Klausur Gdp I Wintersem. 05/06 (24.02.2006)

Diese Lösung erstellt am: 26.03.2006 (also genau einen Tag vor der Nachklausur, die ich allerdings auch schreiben muss)
Erstellt von: dune (generationnext@spray.se)
Viel Erfolg, an alle, die sie sonst noch schreiben müssen.

Aufgabe 1: (2+3+3+3 Punkte)

Beweisen oder widerlegen Sie: Die folgende Funktionen f, f', f'': A*x A* → {0,1}, f''':A*→{0,1} sind berechenbar:

a) $f(P,w)=\begin{cases} 1, \text{ falls P ein Programm ist, das auf die Eingabe w terminiert} \\ 0, \text{ sonst} \end{cases}$

b) $f'(P,w)=\begin{cases} 1, \text{ falls P ein Programm ist, das auf die Eingabe w terminiert} \\ \perp, \text{ sonst} \end{cases}$

c) $f''(P,w)=\begin{cases} 1, \text{ falls P ein Programm ist, das auf die Eingabe w nicht terminiert} \\ \perp, \text{ sonst} \end{cases}$

d) Sei Q ein beliebiges festes Programm.

$f'''(w)=\begin{cases} 1, \text{ falls Q für alle Eingaben terminiert} \\ 0, \text{ sonst} \end{cases}$

Lösung: wie bei (http://www.xiran.de/uni/Klausurnachbereitung/Klausur_Gdp_I_Wintersemester05_06.pdf)

ergänzend:

$f'''(w))= \begin{cases} 1, \text{ falls Q für alle Eingaben terminiert} \\ 0, \text{ sonst} \end{cases}$

berechenbar, da Q ein konstantes Programm ist, d.h. von dem Eingabeparameter (hier w) nicht abhängt !

Regel: wenn eine Funktion berechenbar, dann gibt es zu ihr immer ein zugehöriges Programm / bzw. einen Algorithmus.

Aufgabe 2: (2+4 Punkte)
a) Geben Sie alle Bezeichner an, die in dem folgenden Ausdruck frei vorkommen:

funktion f x:Δ y:Δ' -> Δ'' ≡ h(f x) (g y) x

b) Seien nachf, vorg und mult gegebene Funktionen, die im Bereich der ganzen Zahlen den Nachfolger, Vorgänger bzw. die Multiplikation realisieren. Gegeben sei die folgende Deklaration:

funktion p x:Δ y:Δ' f: Δ'' g :Δ''' h: Δ'''' -> Δ''''' ≡ h(f x) (g y)

- Welchen Wert liefert der Aufruf:
  p 3 4 vorg nachf mult

- Bestimmen Sie den Typ von p

Lösung:
a) freie Bezeichner: **g**,**h**
unter Umständen auch **f**, da f nicht definiert ist, weil *funktion f* von 2 Parameter abhängt (x,y) und unser f x nur von einem Parameter: x.
Man könnte daher genauso die Funktion in der folgenden Form aufschreiben:
funktion f x:Δ y:Δ' -> Δ'' ≡ h(k x) (g y) x
(also an der Stelle von „f" ein „k" f somit nicht gebunden !!!)

b)
f= vorg
g= nachf
h= mult
x=3
y=4
f 3 =2; g 4=5; h 2 5=10 (Es wird der Wert 10 zurückgegeben!)

Typ von p:

$[\Delta, \Delta^1,[\Delta \rightarrow A], [\Delta' \rightarrow B], [A,B \rightarrow \Delta^5]] \rightarrow \Delta^5$

oder in gecurrter Form

Δ=Delta
$\Delta \rightarrow (\Delta^1 \rightarrow ([\Delta \rightarrow \Delta^6] \rightarrow ([\Delta^1 \rightarrow \Delta^7] \rightarrow ([\Delta^6 \rightarrow (\Delta^7 \rightarrow \Delta^5)] \rightarrow \Delta^5))))$

Aufgabe 3: (je korrekte Antwort 0,5 Punkte, je falsche Antwort –0,5 Punkte, mind. Aber 0 Punkte)

Welche der folgenden Aussagen sind korrekt? Bitte kreuzen Sie an.

Zu jedem Programm gibt es
- mindestens einen zugehörigen Prozess, den das Programm kontrolliert.
- höchstens einen zugehörigen Prozess, den das Programm kontrolliert.
- genau einen zugehörigen Prozess, den das Programm kontrolliert. (f)
- beliebig viele zugehörige Prozesse, die das Programm kontrolliert.

Zu einem Zeitpunkt t
- kann es zu einem Programm einen zugehörigen Prozess geben, den das Programm kontrolliert
- muss es zu einem Programm einen zugehörigen Prozess geben, den das Programm kontrolliert
- kann es zu einem Programm beliebig viele zugehörige Prozesse geben, die das Programm kontrollieren.
- muss es zu einem Programm beliebig viele zugehörige Prozesse geben, die das Programm kontrollieren.

Zu jedem Prozess gibt es
- mindestens ein zugehöriges Programm, das den Prozess kontrolliert.
- höchstens ein zugehöriges Programm, das den Prozess kontrolliert.
- genau ein zugehöriges Programm, das den Prozess kontrolliert.
- beliebig viele zugehörige Programme, die den Prozess kontrolliert.

Lösung:
Korrekte Antwort jeweils mit einem gefüllten Punkt versehen.

Aufgabe 4: (3+3 Punkte)

Sei A ein endliches Alphabet. Schreiben Sie ein Programm, das an genau drei Stellen aus seinem Eingabebereich A* undefiniert ist. Geben Sie zuvor eine funktionale Spezifikation Ihres Programms an.

Lösung:
Funktion in PRO:

$A=\{a_1,a_2,\ldots an\}$
typ Eingabe = A*

```
funktion undef3 (x: Eingabe);
solange x=a1 oder x=a2 oder x=a1a2 tue
    zeige (x)
ende
zeige(x);
ergebnis(x)
ende.
```

Spezifikation:

undef3: $A^* \rightarrow A^*$ with
undef3(x) = y
pre: $x=(x_{,1},x_{,2},\ldots xn)$
post: $y=x$ wenn $x\neq a_1$ v $x\neq a_2$ v $x\neq a_1a_2$

Aufgabe 5: (8 Punkte)

Skizzieren Sie ein Übersetzungsschema von PRO nach ASS für die Komposition von Funktionen.
Nehmen Sie an, Sie hätten bereits für zwei berechenbare Funktionen f,f' : N→N Funktionsprozeduren in PRO geschrieben und nach ASS übersetzt. Wie lautet das ASS-Programm für f°f? Machen Sie ggf. geeignete Annahmen für Ihre beiden ASS-Programme.

Lösung:

komp (f,g) = f°g

f: N→N c(f) – ASS-Übersetzung von f
f':N→N c(f') – ASS-Übersetzung von f'

f°f' = f(f')

Komposition in PRO:
X←f(f'(y))

ASS-Programm // mit Annahmen
get
c(f) // Ausgabe in SPR gespeichert
c(f') // Eingabe von SPR erhalten

Aufgabe 6: (4+4+4 Punkte)

Schreiben Sie jeweils eine Funktion in FUN oder ML, die
a) eine Zahlenfolge in Form einer Linkssequenz und eine Zahl n als Parameter besitzt und das n-te Element der Zahlenfolge als Ergebnis liefert.
b) eine Zahlenfolge in Form einer Linkssequenz als Parameter besitzt und das letzte Element der Zahlenfolge liefert,
c) zwei Zahlenfolgen in Form von Linkssequenzen als Parameter besitzt und die Zahl 1 liefert, falls die beiden Folgen gleich sind, und die Zahl 0 sonst.

Geben Sie jeweils die mathematische Funktion an, die Ihr Programm berechnet.

Lösung:

typ zf = {eps} I (int, zf)

a) *Ann*: x>0 (nat≥1)

function suche a:zf x:nat → int ≡
    wenn a={eps} dann error sonst
    wenn x=1 dann erstes(a) sonst suche(rest(a),x-1) ende

suche D*xD→D mit

$$\text{suche}(a,x)= \begin{cases} y & \text{wobei } a=[a_1,a_2,\ldots,a_n],\ x=[1,2,\ldots,n],\ n\geq 1,\ y=a_x \\ \bot & \text{sonst} \end{cases}$$

b)

function letztes a:zf → int ≡
    wenn a={eps} dann error sonst
    wenn rest(a)={eps} dann erstes(a) sonst letztes(rest(a)) ende

letztes D*→D mit

$$\text{leztes}(a)= \begin{cases} y & \text{wobei } a=[a_1,a_2,\ldots,a_n],\ x=[1,2,\ldots,n],\ n\geq 1,\ y=a_n \\ \bot & \text{falls } a=[] \end{cases}$$

c)

function gleich a:zf b:zf → bool ≡
    wenn a={eps} und a={eps} dann 1 sonst
    wenn a={eps} oder b={eps} dann 0 sonst
    wenn erstes(a) = erstes(b) dann gleich(rest(a), rest(b)) sonst 0 ende

gleich D*xD*→bool mit

$$\text{gleich}(a,b)= \begin{cases} 1 & \text{wobei } a=[a_1,a_2,\ldots,a_n],\ b=[b_1,b_2,\ldots,b_n],\ n\geq 0,\ a_i=a_i \\ 0 & \text{sonst} \end{cases}$$

Aufgabe 7: (3+1+3+5 Punkte)

a) Definieren Sie einen Datentyp zur Darstellung natürlicher Zahlen durch die „Bierdeckelnotation“. (Zur Erinnerung: In der Bierdeckelnotation ist die Zahl n durch eine Folge von (n div 5) Fünfergruppen bestehend aus 4 senkrechten Strichen und einen Querstrich sowie weiteren n mod 5 senkrechten Strichen dargestellt: 12 = ~~IIII~~ ~~IIII~~ II

b) Wie lautet in Ihrem Datentyp die Repräsentation der Zahlen 0, 1, 5, 12 ?

c) Geben Sie zwei Konvertierungsfunktionen in funktionaler Programmierung an, wovon die eine eine natürliche Zahl in die Bierdeckelnotation umwandelt und die andere eine Zahl in Bierdeckelnotation in eine natürliche Zahl umwandelt.

d) Schreiben Sie zwei Funktionen add und sub in funktionaler Programmierung, die (ohne Rückgriff auf Konvertierung gem. c)) zwei Zahlen in Bierdeckelnotation addieren bzw. subtrahieren. (Die Subtraktion sei so definiert: sub x y = x – y, falls x ≥ y, und 0 sonst.))

Lösung

a)

typ bd ≡ {ε, I, II, III, IIII} I (~~IIII~~, bd)

b)

0 = ε
1 = I
5 = (~~IIII~~,ε)
12 = (~~IIII~~,(~~IIII~~,II))

c)

```
function: inbd a:nat →bd ≡
        wenn a=0 dann ε sonst
                wenn a≥5 dann (IIII, inbd(a-5))sonst
                        wenn a=1 dann I     sonst
                        wenn a=2 dann II    sonst
                        wenn a=3 dann III   sonst
                        wenn a=4 dann IIII
funkende

function: innat a:bd →nat ≡
        wenn a=ε dann 0     sonst
        wenn a=1 dann I     sonst
        wenn a=2 dann II    sonst
        wenn a=3 dann III   sonst
        wenn a=4 dann IIII  sonst
        5 + innat(rest(a))
funkende
```

d)

```
function: add a:bd b:bd →bd ≡
        wenn a=ε dann b sonst
                wenn erstes(a)=IIII dann IIII add(rest(a), b) sonst
                wenn erstes(b)=IIII dann IIII add(a, rest(b)) sonst
                        wenn a=I und b=ε    dann I sonst
                        wenn a=I und b=I    dann II sonst
                        wenn a=I und b=II   dann III sonst
                        wenn a=I und b=III  dann IIII sonst
                        wenn a=I und b=IIII dann (IIII,ε) sonst

                        wenn a=II und b=ε   dann II sonst
                        wenn a=II und b=I   dann III sonst
                        wenn a=II und b=II  dann IIII sonst
                        wenn a=II und b=III dann  (IIII,ε) sonst
                        wenn a=II und b=IIII dann (IIII,I) sonst

                        wenn a=III und b=ε   dann III sonst
                        wenn a=III und b=I   dann IIII sonst
                        wenn a=III und b=II  dann (IIII,ε)  sonst
                        wenn a=III und b=III dann  (IIII,I) sonst
                        wenn a=III und b=IIII dann (IIII,II) sonst

                        wenn a=IIII und b=ε   dann  IIII sonst
                        wenn a=IIII und b=I   dann (IIII,ε)   sonst
                        wenn a=IIII und b=II  dann (IIII,I)   sonst
                        wenn a=IIII und b=III dann (IIII,II) sonst
                        wenn a=IIII und b=IIII dann (IIII,III)
```

funkende

```
function: sub a:bd b:bd →bd ≡
        wenn b=ε dann a sonst
        wenn a=ε dann ε sonst
                wenn a=I und b=II   dann ε sonst
                wenn a=I und b=III  dann ε sonst
                wenn a=I und b=IIII dann ε sonst
                wenn a=II und b=III dann ε sonst
                wenn a=II und b=IIII  dann ε sonst
                wenn a=III und b=IIII dann ε sonst

                wenn a=IIII und b=I     dann IIII sonst
                wenn a=IIII und b=II    dann III  sonst
                wenn a=IIII und b=III   dann II   sonst
                wenn a=IIII und b=IIII  dann I    sonst
                wenn a=IIII und b=I     dann III sonst
                wenn a=IIII und b=II    dann II  sonst
                wenn a=IIII und b=III   dann I   sonst
                wenn a=IIII und b=IIII  dann ε  sonst
                wenn a=III  und b=I     dann II  sonst
                wenn a=III  und b=II    dann I   sonst
                wenn a=III  und b=III   dann ε   sonst

                wenn a=II und b=I      dann I   sonst
                wenn a=II und b=II     dann ε   sonst
                wenn a=I  und b=I      dann ε   sonst

        wenn erstes(a)=IIII und erstes(b)=IIII dann sub(rest(a), rest(b)) sonst
        wenn erstes(a)≠IIII und erstes(b)=IIII dann ε sonst

        wenn erstes(a)=IIII und rest(a)=I   und b=II    dann IIII sonst
        wenn erstes(a)=IIII und rest(a)=I   und b=III   dann III  sonst
        wenn erstes(a)=IIII und rest(a)=I   und b=IIII  dann II   sonst
        wenn erstes(a)=IIII und rest(a)=II  und b=III  dann IIII  sonst
        wenn erstes(a)=IIII und rest(a)=II  und b=IIII dann III   sonst
        wenn erstes(a)=IIII und rest(a)=III und b=IIII dann IIII  sonst

        IIII sub(rest(a), b)

funkende
```

Aufgabe 8: (8 Punkte)

Eine Multilinksequenz über einem Datentyp D ist eine Linkssequenz, in der jedes Element entweder ein Objekt vom Typ D oder wieder eine Multilinksequenz über D ist.
Entwickeln Sie ein Syntaxdiagramm und eine Grammatik in Backus-Naur-Form für Multilinksequenzen über dem Datentyp nat. Die Syntaxdiagramme für nat können Sie als gegeben annehmen.
Geben Sie für Ihre BNF-Grammatik eine Ableitung für folgende Multilinksequenz an:
((1,(3,leer)),(2,(leer,leer))).

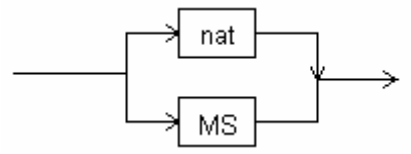
Lösung:
typ MS ≡ leer I (E,MS)
typ E ≡ D I MS
typ D ≡ nat (gegeben)

BNF-Gramatik
<ZifferohneNull>::= 1I2I3I4I5I6I7I8I9
<Ziffer>::= 0 I <ZifferohneNull>
<Ziffernfolge>::= {eps} I <Ziffer><Ziffernfolge>
<nat>::= 0 I <ZifferohneNull><Ziffernfolge>
<E>::= <nat> I <MS>
<MS>::= eps I (<E>,<MS>)

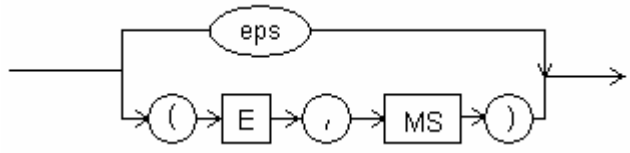
Ableitung von ((1,(3,leer)),(2,(leer,leer))):
<MS>→(<E>,<MS>)→((MS),(<E>,<MS>))→((<E>,<MS>),(<nat>,(<E>,<MS>)))→
((<nat>,(<E>,<MS>)),(2,(<MS>,{eps})))→((1,(<nat>,{eps})),(2,({eps},{eps})))→
((1,(3,{eps})),(2,({eps},{eps})))

Syntax-Diagramm:

Element E



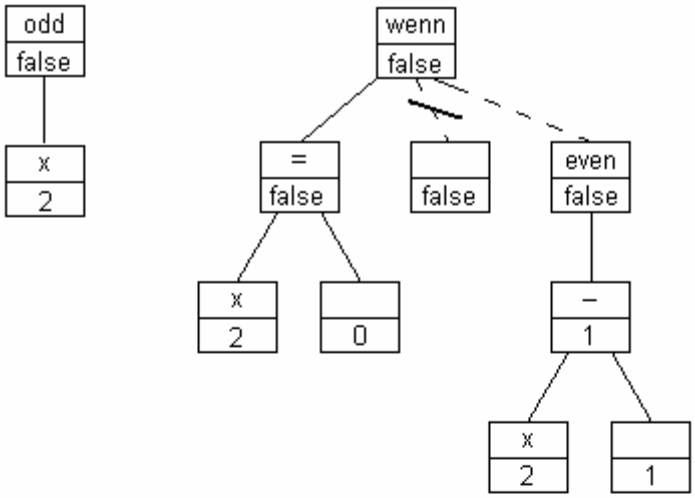
Multiseqenz MS

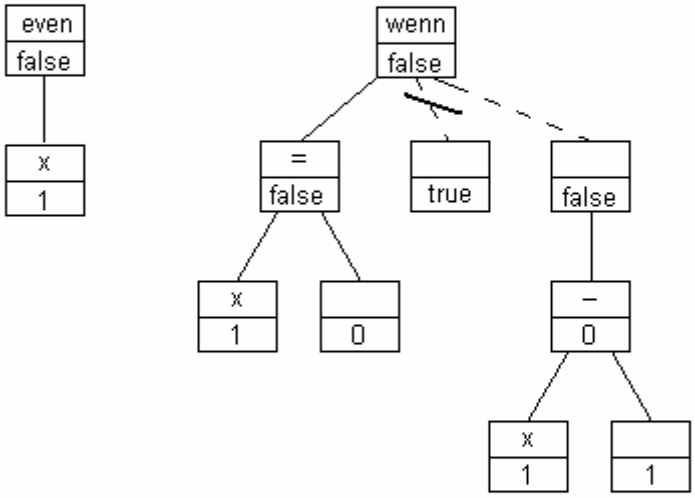


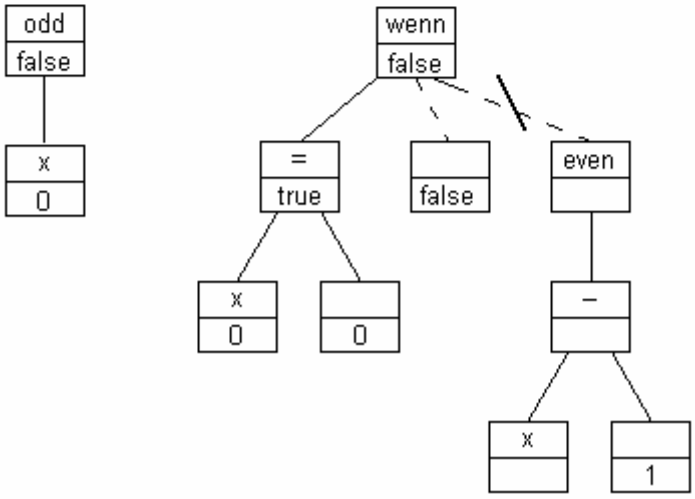
Aufgabe 9: (4 Punkte)

Gegeben seien die beiden Funktionen
funktion even x:nat →bool= wenn x=0 dann true sonst odd (x-1) ende.
funktion odd x:nat→bool= wenn x=0 dann false sonst even(x-1) ende.
Vollziehen die Arbeitsweise der Formularmaschine für odd 2 nach.

Lösung: Formularmaschine für odd2 (odd=ungerade, even=gerade)







Aufgabe 10: (3 + 4 + 5 Punkte)

Ein ternärer markierter Baum ist ein markierter geordneter Baum, in dem jeder Knoten höchstens drei Söhne besitzt.

a) Definieren Sie einen polymorphen ternären markierten Baum in einer funktionalen Programmiersprache.

b) Schreiben Sie eine boolesche Funktion in funktionaler Programmierung, die prüft, ob ein vorgegebenes Objekt in einem ternären Baum als Marke vorkommt oder nicht.

c) Schreiben Sie ein Funktional in funktionaler Programmierung auf ternären Bäumen, das die Markierung jedes Knotens mithilfe einer Transformationsfunktion in eine andere Markierung ändert

Lösung:

a)

typ BaumΔ ≡ {eps} I (Δ, BaumΔ, BaumΔ, BaumΔ)

b)

funktion suche x: BaumΔ a:Δ → bool ≡
    wenn x={eps} dann false sonst
        $\pi_{1,4}$x=a oder suche ($\pi_{2,4}$x, a) oder suche ($\pi_{3,4}$x, a) oder suche ($\pi 4_{,4}$x, a) ende

c)

Funktional:
funktion ändern x:BaumΔ f: [Δ→ Δ] → Baum Δ ≡
    wenn x={eps} dann {eps} sonst
    (f($\pi_{1,4}$x), ändern($\pi_{2,4}$x, f), ändern($\pi_{3,4}$x, f), ändern($\pi_{4,4}$x, f))

ergänzend:

polymorpher Baum → Δ-Parameter sind ersetzbar/austauschbar, wobei Δ für einen Datentyp steht ! (Bsp: int, nat, char, text, Bierdeckelnotation ☺)

typ Baum int ≡ {eps} I (int, Baum int, Baum int, Baum int)
bzw.
typ Baum int ≡ int I (int, Baum int, Baum int, Baum int) (wenn Blätter ebenfalls markiert)