Computer Programs Et09-12

dr. H. Corporaal
dr.ir. A.J.C. van Gemund
Computer Systems Group

http://cs.et.tudelft.nl/~heco
email: heco@cs.et.tudelft.nl

Session 2: Procedures and Processes

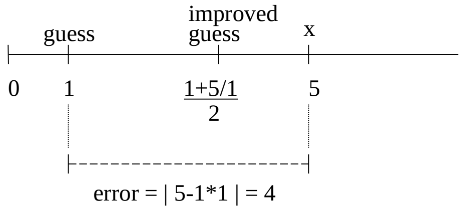
Slide 1.2. 1

Newton’s square root method

- Purpose: compute $y = \sqrt{x}$
- Start with guessing *y*
- If *y* correct then *x/y = y*
- Else try *(x/y + y) /2*
  - note if *y* too small then *x/y* too large

Slide 1.2. 2

(SQRT 5), Newton’s square root

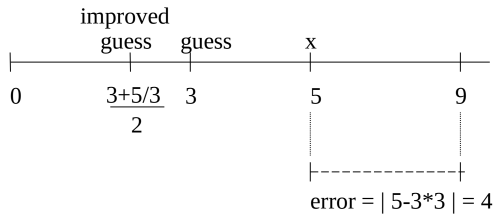


error = | 5-1*1 | = 4

current guess: 1
error: 4
improved guess: 3

Slide 1.2. 3

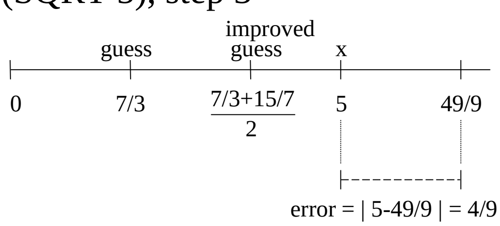
(SQRT 5), step 2



current guess: 3
error: 4
improved guess: 7/3

Slide 1.2. 4

(SQRT 5), step 3



current guess: 7/3
error: 4/9
improved guess: 47/21= 2.24

Slide 1.2. 5

A procedure for Newton's process

```
( define    ( sqrt-iter guess x )
   ( if     ( good-enough? guess x )
            guess
            ( sqrt-iter ( improve guess x )
                        x )))
```

- <good-enough?> and <improve> are black-box procedures
- <sqrt-iter> is a recursive procedure

Slide 1.2. 6

Black box 1: good-enough?

```
( define    ( good-enough? guess x )
    ( <
          ( abs    ( -   x    ( square guess )))
          0.001 ))
```

- The value of good-enough? is always TRUE or FALSE
- <abs> and <square> are primitive procedures

Slide 1.2. 7

Black box 2: improve

```
( define     ( improve     guess x )
     ( average    guess    ( /    x guess )))

( define     ( average     x y )
     ( /       (+    x y )     2 ))
```

- The value of improve is a number
- <average> is a black box within a black box

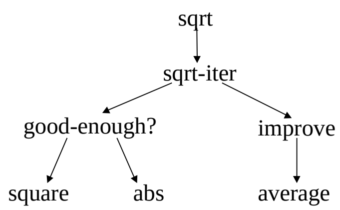
Slide 1.2. 8

The complete Newton’s procedure

```
( define     ( sqrt  x )
    ( sqrt-iter  1  x ))
( define     ( sqrt-iter  guess  x )
    ( if       ( good-enough?  guess  x )
               guess
               ( sqrt-iter  ( improve  guess  x ) x )))
( define     ( good-enough?  guess  x )
    ( <        (abs   ( -    ( square guess ) x )) 0.001 ))
( define     ( improve  guess  x )
    ( average  guess    ( /    x  guess )))
```

Slide 1.2. 9

Call Graph



Slide 1.2. 10

Mr. Skiem, session 1.2.a

demo of Newton's method

Slide 1.2. 11

Newton's procedure using block structures

```
( define   ( sqrt  x )
    ( define( good-enough?  guess  x )
            ( <     ( abs   ( -    ( square guess ) x )) 0.001 ))
    ( define( improve  guess  x )
            ( average  guess   ( -    x  guess )))
    ( define( sqrt-iter  guess  x )
            (if      ( good-enough?  guess  x )
                     guess
                     ( sqrt-iter   ( improve  guess  x ) x )))
    ( sqrt-iter  1  x ))
```

Slide 1.2. 12

Block structure

- Purpose
  - hide implementation details
  - hide irrelevant, local, names
- Use
  - defines should always come first
  - value of the last combination will be returned
  - block structure within block structure is possible

Slide 1.2. 13

Newton's procedure using lexical scoping

```
( define    ( sqrt  x )
    ( define    ( sqrt-iter  guess )
         ( define    ( good-enough? )
              ( <      ( abs    ( -   ( square  guess ) x )) .001 ))
         ( define    ( improve )
              ( average  guess     (/   x  guess )))
         ( if     ( good-enough?  guess )
                  guess
                  ( sqrt-iter   ( improve  guess ))))
    ( sqrt-iter  1))
```

Slide 1.2. 14

Linear recursive process

```
5!   =   5*4*3*2*1
     =   5* ( 4*3*2*1 )
     =   5*4!
1!   =   1

n!   =   n* ( n-1 )!

( define     ( fac  n )
     ( if     ( =   n  1 )
              1
              ( *   n   ( fac   (-  n  1 )))))
```

Slide 1.2. 15

Factorial

```
(fac 5)
(* 5 (fac 4))
(* 5 (* 4 (fac 3)))
(* 5 (* 4 (* 3 (fac 2))))
(* 5 (* 4 (* 3 (* 2 (fac 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
120
```

Slide 1.2. 16

Factorial

- Time complexity = $\Theta(n)$
- Space complexity = $\Theta(n)$

- Definition:

  complexity of $f(n) = \Theta(n) \Leftrightarrow$

  $\exists\, k,l,m \quad \forall n>m \quad k*n < f(n) < l*n$

Slide 1.2. 17

Iterative process

5! = 5*4*3*2*1

(((((1) *2) *3) *4) *5)

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

Slide 1.2. 18

Iterative Factorial

```
(fac 5)
(fac-iter   1 1 5)
(fac-iter   1 2 5)
(fac-iter   2 3 5)
(fac-iter   6 4 5)
(fac-iter  24 5 5)
(fac-iter 120 6 5)
120
```

- Time complexity = $\Theta(n)$
- Space complexity = $\Theta(1)$ i.e. constant

Slide 1.2. 19

Mr. Skiem, session 1.2.b

Iterative versus Recursive Processes

Slide 1.2. 20

Iterative versus recursive

Iterative:
- only remember current values (state variables)

Recursive:
- deferred operations
- remember status of all (deferred) operations

- Linear recursive means, the number of defers is of the same order as an input variable
- A recursive procedure does not mean a recursive process

Slide 1.2. 21

Tree recursion: Fib

Sequence of Fibonacci numbers:

- 0 1 1 2 3 5 8 13 21 ....

Rule:

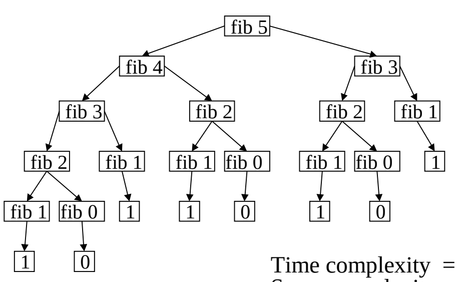
- Next one is sum of previous two numbers

Tree recursion: Fib

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{else} \end{cases}$$

```
( define    ( fib  n )
    ( cond  (( =    n  0 ) 0 )
            (( =    n  1 ) 1 )
            (else  ( +  ( fib   (-  n  1 ))
                         ( fib   ( - n  2 ))))))
```


Fibonacci: Call graph



Time complexity $= \Theta(2^n)$
Space complexity $= \Theta(n)$

Order of growth

- Growth in space
  - (state) variables
- Growth in execution time
  - number of primitive operations

	Θ time (n)	Θ space (n)
fac, linear recursive	n	n
fac, iterative	n	1
fib, tree recursive	c^n	n

Slide 1.2. 25

Fibonacci: Iterative version

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))

Try example: (fib 5) -> 5
```

What is the space and time complexity?

Slide 1.2. 26

Fibonacci

- Exercise 1.13 shows that

$$\boxed{\mathrm{Fib}(n) = \frac{(\phi^n - \psi^n)}{\sqrt{5}}}$$

where: $\phi^n = (1+\sqrt{5})/2$

$\psi^n = (1-\sqrt{5})/2$

Slide 1.2. 27

Mr. Skiem, session 1.2.c

Tree recursion:
- Fibonacci
- Towers of Hanoi

Slide 1.2. 28

Extra material

- Exponentiation
  - 3 methods
- Greatest Common Divisor GCD
- Prime number test
  - straightforward
  - probabilistic method

Slide 1.2. 29

Exponentiation

- linear recursive with $\Theta(n)$ space and time complexity

```
(define (expt b n)
  (if (= n 0)
    1
    (* b (expt b (- n 1))) ))
```

Slide 1.2. 30

Exponentiation

- linear recursive process with Θ(1) space complexity

```
(define (expt b n)
  (expt-iter b n 1))

(define (expt-iter b cnt prod)
  (if (- cnt 0)
      prod
      (expt-iter b
                 (- cnt 1)
                 (* b prod) )))
```

Slide 1.2. 31

Exponentiation

- logaritmic (recursive) process with Θ(log n) space and time complexity

```
(define (expt b n)
  (cond (= n 0) 1
        ((even? n) (square (expt b (/ n 2))))
        (else (* b (expt b (- n 1))) ))

(define (even? n)
  (= (remainder n 2) 0))
```

Slide 1.2. 32

Fast exponentiation example

```
(expt 7 25)
(* 7 (expt 7 24))
(* 7 (sq (expt 7 12)))
(* 7 (sq (sq (expt 7 6))))
(* 7 (sq (sq (sq (expt 7 3)))))
(* 7 (sq (sq (sq (* 7 (expt 7 2))))))
(* 7 (sq (sq (sq (* 7 (sq (expt 7 1)))))))
(* 7 (sq (sq (sq (* 7 (sq (* 7 (expt 7 0))))))))
(* 7 (sq (sq (sq (* 7 (sq (* 7 1)))))))
(* 7 (sq (sq (sq (* 7 (sq 7))))))
(* 7 (sq (sq (sq (* 7 49)))))
.....   etc.
```

Slide 1.2. 33

Greatest Common Divisor GCD

(GCD) algorithm using Euclid's method

- use: GCD(a,b) = GCD(b, a mod b)

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))

(gcd 42 24) =>
(gcd 24 18) =>
(gcd 18 6)  =>
(gcd 6 0)   =>
6
```

Slide 1.2. 34

Testing for primality

- Straightforward method with $\Theta(\sqrt{n})$ time complexity
- Probabilistic method using Fermat's little theorem

Slide 1.2. 35