VL Nichtprozedurale Programmierung

Funktionale Programmierung

Andreas Naderlinger
Fachbereich Computerwissenschaften
Software Systems Center
Universität Salzburg

Recursion

- What is recursion?
- Two cases (at least)
  - Base case first
    - no recursive call; ends ‘looping’
    - often more complicated than the recursive case
  - Recursive case
    - calls itself with a ‘reduced’ problem

Example (f)

Demo (count)

Recursion

- very elegant (e.g., quicksort, towersOfHanoi)
- Sometimes not efficient (e.g. fibonacci)
  – stack growth
  – multiple invocation (tree recursion)

Linear Recursion / Iteration (1) **[SICP 1.2]**

- n! = n * (n-1) * (n-2) * … * 3 * 2 * 1
  - There are different ways to calculate n!
    - e.g. n! = n * (n-1)!

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

Chain of deferred operations

```
(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
```

→ With substitution model (lecture 1)

→ recursive function, recursive process

linear recursive process

Linear Recursion / Iteration (2)

- Another perspective
  - n! = 1 * 2 * 3 * …. * n
- → maintain a running product, together with a counter that counts from 1 up to n

```
(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
```

product ← counter * product
counter ← counter + 1

Linear Recursion / Iteration (3)

- Another perspective
  - n! = 1 * 2 * 3 * …. * n
- → maintain a running product, together with a counter that counts from 1 up to n

```
(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
```

```
(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)))
```

→ See tail-recursion

→ recursive function, iterative process

linear iterative process

Tail recursion (“Endrekursion”)

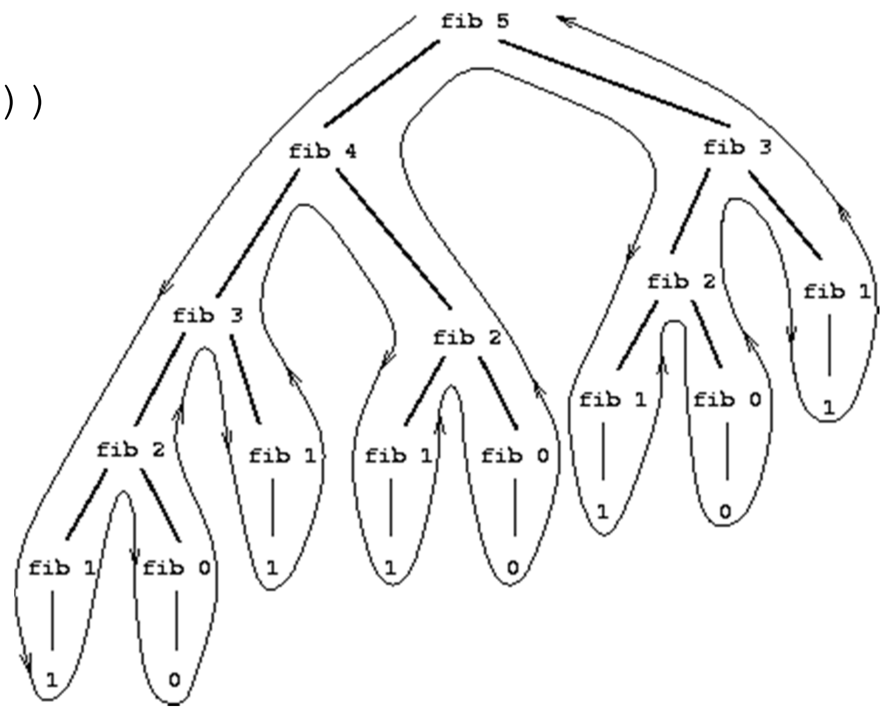
- tail call
  - A call of a subroutine (procedure) that is performed as the final action of a procedure
- tail recursive
  - If this call is a call to the subroutine itself

  - Can be implemented efficiently (depends on language/compiler)
    - Tail optimization (e.g. Racket)
      - No new stack frame for each call (reuse & replace)

Tree Recursion

- Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

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



Note: **tree recursive** processes are natural and **powerful tools** to operate on **hierarchical data structures**

In general, the number of steps required by a tree-recursive process will be proportional to the number of nodes in the tree, while the space required will be proportional to the maximum depth of the tree.

Higher-Order Procedures

- Until now (also in 1st semester)
  - Functions (aka methods in Java, procedures) whose parameters were numbers, booleans, Objects

A more powerful abstraction mechanism:

- Higher-Order Procedures
  - Accept **procedures as arguments** (parameters)

  or

  - **Return procedures**

……… increase the expressiveness of a language!

- **“First-class”** (Christopher Strachey, 1960s)
  - A prog.lang. has **first-class functions** if it treats functions as first-class citizens.
  - A **first-class citizen** is an entity which supports all the operations generally available to other entities.

    e.g. being passed as a parameter, returned from a function, and assigned to a variable

Procedures as arguments [Motivation]

- ```
  (define (sum-integers a b)
    (if (> a b)
        0
        (+ a (sum-integers (+ a 1) b))))
  ```

Computes the sum of the integers from a through b

E.g. (sum-integers 1 4)

How does this look like?

Note: this is not a higher-order-procedure

Demo2-1
  ```

Procedures as arguments [Motivation]

- ```
  (define (sum-integers a b)
    (if (> a b)
        0
        (+ a (sum-integers (+ a 1) b))))
  ```

  Computes the sum of the integers from a through b

- ```
  (define (sum-cubes a b)
    (if (> a b)
        0
        (+ (cube a) (sum-cubes (+ a 1) b))))
  ```

  Computes the sum of the cubes of the integers in the given range

- ...

Differences:

- **Name** of the procedure
- The function of `a` used to compute the **term** to be added
- The function that provides the **next** value of `a`.

General pattern:

```
(define (<name> a b)
  (if (> a b)
      0
      (+ (<term> a)
         (<name> (<next> a) b))))
```

Procedures as arguments

Copied from last slide:

General pattern:

```
(define (<name> a b)
  (if (> a b)
      0
      (+ (<term> a)
         (<name> (<next> a) b))))
```

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

The presence of such a pattern is a strong evidence that there is some useful abstraction ……

cf. Sigma notation

$$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b)$$

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

Demo2-1

How would this look like in Java?

... in Java (version < 8)

```
interface Func {
  int eval(int x);
}

public static int sum(Func term, int a, Func next, int b) {
  if(a > b)  return 0;
  else return term.eval(a) + sum(term, next.eval(a), next, b);
}

public static int sumIntegers(int a, int b) {
  return sum( new Func() { public int eval(int x) { return x; } }, a, new Func() { public int eval(int x) { return x + 1; } }, b);
}

public static int sumEverySecondSquare(int a, int b) {
  return sum( new Func() { public int eval(int x) { return x * x; } }, a, new Func() { public int eval(int x) { return x + 2; } }, b);
}

public static void main(String[] args) {
  System.out.println("sumIntegers(1, 10): " + sumIntegers(1, 10));
  System.out.println("sumEverySecondSquare(1, 10): " + sumEverySecondSquare(1, 10));
  System.out.println("sumEveryThirdCube(1, 10): " + sum( new Func() { public int eval(int x) { return x * x * x; } }, 1, new Func() {
public int eval(int x) { return x + 3; } }, 10));
}
```

See FuncProg_02_higherOrderFuncs.java

In Java8: .... to come!

λ Lambda … anonymous procedure

- Lambda is used to create procedures in the same way as define, except that no name is specified for the procedure.

- E.g. `(lambda (x) (+ x 4))`

- (lambda (<*formal params*>) <*body*>)

- E.g. `((lambda (x y z) (+ x y (square z))) 1 2 3)`
  → 12

Note: Use **lambda** or **λ**

<Strg> + <\>

<Strg> + <?ß\>

Demo2-1b

Procedures as Returned Values

- We will see an example in a few slides…

That’s it for today

References

-SICP
-Brian Harvey, Matthew Wright. Simply Scheme
-Robert W. Sebesta. Concept of Programming Languages, Pearson.
-Felleisen at al. Realm of Racket
-Ravi Chugh. Slides:UCSD: CSE 130 [Winter 2014] Programming Languages
-Neal Ford. Functional Thinking

Thank you!