



Institut für Informatik  
Prof. Dr. Peter Thiemann  
Jochen Walter

Georges-Köhler-Allee 79  
D-79110 Freiburg i. Br.

Freiburg, den 21. Dezember 2000

## Informatik 1, WiSe 2000/2001

### Übungsblatt 7

Die Aufgaben werden in den Übungs- und Programmiergruppen  
vom 8.12. bis zum 14.12. besprochen.

Die Lösungen müssen nicht abgegeben werden!

*Die Aufgaben auf diesem Blatt können in Teams bearbeitet werden. Bevor drscheme gestartet werden kann, muß setup lang eingegeben werden. Der Sprachumfang sollte auf „Full Scheme“ eingestellt werden. Die Aufgaben sind mit einem bis drei Sternen versehen, wobei die Aufgaben mit einem Stern am einfachsten, die mit dreien am schwierigsten sind.*

#### Aufgabe 1 (\*\*):

Geben Sie eine Grammatik für die in der Vorlesung vorgestellten Formeln der Aussagenlogik an. Die Menge der primitiven Aussagen seien  $\{a, b, \dots, z\}$ .

#### Lösung zu Aufgabe 1:

$$\begin{aligned} N &= \{B\} \\ \Sigma &= \{(\,), 0, L, \wedge, \vee, \neg, a, b, \dots, z\} \\ P &= \{ B \rightarrow 0, \\ &\quad B \rightarrow L, \\ &\quad B \rightarrow a \\ &\quad \vdots \\ &\quad B \rightarrow z \\ &\quad B \rightarrow (B \wedge B), \\ &\quad B \rightarrow (B \vee B), \\ &\quad B \rightarrow (\neg B). \end{aligned}$$

Startsymbol ist  $B$ .

#### Aufgabe 2 (\*\*):

Gegeben sei die Grammatik  $\mathcal{G} = (N, \Sigma, P, S)$  mit

$$\begin{aligned} N &= \{S\} \\ \Sigma &= \{a, b\} \\ P &= \{S \rightarrow \varepsilon, S \rightarrow aSb, S \rightarrow SS\}. \end{aligned}$$

Beweisen Sie, daß das Wort  $aabbaababb$  in  $\mathcal{L}(G)$  liegt, indem Sie es aus dem Startsymbol  $S$  erzeugen. Listen Sie dazu alle Satzformen auf, aus denen die Ableitung besteht.

**Lösung zu Aufgabe 2:**

$S, SS, aSbS, aaSbbS, aabbS, abbaSb, abbaSSb, aabbaaSbSb, abbaabSb, abbaabaSbb, abbaababb.$

**Aufgabe 3 (\*\*\*):**

Gegeben seien das Operationsalphabet  $\Omega = \{\underline{0}, \underline{\text{succ}}, \underline{\text{pred}}, \underline{+}, \underline{-}, \underline{*}, \underline{\text{div}}, \underline{\text{mod}}\}$  und eine Stelligkeitsfunktion  $\sigma : \Omega \rightarrow \mathbb{N}$  mit

$$\begin{aligned}\sigma(\underline{0}) &= 0 \\ \sigma(\underline{\text{succ}}) &= 1 \\ \sigma(\underline{\text{pred}}) &= 1 \\ \sigma(\underline{+}) &= 2 \\ \sigma(\underline{-}) &= 2 \\ \sigma(\underline{*}) &= 2 \\ \sigma(\underline{\text{div}}) &= 2 \\ \sigma(\underline{\text{mod}}) &= 2.\end{aligned}$$

Geben Sie eine  $\Omega$ -Algebra  $\mathcal{A}$  von  $(\Omega; \sigma)$  an, indem Sie eine Menge  $A$  (die Trägermenge) und für jedes Operationssymbol  $F \in \Omega^{(n)}$  eine Funktion

$$F_{\mathcal{A}} : A^n \rightarrow A$$

festlegen.

**Lösung zu Aufgabe 3:**

Die naheliegendste  $\Omega$ -Algebra sieht so aus: Als Trägermenge wählen wir die ganzen Zahlen  $A = \mathbb{Z}$ . Dem Operationssymbol  $\underline{0}$  weisen wir die ganze Zahl  $0 : \rightarrow \mathbb{Z}$  zu, die man als nullstellige Funktion auffassen kann.

Den Operationssymbolen  $\underline{\text{succ}}$  und  $\underline{\text{pred}}$  weisen wir die Nachfolgerfunktion  $' : \mathbb{Z} \rightarrow \mathbb{Z}$  bzw. Vorgängerfunktion  $\text{pred} : \mathbb{Z} \rightarrow \mathbb{Z}$  zu.

Den Operationssymbolen  $\underline{+}$ ,  $\underline{-}$ ,  $\underline{*}$ ,  $\underline{\text{div}}$  und  $\underline{\text{mod}}$  weisen wir die Funktionen

$$\begin{aligned}+ : \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{Z}, \\ - : \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{Z}, \\ * : \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{Z}, \\ \text{div} : \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{Z}, \\ \text{mod} : \mathbb{Z} \times \mathbb{Z} &\rightarrow \mathbb{Z}\end{aligned}$$

zu.

Dabei ist  $+$ ,  $-$  und  $*$  die gewöhnliche Addition, Subtraktion und Multiplikation auf den ganzen Zahlen,  $\text{div}$  ist die Ganzzahldivision, und  $\text{mod}$  ist die Modulusfunktion, wobei wir festlegen daß  $m \text{ div } 0 = 0$  und  $m \text{ mod } 0 = m$  ist.

**Aufgabe 4 (\*\*):**

Formulieren Sie das Prinzip der Termination („strukturelle Induktion“) ähnlich wie bei der Wortinduktion aus Abschnitt B2.11 der Vorlesung.

#### Lösung zu Aufgabe 4:

Die Wortinduktion wurde in Teil B2.11 der Vorlesung angegeben:

Falls

- eine Aussage  $P$  für die leere Zeichenkette  $\varepsilon$  gilt und
- für alle Zeichen  $a \in \Sigma$  aus der Gültigkeit von  $P$  für alle Zeichenketten  $w \in \Sigma^*$  die Gültigkeit für die Zeichenketten  $aw$  folgt,

dann gilt die Aussage für alle  $w \in \Sigma^*$ .

In formaler Schreibweise bedeutet das

$$\frac{P(\varepsilon) \quad (\forall w \in \Sigma^*)(\forall a \in \Sigma)P(w) \Rightarrow P(aw)}{(\forall w \in \Sigma^*)P(w)}$$

Wir nennen nun die den betrachteten Termen zugrundeliegende Signatur  $\Sigma$ . Die strukturelle Induktion sieht dann in formaler Schreibweise wie folgt aus:

$$\frac{(\forall c \in \Sigma^{(0)})P(c) \quad (\forall x \in X)P(x) \quad (\forall n \in \mathbb{N})(\forall f \in \Sigma^{(n)})(\forall (t_1, \dots, t_n) \in T_\Sigma^n(X))P(t_1) \wedge \dots \wedge P(t_n) \Rightarrow P(ft_1 \dots t_n)}{(\forall t \in T_\Sigma(X))P(t)}$$

In Worten bedeutet diese Regel: Falls

- eine Aussage  $P$  für alle atomaren Terme (Variablen und nullstellige Operationssymbole) wahr ist und
- für alle  $n$  und für jedes  $n$ -stellige Operationssymbol  $f$  gilt, daß aus der Gültigkeit der Aussage  $P$  für die Komponenten jedes  $n$ -Tupels  $(t_1, \dots, t_n)$  die Gültigkeit von  $P$  für  $ft_1 \dots t_n$  folgt,

dann gilt die Aussage  $P$  für jeden Term aus  $T_\Sigma(X)$ .

#### Aufgabe 5 (Programmierung, \*):

Schreiben Sie eine Scheme-Funktion `list-map1` mit Hilfe der Funktion `list-fold`. (Rufen Sie nicht `list-map1` rekursiv auf!)

#### Lösung zu Aufgabe 5:

```
(load "constr-08.ss")

;;; SIGNATUR
;;; list-map1 : (X -> Y) list(X) -> list(Y)
;;; ERKLÄRUNG
;;; (list-map1 f l) wendet f auf jedes Element von l an.
;;; BEISPIEL
;;; (list-map1 - '()) == ()
;;; (list-map1 - (list 2 -4 3)) == (list -2 4 -3)
;;; (list-map1 / (list 8 1 0.5)) == (list 0.125 1 2)
;;; DEFINITION
```

```
(define list-map1
  (lambda (f l)
    (list-fold
      (lambda (a b) (cons (f a) b))
      '()
      l)))
```

### Aufgabe 6 (Programmierung, \*\*):

Schreiben Sie eine Funktion `btree-fold` analog zu `list-fold` mit der Signatur  $(Y \times Y \rightarrow Y) \ Y \ \text{btree}(X) \rightarrow Y$ . Der Aufruf `(btree c e t)` „ersetzt“ jedes Vorkommen von `'()` in `t` durch `e` und jedes Vorkommen von `make-branch` durch `c` und wertet den entstehenden Ausdruck aus.

Schreiben Sie mit Hilfe dieser Funktion eine Funktion `btree-map` mit der Signatur  $(X \rightarrow Y) \ \text{btree}(X) \rightarrow \text{btree}(Y)$ , die für einen Binärbaum denjenigen Binärbaum zurückliefert, der entsteht, wenn die Funktion auf jedes Element des Baumes angewendet.

Schreiben Sie mit Hilfe von `btree-fold` eine weitere Funktion `btree-flip` mit der Signatur  $\text{btree}(X) \rightarrow \text{btree}(X)$ , die zu einem Binärbaum denjenigen Binärbaum zurückliefert, der entsteht, wenn man in jedem Knoten den linken und rechten Teilbaum vertauscht.

Verwenden Sie die bekannte Definition von `btree`.

### Lösung zu Aufgabe 6:

; Das erlaubt es, `constr-06.ss` im ‘‘Full Scheme’’-level zu laden:

```
(define empty '())
```

```
(load "constr-06.ss")
```

```
;;; SIGNATUR
```

```
;;; btree-fold : (Y X Y -> Y) Y btree(X) -> Y
```

```
;;; ERKLÄRUNG
```

```
;;; (btree-fold c e t) "ersetzt" jedes Vorkommen von '() in t durch e
```

```
;;; und jedes Vorkommen von make-branch durch c und wertet den entstehenden
```

```
;;; Ausdruck aus.
```

```
;;; BEISPIEL
```

```
;;; (btree-fold * 1 '()) == 1
```

```
;;; (btree-fold * 1 (make-branch '() 2 (make-branch '() 3 '()))) == 6
```

```
;;; (btree-fold (lambda (x y z) (append x (list y) z)) '()
```

```
;;; (make-branch '() 2 (make-branch '() 3 '()))) == '(2 3)
```

```
;;; (btree-fold (lambda (x y z) (+ x z 1)) 0
```

```
;;; (make-branch '() 2 (make-branch '() 3 '()))) == 2 ^= btree-size
```

```
;;; DEFINITION:
```

```
(define btree-fold
```

```
  (lambda (c e t)
```

```
    (cond
```

```

((null? t)
 e)
(branch? t)
(c (btree-fold c e (branch-left t))
 (branch-elem t)
 (btree-fold c e (branch-right t))))))

;;; SIGNATUR
;;; btree-map: (X -> Y) btree(X) -> btree(Y)
;;; ERKLÄRUNG
;;; (btree-map c t) "ersetzt" jedes Element x in t durch (c x).
;;; BEISPIEL
;;; (btree-map - '()) == '()
;;; (btree-map - (make-branch '() 2 (make-branch '() 3 '())))
;;; == (make-branch empty -2 (make-branch empty -3 empty))
;;; DEFINITION:
(define btree-map
  (lambda (c t)
    (btree-fold
     (lambda (l e r) (make-branch l (c e) r))
     '()
     t)))

;;; SIGNATUR
;;; btree-flip : btree(X) -> btree(Y)
;;; ERKLÄRUNG
;;; (btree-fold c e t) vertauscht in jedem Knoten den linken mit dem
;;; rechten Teilbaum.
;;; BEISPIEL
;;; (btree-flip '()) == '()
;;; (btree-flip (make-branch '() 2 (make-branch '() 3 '())))
;;; == (make-branch (make-branch empty 3 empty) 2 empty)
;;; DEFINITION:
(define btree-flip
  (lambda (t)
    (btree-fold
     (lambda (l e r) (make-branch r e l))
     '()
     t)))

```