Name: ....

Matr.-Nr. ....

Hinweise zu den OPAL-Aufgaben

Lesen Sie die Aufgaben jeweils bis zum Ende durch; oft gibt es hilfreiche Hinweise!

Sie brauchen nichts abzuschreiben, was schon auf dem Aufgabenblatt steht. Insbesondere brauchen Sie keine Funktions- oder Typdeklarationen und auch keine Spezifikationen abzuschreiben.

Andererseits sollen Sie für sämtliche Funktionen, die Sie neu einführen, Deklaration und Spezifikation mit aufschreiben.

Beachten Sie, daß vorgegebene Namen und Funktionalitäten nicht verändert werden dürfen.

Damit Sie nicht soviel in Ihren Unterlagen blättern müssen: Hier Auszüge aus wichtigen Strukturen.

```
SIGNATURE BOOL
SORT bool
FUN true false : bool                            -- Konstanten
FUN ~          : bool -> bool                    -- Negation
FUN and or     : bool ** bool -> bool            -- Und-, Oderverknüpfung
FUN = |=       : bool ** bool -> bool            -- Gleich, Ungleich
```

```
SIGNATURE Nat
SORT nat
FUN 0 1 2 3 ... 32 100 1000 10000: nat           -- Konstanten
FUN + - * / % : nat ** nat -> nat                -- Grundrechenarten
FUN < <= = >= |= : nat ** nat -> bool            -- Vergleiche
FUN min max: nat ** nat -> nat                   -- Minimum/Maximum zweier Zahlen
```

```
SIGNATURE Real
SORT real
FUN 0 1 2 3 ... 32 100 1000 10000 : real         -- Konstanten
FUN + - * / : real ** real -> real               -- Grundrechenarten
FUN < <= = >= |= : real ** real -> bool          -- Vergleiche
FUN min max: real ** real -> real                -- Minimum/Maximum zweier Zahlen
FUN floor: real -> real                          -- Nachkommastellen abschneiden
```

```
SIGNATURE Denotation
SORT denotation
FUN ++ : denotation ** denotation -> denotation -- Verketten
FUN # : denotation -> nat                       -- Länge
FUN ! : denotation ** nat -> char               -- Zugriff durch Index
```

```
SIGNATURE NatList       -- für nat kann auch eine andere Sorte stehen
TYPE natList == empty
                :: (ft: nat, rt: natList)

FUN ++ : natList ** natList -> natList           -- Verketten
FUN #  : natList -> nat                          -- Länge
```

```
SIGNATURE NatListMap
FUN map: (nat -> nat) -> natList -> natList
```

```
SIGNATURE NatListFilter
FUN filter: (nat -> bool) -> natList -> natList
```

```
SIGNATURE NatListReduce
FUN reduce: (nat ** nat -> nat) ** nat -> natList -> nat
```

```
SIGNATURE NatListZip
FUN zip: (nat ** nat -> nat) -> natList ** natList -> natList
```

• AUFGABE 1 (9 Punkte) Funktionale Programmierung: Teilbarkeit.

Die Teiler einer natürlichen Zahl n sind alle kleineren Zahlen, durch die sich n ohne Rest teilen läßt; die 1 zählt mit.

Beispiel: $teiler(6) = \{1, 2, 3\}$ und $teiler(16) = \{1, 2, 4, 8\}$

Die Teilersumme ist die Summe der Teiler, also $teilersumme(6) = 6$ und $teilersumme(16) = 15$.

1. (1 Punkt) Erläutern Sie den Begriff „Funktionalität“
2. (1 Punkt) Definieren Sie eine Funktion, die bestimmt, ob eine Zahl m Teiler einer Zahl n ist.

```
FUN teilt: nat ** nat -> bool
-- SPC teilt(t, n)
-- PRE t0
-- POST teilt(t, n) <=> t ist Teiler von n
```

3. (2 Punkte) Definieren Sie eine Funktion, die die Teiler einer natürlichen Zahl als natList bestimmt.

```
FUN teiler: nat -> natList
-- SPC  teiler(n) == nL
-- PRE  n ≥ 2
-- POST t < n und teilt(t, n) ==> t in nL
```

4. (2 Punkte) Deklarieren und definieren Sie eine Funktion `teilersumme`, die die Teilersumme einer natürlichen Zahl bestimmt.
5. (3 Punkte) Deklarieren und definieren Sie eine Funktion `zerlegung`, die der folgenden Spezifikation genügt. Dabei müssen höherwertige Listenfunktionen `map` und/oder `filter` und/oder `reduce` verwendet werden.

```
FUN zerlegung: natList -> natList ** natList ** natList
-- SPC zerlegung(nL) == (ab, vk, df)
-- PRE true
-- POST ab ist die Liste aller Zahlen n in nL mit teilersumme(n) > n,
        vk ist die Liste aller Zahlen n in nL mit teilersumme(n) = n,
        df ist die Liste aller Zahlen n in nL mit teilersumme(n) < n
```

• AUFGABE 2 (16 Punkte) Funktionale Programmierung: Computer Aided Burglary.

Auch Diebe gehen mit der Zeit und verwenden Computer. Wir kümmern uns hier um das Problem, welche Gegenstände gestohlen werden sollen. Da niemand beliebig schwere Dinge tragen kann, trifft man anhand des Gewichts und des Wertes eine Auswahl.

Typische Dinge sind	Goldmünze	0,031 kg	978,45 DM
	Hifi-Anlage	10,000 kg	3499,98 DM
	Notebook	2,800 kg	4995,00 DM
	„Schildkrötenbuch"	1,300 kg	85,14 DM
	...		

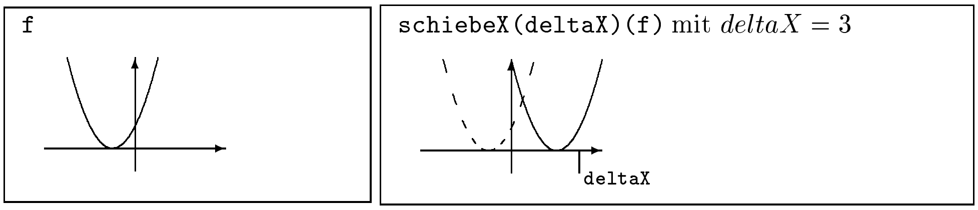
1. (1 Punkt) Geben Sie eine Typdefinition für den Typ `ding` an.
2. (1 Punkt) Analog zum Typ `natList` verwenden wir im folgenden einen Typ `dingList`. Geben Sie (in Opal-Syntax) eine Konstante vom Typ `dingList` mit mindestens drei Gegenständen an.
3. (2 Punkte) Definieren Sie eine Funktion, die einer `dingList` ihr Gesamtgewicht zuordnet.
4. (2 Punkte) Definieren Sie eine Funktion, die einer `dingList` ihren Gesamtwert zuordnet.
5. (2 Punkte) Definieren Sie eine Funktion, um das durchschnittliche Verhältnis von Wert zu Gewicht der Elemente einer `dingList` zu bestimmen. Geben Sie zu dieser Funktion auch eine Spezifikation an!
6. (2 Punkte) Definieren Sie eine Funktion, die aus einer `dingList` alle Elemente entfernt, deren Verhältnis Wert zu Gewicht kleiner als das durchschnittliche Verhältnis ist.
7. (3 Punkte) Definieren Sie eine Funktion
   `FUN auswahl:  real ** dingList -> dingList`,
   die aus einer `dingList` von Anfang an der Reihe nach alle Elemente bestimmt, die man mitnehmen kann, ohne das vorgegebene Gewicht zu überschreiten.
8. (3 Punkte) Definieren Sie eine Funktion
   `FUN zweites:  dingList -> ding`,
   die aus einer `dingList` das Element mit dem *zweithöchsten* Wert bestimmt. Geben Sie zu dieser Funktion auch eine Spezifikation an! (Hilfestellung: Sie dürfen in der Vorbedingung verlangen, daß alle Elemente paarweise unterschiedliche Werte haben.)

Hinweis: Sie können in dieser Aufgabe ein Map-Funktional der Funktionalität
`map:  (ding -> real) -> dingList -> realList`
verwenden.

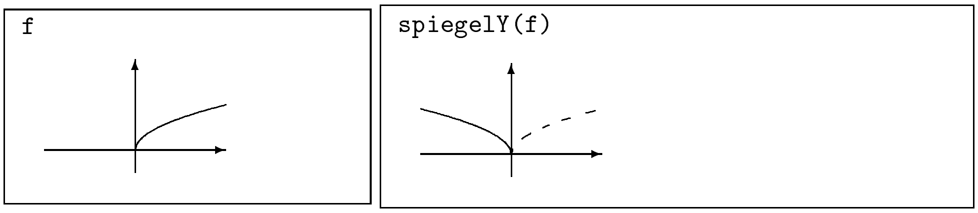
• **AUFGABE 3 (8 Punkte) Funktionale Programmierung: Funktionen höherer Ordnung.**

Wir betrachten reellwertige Funktionen vom Typ `real -> real`. Wir können aus vorgegebenen Funktionen neue Funktionen konstruieren durch Verschieben, durch Spiegeln und durch Verknüpfen.

1. (1 Punkt) Erläutern Sie den Begriff „Funktion höherer Ordnung".
2. (2 Punkte) Deklarieren und definieren Sie ein Funktional `schiebeX`, das eine Funktion f um `deltaX` Einheiten nach rechts verschiebt.



3. (2 Punkte) Deklarieren und definieren Sie ein Funktional `spiegelY`, das eine Funktion f an der y-Achse spiegelt.



4. (3 Punkte) Deklarieren Sie ein Funktional `verknuepfe`, das zwei reellwertige Funktionen mit Hilfe einer weiteren Funktion verknüpft. Die Definition sieht so aus:

```
DEF verknuepfe(f, g)(o)(x) = f(x) o g(x)
```

Bestimmen Sie, ob folgende Ausdrücke korrekt sind, und geben Sie den Typ der korrekten Ausdrücke an!

(a) `verknuepfe(square, cube)`

(b) `verknuepfe(<, =)(*)`

(c) `verknuepfe(floor, floor)(*)(pi)`

Dabei seien `<`, `=` und `*` die üblichen Funktionen auf dem Datentyp `real`, außerdem:

```
FUN square: real -> real
FUN cube:   real -> real
FUN floor:  real -> real
FUN pi: real
```

• AUFGABE 4 (10 Punkte) Imperative Programmierung / Hoarekalkül.

1. (3 Punkte) Erläutern Sie:
   (a) Im Rahmen des Hoare-Kalküls spricht man von Zuweisungs**axiom** jedoch von Schleifen**regel**. Warum diese Unterscheidung?
   (b) Eine **Zusicherung** ist eine Aussage im Sinne der mathematischen Logik. Worüber genau macht eine Zusicherung eine Aussage?
   (c) Warum nennt man ein partiell korrektes Programm überhaupt noch korrekt.
2. (3 Punkte) Spezifizieren Sie formal die Vor- und Nachbedingung einer Prozedur, die folgenden Prozedurkopf haben soll `PROCEDURE Permute(l : CARDINAL ; a :ARRAY OF CHAR; p : ARRAY OF CARDINAL; VAR erg: ARRAY OF CHAR);`. Die gegebenen Arrays haben dabei alle die Länge `l`. Die Prozedur berechnet `a` unter der Permutation `p`. D.h. Im Ergebnisarray `erg` sollen die Elemente von `a` an der jeweils an gleicher Position in `p` angegebenen Stelle erscheinen.
   BSP: Aus: `a = [b,x,r]` und `p = [2,0,1]` würde sich: `erg = [x,r,b]` ergeben.
3. (4 Punkte) Entwickeln und verifizieren Sie mit Hilfe Ihrer Vor- und Nachbedingung ein Modula-2 Programm, das Ihre Spezifikation erfüllt.

• **AUFGABE 5 (3 Punkte) O-Kalkül.**

1. (1 Punkt) Die Algorithmen `Quicksort` und `Insertionsort` gehören beide zur Aufwandsklasse $O(n^2)$. Warum wird trotzdem `Quicksort` in der Regel als "schneller" bezeichnet?
2. (2 Punkte) Zeigen oder widerlegen Sie: $(3n - 27)^2 \in O(n^2)$

• AUFGABE 6 (10 Punkte) Prozeduren, Parameterübergabe in Modula-2.

Gegeben ist ein Hauptmodul, das mehrere Prozeduren und Funktionsprozeduren aufruft.

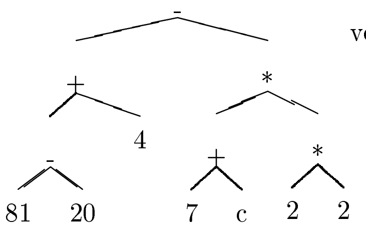
```
MODULE foo;

TYPE INTPTR = POINTER TO INTEGER;

VAR x: INTPTR;  y, z: INTEGER;

PROCEDURE P1(VAR y: INTEGER; z: INTPTR);
BEGIN
  NEW(z);
  z^ := 2 * y;
  y := y * z^;
END P1;

PROCEDURE P2(VAR x: INTEGER) : INTEGER;
BEGIN
  x := x + 1;
  z := x * z;
  RETURN x * z;
END P2;

PROCEDURE P3(z: INTEGER; VAR y: INTEGER);
BEGIN
  y := P2(z);
END P3;

BEGIN
  y := 3;
  P1(y, x);
  z := 20 - y;
  (* Punkt A *)
  NEW(x);
  x^ := P2(z);
  (* Punkt B *)
  z := z DIV 2;
  P3(z, y);
  (* Punkt C *)
END foo.
```

1. (6 Punkte) Geben sie an, welche Werte die globalen Variablen `x`, `y` und `z` an den mit „Punkt A“, „Punkt B“ und „Punkt C“ markierten Stellen annehmen bzw. welche Werte dereferenziert werden.
2. (2 Punkte) Wie wird die Zuweisung an das globale Objekt `z` in der Prozedur P2 bezeichnet?
3. (1 Punkt) Beschreiben sie, als was für eine Art von Parameter bzw. Variable der Bezeichner `z` verwendet wird.
4. (1 Punkt) Werden Zeiger als *call by value* oder als *call by reference* übergeben?

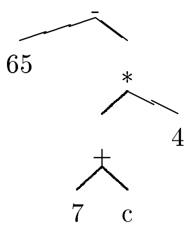
• AUFGABE 7 (10 Punkte) Imperative Programme, Rekursive Datenstrukturen.

Gesucht sind eine Datenstruktur zur Darstellung von arithmetischen Ausdrücken als Binärbäume sowie eine Prozedur, die den arithmetischen Ausdruck vereinfacht. Als Operatoren sind die vier Grundrechenarten (mit Ganzzahldivision) und als Operanden ganzzahlige Konstanten sowie *Variablen* zuzulassen. Bei der Vereinfachung sind alle konstanten Teilbäume auszuwerten und durch das Ergebnis der Auswertung zu ersetzen.

Z.B. wird



vereinfacht zu



1. (2 Punkte) Andere Programmiersprachen sehen parametrisierbare Typdeklarationen vor. Modula-2 fehlt dieses Konstrukt. Welche Vor- oder Nachteile ergeben sich dadurch beim Entwurf von Modula-2 Programmen?
2. (2 Punkte) Geben Sie ein geeignetes `MODULA-2` Definitionsmodul an, das nur den Typ des Binärbaumes opak und die Prozedur `Simplify` exportiert.
3. (6 Punkte) Geben Sie zum Definitionsmodul ein Implementierungsmodul an, das den Binärbaum und die Prozedur implementiert. Es ist darauf zu achten, daß bei der Vereinfachung des Baumes nicht benötigter Speicher wieder freigegeben wird. Gehen Sie davon aus, daß ein korrekt aufgebauter Binärbaum vorliegt.

• AUFGABE 8 (10 Punkte) Miniassembler.

Schreiben Sie ein Assembler–Programm, daß die Anzahl ANZ der Elemente eines nullterminierten Feldes FELD liefert, die durch den Wert WERT ganzzahlig teilbar sind.

Hinweise:

- Das in Ihrem Assembler–Programm gewählte Beispielfeld sollte mindestens eine Länge von vier Elementen haben.
- Ein Element ist ganzzahlig durch einen Wert teilbar, wenn (Element / Wert) * Wert wieder Element ergibt.
- Ihr Programm soll dabei ausschließlich die mnemonischen Befehle und Adressierungsmodi verwenden, die in den unten aufgeführten Tabellen zu finden sind (Übungsassembler).
- Ihr Programm soll dreispaltig angelegt werden: 1. Spalte Labels, 2. Spalte Mnemonics, 3. Spalte Kommentare (siehe das Beispiellayout im Anschluß).

```
          ORG 0            ; Denken Sie daran, daß Sie
          LDA #string      ; das Assemblerprogramm
          STA stringPtr    ; sinnvoll kommentieren sollen.
NEXT      LDA result       ; Ohne Komentare, wäre ein
          JEQ DONE         ; längeres Assemblerprogramm sehr
          SUB #10          ; schwer zu verstehen.
          JMP NEXT         ; Außerdem gibt es ohne Kommentare
DONE      HLT              ; erheblich weniger Punkte!
result    DAT 0            ; Falls das Programm nicht korrekt
stringPtr RES 1            ; ist, kann man meistens anhand
string    DAT 48,29,52,0   ; der Kommentare Flüchtigkeits-
          END              ; fehler erkennen.
```

Assemblerbefehle

Mnemonik	Bedeutung der Operation
HLT	Halt, Ende der Programmabarbeitung
LDA M	Lade den Akkumulator mit dem Wert an Adresse M
STA M	Speichere Akku–Inhalt nach Adresse M
ADD M	Addiere Operand zum Akkumulator
SUB M	Subtrahiere Operand vom Akkumulator
MUL M	Multipliziere Operand durch Akkumulator
DIV M	Dividiere Operand durch Akkumulator
JMP M	Springe zu Adresse M
JEQ M	Springe zu Adresse M, wenn der Akkumulator–Inhalt der Wert 0 ist, ansonsten führe den nächsten Befehl aus.
JNE M	Springe zu Adresse M, wenn der Akkumulator–Inhalt ungleich 0 ist, wenn nicht führe den nächsten Befehl aus.

Pseudoassemblerbefehle

Mnemonik	Bedeutung
ORG s	Definiert die Startadresse des Programms an Adresse s.
DAT d	Erzeugt eine Dualzahl, die der Dezimalzahl d entspricht.
RES n	Reserviert n Speicherzellen (Bytes) mit undefiniertem Wert
END	Zeigt das Ende des Assemblerprogramms an.

Adressierungsmodi am Beispiel des Befehls LDA (andere Befehle analog).

Mnemonik	Bedeutung
LDA A	Lade den Akkumulator mit dem Wert an Adresse A
LDA @ A	Lade den Wert in den Akkumulator, der an der Speicherstelle steht, auf die der Inhalt der Adresse A verweist.
LDA # A	Lade den Akku. mit dem Wert A (Direktwert) bzw. mit der Adresse einer Speicherstelle mit dem Labelnamen A.

• **AUFGABE 9 (6 Punkte) Entwurf eines Schaltnetzes.**

Entwerfen Sie ein Schaltnetz mit zwei Dateneingängen A und B und zwei Steuereingängen S und T, so daß je nach Zustand der Steuereingänge folgende Funktion am Ausgang Y anliegt:

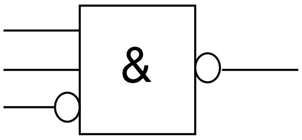
S	T	Y
0	0	A * B
0	1	B
1	0	A
1	1	egal

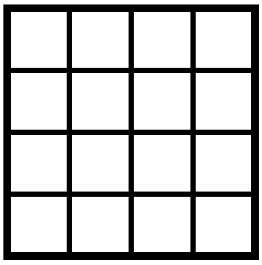
Verwenden Sie dazu folgende Entwurfsmethode:

1. (3 Punkte) Minimierung der Gleichung für Y mittels KV–Diagramm.
2. (2 Punkte) Zeichnen der Schaltung.
3. (1 Punkt) Geben Sie an, welche Funktion das Schaltnetz bei ST = 11 hat.

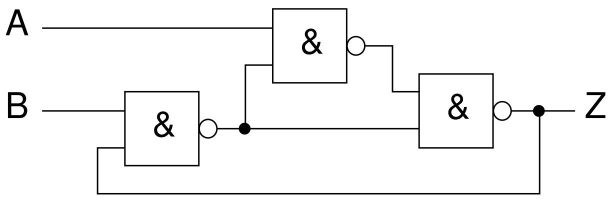
Hinweis:

In der Zeichnung sind auch Gatter mit drei Eingängen möglich, außerdem dürfen die Eingangssignale direkt am Eingang negiert werden:





• **AUFGABE 10 (7 Punkte) Schaltwerke.**



Analysieren Sie das angegebene Schaltwerk nach folgender Vorgehensweise:

1. ($\frac{1}{2}$ Punkt) Auftrennen der Rückkopplung, Zeichnen des sich ergebenden Schaltnetzes, Einzeichnen der Zwischenpunkte C und D.
2. ($\frac{1}{2}$ Punkt) Aufstellen der Gleichungen für C, D und Z+.
3. (2 Punkte) Aufstellen der Wertetabelle für das Schaltnetz.
4. (2 Punkte) Untersuchung der Zustände auf Instabilitäten.
5. (2 Punkte) Zusammenfassen der Zustände zu einer Wertetafel für das Ausgangsschaltwerk.

• AUFGABE 11 (10 Punkte) Entwurf eines Zählers.

Entwerfen Sie einen Zähler mit 3 JK–Flipflops, der an den Q–Ausgängen der JK–Flipflops die Zählfolge 0, 1, 7, 2, 3, 0, 1, ... in binärer Kodierung zählt. Verwenden Sie dazu folgende Entwurfsmethode:

1. (1 Punkt) Kodierung und Benennung der Zustände.
2. (2 Punkte) Erstellen der Wertetafel.
3. (3 Punkte) Bestimmung der JK–Flipflop–Ansteuerung.
4. (4 Punkte) Minimierung der JK–Ansteuerung mittels KV–Diagramm.

Hinweise:

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