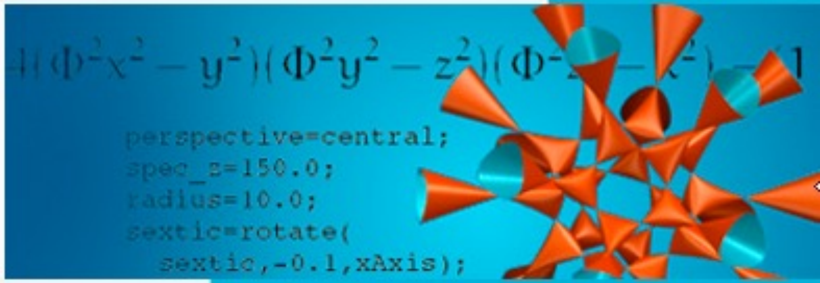
Declarative programming

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 9.6 - 10.5]

Quote: Syntactic sugar

- Quote can be expressed using familiar language
  - Abbreviated spelling
  - Particularly helpful for nested lists

- Transformation rules are applied recursively until the expression no longer contains a quote

$'(e_1 \ldots e\ )_n \Longrightarrow (\text{list } 'e_1 \ldots 'e\ )_n$

Recursive application to sub-expressions

If l is a literal: 'l ⇒ l

If n is an identifier: 'n ⇒ 'n

"Symbol"

Quote: Syntactic sugar

```
'(1 (2 3))
```

```
(list '1 '(2 3))
```

```
(list 1 '(2 3))
```

```
(list 1 (list '2 '3))
```

```
(list 1 (list 2 '3))
```

```
(list 1 (list 2 3))
```

Symbols

- New kind of values
- On the representation of symbolic data

- Syntax:
  - Starts with apostrophe
  - Same as identifier (no spaces)

- Freely definable identifier (like string literal)

- But: no calculation (like concatenation) on symbols

Symbols

- Main functionality: Identity comparison

```
> (symbol=? 'x 'x)
true
```

Do two symbols match?

```
> (symbol=? 'x 'y)
false
```

Example of symbols

Quote

```
(define x 3)
(define y '(1 2 x 4))
> y
```

What result do you expect?

Quote

```
(define x 3)
(define y '(1 2 x 4))
> y
(list 1 2 'x 4)
```

What result do you expect?

When converting to a list, identifiers become symbols.

Quote

```
> '(1 2 (+ 3 4))
```

What result do you expect?

Quote

- Strings have no relationship to the expressions they represent
  - "(+ 3 4)" ≠ 7

- The same applies to symbols
  - The '+' symbol has no relation to addition
  - The symbol 'x has no relation to the constant x

Quote

- Can you mix quote and calculation?
  - Is there a way to write from
    (1 2 (+ 3 4))
    to (list 1 2 7)?

- Example:

```
; Number -> (List of Number)
; given n, generates the list ((1 2) (m 4)) where m is n+1
(check-expect (some-list 2) (list (list 1 2) (list 3 4)))
(check-expect (some-list 11) (list (list 1 2) (list 12 4)))
(define (some-list n) ...)
```

How can the functional body be comfortably defined?

Quote

```
; Number -> (List of Number)
; given n, generates the list ((1 2) (m 4)) where m is n+1
(check-expect (some-list 2) (list (list 1 2) (list 3 4)))
(check-expect (some-list 11) (list (list 1 2) (list 12 4)))
(define (some-list n) '((1 2) ((+ n 1) 4)))
```

Naive approach

```
> (some-list 2)
(list (list 1 2) (list (list (list '+ 'n 1) 4))
```

Wrong result

Quasi-quota

```
; Number -> (List of Number)
; given n, generates the list ((1 2) (m 4)) where m is n+1
(check-expect (some-list 2) (list (list 1 2) (list 3 4)))
(check-expect (some-list 11) (list (list 1 2) (list 12 4)))
(define (some-list n) '((1 2) ((+ n 1) 4)))
```

We need a way to temporarily suspend quoting.

Quasi-quota

- Solution: "quasiquote"
  - Slanted apostrophe (`)
  - Instead of an even apostrophe (') as in the quote

- Functionality with quote:

```
> `(1 2 3)
(list 1 2 3)
> `(a ("b" 5) 77)
(list 'a (list "b" 5) 77)
```

Quasi-quota

- Special feature: quote can be interrupted ("unquote")
  - This takes you back to the programming language
  - Notation: Comma (the symbol: ,)

- Unquote applies to the following expression
  - I.e. for a literal, an identifier or an entire bracket

```
> `(1 2 ,(+ 3 4))
(list 1 2 7)

> `(1 2 ,(+ 3 4) x)
(list 1 2 7 'x)
```

Transformation of quasiquote

- Transformation rules are applied recursively until the expression no longer contains a quote

$$`(e_1 \ldots e\ )_n \Rightarrow (\mathrm{list}\ `e_1 \ldots `e\ )_n$$

$$`,e \Rightarrow e$$

If l is a literal: $`l \Rightarrow l$

If n is an identifier: $`n \Rightarrow {}'n$

Quasiquote: Syntactic sugar

```
`(1 ,2 ,(+ 3 4))
⇒
(list `1 `,2 `,(+ 3 4))
⇒
(list 1 `,2 `,(+ 3 4))
⇒
(list 1 2 `,(+ 3 4))
⇒
(list 1 2 (+ 3 4))
```

Is evaluated to:
`(list 1 2 7)`

Quasi-quota

```
; Number -> (List of Number)
; given n, generates the list ((1 2) (m 4)) where m is n+1
(check-expect (some-list-v2 2) (list (list 1 2) (list 3 4)))
(check-expect (some-list-v2 11) (list (list 1 2) (list 12 4)))
(define (some-list-v2 n) `((1 2) (,(+ n 1) 4)))
```

quasiquote

unquote

Quasiquote: Application example

- Quasi-quota
  - Embedding programs that are evaluated
  - In data
  - And vice versa
- Rule for generating the data
  - Possibly more legible
  - Better maintainability through explicit dependencies
  - Dynamic generation of data
- Principle of template and scripting languages
  - E.g. generation of HTML code

Quasiquote: Application example

```
String String -> deeply nexted list
; produces a (representation of) a web page with
; given author and title
(define (my-first-web-page author title)
  `(html
     (head
       (title ,title)
       (meta ((http-equiv "content-type")
              (content "text-html"))))
     (body
       (h1 ,title)
       (p "I, " ,author ", made this page."))))
```

Quoted list:
Page template

Unquote: Hole in
page template

Quasiquote: Application example

```
String String -> deeply nexted list
; produces a (representation of) a web page with
; given author and title
(define (my-first-web-page author title)
  `(html
     (head
       (title ,title)
       (meta ((http-equiv "content-type")
              (content "text-html"))))
     (body
       (h1 ,title)
       (p "I, " ,author ", made this page."))))
```

Consistent use of title

Consistent use of title

```
'(html
  (head
    (title
      "Hello World"
    )
    (meta (
      (http-equiv "content-type")
      (content "text-html")))
  )
  (body
    (h1
      "Hello World"
    )
    (p
      "I, "
      "Matthias"
      ", made this page."
    )
  )
)
```

```
<html>
  <head>
    <title>
      Hello World
    </title>
    <meta
      http-equiv="content-type"
      content=="text-html" />
  </head>
  <body>
    <h1>
      Hello World
    </h1>
    <p>
      I,
      Matthias,
      made this page.
    </p>
  </body>
</html>
```

Quasiquote: Application example

- unquote
  - Not only: (consistent) insertion of the holes
  - Also: Generate lists

Quasiquote: Application example

```
; List-of-numbers -> ... nested list ...
creates a row for an HTML table from a list of numbers
(define (make-row l)
   (cond
      [(empty? l) empty]
      [else (cons `(td ,(number->string (first l)))
                       (make-row (rest l)))]))

; List-of-numbers List-of-numbers -> ... nested list ...
creates an HTML table from two lists of numbers
(define (make-table row1 row2)
   `(table ((border "1"))
        ,(cons `tr (make-row row1))
        ,(cons `tr (make-row row2))))

> (make-table '(1 2 3 4 5) '(3.5 2.8 -1.1 3.4 1.3))
```

Quasiquote: Application example

```
; List-of-numbers -> ... nested list ...
creates a row for an HTML table from a list of numbers
(define (make-row l)
   (cond
      [(empty? l) empty]
      [else (cons `(td ,(number->string (first l)))
                  (make-row (rest l)))]))

; List-of-numbers List-of-numbers -> ... nested list ...
creates an HTML table from two lists of numbers
(define (make-table row1 row2)
   `(table ((border "1"))
      ,(cons `tr (make-row row1))
      ,(cons `tr (make-row row2))))

> (make-table '(1 2 3 4 5) '(3.5 2.8 -1.1 3.4 1.3))
```

Mixing quote mechanism and cons constructor calls

Recursive creation of a list

Very compact generation of the HTML page

Quasiquote: Application example

```
> (make-table '(1 2 3 4 5) '(3.5 2.8 -1.1 3.4 1.3))
(list
 'table
 (list (list 'border "1"))
 (list
  'tr
  (list 'td "1")
  (list 'td "2")
  (list 'td "3")
  (list 'td "4")
  (list 'td "5"))
 (list
  'tr
  (list 'td "3.5")
  (list 'td "2.8")
  (list 'td "-1.1")
  (list 'td "3.4")
  (list 'td "1.3")))
```

Quasiquote: Application example

```
; List-of-numbers -> ... nested list ...
creates a row for an HTML table from a list of numbers
(define (make-row l)
  (cond
    [(empty? l) empty]
    [else (cons `(td ,(number->string (first l)))
                (make-row (rest l)))]))
```

Why mix quote and cons?

```
; List-of-numbers List-of-numbers -> ... nested list ...
creates an HTML table from two lists of numbers
(define (make-table row1 row2)
  `(table ((border "1"))
     ,(cons `tr (make-row row1))
     ,(cons `tr (make-row row2))))

> (make-table '(1 2 3 4 5) '(3.5 2.8 -1.1 3.4 1.3))
```

Quasiquote: Application example

```
; Lis                          ist ...
crea                           e from a list of numbers
(define (          row l)
  (cond
    [(empty? l) empty]
    [else `(td ,(number->string (first l))
                ,(make-row (rest l))))))

; List-of-numbers l          umbers -> ... nested lis
creates
(define
  `(tab
     ,
     ,(cons `tr (make-row row2))))

> (make-table '(1 2 3 4 5) '(3.5 2.8 -1.1 3.4 1.3))
```

Without cons.

Result becomes the next element in the list.
Result is list → nested list instead of "list of lists"

```
> (make-table '(1 2 3 4 5)
    '(3.5 2.8-1.1 3.4 1.3))
(list
 'table
 (list (list 'border "1"))
  (list 'tr 'td "1"
   (list 'td "2"
    (list 'td "3"
     (list 'td "4"
      (list 'td "5" '()))))))
 list 'tr 'td "7/2"
 (list 'td "14/5"
  (list 'td "-11/10"
   (list 'td "17/5"
    (list 'td "13/10" '()))))))
```

Generic data structures

- So far:
  - A separate type (struct) for each data structure
  - I.e. specific constructor function, selector functions, predicate
  - Functions that operate on it are not reusable

Generic data structures

- Example:

```
(define-struct person (name father mother))

(define (person-has-ancestor p a)
   (cond [(person? p)
            (or
             (string=? (person-name p) a)
             (person-has-ancestor (person-father p) a)
             (person-has-ancestor (person-mother p) a))]
           [else false]))
```

But the functionality is actually generally useful on all data structures

Generic data structures

- Tree structures are omnipresent
  - Hierarchies in personnel
  - Directory structures
  - General: Search trees

- Searching for an element makes sense for all data structures

- Representation through own structure
  → Search must be re-implemented each time
  - Violation of the Don't Repeat Yourself Principle

Generic data structures

- Operations on tree structures
  - Search for an element
  - Determining the depth
  - Counting the nodes/leaves
  - Insert
  - Remove
  - Etc.
- Desirable: "generic implementation"
  - In other words, a single implementation that works for all data types
  - Necessary:
    - Abstracting from the exact data type
    - Names are irrelevant

S-Expressions

- "S-expressions" are expressions that generate structured data
  - Universal data format
  - Structuring the data is part of the data
  - Every S expression has the same data type
- Invention of the LISP programming language
  - Historically: Abbreviation for "List Processing"
  - LISP is a predecessor of BSL or Racket

S-Expressions

- Data definition for S-Expressions

```
An S-Expression is one of:
; - a Number
; - a String
; - a symbol
; - a Boolean
; - an image
; - empty
; - a (list-of S-expression)
```

What's new about this?

S-Expressions

- Data definition for S-Expressions

```
An S-Expression is one of:
; - a Number
; - a String
; - a symbol
; - a Boolean
; - an image
; - empty
; - a (list-of S-expression)
```

What's new about this?

Use of symbols as a structuring tool

No structs permitted

S-Expressions

- Examples:
  - (list 1 (list 'two 'three) "four")
  - "Hi

- No S-expression
  - (make-posn 1 2)
  - (list (make-student "a" "b" 1))

Use of structs.

S-Expressions

- Instead of structural instances:
  - Use of symbols to encode the structure

- Instead of (make-posn 1 2)
  - '(posn 1 2) or
  - '(posn (x 1) (y 2))

S-Expressions

- Inst ces:
  - Us e structure

Specifies the structure as an identifier. Becomes a symbol when quoting.

- Instea of (make-posn 1 2)
  - '(posn 1 2) or
  - '(posn (x 1) (y 2))

Coding the fields via the sequence

Alternative: Coding the fields using their names

S-Expressions

- By means of Struct:

```
(make-person "Heinz"
   (make-person "Horst" false false)
   (make-person "Hilde" false false))
```

- Using S-Expression

```
'(person "Heinz"
   (person "Horst" #f #f)
   (person "Hilde" #f #f))
```

or

```
'(person "Heinz"
   (father (person "Horst" (father #f) (mother #f))
   (mother (person "Hilde" (father #f) (mother #f)))))
```

Attention: Spelling for truth values: #t and #t
To differentiate between identifiers

S-Expressions

- Generality of S-Expressions enables you to write general functions

- However, writing specific functions is made more difficult

- Dependence on certain structural properties is difficult to express

Quote and S-expressions

- The quote operator always returns an S expression

- Attention: This does not apply to quasiquote!
  - Quasiquote allows return to racket
  - A list generated in this way can therefore contain structure instances

Programs as S-Expressions

- Applying the quote operator to an expression
  - Delivers S-expression
  - Represents the expression as data

- Every expression to which the quote operator is applied becomes an S expression
  - Function names or keywords become symbols
  - Nesting of expressions becomes nesting of lists
  - This is how programs can be represented as data

- Outlook: Racket (but not BSL) offers eval function
  - Expects S-expression as argument
  - Interprets this S-expression as a program
  - Returns the result of the program