-count`
  - условие за завършване:
    - `counter > max-count`
    - тогава резултатът е `product` от финалното състояние

```
(factorial 6) ➤  
(fact-iter 1 1 6)  
(fact-iter 1 2 6)  
(fact-iter 2 3 6)  
(fact-iter 6 4 6)  
(fact-iter 24 5 6)  
(fact-iter 120 6 6)  
(fact-iter 720 7 6)  
720 ←
```

The diagram illustrates the execution of a factorial function using an iterative helper function. The sequence of calls to `fact-iter` shows the accumulation of the factorial result. A curved arrow connects the initial call to `(factorial 6)` to the final result `720`.

# Сложность

- время —  $O(n)$
- память —  $O(1)$

# Вложени дефиниции на процедури

*; Nested definitions*

```
(define (factorial n)
  (define (fact-iter product counter max-count)
    (if (> counter max-count)
      product
      (fact-iter (* counter product)
                    (+ counter 1)
                    max-count)))

(fact-iter 1 1 n) )
```

- скриване на имплементационните детайли
- **fact-iter** е помощна процедура за реализацията на **factorial**
- **fact-iter** не е предвидена да бъде използвана отделно от **factorial**
- затова можем да я скрием в блока на процедурата **factorial**
- влагането на дефиниции на процедури ни помага да организираме програмите, които пишем

# fact-iter има достъп до аргументите на factorial

```
; iter has access to all arguments of factorial  
(define (factorial n)  
  (define (iter product counter)  
    (if (> counter n)  
      product  
      (iter (* counter product)  
              (+ counter 1)))))  
  
(iter 1 1)
```