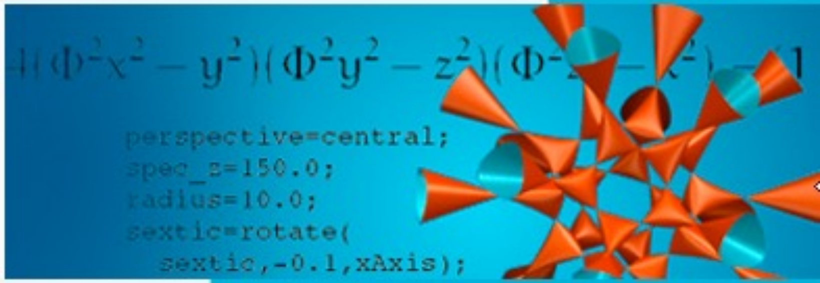
Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



```
perspective=central;
spec_z=150.0;
radius=10.0;
sextic=rotate(
  sextic,-0.1,xAxis);
```

[Script 11]

Local definitions

- So far:
  - Definitions are visible throughout the program
  - From the definition

- Some definitions are only used in a small part of the program
  - Such definitions unnecessarily fill the global environment
  - Make it difficult to understand the program
  - Prevent other definitions under the same name elsewhere

Local definitions

- Example

```
; (list-of String) -> String
; appends all strings in l with blank space between elements
(check-expect (append-all-strings-with-space (list "a" "b" "c"))
              "a b c ")
(define (append-all-strings-with-space l)
 (foldr string-append-with-space
        ""
        l))

String String -> String
; juxtapoint two strings and prefix with space
(define (string-append-with-space s t)
 (string-append s " " t))
```

Local definitions

- Example

```
; (list-of String) -> String
; appends all strings in l with blank space between elements
(check-expect (append-all-strings-with-space (list "a" "b" "c"))
              "a b c ")
(define (append-all-strings-with-space l)
  (foldr string-append-with-space

String
; juxtapoint two strings and prefix with space
(define (string-append-with-space s t)
  (string-append s " " t))
```

foldr is a primitive function that corresponds to our op-elements.

The function string-append-with-space function is only used here.

Local definitions

- Example

```
; (list-of String) -> String
; appends all strings in l with blank space between elements
(check-expect (append-all-strings-with-space (list "a" "b" "c"))
                    "a b c ")
(define (append-all-strings-with-space l)
   (local
          [ ; String String -> String
            ; juxtapoint two strings and prefix with space
            (define (string-append-with-space s t)
              (string-append s " " t))]
          (foldr string-append-with-space
                 ""

                 l))
)
```

Local definitions

- Example

```
; (list-of String) -> String
; appends all strings in l with blank space between elements
(check-expect (append-all-strings-with-space (list "a" "b" "c"))
              "a b c ")
(define (append-all-strings-with-space l)
  (local
        [; String String -> String
         ; juxtapoint two strings anc oref
          efine (string-append-witl spa
          string-append s " " t))]
          dr string-append-with-space
              ""
              l))
)
```

Keyword for local definitions

local definitions

Scope of validity of local definitions

String-append-with-space can no longer be called here.

Local definitions

- (local ...) are expressions
  - Can be used anywhere where expression is expected
  - First clause contains one or more definitions in square brackets
  - Second clause is a sub-expression
    - Can use local definitions
    - The result is the result of the local expression

- Examples

```
> ( local [( define (f x) (+ x 1))] (+ (* (f 5) 6) 7))
43
 > (+ ( local [( define (f x) (+ x 1))] (* (f 5) 6)) 7)
43
> (+ (* ( ( ( local [( define (f x) (+ x 1))] f) 5) 6) 7)
43
```

Local definitions

- (local ...) are expressions
  - Can be used anywhere where expression is expected
  - First clause contains one or more definitions in square brackets
  - Second clause is a sub-expression
    - Can use local definitions
    - The result is the result of the local expression

- Examples

```
> ( local [( define (f x) (+ x 1))] (+ (* (f 5) 6) 7))
43
 > (+ ( local [( define (
43
> (+ (* ( ( ( local [( define (f x) (+ x 1))]   ) 6) 7)
43
```

The value of the local expression is the locally defined function itself.

Local definitions

- Constants can also be defined locally

- Local definitions can access the local environment
  - For example on argument values

Local definitions - Context

- Example:

```
(define (power8 x)
   (* x (* x (* x (* x (* x (* x (* x (* x x))))))))
> (power8 2)
256
```

- Eight multiplications are performed here
- Rewrite using the well-known calculation rule: $a^{2*b} = a^b * a^b$

If we don't have to calculate a^b multiple times, we need fewer multiplications.

Local definitions - Context

```
(define (power8-fast x)
  (local
    [(define r1 (* x x))
     (define r2 (* r1 r1))
     (define r3 (* r2 r2))]
    r3))
```

$r1 = x * x = x^2$

$r2 = r1^2 = x^2 * x^2 = x^4$

$r3 = r2^2 = x^4 * x^4 = x^8$

A total of 3 multiplications instead of 7.

Local definitions - Context

```
(define (power8-fast x)
  (local
    [(define r1 (* x x))
     (define r2 (* r1 r1))
     (define r3 (* r2 r2))]
    r3))
```

New: when defining a local constant, we can use the parameters of the surrounding definition.

Applies to all local definitions.

Definition of a local constant.

We already know: when defining a constant, we can use functions and constants that have already been defined.

Local constants

- Constants are only calculated once during definition
- Intermediate results can be saved in this way
- Global constants do not help in this example: they cannot depend on function parameters
- Abstraction through local constants
  - Avoidance of redundancy
    - In the program text
    - In the calculation
  - Assigning a name to an intermediate result

Static redundancy.
(Don't Repeat Yourself)

Dynamic redundancy.

Names for intermediate results

```
(define (posn+vel p q)
  (make-posn (+ (posn-x p) (vel-delta-x q))
                        (+ (posn-y p) (vel-delta-y q))))


(define (posn+vel p q)
  (local [(define new-x (+ (posn-x p) (vel-delta-x q)))
               (define new-y (+ (posn-y p) (vel-delta-y q)))]
          (make-posn new-x new-y)))
```

Meaning of the values becomes clear. Expression less convoluted.

Avoidance of dynamic redundancy

- Successive Squaring
  - Algorithm for calculating powers
  - Generalization of the power8-fast approach

- First attempt
  - Exponent is a natural number
  - Recognizing natural numbers as a recursive data type
  - Implementation as a recursive function

```
NaturalNumber Number -> Number
(define (exponential n x)
  (if (zero? n)
      1
      (* x (exponential (sub1 n) x))))
```

Avoidance of dynamic redundancy

```
NaturalNumber Number -> Number
(define (exponential n x)
   (if (zero? n)
       1
       (* x (exponential (sub1 n) x))))
```

- Expansion of the function call e.g. for n = 8
- (exponential 8 x)

```
≡ (* x (* x (* x (* x (* x (* x (* x x)))))))
≡ (* (* (* x x) (* x x)) (* (* x x) (* x x)))
```

Associativity

(* x x) is calculated several times: dynamic redundancy.

Avoidance of dynamic redundancy

- How can the known solution be generalized?
- Previous assumption: Exponent is divisible by two (even)
- Generalization: need case differentiation and strategy for odd exponents

```
(define (exponential-fast n x)
  (if (zero? n)
      1
      (local
        [(define y (exponential-fast (quotient n 2) x))
         (define z (* y y))]
        (if (odd? n) (* x z) z))))
```

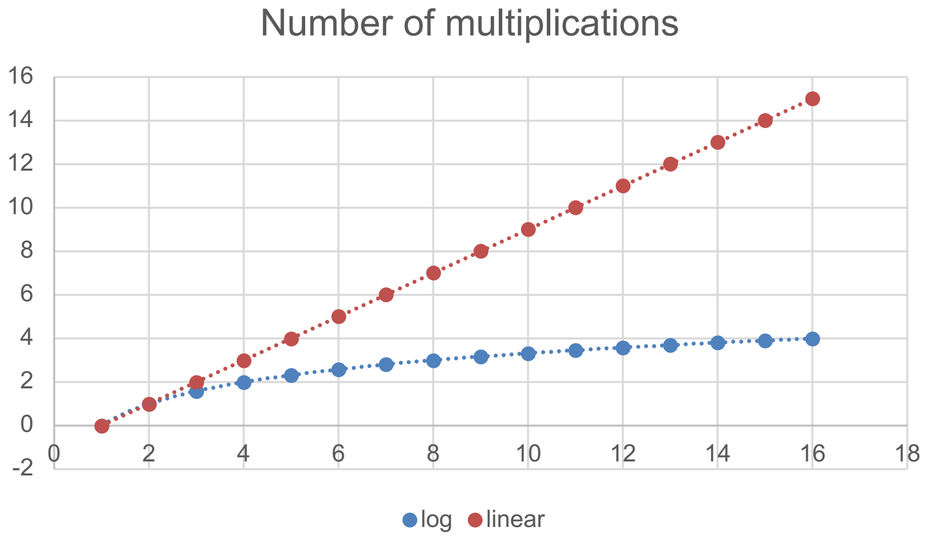
n even: $y = x^{n/2}$
n odd: $y = x^{(n-1)/2}$

quotient: integer division

n even: $z = x^{n/2} * x^{n/2} = x^{n}$
n odd: $z = x * x^{(n-1)/2 (n-1)/2} = x^{n-1}$

Avoidance of dynamic redundancy

- Successive squaring requires approx. $\log_2 (n)$ multiplications compared to (n - 1) multiplications of the naive implementation
  - Avoidance of dynamic redundancy
  - Faster execution



- Demo runtime performance

Scope of local definitions

- Area in the program where a definition may be used: "Area of validity" or "Scope"

- In our case: "lexical scoping" or "static scoping"
  - Scope depends on the location of the definition in the source text
  - In sub-expressions of the "local" expression

Scope of local definitions

```
(local [(define (f n) (if (zero? n)
                          0
                          (+ x (f (sub1 n)))))
        (define x 5)]
  (+ x (f 2)))
```

Use in scope.

Use in scope.

```
(+
  (local [(define x 5)] (+ x 3))
  x)
```

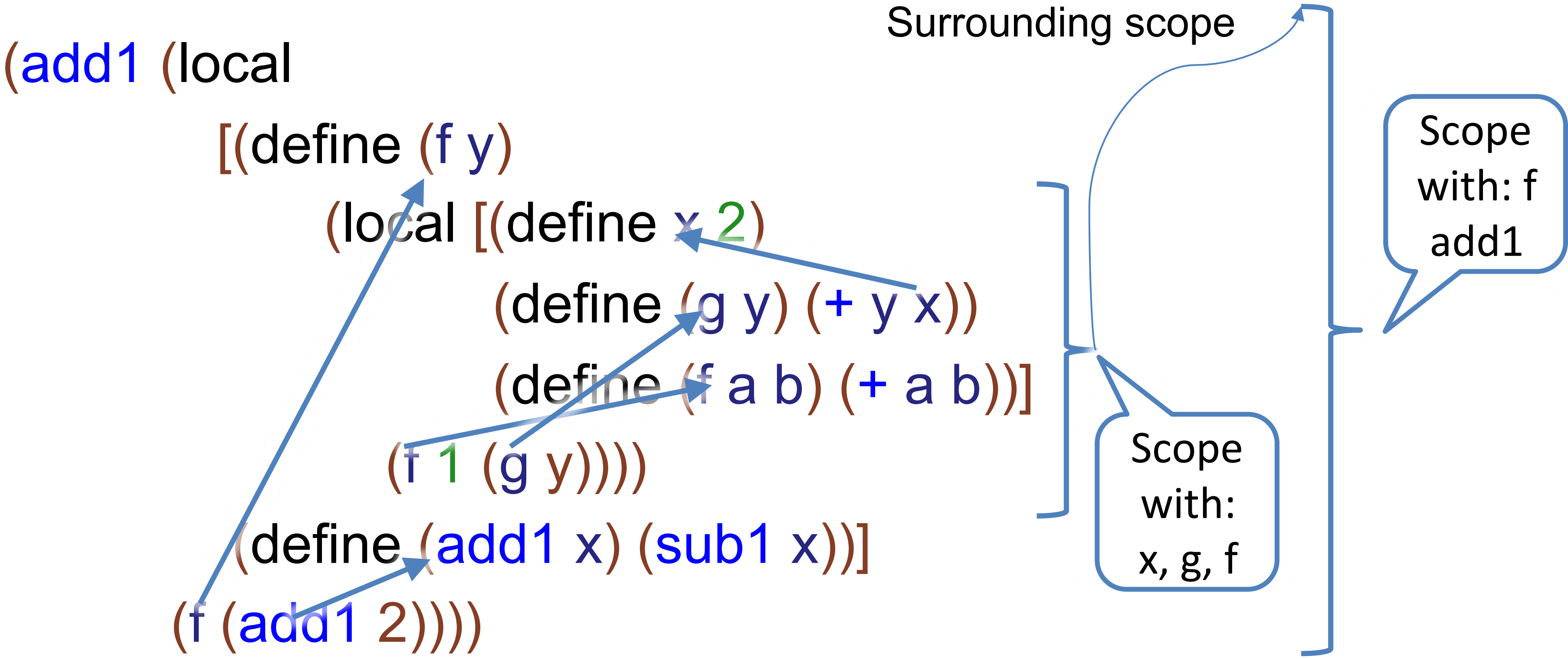
Use outside the scope.

Use in scope.

Nested scopes

- The scope of a local definition is embedded in the scope of the global environment
- (local ...) are expressions
  - May be used wherever expressions are permitted
  - Can also occur in sub-expressions of (local ...)
- Scopes are nested!
  - What happens when names are repeated?

Nested scopes



```
(add1 (local
        [(define (f y)
           (local [(define x 2)
                   (define (g y) (+ y x))
                   (define (f a b) (+ a b))]
             (f 1 (g y))))
         (define (add1 x) (sub1 x))]
     (f (add1 2))))
```

Surrounding scope

Scope with: f add1

Scope with: x, g, f

- Names of the surrounding scope may be reused (overwritten)
  - Search for definition from the inside out

Functions as values: Closures

- Previously: locally defined constants can access function parameters

- Is this also possible for locally defined functions?

- Example
  - A derivative function should return the derivative of the function passed as an argument
  - The result function is to be output using a plot function, for example

```
(Number -> Number) -> Image
( define (plot-function f) ...)
(Number -> Number) -> (Number -> Number)
( define (derivative f) ...)
```

Functions as values: Closures

- An experiment:

```
(Number -> Number) -> (Number -> Number)
(define (derivative f)
  (local
    [(define delta-x 0.001)
     (define (delta-f-x x) (- (f (+ x delta-x)) (f x)))
     (define (g x) (/ (delta-f-x x) delta-x))]
  g))
```

Are we allowed to return locally defined functions? What could be problems?

Functions as values: Closures

- An experiment:

```
(Number -> Number) -> (Number -> Number)
(define (derivative f)
  (local
    [(define delta-x 0.001)
     (define (delta-f-x x) (- (f (+ x delta-x)) (f x)))
     (define (g x) (/ (delta-f-x x) delta-x))]
  g))
```

The locally defined function accesses arguments from derivative. How can the function be evaluated outside the function call that created it?