Informatik II Skript

Steffen Lindner

May 7, 2015

Contents

Chapter 1

Einführung - 14.04.15

Scheme: Ausdrücke, Auswertung und Abstraktion

Dr.Racket: Definitionsfenster (oberer Bereich), Interaktionsfenster (unterer Bereich)

Die Anwendung von Funktionen wird in Scheme ausschließlich in Präfixnotation durchgeführt.

Beispiele

Mathematik	Scheme
44-2	(- 44 2)
f(x,y)	(f x y)
$\sqrt{81}$	(sqrt 81)
9^2	(expt 9 2)
3!	(! 3)

Allgemein: $(< function >< arg1 >< arg2 > ...)$

(+ 40 2) und (odd? 42) sind Beispiele für Ausdrücke, die bei Auswertung einen Wert liefern. (Notation: $\rightsquigarrow$)

(+ 40 2) $\rightsquigarrow$ 42 ($\rightsquigarrow$ = Auswertng / Reduktion / Evalutation)

(odd? 42) $\rightsquigarrow$ #f

Interaktionsfenster: Read $\rightarrow$ Eval $\rightarrow$ Print $\rightarrow$ Read ... (Read-Eval-Print-Loop aka. REPL)

Literale stehen für einen konstanten Wert (auch konstante) und sind nicht weiter reduzierbar.

Literal:

#t, #f (true, false, Wahrheitswerte) (boolean)

"abc", "x", " " (Zeichenkette) (String)

0 1904 42 -2 (ganze Zahlen) (Integer)

0.42 3.1415 (Fließkommazahl) (Reel)

1/2, 3/4 (rationale Zahl) (Rational)

_(″)_/" (Bilder) (Image)

Chapter 2

Ausdrücke, Defines, usw. - 16.04.2015

Auswertung zusammengesetzter Ausdrücke in mehreren Schritten (steps), von "innen nach außen" bis keine Reduktion mehr möglich ist.

$$(+\ (+\ 20\ 20)\ (+\ 1\ 1)) \rightsquigarrow (+\ 40\ (+\ 1\ 1) \rightsquigarrow (+\ 40\ 2) \rightsquigarrow 42$$

Achtung: Scheme rundet bei Arithmetik mit Fließkommazahlen (interne Darstellung ist binär).

Bsp.: Auswertung des zusammengesetzten Ausdrucks 0.7 + (1/2)/0.25 - 0.6/0.3

Arithmetik mit rationalen Zahlen ist exakt.

Ein Wert kann an einen Namen (auch Identifier) gebunden werden , durch

$$(\text{define} < id > < e >)\ (< id > \text{Identifier}, < e > \text{Expression})$$

Erlaubt konsistente Wiederverwendung und dient der Selbstdokumentation von Programmen.

Achtung: Dies ist eine sogenannte Spezifikation und kein Ausdruck. Insbesodnere besitzt diese Spezialform keinen Wert, sondern einen Effekt: Name $< id >$ wird an den Wert von $< e >$ gebunden.

Namen können in Scheme fast beliebig gewählt werden, solange:

1. die Zeichen (kommt noch) nicht vorkommen
2. der Name nicht einem numerischen Literal gleicht
3. kein whitespace (Leerzeichen, Tabulatoren, Return) enthalten ist.

Bsp.: euro → us$

Achtung: Groß-/Kleinschreibung ist in Identifiern nicht relevant.

Eine Lambda-Abstraktion (auch: Funktion, Prozedur) erlaubt die Formulierung von Ausdrücken, die mittels Parametern konkreten Werten abstrahieren:

$$(\text{lambda}\ (< p1 >< p2 > ...) < e >), < e > \text{Rumpf}$$

$< e >$ enthälft Vorkommen der Parameter $< p1 >, < p2 >$...

(lambda ...) ist eine Spezialform. Wert der Lambda-Abstraktion ist $\# < procedure >$

Anwendung (auch: Applikation/Aufruf) der Lambda-Abstraktion führt zur Ersetzung der vorkommenden Parameter im Rumpf durch die angegebenen Argumente:

(lambda (days) (* days (* 155 min-in-a-day))) ⇝ (* 365 (* 155 min-in-a-day)) ⇝ 81468000

In Scheme leitet ein Semikolon einen Kommentar, der bis zum Zeilenende reicht, ein und wird vom System bei der Auswertung ignoriert.

Prozeduren sollten im Programm eine ein-bis zweizeiliger Kurzberschreibung direkt voran gestellt werden.

Chapter 3

Signaturen, Testfälle - 21.04.15

Eine Signatur prüft, ob ein Name an einen Wert einer angegebenen Sorte (Typ) gebunden wird. Signaturverletzungen werden protokolliert.

$$(: < id > < signatur >)$$

Bereits eingebaute Signaturen:

- natural $\mathbb{N}$
- integer $\mathbb{Z}$
- rational $\mathbb{Q}$
- real $\mathbb{R}$
- number $\mathbb{C}$
- boolean
- string
- image

(: ...) ist eine Spezialform ohne Wert, aber Effekt: Signaturprüfung

Prozedur-Signaturen spezifizieren sowohl Signaturen für die Parameter p1, p2, ... , pn als auch den Ergebniswert der Prozedur:

$$(< signaturp_1 > ... < signaturp_n > - >< signatur - ergebnis >)$$

Prozedur-Signaturen werden bei jeder Anwendung eine Prozedur auf Verletzung geprüft.

Testfälle dokumentieren das erwartete Ergebnis einer Prozedur für ausgewählte Argumente:

$$(check - expect < e_1 >< e_2 >)$$

Werte Ausruck $< e_1 >$ aus und teste, ob der erhaltene Wert der Erwarung (= der Wert von $< e_2 >$) entspricht.

Einer Prozedurdefinition sollten Testfälle direkt vorangestellt werden.
Spezialform: Kein Wert, aber Effekt: Testverletzung protokollieren.

Konstruktionsanleitung für Prozeduren

- ; ... (1) Kurzbeschreibung (1-2 zeiliger Kommentar mit Bezug auf Parameter)
- (: ...) (2) Signatur
- (check-expect ...) (3) Testfälle
- (define (lambda (...) ...) (4) Prozedur + Rumpf

Top-Down-Entwurf (Programmieren durch "Wunschdenken")

Bsp.: Zeichen Ziffernblatt (Stunden- und Minutenzeiger) zur Uhrzeit H:m auf einer analogen 24h-Uhr

- Minutenzeiger legt 360°/60 pro Minute zurück (360/60 * m)
- Stundenzeiger legt 360°/12 pro Stunde zurück (360/12 * h + 360/12 * m/60)

Chapter 4

Substitutionsmodell, Fallunterscheidung - 23.0

Reduktionsregeln für Scheme (Fallunterscheidug je nach Ausrucksart)

Wiederhole, bis keine Reduktion mehr möglich:

- Literal (1, "abc", #t, ...) [eval_{lit}]

  l $\rightsquigarrow$ l

- Identifier id (pi, clock-face, ...) [eval_{id}]

  id $\rightsquigarrow$ gebundener Wert

- Lambd-Abstraktion

  (lambda ()) $\rightsquigarrow$ (lambda ()) [eval_{λ}]

- Applikation (f, e1, e2)

  - (1) f, e1, e2 reduziere, erhalte f', e1', e2'
  - (2)
    - Operation f' auf e1', e2', ... falls f' primitive Operation (+, *, ...) [apply_{prim}]
    - Argumentenwert e1', e2', ... Rumpf von f' einsetzen, dann Rumpf reduzieren , falls f' Lambdaabstraktion [apply_{λ}]

Beispiel: Applikation

(+ 40 2)

$\rightsquigarrow$ (#< procedure + > 40 2) $\rightsquigarrow$ 42

eval_{lit} (+)

eval_{lit} (40)

eval_{lit} (2)•

(position-minute-hand 30)

$\rightsquigarrow$ ((lambda (m) (* degrees-per-minute m)) 30)

$\rightsquigarrow$ (* degrees-per-minute 30)

$\rightsquigarrow$ (* degress-per-minute 30)

$\rightsquigarrow$ (#<procedure * > 360/60 30)

Bezeichnen (lambda (x) (* x x)) und (lambda (r) (* r r)) die gleiche Prozedur? $\Rightarrow$ Ja!

Achtung: Das hat Einfluss auf das korrekte Einsetzen von Argumenten für Parameter! (s. apply_{λ})

Das bindenen Vorkommen eines Identifiers x kann im Programmtext systematisch bestimmt werden: suche strik von "innen nach außen" bis zum ersten

- (lambda (x))
- (define x)

(Prinzip der lexikalischen Bindung)

Übliche Notation in der Mathematik: Fallunterscheidung

$$maximum(x_1, x_2) = \begin{cases} x_1, & \text{falls } x_1 \geq x_2 \\ x_2, & \text{sonst} \end{cases}$$

Tests auch (Prädikate) sind Funktionen, die einen Wert der Signatur boolean liefern. Typische primitive Tests:

- (: = (number number → boolean))
- (: < (real real → boolean)), auch $>, \leq, \geq$
- (: string=? (string string → boolean)), auch string>?, string≤?
- (: boolean? (boolean boolean → boolean))
- (: zero? (number → boolean))
- odd?, even?, positive?, negative?, ...

Binäre Fallunterscheidung: if

(if $< t_1 >< e_1 >< e_2 >$)

Mathematisch: $\begin{cases} e_1, & \text{falls } t_1 \\ e_2, & \text{sonst} \end{cases}$

Chapter 5

One-of Signatur - 28.04.15

Die Signatur one-of lässt genau einen der aufgezählten n Werte zu:

$$(\text{one-of} < e_1 > ... < e_n >)$$

Reduktion von if:

$$(\text{if } t_1 \ e_1 \ e_2) \rightsquigarrow \begin{cases} < e_1 >, & \text{falls t1' = \#t ; e2 wird niemals ausgewertet} \\ < e_2 >, & \text{sonst; e1 wird niemals ausgewertet} \end{cases}$$

(1) Reduziere t_1, erhalte t_1'

Spezialform Fallunterscheidung (conditional expression):

$$(\text{cond } (< t_1 >< e_1 >) \ ... \ (< t_n >< e_n >) \ (\text{else} < e_{n+1} >)) \ \text{(else optional)}$$

Werte die Tests in der Reihenfolge $t_1, t_2, ..., t_n$ aus. Sobald t_i # t ergibt werte Zweig e_i aus. e_i ist das Ergebnis der Fallunterscheidung. Wenn t_n # f liefert, dann liefere

$$\begin{cases} Fehlermeldung, & \text{"cond: alle Tests ergaben \#f", falls kein else-Zweig} \\ < e_{n+1} >, & \text{sonst} \end{cases}$$

Reduktion von cond [eval_{cond}]

$$(\text{cond } (< t_1 >< e_1 >)(< t_2 >< e_2 >)...) \rightsquigarrow \begin{cases} < e_1 >, & \text{falls t1' = \#f} \\ (cond(< t_2 >< e_2 >)(...)), & \text{sonst} \end{cases}$$

Reduziere t_1, erhalte t_1'.

(cond) $\rightsquigarrow$ Fehlermeldung "Alle Tests..."

(cond (else $< e_{n+1} >$)) $\rightsquigarrow$ e_{n+1}

cond ist "systematischer Zucker"

(auch: abgleitete Form) für eine verschachtelte Anwendung von 'if':

(cond $(< t_1 >< e_1 >)$ $(< t_1 >< e_1 >)$ ...))) entspricht (if $< t_1 >< e_1 >$ $(if < t_1 >< e_1 > (if...))$

Spezialformen 'and' und 'or':

$$(or < t_! >< t_2 > ... < t_n >) \text{ entspricht } (if < t_1 > \#t(or < t_2 > ...))$$

$$(\text{or}) \rightsquigarrow \#\text{f}$$

$$(and < t_1 > ... < t_n > \rightsquigarrow (if < t_1 > (and < t_2 > ... < t_n >)\#f)$$

$$(\text{and}) \rightsquigarrow \#t$$

Chapter 6

Zusammengesetzte Daten - 30.04.15

Ein Charakter besteht aus drei Komponenten.

- Name des Charakters (name)
- Handelt es sich um einen Jedi? (jedi?)
- Stärke der Macht (force)

→ Datendefinition für zusammengesetzte Daten.

Konkreter Charakter:

Name	"Luke Skywalker"
jedi?	#f
force	25

; Ein Charakter (character) besteht aus

; - Name (name)

; - Jedi-Status (jedi?)

; - Stärke der Macht (force)

(define-records-procedures charakter

make-character

character?

(character-name

character-jedi

character-force))

(make-character n j f) $\rightsquigarrow$ $<records>$ (konstruktion)

(character-name $<record>$ $\rightsquigarrow$ n (Komponentenzugriff)

(character-jedi? $<record>$) $\rightsquigarrow$ j (Komponentenzugriff)

(character-force $<record>$) $\rightsquigarrow$ f (Komponentenzugriff)

Zusammengesetzte Daten = Records in Scheme.

Record-Definition legt fest:

- Record-Signatur
- Konstruktor (Baut aus Komponenten einen Record)

- Prädikat (liegt Record vor?)
- Liste von Selektoren (lesen jewils eine Komponenten des Records)

(define-records-procedures $< t >$

make-$< t >$; Konstruktor

$< t >$?

($< t > - < comp_1 > ... < t > - < comp_n > 4$)) ; Liste der n-Seleketoren

Verträge des Konstruktors / der Selektoren für Record-Signatur $< t >$ mit n Komponenten namens $< comp_1 > ... < comp_n >$:

- (: make-$< t >$ ($< t_1 > ... < t_n > \rightarrow < t >$))
- (: $< t > - < comp_1 >$ ($< t > \rightarrow < t_1 >$))
- ...
- (: $< t > - < comp_n >$ ($< t > \rightarrow < t_n >$))

Es gilt für die Strings n, Booleans j und Integer f:

(character-name (make-character n j f)) = n

(analog für den Rest)

Interaktion von Funktionen (algebraische Eigenschaften).

Spezialform check-property:

(check-property

(for-all (($< id_1 > < sig_1 > ... < id_n > < sig_n >$))

$< e >$))

$< e >$ bezieht sich auf $< id_1 > ... < id_n >$.

Test erfolgreich, falls $< e >$ für bel. gewählte Bindungen für $< id_1 > ... < id_n >$ immer #t ergibt.

Interaktion von Selektor und Konstruktor:

(check-property

(for-all ((n string)

(j booleans)

(f integer))

(string=? (character-name (make-character n j f)) n)))

Beispiel: Die Summe zweier natürlicher Zahlen ist mindestens so groß wie jede dieser Zahlen: $\forall x_1, x_2 \in \mathbb{N} : x_1 + x_2 \geq max\ x_1, x_2$

(check-property

(for-all ((x_1 natural)

(x_2 natural))

($\geq$ (+ x_1 x_2) (max x_1 x_2))))

Konstruktion von Funktionen, die zusammengesetzte Daten konsumieren:

- Welche Record-Komponenten sind relevant für Funktionen?

  → Schablone:

  ; könnte Charakter e ein Sith-Lord sein?

  (: sith? (character → boolean))

  (define sith?

  (lambda (e) ... (character-jedi? c) ... (character-force c) ...))

Konstrukton von Funktionen, die zusammengesetzte Daten konstruieren:

- Der Konstruktor muss aufgerufen werden.

  → Schablone:

  (define

  (lambda (...)

  ... (make-< t > ...) ...))

Chapter 7

Fortsetzung zusammengesetzte Daten - 05.05.1

- lego-character
  - figure
  - character
    - Name
    - jedi?
    - force

Sei $< p >$ ein Prädikat mit Signatur $(< t > \rightarrow boolean)$. Eine Signatur der Form (predicate $< p >$) gilt für jeden Wert x der Signatur $< t >$ sofern $(< p >$ x$) \rightsquigarrow \#t$. Signatur (predicate $< p >$) ist somit spezifischer (restriktiver) als die Signatur $< t >$ selbst.
Einfürhung eines neuen Signaturnamens $< new - t >$ für die Signatur $< t >$:

(define $< new - t >$ (signatur $< t >$))

Bsp.:

(define farbe (signatur (one-of "Karo" "Herz" "Pik" "Kreuz")))

(define latitude (signature (predicate latitude?)))

Chapter 8

Gemische Daten - 07.05.15

Geocoding: Übersetzte eine Ortsangabe mittels des Google Maps Geocoding API (Application Programming Interface) in eine Position auf der Erdkugel.

(: geocoder (string → (mixed geocode geocode-error)))

Ein Geocode besteht aus:

- Adresse (address) (string)
- Ortsangabe (loc) (location)
- Nordostecke (northeast) (location)
- Südwestecke (southwest) (location)
- Typ (type) (string)
- Genauigkeit (accuracy) (string)

(: geocode-address (geocode → string)) ...
Ein geocode-error besteht aus:

- Fehlerart (level) (one-of "TCP" "HTTP" "JSON" "API")
- Fehlermeldung (message) (string)

Teachpack: geocoder.rkt
Gemische Daten Die Signatur

$$(\text{mixed} < t1 > ... < t_n >)$$

ist gültig für jeden Wert, der mindestens eine der Signatur $< t_1 > ... < t_n >$ erfüllt.
Beispiel: Datendefinition:
Eine Antwort des Geocoders ist entweder

- ein Geocode (geocode) oder
- eine Fehlermeldung (geocode-error)

Beispiel (eingebaute Funktion string → number):
(: string→number (string → (mixed number (onfe of #f))))
(string→number "42") ⇝ 42

(string→number "foo") ⇝ #f

Erinnerung:

Das Prädikat $< t >?$ einer Signatur $< t >$ unterscheidet Werte der Signatur $< t >$ von allen anderen Werten:

(: $< t >?$ (any → boolean))

Auch Prädikafür eingebaute Signaturen:

- number?
- complex?
- real?
- rational?
- integer?
- natural?
- string?
- boolean?

Prozeduren die gemische Daten der Signaturen $< t_1 > ... < t_n >$ konsumieren:

Konstruktionsanleitung:

(: $< f >$ ((mixed $< t_1 > ... < t_n >$ → ...))

(define $< f >$ (lambda (x) (cond (($< t_1 >?$ x) ...) ... ($< t_n >?$ x)...)))))

Mittels let lassen sich Werte an lokale Namen binden:

$$(\text{let } ((< id_1 >< e_1 >) \ ... \ (< id_n >< e_n >)) < e >)$$

Die Ausdrücke $< e_1 > ... < e_n >$ werden parallel ausgewertet → $< id_1 > ... < id_n >$ können in $< e >$ (und nur hier) verwendet werden. Der Wert let-Ausdruck ist der Wert von $< e >$.

Achtung: 'let' ist verfügbar ab Sprachebene "DMdA".

'let' ist syntaktischer Zucker.

(let $((< id_1 >< e_1 >) \ ... \ (< id_n >< e_n >)) < e >$)

↔

((lambda $(< id_1 > ... < id_n >) < e >) < e_1 > ... < e_n >$)