Hochschule Reutlingen - Reutlingen University
Fakultät Informatik
Studiengang Medien- und Kommunikationsinformatik
akkreditiert von der ASIIN

Informatik 1

mki-Bachelor 1
Alte Prüfungsaufgaben + Lösungen

Aufgaben sind in Times-,	*Lösungen in blauer Monotype-Corsiva-Schrift gehalten.*
Prüfer:	Prof. Dr. Karlheinz Hug
Prüfungsdauer:	120 Minuten
Zugelassene Hilfsmittel:	Alle (Literatur, Skripten, Übungsmaterial,...)
Umfang:	24 Aufgaben auf 21 Seiten

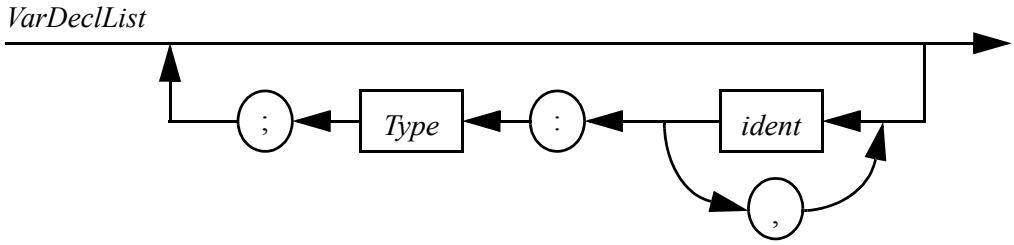
- Zum Bestehen der Prüfung sind **60**, für die Note „1.0“ **120** Punkte erforderlich.
- Lassen Sie die erhaltenen Blätter zusammengeheftet und geben Sie alle Blätter geheftet wieder ab, sonst droht Abzug von Punkten!
- Bearbeiten Sie die Aufgaben auf dazu freigelassenen Stellen! Tragen Sie in Tabellen und an punktierten Stellen ... im Text passende Angaben ein!
- Der Platz reicht normalerweise; vermeiden Sie Extrablätter!
- Teilaufgaben sind meist unabhängig voneinander lösbar.

✌ **Viel Erfolg!**

Aufgabe 1 — Syntax beschreiben

(Punkte: 4 |.....)
WS 05/06

Stellen Sie das folgende Syntaxdiagramm mit einer EBNF-Regel dar!

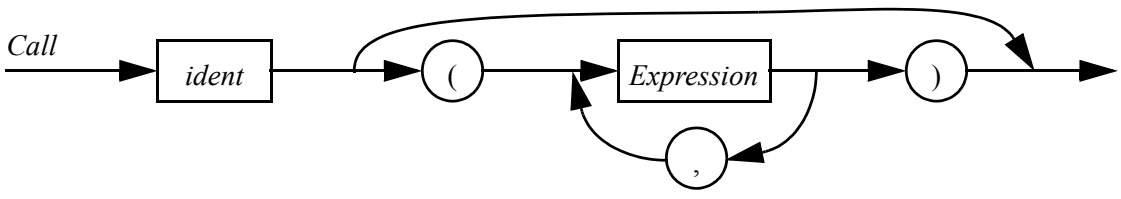


VarDeclList = { ident { "," ident } ":" Type ";" } .

Aufgabe 2 — Syntax beschreiben

(Punkte: 18 |.....)
WS 03/04

(1) Stellen Sie das folgende Syntaxdiagramm mit einer EBNF-Regel dar!

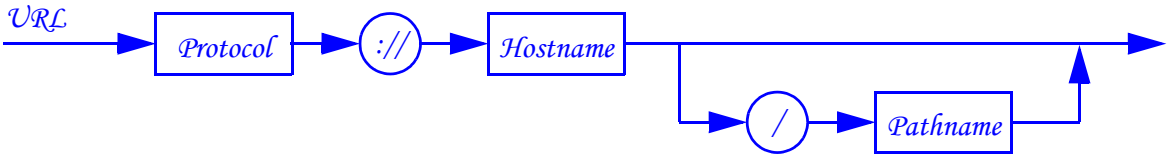


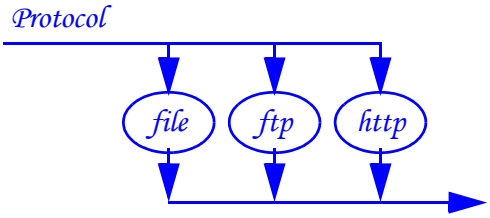
Call = ident ["(" Expression { "," Expression } ")"] .

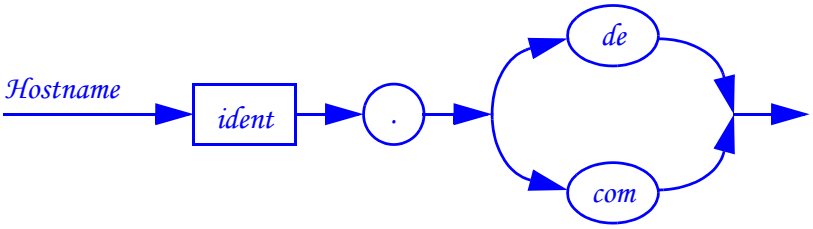
(2) Gegeben ist die Syntax einer Internetadresse in EBNF-Notation:

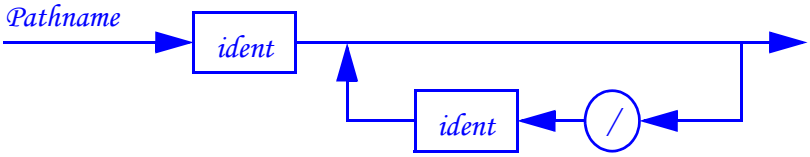
URL = *Protocol* „://“ *Hostname* [„/“ *Pathname*].
Protocol = „file“ | „ftp“ | „http“.
Hostname = *ident* „.“ („de“ | „com“).
Pathname = *ident* { „/“ *ident* }.
(lexikalische Einheit: *ident* = *letter* { *letter* | *digit* | „.“ | „-“ }.)

Zeichnen Sie zu den ersten vier dieser Regeln äquivalente Syntaxdiagramme!



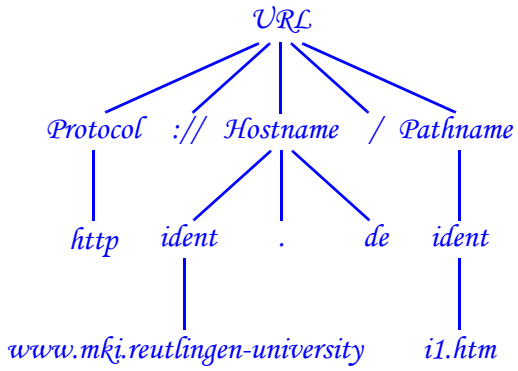






Zeichnen Sie zum folgenden Wort den konkreten Syntax-Zerteilungsbaum (ohne *ident* in einzelne Zeichen zu zerteilen)!

http://www.mki.reutlingen-university.de/i1.htm



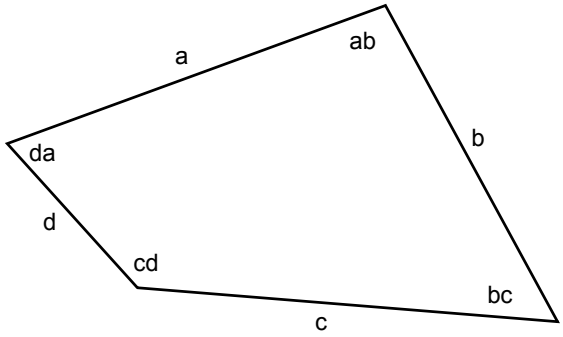
Aufgabe 3 — Invarianten eines Vierecks bestimmen

(Punkte: 13 |.....)
WS 04/05

Bei einem Viereck kann man die vier Seiten, die vier Winkel (im Bogenmaß), den Umfang, die Fläche und ob es ein Drachen (*kite*, zwei Paar Nachbarseiten gleich lang), ein Parallelogramm (*parallelogram*, gegenüberliegende Seiten gleich lang), eine Raute (*rhombus*, vier Seiten gleich lang), ein Rechteck (*rectangle*) oder ein Quadrat (*square*) ist, abfragen. Ergänzen Sie in der folgenden Schnittstelle 13 fehlende Invarianten und korrigieren Sie 13 fehlerhafte! (pi ist die Kreiszahl.)

Programm 3.1
Cleo: Viereck

```
INTERFACE Quadrangle
  QUERIES
    (* Seiten: *)
    a : REAL
    b : REAL
    c : REAL
    d : REAL
    (* Winkel: *)
    ab : REAL
    bc : REAL
    cd : REAL
    da : REAL
    (* Maße: *)
    perimeter  : REAL
    area       : REAL
    (* Eigenschaften: *)
    isKite          : BOOLEAN
    isParallelogram : BOOLEAN
    isRhombus       : BOOLEAN
    isRectangle     : BOOLEAN
    isSquare        : BOOLEAN
  INVARIANTS
    (* Bedingungen an Seiten: *)
    a > 0;
    b > 0;
    ~~c <= 0;~~ c > 0;
    d > 0;
    a + b + c > d;
    b + c + d > a;
    c + d + a > b;
    d + a + b > c;
    (* Bedingungen an Winkel: *)
    ab > 0;
    bc > 0;
    cd > 0;
    ~~da = 0;~~ da > 0;
    ab < pi;
    bc < pi;
    cd < pi;
```



```
    da < pi;
    ab + bc +          da = 999 * pi;  ab + bc + cd + da = 2 * pi;
    (* Bedingungen an Umfang und Fläche: *)
    perimeter = a + b + c + d;
    area > 0;
    isRectangle IMPLIES (area = a * b);
    (* Bedingungen an Eigenschaften und Seiten: *)
    isKite = ((a = b) AND (c = d)) OR ((a = d) AND (b # c));
    isKite = ((a = b) AND (c = d)) OR ((a = d) AND (b = c));
    isParallelogram = (a = c) AND (b = d);
    isRhombus = (a = b) OR (b = c) AND (c = d);
    isRhombus = (a = b) AND (b = c) AND (c = d);
    isRectangle IMPLIES ((a = c) AND (b = d));
    isSquare IMPLIES ((a = b) AND (b = c) AND (c = d));
    (* Bedingungen an Eigenschaften und Winkel: *)
    isKite IMPLIES ((ab = cd) OR (bc = da));
    isParallelogram = (ab = cd) AND (bc = da);
    isRhombus IMPLIES ((ab = cd) AND (bc = 3 * pi));
    isRhombus IMPLIES ((ab = cd) AND (bc = da));
    isRectangle = (ab = bc) AND (bc = cd) AND (cd = da);
    isRectangle = (ab = pi/2) AND (bc = pi/2) OR (cd = pi/2);
    isRectangle = (ab = pi/2) AND (bc = pi/2) AND (cd = pi/2);
    isSquare IMPLIES ((ab = bc) OR (bc = cd) OR (cd = da));
    isSquare IMPLIES ((ab = bc) AND (bc = cd) AND (cd = da));
    isSquare IMPLIES ((ab = pi/2) AND (bc = pi/2) AND (cd = pi/2));
    (* Zusammenhänge zwischen Eigenschaften: *)
    isRhombus = isKite AND isSquare;
    isRhombus = isKite AND isParallelogram;
    isRectangle IMPLIES isRhombus;
    isRectangle IMPLIES isParallelogram;
    isSquare = isRhombus OR isRectangle
    isSquare = isRhombus AND isRectangle
END Quadrangle
```

Aufgabe 4 Zusicherungen ergänzen

(Punkte: 7 |.....)
SS 98

Tragen Sie in das folgende Programmfragment möglichst starke Zusicherungen ein!

```
WHILE ~Stack.IsEmpty () DO
   Anweisungen1
END;

ASSERT (Stack.IsEmpty ());
IF x <= 1 THEN

   ASSERT (Stack.IsEmpty () & (x <= 1));
   Anweisungen2
ELSIF x > 9 THEN

   ASSERT (Stack.IsEmpty () & (x > 1) & (x > 9));
   Anweisungen3
ELSIF x = 8 THEN

   ASSERT (Stack.IsEmpty () & (x > 1) & (x <= 9) & (x = 8));
   Anweisungen4
ELSE

   ASSERT (Stack.IsEmpty () & (x > 1) & (x <= 9) & (x # 8));
   Anweisungen5
END
```

Aufgabe 5 Fachenglisch übersetzen

(Punkte: 5 |.....)
WS 05/06

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

> *An **array** is a structure consisting of a number of elements which are all of the same type, called the **element type**. The number of elements of an array is called its **length**. The elements of the array are designated by indices, which are integers between 0 and the length minus 1.*

Eine Reihung ist eine Struktur, die aus einer Anzahl von Elementen besteht, die alle vom selben Typ sind, der Elementtyp heißt. Die Anzahl der Elemente einer Reihung heißt ihre Länge. Die Elemente der Reihung werden durch Indizes bezeichnet, die ganze Zahlen zwischen 0 und der Länge minus 1 sind.

© Karlheinz Hug, Hochschule Reutlingen

Aufgabe 6

(Punkte: 16 |.....)
SS 98
WS 98/99
WS 03/04
WS 04/05
WS 05/06

Sprachkonstrukte beschreiben

Begründen Sie, warum die folgenden Aussagen für *Component Pascal* richtig oder falsch sind!

Aussage	Begründung
Ein aktueller Parameter ist immer eine Variable.	❒ Richtig ☒ Falsch, weil *ein aktueller Wertparameter ein beliebiger Ausdruck sein kann.*
Eine Zusicherung kann Vereinbarungen enthalten.	❒ Richtig ☒ Falsch, weil *je nach Sichtweise eine Zusicherung ein Aufruf der vordefinierten Prozedur ASSERT oder ein boolescher Ausdruck ist. Beides kann keine Vereinbarungen enthalten.*
Ein Aufruf einer Prozedur kann in einem Ausdruck vorkommen.	☒ Richtig ❒ Falsch, weil *es sich um eine Funktionsprozedur handeln kann und Funktionsaufrufe Ausdrücke sind.*
Vereinbarungen sind Anweisungen.	❒ Richtig ☒ Falsch, weil *Vereinbarungen und Anweisungen syntaktisch strikt in verschiedene Bereiche, nämlich Vereinbarungsteile und Anweisungsteile, getrennt sind.*
Anweisungen sind Ausdrücke.	❒ Richtig ☒ Falsch, weil *strikt zwischen der Ausführung von Anweisungen, die kein Ergebnis liefern, aber Zustände verändern, und der Auswertung von Ausdrücken, die Ergebnisse liefern, aber keine Zustände verändern sollen, unterschieden wird.*
Die Typen aktueller Parameter werden zur Laufzeit ermittelt.	❒ Richtig ☒ Falsch, weil *aktuelle Wertparameter Ausdrücke, aktuelle Referenzparameter Variablen sind und bei beiden ihre Typen statisch, also zur Übersetzungszeit, bestimmt sind.*
Die Typen von Ausdrücken werden zur Laufzeit ermittelt.	❒ Richtig ☒ Falsch, weil *der Typ eines Ausdrucks nur von den darin vorkommenden Operatoren und den Typen der Operanden abhängt und wie diese statisch, also zur Übersetzungszeit, bestimmt ist.*
Die Typen von Formalparametern werden zur Laufzeit ermittelt.	❒ Richtig ☒ Falsch, weil *der Typ eines Formalparameters bei seiner Definition im Quelltext angegeben wird, also statisch festgelegt ist.*

Aufgabe 7

(Punkte: 12 |.....)
SS 98
WS 98/99

Typregeln anwenden

Mit den Vereinbarungen

```
VAR
  a : CHAR;
  b : INTEGER;
  c : REAL;
```

besteht die Anweisung

```
IF ORD (a) < b MOD 3 THEN c := b * 1.2 END
```

Bemerkung: Unter den Anweisungen ist Platz frei gelassen, damit man die Typen der Teilausdrücke hinschreiben kann.

alle Prüfungen der Typverträglichkeit.

Mit den Vereinbarungen

```
VAR
  x : CHAR;
  z : REAL;
```

besteht die Anweisung

```
IF x < 'y' THEN z := ORD (x) * 3.4 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, und wo ein Typverträglichkeitsfehler vorliegt!

```
( ( i  <  x )  OR  ( c  =  CHR (i) ) )  &  ( b  +  1 )
```

INT REAL CHAR INT BOOL INT

implizit angepasst an REAL

explizit angepasst an CHAR

Typfehler! Addition "+" für BOOL nicht definiert!

BOOL BOOL

BOOL

Die Disjunktion links ist in Ordnung.

Der rechte Operand der Konjunktion enthält den Typfehler.

© Karlheinz Hug, Hochschule Reutlingen

Aufgabe 8 Programmstücke verbessern

Die gegebenen Programmstücke sind in äquivalente Programmstücke zu transformieren, d.h. ihre Semantik muss erhalten bleiben.

(Punkte: 2 |.....)
WS 01/02

Ersetzen Sie die folgende Auswahlanweisung durch *eine* äquivalente Zuweisung! (Es gilt i, k, n : INTEGER.)

```
ASSERT ((-1 <= i) & (i <= n) & (0 < n));
IF i = -1 THEN
   k := n - 1
ELSIF i = n THEN
   k := 0
ELSE
   k := i
END
```

k := i MOD n

(Punkte: 14 |.....)
WS 05/06

Ersetzen Sie in der folgenden Funktion zum Prüfen der Primeigenschaft einer Ganzzahl die ineffiziente Zählschleife durch eine Anweisungsfolge, die berücksichtigt, dass

- es sich um ein Suchproblem handelt;
- jede gerade Zahl außer 2 keine Primzahl ist;
- wenn n durch i teilbar ist, es ein k gibt mit $n = i * k$ und ($i \le k$ oder $k \le i$);
- die Wurzelfunktion nicht benutzt werden darf.

Programm 8.1
CP: Primeigenschaft prüfen

```
PROCEDURE IsPrime (n : INTEGER) : BOOLEAN;
   VAR
      result    : BOOLEAN;
      i         : INTEGER;
BEGIN
   IF n <= 1 THEN
      result := FALSE
   ELSE
      result := TRUE;
      FOR i := 2 TO n - 1 DO
         IF n MOD i = 0 THEN
            result := FALSE
         END
      END
   END;
   RETURN result
END IsPrime;
```

(Zu ersetzen! *)*

Den obigen Kasten ersetzende Anweisungsfolge:

```
IF n <= 3 THEN
   result := TRUE
ELSIF n MOD 2 = 0 THEN
   result := FALSE
ELSE
   i := 3;
   WHILE (i * i <= n) & (n MOD i # 0) DO
      INC (i, 2)
   END;
   ASSERT ((i * i > n) OR (n MOD i = 0));
   (* ASSERT (kein Teiler gefunden, also ist n prim OR i ist Teiler von n) *)
   result := i * i > n
END
```

(Punkte: 11 |.....)
WS 06/07

Programm 8.2
CP: Schwache Implementation

Verbessern Sie die Implementation der Funktion F schrittweise so, dass sie halb so lang ist und doppelt so schnell läuft!

```
PROCEDURE F (n : INTEGER) : REAL;
   VAR
      i, m   : INTEGER;
      x      : REAL;
BEGIN
   i := 123 * n;
   m := -n;
   x := 0;
   FOR i := m TO n DO
      IF i < 0 THEN
         x := x - 3 / i
      ELSIF i > 0 THEN
         x := x + 3 / i
      END;
      m := 2 * i + 1
   END;
   RETURN x / 6
END F;
```

Unnötige Zuweisungen streichen:

~~i := 123 * n~~

~~m := 2 * i + 1~~

Auf m verzichten:

~~m := -n~~

FOR i := *-n* TO n DO

Schleife in je eine Schleife für negative und positive Werte von i teilen, und den jeweils unnötigen Zweig der Auswahlanweisung streichen:

```
FOR i := -n TO -1 DO
   IF i < 0 THEN
      x := x - 3 / i
   ELSIF i > 0 THEN
      x := x + 3 / i
   END
END;
```

```
FOR i := 1 TO n DO
   IF i < 0 THEN
      x := x - 3 / i
   ELSIF i > 0 THEN
      x := x + 3 / i
   END
END;
```

Beide Schleifen berechnen Dasselbe. Also Wert einmal berechnen und mit 2 multiplizieren:

```
FOR i := 1 TO n DO
   x := x + 6 / i
END;
```

Die Faktoren 6 in der Schleife (6 / i) und 1/6 im Ergebnis (x / 6) heben sich auf, also weglassen. Gesamtergebnis:

```
PROCEDURE F (n : INTEGER) : REAL;
   VAR
      i : INTEGER;     x : REAL;
BEGIN
   x := 0;
   FOR i := 1 TO n DO
      x := x + 1 / i
   END;
   RETURN x
END F;
```

Aufgabe 9

(Punkte: 16 |.....)
WS 98/99

Schleifenterminierung prüfen und Schleifen transformieren

Begründen Sie zu jeder Schleife, warum sie abbricht oder nicht abbricht!

Schleife	terminiert / läuft endlos, weil...

i := 100;
x := 2;
WHILE (i > 0) & (x > 1) DO
  i := i - ENTIER (x);
  x := F (x)
END
	Terminiert, weil wenn die Fortsetzungsbedingung erfüllt ist, ist x > 1, also ENTIER (x) >= 1, also wird i bei jeder Iteration durch i := i - ENTIER (x) kleiner bis schließlich i <= 0 ist.
	Außerdem ist x <= 1 eine weitere Abbruchbedingung.

k := 123;
FOR i := k TO 2 * k DO
  k := k + i
END
	Terminiert, weil es eine normale Zählschleife mit korrekt benutzter Zählvariable i ist.
	*Beachte: Die Obergrenze 2 * k wird nur einmal zu Beginn der Schleife berechnet!*

x := 1;
y := 0;
REPEAT
  x := x / 2;
  y := y + x
UNTIL y > 1
	Läuft endlos, weil 1/2 + 1/4 + 1/8 + ... = 1 ist.
	y könnte höchstens durch Rundungsfehler größer 1 werden (ist unwahrscheinlich).
	Durch wiederholtes Halbieren wird irgendwann x kleiner als die kleinste darstellbare Zahl, also x = 0, und dann wird y nicht mehr größer.

Die Programmiersprachen *C*, *C++*, *Java* bieten ein Konstrukt für Wiederholungsanweisungen, das vereinfacht so aussieht:

```
for (initAnweisung; Fortsetzungsbedingung; termAnweisung)
{
   iterierteAnweisungen
}
```

Es ist semantisch äquivalent mit dem *Component-Pascal*-Konstrukt

```
initAnweisung;
WHILE Fortsetzungsbedingung DO
   iterierteAnweisungen;
   termAnweisung
END
```

Eine Anweisung kann leer sein.

Formulieren Sie die zweite (FOR) und die dritte (REPEAT) *Component-Pascal*-Schleife in obiger Tabelle mit der *C*-`for`-Schleife!

Bemerkung: Die C-Sprachen missbrauchen unsinnigerweise das in der Mathematik seit 300 Jahren weltweit standardisierte Gleichheitszeichen "=" als Zuweisungszeichen.

```
// FOR:
k = 123;
aux = 2 * k;
for (i = k; i <= aux; i = i + 1)
{
   k = k + i;
}
```

```
// REPEAT:
x = 1;
y = 0;
x = x / 2;
for (y = y + x; y <=1; y = y + x)
{
   x = x / 2;
}
```

Aufgabe 10 Schleifenablauf und -terminierung prüfen

(Punkte: 5 |.....)
WS 03/04

Verfolgen Sie zur Schleife

```
p := 16;
q := 59;
r := 0;
WHILE q > 0 DO
   q := q - 1;
   r := r + p
END
```

sechs Durchläufe und begründen Sie, warum sie abbricht oder nicht abbricht!

q	r
59	0
58	*16*
57	*32*
56	*48*
55	*64*
54	*80*
53	*96*
...	*...*

☒ Terminiert ☐ Läuft endlos, weil

q mit jeder Iteration um 1 kleiner wird und dadurch schließlich q <= 0 gilt.

Aufgabe 11 Schleife umformen

(Punkte: 12 |.....)
WS 06/07

Transformieren Sie die rumpfgesteuerte Bedingungsschleife

```
LOOP                                  + keine zusätzliche Variable erforderlich
   A;                                 + nichts doppelt geschrieben
   IF B THEN
      EXIT                            - nicht strukturiert wegen EXIT
   END;                               - 2 Ausgänge schwer erkennbar, man muss
   C;                                    dazu ganzen Schleifenrumpf lesen
   IF D THEN                          - 2 IF-Anweisungen erforderlich
      EXIT
   END;
   E
END;
ASSERT (B OR D)
```

in eine äquivalente kopfgesteuerte Bedingungsschleife mit der zusätzlichen Variablen

```
VAR done : BOOLEAN;
```

```
A;                                    + strukturiert
done := FALSE;                        + 1 Ausgang klar erkennbar
WHILE ~B & ~done DO                   + 1 IF-Anweisung genügt
   C;
   done := D;                         - zusätzliche Variable done erforderlich
   IF ~done THEN                      - A zweimal hingeschrieben
      E;                              - done zweimal ausgewertet
      A                               - größere Schachtelungstiefe mit echten
   END                                   Anweisungen
END;
ASSERT (B OR (done & D))
```

und schreiben Sie neben jede Variante ihre Stärken (+) und Schwächen (-) hin!

© Karlheinz Hug, Hochschule Reutlingen

Aufgabe 12

(Punkte: 14 |.....)
WS 05/06

Schleifenablauf und -terminierung prüfen

Zu jeder der beiden Schleifen sollen Sie sechs Durchläufe verfolgen, möglichst starke Zusicherungen ergänzen, und begründen, warum sie abbricht oder nicht abbricht!

Schleife 1

```
p := 18;
q := 99;
r := 9;
WHILE (q # 0) OR (r # 0) DO
    q := q - 1;
    r := (r + p) MOD 10;

    ASSERT(r IN {1, 3, 5, 7, 9})
END
```

q	r
99	9
98	*7*
97	*5*
96	*3*
95	*1*
94	*9*
93	*7*
...	*...*

❒ Terminiert ☒ Läuft endlos, weil

wegen r IN {1, 3, 5, 7, 9} stets r # 0 ist, also (q # 0) OR (r # 0) stets erfüllt ist.

Schleife 2

```
p := 12; q := 3; r := 4;
WHILE p > 0 DO
    IF q <= r THEN
        p := p - q;
        q := q + r
    ELSE
        r := 2 * q;
        p := p + r
    END;

    ASSERT((p >= 9) & (q >= 3) & (r >= 4))
END
```

p	q	r
12	3	4
9	*7*	
23		*14*
16	*21*	
58		*42*
37	*63*	
163		*126*
...	*...*	*...*

❒ Terminiert ☒ Läuft endlos, weil

p zwar abwechselnd ab- und zunimmt, aber im ELSE-Zweig mehr zunimmt als im THEN-Zweig abnimmt, also stets p > 0 bleibt.

Aufgabe 13

Programmablauf verfolgen

(Punkte: 7 |.....)

Untersuchen Sie Programm 13.1 wie unten beschrieben!

Programm 13.1
CP: Unbekannter Algorithmus

```
CONST
   number = 8;

VAR
   row                              : ARRAY number OF INTEGER;
   thisLow, thisHigh, thisSum,
   bestLow, bestHigh, bestSum       : INTEGER;

... BEGIN
   ... (* Initialisierung der Variablen. *) ...
   FOR thisHigh := 1 TO number - 1 DO
      IF thisSum <= 0 THEN
         thisLow     := thisHigh;
         thisSum     := 0
      END;
      INC (thisSum, row [thisHigh]);
      IF thisSum > bestSum THEN
         bestLow     := thisLow;
         bestHigh    := thisHigh;
         bestSum     := thisSum
      END
      (* 1 *)
   END
   (* 2 *)
END ...
```

Die folgenden Tabellen ordnen jeder Variablen eine Spalte zu. In den Zeilen (außer den Kopfzeilen) werden die Werte, die die Variablen besitzen, wenn die Programmausführung die Stelle *(* 1 *)* erreicht hat, aufgezeichnet. (Zwischenwerte der Variablen an anderen Programmstellen sind aus der Tabelle nicht ersichtlich.)

Bemerkung: Der Algorithmus bestimmt das Intervall bestLow .. bestHigh mit der maximalen Summe bestSum.

Die Programmausführung befindet sich an der Stelle *(* 1 *)*. Verfolgen Sie den Programmablauf weiter und vervollständigen Sie die Tabelle, indem Sie jeweils den Zustand der Variablen beim Erreichen der Stelle *(* 1 *)* in eine neue Zeile eintragen, bis die Stelle *(* 2 *)* erreicht ist!

row							
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
7	-9	5	1	-7	3	9	-1

thisLow	thisHigh	thisSum	bestLow	bestHigh	bestSum
2	3	6	0	0	7
	4	*-1*			
5	*5*	~~*0*~~ *3*			
	6	*12*	*5*	*6*	*12*
	7	*11*			
	8				

© Karlheinz Hug, Hochschule Reutlingen

Aufgabe 14

Seltsame Funktionen untersuchen

(Punkte: 14 |.....)
WS 98/99

Programm 14.1
CP: Zurückgeben und Setzen

```
PROCEDURE PreSet (OUT i : INTEGER; k : INTEGER) : INTEGER;
BEGIN
   i := k;
   RETURN i
END PreSet;

PROCEDURE PostSet (VAR i : INTEGER; k : INTEGER) : INTEGER;
   VAR n : INTEGER;
BEGIN
   n := i;
   i := k;
   RETURN n
END PostSet;
```

Werten Sie folgende Ausdrücke aus, und zwar *jeweils einzeln* unter der Voraussetzung

```
VAR i : INTEGER; ... i := 1
```

Bemerkung: Die Reihenfolge der Auswertung der Operanden eines Ausdrucks ist nicht festgelegt. Es kann erst der linke, dann der rechte Operand ausgewertet werden, oder umgekehrt.

Ausdruck	**mögliche Werte**	
	des Ausdrucks	**von i nach Auswertung des Ausdrucks**
2 2 PreSet (i, 2) + i *2 1*	*4* *3*	*2*
1 1 i + PostSet (i, 2) *2 1*	*2* *3*	*2*
1 3 PostSet (i, 2) + PreSet (i, 3) *3 3*	*4* *6*	*3* *2*

Ersetzen Sie die Anweisung jeweils durch Anweisungsfolgen ohne Funktionsaufrufe!

Anweisung	**äquivalente Anweisungsfolge**
m := PreSet (n, 2)	*n := 2; m := n*
m := PostSet (n, 2)	*m := n; n := 2*

Warum dürfen die Funktionen PreSet, PostSet nicht wie „normale“ Abfragen in Zusicherungen aufgerufen werden?

Weil sie über ihre Referenzparameter Nebeneffekte auf globale Variablen ausüben und Zusicherungen nebeneffektfrei sein müssen.

Aufgabe 15 — Programmiermängel suchen und beseitigen

(Punkte: 15 |.....)
WS 01/02

Programm 15.1
CP: Defekter Vektorvergleich

Die folgende Funktion soll feststellen, ob die Reihungen x und y elementweise gleich sind. Dabei ist Any ein beliebiger einfacher Typ.

```
PROCEDURE Equal (IN x, y : ARRAY OF Any) : BOOLEAN;
   VAR
      i : INTEGER;
BEGIN (1)
   i := 1; (2)
                        (3)      (4)
   WHILE (x [i] = y [i]) & (i <= LEN (x)) DO
      i := i + 1
   END;
   RETURN i > LEN (y) (5)
END Equal;
```

Programm 15.1 ist syntaktisch korrekt und übersetzbar, enthält aber *fünf Mängel*, die es unbrauchbar machen. Markieren Sie in Programm 15.1 die Mängel mit (1) .. (5)! Beschreiben Sie unten die Mängel und nennen Sie mögliche Folgen und Änderungen, die die Mängel beseitigen!

Mangel	Folge	Änderung
(1) *x kann länger sein als y. Index i orientiert sich nur an x.*	*Indexüberlauf bei y [i] für i >= LEN (y) führt zu Trap.*	*Anfangs auf gleiche Länge prüfen. Entweder Vorbedingung oder Fallunterscheidung: IF LEN (x) # LEN (y) THEN RETURN FALSE ...*
(2) *Indexwert 0 fehlt.*	*x [0] = y [0] nicht geprüft, verfälscht Ergebnis.*	*Initialisierung i := 0.*
(3) *Falsche Reihenfolge der Bedingungen.*	*Indexüberlauf bei x [i] oder y [i] bevor Gültigkeit des Index i geprüft wird führt zu Trap.*	*Reihenfolge der Bedingungen tauschen, kurze Auswertung nutzen.*
(4) *Indexwert LEN (x) zu viel.*	*Indexüberlauf bei x [i] für i = LEN (x) führt zu Trap.*	*Bedingung i < LEN (x).*
(5) *Falls LEN (x) = LEN (y) ist i > LEN (y) zu stark.*	*Ergebnis stets FALSE.*	*RETURN i >= LEN (x).*

Aufgabe 16 — Prozedur implementieren

(Punkte: 12 |.....)
SS 98

Programmieren Sie den Rumpf der folgenden Prozedur, bei der die Größen ab, bc, ca die Winkel eines Dreiecks bezeichnen, dessen Art sie bestimmen soll! pi ist die Kreiszahl. Sie dürfen hier exakte Arithmetik verwenden. Geben Sie eine möglichst schwache Vorbedingung an die Eingabeparameter an, die alle fehlerhaften Aufrufe abfängt!

© Karlheinz Hug, Hochschule Reutlingen

Programm 16.1
CP: Dreiecksart bestimmen

```
PROCEDURE DreiecksartBestimmen
   (ab, bc : REAL; OUT gleichseitig, gleichschenklig, rechtwinklig : BOOLEAN);
   VAR
      ca : REAL;
BEGIN
   ASSERT ((0 < ab) & (0 < bc) & (ab + bc < pi));
   ca             := pi - (ab + bc);
   gleichseitig   := (ab = bc) & (bc = ca);
   gleichschenklig := (ab = bc) OR (ab = ca) OR (bc = ca);
   rechtwinklig   := (ab = pi / 2) OR (bc = pi / 2) OR (ca = pi / 2);
   ASSERT (~gleichseitig OR (gleichschenklig & ~rechtwinklig))
END DreiecksartBestimmen;
```

Aufgabe 17 Prozedur implementieren

(Punkte: 7 |.....)

Die *Cleo*-Schnittstelle

Programm 17.1
Cleo: Uhr

```
INTERFACE Clock

   QUERIES
      hour, minute, second : NATURAL

   INVARIANTS
      hour < 24; minute < 60; second < 60

   ACTIONS
      Tick
         POST
            second = (OLD (second) + 1) MOD 60;
            (second = 0) IMPLIES (minute = (OLD (minute) + 1) MOD 60);
            ((second = 0) AND (minute = 0)) IMPLIES (hour = (OLD (hour) + 1) MOD 24)
         END

END Clock
```

spezifiziert eine Uhr. Implementieren Sie Tick als Prozedur in *Component Pascal*!

Programm 17.2
CP: Ticken

```
PROCEDURE Tick;
BEGIN
   second := (second + 1) MOD 60;
   IF second = 0 THEN
      minute := (minute + 1) MOD 60;
      IF minute = 0 THEN
         hour := (hour + 1) MOD 24
      END
   END
END Tick;
```

Aufgabe 18 Algorithmus entwerfen

(Punkte: 14 |.....)
WS 98/99

Entwerfen Sie einen Algorithmus (mit einem Struktogramm oder in Pseudocode), der in einer mit positiven Ganzzahlen initialisierten Reihung

```
a : ARRAY n OF INTEGER
```

die Elemente mit geradem Wert halbiert und die anderen verdreifacht und um eins erhöht, und dies so lange wiederholt, bis eines der Elemente den Wert eins hat! Schreiben Sie ggf. Vereinbarungen benötigter Hilfsvariablen hin!

Bemerkung: Dies ist eine Variante des Syrakus-Algorithmus, von dem unbekannt ist, ob er für jede Anfangszahl terminiert.

```
VAR
   i      :INTEGER;
   done   : BOOLEAN;
```

```
done := FALSE;
WHILE ~done DO
   FOR i := 0 TO LEN (a) - 1 DO
      IF a [i] MOD 2 = 0 THEN
         a [i]    := a [i] DIV 2;
         done     := done OR (a [i] = 1)
      ELSE
         a [i]    := 3 * a [i] + 1
      END
   END
END
```

Aufgabe 19 Datentyp und Algorithmus entwerfen

(Punkte: 11 |.....)
SS 00

(1) Modellieren Sie die Punktetabelle auf dem Deckblatt einer veröffentlichten Prüfung als einen Verbundtyp! Schreiben Sie geeignete Vereinbarungen hin, die folgende Regeln erfüllen:

- Alle vorkommenden Konstanten außer `0` und `1` haben einen Namen.
- Es kommen keine anonymen Typen vor.
- Die Vereinbarungen sind kommentarlos verständlich.

(2) Entwerfen Sie einen Algorithmus (mit einem Struktogramm oder in Pseudocode), der den Wert eines Summenfelds berechnet und setzt!

(3) Programmieren Sie zu dem Verbundtyp von (1) eine Prozedur, die den Algorithmus von (2) zur Berechnung der Summe der möglichen Punkte benutzt!

(1)

```
CONST
   Aufgabenanzahl = 24;

TYPE
   Punkte = ARRAY Aufgabenanzahl + 1 OF INTEGER;
                (* Index 0 bleibt ungenutzt, dafür oben einer mehr! *)

   Punktetabelle =
      RECORD
         PunkteErreicht,
         PunkteMöglich : Punkte;
         SummeErreicht,
         SummeMöglich : INTEGER;
      END;
```

(2)

```
dieseSumme := 0
FOR i := 1 TO Aufgabenanzahl DO
   INC (dieseSumme, diesePunkte [i])
END
```

(3)

```
PROCEDURE BerechneSummeMöglich (VAR pt : Punktetabelle);
   VAR i : INTEGER;
BEGIN
   pt.SummeMöglich := 0;
   FOR i := 1 TO Aufgabenanzahl DO
      INC (pt.SummeMöglich, pt.PunkteMöglich [i])
   END
END BerechneSummeMöglich;
```

 © Karlheinz Hug, Hochschule Reutlingen

Aufgabe 20 Funktion für Grenzwert programmieren

(Punkte: 13 |.....)

Programmieren Sie eine Funktion, die den Grenzwert des Ausdrucks

$$n + 1 - 1/(\log(n+1) - \log n)$$

für $n \to \infty$ näherungsweise berechnet und als Ergebnis liefert! Verwenden Sie dabei die Funktion Log aus dem Bibliotheksmodul Math!

Programm 20.1
CP: Grenzwert berechnen

```
PROCEDURE F20 (): REAL;
   VAR
      oldResult,
      result      : REAL;
      n           : INTEGER;
BEGIN
   oldResult  := -1;
   result     := 0;
   n          := 1000;                (* zum Beispiel als Anfangswert *)
   WHILE result # oldResult DO
      oldResult  := result;
      result     := n + 1 - 1 / (Math.Log (n + 1) - Math.Log (n));
      INC (n)
   END;
   RETURN result
END F20;
```

Aufgabe 21 Funktion für Reihe programmieren

(Punkte: 15 |.....)
WS 98/99

Programmieren Sie eine Funktion, die den Wert der unendlichen Reihe

$$\sum_{i=0}^{\infty} \frac{1}{(2i+1)^2}$$

näherungsweise berechnet und als Ergebnis liefert! Der exakte Wert der Reihe ist $\pi^2/8$. Formulieren Sie eine Zusicherung, die die semantische Korrektheit der Funktion prüft!

Programm 21.1
CP: Reihe berechnen

```
PROCEDURE F21 (): REAL;
   VAR
      oldResult,
      result      : REAL;
      i           : INTEGER;
BEGIN
   oldResult  := -1;
   result     := 0;
   i          := 1;                   (* 2i + 1 entspricht ungeraden Zahlen 1, 3, 5,... *)
   WHILE result # oldResult DO
      oldResult  := result;
      result     := result + 1 / (i * i);
      INC (i, 2)
   END;
   ASSERT (MAR.Equal (result, (Math.Pi () * Math.Pi ()) / 8);
   RETURN result
END F21;
```

Aufgabe 22 Zeichenmengenmodul implementieren

(Punkte: 23 |.....)
SS 00

Ein Modul für eine Menge von Zeichen ist mit einer booleschen Reihung implementiert:

Programm 22.1
CP: Zeichenmenge

```
MODULE ContainersSetOfChar;
   IMPORT ...
   VAR has : ARRAY ORD (MAX (CHAR)) - ORD (MIN (CHAR)) + 1 OF BOOLEAN;
   (* Queries *)
   PROCEDURE IsEmpty* () : BOOLEAN;
      VAR
         result    : BOOLEAN;
         i         : INTEGER;
   BEGIN
      result    := ~has [0];
      i         := 1;
      WHILE result & (i < LEN (has)) DO
         result := ~has [i];
         INC (i)
      END;
      RETURN result
   END IsEmpty;
   PROCEDURE Has* (x : CHAR) : BOOLEAN; ...
   (* Actions *) ...
END ContainersSetOfChar.
```

(1) Verbessern Sie die Implementation der Abfrage IsEmpty, sodass sich ihre mittlere Laufzeit auf etwa die Hälfte verkürzt!

(2) Erweitern Sie das Modul um eine Abfrage **Count*** () : INTEGER, die die Anzahl der Elemente in der Menge durch Berechnung liefert!

(3) Formulieren Sie zwei Invarianten des Mengenmoduls als boolesche Ausdrücke, in denen die neue Abfrage Count vorkommt!

(1) *Das letzte Element der Reihung wird zum Stoppelement gemacht, um eine Schleifenbedingung zu sparen.*

```
PROCEDURE IsEmpty* () : BOOLEAN;
   VAR
      last  : BOOLEAN;
      i     : INTEGER;
BEGIN
   last                 := has [LEN (has) - 1];
   has [LEN (has) - 1]  := TRUE;
   i                    := 0;
   WHILE ~has [i] DO
      INC (i)
   END;
   has [LEN (has) - 1] := last;
   RETURN (i >= LEN (has) - 1) & ~last
END IsEmpty;
```

(2)

```
PROCEDURE Count* () : INTEGER;
   VAR
      result,
      i        : INTEGER;
BEGIN
   result := 0;
   FOR i := 0 TO LEN (has) - 1 DO
      IF has [i] THEN
         INC (result)
      END
   END;
   RETURN result
END Count;
```

(3)

```
Invarianten:
   Count () >= 0;
   IsEmpty () = (Count () = 0);
```

Aufgabe 23 Funktion strukturiert implementieren

(Punkte: 8 |.....) WS 03/04

Implementieren Sie einen strukturierten Algorithmus zur folgenden Funktion, die untersucht, ob ihr Parameter „Zwillingsziffern“ hat!

Programm 23.1
CP: Zwillings-eigenschaft prüfen

```
PROCEDURE HasSiblings (n : INTEGER) : BOOLEAN;
   (*
      Gibt es zwei gleiche, direkt benachbarte Ziffern in der Dezimaldarstellung von n?
      Beispiele:
         HasSiblings (12341) = FALSE;
         HasSiblings (12234) = TRUE;
         HasSiblings (12223444) = TRUE.
   *)
BEGIN
   n := ABS (n);
   WHILE (n > 10) & ((n MOD 100) MOD 11 # 0) DO
      n := n DIV 10
   END;
   ASSERT ((n <= 10) OR ((n MOD 100) MOD 11 = 0));
   (* ASSERT (keine Zwillinge gefunden OR n MOD 100 ist Zwillingspaar); *)
   RETURN n > 10
END HasSiblings;
```

Aufgabe 24 Funktion strukturiert implementieren

(Punkte: 12 |.....)

Implementieren Sie einen strukturierten Algorithmus zur folgenden Funktion! Dabei ist Comparable ein beliebiger Typ, für den die Ordnungsrelationen <, >, <=, >= definiert sind, und CONST notFound = -1.

Programm 24.1
CP: Fehlstellung suchen

```
PROCEDURE MinIndexOfUnsorted (IN x : ARRAY OF Comparable) : INTEGER;
   (*
      Index des ersten Vorkommens eines Elements in x,
      das kleiner als das vorhergehende Element ist, falls existent; sonst notFound.
      Postcondition:
         (result = notFound) OR (result is the smallest index such that x [result] < x [result - 1]).
   *)
   VAR
      i : INTEGER;
BEGIN
   i := 1;
   WHILE (i < LEN (x)) & (x [i - 1] <= x [i]) DO
      INC (i)
   END;
   ASSERT ((i >= LEN (x)) OR (x [i - 1] > x [i]));
   (* ASSERT (keine Fehlstelle gefunden OR i ist erste Fehlstelle); *)
   IF i >= LEN (x) THEN
      RETURN notFound
   ELSE
      RETURN i
   END
END MinIndexOfUnsorted;
```