Knowledge-based systems in Bioinformatics, 1MB602

Scheme lecture 2

Procedural abstraction and data abstraction

Lecture Overview

- Procedural abstraction
  - Higher order procedures
  - Procedures as arguments
  - Procedures as returned values
- Local variables
- Data abstraction
  - Compound data
  - Principles of data abstraction

LCB THE LINNAEUS CENTRE FOR BIOINFORMATICS http://www.lcb.uu.se

Procedural abstraction

1. 1 + 2 + ... + 100 = (100 * 101)/2 ← $\sum_{k=1}^{100} k$
2. 1 + 4 + 9 + ... + 100^2 = (100 * 101 * 201)/6 ← $\sum_{k=1}^{100} k^2$

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

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

Generalized:

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

LCB THE LINNAEUS CENTRE FOR BIOINFORMATICS http://www.lcb.uu.se

The sum procedure

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

- What is the type of this procedure?

(number → number, number, number → number, number) → number

term proc, a, next proc, b — sum procedure

LCB THE LINNAEUS CENTRE FOR BIOINFORMATICS http://www.lcb.uu.se

Higher order procedures

- A higher order procedure:

  takes a procedure as an argument or returns one as a value

```
(define (sum-integers a b)
 (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))

(define (sum-squares a b)
   (sum square a (lambda (x) (+ x 1)) b))
```

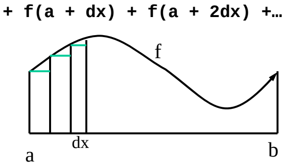
- The sum procedure is a higher order procedure

LCB THE LINNAEUS CENTRE FOR BIOINFORMATICS http://www.lcb.uu.se

Integration as a procedure

Integration under a curve is given roughly by

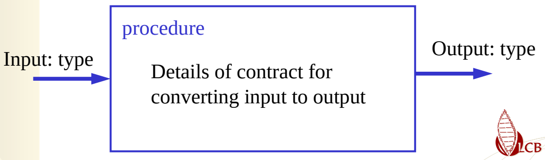
dx(f(a) + f(a + dx) + f(a + 2dx) +…+ f(b))




```
(define (integral f a b)
    (* (sum f a (lambda (x) (+ x dx)) b) dx))
(define dx 1.0e-3)
(define (atan a)
   (integral (lambda (x) (/ 1 (+ 1 (square x)))) 0 a))
```

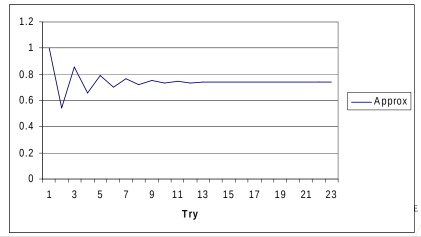
Procedural abstraction

- Process of procedural abstraction
  - Define formal parameters, capture process in body of procedure
  - Give procedure a name
  - Hide implementation details from user, who just invokes names to apply procedures (abstraction barrier)
  - Black box abstraction



Finding fixed points of functions

- Problem: find a *x* that satisfies $f(x) = x$
- Strategy for finding fixed points:
  - Given a guess x_1, let new guess be $f(x_1)$
  - Keep computing *f* for last guess, until close enough
- Example: find a *x* that satisfies $cos(x) = x$



FOR BIOINFORMATICS
http://www.lcb.uu.se

Finding fixed points of functions cont.

- Given a guess *x*, let new guess be *f(x)*
  - $f(x), f(f(x)), f(f(f(x))), f(f(f(f(x)))), \ldots$
- Keep computing *f* for last guess, until close enough

```
(define (fixed-point f guess)
  (define (close? u v)
     (< (abs (- u v)) 0.0001))
  (define (try g)
     (if (close? (f g) g) (f g)
          (try (f g))))
  (try guess))
```

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Using fixed points

- Computing the square root of *x* require finding a *y* such that $y^2 = x$, or $y = x/y$
- This is equivalent to looking for a fix point of the function $f(y) = x/y$

```
(define (sqrt x)
  (fixed-point
     (lambda (y) (/ x y))
     1))
```

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Using fixed points cont.

```
(sqrt 2)
(fixed-point (lambda (y) (/2 y)) 1)
(try 1)
…
(try 2)
…
(try 1)
…
```

$$y = x/y$$
$$(y+y)/2 = (x/y+y)/2$$
$$y = (x/y+y)/2$$

- (sqrt 2) oscillates between 1 and 2
- So damp out the oscillation…

```
(define (sqrt x)
  (fixed-point
   (damp
    (lambda (y)
          (/ x y)))
   1))
```

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

(define (damp f)
   (lambda (x)
     (average x (f x))))
```

Using let to create local variables

- Suppose we wish to compute the function:
  $f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$
- which we also could express as:
  $a = 1 + xy$
  $b = 1 - y$
  $f(x,y) = xa^2 + yb + ab$
- In Scheme:

```
(define (f x y)
   (define (f-help a b)
      (+ (* x (square a))
         (* y b)
         (* a b)))
   (f-help (+ 1 (* x y))
           (- 1 y)))
```

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Using let to create local variables cont.

```
(define (f x y)
   (define (f-help a b)
      (+ (* x (square a))
         (* y b)
         (* a b)))
   (f-help (+ 1 (* x y))
           (- 1 y)))
```

```
(define (f x y)
   ((lambda (a b)
      (+ (* x (square a))
         (* y b)
         (* a b)))
    (+ 1 (* x y))
    (- 1 y)))
```

```
(define (f x y)
   (let ((a (+ 1 (* x y)))
         (b (- 1 y)))
     (+ (* x (square a))
        (* y b)
        (* a b))))
```


Syntax of let expressions

- General form

```
(let ((<var_1> <exp_1>)
      (<var_2> <exp_2>)
      ...
      (<var_n> <exp_n>))
  <body>)
```

- var_i cannot depend on var_j
- If this is desired, use let*, e.g.

```
(let* ((a 3)
       (b (* 5 a)))
  (* a b))
```

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Compound data

- Need ways of gluing data elements together into a unit that can be treated as a simple data element
- Need ways of retrieving data elements
- Need a contract between the “glue” and the “unglue”
- Ideally want the result of this “gluing” to have the property of closure:
  - “the result obtained by creating a compound data structure can itself be treated as a primitive object and thus be input to the creation of another compound object”

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Pairs (cons-cells)

- `(cons <x-exp> <y-exp>)` → `<Pair>`

  Where `<x-exp>` evaluates to a value `<x-val>`, and `<y-exp>` evaluates to a value `<y-val>`

- `(car <Pair>)` → `<x-val>`

  Returns the car-part of the pair <P>

- `(cdr <Pair>)` → `<y-val>`

  Returns the cdr-part of the pair <P>

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Compound data cont.

- Treat a PAIR as a single unit:
  - Can pass a pair as argument
  - Can return a pair as a value

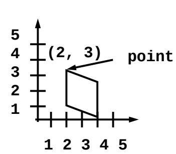
```
(define (make-point x y)
  (cons x y))

(define (point-x point)
  (car point))

(define (point-y point)
  (cdr point))

(define (make-seg pt1 pt2)
  (cons pt1 pt2))

(define (start-point seg)
   (car seg))
```



THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Pair abstraction

- Constructor

  `(cons <x> <y>)` → `<Pair>`
- Accessors

  `(car <Pair>)` → `<x>`

  `(cdr <Pair>)` → `<y>`
- Predicate

  `(pair? <z>)`

  → `#t` if `<z>` evaluates to a pair, else `#f`

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Pair abstraction cont.

- Note that there is a contract between the constructor and the accessors

```
– (car (cons <a> <b> )) → <a>
– (cdr (cons <a> <b> )) → <b>
```

- Note how pairs have the property of closure – we can use the result of a pair as an element of a new pair:

```
– (cons (cons 1 2) 3)
```


An example: rational numbers

- What are the elements of rational numbers n/d?
  - numerator: n
  - denominator: d

- A rational number is a ratio n/d
- a/b + c/d = (ad + bc)/bd
- a/b * c/d = (ac)/(bd)

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Rational number abstraction

1. Constructor
```
(make-rat <n> <d>)
```
2. Accessors
```
(numer <r>)
(denom <r>)
```
3. Contract
```
(numer (make-rat <n> <d>)) → <n>
(denom (make-rat <n> <d>)) → <d>
```
4. Layered Operations
```
(print-rat <r>)
(+rat x y)
(*rat x y)
```
5. Abstraction Barrier
```
Say nothing about implementation!
```

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Rational number abstraction cont.

1. Constructor
2. Accessors
3. Contract
4. Layered Operations
5. Abstraction Barrier

Elements of data abstraction

6. Concrete Representation & Implementation (can alternate!)

```
(define (make-rat n d) (cons n d))
(define (numer r) (car r))
(define (denom r) (cdr r))
```

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Layered rational numbers operation

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))

(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Testing our procedures

```
(define one-half (make-rat 1 2))
(define three-fourths (make-rat 3 4))
(define new (+rat one-half three-fourths))
(print-rat new)
10/8
```

Oops – should be 5/4 not 10/8 !

Rationalize implementation

- Strategy 1: remove common factors when accessing numer and denom
- Strategy 2: remove common factors when creating a rational number

THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Implementation – strategy 1

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

(define (numer r)
  (let ((g (gcd (car r) (cdr r))))
    (/ (car r) g)))

(define (denom r)
  (let ((g (gcd (car r) (cdr r))))
    (/ (cdr r) g)))

(define (make-rat n d)
  (cons n d))
```


Implementation – strategy 2

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

(define (numer r) (car r))

(define (denom r) (cdr r))

(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g)
          (/ d g))))
```

LCB
THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Data abstraction cont.

- What is a pair?
  - If we "glue" two objects together using `cons` we can retrieve the objects using `car` and `cdr`
  - We don't know the implementation only that Scheme supplies procedures (`cons`, `car`, `cdr`) for operating on pairs
- We could implement `cons`, `car`, and `cdr` without using any data structures but only using procedures
- Requirement: contract between constructor and accessor
  - **(car (cons <a> <b>)) → <a>**
  - **(cdr (cons <a> <b>)) → <b>**

LCB
THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

Data abstraction cont.

```scheme
(define (cons x y)
  (define (dispatch m)
      (cond ((= m 0) x)
            ((= m 1) y)
            (else (error "Wrong arg for disp"))))
  dispatch)

(define (car z) (z 0))
(define (cdr z) (z 1))
```

- `(cons x y)` returns a procedure (higher order procedure)
- This style of programming is often called message passing.

LCB
THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se

References

- H. Abelson, G.J. Sussman, Structure and Interpretation of Computer Programs 2nd ed, The MIT Press, Cambridge, Massachusetts, 2000, Chp: 1.3, 2.1, pp: 56-76, 79-94
- 6.001 Spring 2000: Lecture Notes, lecture 4, 5, 6, http://sicp.ai.mit.edu/Spring-2000/lectures/

LCB
THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se