Fachbereich Informatik

Klausur Informatik 2
10. April 2001

Biedl / Lemke
Schöner / Tschirley

Name: ....

Vorname: ....

Matr.-Nr.: ....

Bearbeitungszeit: 120 Minuten

- Benutzen Sie für die Lösung der Aufgaben nur das mit diesem Deckblatt ausgeteilte Papier. **Lösungen, die auf anderem Papier geschrieben werden, können nicht gewertet werden!**
- Schreiben Sie Ihre Lösungen auf das Aufgabenblatt der jeweiligen Aufgabe. Verwenden Sie auch die Rückseite. **Schreiben Sie keine Lösungen einer Aufgabe auf ein Blatt, das nicht zu dieser Aufgabe gehört!** Wenn Sie zusätzliche, von uns ausgegebene Blätter verwenden, geben Sie unbedingt an, zu welcher Aufgabe die Lösung gehört.
- Schreiben Sie deutlich! Doppelte, unleserliche oder mehrdeutige Lösungen können nicht gewertet werden.
- Schreiben Sie nur in **blau** oder **schwarz**. Schreiben Sie nicht mit Bleistift!
- Bei der Implementation von OPAL-Funktionen brauchen *keine* Importe mit angegeben zu werden, wenn Funktionen aus dem gegebenen Teil der Bibliotheca Opalica (oder aus der Struktur `Nat`, `Real`, `Seq` oder den Strukturen für die Listenfunktionale) oder sonstigen (in den Aufgaben) gegebenen Strukturen benutzt werden!
- Die OPAL-Implementierungen müssen nicht dokumentiert werden.
- Erscheint Ihnen eine Aufgabe mehrdeutig, wenden Sie sich an die Betreuer.
- Tragen Sie jetzt (vor Beginn der Bearbeitungszeit) auf *allen* Blättern Ihren Namen und Ihre Matrikelnummer ein.

Aufgabe	Punkte	erreicht
1	15	
2	8	
3	10	
4	14	
5	3	

1. Aufgabe (15 Punkte): Imperative Programmierung

1.1. Umsetzen rekursiver in iterative Programme (6 Punkte)

Setzen Sie die folgende OPAL-Funktion in *zwei* verschiedene Java-Methoden um. Die eine soll rekursiv formuliert sein, die zweite iterativ (mit einer Schleife anstelle der Rekursion). Beide sollen sich so eng wie möglich an der gegebenen OPAL-Funktion orientieren.

```
FUN quersumme: nat -> nat
DEF quersumme == \\n.
  IF n=0 THEN 0
  IF n>0 THEN mod(n,10) + quersumme(div(n,10))
  FI
```

Lösung:
Rekursive Version:

```
static int quersumme(int n) {
    if (n == 0) {
        return (0);
    } else {
        return (n%10 + quersumme(n/10));
    }
}
```

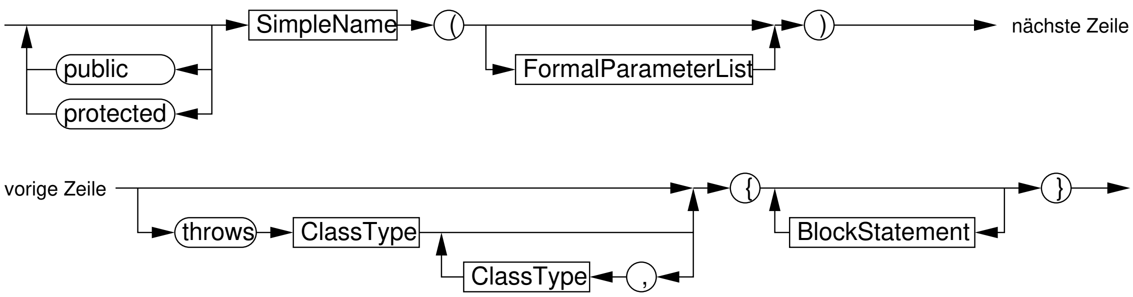
Iterative Version:

```
static int quersumme(int n) {
    int qs = 0;
    while (n != 0) {
        qs = qs + n%10;
        n  = n/10;
    }
    return (qs);
}
```

1.2. Syntaxdiagramme und EBNF-Grammatiken (3 Punkte)

Transformieren Sie das im folgenden gegebene Syntaxdiagramm in eine EBNF-Grammatik.

Beispielsyntax:



Lösung:
EBNF-Grammatik zu dem gegebenen Syntaxdiagramm:

Beispielsyntax ::= { public | protected } *SimpleName* ([*FormalParameterList*])
[throws *ClassType* { , *ClassType* }] { { *BlockStatement* } } .

1.3. Heaps mit beliebigem Grad (6 Punkte)

Als Generalisierung von binären Heaps (Heaps mit dem Grad 2) kann man *Heaps mit dem Grad* d aufbauen. Bei diesen Heaps hat jeder Knoten nicht 2 Kind-Knoten, sondern d Stück. Wird ein solcher Heap in einem Array gespeichert, so berechnen sich die Indizes der Kind-Knoten eines gegebenen Knotens i $\in [0, \dots, N]$ als: d*i + 1, ..., d*i + d. Schreiben Sie eine Java-Methode, die einen Heap des Typs Heap und einen Integer erhält, um die Integer-Zahl in den Heap einzufügen (und die Heap-Bedingung wieder herzustellen).

```
public class Heap {
    int[] data;     // Array, in dem der Heap abgelegt ist
    int heapSize;   // Anzahl der Elemente in 'data', die vom Heap belegt werden
    int d;          // Grad des Heaps
}
```

Lösung:

```
static void heapInsert(Heap h, int x) {
    if (h.heapSize >= h.data.length)
        return; // Hier sollte eine Fehlerbehandlung erfolgen (z.B. Exception)
    h.data[h.heapSize] = x;
    h.heapSize         = h.heapSize + 1;
    upHeap(h, h.heapSize-1);
}

static void upHeap(Heap h, int i) {
    int temp = h.data[i];
    while ((i > 0) && (temp > h.data[(i-1)/h.d])) {
        h.data[i] = h.data[(i-1)/h.d];
        i         = (i-1)/h.d;
    }
    h.data[i] = temp;
}
```

2. Aufgabe (8 Punkte): Aufwandsanalyse

2.1. Aufwandsklassen (3 Punkte)

Gegeben sei die folgende Behauptung: Für zwei beliebige Funktionen $f(x)$ und $g(x)$ ist es immer der Fall, dass mindestens eine der Aussagen $f \in \mathcal{O}(g)$ oder $g \in \mathcal{O}(f)$ zutrifft. D.h. es gibt keine Funktionen f und g, für die $f \notin \mathcal{O}(g) \wedge g \notin \mathcal{O}(f)$. Trifft diese Behauptung zu? Begründen Sie Ihre Antwort (z.B. indem Sie ein Gegenbeispiel angeben).

Lösung:

Sie trifft nicht zu. Ein Beispiel für Funktionen, bei denen die Aussage nicht zutrifft, sind $f(x) = \exp(x) \cdot \sin^2(x)$ und $g(x) = x$. Es lässt sich keine Konstante c finden, sodass ab einem gewissen $n_0 \in \mathbb{N}^+$ die Funktion $f(x)$ immer größer als $g(x)$ ist, und umgekehrt ebensowenig ($f(x)$ ist periodisch jeweils 0 oder gleich der Exponentialfunktion; die Exponentialfunktion wächst asymptotisch schneller als die lineare Funktion, diese aber wiederum schneller als die Konstante 0).

2.2. Aufwandsanalyse eines Programmfragments (5 Punkte)

Bestimmen Sie die Funktion für den zeitlichen Aufwand der folgenden Methode power2 im Worst-Case. Leiten Sie daraus mit Hilfe des $\mathcal{O}$-Kalküls oder der im Skript oder unten angegebenen Regeln eine Aufwandsklasse ab. Nehmen Sie einen Zeitaufwand von einer Einheit für jeden Operator, jeden Funktionsaufruf und jedes return an. Achten Sie auf die Nachvollziehbarkeit Ihres Vorgehens; insbesondere müssen die gezählten Operatoren, etc. eindeutig einzelnen Zeilen der Methode zugeordnet werden können!

```
1  static int power2(int n) {
2      if (n == 0) {
3          return (1);
4      } else {
5          return (power2(n−1) + power2(n−1));
6      }
7  }
```

Mit welchem einfachen Trick ist es möglich, den Aufwand linear zu machen?

Lösung:

Der Aufwand der Methode ist von dem Parameter n abhängig.

$$A(0) = \underbrace{1}_{2} + \underbrace{1}_{3}$$

$$A(n) = \underbrace{1}_{2} + \underbrace{6 + 2A(n-1)}_{5} \qquad \text{für } n > 0$$

$$\Longrightarrow A(y) = \begin{cases} 2 & \text{für } n = 0 \\ 2A(n-1) + 7 & \text{für } n > 0 \end{cases}$$

Mit $c = 2$, $k = 0$ und $b = 7$ ergibt sich aus der Tabelle mit den Rekurrenzrelationen (zweite Zeile) die Aufwandsklasse

$$A(n) \in \mathcal{O}(2^n)$$

Der Aufwand lässt sich linearisieren, indem der rekursive Aufruf nur einmal, statt zweimal, ausgeführt wird. Da beide Aufrufe mit dem gleichen Argument erfolgen und die Funktion keine Seiteneffekte besitzt, kann das Ergebnis eines Aufrufs zwischengespeichert werden. Danach können beide Operanden der Addition durch das gespeicherte Resultat ersetzt werden:

```
static int power2(int n) {
    if (n == 0) {
        return (1);
    } else {
        int temp = power2(n−1);
        return (temp + temp);
    }
}
```

Definition ($\mathcal{O}$-Kalkül):
Sind f und g zwei positiv-reellwertige Funktionen über $\mathbb{N}$, dann bedeutet $f \in \mathcal{O}(g)$, dass es eine Konstante $c \in \mathbb{R}^+$ und ein $n_0 \in \mathbb{N}$ gibt, sodass für alle $n > n_0$ gilt: $f(n) \leq c * g(n)$. g ist dann eine obere Schranke für f.

Regeln für rekurrente Aufwandsformeln:

Rekurrenzrelation		$\mathcal{O}(A)$
$A(n) = A(n-1) + bn^k$		$\mathcal{O}(n^{k+1})$
$A(n) = cA(n-1) + bn^k$	mit $c > 1$	$\mathcal{O}(c^n)$
$A(n) = cA(n/d) + bn^k$	mit $c > d^k$	$\mathcal{O}(n^{\log_d c})$
$A(n) = cA(n/d) + bn^k$	mit $c < d^k$	$\mathcal{O}(n^k)$
$A(n) = cA(n/d) + bn^k$	mit $c = d^k$	$\mathcal{O}(n^k \log n)$

3. Aufgabe (10 Punkte): Programmverifikation

3.1. Formalisieren von Aussagen (2 Punkte)

Formalisieren Sie die folgenden Aussagen (Hilfsprädikate sind mit zu definieren):

1. Der Array arr erfüllt im Indexbereich von 0 bis n die Heap-Bedingung.
2. x ist das Maximum von a und b. (Benutzen Sie hier *kein* Prädikat max!)

Lösung:

1. $\forall i \in \mathbb{N}_0 : \Big(\big((2i+1) \leq n\big) \Rightarrow \big(arr_i > arr_{2i+1}\big)\Big) \wedge \Big(\big((2i+2) \leq n\big) \Rightarrow \big(arr_i > arr_{2i+2}\big)\Big)$
2. $\big(x = a \vee x = b\big) \wedge x \geq a \wedge x \geq b$

3.2. Entwickeln eines Programms aus einer Spezifikation (4 Punkte)

Gegeben seien die folgenden PRE- und POST-Bedingungen, sowie die Invariante I einer Funktion root(int n), die die ganzzahlige Quadratwurzel einer natürlichen Zahl n bestimmt (in der lokalen Variablen res). Implementieren Sie in Java eine Funktion, die diese Aufgabe erledigt und dabei die angegebene Spezifikation erfüllt.

- PRE: $n \in \mathbb{N}^+$
- POST: $res \in \mathbb{N}^+ \wedge \big(res \cdot res \leq n\big) \wedge \big((res+1) \cdot (res+1) > n\big)$
- I: $res \in \mathbb{N}^+ \wedge \big(res \cdot res \leq n\big)$

Lösung:

```
static int root(int n) {
    int res;
    // PRE: n ∈ ℕ+
    res = 1;
    // I: res ∈ ℕ+ ∧ (res · res ≤ n)
    while ((res+1)*(res+1) <= n) {
        res = res + 1;
    }
    // POST: res ∈ ℕ+ ∧ (res · res ≤ n) ∧ ((res + 1) · (res + 1) > n)
    return (res);
}
```

3.3. Partielle Korrektheit (4 Punkte)

Führen Sie den angefangenen Beweis der partiellen Korrektheit für die folgende Funktion zu Ende. Verwenden Sie dafür den Vordruck auf der nächsten Seite.

```
static double heaviside(double x) {
    int res;
    // PRE: true
    if (x < 0) {
        res = 0.0;
    } else {
        res = 1.0;
    }
    // POST: (x < 0 ∧ res = 0) ∨ (x ≥ 0 ∧ res = 1)
    return(res);
}
```

Lösung:

```
static double heaviside(double x) {
    int res;
    // PRE: true
    if (x < 0) {
        // true ∧ x < 0
        // (x < 0 ∧ 0 = 0)
        res = 0.0;
        // (x < 0 ∧ res = 0)
        // (x < 0 ∧ res = 0) ∨ (x ≥ 0 ∧ res = 1)
    } else {
        // true ∧¬(x < 0)
        // (x ≥ 0 ∧ 1 = 1)
        res = 1.0;
        // (x ≥ 0 ∧ res = 1)
        // (x < 0 ∧ res = 0) ∨ (x ≥ 0 ∧ res = 1)
    }
    // POST: (x < 0 ∧ res = 0) ∨ (x ≥ 0 ∧ res = 1)
    return(res);
}
```

```
static double heaviside(double x) {
    int res;
    // PRE: true

    if (x < 0) {

        res = 0.0;

    } else {

        res = 1.0;

    }

    // POST: (x < 0 ∧ res = 0) ∨ (x ≥ 0 ∧ res = 1)
    return(res);
}
```

4. Aufgabe (14 Punkte): SCRAM-Assemblerprogrammierung

Implementieren Sie ein Assemblerprogramm, das die aus dem Übungsbetrieb bekannte Funktion `toBase` nachbildet. Gehen Sie dazu in folgenden Schritten vor:

4.1. (3 Punkte) Implementieren Sie ein SCRAM-Assemblermakro `MOD`, das die Modulo-Operation auf dem Inhalt des AC mit dem übergebenen Parameter ausführt. Das Ergebnis der Berechnung soll im AC stehen, wenn das Makro beendet wird.

Achten Sie darauf, keine unnötigen Instruktionen zu verwenden. Verwenden Sie zur Lösung einen iterativen Algorithmus, der *nicht* die Befehle `DIV` und `MUL` enthält.

Beispiel:

Wenn vor einer Instruktion der Wert 13 im AC enthalten ist, so wird nach dem Befehl `MOD 5` der AC den Wert 3 enthalten.

Lösung:

```
%MACRO          MOD
lbl%IDENT       SUB     %PARAM
                JNS     end%IDENT
                JMP     lbl%IDENT
end%IDENT       ADD     %PARAM
%END
```

4.2. (5 Punkte) Implementieren Sie ein SCRAM-Assemblermakro `POW`, das den Inhalt des AC so oft mit sich selbst multipliziert, wie der übergebene Parameter angibt (es handelt sich hier um die Berechnung des mathematischen Ausdrucks $AC^{Parameter}$). Das Ergebnis der Operation soll im AC hinterlegt werden.

Lösung:

```
%MACRO          POW
                STA     powbase         ; AC sichern
                LDA     %PARAM          ; Parameter abspeichern
                STA     powby           ; x^1 bilden

                JNE     lbl%IDENT       ; x^0 := 1
                LDA     #1              ; \
                STA     powerg          ;  der kurze Weg für x^0
                JMP     end%IDENT       ; /

lbl%IDENT       DEC     powby
                JEQ     end%IDENT       ; x^1 wurde schon abgelegt!
                LDA     powbase         ; \
                MUL     powerg          ;  aufmultiplizieren
                STA     powerg          ; /
                JMP     lbl%IDENT

end%IDENT       LDA     powerg

powbase         RES     1
powby           RES     1
powerg          RES     1
%END
```

4.3. (6 Punkte) Die Zahl, die konvertiert werden soll, befindet sich an der Speicherstelle mit dem Label `src`. Die Zahl ist im Dezimalsystem codiert.

Implementieren Sie mit Hilfe der beiden oben entworfenen Makros das Hauptprogramm, das die Zahl `src` in ein beliebiges Zahlensystem der Basis `base` mit $base \in [2..10]$ konvertiert. Die Basis befindet sich an der Speicherstelle mit dem Label `base`. Das Ergebnis soll sich nach Abarbeitung des Programms an der Speicherstelle mit dem Label `erg` befinden.

Lösung:

```
; Formel: Stelle_i = mod(div(base^i, tmp), base)

                ORG     0

loop            LDA     base    ; base^i berechnen
                POW     i       ;
                STA     tmp     ; und abspeichern

                LDA     src     ; \
                DIV     tmp     ;  div(base^i, tmp)
                JEQ     end     ; /

                MOD     base    ; mod(div(base^i, tmp), base)
                STA     tmp     ; zwischenspeichern

                LDA     #10     ; \
                POW     j       ;  Stellenverschiebung durchführen
                MUL     tmp     ; /

                ADD     erg     ; mit vorherigem Ergebnis aufaddieren
                STA     erg     ;

                INC     i       ; Stelle erhöhen
                INC     j       ;
                JMP     loop

end             HLT

; vorgegebene Variablen
src             DAT     90
erg             DAT     0
base            DAT     8
; temporäre Variablen
tmp             RES     1
i               DAT     0
j               DAT     0
                END
```

Hinweis: Sie dürfen zur Lösung der Aufgabe die folgenden Makros verwenden:

- **INC** Makro zum Inkrementieren einer Variablen. Der Wert der als Parameter angegebenen Variablen wird um 1 erhöht und sowohl im AC als auch in der Variablen selbst abgelegt. Nicht nutzbar im unmittelbaren Adressierungsmodus.
- **DEC** Makro zum Dekrementieren einer Variablen. Der Wert der als Parameter angegebenen Variablen wird um 1 verringert und sowohl im AC als auch in der Variablen selbst abgelegt. Nicht nutzbar im unmittelbaren Adressierungsmodus.
- **JNS** Makro zum bedingten Sprung bei negativem AC-Inhalt. Als Parameter wird analog zu den anderen bedingten Sprüngen die Sprungmarke angegeben, zu der ggf. gesprungen wird. Ist der AC-Inhalt nicht negativ, wird normal fortgefahren. Der Wert des AC wird nicht verändert.

Wenn Sie weitere Makros verwenden, so ist der Quellcode anzugeben.

5. Aufgabe (3 Punkte): Zahlensysteme und SCRAM

5.1. Zahlensysteme (1 Punkte) Zahlen werden im Computer im b-adischen Zahlensystem dargestellt, wobei die Basis 2 ist. Beantworten Sie die folgenden Fragen. Begründen Sie Ihre Antwort!

1. Ist diese Darstellung der natürlichen Zahlen umfassend?

   Lösung:
   Nein. Üblicherweise ist die Anzahl der Bits, die für die Darstellung von Zahlen verwendet wird, begrenzt. Damit können Zahlen ab einer bestimmten Obergrenze nicht mehr dargestellt werden.

2. Ist sie eindeutig?

   Lösung:
   Wenn die negativen Zahlen im 2er-Komplement dargestellt werden, ja. Im 1er-Komplement (oder bei explizitem Vorzeichen-Bit) existieren zwei Darstellungen der Null.

5.2. SCRAM (2 Punkte)

1. Prüfen Sie kurz, ob das `MAR` im SCRAM tatsächlich notwendig ist. Ist es unnötig und kann man es ohne große Probleme entfernen, oder ist seine Anwesenheit zwingend notwendig? Begründen Sie Ihre Antwort!

   Lösung:
   Das Register ist im bekannten SCRAM-Aufbau notwendig, da sonst keine indirekten Speicherzugriffe stattfinden können. Es kann jedoch eingespart werden, wenn das MBR mit MS-FlipFlops aufgebaut wird. Beim Problemfall des Schreibens in den Speicher mit indirekter Adressierung wird dann, nachdem die Speicheradresse aus dem Speicher in das MBR geladen wurde, der Slave-Teil des MBR die Adresse für den Speicherzugriff speichern, so daß der zweite Speicherzugriff erfolgen kann. Während beim Setzen der Leitung x_{12} der Master-Teil dann den Speicherinhalt aufnimmt, hält der Slave-Teil noch die Zugriffsadresse. Erst wenn die fallende Taktflanke kommt, wird dann die Speicheradresse im MBR durch das Speicherwort ersetzt.

2. Zählen Sie auf, was nötig ist, um einen Speicherzugriff durchzuführen. Erläutern Sie kurz den Ablauf.

   Lösung:
   - *Adressierung der Speicherzelle (Schreiben der Adresse in das MAR). Dadurch wird die gewünschte Speicherzelle aktiviert.*
   - *Festlegung der Zugriffsart* Lesen *oder* Schreiben *durch Steuerleitung x_{12}. Je nach Zugriffsart bedeutet dies eine oder zwei Aktionen:*

     Lesen aus dem Speicher:
     (a) *Der Ausgang der selektierten Speicherzelle wird auf den Ausgang des Speichers und damit auf MAR_MUX gelegt.*

     Schreiben in den Speicher:
     (a) *Der Ausgang des MBR wird auf den Eingang der selektierten Speicherzelle gelegt.*
     (b) *Die Speicherzelle bekommt sofort den Impuls zum Laden.*