Lösungen zu den Aufgaben C 1 bis C 6

In den folgenden Lösungen sind die Namen `f1` bis `f6` der Funktionen aus den Aufgaben C 1 bis C 6 durch aussagekräftigere Namen ersetzt, die ihre Berechnung bzw. Ausgabe besser beschreiben.

Aufgabe C 1

```
int prime (int n)
{                               /* --- fast prime number test */
  int d;                        /* divisor to test */
  int i;                        /* increment of divisor */

  if (n <  2) return 0;         /* 2 is the smallest prime number */
  if (n <= 3) return 1;         /* 2 and 3 are prime numbers */
  if (n % 2 == 0) return 0;     /* if n is even or if it is */
  if (n % 3 == 0) return 0;     /* divisible by 3, it is not prime */
  for (d = 5, i = 2; d*d <= n; d += i, i = 6-i)
    if (n % d == 0) return 0;   /* if a divisor found, n is not prime */
  return 1;                     /* if no divisor found, n is prime */
}  /* prime() */
```

Die Funktion `f1` aus Aufgabe C 1 führt einen schnellen Primzahltest durch (vgl. Beispielprogramm `prime.c`, Version 6 der Funktion `prime`). In der Testschleife werden alle Testteiler übersprungen, die durch 2 oder durch 3 teilbar sind, wodurch sich die Zahl der zu testenden potentiellen Teiler deutlich verringert. Denn wenn eine Zahl n durch einen Teiler d, und der Teiler d durch 2 oder durch 3 teilbar ist, dann muß auch n durch 2 bzw. durch 3 teilbar sein. Folglich ist der Test auf Teilbarkeit durch 2 oder 3, der ja sowieso ausgeführt werden muß, ausreichend.

Im einzelnen geht die Funktion so vor: Zunächst wird geprüft, ob die übergebene Zahl n kleiner als 2 ist. Da die Zahl 2 nach Definition die kleinste Primzahl ist, wird in diesem Fall $0 \mathrel{\hat{=}}$ falsch (also: n ist keine Primzahl) zurückgegeben.

Dann wird geprüft, ob die Zahl n kleiner oder gleich 3 ist. Wegen der vorhergehenden Abfrage, die ja für alle Zahlen n kleiner als 2 die Funktion bereits beendet, werden damit gerade die Zahlen 2 und 3 erfaßt, die offenbar Primzahlen sind. Daher wird $1 \mathrel{\hat{=}}$ wahr (also: n ist eine Primzahl) zurückgegeben.

Es folgen Tests, ob n durch 2 oder durch 3 teilbar ist. Diese Tests müssen vor der Schleife ausgeführt werden, da sie sich nicht in das Schema der in der Schleife getesteten Teiler einordnen lassen. Ist n durch 2 oder durch 3 teilbar, so ist n offenbar keine Primzahl und es wird folglich $0 \mathrel{\hat{=}}$ falsch zurückgegeben.

In der Schleife beginnen die Teilversuche mit dem Teiler $d = 5$. (Auf Teilbarkeit durch 4 braucht nicht geprüft zu werden, da jede durch 4 teilbare Zahl offensichtlich auch durch 2 teilbar ist.) Das Teilerinkrement wird auf 2 initialisiert, so daß anschließend der Teiler $d = 5 + 2 = 7$ getestet wird. (Die 6 kann aus den gleichen Gründen wie die 4 übersprungen werden.) Das neue Inkrement ergibt sich zu $i = 6 - 2 = 4$, so daß auf $d = 7$ der Testteiler $d = 7 + 4 = 11$ folgt. Anschließend werden $d = 13, 17, 19, 23, \ldots$ getestet.

Wie man sich leicht klar machen kann, wechselt das Inkrement i immer zwischen 2 und 4, da $6 - 2 = 4$ und $6 - 4 = 2$. Dadurch werden gerade die Vielfachen von 2 und 3 übersprungen. Warum ist das so? Wenn man die Folge der Zahlen, die durch 2, und die Folge der Zahlen, die durch 3 teilbar sind, betrachtet, so stellt man fest, daß sich diese beiden Folgen offenbar stets bei den Vielfachen von 6 treffen. Das Schema zum Überspringen der Vielfachen von 2 und 3 muß sich daher alle sechs Zahlen wiederholen. Das ist in der Schleife der Fall, da die beiden im Wechsel auftretenden Inkremente, 2 und 4, zusammen 6 ergeben.

Betrachten wir nun die Zahlen, die zwischen einem Vielfachen von 6, sagen wir $6k$, mit beliebigem $k \in \mathbb{N}$, und dem darauffolgenden Vielfachen $6(k + 1) = 6k + 6$ liegen:

	$6k$	$6k+1$	$6k+2$	$6k+3$	$6k+4$	$6k+5$	$6k+6$
teilbar durch 2:	•		•		•		•
teilbar durch 3:	•			•			•

$6k$ und $6k+6$ sind beide durch 2 und durch 3 teilbar. Zwischen ihnen liegen zwei Zahlen, die durch 2 teilbar sind, nämlich $6k+2 = 2(3k+1)$ und $6k+4 = 2(3k+2)$, und eine Zahl, die durch 3 teilbar ist, nämlich $6k+3 = 3(2k+1)$. Man erkennt also, daß genau die Zahlen, die einem Vielfachen von 6 direkt vorausgehen ($6k-1$, $6k+5$, $6k+11$ etc.) oder direkt auf ein Vielfaches von 6 folgen ($6k+1$, $6k+7$, $6k+13$ etc.) nicht durch 2 oder durch 3 teilbar sind. Um gerade diese Zahlen zu durchlaufen, muß man folglich abwechselnd eine Zahl (das Vielfache von 6) und drei Zahlen (die drei zwischen einem Vielfachen von 6 und dem nächsten liegenden Zahlen, die durch 2 oder durch 3 teilbar sind) überspringen, also abwechselnd 2 und 4 addieren. Begonnen wird mit der 5, die direkt dem ersten Vielfachen von 6 vorausgeht.

Die Teilversuche werden abgebrochen, wenn das Quadrat des nächsten Testteilers größer ist als die zu testende Zahl n. Der Grund hierfür wurde schon mehrfach besprochen. Wenn eine Zahl n durch eine Zahl d teilbar ist, die größer ist als $\sqrt{n}$ (für die also gilt $d^2 > n$), so muß es eine Zahl d' geben, die kleiner ist als $\sqrt{n}$ und die ebenfalls n teilt. Denn daß die (natürliche) Zahl d ein Teiler der (natürlichen) Zahl n ist, bedeutet ja, daß es eine (natürliche) Zahl d' gibt, so daß $d \cdot d' = n$. Kennt man also einen Teiler d, so kennt man (außer bei Quadratzahlen, bei denen $d = d'$ sein kann) auch einen zweiten, nämlich d'. Ist aber $d > \sqrt{n}$, so muß offenbar $d' < \sqrt{n}$ sein. Da die Testteiler aufsteigend durchlaufen werden und bis zum Testteiler d_{abbruch} (mit $d_{\text{abbruch}}^2 > n$) keine Teiler gefunden wurden, kann es keine Teiler mehr geben. Denn gäbe es einen Teiler d mit $d \geq d_{\text{abbruch}}$, hätte man nach der obigen Überlegung den zu ihm gehörenden zweiten Teiler d' bereits finden müssen.

Wenn in der Schleife ohne Rest geteilt werden kann, wird $0 \mathrel{\hat{=}}$ falsch zurückgegeben, da die Zahl n dann offenbar keine Primzahl ist. Wird in der Schleife kein Teiler gefunden, so hat n keine Teiler. Es wird daher $1 \mathrel{\hat{=}}$ wahr (also: n ist eine Primzahl) zurückgegeben.

Aufgabe C 2

```
int choose (int n, int k)
{                               /* --- compute n choose k */
  int i;                        /* loop variable */
  int r = 1;                    /* result of n choose k */

  for (i = 1; i <= k; i++, n--) /* compute \prod_{i=1}^k (n-i+1)/i */
    r = (r *n) /i;              /* =       n!/(k!(n-k)!) */
  return r;                     /* return result */
}  /* choose() */
```

Die Funktion `f2` aus Aufgabe C 2 berechnet den Binomialkoeffizienten $\binom{n}{k}$ (englisch „n choose k“, da der Binomialkoeffizient auch die Anzahl der Möglichkeiten angibt, k Objekte aus einer Menge von n Objekten auszuwählen) unter Ausnutzung der Umformung

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \prod_{i=1}^{k} \frac{(n-i+1)}{i}.$$

(vgl. Beispielprogramm `choose.c`) Daß das angegebene Produkt berechnet wird, läßt sich aus der Funktion direkt ablesen. Daß es sich um den Binomialkoeffizienten handelt, kann man in einem beliebigen Tafelwerk nachschlagen.

Die Schwierigkeit dieser Aufgabe besteht jedoch darin, zu verstehen, warum die Funktion, die ja nur mit ganzen Zahlen rechnet, nicht zu falschen Ergebnissen gelangt. Denn wird der Quotient zweier ganzer Zahlen berechnet, so werden in C bekanntlich eventuell auftretende Nachkommastellen einfach abgeschnitten. Träten also bei der Division in der Schleife Nachkommastellen auf, so wäre zu erwarten, daß ein falsches Ergebnis berechnet wird, da ja diese Nachkommastellen unberücksichtigt blieben. Der Quotient ist jedoch immer ohne Rest berechenbar, hat also nie Nachkommastellen. Warum?

Betrachten wir dazu, wie das Produkt in der Schleife berechnet wird. Im ersten Schleifendurchlauf hat r (wegen der Initialisierung bei der Definition dieser Variable) den Wert 1. Es wird also

$$r = \frac{1 \cdot n}{1} = n$$

berechnet. Die Division durch 1 ergibt natürlich nie einen Rest; im ersten Schritt treten daher nie Probleme auf.

Im zweiten Schleifendurchlauf wird der Faktor $\frac{n-1}{2}$ hinzumultipliziert. Ist n gerade, so ist dieser Faktor aber offenbar keine ganze Zahl, sondern hat, da für gerades n ja $n-1$ ungerade ist, die Nachkomastelle .5. Die Berechnung läßt sich dennoch immer ausführen, da ja nicht dieser Faktor berechnet wird, sondern (was durch die Klammern zusätzlich deutlich gemacht wird)

$$r = \frac{n(n-1)}{2}.$$

Es wird also zunächst das Produkt $n(n-1)$ berechnet, das dann durch 2 geteilt wird. Das Produkt $n(n-1)$ muß aber gerade sein, denn in ihm werden zwei aufeinanderfolgende Zahlen miteinander multipliziert. Von zwei aufeinanderfolgenden Zahlen muß genau eine gerade sein. Ist n gerade, so ist es n. Ist n ungerade, so ist $n-1$ gerade. Enthält aber ein Produkt eine gerade Zahl, so muß es selbst gerade sein, da es dann (implizit) den Faktor 2 enthält. Auch im zweiten Schritt treten daher keine Probleme auf.

Im dritten Schleifendurchlauf wird der Faktor $\frac{n-2}{3}$ hinzumultipliziert. Wieder wird aber nicht dieser Faktor berechnet, der Nachkommastellen haben könnte, sondern

$$r = \frac{\frac{n(n-1)}{2}(n-2)}{3} = \frac{n(n-1)(n-2)}{2 \cdot 3}.$$

Von drei aufeinanderfolgenden Zahlen muß aber mindestens eine durch 3 teilbar sein. Denn bei der Division durch 3 kann entweder der Rest 0, der Rest 1 oder der Rest 2 übrigbleiben. Prüfen wir diese drei Fälle. Wenn $n \bmod 3 = 0$, dann ist n durch 3 teilbar. Wenn $n \bmod 3 = 1$, dann ist offenbar $n-1$ durch 3 teilbar. Wenn $n \bmod 3 = 2$, dann ist offenbar $n-2$ durch 3 teilbar. Also ist das Produkt $n(n-1)(n-2)$ durch 3 teilbar und die Division ergibt keinen Rest.

In gleicher Weise kann man für die folgenden Schleifendurchläufe argumentieren. Nur eine Schwierigkeit ist noch zu berücksichtigen, die wir am Beispiel $i = 4$ verdeutlichen. Sei n durch 4 teilbar. Dann enthält das Produkt $n(n-1)(n-2)(n-3)$ offenbar den Faktor 4. Wir brauchen aber den Faktor 8, da wir sowohl durch 4, als auch (im zweiten Schleifendurchlauf) durch 2 teilen. Oder anders gesagt: Ist n durch 4 teilbar, so ist n offenbar die Zahl, die wir im zweiten Schleifendurchlauf als gerade identifiziert hatten und die sicherstellt, daß die Division durch 2 ausführbar ist. Dann ist aber ein Faktor 2 aus n herausgeteilt und das Ergebnis u.U. nicht mehr durch 4 teilbar (für $n = 4$ ist z.B. $\frac{n(n-1)}{2} = 6$). Die Division ist dennoch ausführbar, da unter den hinzukommenden Faktoren $n-2$ und $n-3$ wieder genau einer gerade sein muß und folglich den fehlenden Faktor 2 mitbringt. Anders gesagt: Unter den Zahlen n, $n-1$, $n-2$ und $n-3$ ist mindestens eine durch 4 und mindestens eine weitere, von der durch 4 teilbaren verschiedene, durch 2 teilbar. Ihr Produkt muß daher durch 8 teilbar sein.

Es tritt daher auch bei Divisionen durch Zahlen, die Vielfache von Zahlen sind, durch die schon vorher dividiert wurde, kein Rest auf.

Aufgabe C 3

```
int sqrsum (int n)
{                               /* --- compute sum of squares */
  if (n <= 0) return 0;         /* \sum_{i=0}^0 i^2 = 0^2 = 0 */
  return n*n +sqrsum(n-1);      /* \sum_{i=0}^n i^2 */
}  /* sqrsum() */               /* = n^2 +\sum_{i=0}^{n-1} i^2 */
```

Die Funktion `f3` aus Aufgabe C 3 berechnet die Summe der Quadrate der Zahlen 0 bis n. Sie beruht auf der folgenden Rekursionsformel:

$$\sum_{i=0}^{n} i^2 = \begin{cases} 0, & \text{falls } n = 0, \\ n^2 + \sum_{i=0}^{n-1} i^2, & \text{falls } n > 0. \end{cases}$$

Für Werte $n < 0$ liefert die Funktion stets 0 zurück.

Mit dieser Aufgabe und der Aufgabe C 5 sollte noch einmal an einfachen Beispielen das Prinzip der Rekursion eingeübt werden.

Aufgabe C 4

```
int printfrac (int p, int q, int n)
{                               /* --- print a fraction p/q with an */
                                /*     arbitrary number of decimals */
  int c;                        /* character counter */

  c = printf("%i", p /q);       /* print integer part of fraction */
  if (n <= 0) return c;         /* if no decimals requested, abort */
  c += printf(".");             /* print decimal point */
  while (--n >= 0) {            /* while not all decimals printed */
    p = (p %q) *10;             /* compute the remainder of p/q, */
    c += printf("%i", p /q);    /* multiply it by 10 to obtain the */
  }                             /* next digit, and print the digit */
  return c;                     /* return number of printed chars */
}  /* printfrac() */
```

Die Funktion `f4` aus Aufgabe C 4 gibt den Bruch p/q mit n Nachkommastellen aus (vgl. Beispielprogram `fraction.c`). Das verwendete Verfahren ist dem in der Grundschule gelernten Verfahren zur schriftlichen Division äquivalent. Zurückgegeben wird die Anzahl der ausgegebenen Zeichen.

Im einzelnen geht die Funktion so vor: Durch den ersten Aufruf der Funktion `printf` wird der ganzzahlige Anteil des Bruches p/q ausgegeben. Wie auch schon in der Lösung zur Aufgabe C 2 bemerkt wurde, werden bei einer Division ganzer Zahlen eventuell auftretende Nachkommastellen abgeschnitten. So auch hier. Die Funktion `printf` gibt bei erfolgreicher Ausführung die Anzahl der ausgegebenen Zeichen zurück, so daß hier die Anzahl der Vorkommastellen an c zugewiesen wird. Anschließend wird geprüft, ob überhaupt Nachkommastellen ausgegeben werden sollen (n gibt ja die Anzahl der auszugebenden Nachkommastellen an). Ist dies nicht der Fall, wird einfach die Anzahl der bisher (also für den Vorkommateil des Bruches p/q) ausgegebenen Zeichen zurückgegeben.

Sollen Nachkommastellen ausgegeben werden, so wird zunächst ein Dezimalpunkt ausgegeben. Da damit ein Zeichen ausgegeben wird, liefert `printf` den Wert 1 zurück, der zu der Zahl der bisher ausgegebenen Zeichen addiert wird.

In der dann folgenden Schleife werden die Nachkommastellen ausgegeben, und zwar in jedem Schleifendurchlauf eine Stelle, weswegen auch die Zahl n der noch auszugebenden Nachkommastellen in jedem Durchlauf um 1 verringert wird. Die Ausgabe einer Stelle erfolgt nach dem folgenden Schema: Der Bruch $\frac{p \bmod q}{q}$ stellt offenbar nur die Nachkommastellen des Bruches p/q dar. Durch die Multiplikation mit 10 wird die erste dieser Nachkommastellen in die Vorkommastellen gebracht. Da $0 \leq \frac{p \bmod q}{q} < 1$, muß die sich dabei ergebende Vorkommstelle zwischen 0 und 9 (einschließlich) liegen. Diese Vorkommastelle wird durch den Aufruf der Funktion `printf` ausgegeben. Weil die Vorkommastelle (siehe oben) zwischen 0 und 9 liegt, wird genau eine Ziffer ausgegeben, und `printf` liefert 1 zurück, was zu der Zahl der bisher ausgegebenen Stellen addiert wird. In jedem weiteren Schleifendurchlauf wird eine weitere Nachkommastelle in die Vorkommastellen gebracht und ausgegeben. Die im vorhergehenden Schritt ausgegebene Stelle wird durch die Modulo-Operation entfernt. Wenn so viele Stellen ausgegeben wurden, wie n angibt, wird die Schleife beendet, und die Zahl der insgesamt ausgegebenen Zeichen wird zurückgegeben.

Die obige Funktion kann den Bruch p/q auf beliebig viele Stellen genau ausgeben. Eine Ausgabe einer Fließkommazahl vom Datentyp `float` liefert dagegen höchstens 7, eine Ausgabe einer Fließkommazahl vom Datentyp `double` höchstens 16 (korrekte) Nachkommastellen. Benötigt man mehr Nachkommastellen, muß man daher auf eine Funktion wie die obige zurückgreifen. Dies kann z.B. von Bedeutung sein, wenn man bei Brüchen die Periode der Nachkommastellen feststellen möchte (bekanntlich läßt sich ja jeder Quotient ganzer Zahlen entweder als abbrechende oder als periodische Dezimalzahl darstellen). Als Beispiel untersuche man 3/97.

Aufgabe C 5

```
int mirror (char s[])
{                                /* --- print string s mirrored */
  int c;                         /* character counter */

  if (s[0] == '\0') return 0;    /* an empty string has length 0 */
  c = mirror(s+1);               /* print mirrored tail of the string, */
  putchar(s[0]);                 /* then print the first character */
  return c+1;                    /* return number of printed chars */
}  /* mirror() */
```

Die Funktion `f5` aus Aufgabe C 5 gibt die übergebene Zeichenkette `s` gespiegelt aus. Zurückgegeben wird die Zahl der ausgegebenen Zeichen, also die Länge der Zeichenkette `s`.

Die gespiegelte Ausgabe der Zeichenkette beruht auf der folgenden einfachen rekursiven Beschreibung: Gib zunächst die Restzeichenkette, die sich ergibt, wenn man das erste Zeichen der Zeichenkette entfernt, gespiegelt aus, dann gib das erste Zeichen aus. Der erste Teil dieser Anweisung entspricht dem rekursiven Aufruf der Funktion `mirror`, bei dem das erste Zeichen der Zeichenkette `s` übersprungen wird. Das erste Zeichen wird durch den Aufruf der Funktion `putchar` ausgegeben. Beide Funktionsausfrufe sind natürlich nur dann nötig, wenn die Zeichenkette `s` mindestens ein Zeichen enthält. Ist die Zeichenkette `s` leer, d.h. ist das Zeichen `\0` erreicht, das das Ende einer Zeichenkette markiert, kann die Funktion sofort zurückkehren.

Die Zählung der in `s` enthaltenen Zeichen erfolgt so: Die leere Zeichenkette hat offenbar die Länge 0, also wird bei einer leeren Zeichenkette (bei Erreichen des Zeichens `\0`), der Wert 0 zurückgegeben. Anderenfalls wird zu dem Wert, den der rekursive Aufruf lieferte, also der Zahl der Zeichen in der Restzeichenkette, 1, also das durch den Aufruf der Funktion `putchar` ausgegebene Zeichen, addiert und der sich ergebende Wert zurückgeliefert.

Aufgabe C 6

```
int match (char s[], char p[])
{                                /* --- test whether the string s */
                                 /*     matches the pattern p */
  switch (p[0]) {                /* evaluate next patter character */
    case '*': if (match(s, p+1)) return 1; /* '*' matches nothing */
              if (s[0] == '\0')  return 0; /* unmatched char in p */
              return match(s+1, p);        /* '*' matches char in s */
    case '?': if (s[0] == '\0')  return 0; /* nothing to match '?' */
              return match(s+1, p+1);      /* '?' matches char in s */
    default : if (s[0] != p[0])  return 0; /* unmatched char in s/p */
              if (s[0] == '\0')  return 1; /* end of string reached */
              return match(s+1, p+1);      /* chars in s and p match */
  }                              /* recursively match s and p */
}  /* match() */
```

Die Funktion `f6` aus Aufgabe C 6 prüft, ob eine Zeichenkette `s` (string) auf ein gegebenes Muster `p` (pattern) paßt (vgl. Beispielprogramm `match.c`). Das Muster darf zwei Arten sogenannter „Jokerzeichen“ oder „Wildcards“ enthalten: den Stern `*` und das Fragezeichen `?`. Der Stern steht für eine beliebige Anzahl (auch 0) beliebiger Zeichen. Das Fragezeichen steht für genau ein beliebiges Zeichen. Eine Zeichenkette paßt auf ein Muster, wenn es eine Möglichkeit gibt, in dem Muster alle Sterne entweder durch kein, durch ein oder durch mehrere Zeichen und alle Fragezeichen durch jeweils genau ein Zeichen zu ersetzen, so daß man die Zeichenkette erhält. Diese Art des Mustervergleichs wird z.B. von den Kommandos `ls` (Unix) oder `dir` (MSDOS) verwendet.

Um zu verstehen, wie diese Funktion arbeitet, nehmen wir zunächst an, daß das Muster keine „Jokerzeichen“, also keinen Stern und kein Fragezeichen enthält. Dann läßt sich das Passen einer Zeichenkette auf ein Muster offenbar rekursiv so beschreiben: Die Zeichenkette `s` paßt auf das Muster `p`, wenn das erste Zeichen der Zeichenkette und das erste Zeichen des Musters übereinstimmen und der Rest der Zeichenkette `s` (ab dem zweiten Zeichen) auf den Rest des Musters `p` (ab dem zweiten Zeichen) paßt.

Genau dies wird im `default`-Fall der `switch`-Anweisung geprüft. Wenn die ersten Zeichen der Zeichenkette `s` und des Musters `p` nicht gleich sind, dann paßt die Zeichenkette nicht auf das Muster. Folglich wird $0 \mathrel{\hat{=}}$ falsch zurückgegeben. Dann wird geprüft, ob das Ende der Zeichenkette (und damit gleichzeitig das Ende des Musters) erreicht ist, indem mit dem speziellen Zeichen `\0`, das das Ende einer Zeichenkette markiert, verglichen wird (vgl. Aufgabe C 5). Ist dies der Fall, so ist kein rekursiver Aufruf nötig, da ja keine Restzeichenkette und kein Restmuster vorhanden sind. Vielmehr kann in diesem Fall sofort $1 \mathrel{\hat{=}}$ wahr (also: die Zeichenkette paßt auf das Muster) zurückgegeben werden. Ist man noch nicht am Ende der Zeichenkette angelangt, wird — wie oben angegeben — durch einen rekursiven Aufruf der Rest der Zeichenkette `s` mit dem Rest des Musters `p` verglichen.

Nehmen wir nun an, das Muster enthalte ein Fragezeichen. Da ein Fragezeichen für ein beliebiges Zeichen steht, braucht in diesem Fall nicht auf Übereinstimmung mit der Zeichenkette verglichen zu werden. Aber die Zeichenkette muß noch ein Zeichen enthalten. Das Fragezeichen steht *nicht* für 0 Zeichen. Ist man bereits am Ende der Zeichenkette angelangt, wird daher $0 \mathrel{\hat{=}}$ falsch zurückgegeben, denn in diesem Fall fordert das Muster `p` durch das Fragezeichen ein weiteres Zeichen in der Zeichenkette `s`, das diese nicht enthält. Anderenfalls wird wie im `default`-Fall fortgefahren, d.h. es wird durch einen rekursiven Aufruf der Rest der Zeichenkette `s` mit dem Rest des Musters `p` verglichen.

Schließlich betrachten wir den schwierigsten Fall, nämlich das Auftreten eines Sterns in dem Muster `p`. Die Schwierigkeit besteht hier darin, daß man nicht direkt entscheiden kann, ob der Stern noch für das nächste Zeichen in der Zeichenkette `s` steht oder nicht. Dies muß ausprobiert werden. Es wird daher zunächst geprüft, ob die Zeichenkette `s` auf das Muster `p` paßt, wenn der Stern für kein Zeichen in `s` steht. Dazu wird im rekursiven Aufruf der Stern einfach übersprungen, d.h. der Rest des Musters `p` mit der ganzen Zeichenkette `s` verglichen. Führt dies zum Erfolg, so paßt offenbar die Zeichenkette `s` auf das Muster `p`, nämlich eben dadurch, daß der Stern für kein Zeichen in der Zeichenkette `s` steht.

Schlägt der Test dagegen fehl, so wird geprüft, ob man schon am Ende der Zeichenkette `s` angelangt ist. Ist dies der Fall, so kann, wegen des Fehlschlagens des vorhergehenden rekursiven Tests, die Zeichenkette nicht auf das Muster passen. Denn ein Muster, auf das eine leere Zeichenkette paßt, muß entweder selbst leer sein oder darf nur Sterne enthalten. Da aber Sterne durch den direkt vorher ausgeführten rekursiven Aufruf (und die in ihm dann eventuell erfolgenden weiteren rekursiven Aufrufe) übersprungen werden, hätte in diesem Fall der Test erfolgreich verlaufen müssen. Da er nicht erfolgreich war, muß das Muster `p` mindestens ein Fragezeichen oder ein beliebiges anderes Zeichen enthalten. In einer leeren Zeichenkette gibt es nichts, was zu einem dieser Zeichen paßt, also paßt die Zeichenkette `s` nicht auf das Muster `p`.

Enthält die Zeichenkette `s` dagegen weitere Zeichen, so wird geprüft, ob die Zeichenkette `s` auf das Muster `p` paßt, wenn der Stern (auch) für das erste Zeichen der Zeichenkette steht. Dazu wird in einem rekursiven Aufruf das erste Zeichen der Zeichenkette übersprungen. Der Stern wird jedoch beibehalten, da er ja auch für mehr als ein Zeichen stehen kann. Ob er für mehr als ein Zeichen steht, wird dann in der Rekursion geprüft.

Diese Aufgabe demonstriert besonders eindrucksvoll die Mächtigkeit der Rekursion. Mit ihrer Hilfe kann der Mustervergleich durch eine sehr kurze Funktion realisiert werden. Zum Vergleich versuche man, eine Funktion zu schreiben, die das gleiche Problem ohne rekursive Aufrufe löst.

Christian Borgelt