Les 2: Procedures&blokstructuren

SESSIE 1

Les 2: Procedures&blokstructuren

De waaier van procedures die we op dit moment kunnen schrijven is zeer beperkt. Daarom worden in deze les eerst conditionele uitdrukkingen en predikaten geïntroduceerd. Daarna schrijven we een eerste echt programma dat de vierkantswortel berekent van een gegeven getal met de iteratiemethode van Newton (dezelfde als de F=m.a uit de fysica).

Dit voorbeeldprogramma wordt dan verder gebruikt om de notie van lokale procedures en blokstructuren te introduceren. De regels die de zichtbaarheid van lokale procedures en formele parameters vastleggen worden ook uitgelegd.

Elementen van een programmeertaal

	Data	Procedure
Primitive	5 5.0 -3 #t 3.14 #f	+ - < and or * / >= not
Combination		(/ (* 3 5) (+ 1 2)) (if (> 0 x) (- x) x))
Abstraction	(define n 5)	(define (gemiddelde x y) (/ (+ x y) 2)

Terminologie

voorwaardelijke uitdrukkingen	conditional expressions
predikaten	predicates
lokale definitie	local definition
blokstructuur	block structure
zichtbaarheidregels	scope rules
tekstuele zichtbaarheid	lexical scope
vrije variable	free variable
gebonden variable	bound variable

Booleans en predicaten


```
> #t
#t
> #f
#f
> (> 4 3)
#t
> (< 4 3)
#f
> (>= 4 4)
#t
> (= (+ 2 3) 5)
#t
> (even? 5)
#f
> (odd? 5)
#t
```

Operatoren over Booleans

```
> (< 5 0)
#f
> (not (< 5 0))
#t
> (and (> 5 0) (< 4 5))
#t
> (or (< 5 0) (< 4 5))
#t
> (and (< 1 2) (< 2 3) (< 3 4))
#t
> (or (> 1 2) (> 2 3) (> 3 4))
#f
```

(not <expression>)

(and <expression-1> . . . <expression-n>)

(or <expression-1> . . . <expression-n>)

Conditionele uitdrukkingen (IF)

```
(define (abs-value x)
   (if (< x 0)
       (- x)
       x))
```

```
> (abs-value 3)
3
> (abs-value -4)
4
> (abs-value 0)
0
```

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

```
(define (double-non-zero x)
   (if (not (= x 0))
       (* 2 x)))
```

```
> (double-non-zero 4)
8
> (double-non-zero -4)
-8
> (double-non-zero 0)
>
```

geeft NIETS terug; de klant code moet hierop voorzien zijn

(if <predicate>
<consequent>)

Conditionele uitdrukkingen (COND)

```
(define (abs-value x)
  (cond
    ((> x 0) x)
    ((= x 0) 0)
    ((< x 0) (- x))))
```

```
> (abs-value 3)
3
> (abs-value -4)
4
> (abs-value 0)
0
```

```
(define (abs-value x)
  (cond
    ((< x 0) (- x))
    (else x)))
```

```
> (abs-value 3)
3
> (abs-value -4)
4
> (abs-value 0)
0
```

```
(cond
  (<predicate-1> <expression-1>)
  (<predicate-2> <expression-2>)
  . . .
  (<predicate-n> <expression-n>))
```

```
(cond
  (<predicate-1> <expression-1>)
  (<predicate-2> <expression-2>)
  . . .
  (else <expression-n>))
```

Conditionele uitdrukkingen (foute abs procedures)

```
(define (abs-wrong x)
   (if (< x 0)
      (- x)))
```

```
> (abs-wrong 3)
> (abs-wrong -4)
4
> (abs-wrong 0)
>
```

geeft NIETS terug

geeft NIETS terug

```
(define (abs-wrong x)
  (cond
     ((< x 0) (- x))
     ((> x 0) x)))
```

```
> (abs-wrong 3)
3
> (abs-wrong -4)
4
> (abs-wrong 0)
>
```

geeft NIETS terug

Wortels berekenen met de methode van Newton

Definition: $\sqrt{x} = y \iff y >= 0 \text{ and } y^2 = x$

zegt niet direct HOE je de wortel kan

Procedure: IF y is guess for $\sqrt{x}$

THEN $\frac{y + \frac{x}{y}}{2}$ is a better guess

Newton heeft dit bewezen

dit is een voorbeeld van een benaderings of een approximatie methode

Wortels berekenen met Newton's iteratie methode

```
(define (wortel x)
  (probeer 1 x))

(define (probeer gok x)
  (if (goed-genoeg? gok x)
      gok
      (probeer (verbeter gok x) x)))

(define (goed-genoeg? gok x)
  (< (abs (- (kwadraat gok) x)) 0.01))

(define (verbeter gok x)
  (gemiddelde gok (/ x gok)))

(define (gemiddelde x y)
  (/ (+ x y) 2.0))

(define (kwadraat x)
  (* x x))
```

opstarten met gok 1

als de gok goed genoeg is geef hem terug als resultaat

anders probeer opnieuw maar met een betere gok

goed genoeg volgens een zelf gekozen precisie

de beroemde breuk levert een betere gok

hulpjes

```
> (wortel 4)
2.000609756097561
```

Wortels berekenen met Newton's iteratie methode - Trace

```
> (#%require racket/trace)
> (trace wortel)
> (wortel 4)
>(wortel 4)
<2.000609756097561
2.000609756097561
> (trace probeer)
> (wortel 4)
>(wortel 4)
>(probeer 1 4)
>(probeer 2.5 4)
>(probeer 2.05 4)
>(probeer 2.000609756097561 4)
<2.000609756097561
2.000609756097561
```

trace 'laden'

de procedure wortel tracen

trace print een lijn uit voor elke oproep

als je ook de probeer procedure traced kan je de verschillende stappen volgen

(uit wiskunde)
het aantal oproepen dat nodig is hangt af van de gewenste precisie en van de startgok

Les 2: Procedures&blokstructuren

SESSIE 2

'Iter' in naam is gebruikelijk omdat de procedure 'itereert'

```
(define (sqrt x)
  (sqrt-iter 1.0 x))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))

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

(define (average x y)
  (/ (+ x y) 2.0))
```

gewoon een "Engelse" versie van wortel

Gebonden <-> vrije variabelen, scope

```
(define (average x y)
  (/ (+ x y) 2.0))
```

x en y zijn gebonden variabelen en hebben als scope de body van deze procedure; / en + zijn vrije variabelen

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

guess en x zijn gebonden variabelen en hebben als scope de body van deze procedure; abs, square, < en - zijn vrije variabelen

CODE

Locale procedures en blokstructuur

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

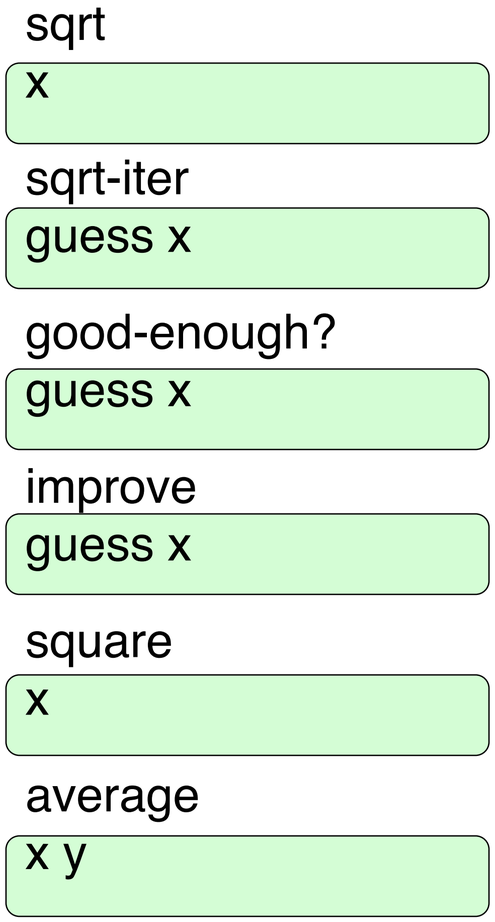
deze 3 definities staan beter lokaal van hebben weinig nut buiten de context van wortelberekenen

merk op dat x geen formele parameter is in de lokale definities maar toch gebruikt wordt in de body's

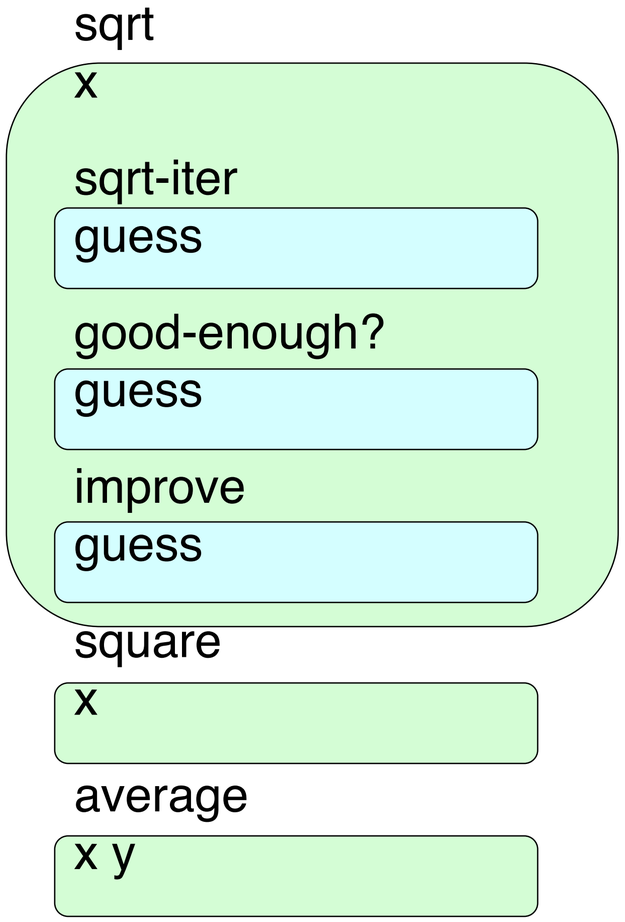
```
> (sqrt 4)
2.000201896335424
```

Lexicale Scoping: Het sqrt voorbeeld

flat version



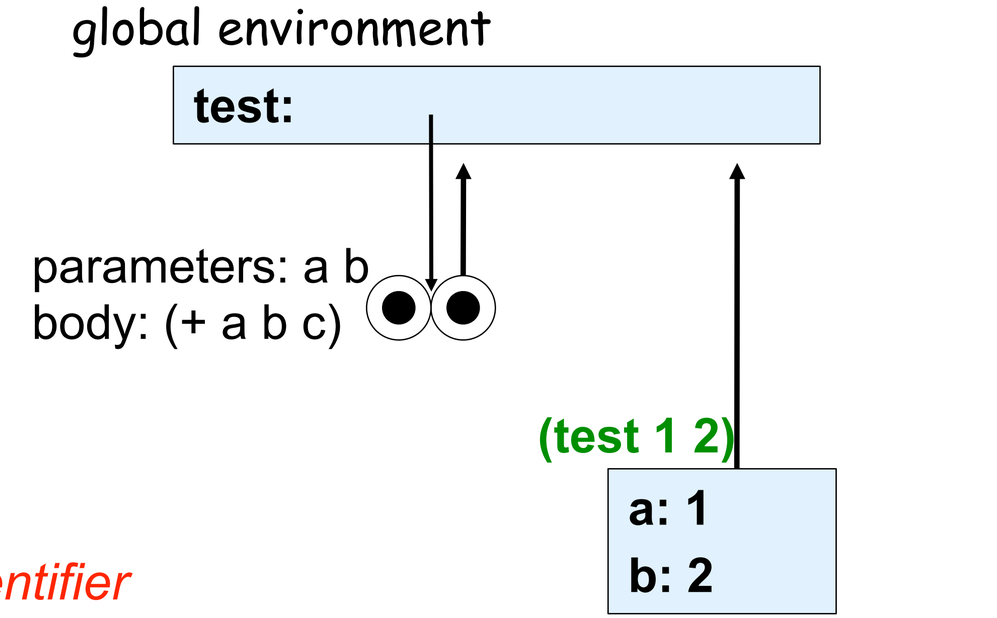
nested version



Lexicale scoping (1)

Wat met een vrije variabele in de body van een procedure?

```
(define (test a b)
   (+ a b c))
```




```
> (test 1 2)
⊗⊗ c: undefined;
cannot reference undefined identifier
```

de variable c is niet gekend wanneer de body van test wordt geëvalueerd

Lexicale scoping (2)

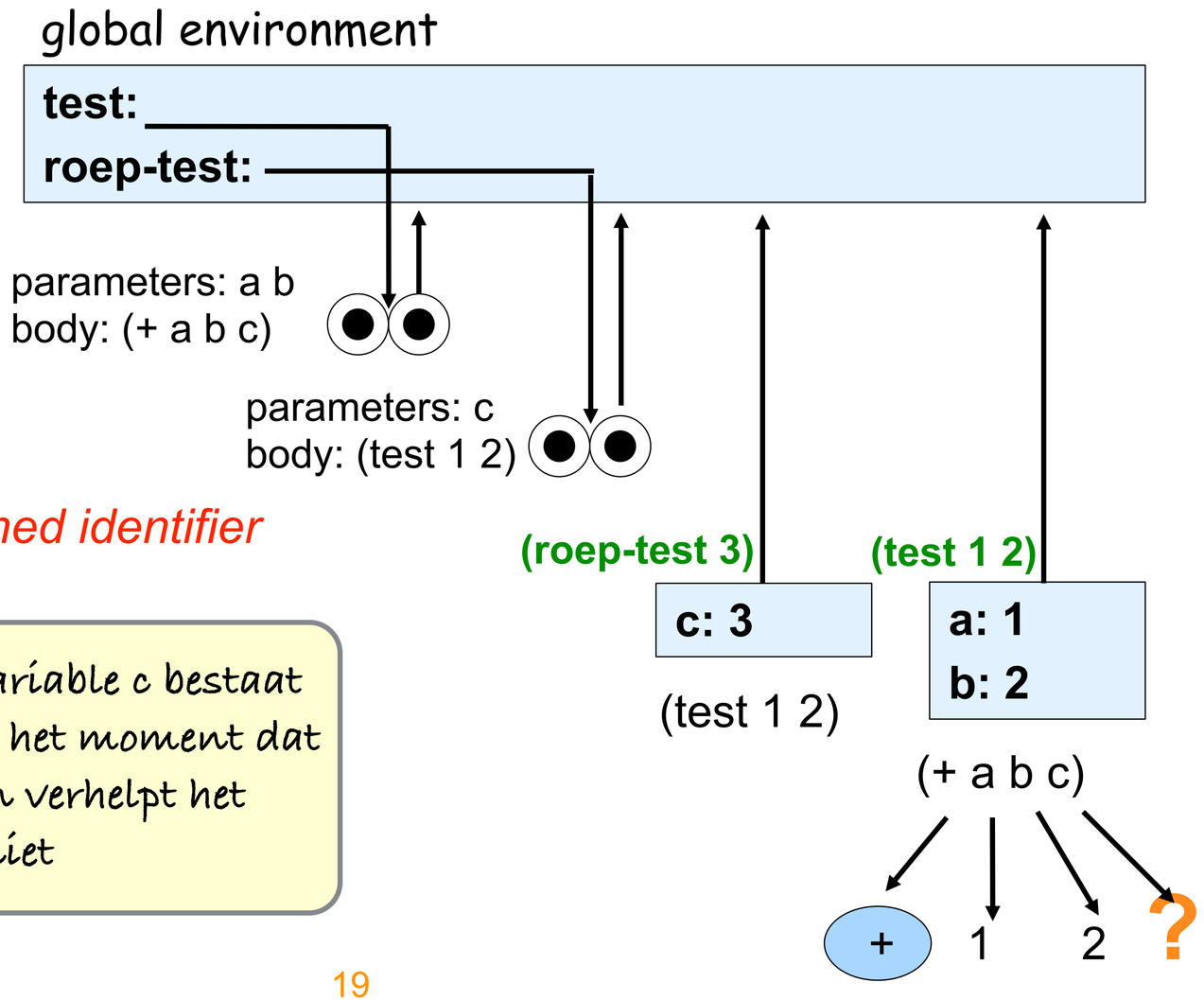
Kan de plaats van oproep helpen?

```
(define (test a b)
   (+ a b c))

(define (roep-test c)
   (test 1 2))
```

```
> (roep-test 3)
⊗⊗ c: undefined;
cannot reference undefined identifier
```

er voor zorgen dat de variable c bestaat (en een waarde heeft) op het moment dat test wordt opgeroepen verhelpt het probleem niet

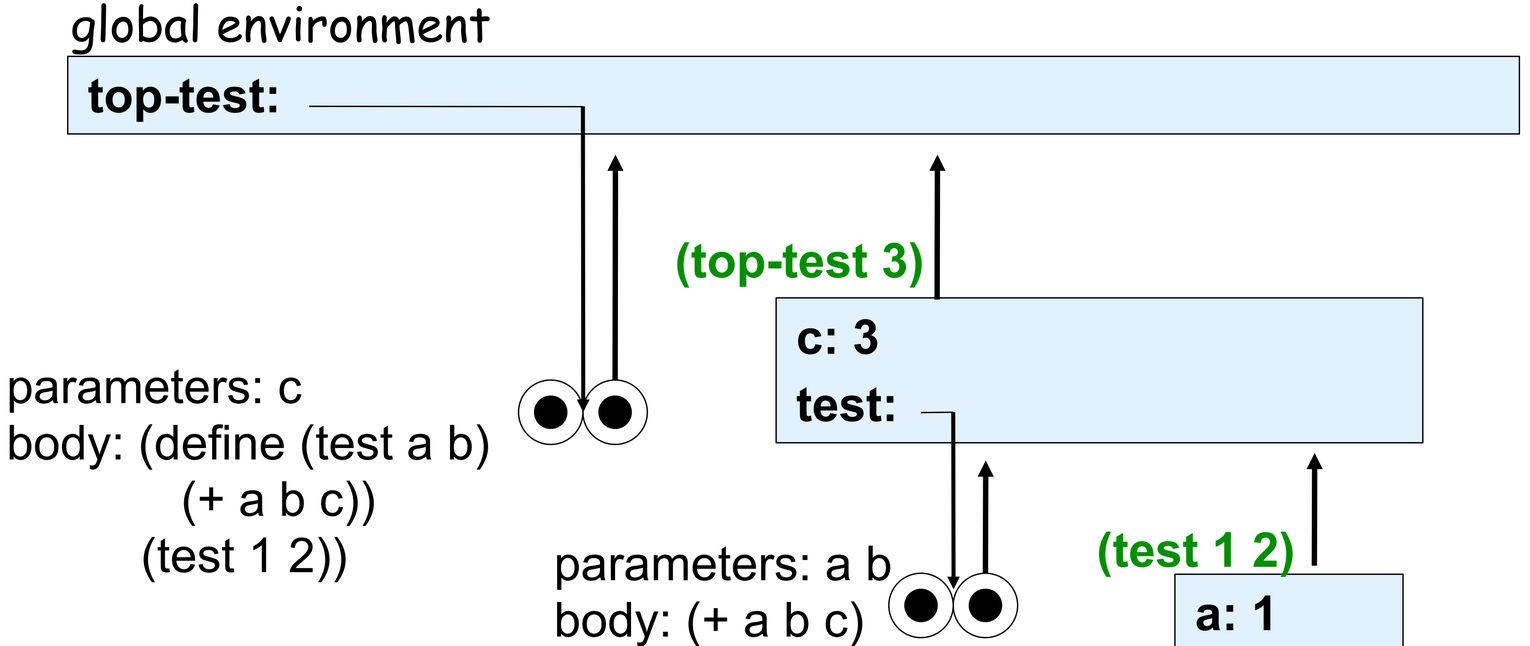


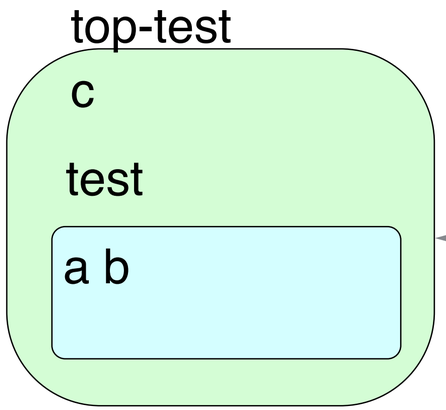
Lexicale scoping (3)

De plaats van de definitie helpt!

```
(define (top-test c)
  (define (test a b)
    (+ a b c))
  (test 1 2))
```

```
> (top-test 3)
6
```





het probleem is weg omdat test nu gedefinieerd staat binnen een procedure die c bindt

c wordt WEL gevonden in de toren van omgevingen

Lexicale scoping <> dynamische scoping

Scheme is een lexicaal gescoopte taal: bij de evaluatie van de body van een procedure wordt de waarde van vrije variabelen gezocht in de omgeving waarin de procedure is gedefinieerd

dat is maar 1 plaats; je ziet het in de programmatekst (= lexicaal of statisch)

Alternatief dat in sommige talen naar voor wordt geschoven is dynamische scoping; daar wordt de waarde van de vrije variabelen gezocht in de omgeving waar de procedure wordt opgeroepen

dat kunnen verschillende plaatsen zijn; je moet over de uitvoering van het programma redeneren (= dynamisch)

Les 3: Procedures&Processen