Vrije Universiteit Brussel
Faculteit van de Wetenschappen
Vakgroep Informatica

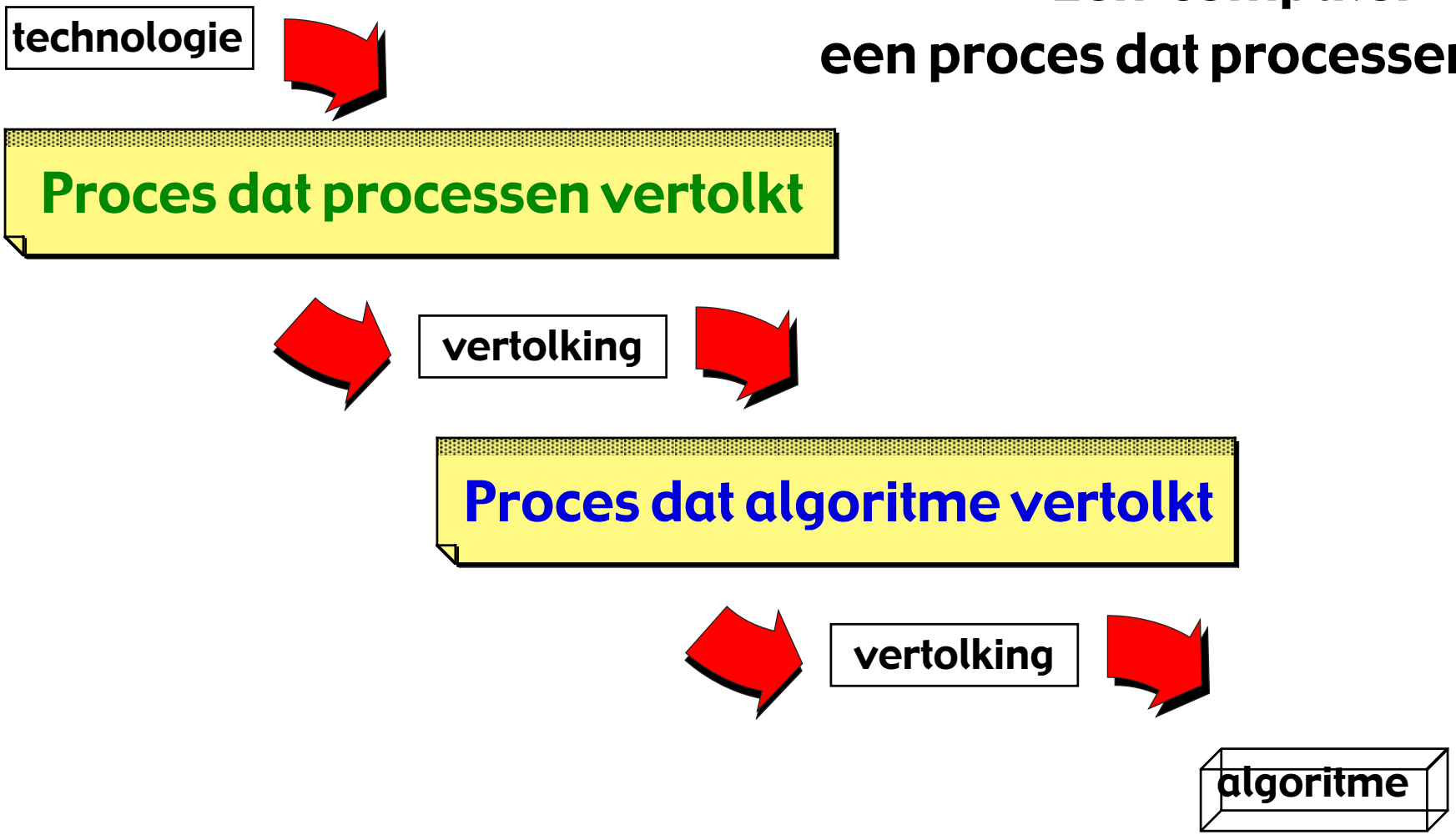
Deel 1a:
Metacirculaire evaluatie

Interpretatie van Computerprogramma's I

Theo D'Hondt

Metacirculaire evaluatie

Een computer =
een proces dat processen uitvoert

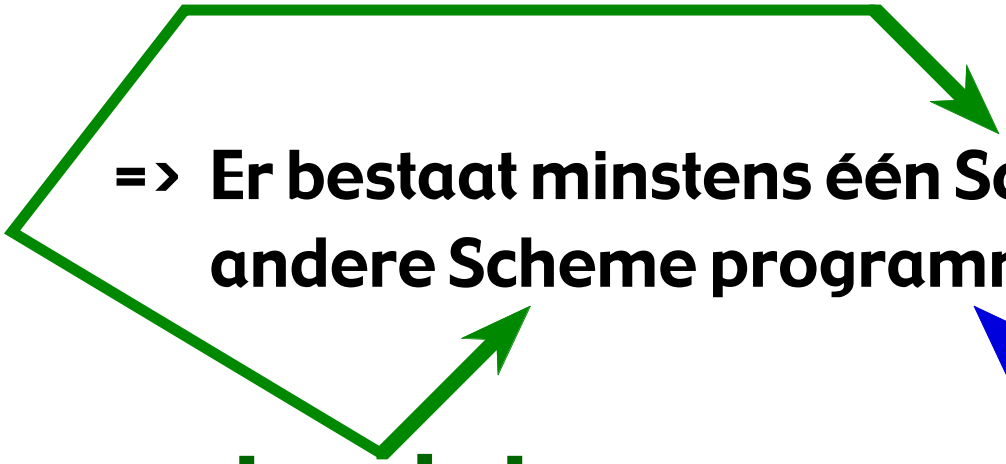


Metacirculaire evaluatie

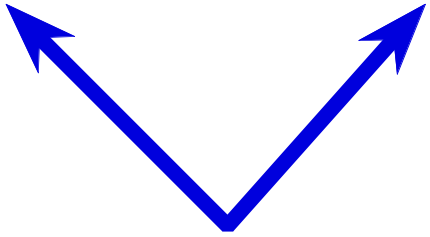
Scheme = taal voor het beschrijven van processen

=> Scheme kan worden gebruikt voor het beschrijven van een proces dat processen vertolkt

=> Er bestaat minstens één Scheme programma dat andere Scheme programma's kan interpreteren

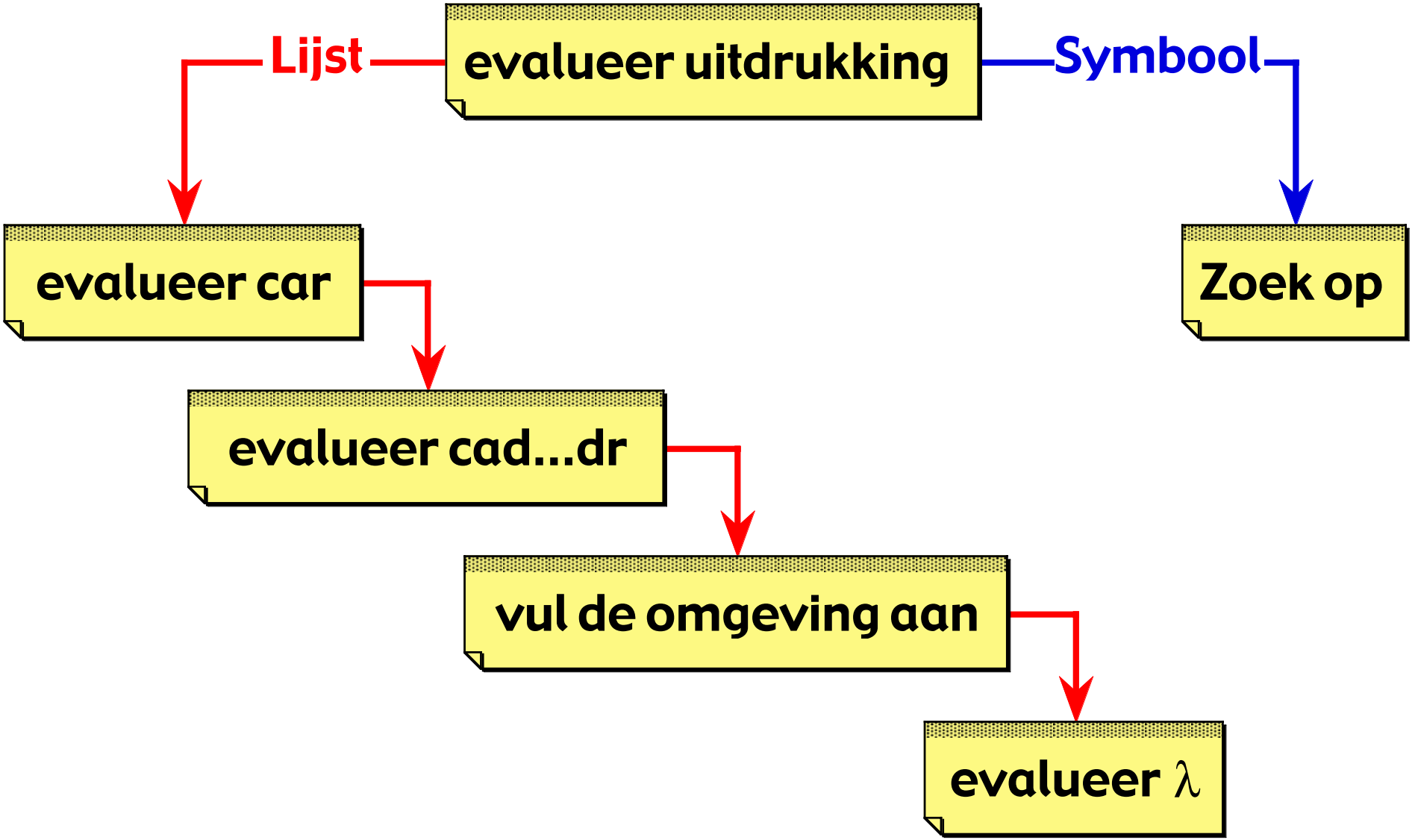


circulair

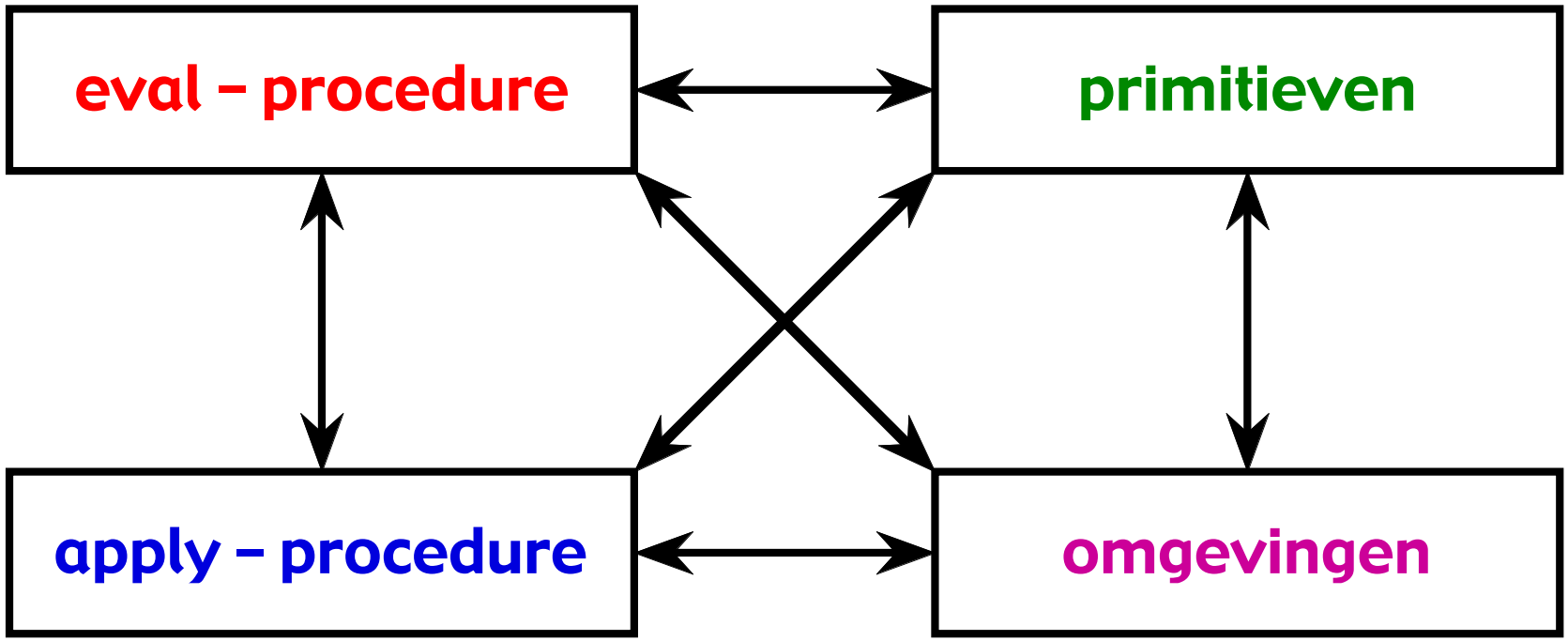


meta

Het evaluatieproces:



Struktuur van het evaluatieproces



De procedure «eval»

evaluatie van een uitdrukking «exp» binnen een omgeving «env»

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)
        ((variable? exp)
         (lookup-variable-value exp env))
        ((quoted? exp)
         (text-of-quotation exp))
        ((assignment? exp)
         (eval-assignment exp env))
        ((definition? exp)
         (eval-definition exp env))
        ((if? exp)
         (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp)
         (eval (cond->if exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

De procedure «eval»

```
(define (eval exp env)
  (cond ((self-evaluating? exp)
         exp)                                          } constanten
        ((variable? exp)
         (lookup-variable-value exp env))              } variabele
        ((quoted? exp)
         (text-of-quotation exp))                      } quotes
        ((assignment? exp)
         (eval-assignment exp env))                    } binding
        ((definition? exp)
         (eval-definition exp env))                    } define
        ((if? exp)
         (eval-if exp env))                            } if
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)       } procedure
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))      } begin
        ((cond? exp)
         (eval (cond->if exp) env))                    } cond
        ((application? exp)
         (apply (eval (operator exp) env)              } apply
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

De procedure «apply»

evaluatie van een procedure «procedure» met argumenten «arguments»

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

De procedure «apply»

evaluatie van een primitieve procedure

evaluatie van een zelfgedefinieerde procedure

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```

opzoeken van de procedure body

uitbreiding van de actuele omgeving met bindingen tussen actuele en formele argumenten

De procedure «list-of-values»

evaluatie van een lijst uitdrukkingen «exps» binnen een omgeving «env»

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))
```

- no-operands?, first-operand en rest-operands zijn operaties op lijsten van uitdrukkingen.
- list-of-values verdeelt eval over de uitdrukkingen uit de lijst: het resultaat is een lijst van waarden bekomen door de uitdrukkingen één-voor-één te evalueren

De procedure «eval-assignment»

evaluatie van een toekenning «exp» binnen een omgeving «env»

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

- **assignment-value** en **assignment-variable** zijn operaties op toekenningen
- **set-variable-value!** is een operatie op omgevingen
- **eval-assignment** evalueert het rechterlid van een toekenning, en kent deze waarde, binnen de omgeving, toe aan de variabele uit het linkerlid; het resultaat van eval-assignment is **'ok**

De procedure «eval-definition»

evaluatie van een definitie «exp» binnen een omgeving «env»

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (eval (definition-value exp) env)
                    env)
  'ok)
```

- `definition-value` en `definition-variable` zijn operaties op definitie
- `define-variable!` is een operatie op omgevingen
- `eval-definition` evalueert het rechterlid van een definitie, en bindt deze waarde, binnen de omgeving, aan de variabele uit het linkerlid; het resultaat van `eval-definition` is `'ok`
- indien de variabele reeds bestaat in de lokale *frame*, is `eval-definition` hetzelfde als `eval-assignment`

De procedure «eval-if»

```
(define (true? x)
  (not (eq? x false)))

(define (false? x)
  (eq? x false))
```

evaluatie van een selectie «exp» binnen een omgeving «env»

```
(define (eval-if exp env)
  (if (n (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

- `if-predicate`, `if-consequent` en `if-alternative` zijn operaties op selecties
- `true?` is een operatie op predikaatwaarden
- `eval-if` selecteert en evalueert de *consequent* indien het *predicate* evalueert naar de waarde *true* zoniet gebeurt dit met de *alternative*

De procedure «eval-sequence»

evaluatie van een reeks van uitdrukkingen «exps» binnen een omgeving «env»

```
(define (eval-sequence exps env)
  (cond
   ((last-exp? exps)
    (eval (first-exp exps) env))
   (else
    (eval (first-exp exps) env)
    (eval-sequence (rest-exps exps) env))))
```

- `last-exp?`, `first-exp` en `rest-exps` zijn operaties op reeksen van uitdrukkingen
- `eval-sequence` evalueert één-voor-één de uitdrukkingen van een reeks; enkel de laatst berekende waarde blijft behouden en dient als resultaat van `eval-sequence`

Voorstelling van uitdrukkingen

uitdrukkingen die na evaluatie zichzelf opleveren

```
(define (self-evaluating? exp)
  (cond ((number? exp) true)
        ((string? exp) true)
        (else false)))
```

- **`self-evaluating?` is een predikaat dat enkel waar is voor getallen of tekst**

Voorstelling van uitdrukkingen

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))
```

quote uitdrukkingen
→
'uitdrukking

```
(define (quoted? exp)
  (tagged-list? exp 'quote))

(define (text-of-quotation exp) (cadr exp))
```

- *quote*-uitdrukkingen zijn koppels van de vorm:

- `quoted?` is een predikaat dat test of een uitdrukking een *quote* is
- met `text-of-quotation` kan de symbolische waarde van de uitdrukking opgevraagd worden

Voorstelling van uitdrukkingen

variabelen

→

variabele

```
(define (variable? exp) (symbol? exp))
```

- het predikaat `variable?` test of een uitdrukking een variabele is

Voorstelling van uitdrukkingen

toekenningen

→

(set! *variabele uitdrukking*)

```
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))
```

- toekenningen zijn tripels van de vorm:

'set!	variabele	uitdrukking

- het predikaat `assignment?` test of een uitdrukking een toekenning is
- met `assignment-variable` en `assignment-value` kunnen de variabele en de uitdrukking opgevraagd worden

Voorstelling van uitdrukkingen

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp) (cddr exp))))
```

definities

→

(define *variabele uitdrukking*)

of

(define (*variabele lijst*) *reeks*)

- **definities zijn 3-tupels van de vorm:**

'define	variabele	uitdrukking

- **`definition?` is een predikaat dat test of een uitdrukking een definitie is**
- **met `definition-variable` en `definition-value` kunnen de variabele en de uitdrukking opgevraagd worden**

Voorstelling van uitdrukkingen

```
(define (definition? exp)
  (tagged-list? exp 'define))

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp) (cddr exp))))
```

definities

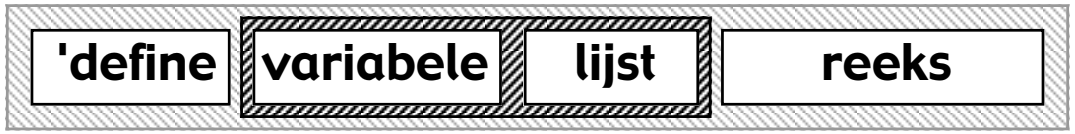
→

(define variabele uitdrukking)

of

(define (*variabele lijst*) *reeks*)

- definities zijn 3-tupels van de vorm:



- worden als synoniem beschouwd:

(define (proc a_1 ... a_n) exp_1 ... exp_m)

(define proc (lambda (a_1 ... a_n) exp_1 ... exp_m))

Voorstelling van uitdrukkingen

```
(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      'false))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))
```

selecties

→

(if *predikaat gevolg alternatief*)

- selecties zijn 4-tupels van de vorm:

'if	predikaat	gevolg	alternatief

- ☑ `if?` is een predikaat dat test of een uitdrukking een selectie is
- ☑ met `if-predicate`, `if-consequent` en `if-alternative` kunnen predikaat, gevolg en alternatief opgevraagd worden
- ☑ met `make-if` kan een selectie aangemaakt worden

Voorstelling van uitdrukkingen

abstracties

→

(lambda *lijst reeks*)

```
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))

(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))
```

- abstracties zijn 3-tupels van de vorm:

'lambda	lijst	reeks

- ☑ `lambda?` test of een uitdrukking een abstractie is
- ☑ met `lambda-parameters` en `lambda-body` kunnen de parameters en de reeks van uitdrukkingen opgevraagd worden
- ☑ met `make-lambda` kan een abstractie aangemaakt worden

Voorstelling van uitdrukkingen

cond-uitdrukkingen
→
(cond *clausules*)

```
(define (cond? exp) (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))
```

- *cond*-uitdrukkingen zijn koppels van de vorm:

'cond	clausules

- cond? test of een uitdrukking een *cond*-uitdrukking is
- met cond-clauses kunnen de clausules opgevraagd worden
- cond-else-clause? test of een clausule de *else*-clausule is
- met cond-predicate en cond-actions worden het predikaat en de acties van een clausule opgevraagd

```
(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      'false
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))
```

clausule
→
(*predikaat reeks*)
of
(else *reeks*)

```
: (cond->if '(cond ((> x y) (set! x (- x y)))
                   ((< x y) (set! y (- y x)))
                   (else x)))
(if (> x y) (set! x (- x y)) (if (< x y) (set! y (- y x)) x))
```

- **cond->if wordt gebruikt om een *cond*-uitdrukking om te vormen naar een reeks van geneste *if*-uitdrukkingen**
- **expand-clauses is een staartrecursieve hulpfunctie**

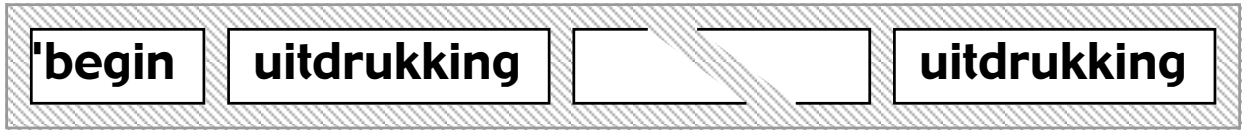
```
(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))

(define (first-exp seq) (car seq))

(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))

(define (make-begin seq) (cons 'begin seq))
```

reeksen

→

(begin *uitdrukking ... uitdrukking*)

- **reeksen zijn lijsten van uitdrukkingen**



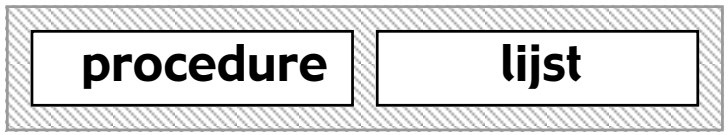
- **met `begin-actions` worden de acties opgevraagd**
- **met `last-exp?` , `first-exp` en `rest-exps` kunnen de uitdrukkingen opgevraagd worden**
- **met `sequence->exp` en `make-begin` kan een *begin*-uitdrukking aangemaakt worden**

Voorstelling van uitdrukkingen

```
(define (application? exp) (pair? exp))

(define (operator exp) (car exp))

(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))

(define (first-operand ops) (car ops))

(define (rest-operands ops) (cdr ops))
```

toepassingen
→
(procedure *lijst*)

- toepassingen zijn koppels van de volgende vorm:



- ☑ `application?` test of een uitdrukking een toepassing is
- ☑ met `operator` en `operands` kan kan de procedure en de lijst van argumenten van een toepassingen opgevraagd worden
- ☑ met `no-operands?` , `first-operand` en `rest-operands` kunnen de argumenten opgevraagd worden

Voorstelling van uitdrukkingen

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))

(define (procedure-body p) (caddr p))

(define (procedure-environment p) (cadddr p))
```

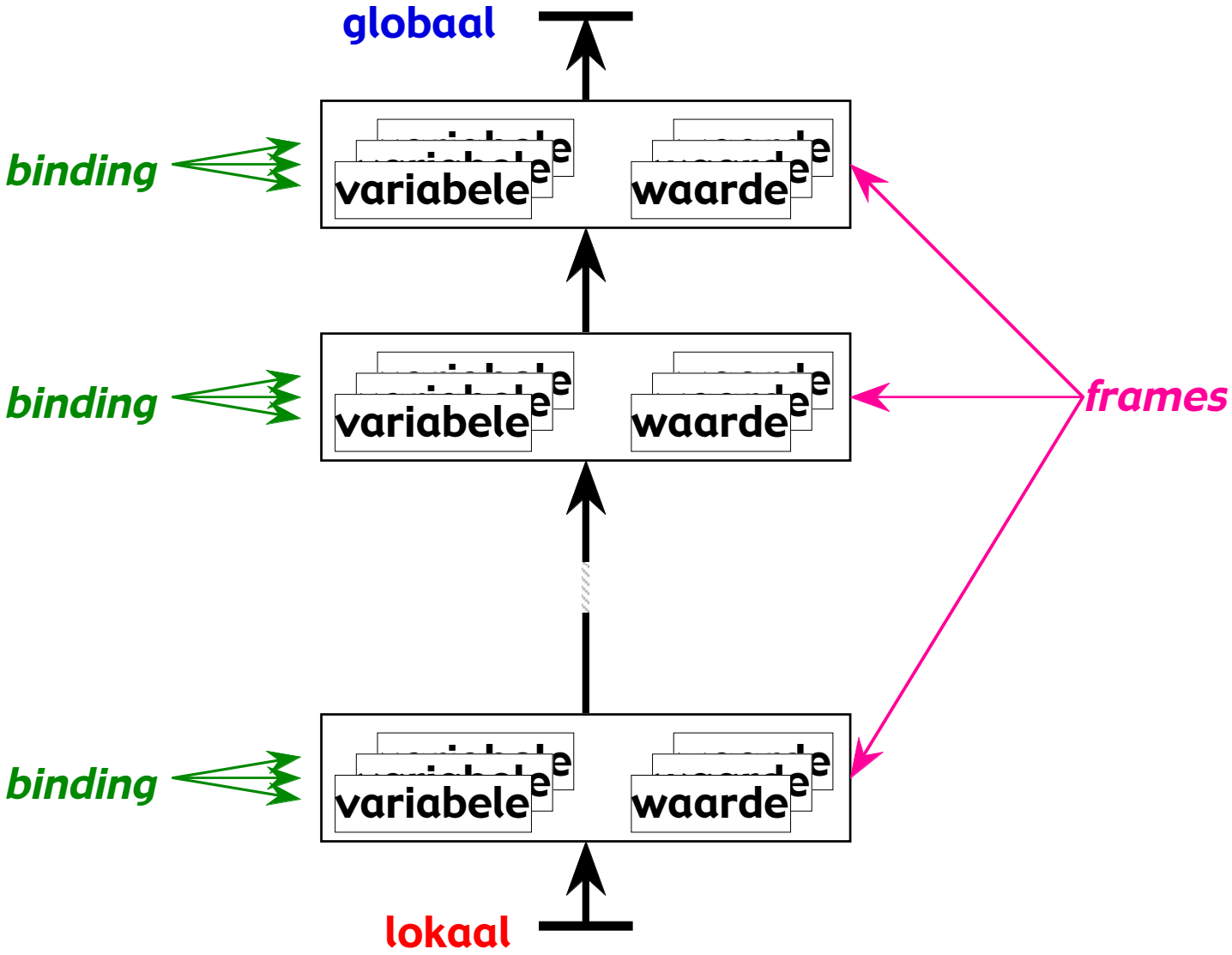
procedure
→
abstractie ⊕ omgeving

- procedures zijn 4-tupels van de vorm:

'procedure	parameters	reeks	omgeving

- ☑ `make-procedure` kombineert parameters, een uitdrukking en een omgeving tot een procedure
- ☑ `compound-procedure?` test of een uitdrukking een procedure is
- ☑ `parameters`, `procedure-body` en `procedure-environment` geven toegang tot de formele argumentelijst, de reeksvan uitdrukkingen en de omgeving van de procedure

Omgevingen


globaal
binding
variabele
waarde
binding
variabele
waarde
frames
binding
variabele
waarde
lokaal

Omgevingen

operaties op omgevingen

```
(define (enclosing-environment env) (cdr env))

(define (first-frame env) (car env))

(define the-empty-environment '())
```

- `enclosing-environment` zoekt de omsluitende omgeving
- `first-frame` zoekt de eerste *frame* in een omgeving
- `the-empty-environment` maakt een lege omgeving aan

Omgevingen

uitbreiden van omgevingen

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

- `extend-environment` maakt een nieuwe *frame* aan boven op een bestaande omgeving
- de lijst van variabelen `vars` en de lijst van overeenstemmende waarden `vals` wordt in deze *frame* ondergebracht

Omgevingen

```
(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))

(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

bewerkingen met *frames*

- `make-frame` maakt een *frame* aan uitgaande van een lijst van variabelen `variables` en de lijst van overeenstemmende waarden `values`
- `frame-variables` en `frame-values` geeft de lijst van variabelen en de lijst van overeenstemmende waarden uit de *frame* `frame`
- `add-binding-to-frame!` zal een *frame* uitbreiden met een variabele `var` en een waarde `val`

Omgevingen

opzoeken van variabelen

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

- `lookup-variable-value` zoekt een variabele `var` op in een omgeving `env` en bepaalt de gebonden waarde ervan

Omgevingen

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

wijzigen van variabelen

- **`set-variable-value!`** zoekt een variabele **`var`** op in een omgeving **`env`** en vervangt de gebonden waarde ervan door **`val`**

Omgevingen

definiëren van variabelen

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

- `define-variable!` zoekt een variabele `var` op in een omgeving `env` en vervangt de gebonden waarde ervan door `val`; indien niet gevonden, wordt een nieuwe binding tussen `var` en `val` aangemaakt binnen de lokale *frame*

Primitieven

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))

(define the-global-environment (setup-environment))
```

globale declaraties

- `primitive-procedure-names` definieert de lijst van namen van primitieve operaties
- `primitive-procedure-objects` definieert de lijst van primitieve objecten
- `setup-environment` creeërt een globale omgeving bestaande uit de hoogste *frame*, en wordt opgeroepen om `the-global-environment` te definiëren

Primitieven

```
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))

(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
;;      more primitives
        ))

(define (primitive-procedure-names)
  (map car primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))
```

primitieve procedures

- ☑ primitieve operaties zijn koppels met de volgende gedaante

'primitive	implementatie

- ☐ primitive-procedure? test of een procedure primitief is

Primitieven

toepassing van primitieve procedures

```
(define apply-in-underlying-scheme apply)

(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

- `apply-in-underlying-scheme` is de versie van `apply` in de meta-*Scheme*
- `apply-primitive-procedure` past een primitieve functie toe in de meta-*Scheme*

Evaluatie

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))

(define the-global-environment (setup-environment))

(driver-loop)
```

de *read-eval-print* lus

```
(define input-prompt
  ";;; M-Eval input:")
(define output-prompt
  ";;; M-Eval value:")
```

Test

```
Welcome to DrScheme, version 102.
Language: Graphical Full Scheme (MrEd).

;;; M-Eval input:
(define (append x y)
  (if (null? x)
      y
      (cons (car x)
            (append (cdr x) y))))

;;; M-Eval value:
ok

;;; M-Eval input:
(append '(a b c) '(d e f))

;;; M-Eval value:
(a b c d e f)

;;; M-Eval input:
```