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 3: Higher-Order Procedures

Slide 1.3. 1

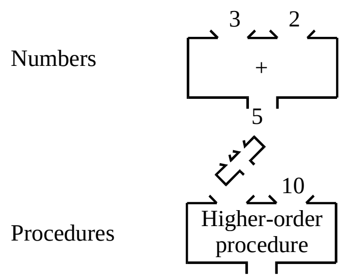
Numbers and procedures are first class

- If something is first class, it can be manipulated (used as argument)
- Numbers are first class:
  - in stead of: (* 2 2)
    (* 3 3)

    you do: (define (square x) (* x x))
    (square 2)
    (square 3)
- New: Procedures are also first class !!!

Slide 1.3. 2

Higher-order procedures



Slide 1.3. 3

Higher-order procedure

Implement $\sum_{i=1}^{n} f(n)$

```
( define   ( square  x )
     ( *     x x ))

( define   ( sum    f  n )
     ( if    ( =    n  0 )
             0                          ; identity of '+' operation
             ( +    ( f    n )
                    ( sum    f   ( -    n  1 ))))))

( sum   square  10 )    ⟶    385
```

Slide 1.3. 4

Higher-order procedure: product

$\prod_{i=a}^{b}$ f (n) = f (a) * ... * f (b)

For example:

fac (n) = $\prod_{i=1}^{n} i$

a=1
b=n
f = identity function

Slide 1.3. 5

Higher-order procedure: product

```
( define   ( product    f   a  b )
     ( if    ( >   a  b )
             1                          ; identity of '*' operation
             ( *    ( f  a )
                    ( product    f    ( +    a  1 ) b ))))
```

Slide 1.3. 6

Using higher-order procedure, fac revisited

```
( define    ( fac   n )
    ( define   ( null   a ) a )              ; Identity function
    ( product   null  1  n ))
```

Slide 1.3. 7

Product: iterative version

```
( define    ( product-iter   f  b  count  result )
    ( if    ( >   count  b )
            result
            ( product-iter
                  f
                  b
                  ( +   count  1 )
                  ( *   result  ( f  count )))))

( define    ( product   f  a  b )
    ( product-iter    f  b  a  1 ))
```

Slide 1.3. 8

Mr. Skiem, session 1.3.a, b

Higher order functions:
- sum
- product

Slide 1.3. 9

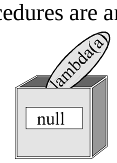
Fac, again

```
( define   ( fac  n )
     ( define   ( null  a )  a )
     ( product  null  1  n  ))
```

- The procedure <null> is not meaningfull
- The name <null> of procedure <null> is never used outside ‘fac’, it could as well be anonymous !

Anonymous functions λ

- Sometimes anonymous functions are needed
  - procedure as argument to higher order procedure
  - procedure as return value
- The value of procedures are anonymous functions



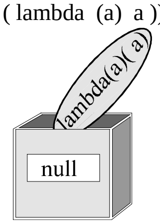

```
( define   ( null  a )  a )
```


Lambda, constructing procedure

```
( define   ( null  a )  a )
```

equal

```
( define   null  ( lambda  (a)  a ))
```



Fac, once again

```
( define     ( null  a )  a )
( define     ( fac    n )
     ( product      null   1   n ))

                equal

( define     ( fac   n )
     ( product     ( lambda (a) a )  1   n ))
```

- The procedure passed to higher-order procedure <product> has no name, it is anonymous !

Slide 1.3. 13

Using lambda functions

- The anonymous procedure that is attached to the name <square> is:

```
( lambda   (a)   (*   a  a ))
```

- Using the name <square>, a number is squared as:

```
( square   10 )
```

- Using the anonymous procedure it can be squared as:

```
(( lambda   (a)   ( *   a  a ))  10 )
```

Slide 1.3. 14

Mr. Skiem, session 1.3.c

Anonymous functions

Slide 1.3. 15

Needing an extra hand

f (n) = n! + (n!)^2

```
( define    ( f   n )
     ( +     ( fac  n )
             ( square    ( fac  n ))))
```

- (fac n) is calculated twice, expensive

Slide 1.3. 16

An extra hand, dedicated variables

f (n) = n! + (n!)^2

```
( define    ( f   n )
     ( define    tmp    ( fac   n ))
     ( +    tmp
            ( square   tmp )))
```

Slide 1.3. 17

An extra hand, anonymous procedure

f (n) = n! + (n!)^2

```
( define    ( f   n )
     (( lambda    ( tmp )
           ( +    tmp
                  ( square  tmp )))
      ( fac   n )))
```

Slide 1.3. 18

An extra hand, special form <let>

f (n) = n! + $(n!)^2$

```
( define    ( f   n )
    ( let    (( tmp    ( fac   n )))
             ( +    tmp
                    ( square   tmp ))))
```

- Using <let> is exactly the same as using an anonymous function (previous slide)

Slide 1.3. 19

Let, general format

```
(let ((var1 exp1)
      (var2 exp2)
      .
      .
      (varn expn)
  body)
```

Equivalence:

```
((lambda (var1 var2 .. varn)
   body)
 exp1 exp2 .. expn)
```

Slide 1.3. 20

Let, traps and pittfalls

- The scope of let-variables is the let-block

```
( define    x  5 )
( +    ( let    (( x  3 ))
            ( +   x    ( *    x    10 )))
       x )  ⟶  38
```

- Let-variables are bound using variables outside the let-block

```
( define    x   2 )
( let    (( x   3 )  ( y    ( +    x   2 )))
     ( *    x   y ))  ⟶  12
```

Slide 1.3. 21

Mr. Skiem, session 1.3.d

Using Let

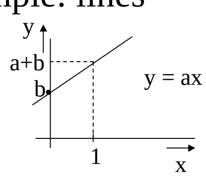
Slide 1.3. 22

Procedures as returned value

- So far we have shown how to use procedure objects as input parameters
- Even more expressive power can be achieved by letting procedures return procedure objects
- Examples:
  - lines
  - derivative

Slide 1.3. 23

Example: lines

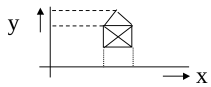


For example:

```
( define   ( y   x )
     ( +    ( *   x  5 )  10 ))
( y  0 ) → 10
( y  1 ) → 15
```

Slide 1.3. 24

Straight lines, lots



- Typing in 'defines' for each line is too much work
- Using lambda, no explicit line names are needed

  (lambda (x) (+ (* x 5) 10))

  (lambda (x) (+ (* x 1) 2))
- y-values can be calculated using these anonymous procedures

  ((lambda (x) (+ (* x 5) 10)) 0) → 10

  ((lambda (x) (+ (* x 1) 2)) 0) → 2

Slide 1.3. 25

Straight lines, lots

- Typing anonymous procedure for a lot of straight lines is still to much work
- We want to generate anonymous procedures

  We want a procedure that returns an anonymous procedure

```
( define   ( line  a  b )
     ( lambda  (x)
          ( +  ( *  x  a )  b )))
(( line  5  10 )  0 )  →  10 )
(( line  5  10 )  1 )  →  15 )
```

Slide 1.3. 26

Example: derivative

- Dg(x) = (g(x+dx) - g(x)) / dx

```
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)
(define dx 0.00001)
(define (cube x) (* x x x))
((deriv cube) 5)  ==> 75.00015
```

Slide 1.3. 27

Why automatically generated anonymous functions

- Names, don’t need them
- Abstraction, don’t need implementation details
- First class, procedures as argument and return value

Slide 1.3. 28