

1. Meta-circulaire evaluatie

Boek sectie 4.1.1 tot en met 4.1.5

Context in cursus

- performantie-experiment
- taalontwerp-experiment

...
M-Eval
meval.scm
Interpreter voor Scheme,
geschreven in Scheme.
Implementeert het
omgevingsmodel.
(~ mechanisatie van lamda-
calculus)

...
D-Eval
dynamiceval.scm
Interpreter voor variant van
Scheme met *dynamic* in
plaats van *lexical* scope,
geschreven in Scheme
(~ oudere Lisp dialecten zoals
Emacs Lisp en AutoCad Lisp)

...
A-Eval
analyzingeval.scm
Geoptimaliseerde interpreter
voor Scheme met *normal* in plaats
van *applicative* volgorde van
evaluatie van argumenten,
geschreven in Scheme
(~ Haskell)

...
L-Eval
lazyeval.scm
Interpreter voor variant van
Scheme met *normal* in plaats
van *applicative* volgorde van
evaluatie van argumenten,
geschreven in Scheme
(~ Assembly, JVM als
interpreter voor Java
bytecode)

Reg-Sim
regsim.scm

Interpreter voor
registermachine-taal,
geschreven in Scheme
(~ Assembly, JVM als
interpreter voor Java
bytecode)

M-Eval meval.scm

Interpreter voor Scheme,
geschreven in Scheme.

Implementeert het
omgevingsmodel.

(~ mechanisatie van lamda-
calculus)

...
M-Eval input:
((lambda (x) (+ x 1)) 3)

...
M-Eval value:
4

...
(+ x 1)) 3

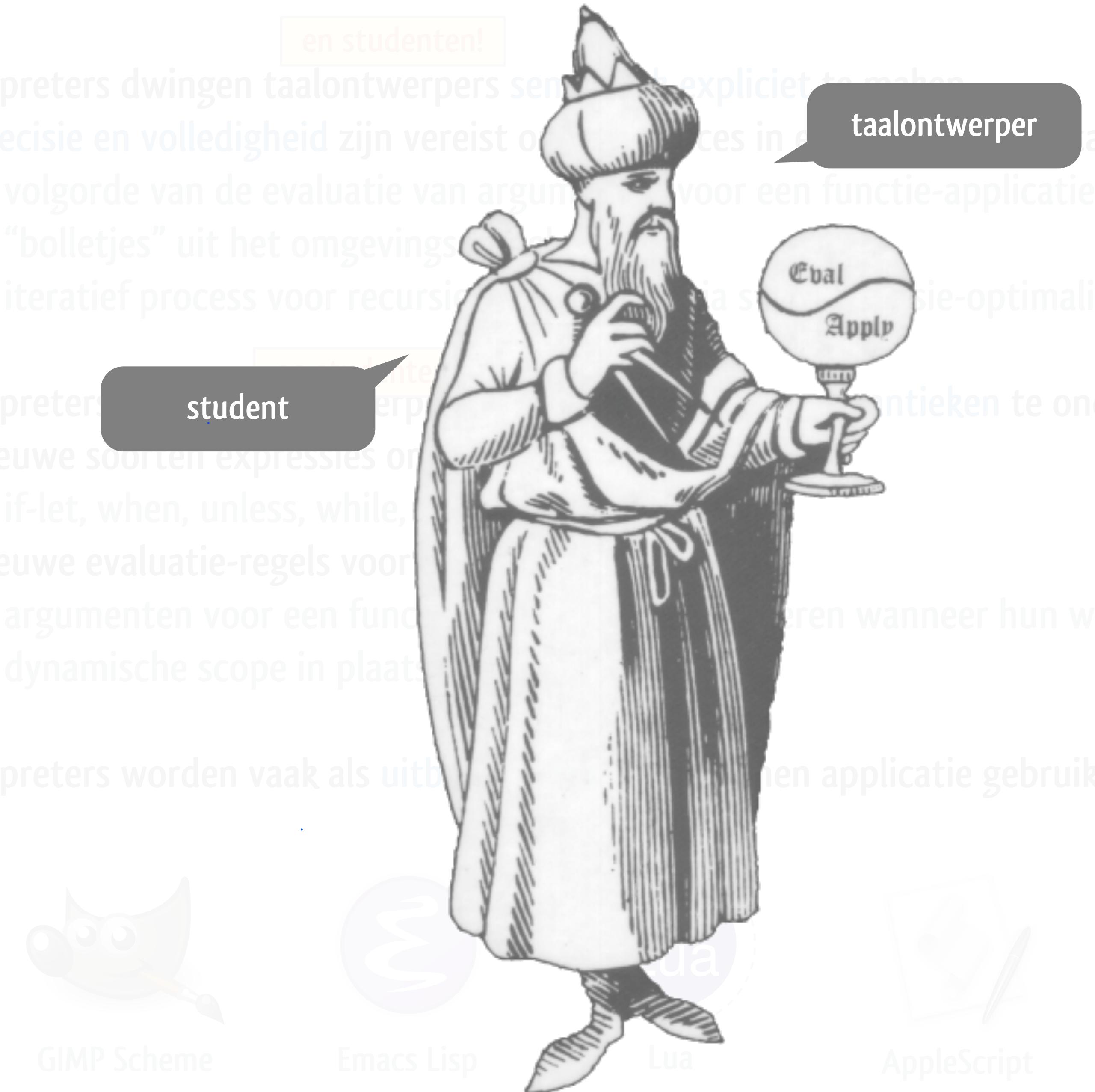
input → Compiler
compiler.scm → output

...
Query input:
(is-planeet ?p)
Query results:
(is-planeet neptunus)
(is-planeet uranus)
(is-planeet saturnus)
(is-planeet jupiter)
(is-planeet mars)
(is-planeet aarde)
(is-planeet venus)
(is-planeet mercurius)

...
Q-Eval
qevel.scm
Interpreter voor
een logische
programmeertaal.
(~ Prolog; mechanisatie van
predicaten-logica)

Waarom interpreters bestuderen

- interpreters dwingen taalontwerpers om expliciet te beschrijven wat de taal uit te drukken
- precisie en volledigheid zijn vereist om verschillende cases in de taal te ondersteunen
- volgorde van de evaluatie van argumenten voor een functie-applicatie
- “bolletjes” uit het omgevingsscope
- iteratief process voor recursieve definities → scope-optimalisatie
- interpreters zijn interessant voor studenten die methodische antieken te onderzoeken
- nieuwe soorten expressies ontdekken
- if-let, when, unless, while, ...
- nieuwe evaluatie-regels voor functies
- argumenten voor een functie kunnen worden gebundeld en alleen gebruikt wanneer hun waarde nodig is
- dynamische scope in plaats van statische
- interpreters worden vaak als uitbreiding van een bestaande applicatie gebruikt



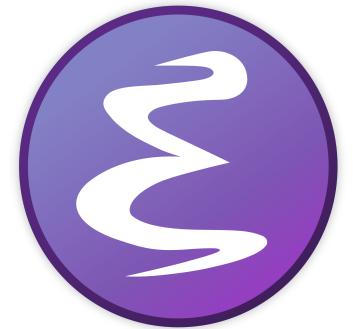
Waarom interpreters bestuderen

en studenten!

- interpreters dwingen taalontwerpers **semantiek expliciet** te maken
 - **precisie en volledigheid** zijn vereist om een proces in een programmeertaal uit te drukken
 - volgorde van de evaluatie van argumenten voor een functie-applicatie
 - omgevingsmodel zonder figuren
 - iteratief proces voor recursieve procedure via staartrecursie-optimalisatie
- interpreters stellen taalontwerpers in staat **alternatieve semantieken** te onderzoeken
 - nieuwe soorten expressies ondersteunen
 - if-let, while, throw ...
 - nieuwe syntax of semantiek voor expressies
 - argumenten voor een functie-applicatie pas evalueren wanneer hun waarde nodig is
 - dynamisch bereik in plaats van lexicaal bereik
- interpreters worden vaak als **uitbreidingsmedium** binnen applicatie gebruikt



GIMP Scheme



Emacs Lisp



AutoCAD Lisp

in Scheme

1.1 Een meta-circulaire interpreter voor Scheme asdas

aanvaardt eigen
definitie als invoer

tolk

asd
sad

Interpretatie-fasen: read-eval-print

The screenshot shows the DrRacket IDE interface. The title bar reads "ICP1_1a0_meval.scm - DrRacket". The menu bar includes "File", "Edit", "Language", "Source", "Contracts", "Help", and "About". The toolbar features icons for saving, running, stopping, and other functions. The main window displays the welcome message: "Welcome to DrRacket, version 7.2 [3m]. Language: R5RS [custom]; memory limit: 128 MB." Below this, a pink box highlights the text ";;; M-Eval input:". In the editor area, the expression "(average 40 (+ 5 5))" is typed. To the right of the editor, a yellow button labeled "eof" is visible. A small preview window on the right shows the language selection: "Language: R5RS [custom]" and the beginning of the expression ";;; M-Eval input: (average 40 (+ 5 5))".

ICP1_1a0_meval.scm - DrRacket

ICP1_1a0_meval.scm ▾ (define ...) ▾     Run  Stop 

Language: R5RS [custom]; memory limit: 128 MB.

```
60.2
;;; M-Eval input:
(average 40 (+ 5 5))

;;; M-Eval value:
25

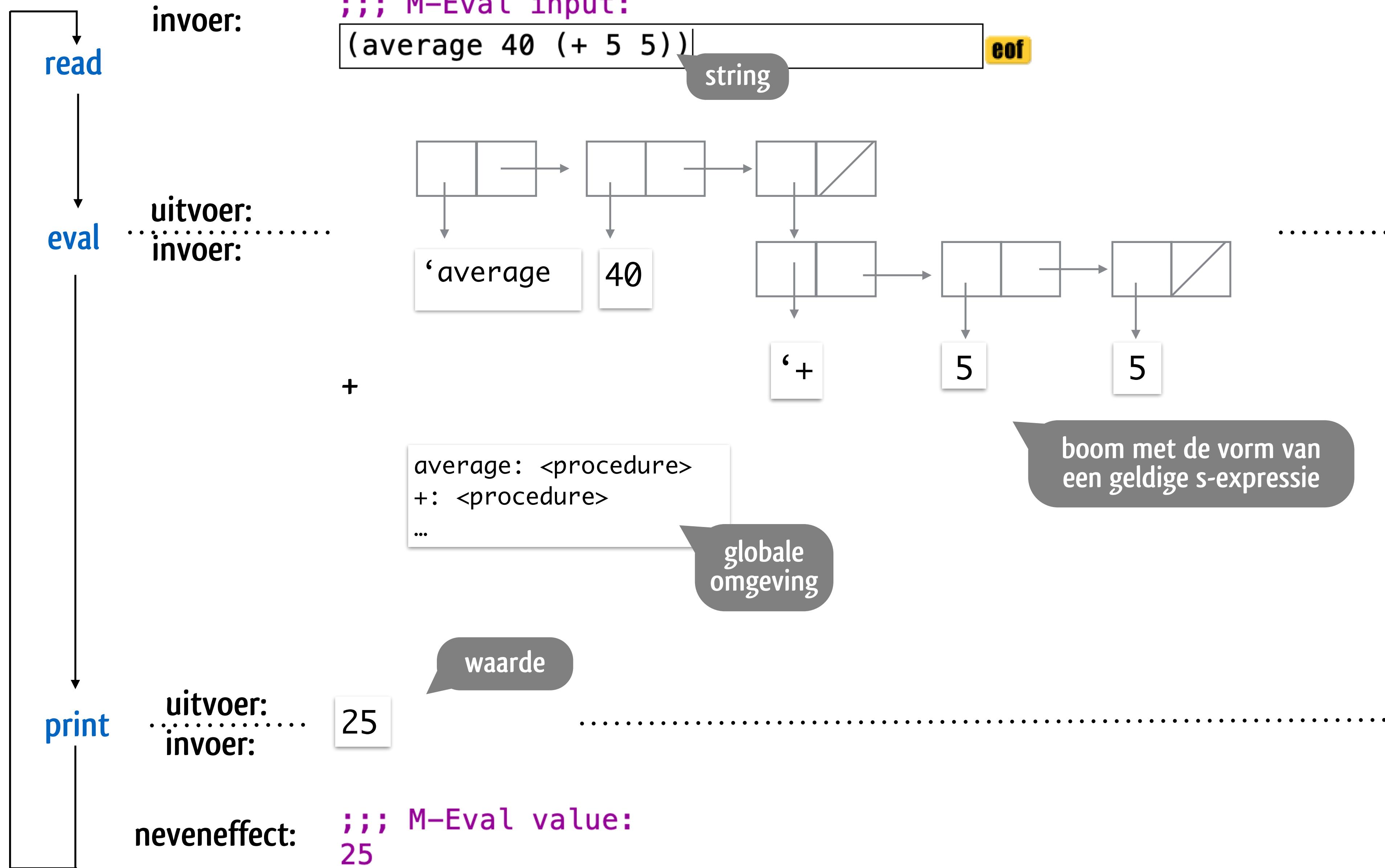
;;; M-Eval input:

eof
```

R5RS custom ▾ 12:2 452.17 MB  

```
graph TD; read[read] --> eval[eval]; eval --> print[print]; print --> end[ ]
```

Interpretatie-fasen: read-eval-print



Interpretatie-fase: read-eval-print



Implementatie van read-eval-print loop

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

```
(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 the-global-environment (setup-environment))
(driver-loop)
```



kern van eigen code

The screenshot shows the DrRacket interface with the file 'ICP1_1a0_meval.scm' open. The code is displayed in the editor area. In the interaction window below, the user has typed '(average 40 (+ 5 5))' and pressed enter. The system responded with '25'. A cursor is visible in the input field, and the word 'eof' is highlighted in yellow at the end of the input line.

```
ICP1_1a0_meval.scm - DrRacket
ICP1_1a0_meval.scm (define ...) Run Stop
Language: R5RS [custom]; memory limit: 128 MB.

;; M-Eval input:
(average 40 (+ 5 5))

;; M-Eval value:
25

;; M-Eval input:
eof

12:2 452.17 MB
```

Read-eval-print loop: print

```
(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)
                     '<env>))
      (display object)))
```

equivalent aan display

Waarom?

behalve voor expressies die naar een
procedure als waarde evalueren

Read-eval-print loop: `read`

voorzien
door Racket

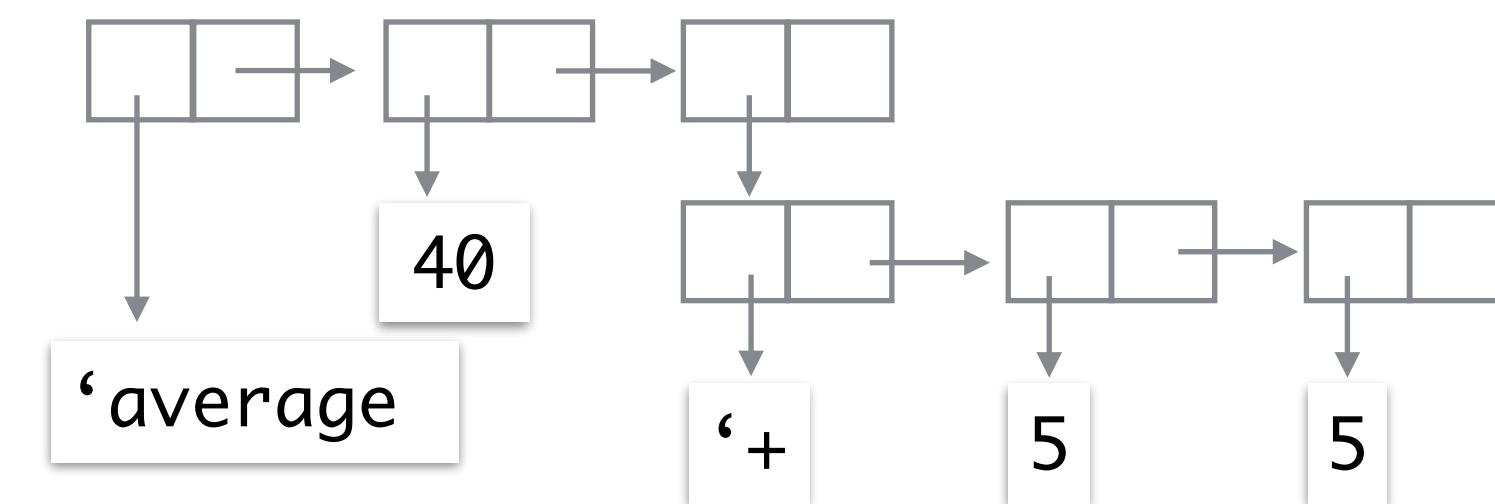
```
> (define x (read))
square
> (display x)
square
> (symbol? x)
#t
>
> (define y (read))
123
> (string? y)
#f
> (number? y)
#t
>
```

```
> (define z (read))
(average 40 (+ 5 5))
> (number? z)
#f
> (list? z)
#t
> (length z)
3
> (car z)
average
> (cddr z)
((+ 5 5))|
>
```

- vraagt de gebruiker om een **string als invoer**
- splitst de ingegeven string in een **lineaire reeks van tokens**
- vormt de reeks van tokens om naar een **boom in de vorm van een geldige s-expressie**

“(average 40
(+ 5 5))”

(average 40)
+ 5 5))



Read-eval-print loop: zonder read

```
> (average 40 (+ 5 5))
```

evaluatie van expressie

```
✖️✖️ average: undefined;
```

cannot reference an identifier before its definition

```
> '(average 40 (+ 5 5))
```

quotatie van expressie

```
(average 40 (+ 5 5))
```

```
> (define z '(average 40 (+ 5 5)))
```

```
> (car z)
```

average

```
> (symbol? (car z))
```

quote maakt dezelfde bomen als read!

```
#t
```

```
> (caddr z)
```

(+ 5 5)

```
> (number? (cadr (caddr z)))
```

```
#t
```

```
>
```



Homoiconicity

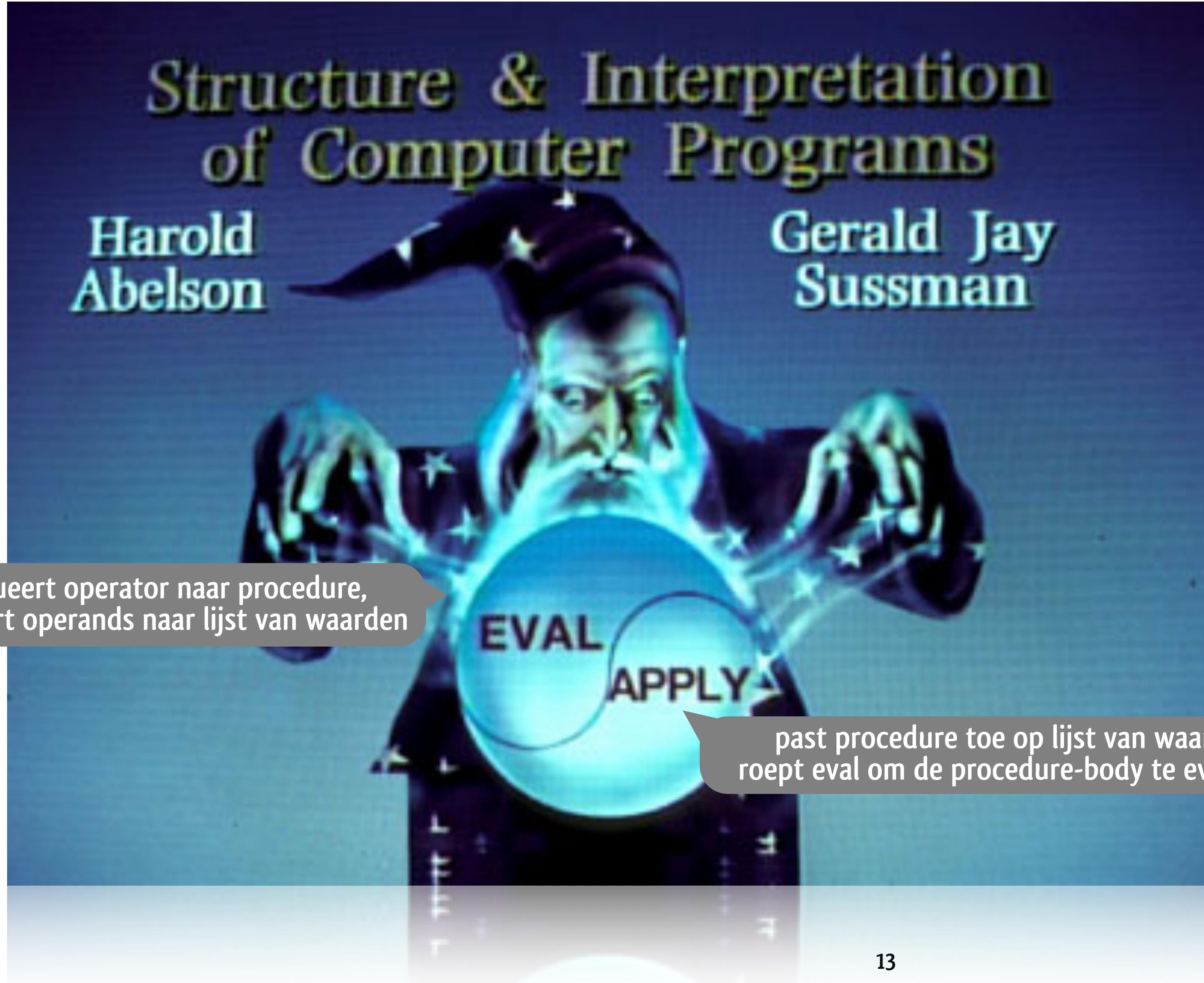
From Wikipedia, the free encyclopedia

In computer programming, **homoiconicity** (from the [Greek](#) words *homo-* meaning "the same" and *icon* meaning "representation") is a property of some [programming languages](#).

A language is **homoiconic** if a program written in it can be manipulated as data using the language, and thus the program's internal representation can be inferred just by reading the program itself. For example, a [Lisp](#) program is written as a regular Lisp list, and can be manipulated by other Lisp code.^[1] In homoiconic languages, all code can be accessed and transformed as data, using the same representation. This property is often summarized by saying that the language treats "[code as data](#)".

In a homoiconic language, the primary representation of programs is also a [data structure](#) in a [primitive type](#) of the language itself. This makes [metaprogramming](#) easier than in a language without this property: [reflection](#) in the language (examining the program's entities at [runtime](#)) depends on a single, homogeneous structure, and it does not have to handle several different structures that would appear in a complex syntax.

Structuur van de evaluator

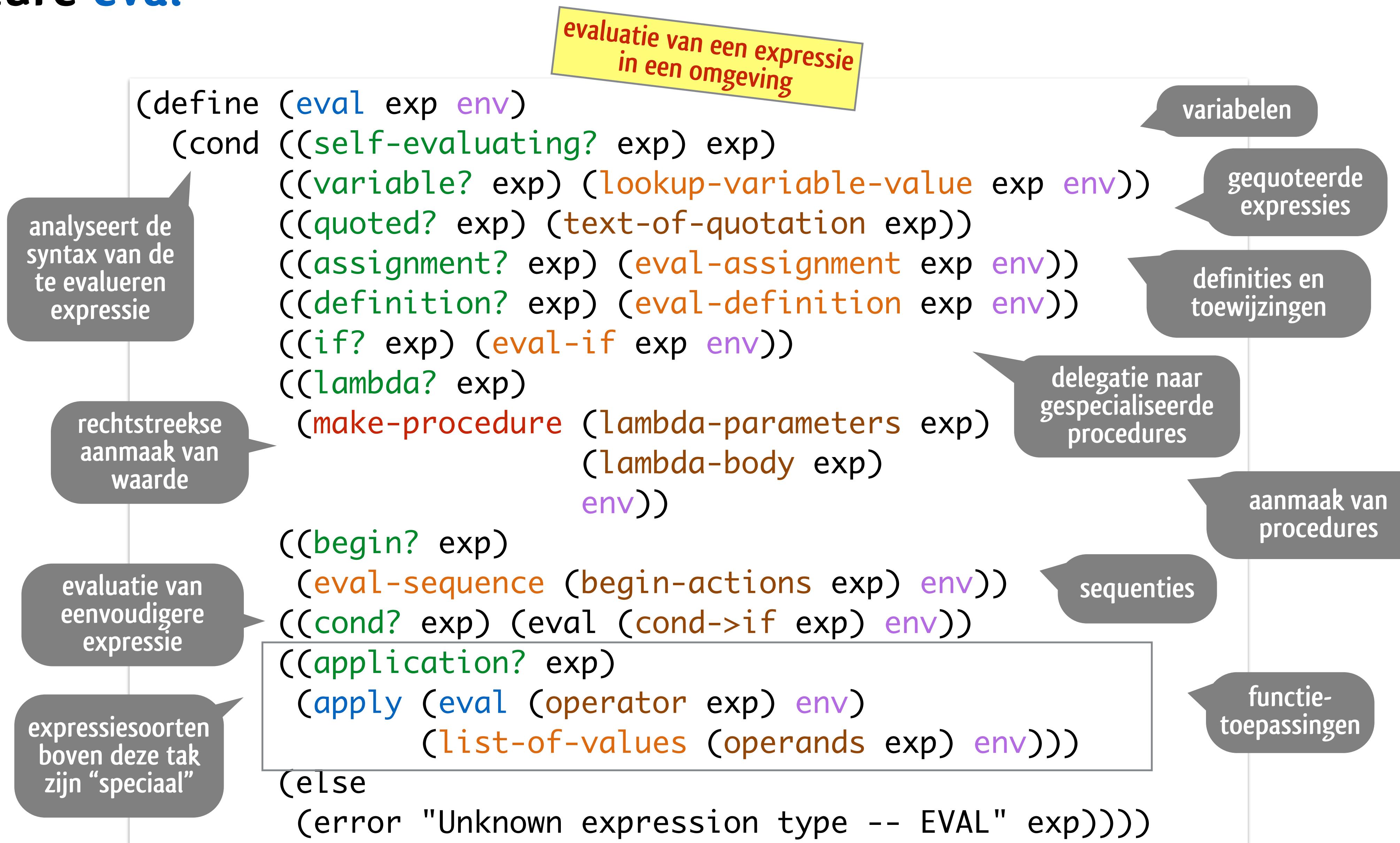


Interessant op  : respect voor het “groene monster”



[Gerald Jay Sussman, 1986, begin lecture 7A, <https://www.youtube.com/watch?v=0m6hoOelZH8&list=PLB63C06FAF154F047>]

Procedure eval



Procedure eval

predicaat dat soort van expressie of waarde nagaat

procedure om expressie of waarde in delen te ontleden

gespecialiseerde evaluatie-procedure

procedure die een waarde aanmaakt

procedure die een omgeving manipuleert

voorbehouden voor eval en apply

Operator Expr Env

Else

(error "Unknown expression type -- EVAL" exp))))

of-values (operands exp) env)))

Operator Expr Env

Voorstelling van enkelvoudige expressies

zelf-evaluuerende expressies

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

```
> (eval 123 the-global-environment)
123
> (eval "dit is een string" the-global-environment)
"dit is een string"
```

evalueren naar
zichzelf

variabelen

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

```
> (eval 'car the-global-environment)
(primitive #<procedure:mcar>)
> (eval 'square the-global-environment)
✖️✖️ Unbound variable square
>
```

worden opgezocht in
omgeving

Voorstelling van samengestelde expressies

in het algemeen

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

lijst die begint met een
bepaald symbool

specifiek per soort van expressie

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

predicaat dat test of een s-expressie
een toewijzing voorstelt

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

abstracties voor lijstbewerkingen
om de samengestelde expressie te
ontleden in deelexpressies

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

(set! <var> <exp>)

Voorstelling van samengestelde expressies

in het algemeen

```
(define (tagged-list? exp)
  (if (pair? exp)
      (eq? (car exp) 'tag)
      false))
> (define exp '(set! x (+ 2 3)))
> (tagged-list? exp 'set!)
#t
> (car exp)
set!
> (cadr exp)
```

specifiek per soort van expressie

```
(define (assignment? exp)
  (tagged-list? exp 'set!))
> (caddr exp)
(+ 2 3)
> (assignment-variable exp)
```

```
(define (assignment-variable exp)
  (cadr exp))
> (assignment-value exp)
(+ 2 3)
```

```
(define (assignment-value exp)
  (caddr exp))
>
(set! <var> <exp>)
```

Afhandeling toewijzing aan variabelen

```
(define (eval exp env)
  (cond ...
    ((assignment? exp) (eval-assignment exp env))
    ... ))
```

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

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

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

(**set! <var> <exp>**)

syntax

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

recursieve oproep van eval om
deelexpressie naar waarde te evalueren

semantiek

Afhandeling definitie van variabelen

```
(define (eval exp env)
  (cond ...
    ((definition? exp) (eval-definition exp env))
    ... ))
```

```
(define (definition? exp) ...)
```

```
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caaddr exp)))
```

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

syntax

*(define <var> <exp>)
(define (<var> <p₁>...<p_n>)
 <exp₁> ... <exp_m>)*

aanmaak van een expressie,
niet van een waarde!

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


Afhandeling if-expressie: syntax

```
(define (eval exp env)
  (cond ...
    ((if? exp) (eval-if exp env)))
    ...))

(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))

  (if <predicate> <consequent> <alternative>)
  (if <predicate> <consequent>

    > (if-consequent '(if (> a b) a b))
    a
    > (if-alternative '(if (> a b) a b))
    b
    > (if-predicate '(if (> a b) a b))
    (> a b)
    > (if-alternative '(if (= x 0) x))
    false
    >
```

Afhandeling if-expressie: semantiek

```
(define (eval exp env)
  (cond ...
    ((if? exp) (eval-if exp env))
    ... ))
```

;;; M-Eval input:
(if '() 'a 'b)

alle andere waarden worden
beschouwd als "waar"

```
(define (true? x)
  (not (eq? x #f)))
```

;;; M-Eval input:
(if (< 2 3) 'a 'b)

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

;;; M-Eval value:
a

#f is de enige waarde die als
"vals" beschouwd wordt

```
(define (eval-if exp env)
  (cond ((true? (eval (if-predicate exp) env))
         (eval (if-consequent exp) env))
        (else (eval (if-alternative exp) env))))
```

om een if-expressie te
evalueren, moeten twee
deelexpressies
geëvalueerd worden

Afhandeling quote-expressie

```
(define (eval exp env)
  (cond ...
    ((quoted? exp) (text-of-quotation exp))
    ... ))
```

vergelijk later met afhandeling
quasiquote-expressies!

semantiek

expressie wordt ontleed,
geen evaluatie nodig

;;; M-Eval input:
(+ 1 2)

;;; M-Eval value:
3

;;; M-Eval input:
(quote (+ 1 2))

;;; M-Eval value:
(+ 1 2)

;;; M-Eval input:
(quote apple)

;;; M-Eval value:
apple

;;; M-Eval input:
(quote (a b c))

;;; M-Eval value:
(a b c)

;;; M-Eval input:
'apple

;;; M-Eval value:
apple

;;; M-Eval input:
'(a b c)

;;; M-Eval value:
(a b c)

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

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

(quote <exp>)
'<exp>

syntax

Afhandeling quote-expressie

vergelijk later met afhandeling
quasiquote-expressies!

(define (eval exp env)

(cond ...

((quoted? exp) (text-of-quotation exp))

...))

;;; M-Eval input:

(+ 1 2)

;;; M-Eval value:

3

;;; M-Eval input:

(quote (+ 1 2))

;;; M-Eval value:

(+ 1 2)

;;; M-Eval input:

(quote apple)

;;; M-Eval value:

apple

> (define exp (read))

'apple

> (pair? exp)

#t

> (car exp)

quote

> (cadr exp)

apple

> (define exp2 (read))

'(apple bear kiwi)

> (car exp2)

quote

> (cadr exp2)

(apple bear kiwi)

>

Procedure **read** vormt ‘<exp> om naar ‘(quote <exp>)

syntax

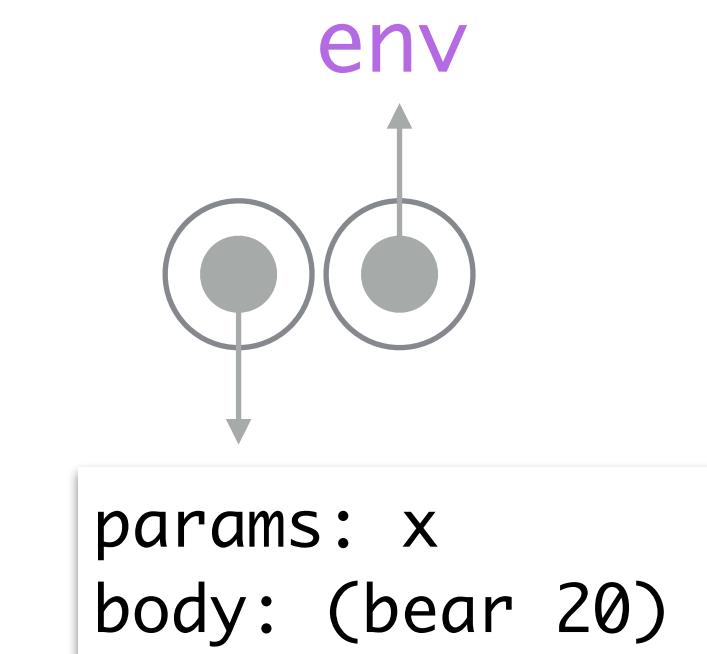
(quote <exp>
'<exp>)

Afhandeling lambda-expressie

semantiek bij
lexical scoping

```
(define (eval exp env)
  (cond ...
    ((lambda? exp)
     (make-procedure (lambda-parameters exp)
                    (lambda-body exp)
                    env)))
    ... ))
```

een lambda-expressie evalueert
naar een procedure-object dat zijn
definitie-omgeving onthoudt



syntax

```
(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))
```

Voorstelling samengestelde procedure-objecten

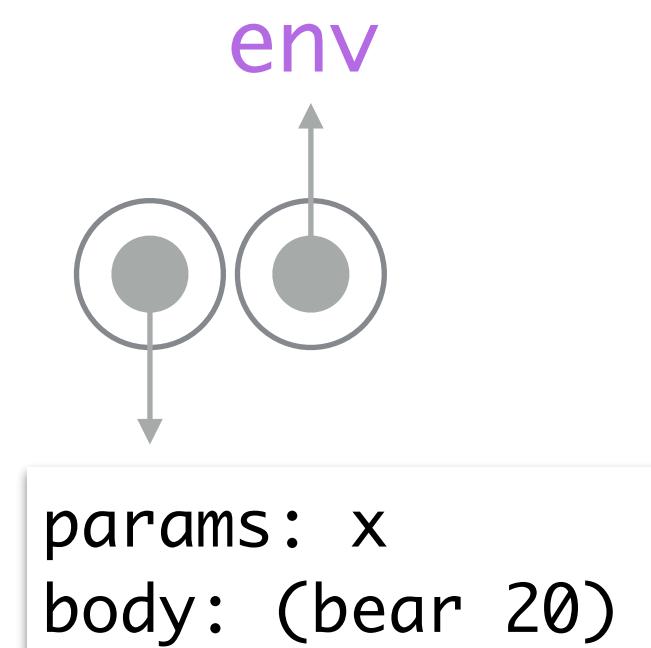
```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

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

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

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

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



Voorstelling samengestelde procedure-objecten

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

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

```
(define (procedure-parameter p)
  (cadr p))
```

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

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

zie verderop voor de echte
voorstelling van omgevingen

```
> (eval '(lambda (x) (* 2 x)) 'env)
  (procedure (x) ((* 2 x)) env)
> (make-procedure '(x) '((* 2 x)) 'env)
  (procedure (x) ((* 2 x)) env)
> (list 'procedure '(x) '((* 2 x)) 'env)
  (procedure (x) ((* 2 x)) env)
>
```

Voorstelling samen~~gestelde~~ procedure-objecten

```
;;; M-Eval input:  
(lambda (x) (+ 1 2 3))
```

```
;;; M-Eval value:  
(compound-procedure (x) ((+ 1 2 3)) <env>)
```

uitvoer van
onze user-print

```
(define (make-procedure parameters body env)  
(list 'procedure parameters body env))  
      > (eval '(lambda (x) (+ 1 2 3)) the-global-environment)  
          (procedure
```

```
(define (compound-procedure? p)  
(tagged-list? p 'procedure))  
((+ 1 2 3))  
(((false true car cdr cons null? + * / = - <)  
#f  
#t
```

uitvoer van
Racket's display op
dezelfde waarde

```
(define (procedure-parameters p)  
(cadr p))  
#f  
#t
```

```
(define (procedure-body p)  
(caddr p))  
#f  
#t
```

```
(define (procedure-environment p)  
(caddar p))  
#f  
#t
```

```
(primitive #<procedure:mcar>)  
(primitive #<procedure:mcdr>)  
(primitive #<procedure:mcons>)  
(primitive #<procedure:null?>)  
(primitive #<procedure:+>)  
(primitive #<procedure:*>)  
(primitive #<procedure:/>)  
(primitive #<procedure:=>)  
(primitive #<procedure:->)  
(primitive #<procedure:<>))))
```

Afhandeling applicatie-expressies

```
(define (eval exp env)
  (cond ...
    ((application? exp)           evaluatie van operator
      (apply (eval (operator exp) env)      naar een procedure
            (list-of-values (operands exp) env)))
    ... ))
```

evaluatie van operands naar een
lijst van concrete argumenten

semantiek

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

```
(define (operator exp)
  (car exp))
```

```
(define (operands exp)
  (cdr exp))
```

syntax

```
> (define x '(f (+ 1 2) 3))
> (application? x)
#t
> (operator x)          operator moet nog
                        geëvalueerd worden!
f
> (operands x)
((+ 1 2) 3)
```

operands moeten nog
geëvalueerd worden!

Evaluatie van operands naar concrete argumenten

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

In welke richting worden de operands geëvalueerd?

semantiek

```
;; M-Eval input:  
(define (f a b) 'ok)
```

```
(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

syntax

```
;; M-Eval value:  
ok
```

deelexpressies van applicatie hebben neveneffecten

```
;; M-Eval input:  
(f (begin (display "ba") 2)  
    (begin (display "ab") 1))
```

Hoe kunnen we deze richting veranderen?

```
baab  
;; M-Eval value:  
ok
```

resultaat van neveneffecten is afhankelijk van de volgorde waarin de deelexpressies worden geëvalueerd!

Procedure apply

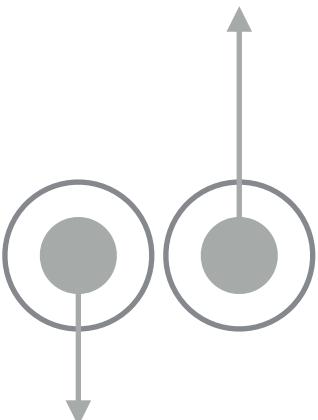
```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters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

procedure waarnaartoe
operator evalueerde

lijst van waarden waarnaartoe
operands evalueerden

toepassing van een
procedure op een lijst van
concrete argumenten

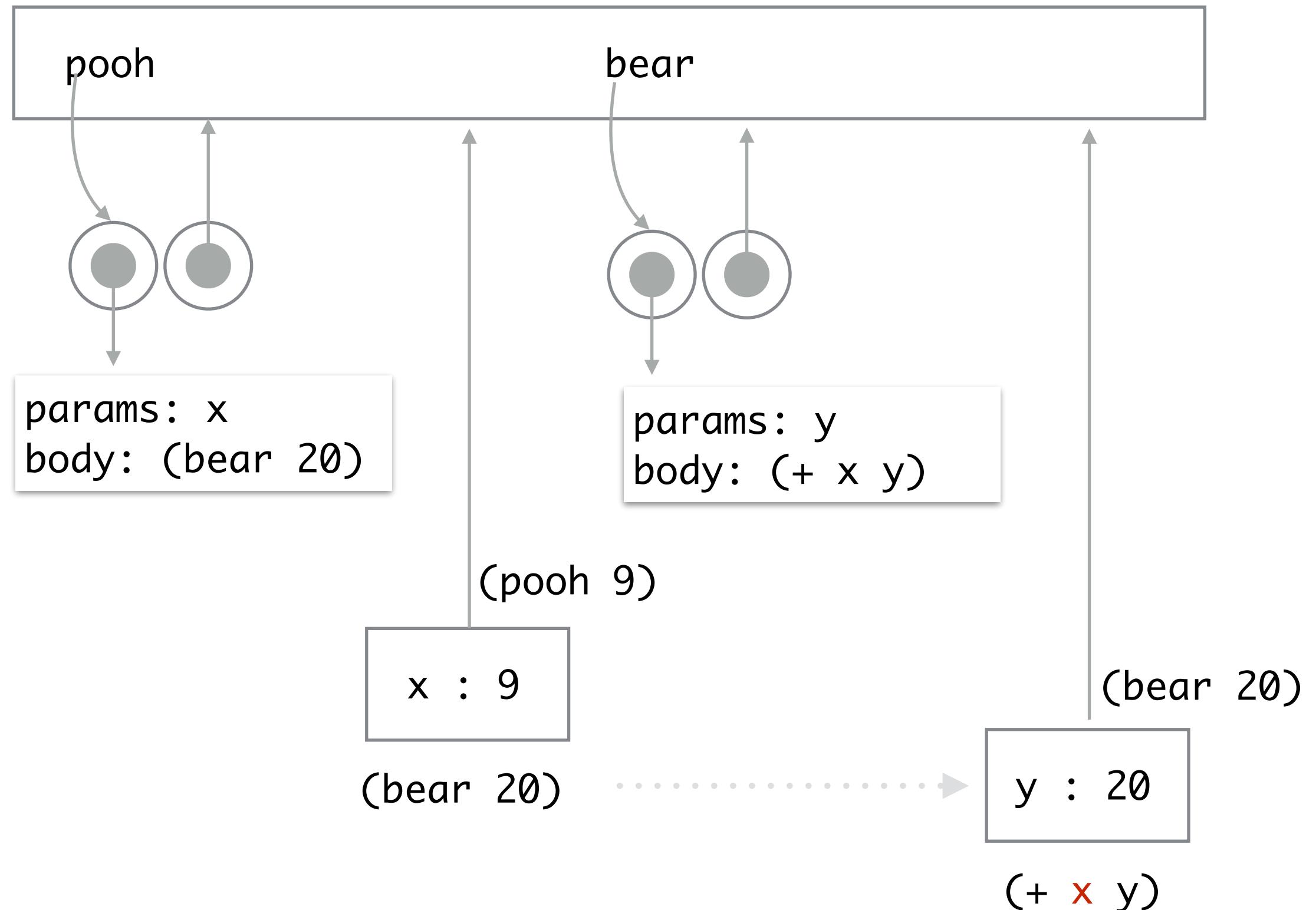
door gebruikers
gedefinieerd



uitbreiding van
definitie-omgeving
met bindingen van formele
parameters aan actuele
argumenten

Procedure **apply** implementeert lexicaal bereik

```
;;; M-Eval input:  
(define (pooh x)  
  (bear 20))  
  
;;; M-Eval value:  
ok  
  
;;; M-Eval input:  
(define (bear y)  
  (+ x y))  
  
;;; M-Eval value:  
ok  
  
;;; M-Eval input:  
(pooh 9)  
!!ERROR!! "Unbound variable" x
```



vrije variabelen worden opgezocht in een uitbreiding van de lexcale omgeving waarin procedure bear aangemaakt werd

Afhandeling expressie-reeksen: syntax

```
(define (eval exp env)
  (cond ...
    ((begin? exp)
     (eval-sequence (begin-actions exp) env))
    ... ))
```

```
(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))
```

syntax

```
> (define exp '(begin (display 1) (display 2) (display 3)))
> (begin-actions exp)
((display 1) (display 2) (display 3))
> (eval exp the-global-environment)
123
```

```
> (eval-sequence (begin-actions exp) the-global-environment)
123
> |
```

Afhandeling expressie-reeksen: semantiek

```
(define (eval exp env)
  (cond ...
    ((begin? exp)
     (eval-sequence (begin-actions exp) env))
    ... ))
```

Wordt ook opgeroepen
vanuit apply om de body van
een procedure te evalueren.

semantiek

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

Verschil met
list-of-values?

geeft als waarde de waarde van
de laatste deelexpressie terug

evalueer elke deelexpressie

Afhandeling cond-expressie

```
(define (eval exp env)
  (cond ...
    ((cond? exp) (eval (cond->if exp) env))
    ...))
```

wordt vóór de evaluatie
omgevormd naar een
eenvoudigere expressie

```
(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))
```

syntax

```
(cond <clause1>
  ...
  <clausen>)
waarbij <clausei>
  = (<predicate> <action1> ... <actionn>)
  | (else <action1> ... <actionn>)
```

Afhandeling cond-expressie

wordt voor de evaluatie
omgevormd naar een
eenvoudigere expressie

```
(define (eval exp env)
  (cond ...
    ((cond? exp) (eval (cond->if exp) env)))
    ...))

> (define exp '(cond ((> x y) 'a)
                      ((= x y) 'b)
                      (else 'c 'd)))
(define (cond? exp)
  (tagged-list? exp 'cond))
> (cond? exp)
#t
(define (cond-clauses exp)
  (cdr exp))
> (cond-clauses exp)
(((> x y) 'a) ((= x y) 'b) (else 'c 'd))
> (map cond-predicate (cond-clauses exp))
((> x y) (= x y) else)
> (cond-actions (car (cond-clauses exp)))
('a)
> (cond-actions (caddr (cond-clauses exp)))
('c 'd)
>

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

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

Afhandeling cond-expressie

```
(define (eval exp env)
  (cond ...
    ((cond? exp) (eval (cond->if exp) env))
    ... ))
```

```
(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)))))))
```

genereert een if-expressie

vormt een lijst van expressies om naar
een geldige begin-expressie

Afhandeling cond-expressie

```
(define (eval exp env)
  (cond ...
    ((cond? exp) (ev
      ...)))
    > (define exp '(cond ((> x y) 'a)
                           ((= x y) 'b)
                           (else 'c 'd)))
    > (cond->if exp)
      (if (> x y) 'a (if (= x y) 'b (begin 'c 'd)))
    > (expand-clauses '())
      false
    > (expand-clauses (list '(else 'c 'd)))
      (begin 'c 'd)
    > (expand-clauses (list '((= x y) 'b)
                           '(else 'c 'd)))
      (if (= x y) 'b (begin 'c 'd))
    > (expand-clauses (list '(> x y) 'a)
                           '((= x y) 'b)
                           '(else 'c 'd)))
      (if (> x y) 'a (if (= x y) 'b (begin 'c 'd)))
    >
```

vormt een lijst van expressies om naar
een geldige begin-expressie

first))
t -- COND->IF"

make-if (cond-precurse first)

(sequence->exp (cond-actions first))
(expand-clauses rest))))))

Hulpprocedures bij de omvorming van expressies

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

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

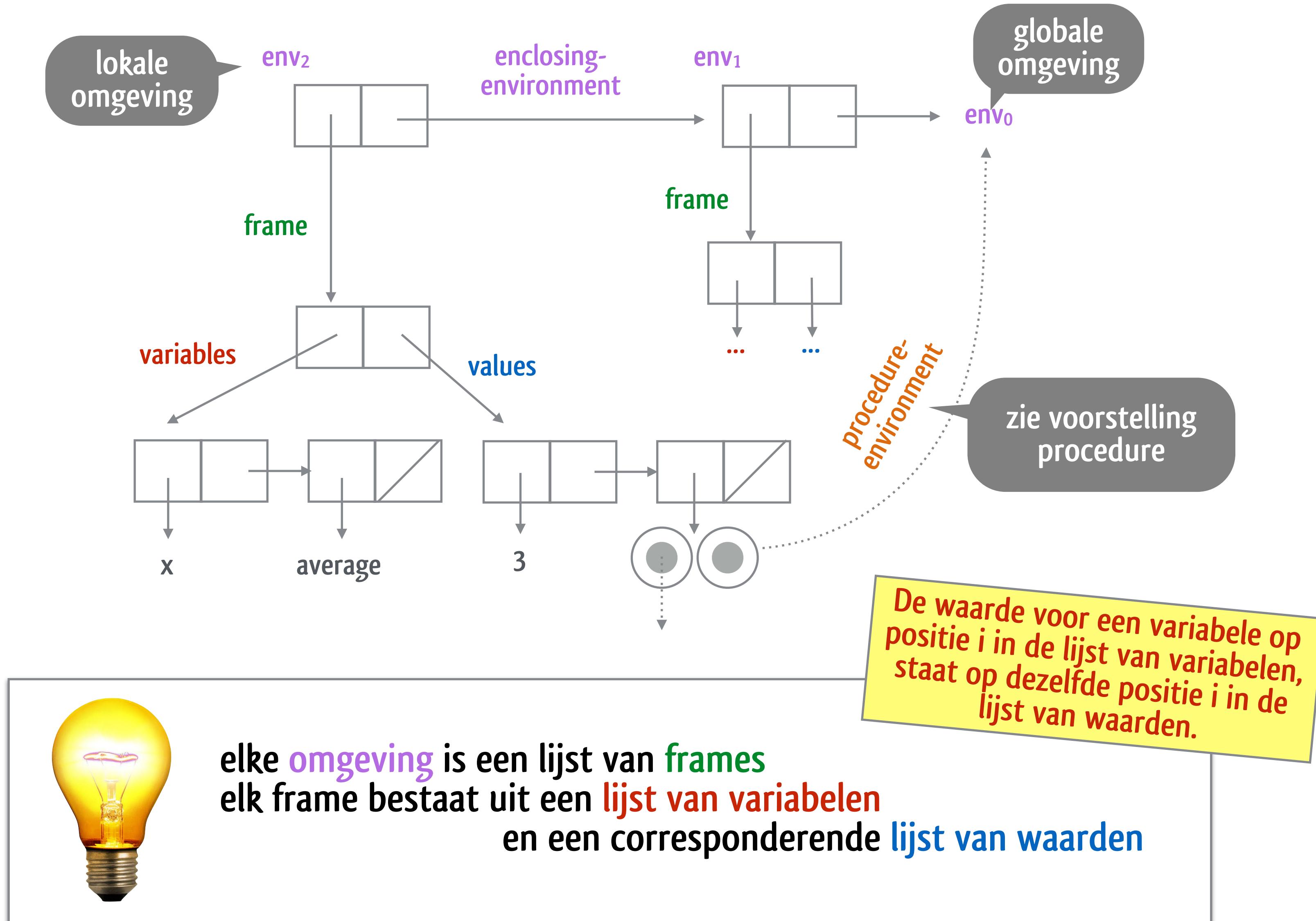
```
(define (make-begin seq)
  (cons 'begin seq))
```

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

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

Deze procedures evalueren NIETS!
Zij opereren op bomen met de
vorm van een s-expressie, en geven
zulke bomen terug.

Voorstelling omgevingen



Voorstelling omgevingen

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

frame = lijst van variabelen met corresponderende lijst van waarden

```
(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))))
```

nieuwe binding aan een frame toevoegen

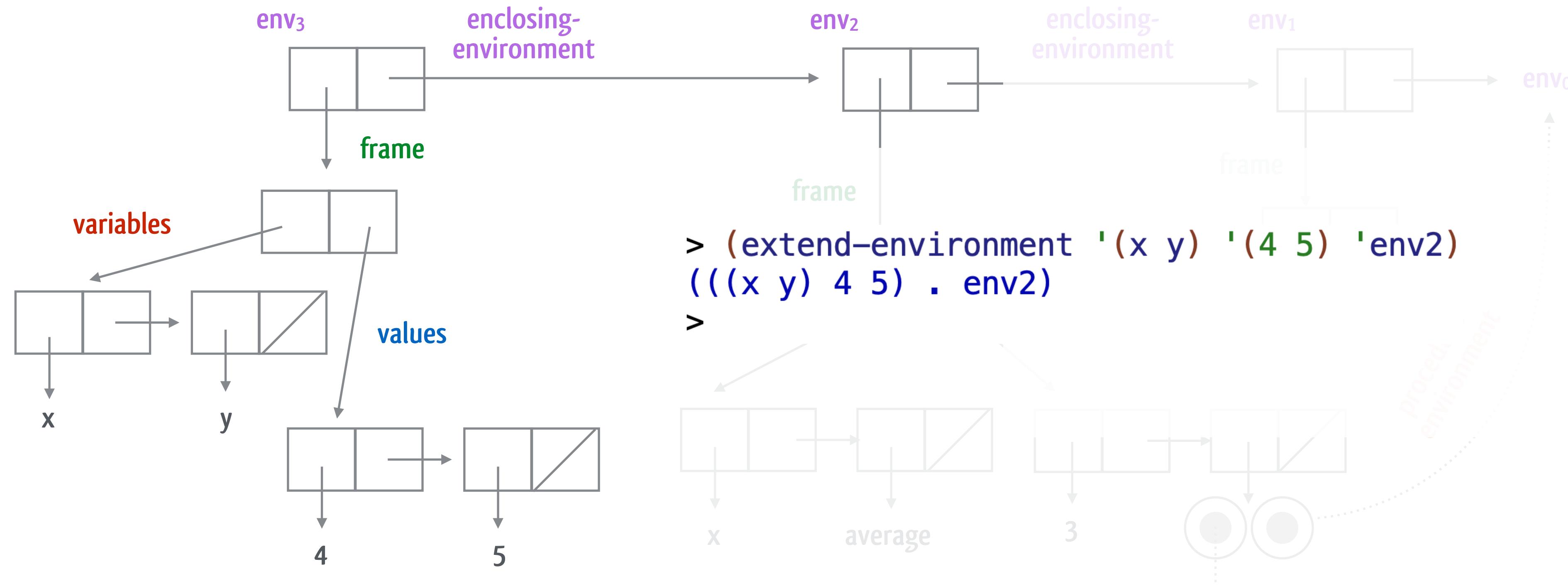
```
(define the-empty-environment '())
```

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

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

omgeving = lijst van frames

Operaties op omgevingen: uitbreiden



```
(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))))
```

Operaties op omgevingen: opzoeken

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) opzoeken in volgend frame
              (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) variabele gevonden
              (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) niet gevonden in laatste frame
                  (frame-values frame))))))
    (env-loop env)))
```

Operaties op omgevingen: opzoeken

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      ...))
```



- o start de zoektocht in het **eerste frame** van de omgeving
- o doorzoek de **lijst van variabelen** en de **lijst van waarden** in frame in aan hetzelfde tempo
- o indien de **variabele** op de huidige positie in de **lijst van variabelen** staat, staat de **waarde** van de variabele op dezelfde positie in de **lijst van waarden**
- o indien niet gevonden in het **huidige frame**, zet de zoektocht verder in de **omliggende omgeving**

Overgaan naar cdr van variabelen-lijst, betekent ook overgaan naar cdr van waarden-lijst.

start opzoeking in

(frame-values frame))))

Operaties op omgevingen: toewijzingen

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars) opzoeken in volgend frame
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars)) variabele gevonden
             (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) niet gevonden in laatste frame
                  (frame-values frame)))))

start opzoeking in eerste frame
  (env-loop env)))
```

Operaties op omgevingen: definities

```
(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))))
```

Waarom moet de zoektocht
beperkt blijven tot de
lokale omgeving?

geen bestaande binding gevonden

aanpassen bestaande binding

start zoektocht
door eerste frame

Initialisatie van de globale omgeving

```
(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))
(driver-loop)
```

name van primitieve procedures

primitieve procedures

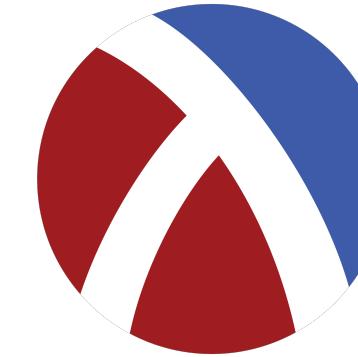
lege omgeving, uitgebreid met definities voor primitieve procedures zoals car, +, ...

Voorstelling primitieve procedures

```
(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 '< <)
        (list 'display display))))
```



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

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

Afhandeling oproepen van primitieve procedures

```
(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))))
```

synoniem voor originele
apply van Racket

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

is

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          > (apply-in-underlying-scheme + '(1 2 3 4))
          10
          > (apply-in-underlying-scheme + '(cons a b))
          ❌ ✗ +: contract violation
          expected: number?
          given: cons
          argument position: 1st
          "other arguments...":
          >
```



De Racket apply verwacht een Racket procedure als eerste argument,
en een lijst van Racket waarden om deze op toe te passen

De meta-circulaire interpreter in actie

The screenshot shows the DrRacket interface with the file "ICP1_1a0_meval.scm" open. The window title is "ICP1_1a0_meval.scm - DrRacket". The menu bar includes "File", "Edit", "Syntax", "Language", "Definitions", "Tools", "Help", and "About". The toolbar contains icons for "Check Syntax" (blue checkmark), "Debug" (magnifying glass), "Macro Stepper" (green arrow), "Run" (green arrow), and "Stop" (red square). The main window displays the following Racket code and its execution:

```
;;; M-Eval input:  
(define (fac num)  
  (cond ((= num 0) 1)  
        (else (* num (fac (- num 1))))))  
  
;;; M-Eval value:  
ok  
  
;;; M-Eval input:  
(fac 5)  
  
;;; M-Eval value:  
120  
  
;;; M-Eval input:  
[empty input field]  
eof
```

The status bar at the bottom shows "R5RS custom" selected, "20:2" lines of code, "358.26 MB" memory usage, and a small green icon.

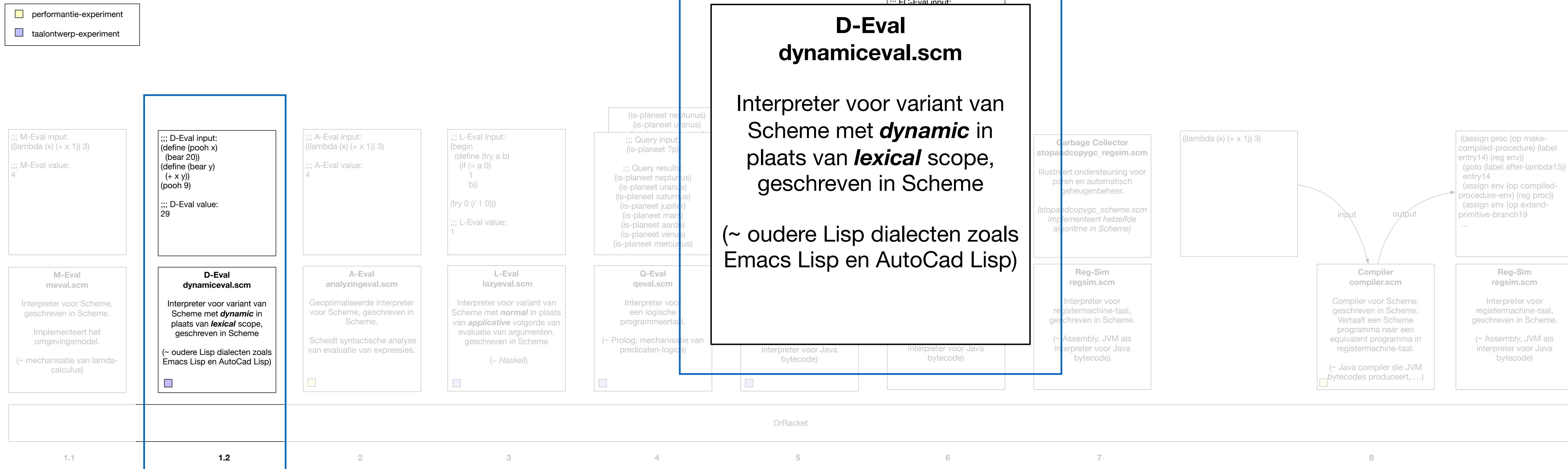
Interessant op : kern van alle talen



[Gerald Jay Sussman, 1986, einde lecture 7A, <https://www.youtube.com/watch?v=0m6hoOelZH8&list=PLB63C06FAF154F047>]

1.2 Variant voor Scheme met **dynamisch bereik**

Context in cursus

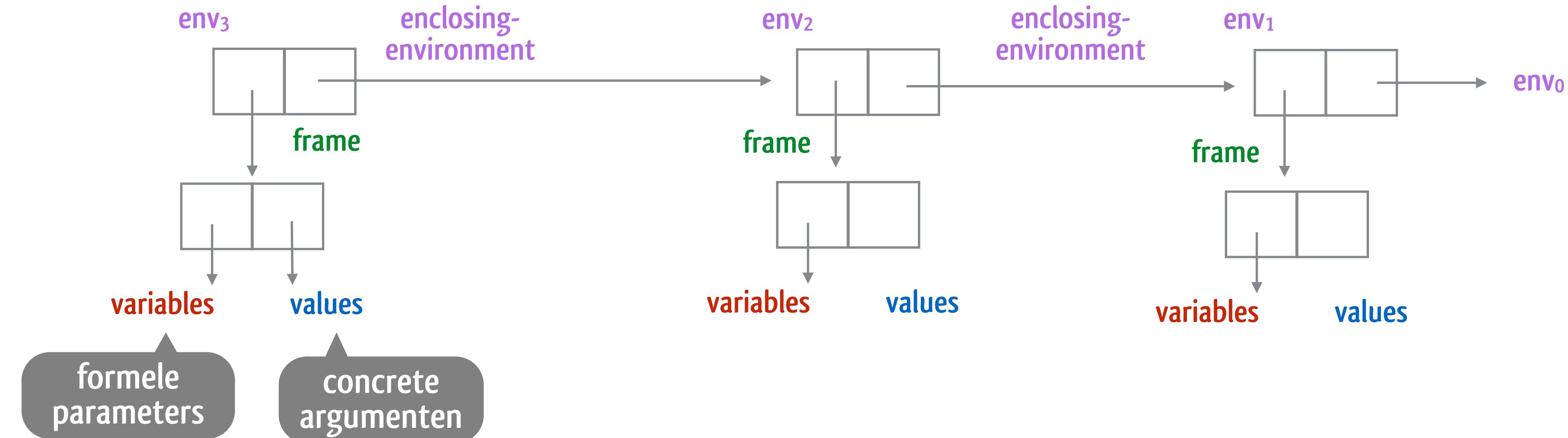


Procedure apply implementeert lexicaal bereik

- de **body** van de opgeroepen procedure wordt geëvalueerd in een **uitbreiding** van de **definitie-omgeving** van de procedure
- de uitbreiding bestaat uit een **nieuw frame**
- dat de **concrete argumenten** van de oproep bindt aan de **formele parameters** van de procedure
- bij de evaluatie van de **body** zullen vrije variabelen niet gevonden worden in het **nieuwe frame**
- maar ergens verderop in de lijst van bestaande **frames** van de **definitie-omgeving**

```
(define (apply procedure arguments)
  (cond ...
    ((compound-procedure? procedure)
     (eval-sequence
      (procedure-body procedure)
      (extend-environment
       (procedure-parameters procedure)
       arguments
       (procedure-environment procedure))))))
    ... ))
```

onthouden bij de evaluatie van een lambda-expressie tot een procedure



Lexicaal bereik in de praktijk

vrije variabelen worden opgezocht in een definitie-omgeving van de opgeroepen procedure

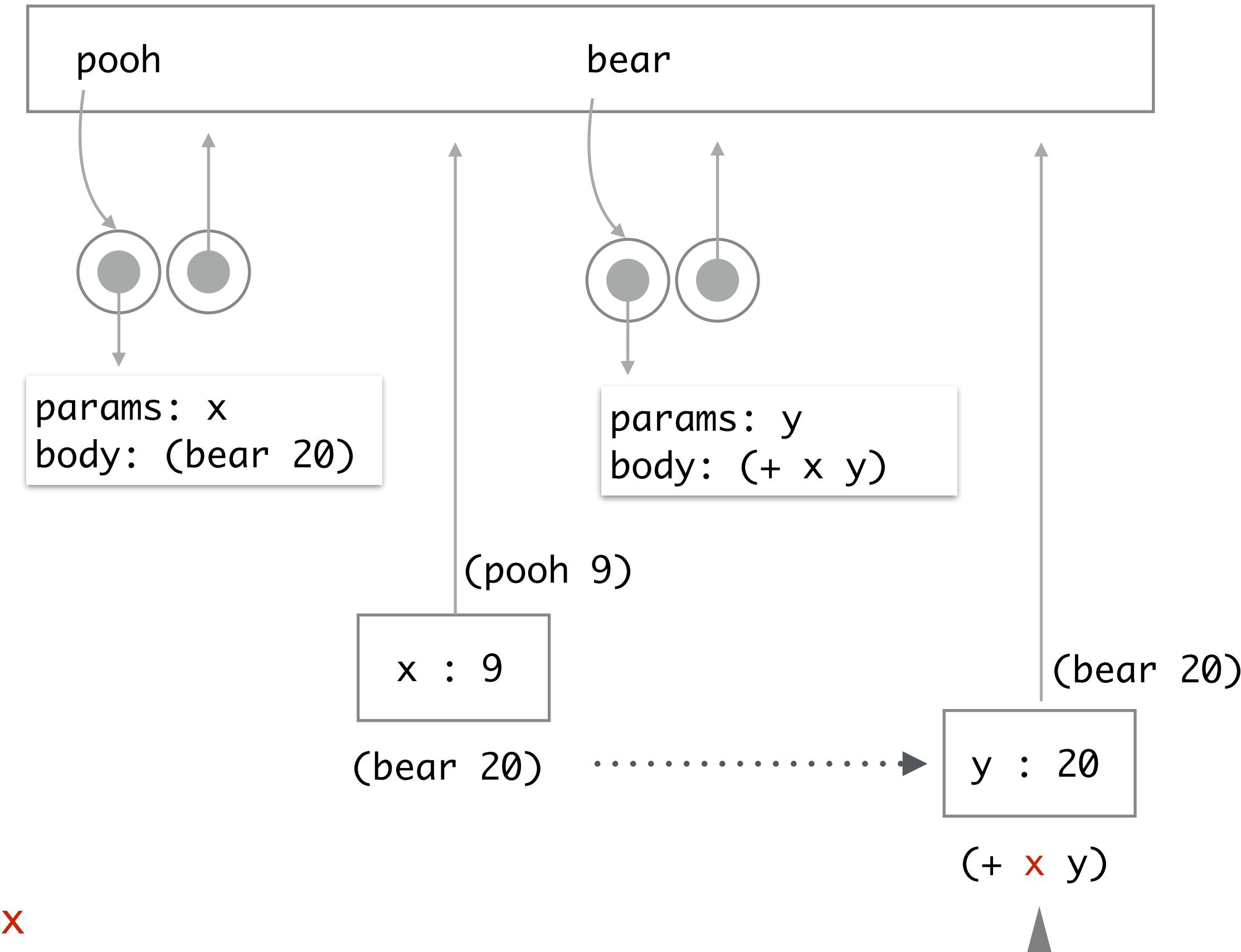
;;; M-Eval input:
(define (pooh x)
 (bear 20))

;;; M-Eval value:
ok

;;; M-Eval input:
(define (bear y)
 (+ x y))

;;; M-Eval value:
ok

;;; M-Eval input:
(pooh 9)
!!ERROR!! "Unbound variable" x



het bereik van parameter x is slechts beperkt tot
de implementatie van procedure pooh

Dynamisch bereik als alternatief in de praktijk

vrije variabelen worden opgezocht in de omgeving van de procedure-oproep

;;; D-Eval input:
(define (pooh x)
 (bear 20))

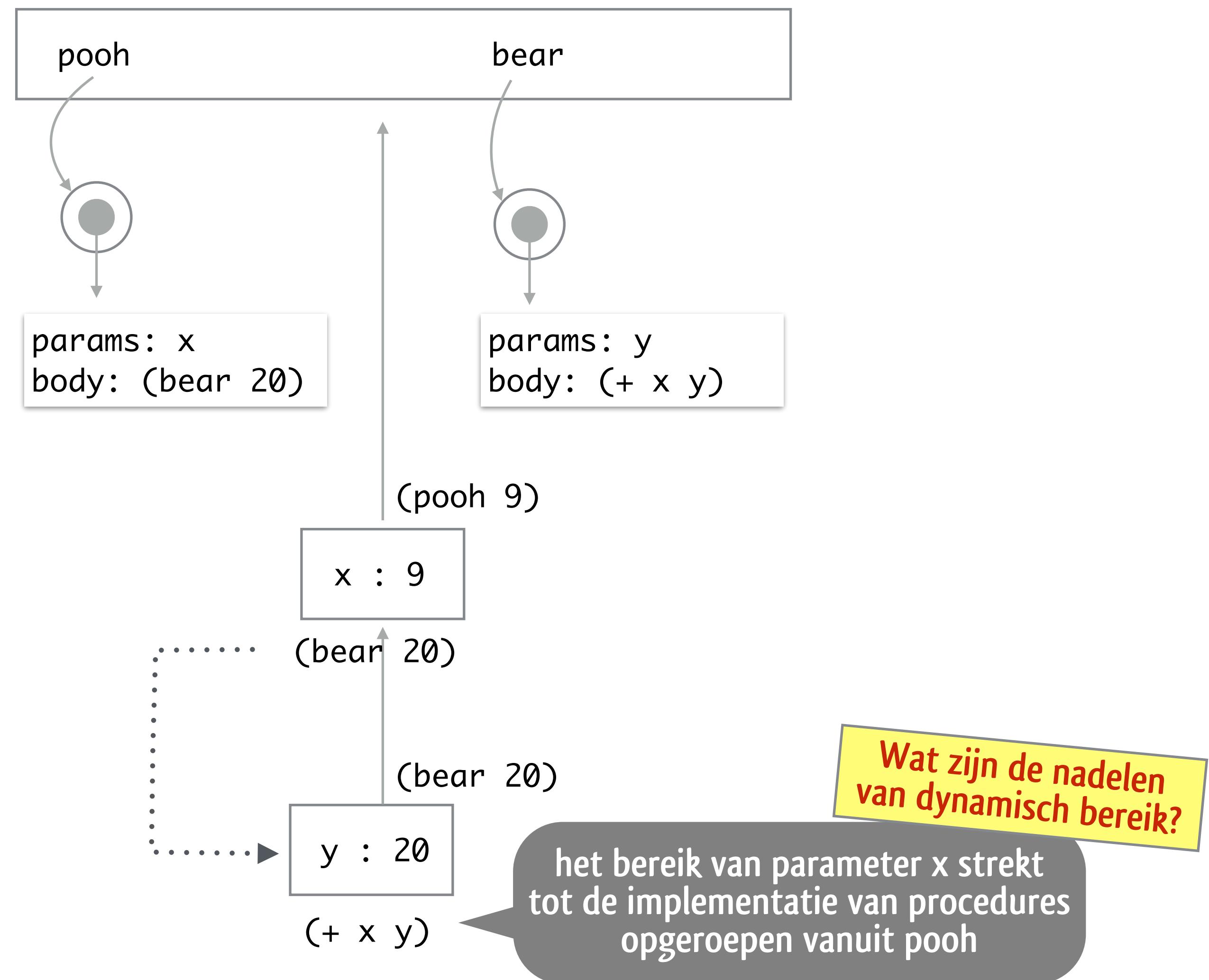
;;; D-Eval value:
ok

;;; D-Eval input:
(define (bear y)
 (+ x y))

;;; D-Eval value:
ok

;;; D-Eval input:
(pooh 9)

;;; D-Eval value:
29

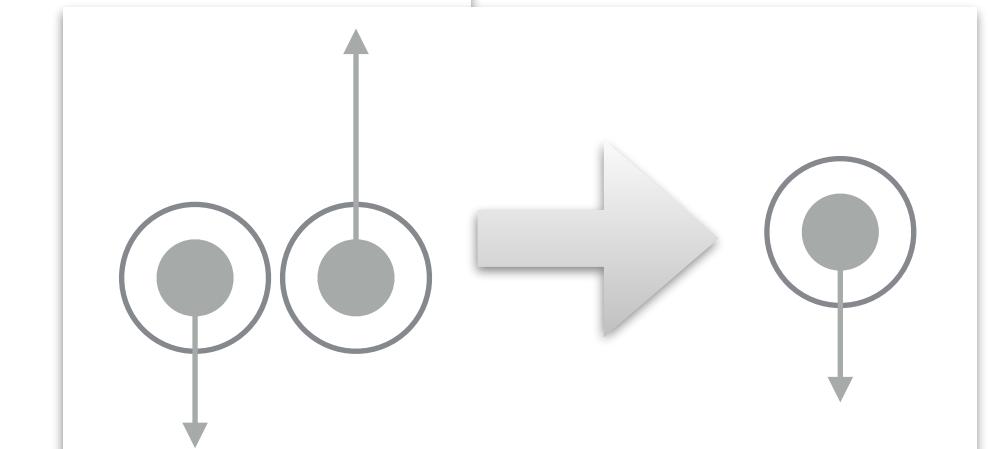


Procedure eval voor dynamisch bereik

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ...
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         '*no-environment*))
        ...
        ((application? exp)
         (dynamic-apply
          (eval operator exp) env)
          (list-of-values (operands exp) env)
          env)))
  (else
   (error "Unknown expression type -- EVAL" exp))))
```

apply krijgt de omgeving van de oproep mee, om de implementatie van de opgeroepen procedure te evalueren

procedures hoeven de definitieomgeving niet langer te onthouden



Procedure dynamic-apply voor dynamisch bereik

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

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

Variant met dynamisch bereik in actie



```
ICP1_1_dynamiceval.scm - DrRacket
ICP1_1_dynamiceval.scm (define ...) ▾ Macro Stepper #▶ Run ▶ Stop □

;;; D-Eval input:
(define n 3)

;;; D-Eval value:
ok

;;; D-Eval input:
(define (make-adder n)
  (lambda (x) (+ n x)))

;;; D-Eval value:
ok

;;; D-Eval input:
(define inc (make-adder 1))

;;; D-Eval value:
ok

;;; D-Eval input:
(inc 7)

;;; D-Eval value:
10

;;; D-Eval input:

```

procedures onthouden hun definitie-omgeving
niet langer waardoor ontwikkelaars geen
closures meer kunnen aanmaken:
de lijst van frames in een omgeving stemt
overeen met de zogenaamde call stack

Interessant op : dynamisch bereik in eerste Lisps

```
(define apply
  (lambda (proc args env) ;!
    (cond
      ((primitive? proc) ;magic
       (apply-primop proc args))
      ((eq? (car proc) 'lambda)
       ; proc = (LAMBDA bvs body)
       (eval (caddr proc) ;body
             (bind (cadr proc) ;bvs
                   args
                   env)))) ;env !)
      (else error-known-procedure)))))
```

Herhalingsvragen

- op welke twee manieren kan de interpreter uitgebreid worden met ondersteuning voor nieuwe soorten van expressies?
- welke takken van procedure eval worden achtereenvolgens doorlopen om een expressie zoals `(define (f x) x)` volledig te evalueren?
- welke code zou je moeten aanpassen om procedures als `(CALL f 3)` in plaats van `(f 3)` op te roepen?
- waarom roept procedure eval de procedure apply op?
waarom roept procedure apply opnieuw procedure eval op?
- hoe worden samengestelde expressies doorgaans geëvalueerd?
- waarom is het geen goed idee om hogere-orde procedures zoals `map` als primitieven te implementeren?
- hoe en waarom verschilt de afhandeling van de toewijzing van een variabele van de definitie van een variabele?
- duid de verschillen aan tussen de code voor de interpreter met statisch bereik en code voor de interpreter met dynamisch bereik
- waarom gebruiken moderne Lispen statisch bereik? illustreer enkele problemen van dynamisch bereik aan de hand van een voorbeeld