Funktionale und Objektorientierte Programmierung

—

Dieses Skript richtet sich nach der Vorlesung von
Prof. Dr. rer. nat. Karsten Weihe
Formulierungen sind teils von den Folien übernommen.

Technische Universität Darmstadt

9. November 2017

Inhaltsverzeichnis

1 Grundlagen der Programmierung

Was ist Progrmamieren?

Schauen wir zunächsteinmal, wie einige der „großen Köpfe“ der Informatik das Programmieren definieren.

> „To program is to understand“
> *Kristen Nygaard*

> „Programming is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas“
> *Marvin Minsky, Gerald J. Sussman*

1.1 Strukturierungsmechanismen

1.1.1 ... einer Programmiersprache

Eine Programmiersprache ist mehr als ein Hilfsmittel um einen Computer anzuweisen, Aufgaben durchzuführen. Sie dient auch als **Rahmen**, innerhalb dessen wir **unsere Ideen** über die **Problemdomäne organisieren.**

Wenn wir eine Sprache beschreiben, sollten wir die Hilfsmittel beachten, die sie uns zum Kombinieren von einfachen Ideen anbietet, um komplexere Ideen zu bilden.

Jede vollwertige Programmiersprache hat drei Mechanismen, um Prozessideen zu strukturieren:

- ***Primitive* Ausrücke**

  - Repräsentieren die einfachsten Einheiten der Sprache

  - Im Deutschen: jedes Wort ist ein primitiver Ausdruck

- ***Kombinationsmittel***

  - Zusammengesetzte Elemente werden aus einfacheren Einheiten konstruiert

  - Im Deutschen: Zusammensetzung mehrerer Wörter zu einem Satz.

- ***Abstraktionsmittel***

  - Zusammengesetzte Elemente können benannt und weiter als Einheiten manipuliert werden

  - Im Deutschen: Definition eines Begriffs („Ein Auto ist ...“), so dass der Begriff danach als „Kurzform“ für die Erklärung nutzbar ist

1.1.2 ... in der Elektronik

- ***Primitive* Ausrücke**

  - Widerstände, Kondensatoren, Induktivitäten, Spannungsquellen, ...

- ***Kombinationsmittel***

  - Richtlinien für das Verdrahten der Schaltkreise

  - Standardschnittstellen (z.B. Spannungen, Strömungen) zwischen den Elementen. Diese Schnittstellen können auch Anforderungen an konkrete zulässige Werte oder Einheiten stellen („5 mA“)

- ***Abstraktionsmittel***

  - “Black box” Abstraktion – denke über einen Unter-Schaltkreis als eine Einheit: z.B. Verstärker, Regler, Empfänger, Sender, ...

1.2 Sprachelemente

1.2.1 Die Primitiven

Zahlen

Zahlen sind selbstauswertend: Die Werte der Zifferfolge ist die Zahl, die, die sie bezeichnen.

$$23 \Rightarrow 23$$

$$-36 \Rightarrow -36$$

Boolesche Werte

Boolsche Werte können nur *wahr* oder *falsch* sein. Diese sind ebenfalls selbstauswertend. Sie werden als

$$True \text{ oder } False$$

bezeichnet. Prozeduren sind in der Programmierung auch als „Funktionen“ oder „Methoden“ bekannt. Beispiele für eingebaute Prozeduren sind

$$+,\ *,\ /,\ -,\ =,\ usw.$$

Aber was ist der Wert von so einem Ausdruck? Der Wert von + ist eine Prozedur, die Zahlen addiert. Dies werden wir später als „Higher-Order Procedures“ kennen lernen. Auswertung: Nachschlagen des dem Namen zugewiesenen Wertes.

1.2.2 Besonderheiten bei Zahlen

DrRacket rechnet immer genau, wenn das möglich ist. Ganze und endliche Zahlen berechnet er so wie es „üblich ist“. Brüche mit periodischem Ergebnis werden ebenfalls in

einem Text als Bruch - etwa `7/3` dargestellt. Im Programm werden sie jedoch als Periode angezeigt. Beispielsweise `- 2.3`. Doch kann nicht jede Zahl *exakt* dargestellt werden. Man wird hier durch die Verwendung des Binärsystems eingeschränkt. Es gilt:

> „je mehr Nachkommastellen die Zahl besitzt, desto ungenauer wird in der Regel ihre Darstellung im Binärsystem“

Zahlen unendlicher Länge ohne Periode werden wie folgt dargestellt: `#i „inexact“`
Hier einige Beispiele:

$$e : \texttt{\#i2.718281828459045}$$

$$pi : \texttt{\#i3.141592653589793}$$

$$\sqrt{2} : \texttt{\#1.4142135623730951}$$

Mit Brüchen oder „inexakten“ Zahlen kann normal gerechnet werden – das Ergebnis ist aber ggf. wieder „inexact“

$$(\sqrt{2})^2 = \texttt{\#i2.0000000000000004}$$

1.2.3 Kombination

Der Wert einer Kombination wird bestimmt durch die Ausführung der (durch den Operator) angegebenen Prozedur mit den Werten der Operanden. In Racket ist eine Sequenz von Ausdrücken eingeschlossen in Klammern. Die Ausdrücke sind primitiv oder erneut zusammengesetzt.

Hier ein Beispiel: Numerische Ausdrücke können mit Ausdrücken kombiniert werden, die primitive Prozeduren repräsentieren (z.B. + oder *), um einen zusammengesetzten Ausdruck zu erstellen. Kombinationen können verschachtelt werden. Dafür müssen sie die Regeln einfach Rekursiv anwenden.

```
1 (+ 4 (* 2 3)) = (4 + (2 * 3)) = 10
2 (* (+ 3 4) (- 8 2))
3 = ((3 + 4) * (8 - 2))
4 = 42
```

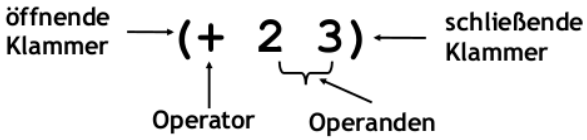


Abbildung 1:

WICHTIG: Eine Kombination bedeutet immer eine Anwendung einer Prozedur. Klammern können nicht eingefügt oder weggelassen werden, ohne die Bedeutung des Ausdrucks zu ändern.

1.2.4 Abstraktion

1.3 Formales

2 Strukturierte Datentypen

3 Rekursive Datentypen und Strukturelle Rekursion

3.1 Listen

Mit Strukturen können Datenobjekte mit einer festen Zahl von Daten gespeichert werden. Häufig wissen wir jedoch nicht, aus wie vielen Datenelementen eine Datenstruktur besteht. Oder die Struktur der Daten ist rekursiv. Mit rekursiven Datentypen können auch beliebig große Datenobjekte strukturiert abgespeichert werden. Die Idee davon ist die Folgende: Ein Element der Datenstruktur speichert (direkt oder indirekt) ein Exemplar der Datenstruktur. Dies nennt man dann eine *rekursive Datenstruktur*. Um eine endliche Datenstruktur zu bekommen benötigt man einen *Rekursionsanker*. Diesen Rekursionsanker modellieren wir mit der Technik zu heterogenen Daten aus dem letzten Kapitel.

Eine Liste ist entweder die leere Liste *the-emtpylst*, oder *(make-lst s r)*, wobei s ein Wert ist und r eine Liste. Im folgenden sehen wir eine Modellierung von Listen mit Struktur.

```
1 ;; a list with 0 elements
2 ;; (define list0 the-emptylst)
3 (define list0 empty)
4
5 ;; a list with 1 element
6 ;; (define list1 (make-lst 'a the-emptylst))
7 (define list1 (cons 'a empty))
```

```
8
9  ;; a list with 2 elements
10 ;; (define list2 (make-lst 'a
11 ;;                (make-lst 'b the-emptylst)))
12 (define list2 (cons 'a (cons 'b empty)))
13
14 ;; get the 2nd element from list2
15 ;; (lst-first (lst-rest list2)) -> 'b
16 (first (rest list2)) ;; -> 'b
```

Listen sind ein wichtiger Datentyp, weshalb es einen eingebauten Datentyp existiert. Der Konstruktor *cons* besitzt zwei argumente. cons entspricht unserem *make-lst* Eigenbeispiel und steht wie es leicht zu vermuten ist für *cons*truktor. Zudem existiert eine leere Liste *emtpy* die unserer leeren Liste *the-emptylst* entspricht. Auf die Sektoren der Liste kann man mit *first* und *rest* zugreifen. Mit greift man auf das erste Element und mit *rest* auf den Rest der Liste zu. Zudem haben beide noch *historische Namen* die da lauten *car* für *first* und *cdr* für rest. Die Prädikate *lst?* und *emtpy?* entsprechen *lst?* und *emptylst?*. *lst?* checkt ob es eine Liste ist und *emptylst?* ob die Liste leer ist. Im Folgenden nun ein Beispiel:

```
1  ; a list with 0 elements
2  ;; (define list0 the-emptylst)
3  (define list0 empty)
4
5  ;; a list with 1 element
6  ;; (define list1 (make-lst 'a the-emptylst))
7  (define list1 (cons 'a empty))
8
9  ;; a list with 2 elements
10 ;; (define list2 (make-lst 'a
11 ;;                (make-lst 'b the-emptylst)))
12 (define list2 (cons 'a (cons 'b empty)))
13
14 ;; get the 2nd element from list2
```

```
15 ;; (lst-first (lst-rest list2)) -> 'b
16 (first (rest list2)) ;; -> 'b
```

Wie sie sehen besteht der einzige Unterschied zwischen *make-lst* und *cons* darin, dass *cons* als zweites argument *empty* oder *(cons ...)*. Zum Beispiel :

```
1 (cons 1 2)
```

ist ein Fehler

```
2 (make-lst 1 2)
```

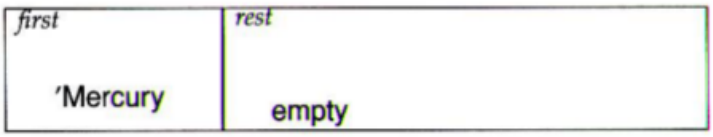
aber nicht.
cons verhindert also inkorrekte Nutzung der Prozedur. In anderen LISP-basierten Dialekten fehlt allerdings diese Überprüfung.

Eine bessere Emulation sähe wie folgt aus:

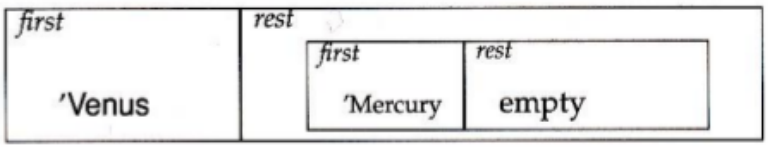
```
1 (define-struct lst (first rest))
2 (define-struct emptylst ())
3 (define the-emptylst (make-emptylst))
4 (define (our-cons a-value a-list)
5   (cond
6     [(emptylst? a-list) (make-lst a-value a-list)]
7     [(lst? a-list) (make-lst a-value a-list)]
8     [else (error 'our-cons "list as second argument expected")]))
```

Dies kann aber nicht verhindern, dass man *make-lst* direkt verwendet. Im folgenden noch einige visuelle Beispiele.

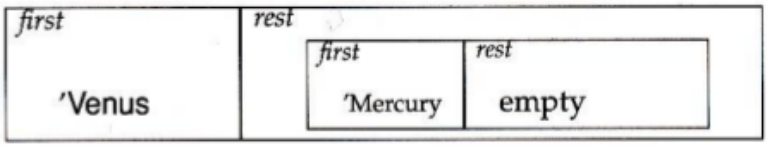
```
1 (cons 'Mercury empty)
```



```
(cons 'Venus
      (cons 'Mercury empty))
```




```
(cons 'Earth
  (cons 'Venus
        (cons 'Mercury empty)))
```



In den folgenden Beispielen sehen wir: Listen speichern beliebige Datentypen, auch gemischte Daten.

```
(cons 0
  (cons 1
    (cons 2
      (cons 3
        (cons 4
          (cons 5
            (cons 6
              (cons 7
                (cons 8
                  (cons 9 empty)))))))))))
```

```
(cons 'RobbyRound
  (cons 3
    (cons true empty)))
```

3.2 Abgeschlossenheitseigenschaft

Eine Operation zum Kombinieren von Daten besitzt die Abgeschlossenheitseigenschaft, wenn die Ergebnisse der Kombination von Datenobjekten wiederum mit der Operation kombiniert werden können. Betrachten wir hier *cons* als Beispiel.

4 Auswertungsreihenfolge und Lexikalisches Scoping

4.1 Intermezzo: Syntax und Semantik der HtDP-TL

4.1.1 Syntax

Ähnlich wie die natürlichen Sprachen haben auch die Programmiersprachen ein Vokabular und eine Grammatik. Das Vokabular ist eine Sammlung der "Wörter", aus denen wir Sätzeïn unserer Sprache bilden können. Ein Satz in einer Programmiersprache ist ein Ausdruck oder eine Funktion. Die Grammatik der Sprache sagt uns, wie wir ganze Sätze aus Wörtern bilden. Der Ausdruck Syntax bezieht sich auf Vokabular und Grammatik von Programmiersprachen.

4.1.2 Semantik

Nicht alle grammatikalisch richtigen Sätze sind sinnvoll, weder in Deutsch noch in Programmiersprachen. "Die Katze ist schwarzïst ein sinnvoller Satz. "Der Ziegel ist ein Auto"macht wenig Sinn, auch wenn der Satz grammatikalisch richtig ist.

Um herauszufinden ob ein Satz sinnvoll ist, müssen wir die Bedeutung (Semantik) der Wörter und Sätze verstehen. Für Programmiersprachen gibt es verschiedene Wege,

um den Sinn von einzelnen Sätze/Ausdrücken zu erklären. Den Sinn von HtDP-TL-Programmen diskutieren wir mit einer Erweiterung der bekannten Gesetze aus Arithmetik und Algebra (Substitutionsmodell).

Es existieren vier Kategorien von Wörtern und jedes Wort ist durch eine Zeile definiert.

- Variablen <var>: Namen von Werten
- Funktionen <fct>: Namen von Funktionen
- Konstanten <con>: boolean, Symbole, numerische Konstanten
- Primitive Operationen <prm>: Die Grundfunktionen, die HtDP-TL von Anfang an zur Verfügung stellt

Die Notation geschieht wie folgt: Zeilen zählen einfache Beispiele auf, getrennt durch ein „|“ und Punkte bedeuten, dass es noch mehr Dinge derselben Art in der Kategorie gibt

```
<var> = x | a | alon | ...
<fct> = area-of-disk | perimeter | ...
<con> = true | false | 'a | 'doll | 'sum | ... | 1 | -1 | 3/5 / 1.22 | ...
<prm> = + | - | * | ...
```