Domänenspezifische Sprachen blitzschnell entwickeln mit Racket

Michael Sperber

Created: 2022-05-16 Mon 18:17

@ctive group

Active Group GmbH

- Individualsoftware
- funktionale Programmierung
- Scala, Clojure, F#, Haskell, OCaml, Erlang, Elixir, Swift
- Schulungen und Coaching
- Tübingen

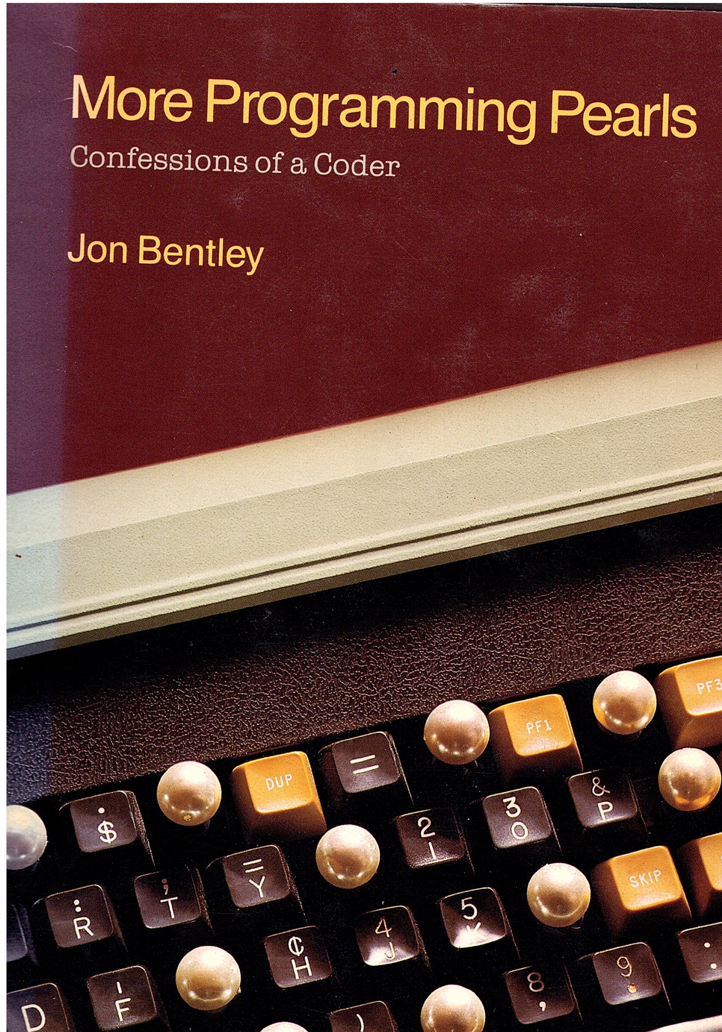
Blog: https://funktionale-programmierung.de

@ctive group

Domänenspezifische Sprachen

- SQL
- HTML
- CSS
- PostScript
- PHP
- YAML
- LaTeX
- VBA
- PIC
- DOT
- PlantUML
- Gherkin
- BPMN

Quelle



@ctive group

I/O Fit For Humans

Software beauty is sometimes skin deep. No matter how wonderful your program is on the inside, an ill-designed interface can drive users away. And more than one shoddy program has fooled users with snazzy input and output. Input and output may be a small part of the system from your view as a programmer, but the interface is a large part of the user's view of your software.

These columns describe several aspects of input and output. Column 9 applies principles of language design to making software interfaces that are little languages. Column 10 is about producing documents that are pleasant to look

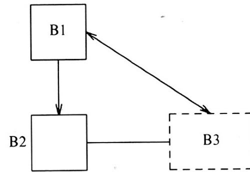
Pic

The third approach to describing pictures is the topic of this column: a little language. In Kernighan's Pic language, for instance, the first figure in this section is described as

```
ellipse "Source" "Code"
arrow
box "Compiler"
arrow
ellipse "Object" "Code"
```

The first input line draws an ellipse of default size and stacks the two strings at its center. The second line draws an arrow in the default direction (moving right), and the third line draws a box with the text at its center. The implicit motion after each object makes it easy to draw the picture and convenient to add new objects to an existing picture.

This nonsense picture illustrates several other devices that Pic supports, including lines, double arrowheads, and dashed boxes.



The program that draws it places objects by implicit motions, by explicit motions, and by connecting existing objects:

```
boxht = .4; boxwid = .4
down     # set default direction
B1: box "B1"
arrow
B2: box
"B2 " at B2.w rjust
line right .6 from B2.e
B3: box dashed wid .6 "B3"
line <-> from B3.n to B1.e
```

The `boxht` and `boxwid` variables represent the default height and width of a box in inches. Those values can also be explicitly set in the definition of a particular box. Text following the # character is a comment, up to the end of the line. Labels such as `B1`, `B2` and `B3` name objects; `LongerName` is fine too. The western point of box `B2` is referred to as `B2.w`; one could also refer to `B2.n` or `B2.nw`, for the northwest corner. A line of the form *string* `at` *position* places a text string at a given position; `rjust` right-justifies the string

Was gehört zur Implementierung einer Sprache?

- Syntax
- Semantik
- Dokumentation
- Compiler oder Interpreter
- IDE

Racket

- Programmiersystem, seit 1995 entwickelt
- ursprünglich für die Anfängerausbildung entwickelt
- Basis: funktionale Programmiersprache Scheme
- "language-oriented programming"

Sprachen in Racket

```
#lang racket
(require 2htdp/image) ; draw a picture
(let sierpinski ([n 8])
  (cond
    [(zero? n) (triangle 2 'solid 'red)]
    [else (define t (sierpinski (- n 1)))
```

```
#lang scribble/base
@; Generate a PDF or HTML document
@title{Bottles: @italic{Abridged}}
@(apply
  itemlist
  (for/list ([n (in-range 100 0 -1)])
```

```
#lang typed/racket
;; Using higher-order occurrence typing
(define-type SrN (U String Number))
(: tog ((Listof SrN) -> String))
(define (tog l)
  (apply string-append (filter string? l)))
```

```
#lang datalog
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :-
  parent(A, C), ancestor(C, B).
parent(john, douglas).
parent(bob, john).
```

```
#lang racket/gui
(define f (new frame% [label "Guess"]))
(define n (random 5)) (send f show #t)
(define ((check i) btn evt)
  (message-box "." (if (= i n) "Yes" "No")))
(for ([i (in-range 5)])
```

```
#lang web-server/insta
;; A "hello world" web server
(define (start request)
  (response/xexpr
   '(html
     (head (title "Racket"))
```

@ctive group

DrRacket

Namenlos - DrRacket

Namenlos ▾ (define ...) ▾ Syntaxprüfung Stepper Start Stop

```
; Ein Gürteltier hat folgende Eigenschaften:
; - lebendig oder tot
; - Gewicht
(define-record-functions dillo
  make-dillo
  (dillo-alive? boolean)
  (dillo-weight number))
```

Willkommen bei DrRacket, Version 7.4 [cs].
Sprache: Schreibe Dein Programm! - Anfänger; memory limit: 128 MB.
>

Schreibe Dein Programm! - Anfänger ▾ 3:2 563.85 MB

@ctive group

Abkömmling von Lisp

```
(+ 23 42)
(+ 23
   (* 42 7))

(define pi 25)

(define (circumference radius)
  (* 2 pi radius))

(circumference 5)
```

S-Expressions

```
(<operator> <operand> ...)
```

@ctive group

Syntaktischer Zucker

```
(let ((x 5)
      (y 7))
  (+ x y))

-->

((lambda (x y) (+ x y)) 5 7)
```

@ctive group

Racket-Sprache

```
#lang racket

(+ 23 42)
(+ 23
   (* 42 7))

(define pi 25)

(define (circumference radius)
  (* 2 pi radius))
```

Makros

```
(define-syntax-rule (swap! x y)
  (let ((z x))
    (set! x y)
    (set! y z)))

(define a 23)
(define b 42)
(swap! a b)
```

@ctive group

Makro in C

```
#define min(x, y) (x < y) ? x : y
```

@ctive group

Makro in C

```
#define min(x, y) ((x) < (y)) ? (x) : (y))
```

@ctive group

Makro in Racket

```
(define-syntax-rule (min x y)
  (let ((x* x)
        (y* y))
    (if (< x* y*)
        x*
        y*)))
```

Hygiene

```
(define-syntax-rule (swap! x y)
  (let ((z x))
    (set! x y)
    (set! y z)))

(define a 15)
(define z 22)

(swap! a z)
```

@ctive group

Modulsystem

```
#lang racket
(provide swap!)
(define-syntax-rule (swap! x y)
  (let ((z x))
    (set! x y)
    (set! y z)))
```

Makros importieren

```
#lang racket
(require "swap.rkt")

(define a 15)
(define z 22)

(swap! a z)
```

@ctive group

Eingebettete Domänenspezifische Sprache: scsh

```
;; M4 preprocess each file in the current directory, then pipe
;; the input into cc. Errlog is foo.err, binary is foo.exe.
;; Run compiles in parallel.
(for-each (lambda (file)
            (let ((outfile (replace-extension file ".exe"))
                  (errfile (replace-extension file ".err")))
              (& (| (m4) (cc -o ,outfile))
                 (< ,file)
                 (> 2 ,errfile))))
          (directory-files))
```

@ctive group

Haskell List Comprehensions

```
let triangles =
      [ (a,b,c) | c <- [1..10], b <- [1..10], a <- [1..10] ]

let rightTriangles =
      [ (a,b,c) |
        c <- [1..10],
        b <- [1..c],
        a <- [1..b],
        a^2 + b^2 == c^2 ]
```

Racket List Comprehensions

```
(define triangles
  (|| (list a b c)
      (<- c (from-to 1 10))
      (<- b (from-to 1 10))
      (<- a (from-to 1 10))))

(define right-triangles
  (|| (list a b c)
      (<- c (from-to 1 10))
      (<- b (from-to 1 c))
      (<- a (from-to 1 b))
      (= (+ (sqr a) (sqr b)) (sqr c))))
```

@ctive group

Varargs

```
(define-syntax swap!
  (syntax-rules ()
    ((swap! x y)
     (let ((z x))
       (set! x y)
       (set! y z)))
    ((swap! x y z)
     (begin
       (set! x y)
       (set! y z)))))
```

@ctive group

Varargs

```
(define-syntax destructure
  (syntax-rules ()
    ((destructure exp (v1 ...) body)
     (apply (lambda (v1 ...)
              body)
            exp))))

(destructure (list 1 2 3) (a b c) (+ a b c))
```

@ctive group

Varargs

```
(define-syntax-rule (destructure exp (v1 ...) body)
  (let ((v exp))
    (destructure* v (v1 ...) body)))

(define-syntax destructure*
  (syntax-rules ()
    ((destructure* v () body) body)
    ((destructure* v (v1 v2 ...) body)
     (let ((v1 (car v))
           (rest (cdr v)))
       (destructure* rest (v2 ...) body)))))
```

@ctive group

Schlüsselwörter in Patterns

```
(define-syntax if*
  (syntax-rules ()
    ((if* test then consequent else alternative)
     (if test consequent alternative))))

(if* (> a b) then 1 else 2)
(if* (> a b) else 1 then 2)
```

@ctive group

Literale in Makros

```
(define-syntax if*
  (syntax-rules (then else)
    ((if* test then consequent else alternative)
     (if test consequent alternative))))

(if* (> a b) then 1 else 2)
(if* (> a b) else 1 then 2)
; if*: bad syntax in: (if* (> a b) else 1 then 2)
```

@ctive group

List Comprehensions im Haskell-Standard

Translation: List comprehensions satisfy these identities, which may be used as a translation into the kernel:

```
[ e | True ]          = [e]
[ e | q ]             = [ e | q, True ]
[ e | b, Q ]          = if b then [ e | Q ] else []
[ e | p <- l, Q ]     = let ok p = [ e | Q ]
                            ok _ = []
                        in concatMap ok l
[ e | let decls, Q ]  = let decls in [ e | Q ]
```

where e ranges over expressions, p over patterns, l over list-valued expressions, b over boolean expressions, *decls* over declaration lists, q over qualifiers, and Q over sequences of qualifiers. `ok` is a fresh variable. The function `concatMap`, and boolean value `True`, are defined in the Prelude.

@ctive group

List Comprehensions in Racket

```
(define-syntax ||
  (syntax-rules (<- let)
    ((|| e #t) (list e))
    ((|| e q) (|| e q #t))
    ((|| e (<- p l) Q ...)
     (let ((ok
            (lambda (p)
              (|| e Q ...))))
       (concat-map ok l)))
    ((|| e (let decls) Q ...)
     (let decls
       (|| e Q ...)))
    ((|| e b Q ...)
     (if b
         (|| e Q ...)
         '()))))
```

@ctive group

Was fehlt noch

- grundsätzlich andere Funktionsweise
- andere Syntax an der Oberfläche

Talk @ rC3 2020:

All Programming Languages Suck? Just Build Your Own!

Was gehört zu einer Sprache?

"Every powerful language has three mechanisms for accomplishing this:

- **primitive expressions**, which represent the simplest entities the language is concerned with,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units."

Abelson/Sussman/Sussman: Structure and Interpretation of Computer Programs (MIT Press, 1996)

@ctive group

Vorgehensweise

- mach's mit Racket
- mach erstmal eine Bibliothek
- mach erstmal eine Kombinatorbibliothek
- mach's erstmal mit Funktionen höherer Ordnung
- mach's erstmal mit Klammern
- abstrahiere über Syntax mit Makros

Wenn's sein muss

- komplette Sprache
- was anderes als Klammern

Wenn's unbedingt sein muss

- mach's mit einer anderen funktionalen Sprache
- mach's mit einer anderen Sprache
- benutze Xtext / Spoofax / JetBrains MPS
- schreibe eigenen Parser etc.

Mach keinen Scheiß

- Syntax: schwierig
- Semantik: schwierig
- Variablen: schwierig
- Abstraktion: schwierig

Was gibt's noch?

- Makros durch Code definieren
- makro-definierende Makros
- Phasen
- `syntax-parse`
- Hygiene umgehen
- "proper tail calls" und First-Class-Continuations für Kontrollabstraktion
- PLT Redex

Zusammenfassung

- DSLs ♥ Racket
- klein anfangen
- groß rauskommen
- viel lernen
- Spaß haben