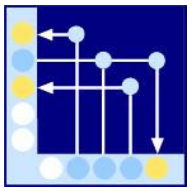
Hochschule Aalen
Fakultät Elektronik und Informatik
Studiengang Informatik



Programmieren in C++

Vorlesung im Wintersemester 2017/2018
Prof. Dr. habil. Christian Heinlein

7. Übungsblatt (18. Dezember 2017)

Aufgabe 12: Typ- und Funktionsschablonen

Teilaufgabe 12.a)

Definieren Sie eine Typ- oder Typalias-Schablone `List` mit ggf. erforderlichen Hilfstypschablonen sowie Funktionsschablonen `empty`, `cons`, `head`, `tail` und `disp`, die z. B. wie folgt verwendet werden können:

```
List<int> ls1 = empty<int>();
List<int> ls2 = cons(3);
// Oder:
List<int> ls2 = cons(3, empty<int>());
ls2 = cons(1, cons(2, ls2));
cout << head(ls2) << endl;              // Ausgabe: 1
List<int> ls3 = tail(ls2);
cout << head(ls3) << endl;              // Ausgabe: 2
cout << head(tail(ls3)) << endl;        // Ausgabe: 3
disp(ls2);
```

Das heißt:

- `empty<T>()` liefert eine leere Liste mit Elementtyp `T`.
  (Da die Funktion keine Parameter besitzt, kann der Typparameter `T` nicht deduziert werden und muss deshalb explizit angegeben werden.)
- `cons(x)` erzeugt eine einelementige Liste mit dem Element `x` und entsprechendem Elementtyp.
- `cons(x, ls)` erzeugt eine Liste mit erstem Element `x` und Restliste `ls`.
  (Der Typ von `x` und der Elementtyp von `ls` müssen gleich sein.)
- Für eine nichtleere Liste `ls` liefert `head(ls)` ihr erstes Element und `tail(ls)` ihre Restliste. (Für leere Listen dürfen diese Funktionen nicht aufgerufen werden.)
- `disp(ls)` gibt den Speicher frei, der von den Knoten der Liste `ls` belegt wird.

Hinweise:

- Damit Klienten nur die o. g. Funktionen verwenden können, sollten alle Elemente von Klassen privat sein.
- Um eventuelle Namenskonflikte mit Elementvariablen zu vermeiden, sollten `friend`-Deklarationen ganz am Anfang einer Klasse stehen.
- Wenn eine Funktion mit einem Defaultargument zuerst deklariert und später definiert wird, sollte das Defaultargument nur bei der Deklaration angegeben werden.

Teilaufgabe 12.b)

Definieren Sie weitere Funktionsschablonen, die wie folgt verwendet werden können:

- `map(xs, f)` erhält als Parameter eine Liste `xs` mit beliebigem Elementtyp `X` sowie eine Funktion `Y f (X)` mit Parametertyp `X` und beliebigem Resultattyp `Y`.
  Als Resultat erhält man eine Liste mit Elementtyp `Y`, deren Elemente jeweils durch einen Aufruf der Funktion `f` aus den entsprechenden Elementen der Liste `xs` entstehen.
  Zum Beispiel:

```
typedef const char* str;
List<str> ls1 = cons("a", cons("ab", cons("abc")));
List<size_t> ls2 = map(ls1, strlen);
// ls2 enthält die size_t-Werte 1, 2, 3.
```

- `map(xs1, xs2, f)` erhält als Parameter Listen `xs1` und `xs2` mit beliebigen Elementtypen `X1` und `X2` sowie eine Funktion `Y f (X1, X2)` mit Parametertypen `X1` und `X2` und beliebigem Resultattyp `Y`.
  Als Resultat erhält man eine Liste mit Elementtyp `Y`, deren Elemente jeweils durch einen Aufruf der Funktion `f` aus den entsprechenden Elementen der Listen `xs1` und `xs2` entstehen.
  Wenn `xs1` und `xs2` unterschiedlich lang sind, ist die Resultatliste so lang wie die kürzere von ihnen.
  Zum Beispiel:

```
char elem (str s, int i) { return s[i]; }
List<str> ls1 = cons("a", cons("ab", cons("abc")));
List<int> ls2 = cons(0, cons(1));
List<char> ls3 = map(ls1, ls2, elem);
// ls3 enthält die char-Werte 'a', 'b'.
```

- `filter(xs, p)` erhält als Parameter eine Liste `xs` mit beliebigem Elementtyp `X` sowie eine Funktion `bool p (X)` mit Parametertyp `X` und Resultattyp `bool`.
  Als Resultat erhält man eine Liste mit Elementtyp `X`, die alle Elemente der Liste `xs` enthält, die das Prädikat `p` erfüllen, d. h. für die der Aufruf der Funktion `p true` liefert.
  Zum Beispiel:

```
bool odd (int x) { return x % 2; }
List<int> ls1 = cons(1, cons(2, cons(3)));
List<int> ls2 = filter(ls1, odd);
// ls2 enthält die int-Werte 1, 3.
```

- `fold(xs, f, x)` erhält als Parameter eine Liste `xs` mit beliebigem Elementtyp `X`, eine Funktion `X f (X, X)` mit zwei Parametern und Resultat des Typs `X` sowie ein „neutrales Element“ `x` des Typs `X`.
  Als Resultat erhält man bei einer leeren Liste den Wert `x`, bei einer Liste mit einem Element `x1` den Wert `f(x1, x)`, bei einer Liste mit zwei Elementen `x1` und `x2` den Wert `f(x1, f(x2, x))` usw.
  Zum Beispiel:

```
double add (double x, double y) { return x + y; }
List<double> ls = ......;
double sum = fold(ls, add, 0.0);
// sum enthält die Summe aller Elemente der Liste ls.
```

- `seq(x, f, n)` erhält als Parameter einen Startwert `x` mit beliebigem Typ `X`, eine Funktion `X f (X)` mit Parameter- und Resultattyp `X` sowie eine nichtnegative ganze Zahl `n`.
  Als Resultat erhält man eine Liste der Länge `n` mit den Elementen `x`, `f(x)`, `f(f(x))`, ...
  Zum Beispiel:

```
double inc (double x) { return x + 1; }
List<double> ls = seq(1.0, inc, 10);
// ls enthält die double-Werte von 1 bis 10.
```

Anmerkung: Alle Funktionen lassen sich rekursiv in wenigen Zeilen formulieren.

Teilaufgabe 12.c)

Was gibt das folgende Programmfragment aus?

```
double add (double x, double y) { return x + y; }
double sub (double x, double y) { return x - y; }
double mul (double x, double y) { return x * y; }
double div (double x, double y) { return x / y; }

double id (double x) { return x; }
double inc (double x) { return x + 1; }
double neg (double x) { return -x; }

const int N = 100;
cout << fold(map(seq(1.0, id, N), seq(1.0, inc, N), div), add, 0.0) << endl;
cout << fold(map(seq(1.0, neg, N), seq(1.0, inc, N), div), add, 0.0) << endl;
```

Aufgabe 13: Template-Metaprogrammierung

Definieren Sie eine rekursive Typschablone `Bin<N, K>` mit geeigneten Spezialisierungen, sodass das statische konstante Datenelement `Bin<N, K>::value` für $N = 0, 1, \ldots$ und $K = 0, \ldots, N$ jeweils den zur Übersetzungszeit berechneten Wert des Binomialkoeffizienten

$$\binom{N}{K} = \begin{cases} 1 & \text{für } K = 0 \text{ oder } K = N \\ \binom{N-1}{K-1} + \binom{N-1}{K} & \text{sonst} \end{cases}$$

enthält!