Fachbereich
Informatik

Klausur Informatik A
27. März 1998

Name: ....

Matr.-Nr. ....

Bearbeitungszeit: 240 Minuten

Spielregeln (**Jetzt lesen!**):

- Benutzen Sie für die Lösung der Aufgaben **nur** das mit diesem Deckblatt ausgeteilte Papier. Lösungen, die auf anderem Papier geschrieben werden, können **nicht** bewertet werden. Schreiben Sie Ihre Lösung auch auf die Rückseiten der Blätter; benötigen Sie für eine Lösung mehr als ein Blatt, finden Sie am Ende der Klausur Leerblätter. Zusätzliches Papier können Sie von den Tutoren bekommen.
- Tragen Sie jetzt (vor Beginn der eigentlichen Bearbeitungszeit!) auf *allen* Blättern Ihren Namen und Ihre Matrikelnummer ein.
- Schreiben Sie deutlich! Unleserliche oder zweideutige Lösungen können nicht gewertet werden.
- Schreiben Sie **nicht** mit Bleistift!
- Schreiben Sie nur in **blau** oder **schwarz**!

Viel Erfolg!

Bewertung
(bitte offenlassen :–)

Aufgabe	Punkte	Erreichte Punkte
1	11	
2	10	
3	12	
4	5	
5	8	
6	11	
7	9	
8	6	
9	10	
10	8	
11	9	
Summe	99	

• AUFGABE 1 (11 Punkte) Waggons und Züge in Opal, Higher-Order-Functions.

Bei der Bahn sollen Waggons von Personenzügen rechnergestützt verwaltet werden. Es gibt zwei Arten von Waggons:

- Passagierwagen
- Speisewagen

Jeder Waggon besitzt eine maximal zulässige Geschwindigkeit in km/h. Ein Passagierwagen hat eine bestimmte Anzahl von Plätzen.

1. (1 Punkt) Geben Sie eine geeignete Opal-Typdefinition für den Datentyp `Waggon` an, in dem die relevanten Daten – so wie oben angegeben – enthalten sind.
2. (1 Punkt) Was versteht man unter der „induzierten Signatur“ eines Datentyps und aus welchen Teilen besteht sie? Geben Sie die induzierte Signatur des Typs `Waggon` an und ordnen Sie jede Operation einem der oben angegebenen Teile zu.
3. (1 Punkt) Die Bahn wird schneller. Deklarieren und definieren Sie eine Funktion, die die maximale Geschwindigkeit eines Waggons um einen gegebenen Betrag schneller als bisher macht.

Ein kompletter Zug kann als Sequenz von Waggons repräsentiert werden: Typ `seq[Waggon]`. Bei der Definition der Funktionen in den folgenden Aufgaben **müssen** höherwertige Funktionen wie `map`, `filter` oder `reduce` verwendet werden. Explizite Rekursion **darf nicht** verwendet werden.

---

4. (1 Punkt) Deklarieren und definieren Sie eine Funktion, die einen ganzen Zug um 10 km/h schneller macht (Maximalgeschwindigkeiten).
5. (2 Punkte) Deklarieren und definieren Sie eine Funktion, die aus einem Zug alle die Waggons heraussucht, die schneller als 160 km/h fahren können.
6. (2 Punkte) Deklarieren und definieren Sie eine Funktion, die die Geschwindigkeit des „langsamsten“ Waggons eines Zuges ermittelt.
7. (3 Punkte) Deklarieren und definieren Sie eine Funktion, die die Anzahl der Plätze eines gegebenen Zugs bestimmt.

• AUFGABE 2 (10 Punkte) Opal, Ausdrücke als Bäume, Auswertung, Rekursion.

Im folgenden betrachten wir einfache arithmetische Ausdrücke, die als Bäume dargestellt werden. Ein Ausdruck kann eine ganze, positive Zahl sein oder eine zweistellige Operation, die wieder auf zwei Ausdrücke angewendet wird.

Gegeben sind folgende Opal-Typdefinitionen:

```
DATA opkind == add sub mul div

DATA expr == num (val : nat)
             node (op : opkind, left : expr, right : expr)
```

Der Ausdruck $8 + (5 - 3) * 4$ z. B. sieht als Baum folgendermaßen aus:

```
node(add,num(8),node(mul,node(sub,num(5),num(3)),num(4)))
```

Graphisch läßt er sich wie folgt darstellen:

```
   +
  / \
 8   *
    / \
   -   4
  / \
 5   3
```

1. (1 Punkt) Wie nennt man Typen wie `opkind`?
2. (3 Punkte) Deklarieren und definieren Sie eine Funktion `correct?`, die überprüft, ob ein Ausdruck *korrekt* ist. Das ist dann der Fall, wenn nirgends bei der Auswertung negative Zahlen auftreten und nirgends durch null geteilt wird. **Hinweis.** Evtl. läßt sich die Funktion der nächsten Aufgabe sinnvoll einsetzen!
3. (2 Punkte) Deklarieren und definieren Sie eine Funktion `eval` zum Auswerten eines korrekten Ausdrucks. Geben Sie eine geeignete Vorbedingung (Precondition) an.
4. (3 Punkte) Deklarieren und definieren Sie eine Funktion `exprMap`, mit der Veränderungen an bestimmten Blättern (Zahlen) eines Ausdrucks vorgenommen werden können. Die zu verändernden Zahlen sollen durch eine Funktion `nat -> bool` (Prädikat) bestimmt werden. Die Änderung der Zahlen ebenfalls durch eine entsprechende Berechnungsfunktion. Herauskommen soll der (möglicherweise) veränderte Ausdruck.
5. (1 Punkt) Deklarieren und definieren Sie eine Funktion `squareEvens`, die in einem Ausdruck alle geraden Zahlen quadriert. Benutzen Sie dabei keine explizite Rekursion! Das Ergebnis soll der Ausdruck mit den quadrierten Zahlen sein.

   Beispiel:

```
   *                    *
  / \                  / \
 2   +                4   +
    / \      ==>         / \
   *   8                *   64
  / \                  / \
 7   *                7   *
    / \                  / \
   6   3                36   3
```

• AUFGABE 3 (12 Punkte) Opal, Geraden in der Ebene, Spezifikationen.

Jede Gerade in der kartesischen Ebene läßt sich durch eine Funktionsgleichung der folgenden Form darstellen:

$$f(x) = ax + b$$

Dabei ist a die *Steigung* und b der *Abschnitt* (Schnittpunkt mit der Y-Achse) der Geraden. In Opal können wir Geraden entweder mit dieser Repräsentation darstellen oder aber als „echte" Funktion vom Typ `real -> real`.

Damit ergibt sich folgender Opal-Datentyp für Geraden:

```
DATA line == asPair (a : real, b : real)
             asFun (f : real -> real)
```

Wichtig. Die Variante `asFun` läßt auch andere Funktionen außer Geraden zu. So hat z.B. die Sinus-Funktion in Opal ja ebenfalls den Typ `real -> real`. Im folgenden soll jedoch davon ausgegangen werden, daß es sich bei jeder so repräsentierten Funktion tatsächlich um eine Gerade handelt.

Gehen Sie davon aus, daß die arithmetischen Funktionen in Opal exakt arbeiten, d. h., es entstehen keinerlei Rundungsfehler.

1. (1 Punkt) **Spezifizieren** und definieren Sie eine Funktion `FUN apply : line ** real -> real`, die den Funktionswert einer Geraden an einer gegebenen Stelle ausrechnet.
2. (1 Punkt) Gegeben ist folgende Spezifikation einer Funktion. Beschreiben sie in Worten **kurz**, was die Funktion berechnet und implementieren Sie sie.

```
FUN p : line ** real ** real -> bool
-- SPC p (l,x,y) = b
-- PRE  true
-- POST b <==> (asFun?(l)  and  f(l)(x) = y)         or
--             (asPair?(l) and  a(l)*x + b(l) = y)
```

Zwei gegebene verschiedene Punkte in der Ebene definieren eindeutig eine Gerade. Seien die zwei Punkte (x_0, y_0) und (x_1, y_1) gegeben, dann ist: $a = \frac{y_1 - y_0}{x_1 - x_0}$ und $b = y_0 - ax_0$). Als zusätzliche Voraussetzung muß gelten, daß die X-Werte der Punkte verschieden sind, da wir die Gerade sonst nicht als Funktion darstellen könnten. Im folgenden soll daher unter „verschiedenen Punkten" verstanden werden, daß (mindestens) die X-Werte differieren.

---

3. (2 Punkte) Deklarieren, **spezifizieren** und definieren Sie eine Funktion `makeLine`, die aus zwei verschiedenen Punkten die dazugehörige Gerade ermittelt.
4. (3 Punkte) Deklarieren, **spezifizieren** und definieren Sie eine Funktion `changeRep`, die die Repräsentation einer Geraden in die jeweils andere umwandelt.
5. (2 Punkte) Deklarieren, **spezifizieren** und definieren Sie eine Funktion `intersect?`, die überprüft, ob zwei Geraden **genau** einen Schnittpunkt haben, d. h., ob sie nicht parallel zueinander (oder übereinander) liegen.
6. (3 Punkte) Deklarieren, **spezifizieren** und definieren Sie eine Funktion `intersection`, die den Schnittpunkt zweier sich in einem Punkt schneidender Geraden berechnet.

• AUFGABE 4 (5 Punkte) Prozeduren und Parameterübergabe in Modula.

Gegeben sind eine Prozedur `P` und ein Programmstück aus dem Hauptprogramm mit den globalen Variablen `a`, `b`, `c` und `d`.

```
TYPE Ptr = POINTER TO CARDINAL;

PROCEDURE P (...);
 BEGIN
   a:=a+10;
   c^:=b+2;
   NEW(c);
   c^:=b+6;
   b:=a+1;
END P;

(* Hauptprogramm *)

 VAR  a,b : CARDINAL;
      c,d : Ptr;

BEGIN
 a := 5;
 b := 6;
 NEW(c);
 c^ := a + b;
 NEW(d);
 d^ := c^;
 P (d,b);
 (* XXX *)
```

Es sollen zwei verschiedene Versionen des Programms betrachtet werden, die sich jeweils durch einen anderen Prozedurkopf der Prozedur `P` unterscheiden. Die einzelnen Prozedurköpfe sind wie folgt gegeben:

```
(* K1 *) PROCEDURE P (c : Ptr; a : CARDINAL);
(* K2 *) PROCEDURE P (VAR d : Ptr; VAR b : CARDINAL);
```

1. (4 Punkte) Welchen Wert haben die globalen Variablen `a` und `b` bzw. auf welche Werte zeigen `c` und `d` an der mit `(* XXX *)` gekennzeichneten Programmstelle? Geben Sie die Werte für die oben angegebenen Prozedurkopfvarianten K1 und K2 an. Benutzen Sie für diese Angaben untenstehende Tabelle.
2. (1 Punkt) Gibt es die Unterscheidung zwischen Call-By-Value und Call-By-Reference auch in funktionalen Sprachen, also z.B. in Opal? Bitte begründen Sie kurz!

	a	b	c^	d^
K1				
K2				

• AUFGABE 5 (8 Punkte) Aufwandsberechnungen, $\mathcal{O}$-Kalkül.

Definition: Sind f und g zwei 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, so daß für alle $n > n_0$ gilt: $f(n) \leq c * g(n)$.

Für die folgenden beiden Aufgabenteile dürfen die untenstehenden Rechenregeln **nicht** benutzt werden.

1. (2 Punkte) Zeigen oder widerlegen Sie anhand der Definition des $\mathcal{O}$-Kalküls: $(n+2)^2 \in \mathcal{O}(n^2)$
2. (2 Punkte) Zeigen Sie anhand der Definition des $\mathcal{O}$-Kalküls: $\mathcal{O}(f) * \mathcal{O}(g) \subseteq \mathcal{O}(f * g)$
3. (3 Punkte) Gegeben sei folgendes Programmstück in Modula:

```
i:=0;
WHILE i < n DO
  j := 0;
  WHILE (j < i) DO
    j := j+1;
    FOO(k)
  END; (* WHILE *)
  i := i+1;
END; (* WHILE *)
```

   Die Prozedur `FOO` hat linearen Aufwand in Bezug auf ihr Argument. Die Variable k hängt nicht von n ab und ist somit konstant.

   Bestimmen Sie den Aufwand des Programmstücks in Abhängigkeit der Eingabegröße n in Summenschreibweise und vereinfachen Sie die Formel zu einer einfachen, dichten oberen Schranke. Kommentieren Sie Ihre Rechenschritte (z.B. benutzte Rechenregeln).

Hinweise. Es gilt

$$\sum_{q=1}^{r} q = \frac{r^2 + r}{2}$$

Rechenregeln:

$$\begin{array}{lrcl}
(1) & f & \in & \mathcal{O}(f) \\
(2) & k * \mathcal{O}(f) & = & \mathcal{O}(f), k \text{ konstant} \\
(3) & \mathcal{O}(\mathcal{O}(f)) & = & \mathcal{O}(f) \\
(4) & \mathcal{O}(f) * \mathcal{O}(g) & = & \mathcal{O}(f * g) \\
(5) & \mathcal{O}(f * g) & = & f * \mathcal{O}(g) \\
(6) & \mathcal{O}(f) + \mathcal{O}(g) & = & \mathcal{O}(f + g) \\
(7) & f + \mathcal{O}(g) & \subseteq & \mathcal{O}(f + g) \\
(8) & \mathcal{O}(f) + \mathcal{O}(g) & = & \mathcal{O}(f), \text{ falls } \mathcal{O}(g) \subseteq \mathcal{O}(f) \\
(9) & a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 & \in & \mathcal{O}(n^k), a_k > 0
\end{array}$$

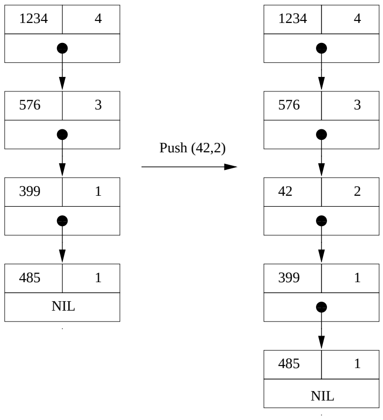
---

4. (1 Punkt) Wie würde sich der Aufwand ändern, wenn die Eingabegröße des obigen Algorithmus nicht n, sondern k wäre? Begründen Sie kurz.

• AUFGABE 6 (11 Punkte) Imperative Programme, Rekursive Datenstrukturen in Modula.

In einer Firma sollen Aufträge verwaltet werden, die unterschiedlich hohe Bearbeitungsprioritäten haben. Jeder Auftrag besitzt eine eindeutige Auftragsnummer und eine Priorität, die durch eine ganze Zahl ausgedrückt wird. Je höher die Zahl, umso höher die Priorität.

Definieren Sie eine Datenstruktur in Modula-2, um einen *Prioritäts-Stack* zu realisieren. Ein *einfacher Stack* besteht aus einer Liste von Einträgen, wobei neue Einträge stets vorne eingefügt werden und das Entfernen bzw. Herausholen von Einträgen ebenfalls nur von vorne geschehen darf. Diese Eigenschaften sollen etwas gelockert werden, um den Einträgen je nach der ihnen zugeordneten *Priorität* eine bestimmte Position in der Liste zu geben. Höher priorisierte Aufträge stehen weiter vorne im Stack als niedrigere. Dadurch wird beim Entfernen des vordersten Eintrags zur Bearbeitung stets der Auftrag genommen, der die höchste Priorität hat. Es darf also hinter keinem Auftrag, der die Priorität x hat, ein weiterer Auftrag stehen, der eine Priorität y mit $y > x$ besitzt.



Einfügen eines Auftrags mit Priorität 2 und Auftragsnummer 42.

1. (2 Punkte) Geben Sie ein geeignetes Modula-2 Definitionsmodul an, das nur den Typ des Stacks opak und folgende Prozeduren oder Funktionsprozeduren exportiert:
   - `Push` – Einfügen eines gegebenen Auftrags in einen Stack an eine der Priorität entsprechenden Stelle.
   - `Pop` – Herausholen und Entfernen des ersten Stack-Eintrages.
   - `Promote` – Erhöhen der Priorität eines bestimmten Auftrags im Stack um eins und gegebenenfalls Verschieben des Eintrags.
2. (9 Punkte) Geben Sie zum Definitionsmodul ein Implementierungsmodul an, das den Stack und die obigen Prozeduren implementiert. **Kommentieren Sie Ihr Programm!**

• **AUFGABE 7 (9 Punkte) Programmverifikation, Hoare-Kalkül.**

Spezifizieren, implementieren und verifizieren Sie eine Prozedur

```
PROCEDURE Find (a : ARRAY OF CARDINAL; x : CARDINAL; VAR r : CARDINAL),
```

die in einem Array von ganzen Zahlen die erste Zahl ermittelt, die gößer als ein gegebener Wert x ist. Es wird garantiert, daß eine solche Zahl existiert.

Beispiel (in Pseudocode):

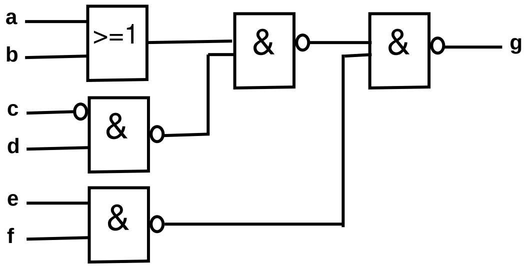
```
a := [2,3,5,7,3,2,5,10,4,12,9,2];
Find (a,7,r);
  ==> r = 10
```

Im einzelnen bedeutet das die Bearbeitung folgender Teilaufgaben:

1. (2 Punkte) Spezifizieren Sie die Prozedur `Find` wie oben beschrieben formal.
2. (2 Punkte) Geben Sie eine **geeignete** Invariante für den Rumpf der Prozedur an und beschreiben Sie kurz ihre Bedeutung.
3. (1 Punkt) Implementieren Sie die Prozedur, so daß sie der Spezifikation genügt.
4. (4 Punkte) Beweisen Sie die partielle Korrektheit des Rumpfes mittels der Invariante und der Regeln des Hoare-Kalküls. **Geben Sie dabei die verwendeten Regeln an!**

• **AUFGABE 8 (6 Punkte) Schaltnetz–Analyse.**

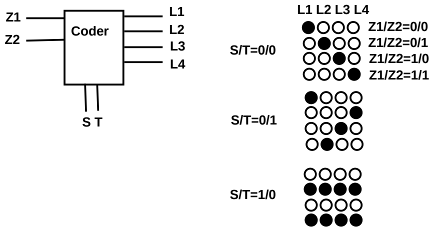
Realisieren Sie die nachfolgende Schaltung g(a,b,c,d,e,f) ausschließlich mit NOR-Gattern.



1. (1 Punkt) Zeichnen sie ein Negations-Gatter mit Hilfe **eines** NOR-Gatters.
2. (2 Punkte) Formen Sie mittels Boolscher Algebra die vorgegebene Schaltung so um, daß sie nur noch NOR-Gatter verwendet. Benutzen Sie dazu die DeMorganschen Regeln.
3. (3 Punkte) Zeichnen Sie das Schaltnetz unter ausschließlicher Verwendung von NOR-Gattern. Die Eingänge dürfen negiert werden.

• AUFGABE 9 (10 Punkte) Entwurf eines Schaltnetzes.

Entwerfen Sie ein Schaltnetz mit zwei Eingängen (Z1,Z2), zwei Steuereingängen (S,T) und vier Ausgängen (L1, L2, L3, L4), mit denen ein Lauflicht gesteuert wird. Es ist egal, welche Lichter im Modus S = T = 1 angesteuert werden.



Verwenden Sie dazu folgende Entwurfsmethode:

1. (2 Punkte) Aufstellen der Wertetabelle für L1,L2,L3 und L4.
2. (4 Punkte) Minimierung der Gleichungen für L1 und L2 mittels KV–Diagramm.
3. (3 Punkte) Zeichnen der Schaltnetze für L1 und L2.
4. (1 Punkt) Welche Funktionalität des Schaltnetzes ergibt sich durch die Festlegung bei der Minimierung für den Modus S = T = 1?

• AUFGABE 10 (8 Punkte) Entwurf eines Zählers.

Entwerfen Sie den Mikroprogrammzähler des SCRAM mit 4 JK–Flipflops und einem externen Eingang **x**0, der die INC und CLEAR Funktionalität steuert. Bei **x**0 = 0 soll der Zähler von 0 bis 8 zählen und bei **x**0 = 1 soll der Mikroprogrammzähler auf 0 zurückgestellt werden. Die Zählfolge soll in binärer Form an den Ausgängen der Flipflops anliegen. Verwenden Sie dazu folgende Entwurfsmethode:

1. (2 Punkte) Zeichnen des Zustandsübergangsdiagramms.
2. (2 Punkte) Erstellen der Wertetafel mit den Eingangskombinationen und deren Folgezuständen.
3. (2 Punkte) Bestimmung der JK–Flipflop–Ansteuerung aus den Übergängen.
4. (2 Punkte) Bestimmung der disjunktiven Normalform für zwei beliebige der vier JKs aus der Wertetafel.

Wertetafel des JK–Flipflops:

J	K	Q	~ Q	Funktion
0	0	Q	~ Q	speichern
0	1	0	1	löschen
1	0	1	0	setzen
1	1	~ Q	Q	toggeln

Übergangstabelle des JK–Flipflops:

Q	Q+	Funktion	J	K
0 →	0	speichern, löschen	0	*
0 →	1	setzen, toggeln	1	*
1 →	0	löschen, toggeln	*	1
1 →	1	speichern, setzen	*	0

• **AUFGABE 11 (9 Punkte) Assemblerprogrammierung.**

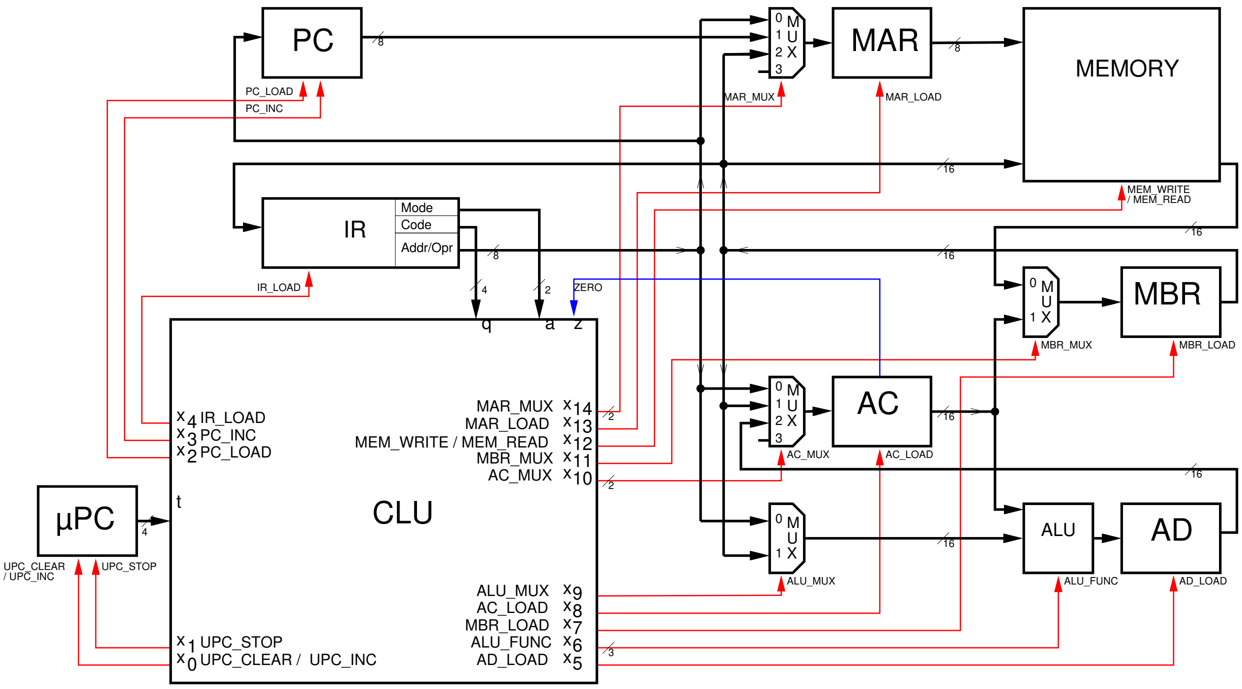
Gegeben ist der in der Vorlesung behandelte SCRAM Rechner und seine Assemblerbefehle. Die zur Lösung notwendigen Unterlagen sind der Klausur beigefügt.

1. (9 Punkte) Bilden sie den Log2 aller Werte eines Null-terminierten Feldes. Das Feld enthält ansonsten keine Nullen.

   Der ganzzahlige Log2 berechnet sich wie folgt :

$$log_2(x) = \begin{cases} 0, \text{ falls } x < 2 \\ 1 + log_2(\frac{x}{2}) \text{ sonst} \end{cases}$$

Der SCRAM (Simple but Complete Random Access Machine)



SCRAM–Befehle und Adressierungsarten

Code	Mnemonik	Zyklen	Bedeutung der Operation
\$ 00 00	HLT	4	Halt, Ende der Programmabarbeitung.
\$ m1 op	LDA *A*	5 6 8	Laden des Akkumulators.
\$ m2 op	STA *A*	- 5 8	Speichern des Akkumulators.
\$ m3 op	ADD *A*	5 7 9	Addiere den Wert zum Akkumulator.
\$ m4 op	SUB *A*	5 7 9	Subtrahiere den Wert vom Akkumulator.
\$ m5 op	MUL *A*	5 7 9	Multipliziere Akkumulator mit dem Wert.
\$ m6 op	DIV *A*	5 7 9	Dividiere Akkumulator durch den Wert.
\$ m7 op	AND *A*	5 7 9	Bitweises UND des Akku. mit dem Wert.
\$ m8 op	OR *A*	5 7 9	Bitweises ODER des Akku. mit dem Wert
\$ m9 op	XOR *A*	5 7 9	Bitweises Exklusiv–ODER des Akku.
\$ 0A 00	NOT	5	Bitweise Negation des Akkumulators.
\$ 0B op	JMP *A*	4	Unbedingter Sprung an die Adresse *A*.
\$ 0C op	JEQ *A*	5	Bedingter Sprung an die Adresse *A*, wenn der Akkumulator den Wert Null enthält.
\$ 0D op	JNE *A*	5	Bedingter Sprung an die Adresse *A*, wenn der Akkumulator nicht den Wert Null enthält.
\$ 0F 00	NOP	4	Die “No Operation”.

Die Bezeichnung *op* beim Befehlscode steht dabei für die Operandenadresse, bzw. den unmittelbaren Wert. In der Spalte *Zyklen* steht die Anzahl der Taktzyklen bei unmittelbarer-, direkter- und indirekter Adressierung. Die mit *m* gekennzeichneten Bits enthalten die Adressierungsart.

Bit 15	14	Code	Adressierungsart	Schreibweise
0	0	0	Direkte Adressierung	LDA A
0	1	1	Unmittelbare Adressierung	LDA #A
1	0	2	Indirekte Adressierung	LDA @A

• AUFGABE 1

```
-- Achtung, nur Programmfragmente!

DATA Waggon ==  speise (kmh : nat)
                passagier (plaetze : nat, kmh : nat)

-- induzierte Signatur
-- Konstruktoren:
FUN speise : nat -> Waggon
FUN passagier : nat ** nat -> Waggon

-- Selektoren:
FUN kmh plaetze : Waggon -> nat

-- Diskriminatoren
FUN speise? passagier? : Waggon -> bool

FUN machschneller : nat -> Waggon -> Waggon
DEF machschneller (x)(w) ==
  IF speise?(w) THEN speise(kmh(w)+x) ELSE passagier(plaetze(w),kmh(w)+x) FI

FUN zugschneller : seq[Waggon] -> seq[Waggon]
DEF zugschneller == map(machschneller(10))

FUN schnellerals160 : seq[Waggon] -> seq[Waggon]
DEF schnellerals160 = filter(\\w.kmh(w) > 160)

FUN langsamster : seq[Waggon] -> nat
-- SPC langsamster(s) = x
-- PRE ::?(s)
DEF langsamster(s) == reduce (min,kmh(ft(s)))(map(kmh)(s))

FUN plaetze : seq[Waggon] -> nat
DEF plaetze(s) ==
 reduce(+,0)(map(plaetze)(filter(passagier?)(s)))
```

• AUFGABE 2

```
-- Achtung, nur Programmfragmente

DATA opkind == add sub mul div

DATA expr == num (val : nat)
             node (op : opkind, left : expr, right : expr)

-- eine sehr ineffiziente L"osung, daf"ur "ubersichtlich
FUN correct? : expr -> bool

DEF correct? (num(_)) == true
DEF correct? (node(sub,l,r)) ==
  IF not(correct?(l)) or not(correct?(r)) THEN false
  ELSE
      eval(l) >= eval(r) FI
DEF correct? (node(div,l,r)) ==
  IF not(correct?(l)) or not(correct?(r)) THEN false
  ELSE
      eval(r) != 0
-- alle anderen Operationen sind unkritisch
DEF correct? (node(_,l,r)) == correct?(l) and correct?(r)
```

```
FUN eval: expr -> nat
-- SPC eval(s) = v
-- PRE correct?(s)
-- POST s ‘‘=’’ v
DEF eval(num(v)) == v
DEF eval(node(op,l,r)) ==
    optofun(op)(eval(l),eval(r))

FUN optofun : opkind -> (nat ** nat -> nat)
DEF optofun (add) == +
DEF optofun (sub) == -
DEF optofun (mul) == *
DEF optofun (div) == /


FUN exprmap: (nat -> bool) -> (nat -> nat) -> expr -> expr
DEF exprmap (p)(f)(num(v)) == IF p(v) THEN num(f(v)) ELSE num(v) FI
DEF exprmap (p)(f)(node(op,l,r)) == node(op,exprmap(p)(f)(l),exprmap(p)(f)(r))

FUN squareevens : expr -> expr
FUN squareevens == exprmap(even?)(\\x.x*x)
```

• AUFGABE 3

```
-- Achtung, nur Programmfragmente!

DATA line == asFun (f : real -> real)
             asPair (a : real , b : real)

FUN apply : line ** real -> real
-- SPC apply(l,x) = y
-- PRE true
-- POST asFun?(l) => y = f(l)(x)
--      asPair?(l) => y = a(l)*x + b(l)
DEF apply (asFun(f),x) == f(x)
DEF apply (asPair(a,b),x) == a*x + b

-- Testet, ob ein Punkt auf einer gegebenen Geraden liegt.
FUN p : line ** real ** real -> bool
-- SPC p (l,x,y) = b
-- PRE  true
-- POST b <==> f(l)(x) = y, falls asFun?(l)
--              a(l)*x + b(l) = y, sonst
DEF p == apply(l,x) = y

FUN makeLine : real ** real ** real ** real -> line
-- SPC makeLine (x0,y0,x1,y1) == l
-- PRE x0 != x1
-- POST l = asPair(\frac{y_1 - y_0}{x_1 - x_0},y_0 -x_0(\frac{y_1 - y_0}{x_1 - x_0})
DEF makeLine (x0,y0,x1,y1) ==
  LET
     a == (y1 - y0 ) / (x1 - x0)
     b == y0 - (x0 * a)
  IN
     asPair (a,b)

FUN exchange : line -> line
```

```
-- SPC exchange (l) = lnew
-- PRE true
-- POST asFun?(l) => asPair?(lnew): lnew = makeLine(0,apply(l,0),1,(apply(l,1)))
--      asPair?(l) => asFun?(lnew): f(lnew) = \\x.a(l)*x + b(l)
-- oder
--  l = lnew, aber asFun?(l) <==> asPair?(lnew) bzw. \forall x : apply(l,x) = apply(lnew,x)
DEF exchange (asFun(f)) ==  makeLine(0,f(0),1,f(1))
DEF exchange (asPair(a,b)) == asFun(\\x.a*x + b)

%FUN online? : real ** real ** real **real **real **real -> bool
%-- SPC online(x0,y0,x1,y1,x2,y2) = b
%-- PRE x0 != x1 != x2 != x0
%-- POST b <==> p(makeLine(x0,y0,x1,x0),x2,y2)
%DEF online? (x0,y0,x1,y1,x2,y2) == p(makeLine(x0,y0,x1,x0),x2,y2)

FUN intersect? : line ** line -> bool
-- SPC intersect?(l1,l2) = b
-- PRE true
-- POST b <==> l1' = l2' (Ableitungen gleich, Steigungen gleich)
DEF intersect (l1,l2) ==
 LET l1pair == IF asFun?(l1) THEN exchange(l1) ELSE l1
     l2pair == IF asFun?(l2) THEN exchange(l2) ELSE l2
 IN
     a(l1pair) != a(l2pair)

FUN intersection : line ** line -> real ** real
-- SPC intersection (l1,l2) == (x,y)
-- PRE intersect?(l1,l2)
-- oder
-- PRE l1' != l2' (Ableitungen verschieden, d.h., Steigungen
-- verschieden, d.h. Geraden nicht parallel oder gleich)
-- POST apply(l1,x) = apply(l2,x)
-- oder
-- POST p(l1,x,y) AND p(l2,x,y)
DEF intersection (l1,l2) ==
 LET l1pair == IF asFun?(l1) THEN exchange(l1) ELSE l1
     l2pair == IF asFun?(l2) THEN exchange(l2) ELSE l2
     x   == (b(l2pair)-b(l1pair)) / (a(l1pair) - a(l2pair))
 IN
     (x,apply(l1,x))
```

• **AUFGABE 4**

1.

	a	b	c^	d^
K1	5	17	11	8
K2	15	16	12	11

2. Diese Unterscheidug gibt es nicht, alle Argumente werden Call-By-Value übergeben. In funktionalen Sprachen gibt es keine veränderbaren Variablen (referentielle Transparenz), daher auch keine veränderbaren Parameter. Würde dem Begriff der Funktion zuwiderlaufen.

• **AUFGABE 5**

1. Behauptung gilt. Beweis: $(n+2)^2 = n^2 + 4n + 4$ Z.z. $\exists c \in \mathbb{R}, n_0 \in \mathbb{N} : \forall n > n_0 : n^2 + 4n + 4 \leq cn^2$ Setze $c = 9$. Also $n^2 + 4n + 4 \leq 9n^2 \iff 4n + 4 \leq 8n^2 \iff 4n + 4 \leq 4n^2 + 4n^2$ Das ist offensichtlich wahr für $n \geq 1$, denn es gilt $4n \leq 4n^2$ und $4 \leq 4n^2$ bzw. $n \leq n^2$ und $1 \leq n^2$. Q. E. D.

2. $\mathcal{O}(f) * \mathcal{O}(g) \subseteq \mathcal{O}(f * g)$. Beweis: Sei $h \in \mathcal{O}(f) * \mathcal{O}(g)$ beliebig, aber fest. Es gibt dann eine Zerlegung $(*) h = f' * g'$, wobei $f' \in \mathcal{O}(f)$ und $g' \in \mathcal{O}(g)$. Dann gilt nach Definition:

   $\exists c_f \in \mathbb{R}, n_{0_f} \in \mathbb{N} : (1)\ \forall n > n_{0_f} : f'(n) \leq c * f(n)$ und $\exists c_g \in \mathbb{R}, n_{0_g} \in \mathbb{N} : (2)\ \forall n > n_{0_g} : g'(n) \leq c * g(n)$

Zu zeigen: $\exists c \in \mathbb{R}, n_0 \in \mathbb{N} : (2)\ \forall n > n_0 : h(n) \leq c * f(n) * g(n)$

Sei $c = c_f * c_g$ und $n_0 = max(n_{0_f}, n_{0_g})$, also z. z.: $\forall n > max(n_{0_f}, n_{0_g})\ h(n) \leq c_f * c_g * f(n) * g(n) \overset{(*)}{\Longleftrightarrow} f'(n) * g'(n) \leq c_f * c_g * f(n) * g(n) \Longleftrightarrow f'(n) * g'(n) \leq c_f * f(n) * c_g * g(n)$ Das gilt offensichtlich wegen $(1), (2)$ für $n > max(n_{0_f}, n_{0_g})$. Also ist nach Definition $h \in \mathcal{O}(f * g)$. Q. E. D.

3. Beobachtung: Obwohl `FOO` linearen Aufwand hat, hat es nur konstanten Aufwand bzgl. der Eingabegröße n, da es mit der Konstanten k aufgerufen wird.

   $\sum_{i=0}^{n-1}(c_{inc} + \sum_{j=0}^{i-1}(c_{inc} + c_{foo})) = \sum_{i=1}^{n}(c_{inc} + \sum_{j=1}^{i}(c_{inc} + c_{foo})) = \sum_{i=1}^{n} c_{inc} + \sum_{i=1}^{n}\sum_{j=1}^{i}(c_{inc} + c_{foo}) = c_{inc}n + \sum_{i=1}^{n}\sum_{j=1}^{i}(c_{inc} + c_{foo}) = c_{inc}n + \sum_{i=1}^{n} i(c_{inc} + c_{foo}) = c_{inc}n + (c_{inc} + c_{foo})\sum_{i=1}^{n} i = c_{inc}n + (c_{inc} + c_{foo})\frac{(n^2+n)}{2} = c_{inc}n + \frac{c_{inc}+c_{foo}}{2}(n^2+n) = (c_{inc} + \frac{c_{inc}+c_{foo}}{2})n + \frac{c_{inc}+c_{foo}}{2}n^2 \overset{(9)}{\in} \mathcal{O}(n^2)$

4. Die innere Schleife würde trotz der Verschachtelung nur eine konstante Anzahl mal durchlaufen werden. Da `FOO` linear ist und mit k aufgerufen wird, wäre der Algorithmus linear.

• AUFGABE 6

```
DEFINITION MODULE Stack;

TYPE Stack;

PROCEDURE Push (VAR s : Stack; num,pri : CARDINAL);
(* PRE num kommt noch nicht im Stack vor *)

PROCEDURE Pop (VAR s : Stack; VAR num,pri : CARDINAL);
(* PRE Stack hat mindestens ein Element *)

PROCEDURE Promote (VAR s : Stack; num : CARDINAL);
(* PRE num kommt im Stack vor *)

END Stack.

======================================================================
IMPLEMENTATION MODULE Stack;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM InOut IMPORT WriteLn, WriteString, WriteCard;

TYPE Stack = POINTER TO StackElement;

   StackElement = RECORD
              num,pri : CARDINAL;
              next : Stack;
            END; (* RECORD *)

PROCEDURE Push (VAR s : Stack; num,pri : CARDINAL);
VAR current,prev : Stack;
 done : BOOLEAN;
BEGIN
 done := FALSE;
IF s = NIL THEN (* Puffer ist ganz leer *)
  NEW(s);
  s^.pri := pri;
  s^.num := num;
  s^.next:=NIL;
  done:=TRUE;
ELSE
  current := s;
  prev:=NIL;
```

```
  WHILE (current # NIL) AND (NOT(done)) DO
    IF (current^.pri > pri) THEN (* Element muss weiter nach hinten *)
      prev:=current;
      current := current^.next;
    ELSE (* Stelle gefunden *)
      IF prev = NIL THEN (* gleich vor das erste! *)
        NEW (prev);
        prev^.pri := pri;
        prev^.num := num;
        prev^.next := current; (* Es gilt current = s *)
        s:=prev;
      ELSE (* Wir sind mitten drin *)
        NEW (prev^.next);
        prev^.next^.pri := pri;
        prev^.next^.num := num;
        prev^.next^.next := current;
      END; (* IF *)
      done := TRUE;
    END; (* IF *)
  END; (* WHILE *)
 END; (* IF *)
 IF NOT(done) THEN (* hinter das letzte *)
   NEW(current);
   current^.pri := pri;
   current^.num := num;
   current^.next:=NIL;
   prev^.next := current;
 END; (* IF *)
END Push;

PROCEDURE Pop (VAR s : Stack; VAR num,pri : CARDINAL);
VAR save : Stack;
BEGIN
 num:=s^.num;
 pri:=s^.pri;
 save := s;
 s:=s^.next;
 DISPOSE(save);
END Pop;

PROCEDURE Promote (VAR s : Stack; num : CARDINAL);
VAR current, prev : Stack;
BEGIN
 IF s^.num = num THEN (* Es ist sowieso schon das erste Element, *)
                      (* Verschieben ist nicht notwendig *)
    s^.pri := s^.pri + 1;
  ELSE (* zuerst das Element entfernen *)
    prev := s;
    current := s^.next;
    WHILE (current^.num # num) DO (* Element suchen *)
      prev := current;
      current := current^.next
     END; (* WHILE *)
    prev^.next := current^.next;  (* Das Element "uberspringen *)
    Push (s,current^.num,current^.pri+1); (* Das Element mit neuer *)
                                          (* Prioritaet pushen *)
    DISPOSE (current);   (* Das alte loeschen *)
  END; (* IF *)
```

```
END Promote;
END Stack.
```

• AUFGABE 7

1. Spezifikation:

   `PROCEDURE Find (a : ARRAY OF CARDINAL; x : CARDINAL; VAR r : CARDINAL)`

   `PRE` $\exists i \in \{0, \dots, HIGH(a)\} : A[i] > x$

   `POST` $r = A[j], j \in \{0, \dots, HIGH(a)\} \land A[j] > x \land \forall i \in \{0, \dots, j-1\} : A[i] \leq x$

2. Invariante $I \equiv \forall i \in \{0, \dots j-1\} : A[i] \leq x$. Bisher (alle mit Index $< j$) waren alle überpfrüften Elemente kleiner oder gleich x.

3. Implementierung

```
PROCEDURE Find (a : ARRAY OF CARDINAL; x : CARDINAL; VAR r : CARDINAL);
VAR j : CARDINAL;
BEGIN
 j:=0;
 WHILE (A[j] <= x) DO
   j:=j+1;
  END; (* WHILE *)
 r:= A[j];
END Find;
```

4. Korrektheit

   PRE $\{0 = 0\}$

   `j:=0;`

   $\{j = 0\}$ (Zuweisungsaxiom)

   $\{I\}$

   PRE

   $\{j \leq HIGH(a)\}$ (Folgerung aus PRE und $j = 0$)

   `WHILE (A[j] <= x) DO`

   $\{A[j] \leq x\}$ (Schleifenregel, Abbruchbedingung)

   $\{I\}$

   $\{\forall i \in \{0, \dots, j\} : A[i] \leq x\}$

   $\{PRE\}$

   $\{\exists i \in \{j+1, \dots, HIGH(a)\} : A[j] > x\}$ (Folgerung aus PRE und I)

   `j := j + 1;`

   $\{I\}$ (Zuweisungsaxiom)

   $\{\exists i \in \{j, \dots, HIGH(a)\} : A[j] > x\}$ (Folgerung)

   $\{j \leq HIGH(a)\}$ (Folgerung aus PRE)

   `END; (* WHILE *)`

   $\{A[j] > x\}$ (Schleifenabbruch)

   $\{j \leq HIGH(a)\}$

   $\{I\}$

   `r := A[j];`

   $\{r = A[j]\}$ (Zuweisungsaxiom)

$\{r > x\}$ (Zuweisungsaxiom)

$\{POST\}$ (Folgerung)