• AUFGABE 1 (4 Punkte) Grundbegriffe. Erklären Sie die folgenden Begriffe (in *ein* oder *zwei* Sätzen).

a. „Bottom“ bzw. '$\perp$'.

b. „Currying“.

c. Strikte und nicht strikte Funktion. Geben Sie jeweils ein Beispiel an.

d. Funktion höherer Ordnung.

• AUFGABE 2 (14 Punkte) Typen.

a. (2 Punkte) Was ist der Unterschied zwischen Basistypen und zusammengesetzten Typen? Geben Sie jeweils ein Beispiel an.

b. (6 Punkte) Bestimmen Sie nach dem Verfahren der Typ-Inferenz den Typ der Funktionen `z` und `s`. Diese Funktionen sind wie folgt definiert:

```
z f x = x
s n f x = f (n f x)
```

c. (6 Punkte) In welchem der folgenden Ausdrücke tritt ein Typfehler auf? Geben Sie den Typ der wohlgetypten Ausdrücke an. Definitionen der Funktionen finden Sie auf dem Formelblatt.

(i) `map sum`

(ii) `length last`

(iii) `foldr (+)`

• AUFGABE 3 (16 Punkte) Listen. Beweisen Sie die folgenden Gesetzmäßigkeiten induktiv. Definitionen der verwendeten Funktionen finden Sie auf dem Formelblatt.

a. (4 Punkte) `foldr count 0 xs = length xs`, wobei `count x n = n+1`

b. (4 Punkte) `reverse(xs ++ ys) = (reverse ys) ++ (reverse xs)`

c. (8 Punkte) `length(concat xss) = sum (map length xss)`

• AUFGABE 4 (8 Punkte) Listenfunktionen.

a. (4 Punkte) Geben Sie eine rekursive Funktion `inits` an, die als Eingabe eine Liste erhält und als Resultat die Liste aller Listenanfänge der gegebenen Argumentliste liefert. Beispiel:

```
inits [1, 2, 3] = [[],[1],[1,2],[1,2,3]]
```

b. (4 Punkte) Realisieren Sie mit Hilfe der Funktion `inits` eine weitere Funktion `segs`, die eine Liste der Segmente der Argumentliste als Resultat hat. Beispiel:

```
segs [1, 2, 3] = [[1], [1, 2], [1, 2, 3], [2], [2, 3], [3], []]
```

• AUFGABE 5 (16 Punkte) List-Comprehensions und Programmtransformation.

a. (4 Punkte) Geben Sie eine Funktion `quads` an, die zu einer natürlichen Zahl `n` die Liste aller Quadrate der geraden Zahlen von `1` bis `n` liefert. Die Funktion muß mit Hilfe einer List-Comprehension realisiert werden.

b. (4 Punkte) Transformieren Sie die List-Comprehension der Funktion `quads` formal in einen äquivalenten Ausdruck, der ohne List-Comprehension auskommt, dafür aber höherwertige Funktionen verwendet, z.B. `map`. Benutzen Sie zur Transformation die auf dem Formelblatt angegebenen Transformationsregeln für List-Comprehensions.

Hinweis: Führen Sie zunächst zwei Hilfsfunktionen `sqr` und `even` ein, die eine Zahl quadrieren bzw. testen, ob eine Zahl gerade ist.

c. (8 Punkte) Transformieren Sie die Funktion `quads` (aus der letzten Teilaufgabe) formal in eine rekursive Form, die keine *höherwertigen* Hilfsfunktionen verwendet. Auf dem Formelblatt angegebene Gesetzmäßigkeiten können Sie ohne Beweis verwenden.

Hinweis: Die Rekursion wird über natürliche Zahlen geführt. Außerdem können Sie folgende Gesetzmäßigkeit verwenden:

```
filter even [1..n] = filter even [1..(n-1)] ++ filter even [n]
```

• AUFGABE 6 (12 Punkte) Benutzerdefiniert Datenstrukturen. Gegeben sei folgende Repräsentation Boolescher Ausdrücke in Gofer:

```
data Prop = Con Bool       -- Konstanten
          | Var Char       -- Variablen(namen)
          | And Prop Prop  -- Konjunktion
          | Or  Prop Prop  -- Disjunktion
          | Imp Prop Prop  -- Implikation
          | Not Prop       -- Negation
```

Die Formel $x \wedge (y \rightarrow \text{false})$ wird somit durch folgenden Ausdruck repräsentiert:

```
And (Var 'x') (Imp (Var 'y') (Con False))
```

a. (6 Punkte) Geben Sie eine Funktion `eval` an, die einen gegebenen booleschen Ausdruck vom Typ `Prop`, der *keine* Variablen enthält, auswertet. Ergebnis der Funktion ist also der boolesche Wert des als Argument angegebenen Ausdrucks.

   Hinweis: $a \rightarrow b \equiv \neg a \vee b$.

b. (6 Punkte) Geben Sie eine Funktion `unparse` an, die einen Booleschen Ausdruck vom Typ `Prop` in eine vollständige geklammerte textuelle Form vom Typ `String` überführt. Dabei sollen die Konstanten durch `0` bzw. `1` und Variablen durch ihren (einbuchstabigen) Namen repräsentiert werden. Die Operationen Negation, Implikation, Konjunktion und Disjunktion werden durch die Symbole `~`, `->`, `^` bzw. `v` dargestellt. Der Goferterm

   ```
   And (Var 'x') (Imp (Not (Var 'y')) (Con False))
   ```

   wird somit als `(x ^ ((~ y) -> 0))` dargestellt.

• AUFGABE 7 (4 Punkte) Grundbegriffe. Erklären Sie die folgenden Begriffe (in *ein* oder *zwei* Sätzen).

a. Zustand

b. Zusicherung

c. Invariante

d. totale Korrektheit

• AUFGABE 8 (9 Punkte) Zusicherungen, Hoare-Kalkül.

a. (2 Punkte) Erklären Sie den Zusammenhang zwischen Zuweisung und Zustand.

b. (3 Punkte) Geben Sie die Hoare-Regel für die Zuweisungsoperation an.

c. (4 Punkte) Gegeben sei folgendes Programmstück (mit Raum für Vor- und Nachbedingung):

   ```
   {...}
   i := i + 1;
   {...}
   ```

   Die Variable `i` nehme natürliche Zahlen als Werte an. Nach Ausführung des Programmstücks soll `i` einen Wert kleiner als 43 annehmen. Formulieren Sie formal die Nachbedingung. Leiten Sie daraus unter Anwendung der Zuweisungsregel die Vorbedingung her.

• AUFGABE 9 (16 Punkte) Zusicherungen, imperative Programmierung.

Gesucht ist eine Funktionsprozedur `Mean`. Sie erhält als Eingabe ein nichtleeres Feld von Zahlen vom Datentyp `REAL` sowie dessen Länge und soll das arithmetische Mittel der Zahlen als Resultat liefern. Der Prozedurkopf sieht somit wie folgt aus:

```
PROCEDURE Mean (a: ARRAY OF REAL; n: CARDINAL): REAL;
```

a. (6 Punkte) Spezifizieren Sie formal die Vor- und Nachbedingung für den Rumpf der Prozedur `Mean`.

b. (4 Punkte) Geben Sie einen Rumpf an, der dieser Spezifikation genügt.

c. (6 Punkte) Beweisen Sie die partielle Korrektheit des Rumpfes mittels der Regeln des Hoare-Kalküls. Geben Sie dabei die jeweils verwendeten Regeln an.

• **AUFGABE 10 (9 Punkte) Aufwandsabschätzung.** Gegeben seien zwei Prozeduren mit bekanntem asymptotischem Aufwand,

Prozedur `G(n : INTEGER)` mit Aufwand $\mathcal{O}(log(n))$,
Prozedur `F(n : INTEGER)` mit Aufwand $\mathcal{O}(n^2)$,

sowie die Variablen n, i, j : `INTEGER`.

Leiten Sie den Aufwand folgender Programmfragmente in Abhängigkeit von n her:

a. (3 Punkte)

```
i := 0;
WHILE (i < n) DO
  F(i);
  i := i + 1;
END;
```

b. (3 Punkte)

```
i := 0;
WHILE (i < n) DO
  G(i);
  F(i);
  i := i + 1;
END;
```

c. (3 Punkte)

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

• **AUFGABE 11 (12 Punkte) Spezifikation und Implementierung.** Gegeben seien folgende funktional spezifizierte Funktionen auf Listen:

```
create :: [Char]
-- erzeugt ein neues Objekt vom Typ [Char]
create = []

empty :: [Char] -> Bool
-- True falls die Liste leer ist, False sonst.
empty xs = xs == []

first :: [Char] -> Char
-- liefert das erste Element einer Liste von Char
-- undefiniert falls die Liste leer ist
first (x:xs) = x

cons :: Char -> [Char] -> [Char]
-- fuegt ein Zeichen am Anfang der Liste ein
cons x xs = [x] ++ xs

rev :: [Char] -> [Char]
-- nimmt eine Liste von Char und dreht sie um
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

a. (4 Punkte) Geben Sie ein Modula-2 Definitionsmodul an, das diese Funktionen exportiert. Dieses Definitionsmodul soll einen Typ `TCharLi` opaque exportieren.

b. (8 Punkte) Geben Sie ein passendes Implementationsmodul an. Implementieren Sie dabei den Typ `TCharLi` als dynamische Datenstruktur.

• AUFGABE 1

a. „Bottom“ bzw. ‘$\perp$’ bezeichnet den undefinierten Wert.

b. Unter „Currying“ versteht man die Transformation einer mehrstelligen Funktion in eine einstellige, funktionswertige Funktion.

c. Eine Funktion f heißt strikt, falls $f(\perp) = \perp$; andernfalls heißt sie non-strikt.

d. Eine Funktion, welche eine andere Funktion als Argument annimmt bzw. eine Funktion als Wert liefert heißt Funktion höherer Ordnung.

• AUFGABE 2

a. Ein Typ beschreibt eine Menge von Werten aus dem Universum aller möglichen Werte. Zum einen gibt es Basistypen, deren Werte als Primitive vorliegen. Zum anderen zusammengesetzte Typen, deren Werte sich aus den Werten anderer Typen zusammensetzen.

b.

(1.1.1)	$f :: t_1$	Parameter
(1.1.2)	$x :: t_2$	Parameter
(1.2)	$x :: t_3$	rechte Seite
(1.3)	$z :: t_1 \to t_2 \to t_3$	allg. Typ
(1.4)	$t_2 = t_3$	Gleichungsregel mit 1.1.1 und 1.1.2
(1.5)	$z :: \alpha \to \beta \to \beta$	Resultat

(2.1.1)

$n :: t_1$ Parameter

(2.1.2)

$f :: t_2$ Parameter

(2.1.3)

$x :: t_3$ Parameter

(2.2)

$f\ (n\ f\ x) :: t_4$ rechte Seite

(2.3) $z :: t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4$ allg. Typ

(2.4.1)

$n\ f\ x :: t_5$

(2.4.2)

$f :: t_5 \rightarrow t_4$ Applikationsregel mit 2.2

(2.5) $t_2 = t_5 \rightarrow t_4$ Gleichungsregel mit 2.1.2 und 2.4.2

(2.6.1)

$x :: t_6$

(2.6.2)

$n\ f :: t_6 \rightarrow t_5$ Applikationsregel mit 2.4.1

(2.7) $t_3 = t_6$ Gleichungsregel mit 2.1.3 und 2.6.1

(2.8.1)

$f :: t_7$

(2.8.2)

$n :: t_7 \rightarrow t_6 \rightarrow t_5$ Applikationsregel mit 2.6.2

(2.9) $t_7 = t_5 \rightarrow t_4$ Gleichungsregel mit 2.4.2 und 2.8.1

(2.10) $t_1 = t_7 \rightarrow t_6 \rightarrow t_5$ Gleichungsregel mit 2.1.1 und 2.8.2

(2.11)

$z :: ((\alpha \rightarrow \beta) \rightarrow \gamma \rightarrow \alpha) \rightarrow (\alpha \rightarrow \beta) \rightarrow \gamma \rightarrow \beta$ Resultat

c. (i) `map sum :: [[Int]] -> [Int]`

(ii) Typfehler bei `length last`, da `length` eine Liste erwartet.

(iii) `foldr (+) :: Int -> [Int] -> Int`

• AUFGABE 3

a. Beweis durch Induktion über die Struktur von `xs`:

Induktionsanfang (`xs = []`)

```
    foldr count 0 []
=     { foldr.1 }
    0
=     { length.1 }
    length []
```

Induktionsschluss (`xs` $\curvearrowright$ `x:xs`)

```
    foldr count 0 (x:xs)
=     { foldr.2 }
    count x (foldr count 0 xs)
=     { Induktionshypothese }
    count x (length xs)
=     { count }
    (length xs) + 1
=     { length.2 }
    length (x:xs)
```

Q.E.D

b. Beweis durch Induktion über die Struktur von `xs`:

Induktionsanfang (`xs = []`)

```
    reverse ([] ++ ys)
=     { ++.1 }
    reverse ys
=     { [] neutrales Element von ++ }
    (reverse ys)  ++ []
```

```
=     { reverse.1 }
   (reverse ys) ++ (reverse [])
```

Induktionsschluss (xs $\curvearrowright$ x:xs)

```
    reverse ((x:xs) ++ ys)
=     { ++.2 }
    reverse (x:(xs ++ ys))
=     { reverse.2 }
    (reverse (xs ++ ys)) ++ [x]
=     { Induktionshypothese }
    ((reverse ys) ++ (reverse xs)) ++ [x]
=     { ++ ist assoziativ }
    (reverse ys) ++ ((reverse xs) ++ [x])
=     { reverse.2 }
    (reverse ys) ++ (reverse (x:xs))
```

Q.E.D

c. Beweis durch Induktion über die Struktur von xss:

Induktionsanfang (xs = [])

```
    length (concat [])
=     { concat.1 }
    length []
=     { length.1 }
    0
=     { sum.1 }
    sum []
=     { map.1 }
    sum (map length [])
```

Induktionsschluss (xs $\curvearrowright$ x:xs)

```
    length (concat (xs:xss))
=     { concat.2 }
    length (xs ++ (concat xss))
=     { Hilfssatz }
    (length xs) + (length (concat xss))
=     { Induktionshypothese }
    (length xs) + (sum (map length xss))
=     { sum.2 }
    sum ((length xs):(map length xss))
=     { map.2 }
    sum (map length (xs:xss))
```

Q.E.D

Als Hilfssatz beweisen wir noch, daß `length` ein Homomorphismus ist: `length (xs ++ ys) = (length xs) + (length ys)`. Beweis durch Anwendung von (d) oder durch Induktion über die Struktur von xs:

Induktionsanfang (xs = [])

```
    length ([] ++ ys)
=     { ++.1 }
    length ys
=     { length.1 }
    (length []) + (length ys)
```

Induktionsschluss (xs $\curvearrowright$ x:xs)

```
    length ((x:xs) ++ ys)
=     { ++.2 }
    length (x:(xs ++ ys))
=     { length.2 }
    1 + (length (xs ++ ys))
```

```
=     { Induktionshypothese }
   1 + (length xs) + (length ys)
=     { length.2 }
   (length (x:xs)) + (length ys)
```

Q.E.D

• AUFGABE 4

a.

```
inits          :: [a] -> [[a]]
inits []        = [[]]
inits (x:xs)    = [] : map (x:) (inits xs)
```

b.

```
segs          :: [a] -> [[a]]
segs []        = [[]]
segs (x:xs)    = map (x:) (inits xs)  ++ segs xs
```

- **AUFGABE 5**

a.

```
quads   :: Int -> [Int]
quads n  = [x ^ 2 | x <- [1..n], x `mod` 2 == 0]
```

b.

```
    [x ^ 2 | x <- [1..n], x `mod` 2 == 0]
=
    [sqr x | x <- [1..n], even x]
    where even x= x `mod` 2 == 0
    sqr  x= x ^ 2
=     { (3) }
    [sqr x | x <- filter even [1..n]]
    where even x= x `mod` 2 == 0
    sqr  x= x ^ 2
=     { (2) }
    map sqr (filter even [1..n])
    where even x = x `mod` 2 == 0
    sqr  x = x ^ 2
```

c.

```
    quads 1
=
    map sqr (filter even [1..1])
=
    map sqr []
=
    []
```

even n = True:

```
    quads n
=
    map sqr (filter even [1..n])
=     { Regel aus dem Hinweis }
    map sqr (filter even ([1..n - 1] ++ [n]))
=     { Regel auf Formelblatt }
    map sqr (filter even [1..n - 1] ++ filter even [n])
=     { Regel auf Formelblatt }
    map sqr (filter even [1..n - 1]) ++ map sqr (filter even [n])
=
    quads (n - 1) ++ map sqr (filter even [n])
=     { filter.1, filter.2, even n = True }
    quads (n - 1) ++ map sqr [n]
=     { map.1, map.2 }
    quads (n - 1) ++ [sqr n]
```

even n = False:

```
    quads n
=
    map sqr (filter even [1..n])
=     { Regel aus dem Hinweis }
    map sqr (filter even ([1..n - 1] ++ [n]))
=     { Regel auf Formelblatt }
    map sqr (filter even [1..n - 1] ++ filter even [n])
=     { Regel auf Formelblatt }
    map sqr (filter even [1..n - 1]) ++ map sqr (filter even [n])
=
    quads (n - 1) ++ map sqr (filter even [n])
=     { filter.1, filter.3, even n = False }
    quads (n - 1) ++ map sqr []
```

$= \quad \{ \text{map.1} \}$

```
quads (n - 1)
```

```
quads 1              = []
quads n | even n     = quads (n - 1) ++ [sqr n]
quads n | otherwise  = quads (n - 1)
```

• AUFGABE 6

a.

```
eval            :: Prop -> Bool
eval (Con c)     = c
eval (And a b)   = apply (&&) a b
eval (Or  a b)   = apply (||) a b
eval (Imp a b)   = apply imp a b
eval (Not a)     = not (eval a)
                   where
                     apply op a b = op (eval a) (eval b)
                     imp   a b    = (not a) || b
```

b.

```
unparse            :: Prop -> String
unparse (Con c)     = unparseCon c
unparse (Var x)     = [x]
unparse (And a b)   = parents a " ^ "  b
unparse (Or  a b)   = parents a " v "  b
unparse (Imp a b)   = parents a " -> " b
unparse (Not a)     = "(~ " ++ unparse a ++ ")"
                      where
                        unparseCon True  = "1"
                        unparseCon False = "0"
                        parents a op b
                          = "(" ++ unparse a ++ op
                            ++ unparse b ++ ")"
```

• AUFGABE 7

a. **Zustand:** Unter dem Zustand eines Programmes versteht man die aktuelle Belegung aller Variablen zu einem bestimmten Zeitpunkt der Ausfuehrung des Programmes.

b. **Zusicherung:** Eine Zusicherung ist eine logische Aussage (ein Prädikat) über den Zustand eines Programmes an einer bestimmten Stelle des Programmes.

c. **Invariante:** Eine Invariante ist eine Zusicherung, die vor und nach der Ausführung eines Programmes oder eines Programmfragmentes gilt.

d. **totale Korrektheit:** Ein Programm wird dann „total korrekt“ genannt, wenn es partiell korrekt ist und terminiert.

• AUFGABE 8

a. Die Zuweisungsoperation ist die fundamentale Operation, die Zustandsänderungen verursacht.

b. Ist P ein Prädikat, x eine Variable und E ein Ausdruck, so lautet das Zuweisungsaxiom wie folgt:

$$\{P[E/x]\}\ x := E\ \{P\}$$

c. $\{\mathtt{i} \in \mathbb{N} \wedge i < 42\}$
$\{\mathtt{i} \in \mathbb{N} \wedge i + 1 < 43\ \}$

```
i := i + 1;
```

$\{\mathtt{i} \in \mathbb{N} \wedge i < 43\}$

- **AUFGABE 9**

```
Procedure Mittel(A: Array OF REAL, n : CARDINAL) : REAL;
(* E: a     : Array von Reals                          *)
(*    n     : Anzahl der Elemente von a                *)
(* A: Mittelwert der Elemente von a                    *)
VAR
 ergebnis : REAL; (* Enthaelt das Ergebnis der Prozedur     *)
 i : CARDINAL;    (* Laufvariabele der Schleife             *)
 sum : REAL;      (* Akumulator fuer die Summe der Elemente *)
                  (* des Arrays                             *)

BEGIN (* Mittel *)
```

$\{\Phi \equiv \mathtt{A} = (a_0, ..., a_{\mathtt{n}-1}) \in R^n \wedge \mathtt{n} \in Nat^+ \}$ *Vorbedingung*
$\{\Phi \wedge 0 = \sum_{j=0}^{0-1} a_j \wedge 0 \leq \mathtt{n} \}$ *leere Summe*

```
  i := 0;
```

$\{ \Phi \wedge 0 = \sum_{j=0}^{\mathtt{i}-1} a_j \wedge \mathtt{i} \leq \mathtt{n} \}$ *Zuweisungsax.*

```
  sum = 0;
```

$\{ \Phi \wedge \mathtt{sum} = \sum_{j=0}^{\mathtt{i}-1} a_j \wedge \mathtt{i} \leq \mathtt{n} \}$ *Zuweisungsax.*

```
  WHILE i < n DO
```

$\{ \Phi \wedge \mathtt{sum} = \sum_{j=0}^{\mathtt{i}-1} a_j \wedge \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} < \mathtt{n} \}$ *Schleifenregel*
$\{ \Phi \wedge \mathtt{sum} + a_i = \sum_{j=0}^{\mathtt{i}-1} a_j + a_i \wedge \mathtt{i} + 1 \leq \mathtt{n} \}$ *Arithmetik*

```
    sum = sum + a[i];
```

$\{ \Phi \wedge \mathtt{sum} = \sum_{j=0}^{\mathtt{i}-1} a_j + a_i \wedge \mathtt{i} + 1 \leq \mathtt{n} \}$ *Zuweisungsax.*
$\{ \Phi \wedge \mathtt{sum} = \sum_{j=0}^{\mathtt{i}+1-1} a_j \wedge \mathtt{i} + 1 \leq \mathtt{n} \}$ *Umformung*

```
    i := i + 1;
```

$\{ \Phi \wedge \mathtt{sum} = \sum_{j=0}^{\mathtt{i}-1} a_j \wedge \mathtt{i} \leq \mathtt{n} \}$ *Zuweisungsax.*

```
  END; (* WHILE *)
```

$\{ \Phi \wedge \mathtt{sum} = \sum_{j=0}^{\mathtt{i}-1} a_j \wedge \mathtt{i} \leq \mathtt{n} \wedge \neg(\mathtt{i} < \mathtt{n}) \}$ *Schleifenregel und Invariante*
$\{ \Phi \wedge \mathtt{sum} = \sum_{j=0}^{\mathtt{n}-1} a_j \}$ *Umformung*
$\{ \Phi \wedge \frac{\mathtt{sum}}{\mathtt{n}} = \frac{(\sum_{j=0}^{\mathtt{n}-1} a_j)}{\mathtt{n}} \}$ *Arithmetik*

```
  ergebnis := sum/n;
```

$\{ \Phi \wedge \mathtt{ergebnis} = \frac{(\sum_{j=0}^{\mathtt{n}-1} a_j)}{\mathtt{n}} \}$ *Nachbedingung*

```
  RETURN(ergebnis);

END Mittel;
```

- **AUFGABE 10**

a. $C_1 + \sum_{i=0}^{n-1} O(i^2) + C_2$

$= \{$ Arithmetik, Konstanten $\}$

$\sum_{i=1}^{n} O((i-1)^2) + \sum_{i=1}^{n} C_2$

$= \{$ Konstante, $O(i^2) < O(n^2)$ $\}$

$\sum_{i=1}^{n} O(n^2)$

$= \{Arithmetik\}$

$n * O(n^2)$

$= \{$ Regel 1 und 5 $\}$

$O(n^3)$

b. $C_1 + \sum_{i=0}^{n-1}(O(log(i)) + O(i^2) + C_2)$

$= \{$ Arithmentik, Konstante $\}$

$\sum_{i=1}^{n}(O(log(i-1)) + O((i-1)^2)$

$= \{\sum_{i=1}^{n}(O(log(i-1))) = \sum_{i=1}^{n}(O(i^2))$, Regel 3 $\}$

$\sum_{i=1}^{n}(O(i^2))$

$= \{\sum_{i=1}^{n}(O(i^2)) < \sum_{i=1}^{n}(O(n^2)), Arithmetik\}$

n * (O(n^2))

={ Arithmetik }

O(n^3)

c. $C_1 + \sum_{i=0}^{n-1}(C_1 + \sum_{j=0}^{i-1}(O(log(j)) + C_2) + C_2)$

$= \{$ Konstanten $\}$

$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} O(log(j))$

$= \{ j \leq i \}$

$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} O(log(i))$

$= \{$ Arithmetik, Regel 1 und 5 $\}$

$\sum_{i=0}^{n-1} O(i * log(i))$

$= \{ \sum_{i=0}^{n-1} O(i * log(i)) < \sum_{i=0}^{n-1} O(n * log(n)) \}$

$\sum_{i=0}^{n-1} O(n * log(n))$

$= \{$ Arithmetik, Regel 5 $\}$

$O(n^2 * log(n))$

- **AUFGABE 11**

a.

```
DEFINITION MODULE klausur;

TYPE TCharLi;

PROCEDURE Create(VAR liste : TCharLi);

PROCEDURE Empty(liste : TCharLi):BOOLEAN;

PROCEDURE First(liste : TCharLi): CHAR;

PROCEDURE App(VAR liste : TCharLi; c : CHAR);

PROCEDURE Rev(liste : TCharLi; VAR Ergebnisz : TCharLi);

END klausur.
```

b.

```
IMPLEMENTATION MODULE klausur;

FROM Storage IMPORT ALLOCATE;

TYPE TCharLi = POINTER TO CharLiNode;

     CharLiNode = RECORD
                    next : TCharLi;
                    inh  : CHAR;
                  END; (* RECORD *)


PROCEDURE Create(VAR liste : TCharLi);
BEGIN
  liste  := NIL;
END Create;

PROCEDURE Empty(liste : TCharLi):BOOLEAN;
BEGIN
  IF (liste = NIL) THEN
    RETURN(TRUE)
  ELSE
    RETURN(FALSE)
  END;
END Empty;

PROCEDURE First(liste : TCharLi): CHAR;
BEGIN
  IF NOT Empty(liste) THEN
    RETURN(liste^.inh)
  ELSE
    HALT;    (* FEHLERFALL *)
  END; (* IF *)
END First;

PROCEDURE App(VAR liste : TCharLi; c : CHAR);
VAR
  n, akt : TCharLi;
BEGIN
  NEW(n);
  n^.inh := c;
  n^.next := NIL;
  IF Empty(liste) THEN
    liste := n;
  ELSE
    akt := liste;
    WHILE (akt^.next <> NIL) DO; (* Suche Ende der Liste *)
      akt := akt^.next;
    END; (* WHILE *)
    akt^.next := n               (* Fuege neues Element an *)
  END; (* IF *)
END App;

PROCEDURE Rev(liste : TCharLi; VAR ergebnis : TCharLi);
VAR
    z : POINTER TO CharLiNode;
    l : TCharLi;
BEGIN
```

```
  IF liste^.next = NIL THEN
    ergebnis := liste;
  ELSE
    Rev(liste^.next, ergebnis);
    App(ergebnis,liste^.inh);
  END; (* IF *)
END Rev;

END klausur.
```