Formale Aspekte der Programmierung

Einführung in die Programmierung

Johannes Brauer

2. März 2020

Syntax und Semantik einer Programmiersprache

- zur Wiederholung:

  Was bedeuten die Begriffe *Syntax* und *Semantik*?

(Vereinfachte) Syntax von Racket

```
<def> = (define <var> <exp>)
<exp> = <var>
        | <con>
        | (<prm> <exp> ... <exp>)
        | (<var> <exp> ... <exp>)
        | (cond [<exp> <exp>] ... [<exp> <exp>])
        | (cond [<exp> <exp>] ... [else <exp>])
        | (lambda [<var> <var> ... <var>] <exp>)
```

- Racket-Programme bestehen (bisher) aus Definitionen (`<def>`) und Ausdrücken (`<exp>`)
- Ausdrücke können sein:
  - Variablen (`<var>`) oder Konstanten (`<con>`)
  - Funktionsanwendungen (Funktionsaufrufe)
  - Bedingte Ausdrücke
  - Lambda-Ausdrücke

Semantik einer Programmiersprache

- Die Syntax beschreibt den Aufbau grammatikalisch korrekter Sätze
- Die Semantik beschreibt die Bedeutung grammatikalisch korrekter Sätze
- Die Semantik natürlich-sprachlicher Sätze wird häufig durch Sätze mit einfacheren (bereits bekannten) Begriffen beschrieben
- Die Semantik einer Programmiersprache
  - legt fest, welche Wirkung jedes Sprachelement oder -konstrukt im Programmablauf hervorruft
  - wird beschrieben durch
    - Sätze in natürlicher Sprache (häufig anzutreffen bei gängigen Programmiersprachen (z.B. C, Pascal, Java, Smalltalk); Problem: Beschreibung umfangreich und nicht eindeutig
    - Menge von Verhaltensregeln, die die Funktionsweise von Programmen bestimmen; kann formal und damit unzweideutig erfolgen

(Vereinfachte) Semantik von Racket

- Für die Beschreibung der Semantik von Racket greifen wir auf einfache Regeln der Algebra zurück.
- Die Semantik definiert wie, Racket-Ausdrücke ausgewertet werden.
- Auswertung bedeutet, dass ein Ausdruck durch Anwenden von Regeln solange umgeformt wird, bis ein Wert übrig bleibt.
- Werte sind (bisher):
  - Zahlen (2, 3.5, 1/3)
  - boolesche Werte (`#true`, `#false`)
  - Symbole (`'Karl`, `'hallo`)
  - Listen, bestehend aus Zahlen, booleschen Werten, Symbolen und Listen
  - Lambda-Ausdrücke
- Zahlen, booleschen Werte und Symbole sind Ausdrücke, die zu sich selbst ausgewertet werden (self evaluating expressions).

Semantik der Funktionsanwendung

- Dieses Thema wurde bereits in Kapitel EidP04-Auswertungsregeln behandelt.

Auswertungsregeln für Racket-Pseudofunktionen

- *Pseudofunktionen* (in der Lisp-Terminologie auch *special forms* genannt) heißen so, weil
  - für sie besondere Auswertungsregeln gelten und/oder
  - ihre Auswertung keinen Wert liefert.
- Bisher benutzte Pseudofunktionen:
  - `define`
  - `lambda`
  - `cond`
  - `else`

Auswertungsregeln für `define` und `lambda`

`define` Der Ausdruck `(define var exp)` liefert keinen Wert, sondern hat lediglich den Effekt, dass der Variablen `var` der Wert von `exp` in der globalen Umgebung zugeordnet wird.

`lambda` Der Ausdruck `(lambda [v1 ...vn] exp)` liefert als Wert eine (namenlose) Funktion mit den formalen Parametern `v1 ...vn` und der Berechnungsvorschrift `exp`.

Auswertungsregel für `cond` ohne `else` Zwei Fälle sind zu unterscheiden:

1. Die Auswertung der ersten Frage (Bedingung) liefert `#false`:

   ```
   (cond
     [#false ...]
     [frage2 antwort2]
     ....)
     = (cond
         [frage2 antwort2]
         ...)
   ```

   Das heißt, die erste Frage-Antwort-Kombination wird eliminiert.

2. Die Auswertung der ersten Frage (Bedingung) liefert `#true`:

```
(cond
  [#true exp]
  ...)
  = exp
```

Das heißt, der Wert des ganzen `cond`-Ausdrucks ist `exp`.

Es ist ein Fehler, wenn keine Frage `#true` liefert.

Auswertungsregel für `cond` mit `else`

- Die Auswertung erfolgt wie bei `cond` ohne `else`. Wenn dabei als letzte die `else`-Zeile übrig bleibt, gilt:

```
(cond
   [else exp])
= exp
```

Das heißt, der Wert des ganzen `cond`-Ausdrucks ist die Antwort hinter `else`.

Korrektheit von Funktionen

Frage:

> Was sagt uns die Mitteilung von DrRacket:
>
> Alle 4 Tests bestanden!
>
> ???
>
> Software wird zur Benutzung freigegeben, nicht wenn sie nachweislich korrekt ist, sondern wenn die Häufigkeit, mit der neue Fehler entdeckt werden, auf ein für die Geschäftsleitung akzeptables Niveau gesunken ist.
>
> DAVID L. PARNAS

- Voraussetzungen für den **Beweis** der Korrektheit einer Funktion:
  - Formalisierung der Spezifikation eines Programms
  - formale Definition der Semantik der Programmiersprache
- Funktionale (zustandslose) Programmierung:
  - Semantik definiert durch Substitutionsmodell.
  - Beweise durch vollständige Induktion über die Rekursionstiefe
- Nicht-funktionale, zustandsbehaftete Programmierung:
  - Beweisverfahren werden überaus komplex.

Korrektheit rekursiver Funktionen

Beweis mittels rekursiver Induktion:

1. Formuliere die Behauptung, die für den Wert der Funktion gelten muss.
2. Zeige, dass die Behauptung für einen Aufruf mit Rekursionstiefe 0 gilt.
3. Zeige, dass aus der Gültigkeit der Behauptung für Aufrufe mit Rekursionstiefe gleich n die Gültigkeit der Behauptung für Aufrufe mit Rekursionstiefe $n + 1$ folgt.

Definition der Rekursionstiefe

> Findet beim Ausführen des Aufrufs kein rekursiver Aufruf statt, dann ist die Rekursionstiefe 0, und sonst ist sie um 1 größer als die größte Rekursionstiefe aller weiteren Aufrufe, die durch diesen Aufruf verursacht werden.

Beispiel: Funktion `len`

```
;; berechnet die Laenge einer Liste
(define len
  (lambda [lst]
  (cond
    [(empty? lst) 0]
    [else (+ 1 (len (rest lst)))])))
```

Behauptung: Der Aufruf

`(len '` $(e_1\ e_2\ \ldots\ e_n)$ `)`

liefert die Anzahl n der Elemente der Argumentliste.

Verankerung: Der Aufruf `(len '())` liefert nach Ersetzungsmodell:

```
(len '())
= (cond
    [(empty? '()) 0]
    [else (+ 1 (len (rest '())))])
= (cond
     [true 0]
     [else (+ 1 (len (rest '())))])
= 0
```

Wenn die Rekursionstiefe $k = 0$ ist, liefert die Funktion 0, d.h. die Länge der leeren Liste.

Induktionsannahme: Die Behauptung gilt für Rekursionstiefe $k = m$.

Induktionsschluss Wir betrachten jetzt den Aufruf `(len '` $(e_1\ e_2\ \ldots\ e_{m+1})$ `)`

Dieser Aufruf wird gemäß Ersetzungmodell ausgewertet als
`(len '` $(e_1\ e_2\ \ldots\ e_{m+1})$ `)`
$=$ `(cond`
`[(empty? '` $(e_1\ e_2\ \ldots\ e_{m+1})$ `) 0]`
`[else (+ 1 (len (rest '` $(e_1\ e_2\ \ldots\ e_{m+1})$ `)))])`
$=$ `(cond`
`[false 0]`
`[else (+ 1 (len (rest '` $(e_1\ e_2\ \ldots\ e_{m+1})$ `)))])`
$=$ `(+ 1 (len (rest '` $(e_1\ e_2\ \ldots\ e_{m+1})$ `)))`
$=$ `(+ 1 (len '` $(e_2\ \ldots\ e_{m+1})$ `))`

Der Aufruf `(len '` $(e_2\ \ldots\ e_{m+1})$ `)` besitzt die Rekursionstiefe m und liefert gemäß Induktionsannahme m:
$=$ `(+ 1 (len '` $(e_2\ \ldots\ e_{m+1})$ `))`
$=$ `(+ 1 m)`
$= m + 1$

Lokale Definitionen

Die globale Umgebung

- Man beachte, dass Ausdrücke wie `(+ x 2)` oder `(f 7 8)` – jeweils isoliert betrachtet – nicht ausgewertet werden können, da nicht klar ist, wozu `x` bzw. welche Funktion `f` ausgewertet werden soll.
- Die Auswertung ist nur in einer *Umgebung* möglich, in der die Werte aller Variablen definiert sind.
- Durch die Pseudofunktion `define` wird einer Variablen ein Wert zugeordnet. Diese Assoziation merkt sich der Interpreter in einem Gedächtnis, das *globale Umgebung* genannt wird.
- Hinweis zur Benutzung von DrRacket: Bei jeder Betätigung der `Ausführen`-Schaltfläche wird zunächst die globale Umgebung gelöscht und anschließend werden die durch die im Definitionsfenster per `define` erzeugten Assoziationen in die globale Umgebung übernommen.

Lokale Umgebungen

Ineffiziente Funktion `sumprod`

```
;; berechnet eine zweielementige Liste, bestehend
;; aus der Summe und dem Produkt der
;; Elemente der Zahlenliste
;; sumprod: (list-of number) -> (list-of number)
(check-expect (sumprod empty) '(0 1))
(check-expect (sumprod '(3 4 5)) '(12 60))
(define sumprod
  (lambda [lon]
    (cond
      [(empty? lon) (list 0 1)]
      [else (list
             (+ (first lon)
                (first (sumprod (rest lon))))
             (* (first lon)
                (first (rest (sumprod
                              (rest lon))))))])))
```

Warum ist die Funktion ineffizient?

Verwendung einer *lokalen Definition*

```
;; berechnet ...
;; sumprod: (list-of number) -> (list-of number)
(check-expect (sumprod empty) '(0 1))
(check-expect (sumprod '(3 4 5)) '(12 60))
(define sumprod
  (lambda [lon]
    (cond
      [(empty? lon) (list 0 1)]
      [else (let [(sp (sumprod (rest lon)))]
                ;;
              (list
               (+ (first lon)
                  (first sp))
                      ;;
               (* (first lon)
                  (first (rest sp)))))])))
                              ;;
```

Lokale Definitionen dienen (u.a.) zur Benennung des Wertes eines Ausdrucks zum Zweck der Vermeidung von Mehrfachberechnungen.

Syntax und Semantik von `let`

- Die Syntax der Anwendung der Pseudofunktion `let`, ein so genannter `let`-Ausdruck, hat die folgende Form:
  `(let [` $(v_1\ e_1) \ldots (v_n\ e_n)$ `] exp)`
- Dabei sind
  - die v_i Variablenbezeichner,
  - die e_i Ausdrücke sowie
  - *exp*, der *Rumpf* des `let`-Ausdrucks, ebenfalls ein Ausdruck.
- Semantik informell: Der Wert eines `let`-Ausdrucks ist der Wert, der sich aus der Auswertung seines Rumpfes ergibt, wenn alle darin vorkommenden v_i durch die Werte der korrespondierenden e_i ersetzt werden.

- Für die formale Definition der Semantik genügt es hier festzuhalten, dass der Ausdruck
`(let [` $(v_1\ e_1)\ldots(v_n\ e_n)$ `] exp)`
durch den folgenden `lambda`-Ausdruck ersetzt werden kann:
`((lambda` $(v_1 \ldots v_n)\ exp) e_1 \ldots e_n$ `)`

- `let` ist syntaktischer Zucker.

Pragmatik von `let`

- Einen Anwendungszweck von `let` haben wir in der Funktion `sumprod` bereits gesehen: Benennung eines Zwischenergebnisses zur Vermeidung von Mehrfachberechnungen.

- Die Benennung von Werten kann ein Programm besser lesbar machen:

```
;; berechnet ...
;; sumprod: (list-of number) -> (list-of number)
(define sumprod
  (lambda [lon]
    (cond
    [(empty? lon) (list 0 1)]
    [else (let* [(sp (sumprod (rest lon)))
                 (restsumme (first sp))
                 (restprodukt (first (rest sp)))]
                (list
                  (+ (first lon) restsumme)
                  (* (first lon) restprodukt)))])))
```

Unterschied zwischen `let` und `let*`

- Die Variablen v_i in einem `let`-Ausdruck sind nur im Rumpf sichtbar aber nicht in den e_i.

- Um in der Funktion `sumprod` die Variablen `restsumme` und `restprodukt` unter Verwendung der Variablen `sp` zu definieren, müssten die let-Ausdrücke verschachtelt werden. Das wird leicht unübersichtlich.

- Deshalb gibt es mit `let*` weiteren syntaktischen Zucker: Hier sind in e_i alle v_j mit $j < i$ sichtbar.

- Formal gilt:
`(let* [` $(v_1\ e_1)\ldots(v_n\ e_n)$ `] exp)`
ist äquivalent zu `(let [` $(v_1\ e_1)$ `] (let* [` $(v_2\ e_2)\ldots$ `] exp))`

Lokale Hilfsfunktionen

- Wenn Hilfsfunktionen nur reinen Hilfscharakter haben, d.h. nur im Kontext der Hauptfunktion sinnvoll sind, kann es – schon zur Vermeidung von Namenskonflikten in der globalen Umgebung – zweckmäßig sein, Hilfsfunktionen lokal zu definieren.

- Betrachten wir dazu eine weitere Variante von `sumprod` mit zwei Hilfsfunktionen.

`sumprod` mit Hilfsfunktionen

```
;; berechnet ...
;; sumprod: (list-of number) -> (list-of number)
(define sumprod
  (lambda [lon] (list (sum lon) (prod lon))))

(define sum
  (lambda [lon]
    (cond
      [(empty? lon) 0]
      [else (+ (first lon) (sum (rest lon)))])))

(define prod
```

```
  (lambda [lon]
    (cond
      [(empty? lon) 1]
      [else (* (first lon) (prod (rest lon)))])))
```

`sumprod` mit lokalen Hilfsfunktionen

```
;; sumprod: (list-of number) -> (list-of number)
(define sumprod
  (lambda [lon]
    (letrec
     [(sum
       (lambda [lon]
         (cond [(empty? lon) 0]
           [else (+ (first lon)
                    (sum (rest lon)))])))
      (prod
       (lambda [lon]
         (cond [(empty? lon) 1]
           [else (* (first lon)
                  (prod (rest lon)))])))]
   (list (sum lon) (prod lon)))))
```

Bei der dritten `let`-Variante `letrec` sind alle v_i in allen e_i bekannt.