# Programming Languages (Langages Evolué)

Roel Wuyts

Scheme Basics

# Scheme: Ideal

"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today." [R5RS]

# Scheme History

- Lisp
- CommonLisp
  - Everything and more
  - Union of a large number of dialects
- Scheme:
  - 75/78 first versions, 84/88/92/96 normalisation
  - statically scoped + block structure
  - first  class escape
  - single namespace + no position
  - full language description + semantics in 50 pages
  - -> C

# Scheme naming conventions

- ...? for predicates
  - e.g. equal?, boolean?
- ...! for side-effect
  - e.g. set!
- *global*
- char-, string-, vector- procedures
- type1->type2: conversion

# Basic Principles

- Execution principle
- Naming
- Basic elements
- Quoting and Quasi-quoting
- Procedures
- Special forms
- Recursion
- higher-order procedures

# Execution Principle

- Read-Eval-Print:
  - Read an expression
  - Evaluate it
  - Print the result and loop

# Read-Eval-Print

- Using a Scheme interpreter

  > (+ 1 3)
  4

- In this document

  ```
  (+ 1 3) => 4
  ```

- Note: Scheme uses prefix notation
  - makes it easy to variable number of arguments
  - makes nesting easy

# Some Simple Examples

```
4                   ;; self-evaluable
=> 4

(* 5 6)             ;; applying *
=> 30

(+ 2 4 6 8)         ;; applying +
=> 20

(* 4 (* 5 6))       ;; nested expressiosn
=> 120

(* 7 (- 5 4) 8)
=> 56

(- 6 (/ 12 4) (* 2 (+ 5 6)))
=> -19
```

# Naming

- A name identifies a variable whose value is the object
- Naming is done with define

```
(define size 2)
size => 2

(* 5 size) => 10

(define pi 3.14159)
(define radius 10)
(define circumference (* 2 pi radius))
circumference => 62.8318
```

# Basic Elements

- S-expressions (symbolic expressions) - a.k.a *forms*
- simple data types:
  - booleans
  - numbers
  - characters
  - symbols
  - strings
  - dotted pairs

# Booleans

- `#t` and `#f`
- Examples:
  - `(boolean? #t) => #t`
  - `(boolean? “hello”) => #f`
- Note: self-evaluating:
  - `#t => #t`
- *not* returns `#t` only if the argument is `#f`:
  - `(not ’()) => #f`

# And

- (and test1 test2 .... testn)
  - *and* is lazy:
    - evaluation goes from left to right.
    - first test that evaluates to `#f` ->result is `#f`
    - if there are no expression, result is `#t`
- Examples:
- ```
  (and (= 2 2) (> 2 1)) => #t
  (and (= 2 2) (< 2 1)) => #f
  (and 1 2 'c '(f g))   => (f g)
  ```
- Similar for *or*
  ```

# Numbers

- Self-evaluating

```
1.2 => 1.2
1 => 1
2+3i => 2+3i
```

- Number predicates

```
(number? 42) => #t
(complex? 2+3i) => #t
(real? 2+3i) => #f
(real? 3.1416) => #t
(real? 22/7) => #t
(rational? 3.1416) => #t
(rational? 22/7) => #t
(integer? 22/7) => #f
(integer? 42) => #t
```

# Number Comparison

- Using the general-purpose equality predicate eqv? :

```
(eqv? 42 42 )        => #t
(eqv? 42 #f )         => #f
(eqv? 42 42.0 )      => #f
```

- Using the special number-equality predicate = :

```
(=42 42 )               => #t
(=42 #f )               => ERROR!!!
(=42 42.0)             => #t
```

- Other number comparisons: <,<= ,>,>= .

# Some operations on numbers

```
(+ 1 2 3) => 6
(- 5.3 2) => 3.3
(* 1 2 3) => 6
(/22 7) => 22/7
(expt 23) => 8
(expt 4 1/2) => 2.0
(-4) => -4
(/4) => 1/4
(max 1 3 4 2 3) => 4
(min 1 3 4 2 3)=> 1
(abs 3) => 3
(abs -4) => 4
```

# Characters

- letter prefixed by `#\`
- self-evaluating

  `#\a => \#a`
- Some constants: `#\tab #\space or #\, #\newline`
- Comparison predicates:
  - `char=?`, `char<?`, `char<=?`, `char>?`, `char>=?`
  - use `ci` to make case insensitive (e.g. `char-ci=?`, ...)
- `char-downcase` and `char-upcase`

# Symbols

- For referencing variables
- Not self-evaluating: evaluate to value of variable
- Can be manipulated using ’ (*quote*)
- Examples

```
(symbol? 'xyz)                  => #t
(symbol? 42)                    => #f
(eqv? 'Calorie 'calorie)        => #t
```

# Strings

- "abc d d"
- self-evaluating

```
"abc" => "abc"

(string #\s #\q #\u #\e #\a #\k) => "squeak"

(string-ref "squeak" 3) => #\e

(string-append "sq" "ue" "ak")    => "squeak"

(define hello "hello")
(string-set! hello 1 #\a)
hello => "hallo"

(string? hello) => #t
```

# Some conversions

```
(char->integer #\d) => 100
(integer->char 50) => #\2

(number->string 16) => “16”
(integer? (string->number “16”)) => #t
(string->number “Am I a hot number?”)=> #f
(symbol->string ‘symbol )=> “symbol”
(string->symbol “string” )=> string

(string->list “me”) => (#\m #\e )
list->string, vector->list,  and list->vector
```

# Dotted Pair

- Two values => ordered couple
  - `(cons ‘a ‘b) => (a . b)`
  - `(car (cons ‘a ‘b)) => a`
  - `(cdr (cons ‘a ‘b)) => b`
- Not self-evaluating

```
‘(1 . #t)    =>    (1 . #t)
(1 . #t)     =>    Error
```

# Nested Dotted Pairs

```
(cons (cons 1 2) 3) => ((1 . 2) . 3)

(car (cons (cons 1 2) 3)) => (1 . 2)

(cdr (car (cons (cons 1 2) 3))) => 2

(car (car (cons (cons 1 2) 3))) => 1

(caar (cons (cons 1 2) 3)) => 1

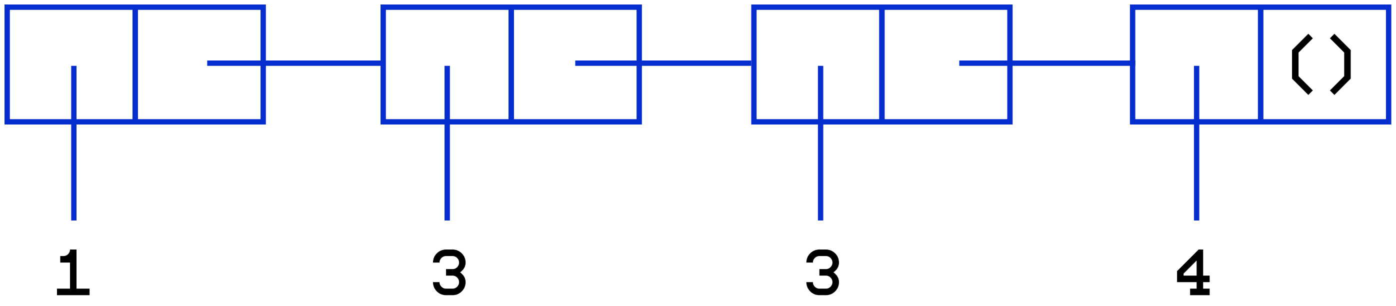

(cons 1 (cons 2 (cons 3 (cons 4 5))))
 => (1 2 3 4 . 5) <=> (1. ( 2. ( 3. ( 4 . 5))))
```

# Lists

- Empty list: ()
```
'() => ()
```

- Some examples:
```
(list 1 (+ 1 2) 3 4) => (1 3 3 4)
(list 'a 'b 'c) => (a b c)
'(1 2 3 4) => (1 2 3 4)

(null? '(a b)) => #f
(null? '()) => #t
```



# Lists and dotted pairs

- A list is a dotted pair whose second element is the empty list
  So: `(cons 1 '())  =>  (1 . ())  <=>  (1)`
  Or: `(1 2 3 4) <=> (1 . (2 . (3 . (4 . ()))))`

- So this works as well:

```
(car '(a b c d)) => a
(cdr '(a b c d)) => (b c d)
(cons 'a '(b c d)) => (a b c d)
```

- Note: the empty list is not a pair!

```
(car '()) => Error
(pair? '()) => #f
(null? '()) => #t
```

# Some list procedures

- (append list1 list2)

```
(append '(a b c) '(d e)) => (a b c d e)
(append '(a b c) '()) => (a b c)
```

- Some other ones that are used less often:

```
(define y (list 1 2 3 4))
(list-ref y 0)=> 1
(list-ref y 3)=> 4
(list-tail y 1)=> (234)
(list-tail y 3)=> (4)
```

# Quoting

- Have already seen:
  `‘Hello => hello`
- Quoting takes the expression ‘literally’
  - No evaluation is done
  - forces lists to be treated as data
    `’(a b c) = (list ‘a ‘b ‘c) ≠ (list a b c)`
  - allows us to manipulate symbols (like the ‘a)
- The ’ syntax is actually syntactic sugar for *quote*
  `(quote Hello) => hello`

# Concept: Syntactic Sugar

- Special syntactic forms that are simply convenient alternative surface structures for things that can be written in more uniform ways are sometimes called syntactic sugar
  - So ‘abbreviations’
  - Goal: to ease readability or writability
- Will see more about this later in Scheme
  - as well as in other languages

# Quasi-quoting

- Punches "holes" in quoted expression
- `s , syntactic sugar for (quasiquote s)
  '(a b c) => (a b c)

  (define b 4)
  `(a ,b c) => (a 4 c)
  `(a ,b ,c) => ERROR

# Procedures

- Evaluate
  `cons => #<primitive:cons>`
  - cons refers to the primitive cons procedure
- Some expressions to think about:
  ```
  (car '(+ 1 2)) => +
  (car (list + 1 2)) => #<primitive:+>
  ```

# Rolling your own

- (define (name arguments)
    body)
- Example:

```
(define (double x)
   (* x 2))

double => #<procedure:double>

(double 4) => 8
```

# Procedures and lambda

- (define (name arguments)
      body)
  =
  (define name (lambda (arguments) body))

# Arguments for procedures

- Two possibilities:
  - list of symbols
  - dotted pair

# List of symbols

- Used most often
- Example

```
(define (myadd x y)
    (+ x y))

(myadd (+ 1 2) 3) => 6

(mylist (+1 2) 2 3) => error
```

# Dotted Pair

- Used for variable arity procedures
- Example

```
(define (weirdo x y . rest)
   (list x y rest))
(weirdo 1 2 4 5 6 0 9) => (1 2 (4 5 6 0 9))

(define (F x) x)
(define (G . x) x)
(F 1 2 3) => ERROR
(G 1 2 3) => (1 2 3)
```

# Special Forms

- Normal procedure application: applicative order:
  - first evaluate operator
    then evaluate operands (order unspecified)
    then apply procedure to arguments
- For special forms this is *not* the case!
  E.g. `(if (= x 0) 1 3)`
  - other special forms: *define*, *cond*, *if*, *quote*, ...
- Later we will write our own special forms
  - extremely important feature!!

# If

- (if *test whenTrue whenFalse*)
- Example

```
(define (sign x)
   (if (< x 0) -1 1))

(sign 34) => 1
(sign -3.45) => -1
```

# cond

- (cond
  (*predicate-expression1 action1*)
  (*predicate-expression2 action2*)
  ...
  (else *actionN*))

- Example

```
(define (val expr)
   (cond
      ((number? expr) 1)
      ((list? expr) 2)
      (else 3)))
```

# case

- (case *expression*
  ((*choice choice* ...) *expression*)
  ((*choice choice* ...) *expression*)
  ...
  (else *expression*))
- Example

```
(define (useless expr)
  (case (remainder expr 6)
    ((0 2 3) "ahum")
    ((1 4 5) "oomph")
    (else "auch")))
```

# lambda

- (lambda (*name name* ...) *expression*)
- Defines a procedure
  - remember: nameless
  - can be returned, passed as argument, ...
    - used frequently to build procedure on the fly
- Example

```
(lambda (x) (+ x 2)) => #<procedure:5:2>
((lambda (x) (+ x 2)) 6) => 8
```

# apply

- (apply *proc-expr argList*)
- apply a function using items from a list as the arguments
- Example

```
(apply + (list 1 2 3 4)) => 10

(define plusAliass +)
(apply plusAlias '(1 2 3 4)) => 10
```

# Recursion

- Faculty:

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

(fac 5) => 120

(require (lib “trace.ss”))
(trace fac)
(fac 5)
```

# Tracing recursive faculty

```
(require (lib “trace.ss”))
(trace fac)
(fac 5)

|(fac 5)
| (fac 4)
| |(fac 3)
| | (fac 2)
| | |(fac 1)
| | | (fac 0)
| | | 1
| | |1
| | 2
| |6
| 24
|120
```

# Iterative version

```
(define (fac n)
   (define (fac-iter counter result)
      (if (> counter n)
         result
         (fac-iter
            (+ counter 1)
            (* counter result))))
(fac-iter 1 1))
```

# Trace result for iterative version

```
>>> (fac 5)
Computing (#<PROCEDURE fac> 5)
Computing (#<PROCEDURE fac-iter> 1 1)
   Computing (#<PROCEDURE fac-iter> 2 1)
      Computing (#<PROCEDURE fac-iter> 3 2)
         Computing (#<PROCEDURE fac-iter> 4 6)
            Computing (#<PROCEDURE fac-iter> 5 24)
               Computing (#<PROCEDURE fac-iter> 6 120)
               (#<PROCEDURE fac-iter> 6 120) --> 120
            (#<PROCEDURE fac-iter> 5 24) --> 120
         (#<PROCEDURE fac-iter> 4 6) --> 120
      (#<PROCEDURE fac-iter> 3 2) --> 120
   (#<PROCEDURE fac-iter> 2 1) --> 120
(#<PROCEDURE fac-iter> 1 1) --> 120
(#<PROCEDURE fac> 5) --> 120
120
```

# Concept: Higher-Order

- Have already seen: functions are first-class
- A higher-order function is a function that takes other functions as arguments
- Example

```
(define compose
     (lambda (g f)
         (lambda (x)
             (g (f x)))))

((compose
   (lambda (x) (+ x 2))
   (lambda (x) (* 3 x))) 7) => 23
```

# Order of Functions

- The order of data
  - Order 0: Non function data
  - Order 1: Functions with domain and range of order 0
  - Order 2: Functions with domain and range of order 1
  - Order k: Functions with domain and range of order k-1

# Enumerating lists

- (map *procedure list1 list2 ... listN*)
- to construct a new list by applying a function to each item on one or more existing lists
- Example

```
(map (lambda (x) (* x x)) '(1 2 3))
=> (1 4 9)

(map (lambda (x y) (+ x y)) '(2 3) '(10 10))
=> (12 13)
```

# Some other list predicates

```
(filter (lambda (x) (> x 0)) '(-2 3 4 -5 6)) => (3 4 6)

(member (list 'a) '(b (a) c)) =>  ((a) c)

(reverse '(a b c))                 =>  (c b a)
(reverse '(a (b c) d (e (f)))) =>  ((e (f)) d (b c) a)
```

# Wrap-up

- Scheme: functional programming language
  - basic structures
  - defining procedures
  - applying procedures
    - special forms
  - recursion vs. iteration
  - higher-order procedures

# References

- http://www.ulb.ac.be/di/rwuyts/INFO020_2003/

- H. Abelson, G.J. Sussman, J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.