

Informatik II Skript

Steffen Lindner

May 1, 2015

Contents

1	Einf[Pleaseinsertintopreamble]hrung - 14.04.15	3
2	Ausdrücke, Defines, usw. - 16.04.2015	4
3	Signaturen, Testfälle - 21.04.15	5
4	Substitutionsmodell, Fallunterscheidung - 23.04.15	7
5	One-of Signatur - 28.04.15	9
6	Zusammengesetzte Daten, check-property - 30.04.15	10

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
$44-2$	<code>(- 44 2)</code>
$f(x,y)$	<code>(f x y)</code>
$\sqrt{81}$	<code>(sqrt 81)</code>
9^2	<code>(expt 9 2)</code>
$3!$	<code>(! 3)</code>

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) (Real)

`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

$$(\text{define } \langle id \rangle \langle e \rangle) \text{ } (\langle id \rangle \text{ Identifier, } \langle e \rangle \text{ Expression})$$

Erlaubt konsistente Wiederverwendung und dient der Selbstdokumentation von Programmen.

Achtung: Dies ist eine sogenannte Spezifikation und kein Ausdruck. Insbesondere besitzt diese Spezialform keinen Wert, sondern einen Effekt: Name $\langle id \rangle$ wird an den Wert von $\langle e \rangle$ 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.: $\text{euro} \rightarrow \text{us\$}$

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:

$$(\text{lambda } (\langle p1 \rangle \langle p2 \rangle \dots) \langle e \rangle), \langle e \rangle \text{ Rumpf}$$

$\langle e \rangle$ enthält Vorkommen der Parameter $\langle p1 \rangle, \langle p2 \rangle \dots$

$(\text{lambda } \dots)$ ist eine Spezialform. Wert der Lambda-Abstraktion ist $\# \langle procedure \rangle$

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

$$(\text{lambda } (\text{days}) (* \text{ days } (* 155 \text{ min-in-a-day}))) \rightsquigarrow (* 365 (* 155 \text{ min-in-a-day})) \rightsquigarrow 81468000$$

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 p_1, p_2, \dots, p_n als auch den Ergebniswert der Prozedur:

$$(< signaturp_1 > \dots < signaturp_n > - > < signatur - ergebnis >)$$

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 Ausdruck $< e_1 >$ aus und teste, ob der erhaltene Wert der Erwartung (= 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 $360^\circ/60$ pro Minute zurück ($360/60 * m$)
- Stundenzeiger legt $360^\circ/12$ pro Stunde zurück ($360/12 * h + 360/12 * m/60$)

Chapter 4

Substitutionsmodell, Fallunterscheidung - 23.04

Reduktionsregeln für Scheme (Fallunterscheidung je nach Ausdrucksart)

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

- Lambda-Abstraktion

$(\text{lambda } ()) \rightsquigarrow (\text{lambda } ())$ [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

$(+ \ 40 \ 2)$

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

$\text{eval}_{lit} (+)$

$\text{eval}_{lit} (40)$

$\text{eval}_{lit} (2)$ •

$(\text{position-minute-hand } 30)$

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

$\rightsquigarrow (* \ \text{degrees-per-minute } 30)$

$\rightsquigarrow (* \ \text{degrees-per-minute } 30)$

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

Bezeichnen $(\text{lambda } (x) \ (* \ x \ x))$ und $(\text{lambda } (r) \ (* \ r \ r))$ 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

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

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

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

Binäre Fallunterscheidung: if

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

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

Chapter 5

One-of Signatur - 28.04.15

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

$$(\text{one-of } \langle e_1 \rangle \dots \langle e_n \rangle)$$

Reduktion von if:

$$(\text{if } t_1 \ e_1 \ e_2) \rightsquigarrow \begin{cases} \langle e_1 \rangle, & \text{falls } t_1' = \#t ; e_2 \text{ wird niemals ausgewertet} \\ \langle e_2 \rangle, & \text{sonst; } e_1 \text{ wird niemals ausgewertet} \end{cases}$$

(1) Reduziere t_1 , erhalte t_1'

Spezialform Fallunterscheidung (conditional expression):

$$(\text{cond } (\langle t_1 \rangle \langle e_1 \rangle) \dots (\langle t_n \rangle \langle e_n \rangle) (\text{else } \langle e_{n+1} \rangle)) (\text{else optional})$$

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

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

Reduktion von cond [eval_{cond}]

$$(\text{cond } (\langle t_1 \rangle \langle e_1 \rangle) (\langle t_2 \rangle \langle e_2 \rangle) \dots) \rightsquigarrow \begin{cases} \langle e_1 \rangle, & \text{falls } t_1' = \#f \\ (\text{cond } (\langle t_2 \rangle \langle e_2 \rangle) \dots), & \text{sonst} \end{cases}$$

Reduziere t_1 , erhalte t_1' .

$(\text{cond }) \rightsquigarrow$ Fehlermeldung "Alle Tests..."

$(\text{cond } (\text{else } \langle e_{n+1} \rangle)) \rightsquigarrow e_{n+1}$

cond ist "systematischer Zucker"

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

$(\text{cond } (\langle t_1 \rangle \langle e_1 \rangle) (\langle t_1 \rangle \langle e_1 \rangle) \dots))$ entspricht $(\text{if } \langle t_1 \rangle \langle e_1 \rangle (\text{if } \langle t_1 \rangle \langle e_1 \rangle (\text{if } \dots)))$

Spezialformen 'and' und 'or':

$(\text{or } \langle t_1 \rangle \langle t_2 \rangle \dots \langle t_n \rangle)$ entspricht $(\text{if } \langle t_1 \rangle \#t (\text{or } \langle t_2 \rangle \dots))$

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

$(\text{and } \langle t_1 \rangle \dots \langle t_n \rangle) \rightsquigarrow (\text{if } \langle t_1 \rangle (\text{and } \langle t_2 \rangle \dots \langle t_n \rangle) \#f)$

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

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 jeweils eine Komponenten des Records)

(define-records-procedures $\langle t \rangle$

make- $\langle t \rangle$; Konstruktor

$\langle t \rangle$?

($\langle t \rangle$ - $\langle comp_1 \rangle$... $\langle t \rangle$ - $\langle comp_n \rangle$ 4)) ; Liste der n-Selektoren

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

- ($\text{make-}\langle t \rangle (\langle t_1 \rangle \dots \langle t_n \rangle \rightarrow \langle t \rangle$))
- ($\langle t \rangle$ - $\langle comp_1 \rangle (\langle t \rangle \rightarrow \langle t_1 \rangle$))
- ...
- ($\langle t \rangle$ - $\langle comp_n \rangle (\langle t \rangle \rightarrow \langle t_n \rangle$))

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 (($\langle id_1 \rangle$ $\langle sig_1 \rangle$... $\langle id_n \rangle$ $\langle sig_n \rangle$))

$\langle e \rangle$))

$\langle e \rangle$ bezieht sich auf $\langle id_1 \rangle$... $\langle id_n \rangle$.

Test erfolgreich, falls $\langle e \rangle$ für bel. gewählte Bindungen für $\langle id_1 \rangle$... $\langle id_n \rangle$ 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) ...))

Konstruktion von Funktionen, die zusammengesetzte Daten konstruieren:

- Der Konstruktor muss aufgerufen werden.

→ Schablone:

(define

(lambda (...)

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