Informatik 1

Prüfung im Sommersemester 2001 - Musterlösung

- Aufgaben sind in Times- *Times-Kursiv*-, `Courier`- und **`Courier-Fett`**-,
  Lösungen in blauer Monotype-Corsiva-Schrift gehalten.

Fachhochschule Reutlingen, Hochschule für Technik und Wirtschaft
Fachbereich Elektrotechnik und Maschinenbau

Prüfungsfach/Studiengang/Semester:	Informatik 1 in Elektronik 1
Prüfer:	Prof. Dr. Karlheinz Hug
Prüfungsdauer:	120 Minuten
Zugelassene Hilfsmittel:	Alle (Vorlesungsmitschrift, Literatur,...)
Anzahl der Aufgaben- und Lösungsblätter:	11

Aufgabe	**Punkte**		**Aufgabe**	**Punkte**	
	möglich	**erreicht**		**möglich**	**erreicht**
1	10		6	19	
2	18		7	20	
3	19		8	15	
4	18		9	9	
5	12				
Summe möglich:		**140**	**Summe erreicht:**		

- Zum Bestehen der Prüfung sind 60, für die Note „Eins“ 120 Punkte erforderlich.
- Bearbeiten Sie die Aufgaben auf ggf. dazu freigelassenen Stellen! Tragen Sie in Tabellen und an punktierten Stellen ... im Text passende Angaben ein! (Ausgenommen die Punktzahl jeweils rechts oben!)
- Falls der Platz nicht reicht, verwenden Sie die vorangehenden Blattrückseiten!

Viel Erfolg!

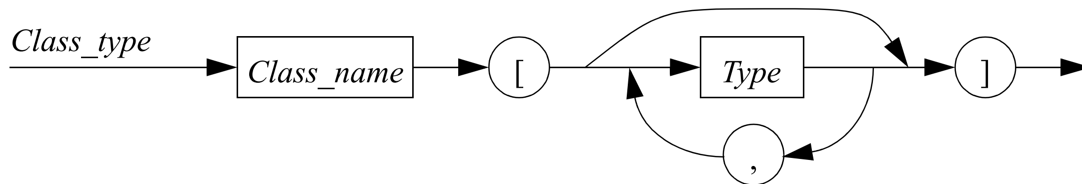
Aufgabe 1: Spezifikation durch Vertrag: Menge (Punkte: 10 |..........)

Ergänzen Sie die Spezifikation des Moduls, das eine Menge im Sinne der Mengenlehre modelliert, mittels Nachbedingungen und Invarianten! `Element` ist ein beliebiger Elementtyp. Die Aktionen sind idempotent, d.h. der Effekt von `Put(x); Put(x)` ist gleich dem von `Put(x)`; der von `Remove(x); Remove(x)` ist gleich dem von `Remove(x)`; der von `WipeOut; WipeOut` ist gleich dem von `WipeOut`.

```
MODULE Set
  QUERIES
    IsEmpty : BOOLEAN                     (* Enthält die Menge kein Element? *)
    Count : INTEGER                       (* Wieviele Elemente enthält die Menge? *)
    Has (IN x : Element) : BOOLEAN        (* Enthält die Menge x? *)
      POST
        result IMPLIES (Count > 0)
  ACTIONS
    Put (IN x : Element)                  (* Füge x zur Menge hinzu. *)
      POST
        Has (x)
        NOT OLD (Has (x)) = (Count = OLD (Count) + 1)
        OLD (Has (x)) = (Count = OLD (Count))
    Remove (IN x : Element)               (* Entferne x aus der Menge. *)
      POST
        NOT Has (x)
        NOT OLD (Has (x)) = (Count = OLD (Count))
        OLD (Has (x)) = (Count = OLD (Count) - 1)
    WipeOut                               (* Entferne alle Elemente aus der Menge. *)
      POST
        Count = 0
  INVARIANTS
    Count >= 0
    IsEmpty = (Count = 0)
END Set
```

Aufgabe 2: Syntaxbeschreibung (Punkte: 18 |..........)

Stellen Sie das Syntaxdiagramm



mit einer EBNF-Regel dar!

Class_type = *Class_name "[" [Type { "," Type }] "]" .*

Gegeben ist die Syntax einer einfachen Kommandosprache in EBNF-Notation:

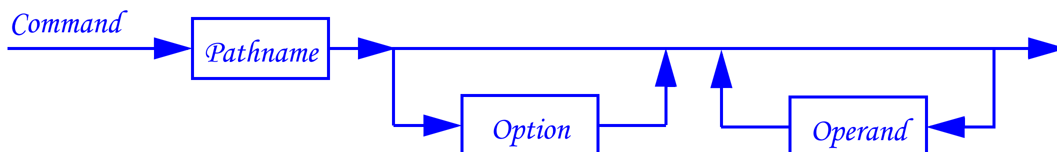
Command = *Pathname* [*Option*] { *Operand* }.

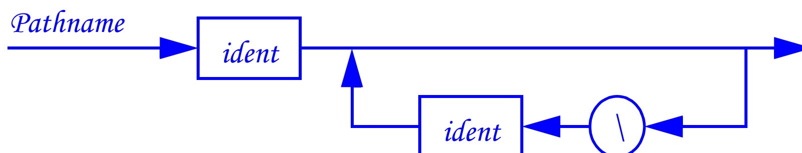
Pathname = *ident* { „\" *ident* }.

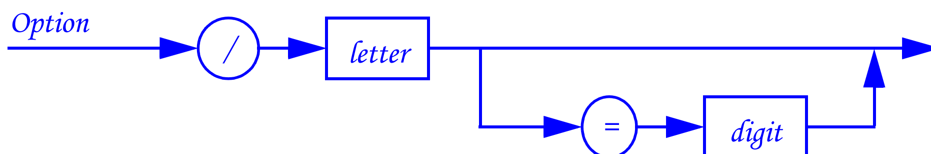
Option = „/" *letter* [„=" *digit*].

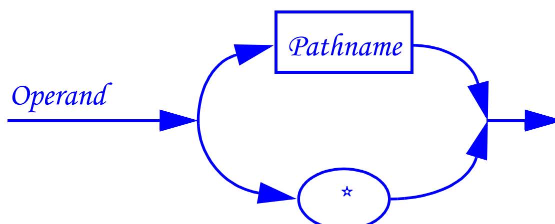
Operand = *Pathname* | „*".

Zeichnen Sie zu diesen vier Regeln äquivalente Syntaxdiagramme!









Ein Wort dieser Sprache ist z.B.: `COPY /C=2 Stud\Mod *`

Geben Sie eine Ableitung für dieses Wort an! (Es genügen fünf Hauptschritte, die jeweils mehrere einzelne Ersetzungen umfassen können.)

Command → Pathname [Option] { Operand }

→ "COPY" Option { Operand }

→ "COPY" "/C=2" Operand { Operand }

→ "COPY" "/C=2" Pathname { Operand }

→ "COPY" "/C=2" "Stud\Mod" Operand { Operand }

→ "COPY" "/C=2" "Stud\Mod" ""*

*→ "COPY /C=2 Stud\Mod *"*

Aufgabe 3: Sprachkonstrukte, Typregeln

(Punkte: 19 |..........)

Begründen Sie zu folgenden Aussagen, warum sie richtig bzw. falsch sind!

Aussage	Begründung / Bedingung
Der Typ einer Variable ist zur Laufzeit veränderbar.	❒ Richtig ☒ Falsch, weil *der Typ einer Variable bei ihrer Vereinbarung angegeben wird und daher zur Übersetzungszeit festgelegt ist.*
Typsicherheit bedeutet, dass jeder Typ sicher gegen Angriffe von Viren geschützt ist.	❒ Richtig ☒ Falsch, weil *Typsicherheit bedeutet, dass jede typisierte Größe nur Werte aus dem Wertebereich ihres Typs annehmen kann. Sicherheit vor Virenangriffen ist etwas ganz anderes.*
Ein Aufruf einer Prozedur kann in einem Ausdruck vorkommen,...	..., wenn *es sich um eine Funktionsprozedur handelt.*

Übersetzen Sie den folgenden Abschnitt aus dem Component Pascal Language Report!

> *Expressions are constructs denoting rules of computation whereby constants and current values of variables are combined to compute other values by the application of operators and function procedures. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands.*

Ausdrücke sind Konstrukte, die Berechnungsvorschriften beschreiben, wobei Konstanten und aktuelle Werte von Variablen kombiniert werden, um andere Werte durch Anwendung von Operatoren und Funktionsprozeduren zu berechnen. Ausdrücke bestehen aus Operanden und Operatoren. Klammern können benutzt werden, um spezifische Verknüpfungen von Operatoren und Operanden auszudrücken.

Mit den Vereinbarungen

VAR

b : CHAR;
d . REAL;
e : ARRAY n OF SomeType;

besteht die Anweisung

```
IF b < 'c' THEN d := LEN (e) * 1.3 END
```

alle Prüfungen der Typverträglichkeit.

Geben Sie mit den Vereinbarungen

```
VAR b : BOOLEAN; c : CHAR; i : INTEGER; x : REAL;
```

zu jedem Teilausdruck des folgenden Ausdrucks an, von welchem Typ er ist und wo ein Wert implizit an einen anderen Typ angepasst wird, bzw. wo ein Typverträglichkeitsfehler vorliegt!

```
( x   +   2       >      i )    OR   ( b    =    c )
```

REAL *INT* *INT* *BOOL* *CHAR*

implizit angepasst an REAL

REAL

Typfehler! BOOL u. CHAR nicht vergleichbar!

implizit angepasst an REAL

BOOL

Aufgabe 4: Optimierung

(Punkte: 18 |..........)

Die gegebenen Programmstücke sind durch äquivalente Programmstücke zu ersetzen, d.h. ihre Semantik muss erhalten bleiben.

Vereinfachen Sie den folgenden booleschen Ausdruck so weit wie möglich!

```
((x > 6) OR ~(x < 7)) & ~(x <= 5) & (x <= 8)
```

((x > 6) OR (x >= 7)) & (x > 5) & (x <= 8)

(x > 6) & (x > 5) & (x <= 8)

(x > 6) & (x <= 8)

Transformieren Sie die folgende Anweisung in eine äquivalente Anweisungsfolge, die um sechs Größenordnungen schneller läuft!

```
FOR i := 1 TO 1000000 DO
  IF k = i THEN
    x :=  5 * k
  END;
  y := x + 8
END
```

Sechs Größenordnungen (!) schneller ist nur zu erreichen, wenn die Schleife eliminiert wird! In der Tat kann k = i höchstens einmal wahr sein. Die Zuweisung an y ist nur beim letzten Mal relevant, kann also aus der Schleife rausgezogen werden.

```
IF (1 <= k) & (k <= 1000000) THEN
  x := 5 * k
END;
i := 1000001;
y := x + 8
```

Transformieren Sie die folgende Anweisung in drei bis vier Schritten und/oder mittels einer Wertetabelle in eine äquivalente Zuweisung!

```
IF ~(a = TRUE) THEN
  c := ~FALSE OR a
ELSE
  IF ~b # FALSE THEN
    c := ~TRUE & b
  ELSE
    c := ~b & TRUE
  END
END
```

Schematische Lösung mit Wahrheitstabelle:

a	b	c
F	F	T
F	T	T
T	F	F
T	T	F

also: c := ~a

Schritt 1: Verknüpfungen mit Konstanten ersetzen:

```
IF ~a THEN
  c := TRUE
ELSE
  IF ~b THEN
    c := FALSE
  ELSE
    c := ~b
  END
END
```

Schritt 2: Negation sparen, Zweige vertauschen:

```
IF a THEN
   IF b THEN
      c := ~b
   ELSE
      c := FALSE
   END
ELSE
   c := TRUE
END
```

Schritt 3: Innere IF-Anweisung ersetzen:

```
IF a THEN
   c := FALSE
ELSE
   c := TRUE
END
```

Schritt 4: Äußere IF-Anweisung ersetzen: c := ~a

Aufgabe 5: Schleifenablauf und -terminierung

(Punkte: 12 |..........)

Verfolgen Sie jeweils drei Durchläufe der beiden Schleifen und begründen Sie zu jeder Schleife, warum sie abbricht bzw. warum nicht!

<table>
<tr><th colspan="2">Schleife 1</th><th colspan="2">Schleife 2</th></tr>
<tr><td colspan="2"><code>i := 100;
k := 2;
WHILE (i > 0) & (k > 1) DO
  i := i - k;
  k := k + 1
END</code></td><td colspan="2"><code>i := 2;
k := 0;
WHILE k <= 1 DO
  i := i DIV 2;
  k := k + i
END</code></td></tr>
<tr><td>i</td><td>k</td><td>i</td><td>k</td></tr>
<tr><td>100
98
95
91
...</td><td>2
3
4
5
...</td><td>2
1
0
0
...</td><td>0
1
1
1
...</td></tr>
<tr><td colspan="2">☒ Terminiert ☐ Läuft endlos, weil
i wegen i := i - k und k > 1 bei jedem Durchlauf kleiner wird, damit nach endlich vielen Durchläufen i <= 0 gilt und damit die Fortsetzungsbedingung (i > 0) & (k > 1) nicht mehr gilt.</td><td colspan="2">☐ Terminiert ☒ Läuft endlos, weil
nach dem zweiten Durchlauf die Werte von i und k gleich bleiben, nämlich i = 0 und k = 1, daher die Fortsetzungsbedingung k <= 1 immer gilt.</td></tr>
</table>

Aufgabe 6: Programmablaufverfolgung (Punkte: 19 |..........)

Untersuchen Sie das Programmstück wie unten beschrieben!

```
VAR row : ARRAY 6 OF INTEGER;
    hit : BOOLEAN; gap, i1, i2, a : INTEGER;
(* Initialisierung der Variable row. *)
hit := TRUE;
gap := LEN (row);
WHILE hit OR (gap > 1) DO
  hit := FALSE;
  gap := MAX (SHORT (ENTIER (gap / 1.3)), 1);
  FOR i1 := 0 TO (LEN (row) - 1) - gap DO
    i2 := i1 + gap;
    IF row [i1] > row [i2] THEN
      a        := row [i1];
      row [i1] := row [i2];
      row [i2] := a;
      hit      := TRUE
    END
  END (* 1 *)
END    (* 2 *)
```

Die folgende Tabelle ordnet jeder Variable eine Spalte zu. Der Programmablauf hat die Stelle (* 1 *) erreicht. Verfolgen Sie den Ablauf weiter und vervollständigen Sie die Tabelle, indem Sie jeweils die Zustände der Variablen bis zum Erreichen der Stelle (* 1 *) in eine neue Zeile eintragen, bis die Stelle (* 2 *) erreicht ist! (Streichen Sie ggf. Zwischenwerte der Variablen so, dass die Änderungen nachvollziehbar sind!)

row						hit	gap	i1	i2	a
[0]	[1]	[2]	[3]	[4]	[5]					
7	82	60	31	5	28					
5	28	60	31	7	82	~~T~~ ~~F~~ ~~T~~ T	~~6~~ 4	~~0~~ ~~1~~ 2	~~4~~ 5	~~7~~ 82
5	*7*	*60*	*31*	*28*	*82*	*~~F~~ T*	*3*	*~~0 1 2~~ 3*	*~~3 4~~ 5*	*28*
5	*7*	*28*	*31*	*60*	*82*	*~~F~~ T*	*2*	*~~0 1 2 3~~ 4*	*~~2 3 4~~ 5*	*60*
5	*7*	*28*	*31*	*60*	*82*	*F*	*1*	*~~0 1 2 3 4~~ 5*	*~~1 2 3 4~~ 5*	*60*

Aufgabe 7: Implementation von Funktionen zu Reihungen

(Punkte: 20 |..........)

Implementieren Sie die folgenden Funktionen!

Dabei ist `Any` ein beliebiger Typ und `CONST notFound = -1`.

```
PROCEDURE
  Count (IN x : ARRAY OF Any; item : Any) : INTEGER;
  (* Anzahl der Vorkommen von item in x.
     Postcondition:
       result is the number of indexes i with x [i] = item. *)
  VAR
    result,
    i : INTEGER;
BEGIN
  result := 0;
  FOR i := 0 TO LEN (x) - 1 DO
    IF x [i] = item THEN
      INC (result)
    END
  END;
  RETURN result
END Count;
```

```
PROCEDURE
  MaxIndexOf (IN x : ARRAY OF Any; item : Any) : INTEGER;
  (* Index des letzten Vorkommens von item in x, falls existent;
     sonst notFound.
     Postcondition: (result = notFound) OR
       (result is the largest index such that x [result] = item). *)
  result is the number of indexes i with x [i] = item. *)
  VAR
    i : INTEGER;
BEGIN
  i := LEN (x) - 1;
  WHILE (i >= 0) & (x [i] # item) DO
    DEC (i)
  END;
  ASSERT ((i < 0) OR (x [i] = item));
  IF i < 0 THEN
    RETURN notFound
  ELSE
    RETURN i
  END
END MaxIndexOf;
```

Aufgabe 8: Programmierfehlerquellen (Punkte: 15 |..........)

Die Funktion

```
PROCEDURE Mean (IN a: ARRAY OF REAL; len: INTEGER): REAL;
  VAR   sum : REAL;   i : INTEGER;
BEGIN
  FOR i := 1 TO len DO
    sum := sum + a [i]
  END;
  RETURN sum / len
END Mean;
```

soll den arithmetischen Mittelwert der Reihung `a` berechnen; `len` ist die Elementanzahl von `a`. Die Funktion ist syntaktisch korrekt und übersetzbar, enthält aber *fünf Mängel*, die sich als Ursachen für Laufzeitfehler erweisen können.

Spüren Sie die Schwachstellen auf, nennen Sie mögliche *Folgen* und geeignete *Änderungen*, um die Funktion zuverlässig zu gestalten!

Mangel	*Folge*	*Änderung*
(1) Zeile 1: Parameter len ist überflüssig, da die Länge von a mit der Funktion LEN abfragbar ist.	*Mögliche Inkonsistenz len # LEN (a) führt zu falscher Berechnung.*	*Auf len verzichten.*
(2) Zeilen 3 - 4: Initialisierung von sum fehlt.	*Falsches Ergebnis.*	*Initialisierung sum := 0.*
(3) Zeile 4: Falsche Untergrenze für Indexvariable i, Indexwert 0 fehlt.	*Falsches Ergebnis.*	*Initialisierung i := 0.*
(4) Zeile 4: Falsche Obergrenze für Indexvariable i, Indexwert len zu viel, falls len = LEN (a).	*Indexüberlauf bei a [i] für i = len führt zu Trap.*	*Obergrenze für i: len - 1 oder besser LEN (a) - 1.*
(5) Zeile 7: Division durch 0 möglich, wenn len = 0.	*Trap.*	*len # 0 prüfen oder garantieren, oder besser RETURN sum / LEN (a) einsetzen, da LEN (a) garantiert größer 0 ist.*

Aufgabe 9: Programmierrichtlinien (Punkte: 9 |..........)

Viele Bücher über Programmiertechnik enthalten Empfehlungen und Regeln für guten Programmierstil. Durch Ihre Programmiererfahrung haben Sie sich Leitlinien angeeignet, an die Sie sich selbst beim Programmieren halten.

Formulieren Sie drei Ihrer Programmierrichtlinien mit eigenen Worten und begründen Sie jeweils, welch gutem Zweck die Richtlinie dient!

Keine Musterlösung, da hier individuelle Antwort gefordert!