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

a. Partielle Parametrisierung.

b. „Lazy evaluation“ vs. „Eager evaluation“ bzw. „Strict evaluation“.

c. Referentielle Transparenz.

d. Polymorphe Funktion. Geben Sie ein Beispiel an.

• **AUFGABE 2 (14 Punkte) Typen.**

a. (2 Punkte) Was versteht man unter dem Prinzip „strong-typing“?

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

```
cons a b f = f a b
      hd l = l first
```

Dabei habe `first` den Typ `first` $:: \alpha \to \beta \to \alpha$

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 snd`

(ii) `snd (min, max)`

(iii) `snd . map`

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

a. (8 Punkte) `filter p (map f xs) = map f (filter (p . f) xs)`

b. (8 Punkte) `foldr f (foldr f e ys) xs = foldr f e (xs ++ ys)`

• **AUFGABE 4 (8 Punkte) Listenfunktionen.**

a. (4 Punkte) Geben Sie eine *rekursive* Funktion `split` an, die als Argumente eine Zahl `m` und eine Liste `xs` aus Zahlen erhält und als Ergebnis ein Paar aus zwei Listen liefert. Dabei enthält die erste Ergebnisliste alle Elemente der Argumentliste `xs`, die kleiner oder gleich `m` sind und die zweite Ergebnisliste alle, die echt größer als `m` sind. Beispiel:

```
split 5 [4, 2, 8, 1, 8, 9, 5] = ([4, 2, 1, 5], [8, 8, 9])
```

b. (4 Punkte) Realisieren Sie mit Hilfe der Funktion `split` eine weitere rekursive Funktion `qsort`, die eine gegebene Liste mit Hilfe des Quicksort-Algorithmus sortiert. Beispiel:

```
qsort [4, 2, 8, 1, 8, 9, 5] = [1, 2, 4, 5, 8, 8, 9]
```

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

a. (2 Punkte) Gegeben sei die folgende Funktionsdefinition:

```
foo k ((k', d):as) | k == k'   = d
foo k (_      :as) | otherwise = foo k as
```

Geben Sie den Typ von `foo` an (ohne Herleitung). Gehen Sie dabei davon aus, daß `==` nur auf den ganzen Zahlen (`Int`) definiert ist. Beschreiben Sie informell welche Aufgabe die Funktion `foo` erfüllt.

b. (4 Punkte) Geben Sie eine Funktion `foo'` an, die äquivalent zur Funktion `foo` ist und die *ohne Rekursion* auskommt und stattdessen eine List-Comprehension benutzt.

Hinweis: Benutzen Sie die zwei Funktionen `fst` und `snd`, deren Definition sie auf dem Formelblatt finden. Das erleichtert das Lösen der folgenden Teilaufgabe.

c. (4 Punkte) Transformieren Sie die Funktion `foo'` aus der vorangehenden Teilaufgabe in eine äquivalente Funktion `foo''` indem Sie die List-Comprehension in einen äquivalenten Ausdruck umwandeln, der ohne

List-Comprehension auskommt, dafür aber die höherwertigen Funktionen `map`, `filter` und/oder `concat` verwendet. Benutzen Sie dazu die auf dem Formelblatt angegebenen Regeln und geben sie die Transformationsschritte zusammen mit einem Hinweis auf die jeweils verwendete Regel an.

Hinweis: Führen Sie für das Prädikat von `filter` eine Hilfsfunktion ein.

d. (6 Punkte) Zeigen Sie, daß `foo` und `foo''` äquivalent sind, d.h. daß sie für gleiche Argumente das gleiche Ergebnis liefern, also für alle `k` und `as` gilt

`foo k as` = `foo'' k as`

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

```
type 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 $x \wedge (y \rightarrow 0)$ wird somit durch folgenden Ausdruck repräsentiert:

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

a. Geben Sie eine Funktion `subst` an, die einen booleschen Ausdruck `p` vom Typ `Prop`, eine Variable `x` vom Typ `Char` und einen zweiten booleschen Ausdruck `q` als Eingaben hat. Als Resultat liefert sie den Ausdruck, den man erhält, wenn man in `q` alle Vorkommen von `x` durch `p` ersetzt. Beispiel:

```
subst (And (Var 'y') (Var 'z')) 'z' (Con False)
          = And (Var 'y') (Con False)
```

b. Geben Sie eine Funktion `norm` an, die zu einen gegebenen booleschen Ausdruck `p` vom Typ `Prop` einen äquivalenten booleschen Ausdruck berechnet, der keine Disjunktionen (`Or`) und Implikationen (`Imp`) enthält. *Hinweis:* $a \rightarrow b \equiv \neg a \vee b$.

• AUFGABE 7 (6 Punkte) Grundbegriffe. Erklären Sie die folgenden Begriffe (in *ein* oder *zwei* Sätzen).

a. Pointer

b. Abstrakter Datentyp

c. partielle Korrektheit

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

a. (2 Punkte) Worüber machen Hoare-Regeln Aussagen?

b. (3 Punkte) Geben sie die Hoare-Regel fuer die Fallunterscheidung an.

c. (4 Punkte) Wenden sie die Fallunterscheidungsregel auf folgendes Programmfragment mit `i,j` $\in$ `Nat` an:

```
IF i = 0 THEN
   j := 1;
ELSE
   j := 0;
END;
```

$\{\ (j = 1 \wedge i = 0) \vee (j = 0 \wedge i \neq 0)\ \}$ *Fallunterscheidung*

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

Gesucht ist eine Funktionsprozedur `Max`. Sie erhält als Eingabe ein nichtleeres Feld von Zahlen vom Datentyp `REAL` sowie die Anzahl seiner Elemente `n` und soll als Ergebniss den maximalen Wert des Eingabefelds liefern. Der Prozedurkopf sieht somit wie folgt aus:

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

a. (6 Punkte) Geben Sie die Vor- und Nachbedingung für den Rumpf der Prozedur `Max` an.

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 $\mathcal{O}(\mathtt{n})$,
Prozedur `F(n : INTEGER)` mit Aufwand $\mathcal{O}(\mathtt{1}og(\mathtt{n}))$ und $|F(n)| < C$,

sowie die Variablen `i`, `j` : `INTEGER`, und die positiven Konstanten `k`, `l`.

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

a. (3 Punkte)

```
i := k;
WHILE (i >= 0) DO
  G(i);
  F(i);
  i := i - 1;
END;
```

b. (3 Punkte)

```
i := 0;
WHILE (i < n) DO
  G(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 (10 Punkte) Spezifikation und Implementierung. Gegeben sei folgendes Definitionmodul für dynamische Listen:

```
DEFINITION MODULE CharLi;

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

PROCEDURE Create():TCharLi;
(* create:: TCharLi
   -- erstelle ein neues Objekt vom Typ Liste
   create = []
*)

PROCEDURE Empty(liste : TCharLi):BOOLEAN;
(* empty:: TCharLi -> Bool
   -- True falls die Liste leer ist, false sonst.
   empty xs = xs == []
*)

PROCEDURE First(liste : TCharLi): CHAR;
(* first:: TCharLi -> Char
   -- liefert den Kopf einer Liste von Chars
   first (x:_) = x
*)

PROCEDURE Cons(liste : TCharLi; c : CHAR): TCharLi;
(* cons :: Char -> TCharLi -> TCharLi
   -- fuegt ein Zeichen am Anfang der Liste ein
   cons x xs = (x:xs)
*)

PROCEDURE Tail(liste : TCharLi) : TCharLi;
(* tail :: TCharLi -> TCharLi
   -- liefert den Rest einer Liste
   tail (x:xs) = xs
```

```
*)

PROCEDURE Dispose(liste : TCharLi) : TCharLi;
(*  dispose :: TCharLi -> TCharLi
    -- dealloziert die Liste (empty dispose xs == True)
    dispose [] = []
    dispose xs = dispose (tail xs)
*)
END CharLi.
```

Implementieren sie ausschließlich auf den Funktionen von `CharLi` aufbauend ein Modul `CharStack` das folgendermaßen spezifiziert sei.

```
createS :: TCharStack
-- erstellt ein neues Objekt vom Typ TCharStack
createS = create

emptyS :: TCharStack -> Bool
-- True falls der Stack leer ist, false sonst.
emptyS xs = empty xs

push :: TCharStack -> Char -> TCharStack
-- Legt ein Neues Element auf den Stack
push xs c = cons c xs

pop :: TCharStack -> TCharStack
-- Liefert den Stack ohne das erste Element
pop xs = tail xs

top :: TCharStack -> Char
-- liefert das erste Element
top [] = error "top auf leeren Stack angewandt\n"
top xs = first xs

blubber :: TCharStack -> Char -> TCharStack
-- sortiert Element an die richtige Stelle in einem sortierten Stack
blubber xs x | emptyS xs = push createS x
             | x < top xs = push xs x
             | otherwise = push (blubber (pop xs) x) (top xs)

ssort :: TCharStack -> TCharStack
-- sortiert einen Stack
ssort xs = hsort xs createS
           where
                hsort [] e = e
                hsort ys e = hsort (pop ys) (blubber e (top ys))
```

Gehen Sie dabei folgendermaßen vor:

a. (5 Punkte) Geben Sie das Definition-Modul an. Exportieren Sie den Typ `TCharStack` opaque. Die Implementierung von `TCharStack` soll komplett verborgen bleiben.

b. (5 Punkte) Gehen Sie davon aus, daß eine Implementierung für die Prozeduren `createS`, `emptyS`, `push`, `pop` und `top` schon vorhanden ist. Geben Sie den verbleibenden Teil des Implementation-Modul an, also die Typdeklaration und Implementierungen für `blubber` und `ssort`. Bedenken Sie, das Sie über die Implementierung von `TCharLi` nichts wissen. Sie können nur die Funktionen aus dem Definition-Modul benutzen.

Funktionsdefinitionen

```
(++)                         :: [a] -> [a] -> [a]
[] ++ ys                     =  ys
(x:xs) ++ ys                 =  x:(xs++ys)

concat                       :: [[a]] -> [a]
concat []                    =  []
concat (xs:xss)              =  xs ++ (concat xss)

length                       :: [a] -> Int
length []                    =  0
length (x:xs)                =  (length xs) + 1

reverse                      :: [a] -> [a]
reverse []                   =  []
reverse (x:xs)               =  (reverse xs) ++ [x]

fst                          :: (a, b) -> a
fst (a, _)                   =  a

snd                          :: (a, b) -> b
snd (_, b)                   =  b

head                         :: [a] -> a
head (x:_)                   =  x

filter                       :: (a -> Bool) -> [a] -> [a]
filter _ []                  =  []
filter p (x:xs) | p x        =  x :  filter p xs
filter p (x:xs) | otherwise  =  filter p xs

map                          :: (a->b) -> [a] -> [b]
map f []                     =  []
map f (x:xs)                 =  f x :  (map f xs)

foldr                        :: (a->b->b)->b->[a]->b
foldr op i []                =  i
foldr op i (x:xs)            =  op x (foldr op i xs)
```

```
min                          :: Int -> Int -> Int
min a b | a < b              =  a
min _ b | otherwise          =  b

max                          :: Int -> Int -> Int
max a b | a > b              =  a
max _ b | otherwise          =  b
```

Transformation von List-Comprehensions

(LC.1)

`[x | x <- xs]` = `xs`

(LC.2)

`[f x | x <- xs]` = `map f xs`

(LC.3)

`[`*exp* `| x <- xs, p x,...]`

= `[`*exp* `| x <- filter p xs,...]`

(LC.4)

`[`*exp* `| x <- xs, y <- ys,...]`

= `concat [[`*exp* `| y <- ys,...] | x <- xs]`

O-Kalkül Regeln

(**T** 1)	$f = \mathcal{O}(f)$	
(**T** 2)	$c \cdot \mathcal{O}(f) = \mathcal{O}(f)$	c positive Konstante
(**T** 3)	$\mathcal{O}(f) + \mathcal{O}(f) = \mathcal{O}(f)$	
(**T** 4)	$\mathcal{O}(\mathcal{O}(f)) = \mathcal{O}(f)$	
(**T** 5)	$\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$	
(**T** 6)	$\mathcal{O}(f \cdot g) = f \cdot \mathcal{O}(g)$	

• AUFGABE 1

a. Unter partieller Parametrisierung versteht man das Versorgen einer Funktion mit einem *Teil* ihrer Argumente. Resultat ist eine Funktion mit weniger Argumenten.

b. Unter „Eager evaluation“ bzw. „Strict evaluation“ versteht man eine Auswertungsstrategie, welche zuerst die Argumente einer Funktion auswertet, bevor sie den Rumpf der Funktion auswertet. Eine Auswertungsstrategie, bei welcher zuerst der Rumpf einer Funktion ausgewertet wird und nur bei Bedarf die Argumente, wird als „Lazy-Evaluation“ bezeichnet.

c. Unter referentieller Transparenz versteht man die Eigenschaft funktionaler Sprachen, daß Ausdrücke Werte bezeichnen bzw. denotieren, sowie daß der Wert eines Ausdrucks lediglich von den Werten seiner Unterausdrücke abhängt.

d. Eine Funktion heisst polymorph, wenn sie unabhängig von der konkreten Struktur ihrer Argumente definiert werden kann. In diesem Fall treten in ihrem Typ Typvariablen auf.

• AUFGABE 2

a. Unter „strong-typing“ versteht man das Prinzip, daß jedem Ausdruck ein eindeutiger Typ zugewiesen werden kann, der sich allein aus den Konstituenten des Ausdrucks herleiten läßt. Ausdrücke, denen kein Typ zugeordnet werden kann, besitzen auch keinen Wert.

b.

(1.1.2)	$a :: t_1$	Parameter
(1.1.3)	$b :: t_2$	Parameter
(1.1.4)	$f :: t_3$	Parameter
(1.2)	$f\ a\ b :: t_4$	rechte Seite
(1.3)	$cons :: t_1 \to t_2 \to t_3 \to t_4$	allg. Typ
(1.4.1)	$b :: t_5$	
(1.4.2)	$f\ a :: t_5 \to t_4$	Applikationsregel
(1.5.1)	$a :: t_6$	
(1.5.2)	$f :: t_6 \to (t_5 \to t_4)$	Applikationsregel
(1.6)	$t_5 = t_2$	Gleichungsregel
(1.7)	$t_4 = t_1$	Gleichungsregel
(1.8)	$cons :: \alpha \to \beta \to (\alpha \to \beta \to \gamma) \to \gamma$	Resultat

(2.1.1)	$l :: t_1$	Parameter
(2.2)	$l\ first :: t_2$	rechte Seite
(2.3)	$hd :: t_1 \to t_2$	allg. Typ
(2.4.1)	$first :: t_3$	
(2.4.2)	$l :: t_3 \to t_2$	
(2.5)	$t_1 = t_3 \to t_2$	
(2.6)	$t_3 = \alpha \to \beta \to \alpha$	
(2.7)	$hd :: ((\alpha \to \beta \to \alpha) \to \gamma) \to \gamma$	

c. (i) `map snd :: [(a,b)] -> [b]`

(ii) `snd (min,max) :: (Int,Int) -> Int`

(iii) `snd . map ::` Typfehler; `map` liefert eine Liste aber `snd` erwartet ein Paar.

• **AUFGABE 3**

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

Induktionsanfang (`xs = []`)

```
   filter p (map f [])
=    { map.1 }
   filter p []
=    { filter.1 }
   []
=    { map.1 }
   map f []
=    { filter.1 }
   map f (filter (p . f) [])
```

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

```
   filter p (map f (x:xs))
=    { map.2 }
   filter p ((f x): (map f xs))
=    { filter.2; • p(fx) }
   (f x) : (filter p (map f xs))
=    { Induktionshypothese }
   (f x) : (map f (filter (p . f) xs))
=    { map.2 }
   map f (x : (filter (p . f) xs))
=    { filter.2; • (p.f)(x) }
   map f (filter (p . f) (x:xs))
```

```
   filter p (map f (x:xs))
=    { map.2 }
   filter p ((f x): (map f xs))
=    { filter.2; • ¬p(fx) }
   filter p (map f xs)
=    { Induktionshypothese }
   map f (filter (p . f) xs)
=    { filter.2; • ¬(p.f)(x) }
   map f (filter (p . f) (x:xs))
```

Q.E.D

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

Induktionsanfang (`xs = []`)

```
   foldr f e ([] ++ ys)
=    { ++.1 }
   foldr f e ys
=    { foldr.1 }
   foldr f (foldr f e ys) []
```

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

```
   foldr f e ((x:xs) ++ ys)
=    { ++.2 }
   foldr f e (x:(xs ++ ys))
=    { foldr.2 }
   f x (foldr f e (xs ++ ys))
=    { Induktionshypothese }
   f x (foldr f (foldr f e ys) xs)
=    { foldr.2 }
   foldr f (foldr f e ys) (x:xs)
```

Q.E.D

• AUFGABE 4

a.

```
split                   :: Int -> [Int] -> ([Int], [Int])
split _ []               = ([], [])
split m (x:xs) | m <= x  = (x:ls, gs)
split m (x:xs) | m > x   = (ls, x:gs)
                           where
                             (ls, gs) = split m xs
```

b.

```
qsort        :: [Int] -> [Int]
qsort []      = []
qsort (x:xs)  = qsort ls ++ [x] ++ qsort gs
                where
                  (ls, gs) = split x xs
```

• AUFGABE 5

a.

```
foo :: Int -> [(Int, a)] -> a
```

Das zweite Argument von `foo` ist eine Assoziationsliste über Schlüsseln vom Typ `Int`. `foo` liefert den ersten Wert, der in der Liste mit dem (im ersten Argument) gegebenen Schlüssel assoziiert wird.

b.

```
foo' k as = head [snd a | a <- as, fst a == k]
```

c.

$$\begin{aligned}
\texttt{[x | x <- xs]} &= \texttt{xs} \\
\texttt{[f x | x <- xs]} &= \texttt{map f xs} \\
\texttt{[e | x <- xs, p x,...]} &= \texttt{[e | x <- filter p xs,...]} \\
\texttt{[e | x <- xs, y <- ys,...]} &= \texttt{concat [[e | y <- ys,...] | x <- xs]}
\end{aligned}$$

```
   [snd a | a <- as, fst a == k]
=
   [snd a | a <- as, fsteq k a] where fsteq k a = (k == fst a)
=    { (3) }
   [snd a | a <- filter (fsteq k) as] where fsteq k a = (k == fst a)
=    { (2) }
   map snd (filter (fsteq k) as) where fsteq k a = (k == fst a)
```

```
foo'' k as = (head . map snd . filter (fsteq k)) as
             where
               fsteq k a = (k == fst a)
```

d. für `[]`:

```
foo'' k []
```

```
      =    { wg. head }
        ⊥
      =    { keine Regel paßt }
        foo k []
```

für ((k', d):as):

1. Fall k == k':

```
    foo'' k ((k', d):as)
 =
    head (map snd (filter (fsteq k) ((k', d):as)))
 =    { fsteq k (k', d) = True wg. k == k',filter.2 }
    head (map snd ((k', d):(filter (fsteq k) as)))
 =    { map.2, snd.1 }
    head (d:(map snd (filter (fsteq k) as)))
 =    { head.1 }
    d
 =    { k == k',foo.1 }
    foo k ((k', d):as)
```

2. Fall k /= k':

```
    foo'' k ((k', d):as)
 =
    head (map snd (filter (fsteq k) ((k', d):as)))
 =    { fsteq k (k', d) = False wg. k /= k',filter.3 }
    head (map snd (filter (fsteq k) as))
 =
    foo'' k as
 =    { nach IV }
    foo k as
```

```
 =    { k /= k',foo.2 }
    foo k ((k', d):as)
```

• AUFGABE 6

a.

```
subst                          :: Prop -> Char -> Prop
subst p x (Con b)               = Con b
subst p x (Var y) | x == y      = p
                  | otherwise   = Var y
subst p x (And a b)             = And (substpx a) (substpx b)
subst p x (Or  a b)             = Or  (substpx a) (substpx b)
subst p x (Imp a b)             = Imp (substpx a) (substpx b)
subst p x (Not a)               = Not (substpx a)
                                where
                                  substpx = subst p x
```

b.

```
norm            :: Prop -> Prop
norm (Con b)     = Con b
norm (Var x)     = Var x
norm (And a b)   = And (norm a) (norm b)
norm (Or  a b)   = norm (Not (And (Not a) (Not b)))
norm (Imp a b)   = norm (Or (Not a) b)
norm (Not a)     = Not (norm a)
```

• AUFGABE 7

a. **Pointer:** Ein Pointer ist ein *Verweis* auf ein Datenobjekt. Operationen auf Pointern sind die Referenzierung (Wertzuweisung) und Dereferenzierung. Die Datenobjekte liegen i.d.R. in einem globalen Speicherbereich (Heap).

b. **Abstrakter Datentyp:** Ein Abstrakter Datentyp ist die Spezifikation aller Zustände sowie aller Zustandstransformationen eines Datentypes.

c. **partielle Korrektheit:** Ein Programm wird *partiell Korrekt* genannt, wenn es die Transvormation der *Vorbedingung* in die *Nachbedingung* gewährleistet. (totale Korrektheit - Termination).

- **AUFGABE 8**

a. Hoare-Regeln sind Schlußregeln, mit denen Beweise ueber die Gültigkeit von Zusicherungen geführt werden können.

b. siehe Pepper-Script

c. $\{ \ i, j \in Nat. \ \}$

```
IF i = 0 THEN
```

$\{ \ (1 = 1 \land i = 0) \ \}$ Fallunterscheidung

```
j := 1;
```

$\{ \ (j = 1 \land i = 0) \ \}$ *Zuweisung*

```
ELSE
```

$\{ \ 0 = 0 \land i \neq 0 \ \}$ *Fallunterscheidung*

```
j := 0;
```

$\{ \ j = 0 \land i \neq 0 \ \}$ *Zuweisung*

```
END;
```

$\{ \ (j = 1 \land i = 0) \lor (j = 0 \land i \neq 0) \ \}$ *Fallunterscheidung*

- **AUFGABE 9**

```
Procedure Max(A: Array OF REAL, n : CARDINAL) : REAL;
(* E: A     : Array von Reals                         *)
(*    n     : Anzahl der Elemente von a               *)
(* A: Maximaler Wert in A                             *)
VAR
  zwerg : REAL;    (* Enthaelt das Ergebnis der Prozedur     *)
  i : CARDINAL;    (* Laufvariabele der Schleife             *)
BEGIN (* Max *)
```

$\{$ **15:** $\Phi \equiv \mathtt{A} = (a_0, ..., a_{\mathrm{n}-1}) \in R^n \land n \in Nat \ \}$ *Vorbedingung*
$\{$ **14:** $\Phi \land a_0 \geq a_0..a_{0-1} \land 0 \leq n \ \}$ *Maximum über leeren Bereich*

```
i := 0;
```

$\{$ **13:** $\Phi \land a_0 \geq a_0...a_{i-1} \land i \leq n \ \}$ *Zuweisungsregel*

```
zwerg := A[0];
```

$\{$ **12:** $\Phi \land zwerg \in a_0...a_{n-1} \land zwerg \geq a_0...a_{i-1} \land i \leq n \ \}$ *Zuweisungsregel*

```
WHILE i < n DO
```

$\{$ **11:** $\Phi \land zwerg \in a_0...a_{n-1} \land zwerg \geq a_0...a_{i-1} \land i < n \ \}$ *Schleifenregel*

```
IF zwerg < A[i] THEN
```

$\{$ **10:** $\Phi \land zwerg \in a_0...a_{n-1} \land zwerg \geq a_0...a_{i-1} \land (zwerg < a_i) \land i < n \ \}$
Fallunterscheidung
$\{$ **9:** $\Phi \land a_i \in a_0...a_{n-1} \land a_i \geq a_0...a_{i-1} \land i < n \ \}$

```
zwerg := A[i];
```

$\{$ **8:** $\Phi \land zwerg \in a_0...a_{n-1} \land zwerg \geq a_0...a_{i-1} \land (zwerg = a_i) \land i < n \ \}$
Zuweisung

```
END;
```

{ **7:** $\Phi \wedge zwerg \in a_0...a_{n-1} \wedge zwerg \geq a_0...a_{i-1} \wedge (zwerg \geq a_i \vee zwerg = a_i) \wedge i < n$ } *Fallunterscheidung*
{ **6:** $\Phi \wedge zwerg \in a_0...a_{n-1} \wedge zwerg \geq a_0...a_i \wedge i < n$ } *Arithmetik*
{ **5:** $\Phi \wedge zwerg \in a_0...a_{n-1} \wedge zwerg \geq a_0...a_{i+1-1} \wedge i + 1 \leq n$ } *Arithmetik*

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

{ **4:** $\Phi \wedge zwerg \in a_0...a_{n-1} \wedge zwerg \geq a_0...a_{i-1} \wedge i \leq n$ } *Zuweisungsregel*

```
END;
```

{ **3:** $\Phi \wedge zwerg \in a_0...a_{n-1} \wedge zwerg \geq a_0...a_{i-1} \wedge i \leq n \wedge i \geq n$ } *Schleifenregel*
{ **2:** $\Phi \wedge zwerg \in a_0...a_{n-1} \wedge zwerg \geq a_0...a_{i-1} \wedge i = n$ } *Arithmetik*
{ **1:** $\Phi \wedge zwerg \in a_0...a_{n-1} \wedge zwerg \geq a_0...a_{n-1}$ } *Nachbedingung*

```
  RETURN(zwerg);

END Max;
```

- **AUFGABE 10**

a. Der Aufwand dieses Fragmentes ist nicht von `n` abhänig und auch sonst ziemlich konstant (`i := k!`).

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

= { Arithmentik, Konstanten }

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

$= \{i \leq n\}$

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

= { Arithmetik }

$n * O(log(n)) = \{Regel5\}$

O(n log(n))

c. $C_1 + \sum_{i=0}^{n-1}(C_2 + \sum_{j=0}^{i-1}(O(j) + C3) + C4)$

= { Konstanten }

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

$= \{ j < i \}$

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

= { Arithmetik, Regel 1 und 5 }

$\sum_{i=0}^{n-1} O(i^2)$

$= \{\sum_{i=0}^{n-1} O(i^2) < \sum_{i=0}^{n-1} O(n^2)\}$

$\sum_{i=0}^{n-1} O(n^2)$

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

$O(n^3)$

- **AUFGABE 11**

  - ```
    DEFINITION MODULE CharStack;

    FROM CharLi IMPORT TCharLi;

    TYPE TCharStack = TCharLi;

    PROCEDURE CreateS():TCharStack;

    PROCEDURE EmptyS(stack : TCharStack) : BOOLEAN;

    PROCEDURE Push(stack : TCharStack; c : CHAR) : TCharStack;

    PROCEDURE Pop(stack : TCharStack) : TCharStack;

    PROCEDURE Top(stack : TCharStack) : CHAR;

    PROCEDURE Blubber(stack : TCharStack; c : CHAR) : TCharStack;

    PROCEDURE SSort(stack : TCharStack) : TCharStack;

    END CharStack.
    ```

  - ```
    IMPLEMENTATION MODULE CharStack;

    FROM CharLi IMPORT TCharLi, Create, Empty,
                       First, Cons, Tail, Dispose;

                    .
                    .
                    .


    PROCEDURE Blubber(stack : TCharStack; c : CHAR) : TCharStack;
    BEGIN
      IF EmptyS (stack) THEN
        RETURN(Push(CreateS(),c));
      ELSIF ORD(c) < ORD(Top(stack)) THEN
        RETURN(Push(stack,c));
      ELSE
        RETURN(Push(Blubber(Pop(stack),c),Top(stack)));
      END;
    END Blubber;


    PROCEDURE SSort(stack : TCharStack) : TCharStack;
    ```

```
  PROCEDURE HSort(ys, e : TCharStack) : TCharStack;
  BEGIN
    IF EmptyS(ys) THEN
      RETURN(e);
    ELSE
      RETURN(HSort(Pop(ys),Blubber(e,Top(ys))));
    END; (* IF *)
  END HSort;

VAR erg,t : TCharStack;
BEGIN
  erg := HSort(stack,CreateS());
  t := Dispose(stack);
  RETURN(erg);
END SSort;

END CharStack.
```