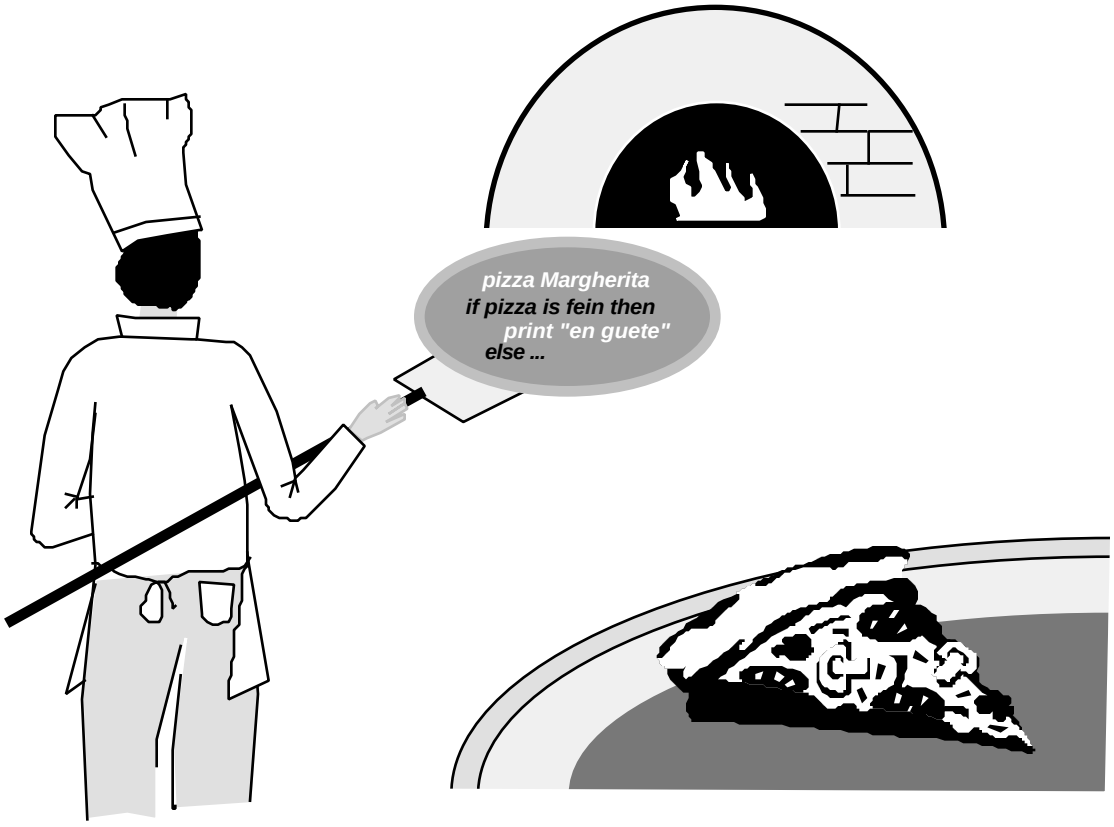
## Gruppenunterricht zum Thema:

# Paradigmen von Programmiersprachen




| | |
|---|---|
| Fach: | Informatik |
| Schultyp: | Fachhochschule / Ingenieurschule / HTL |
| Schulstufe: | Vertiefung Informatik |
| Vorkenntnisse: | Imperative Programmiersprache<br>Aussagenlogik |
| Bearbeitungsdauer: | 8 Lektionen à 45 Minuten |
| Autoren: | **Daniel Brennwalder, Christoph Stamm** |
| Betreuer: | Dr. Werner Hartmann |
| Fassung vom: | 6. November 1997 |

## Einführung

Unter einem Paradigma versteht man ein Muster oder ein charakteristisches Beispiel.

Informatiker unterscheiden heute mehr als zehn verschiedene Paradigmen von Programmiersprachen. Jedes dieser Paradigmen besteht aus einer Anzahl von Konzepten, welche den Entwurf von Software prägen und damit auch direkt die Struktur des Programms bestimmen. Diese Konzepte haben einen grossen Einfluss auf die Art, wie wir Probleme und deren Lösungswege formulieren. Sie beeinflussen somit auch direkt unser Denken.

Die Umsetzung eines derart formulierten Lösungsweges in eine Programmiersprache kann nur dann effizient geschehen, wenn die Programmiersprache die Konzepte des Paradigmas gut wiedergibt. Die heutigen Programmiersprachen unterstützen jedoch selten nur *ein* Paradigma, vielmehr werden Konzepte von verschiedenen Paradigmen benutzt. Als Beispiel sei hier die prozedurale (imperative) Programmiersprache Pascal aufgeführt, die auch Funktionen, Rekursion und dynamische Speicherallokation mittels Zeigern anbietet. Diese drei Konzepte sind aus anderen Paradigmen entliehen.

Sie lernen in den nächsten acht Lektionen vier neue Paradigmen kennen. Die Auswahl der vier beruht darauf, dass drei davon eine grosse Bedeutung und Verbreitung in Forschung und Wirtschaft haben, und dass das vierte Paradigma eine mögliche Alternative für die Zukunft darstellt.

# Inhaltsverzeichnis

# Arbeitsanleitung

## Unterricht mit der Puzzle-Methode, wie geht das?

Die Klasse wird wenn möglich in vier gleich grosse Gruppen eingeteilt. Jeder Gruppe wird ein Paradigma zugeteilt, das es zu bearbeiten gilt. Die ganze Unterrichtseinheit gliedert sich in drei Phasen:

### 1. Selbststudium (Zeit: 3 Lektionen)

Sie bearbeiten das Thema Ihrer Gruppe selbständig.

### 2. Expertenrunde (Zeit: 2 Lektionen)

Sie treffen sich mit den anderen Mitgliedern Ihrer Gruppe. Jetzt können Sie Unklarheiten diskutieren. Dann besprechen Sie, was Sie Ihren Mitstudenten unterrichten möchten.

### 3. Unterrichtsrunde (Zeit: 3 Lektionen)

Sie haben sich mit dem Stoff auseinandergesetzt. Nun sind Sie in der Lage, Ihre Mitstudenten zu unterrichten. Dazu haben Sie für Ihr Thema 30 Minuten Zeit. Bei den drei anderen Themen nehmen Sie wieder die «Studentenrolle» ein. Nun unterrichten Ihre Kollegen.

# Lernziele

Nach dieser Unterrichtseinheit kennen Sie vier neue wichtige Paradigmen bei ihrem Namen. Sie verstehen deren grundlegenden Ideen und Konzepte. Ausserdem sind Sie in der Lage, die zentralen Punkte zu jedem Paradigma aufzuzählen und zu erklären. Dabei erkennen Sie auch die wesentlichsten Unterschiede zum imperativen Paradigma.

# Objektorientiertes Paradigma: Arbeitsblätter fürs Selbststudium

## 0 So gehen Sie vor

### Pizzabeispiel

Es wurden Ihnen zwei Beispielprogramme abgegeben. Beide beschreiben, wie eine Pizza zubereitet wird. Das eine ist im imperativen, das andere im objektorientierten Paradigma formuliert. Lesen Sie zuerst das imperative Beispiel durch. Dadurch bekommen Sie eine Idee, wie die Pizza gemacht werden soll. Dies sollte Ihnen keine Schwierigkeiten bereiten. Gehen Sie danach zum objektorientierten Pizzabeispiel über. Studieren Sie auch dieses und versuchen Sie, die Fragen zu beantworten. Dazu müssen Sie das Beispiel nicht im Detail verstanden haben.

Beachten Sie die untenstehenden Bemerkungen zum Pizzabeispiel. Die Lehrperson steht Ihnen nur zur Verfügung, falls Sie Verständnisschwierigkeiten haben.

Für diesen Teil sollten Sie nicht mehr als 30 Minuten einsetzen.

### Vermittlung des Lernstoffes

In diesem Teil lernen Sie die wichtigsten Eigenschaften und Begriffe des objektorientierten Paradigmas kennen. Schauen Sie jeweils die Lösung nach, wenn Sie eine Kontrollfrage beantwortet haben. Gehen Sie erst weiter, wenn Sie den Stoff verstanden haben.

Nehmen Sie sich für diesen Teil 75 Minuten Zeit.

### Studenten-Lernkontrolle

Hier können Sie überprüfen, ob Sie den Lernstoff verstanden haben. Seien Sie ehrlich zu sich selber: Falls Sie die Lösung nicht alleine finden, gehen Sie zurück zum Abschnitt «Vermittlung des Lernstoffes» und suchen Sie dort nach Ideen.

Sie haben für diesen Teil 30 Minuten zur Verfügung.

## 1 Bemerkungen zum Pizzabeispiel

Beginnen Sie mit dem Lesen des Programmes beim letzten Abschnitt. Lesen Sie eine Zeile und sehen Sie nach, was diese Zeile bewirkt.

*tomaten at: i put: x* bedeutet *tomaten[i]:=x*

*tomaten at: i receive: aktion*: Die Tomate mit der Nummer *i* wird veranlasst, die *aktion* auszuführen.

## 2 Vermittlung des Lernstoffes

In diesem Abschnitt werden Ihnen die wichtigsten Punkte des objektorientierten Paradigmas vorgestellt.

### 2.1 Klassenhierarchie

Im objektorientierten Paradigma existieren keine Typen, wie Sie sie vom imperativen Paradigma her gewohnt sind. Es wird mit Klassen gearbeitet.

Imperativ

```
type Tomate is record
      zustand: (frisch, geschält, gehackt);
end Tomate;

type Mozzarella is record
      zustand: (frisch, gerieben, verkleinert);
end Mozzarella;
```

Objektorientiert

```
classname                   Zugaben
superclass                  -
instanceVariableNames       zustand

classname                   Tomate
superclass                  Zugaben
instanceVariableNames       -

classname                   Mozzarella
superclass                  Zugaben
instanceVariableNames       -
```

In den objektorientierten Klassen des obenstehenden Beispiels besteht eine Hierarchie, die Sie sicher leicht herausfinden werden.

### Kontrollfrage 2.1a

Zeichnen Sie bitte eine Struktur, die diese Hierarchie zum Ausdruck bringt. Vorschlag: Verwenden Sie eine Baumstruktur. Die «Kinderknoten» sind vom «Vaterknoten» abhängig.

### Hilfe

Achten Sie auf die Bezeichnungen bei der Deklaration der Klassen.

### Kontrollfrage 2.1b

Warum werden wohl gerade diese Klassen zu einer Hierarchie zusammengefasst? Halten Sie Ihre Vermutung in zwei bis drei Sätzen fest.

### Hilfe

Vergleichen Sie die Instanzvariablen der Klassen *PizzaTeig* und *Tomate.* Ergibt eine Instanzvariable *dicke* für eine Tomate einen Sinn?

## 2.2 Vererbung

Die Klassenhierarchie bietet bestimmte Vorteile. Einer davon ist die Vererbung. Sie kennen den folgenden Programmausschnitt.

```
classname                Zugaben
superclass               -
instanceVariableNames    zustand
instanceMethods
   Zustand: Anfangzustand
      zustand:=anfangszustand.

classname                Tomate
superclass               Zugaben
instanceVariableNames    -
instanceMethods
   Schälen
      zustand:=geschält.
   Hacken
      zustand:=gehackt.

classname                Mozzarella
superclass               Zugaben
instanceVariableNames    -
instanceMethods
   Verkleinern
      zustand:=verkleinert.
```

### Kontrollfrage 2.2a

Notieren Sie sich die Klassen, die «Dinge» erben. Schreiben Sie dazu zu jeder Klasse auf, was sie von wem erbt.

### Hilfe

Betrachten Sie die Lösung von Frage 2.1a.

### Kontrollfrage 2.2b

Übersetzen Sie das vorangegangene Programmstück in das imperative Paradigma. Benutzen Sie dazu die im imperativen Beispiel verwendete Syntax oder eine imperative Programmiersprache, die Sie gut kennen.

### Hinweise

Schreiben Sie nicht einfach das imperative Beispiel ab.

Benutzen Sie nur die folgenden Datenstrukturen:

```
type Tomate is record
   zustand:(frisch, geschält, gehackt);
end Tomate;

type Mozzarella is record
   zustand:(frisch, gerieben, verkleinert);
end Mozzarella;
```

Verwenden Sie keine Initialisierung der Form:

```
zustand:(frisch,geschält,gehackt):=frisch;
```

Schreiben Sie eine Prozedur für die Initialisierung.

### Hilfe

Schreiben Sie folgende Prozeduren: *AnfangsZustandTomate*, *AnfangsZustandKäse*, *Hacken*, *Schälen*, *Verkleinern*.

### Kontrollfrage 2.2c

Welchen Vorteil bringt die Vererbung im Vergleich zur imperativen Lösung? Richten Sie Ihr Augenmerk auf die Initialisierung des Anfangszustandes. Zwei bis drei Sätze genügen als Antwort.

### Hilfe

Betrachten Sie den Aufwand, den Sie in beiden Fällen betreiben müssen. Welche Lösung ist eleganter?

## 2.3 Nachrichten und Methoden

Wir kommen nun zu einem Punkt, der Sie sicher ein bisschen verwirrt hat. Klassen besitzen Methoden: Instanzmethoden und Klassenmethoden.

Methoden führen gewisse Aktionen aus. Um eine Methode zu starten, schicken wir einem Objekt eine Nachricht.

Betrachten wir folgenden Code:

```
teig:=PizzaTeig new.
```

Dies bedeutet: Hallo Objekt *PizzaTeig*, führe bitte die Methode *new* aus. Ein bisschen formeller: Der Klasse *PizzaTeig* wird die Nachricht *new* geschickt. Das Resultat der Methode *new* wird der Variable *teig* zugeordnet. Dabei ist *new* eine Klassenmethode, das heisst eine Methode, die nur von einer Klasse ausgeführt werden kann. Wir haben in unserem Beispiel angenommen, dass die Methode *new* für alle Klassen schon vordefiniert ist, das heisst, sie muss nicht programmiert werden.

Nun ist *teig* eine Instanz der Klasse *PizzaTeig*. Eine Instanz einer Klasse besitzt Instanzvariablen und Instanzmethoden. Für den Teig sind dies:

```
classname                PizzaTeig
```

```
superclass                -
instanceVariableNames     radius dicke zustand
instanceMethods
   Einkaufen
      radius:=5.
      dicke:=4.5.
      zustand:=geknetet.
   AusrollenAufRadius: sollradius UndDicke: solldicke
      while ((radius <= sollradius) OR (dicke >= solldicke))
         [dicke:=dicke/1.5. radius:=radius+2.]
      zustand:=ausgerollt.
```

Instanzen sind auch Objekte. Mit einer Nachricht an eine Instanz kann eine *Instanzmethode* ausgeführt werden. Zum Beispiel:

```
teig AusrollenAufRadius: 15 UndDicke: 0.5.
```

### Kontrollfrage 2.3a

Suchen Sie im imperativen Beispiel die Zeilen, welche sinngemäss das gleiche bewirken wie *teig:=PizzaTeig new* und *teig AusrollenAufRadius: 15 UndDicke: 0.5*.

### Hilfe

*teig:=PizzaTeig new* erzeugt eine Instanz, gewissermassen eine Variable des Typs *PizzaTeig*.

### Kontrollfrage 2.3b

Versuchen Sie, den grundlegenden Unterschied der objektorientierten und imperativen Denkweise zu finden.

Notieren Sie Ihre Antwort. Schreiben Sie dazu etwa fünf Sätze. Grübeln Sie nicht zu lange darüber nach. Sollten Sie Mühe haben, den Unterschied sauber zu formulieren, ist das nicht weiter schlimm. Schreiben sie auf, was Sie denken. Der grundlegende Unterschied ist Ihnen sicher nicht entgangen.

### Hinweis

Natürlich könnte man hier diverse Unterschiede aufzählen. Suchen Sie in folgender Richtung: Normalerweise verlangt eine Funktion in imperativen Programmen gewisse Parameter. Diese Parameter sind von einem bestimmten Typ. Man fragt sich also, auf welche Datentypen bestimmte Operationen anwendbar sind.

### Hilfe

Vergleichen Sie nun mit einer objektorientierten Formulierung: *tomate Zustand: frisch. mozzarella Zustand: frisch.* Hier ist *tomate* eine Instanz der Klasse *Tomate* und *mozzarella* eine Instanz der Klasse Mozzarella. Die Methode *Zustand* ist als Operation aufzufassen.

## 2.4 Einkapselung

Wie wir gesehen haben, besteht ein Objekt aus Variablen und Methoden. Die Methoden werden dazu benutzt, anderen Objekten Nachrichten zu schicken. Nur über eine Nachricht an ein Objekt können dessen Instanzvariablen verändert werden. Und darauf bezieht sich der Begriff Einkapselung: Ohne Benutzung einer Methode kann nicht auf die Instanzvariablen zugegriffen werden. Diese sind gewissermassen «eingekapselt».

Machen wir dazu einen Vergleich mit dem imperativen Pizzabeispiel. Dort kann an jeder Stelle im Code auf das Feld *Zustand* einer Variable des Typs *Tomate* zugegriffen werden.

Wenn Sie im objektorientierten Paradigma den Zustand einer Instanz der Klasse *Tomate* wissen wollen, so müssen Sie die *Tomate* fragen.

Noch eine letzte Aufgabe in diesem Abschnitt.

### Kontrollfrage 2.4

Ergänzen Sie das objektorientierte Pizzaprogramm so, dass Sie den Zustand von Instanzen der Klassen *Tomate* und *Mozzarella* abfragen können.

### Hinweis

Wenn Sie einen Wert *x* zurückgeben wollen, dann benutzen Sie *return x*.

### Hilfe

Schreiben Sie eine zusätzliche Methode. Achten Sie darauf, in welcher Klasse Sie diese Methode einfügen.

## 3 Studenten-Lernkontrolle

Sie sind nun beim letzten Abschnitt des Selbststudiums angelangt. Hier können Sie prüfen, ob Sie alles verstanden haben. Falls Sie Probleme bei der Lösung dieser Aufgaben haben, gehen Sie bitte nochmals zum Abschnitt «Vermittlung des Lernstoffes» zurück. Lesen Sie die Punkte nochmals durch, die Sie noch nicht ganz verstanden haben. Wenn Sie den Stoff verstanden haben, aber trotzdem nicht weiterkommen, lesen Sie die Hinweise durch.

Für die Lernkontrolle haben Sie 30 Minuten Zeit. Sie machen diese Übung nur für sich selbst, das heisst sie wird nicht bewertet.

### Aufgabe 3.1

In dieser Aufgabe geht es um die Kontoverwaltung einer Bank.

Richten Sie zwei Arten von Konten ein:

Sparkonto

- Zins 4.5%.

- Bezugslimite: Es dürfen maximal 5000.- pro Bezug abgehoben werden.

- Das Konto darf nicht überzogen werden.

Gehaltskonto

- Zins 2.5%.

- Das Konto darf um maximal 2000.- überzogen werden.

Implementieren Sie folgende Operationen:

Kontostand abfragen

Einzahlen

Abheben

Erstellen Sie die verschiedenen Klassen mit Instanzvariablen und Instanzmethoden. Benützen Sie dazu die bisher in dieser Unterrichtseinheit verwendete Syntax.

### Hinweise

Falls beim Abheben eine Bedingung verletzt werden würde, dann wird keine Aktion vorgenommen.

Kümmern Sie sich nicht um die Initialisierung der Konten. Implementieren Sie nur die oben erwähnten Operationen.

Identifizieren Sie ein Konto über die Kontonummer. Verwenden Sie keine Namen der Kontoinhaber.

Syntax des *If*-Statements: (condition) ifTrue:[...] if False:[...].

Ihre Lösung ist in Ordnung, wenn Sie ein korrektes Programm schreiben und die Vorteile des objektorientierten Paradigmas ausnutzen.

### Hilfe

Ein Sparkonto und ein Gehaltskonto sind ähnliche Objekte. Erstellen Sie eine vernünftige Klassenhierarchie. Überlegen Sie sich, welche Instanzvariablen und Instanzmethoden vererbt werden können.

# Objektorientiertes Paradigma: Lösungsblätter zum Selbststudium

## 1 Lösungen zum Pizzabeispiel

### Antwort 1.1

Zustände, die die Tomaten annehmen: *frisch, geschält, gehackt.*

### Antwort 1.2

Einkaufsliste für den Pizzaiolo:

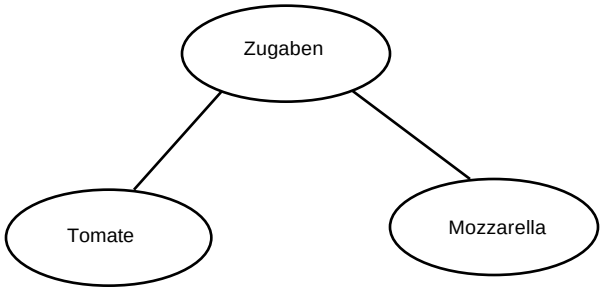
- *1 Pizzateig*
- *2 Tomaten*
- *2 Mozzarella*
- *Pfeffer*

### Antwort 1.3

Zustände, die der Teig annimmt: *geknetet, ausgerollt.*

## 2 Lösungen der Kontrollfragen

### Antwort 2.1a




### Antwort 2.1b

Diese Klassen werden zu einer Hierarchie zusammengefasst, weil eine Tomate und ein Mozzarella Zugaben für die Pizza sind. Sie sind also ähnliche Objekte. Beide werden durch einen Zustand charakterisiert. Sie haben also gewisse gemeinsame Grundeigenschaften. Über die Grundeigenschaften hinaus unterscheiden sie sich aber. Sie haben auch spezielle Eigenschaften. So muss zum Beispiel eine Tomate geschält werden. Bei einem Mozzarella ist dies nicht gerade sinnvoll.

Zu den Bezeichnungen: Die obere Klasse der Hierarchie wird als *Oberklasse* bezeichnet, in unserem Fall Zugaben. Auf ihr aufgebaut werden die *Unterklassen* Tomate und Mozzarella.

### Antwort 2.2a

Die Unterklassen erben die Instanzvariablen und die Instanzmethoden der Oberklasse. Konkret: Die Klassen *Tomate* und *Mozzarrella* erben die Instanzvariable *zustand* und die Instanzmethode *Zustand.*

Auf die Begriffe Instanzvariable und Instanzmethode werden wir noch eingehen. Betrachten Sie eine Instanzvariable als eine «normale» Variable und eine Instanzmethode als eine Prozedur.

### Antwort 2.2b

Ihr Code sollte etwa wie folgt aussehen:

```
type Tomate is record
   zustand: (frisch, geschält, gehackt);
end Tomate;

type Mozzarella is record
   zustand: (frisch, gerieben, verkleinert);
end Mozzarella;

procedure AnfangsZustandTomate(t: in out Tomate) is
begin
   t.zustand:= frisch;
end AnfangszustandTomate;

procedure AnfangsZustandKäse(m: in out Mozzarella) is
begin
   m.zustand:= frisch;
end AnfangszustandKäse;

procedure Hacken(t: in out Tomate) is
begin
   t.zustand:= gehackt;
end Hacken;

procedure Schälen(t: in out Tomate) is
begin
   t.zustand:= geschält;
end Schälen;

procedure Verkleinern(m: in out Mozzarella) is
begin
   m.zustand:= verkleinert;
end Verkleinern;
```

### Antwort 2.2c

Für die Initialisierung des Anfangszustandes brauchen Sie beim imperativen Paradigma verschiede Prozeduren. Eine für die Tomaten und eine für den Mozzarella. Dies, weil *Tomaten* und *Mozzarella* unterschiedliche Typen sind.

Der Vorteil des objektorientierten Paradigmas liegt nun darin, dass dieses Problem gar nicht auftritt. Die Initialisierung wird durch die Instanzmethode *Zustand* der Oberklasse vorgenommen. Wie schon erwähnt, werden Instanzvariablen und Instanzmethoden an die Unterklassen vererbt.

### Antwort 2.3a

*teig:=PizzaTeig new* entspricht sinngemäss *teig: PizzaTeig.*

*teig AusrollenAufRadius: 15 UndDicke: 0.5* entspricht *TeigAusrollen (15, 0.5, teig).*

### Antwort 2.3b

Beim objektorientierten Paradigma betrachten wir Objekte. Diese Objekte können bestimmte Operationen ausführen. Operationen werden ausgeführt, indem Methoden des Objektes gestartet werden.

Verschiedene Objekte reagieren auf die gleiche Nachricht unterschiedlich. Betrachten wir folgenden Code:

```
   teig:=PizzaTeig new.
   teig Einkaufen.
   zutaten:=Ingredienzen new.
   zutaten Einkaufen.
```

Das Objekt *teig* reagiert auf die Nachricht *Einkaufen* völlig anders als das Objekt *zutaten.* Die Methoden sind also den Objekten zugeordnet.

Im Unterschied dazu werden im imperativen Paradigma Prozeduren definiert, welchen dann Variablen übergeben werden können. Diese Prozeduren können aber nicht auf beliebige Datentypen angewendet werden. Hier werden also den Prozeduren zulässige Datentypen zugeordnet. Als Beispiel dazu:

```
procedure TeigEinkaufen(teig: in out PizzaTeig) is
begin
   radius: Mass:=5;
   dicke: Mass:=4.5;
```

```
   teig.zustand:=geknetet;
end TeigEinkaufen;

procedure ZutatenEinkaufen(zutaten: in out Ingredienzen) is
begin
   for t in zutaten.tomaten'range loop
      zutaten.tomaten[t].zustand:=frisch;
   end loop;
   for t in zutaten.käse'range loop
      zutaten.käse[t].zustand:=frisch;
   end loop;
end ZutatenEinkaufen;
```

## Antwort 2.4

```
classname                Zugaben
superclass               -
instanceVariableNames    zustand
instanceMethods
   Zustand: anfangzustand
      zustand:=anfangszustand.
   AktuellerZustand
      return zustand.
```

Da *Tomate* und *Mozzarella* Unterklassen der Oberklasse *Zugaben* sind, haben Instanzen der Klassen *Tomate* und *Mozzarella* die Methode *AktuellerZustand* verfügbar.

# 3 Lösungen zur Studenten-Lernkontrolle

## Lösung 3.1

```
classname                Konto
superclass               -
instanceVariableNames    kStand zins
instanceMethods
   Kontostand
      return kStand.
   Einzahlen: betrag
      kStand:=kStand + betrag.

classname                Sparkonto
superclass               Konto
instanceVariableNames    bezugsLimite
instanceMethods
   Abheben: betrag
      ((betrag <= bezugsLimite) AND (kStand >= betrag))
      ifTrue:[kStand:=kStand - betrag.].

classname                Gehaltskonto
superclass               Konto
instanceVariableNames    überzugsLimite
instanceMethods
   Abheben: betrag
      (kStand - betrag >= überzugsLimite)
      ifTrue:[kStand:=kStand - betrag.].
```

## Bemerkungen

Selbstverständlich ist der obige Code nicht der einzig richtige. So sind zum Beispiel die Variablen *monatsLimite* und *überzugsLimite* nicht unbedingt notwendig. Die Zahlen könnten direkt ins Programm geschrieben werden.

Nachrichten an die Instanzen der Konten können nach folgendem Muster verfasst werden: *sp1 Abheben: 100.* Mit *sp1* wird das Sparkonto Nummer 1 bezeichnet.

Falls Sie eine Klassenhierarchie entwickelt haben, die sich grundlegend von dieser Lösung unterscheidet, dann vergleichen Sie Ihre mit dieser Lösung. Wenn Sie sicher sind, dass Ihre Lösung korrekt ist und die Vererbung sinnvoll genutzt wird, dann sind Sie jetzt fertig. Wenn nicht, lesen Sie nochmals die Abschnitte «Klassenhierarchie» und «Vererbung» durch. Vergewissern Sie sich, dass die vorgeschlagene Lösung sinnvoll ist.

# Funktionales Paradigma: Arbeitsblätter fürs Selbststudium

## 0 So gehen Sie vor

### Pizzabeispiel

Es wurden Ihnen zwei Beispielprogramme abgegeben. Beide beschreiben, wie eine Pizza zubereitet wird. Das eine ist im imperativen, das andere im funktionalen Paradigma formuliert. Lesen Sie zuerst das imperative Beispiel durch. Dadurch bekommen Sie eine Idee, wie die Pizza gemacht werden soll. Dies sollte Ihnen keine Schwierigkeiten bereiten. Gehen Sie danach zum funktionalen Pizzabeispiel über. Studieren Sie auch dieses und versuchen Sie, die Fragen zu beantworten. Dazu müssen Sie das Beispiel nicht im Detail verstanden haben.

Beachten Sie die untenstehenden Bemerkungen zum Pizzabeispiel. Die Lehrperson steht Ihnen nur zur Verfügung, falls Sie Verständnisschwierigkeiten haben.

Für diesen Teil sollten Sie nicht mehr als 30 Minuten einsetzen.

### Vermittlung des Lernstoffes

In diesem Teil lernen Sie die wichtigsten Eigenschaften und Begriffe des funktionalen Paradigmas kennen. Schauen Sie jeweils die Lösung nach, wenn Sie eine Kontrollfrage beantwortet haben. Gehen Sie erst weiter, wenn Sie den Stoff verstanden haben.

Nehmen Sie sich für diesen Teil 75 Minuten Zeit.

### Studenten-Lernkontrolle

Hier können Sie überprüfen, ob Sie den Lernstoff verstanden haben. Seien Sie ehrlich zu sich selber: Falls Sie die Lösung nicht alleine finden, gehen Sie zurück zum Abschnitt «Vermittlung des Lernstoffes» und suchen Sie dort nach Ideen.

Sie haben für diesen Teil 30 Minuten zur Verfügung.

## 1 Bemerkungen zum Pizzabeispiel

Sie bearbeiten das *funktionale* Paradigma. Versuchen Sie, im Programm die *Funktionen* zu finden.

Eine Tomate kann entweder *frisch, geschält* oder *gehackt* sein:
*data TomatenZustand == frisch++geschält++gehackt*

Ein *PizzaTeig* ist gekenntzeichnet durch einen *Radius,* eine *Dicke* und einen *Teigzustand*:
*type PizzaTeig == Radius#Dicke#TeigZustand*

Ein Element *a* wird an die erste Stelle einer Liste mit den Elementen *b, c* und *d* gesetzt.
*a :: [b c d]* wird *[a b c d]*

*[]* bezeichnet die leere Liste.

## 2 Vermittlung des Lernstoffes

### 2.1 Definieren von Funktionen

Falls Sie noch nie ein in einer funktionalen Sprache geschriebenes Programm gesehen haben, wird sie das Pizzaprogramm wahrscheinlich ein bisschen verblüfft haben. Eine Sammlung von «komischen Funktionen», die zudem noch eine Pizza herstellen sollen? Das kann ja nicht gut gehen!

So schlimm ist es aber nicht, mit den Funktionen «funktioniert» es tatsächlich.

Beim funktionalen Paradigma formuliert man ein Problem, indem man es als eine *Sammlung von Funktionen* darstellt. Funktionen werden dargestellt, wie Sie es aus der Mathematik gewohnt sind.

Ein Beispiel:

```
dec sqr: num -> num;
--- sqr(x) <= x*x;
```

In der Analysis würde man schreiben:

$$sqr: Z \rightarrow Z$$
$$sqr(x) = x \ \ x$$

Als Definitions- und Wertebereich kann statt den ganzen Zahlen zum Beispiel auch eine Liste von Gewürzen verwendet werden:

```
dec Würzen:list(Gewürz) -> list(Gewürz);
```

## 2.2 Pattern Matching

Wir lernen nun eines der wichtigsten Hilfsmittel des funktionalen Programmierens kennen: Pattern Matching. Mit Pattern Matching wird das Übereinstimmen von Mustern bezeichnet. Auf den Begriff wird in der Lösung dieser Aufgabe näher eingegangen.

Betrachten wir folgendes Programmstück:

```
dec Backen:Pizza#BackZeit#Temperatur -> Pizza;
--- Backen((teig,zutaten,belegt),0,t) <= (teig,zutaten,gebacken);
--- Backen((teig,zutaten,belegt),dauer,temperatur) <=
        Backen ((teig,zutaten,belegt),dauer - 1,temperatur);
--- Backen((teig,zutaten,unbearbeitet),d,t) <=
        Backen(PizzaBelegen(teig,zutaten,unbearbeitet),d,t);
```

### Kontrollfrage 2.2

Übersetzen Sie bitte dieses Programmstück in die Umgangssprache. Versuchen Sie eine Wenn-Dann-Beziehung zu finden. Tun Sie das für die drei Fälle, die oben aufgeführt sind.

### Hilfe

Beispiel für die erste Zeile: Wenn der Teig mit den Zutaten belegt und die Backzeit abgelaufen ist, dann ist die Pizza gebacken.

## 2.3 Der Ersatz für den Range-Loop

Vielleicht haben Sie bemerkt: Im ganzen funktionalen Programm finden Sie keinen einzigen Loop (for-Schleife oder while-Schleife). Dies aus gutem Grund: Das funktionale Paradigma kennt kein derartiges Konstrukt.

Betrachten Sie den folgenden imperativ formulierten Programmabschnitt.

```
type Tomate is record
   zustand: (frisch, geschält, gehackt):=frisch;
end Tomate;

tomaten: array N of Tomate;
```

```
procedure TomatenSchälen(t: in out tomaten) is
begin
   for i in t'range loop
      t[i].zustand:=geschält;
   end loop;
end TomatenSchälen;
```

### Kontrollfrage 2.3

Übersetzen Sie nun bitte diesen Abschnitt in das funktionale Paradigma. Benutzen Sie dazu die im Pizzabeispiel verwendete Syntax.

### Hilfe

Sie haben kein while- oder loop-Konstrukt zur Verfügung. Trotzdem kann man dieses Problem lösen. Schauen Sie sich das Pizza-Beispiel an. Dort sind ähnliche Probleme gelöst worden.

Nehmen Sie eine Tomate aus der Liste, schälen Sie diese und fügen Sie sie wieder in die Liste ein. Ein Beispiel:

```
--- f(erstesElement::rest) <= erstesElement::f(rest);
```

## 2.4 Auswertungsreihenfolge

Mit Pattern Matching bestimmt das System, welche Funktion zu einem Funktionsaufruf passt. Wir geben also an, unter welchen Bedingungen eine Funktion ausgeführt wird. Gewisse Probleme lassen sich aber mit Pattern Matching allein nur schlecht lösen. Betrachten Sie folgenden Programmausschnitt:

```
dec TeigAusrollen:PizzaTeig -> PizzaTeig;
--- TeigAusrollen(rad,dicke,geknetet) <=
        if (rad < 15) and (dicke > 0.5) then
           TeigAusrollen (rad + 2,dicke/1.5,geknetet);
        else
           (rad,dicke,ausgerollt);
```

### Kontrollfrage 2.4a

Könnte man in diesem Fall auch ohne das *if*-Konstrukt zum Ziel kommen? Wenn Sie glauben, dass dies möglich ist, dann schreiben Sie bitte die ersten zwei Programmzeilen auf. Falls Sie nicht dieser Ansicht sind, dann notieren Sie sich Ihre Überlegungen in ein paar Sätzen.

### Hilfe

Betrachten Sie die Folgen von Radius und Durchmesser, die durch die diversen Funktionsaufrufe gebildet werden. Ein Beispiel für den Radius: 5 7 9 11 ... Statt dem *if*-Konstrukt benützen Sie Pattern Matching.

Mit Pattern Matching geben Sie dem System einen Hinweis, unter welchen Bedingungen eine Funktion ausgewertet werden soll. Wenn diese Bedingungen mit Pattern Matching allein nicht genau formuliert werden können, kann man das *if*-Konstrukt zu Hilfe nehmen. Wir legen also implizit eine Reihenfolge der Funktionsauswertung fest.

Vergeichen wir dies mit dem imperativen Paradigma. Hier ist festgelegt, dass während dem Programmablauf eine Zeile nach der anderen abgearbeitet wird.

```
zutaten.tomaten[t].zustand:=geschält;
zutaten.tomaten[t].zustand:=gehackt;
```

Ein Vertauschen der Zeilen würde auch das Resultat des Programms beeinflussen. Im Gegensatz dazu kann man beim funktionalen Paradigma die Zeilen vertauschen.

```
dec TomatenRüsten:Tomate -> Tomate;
--- TomatenRüsten(frisch) <= TomatenRüsten(geschält);
--- TomatenRüsten(geschält) <= gehackt;
```

Damit können wir uns nochmals vor Augen halten, wie wir im funktionalen Paradigma ein Problem betrachten und formulieren.

Wir formulieren ein Problem als eine Sammlung von Funktionen, welche die Lösung spezifiziert. Ob nun das System mit der Auswertung der Funktionen von oben, von unten, oder sonstwie beginnt, ist uns völlig egal, solange es nur das Pattern Matching korrekt ausführt.

Unser Programm sieht also ein bisschen abstrakt formuliert so aus:

$$f(x) = \begin{cases} \text{condition}_1(x),\ \text{expr}_1(x) \\ \text{condition}_2(x),\ \text{expr}_2(x) \\ \vdots \\ \text{condition}_N(x),\ \text{expr}_N(x) \end{cases}$$

mit $\text{condition}_i(x)$ = true, falls das Pattern «gematcht» werden kann.

Im Gegensatz dazu gehen wir beim imperativen Paradigma von einer streng sequentiellen Auswertungsreihenfolge aus.

### Kontrollfrage 2.4b

Leider hat auch die eben beschriebene Theorie einen Haken. Was tut Ihrer Meinung nach das folgende Programmstück? Notieren Sie Ihre Überlegungen in ein paar Sätzen.

```
dec TomatenRüsten:Tomate -> Tomate;
--- TomatenRüsten(ungewaschen) <= gewaschen;
--- TomatenRüsten(ungewaschen) <= gehackt;
--- TomatenRüsten(ungewaschen) <= geschält;
```

### Hilfe

Wie werden solche Funktionen in der Mathematik bezeichnet?

## 2.5 Side-Effects

Sie haben sicher auch schon Funktionen in einer imperativen Programmiersprache verwendet. Zwischen «imperativen Funktionen» und «funktionalen Funktionen» besteht jedoch ein grundlegender Unterschied.

Betrachten wir dazu das folgende Beispiel.

```
type Tomate is record
   zustand:(frisch,geschält,gehackt):=frisch;
end Tomate;

type Mozzarella is record
   zustand:(frisch,gerieben,verkleinert) := frisch;
end Mozzarella;

type Ingredienzen is record
   tomaten: array N of Tomate;
   käse: array N of Mozzarella;
   gewürz: set of (pfeffer, paprika, basilikum);
end Ingredienzen;

gutelaune: boolean; -- globale Variable --

function TomatenVorbereiten(t: in Tomate) return Tomate is
begin
   if gutelaune then
      t.zustand:=geschält;
      gutelaune:=not(gutelaune);
   end if;
   t.zustand:=gehackt;
   return t;
end TomatenVorbereiten;
```

Nur falls Luigi gut gelaunt ist, gibt es geschälte Tomaten auf der Pizza. Ein bisschen formeller: Die Funktion *TomatenVorbereiten* liefert auf identischen Input unterschiedliche Resultate. Dieser Effekt ist im funktionalen Paradigma nicht möglich. Eine Funktion stellt wie in der Mathematik einen Wert dar, der nur vom Input abhängig ist. Dabei spielt es keine Rolle, zu welchem Zeitpunkt die Funktion ausgewertet wird.

Ist Ihnen schon aufgefallen, dass es im ganzen «funktionalen Pizzaprogramm» keine Variablen gibt, denen Sie etwas zuweisen können? Die einzigen Gebilde, die man als Variablen bezeichnen könnte, werden für die Benennung von Zwischenresultaten verwendet, wie *erstesElement* und *rest* im folgenden Beispiel.

```
dec Würzen:list(Gewürz) -> list(Gewürz);
--- Würzen(erstesElement::rest) <= [erstesElement];
```

Beim imperativen Paradigma bezeichnet jede Variable einen Speicherbereich. Während der Programmlaufzeit werden diese Bereiche laufend modifiziert. Bei Programmende ist das Resultat durch die Werte der Variablen, oder genauer, durch die Belegung des Speichers gegeben. Wenn eine Variable einen anderen Wert annimmt, nennt man dies einen *side-effect.*

Im Gegensatz sind funktionale Paradigmen non-side-effecting. Es gibt keine Variablen, die einen festen Speicherbreich bezeichnen, der dann laufend modifiziert wird.

## 3 Studenten-Lernkontrolle

Sie sind nun beim letzten Abschnitt des Selbststudiums angelangt. Hier können Sie prüfen, ob Sie alles verstanden haben. Falls Sie Probleme bei der Lösung dieser Aufgaben haben, gehen Sie bitte nochmals zum Abschnitt «Vermittlung des Lernstoffes» zurück. Lesen Sie die Punkte nochmals durch, die Sie noch nicht ganz verstanden haben. Wenn Sie den Stoff verstanden haben, aber trotzdem nicht weiterkommen, lesen Sie die Tips durch.

Für die Lernkontrolle haben Sie 30 Minuten Zeit. Sie machen diese Übung nur für sich selbst, das heisst sie wird nicht bewertet.

### Aufgabe 3.1

Schreiben Sie ein Programm, dass Sie Summe einer Liste von ganzen Zahlen berechnet. Benützen Sie dazu die bisher in dieser Unterrichtseinheit verwendete Syntax.

#### Hilfe

Obwohl wir sie bis jetzt noch nicht benötigt haben, existieren die üblichen Operatoren wie +, -, * und / auch bei funktionalen Sprachen.

In Ihrer Lösung muss Rekursion vorkommen.

Betrachten Sie nochmals die Lösung der Aufgabe 2.3.

### Aufgabe 3.2

Schreiben Sie ein Programm, das alle Elemente einer Liste von Zahlen entfernt, die einen bestimmten Wert haben.

#### Hilfe

In Ihrer Lösung muss Rekursion vorkommen.

Benützen Sie ein *if*-Konstrukt. Die Lösung ist im Aufbau ähnlich wie die Lösung der Aufgabe 3.1.

# Funktionales Paradigma: Lösungsblätter zum Selbststudium

## 1 Lösungen zum Pizzabeispiel

### Antwort 1.1

Zustände, die die Tomate annehmen: *frisch, geschält, gehackt.*

### Antwort 1.2

Einkaufsliste:

*- 1 Pizzateig*
*- 2 Tomaten*
*- 2 Mozzarella*
*- pfeffer, paprika, basilikum*

### Antwort 1.3

Zustände, die der Teig annimmt: *geknetet, ausgerollt.*

## 2 Lösungen der Kontrollfragen

### Antwort 2.2

«Wenn der Teig mit den Zutaten belegt und die Backzeit abgelaufen ist, dann ist die Pizza gebacken.

Falls der Teig mit den Zutaten belegt, die Backzeit aber nicht abgelaufen ist, dann backe weiter.

Falls die Pizza noch unbearbeitet ist, dann warte noch mit backen und belege zuerst die Pizza. Wenn Du damit fertig bist, dann kannst du backen.»

Jeder Funktionsaufruf zeigt ein gewisses «Pattern» (Muster). Wie zum Beispiel der erste Funktionsaufruf im Pizzaprogramm:

```
Backen
(
   (
      (5,4.5,geknetet),
      (
         (frisch,frisch),
         (frisch,frisch),
         (pfeffer,paprika,basilikum)
      ),
      unbearbeitet
   ),
   20,
   250
)
```

Mit obigem Funktionsaufruf wird

```
--- Backen((teig,zutaten,unbearbeitet),d,t) <=
        Backen(PizzaBelegen(teig,zutaten,unbearbeitet),d,t);
```

ausgeführt, weil der Funktionsaufruf mit dem Kopf der Funktion übereinstimmt. Funktion und Funktionsaufruf «matchen» somit.

### Antwort 2.3

Da keine loop- oder while-Konstrukte verfügbar sind, müssen wir statt Iteration Rekursion verwenden. Ihr Code sollte etwa folgendermassen aussehen.

```
data TomatenZustand == frisch++geschält++gehackt;
type Tomate == TomatenZustand;
dec TomatenSchälen:list(Tomate) => list(Tomate);
--- TomatenSchälen([]) <= [];
--- TomatenSchälen(frisch::rest) <= geschält::TomatenSchälen(rest);
```

Das funktionale Paradigma besitzt keine Konstrukte, die Iteration unterstützen. Stattdessen wird Rekursion eingesetzt.

### Antwort 2.4a

Man kann eine Lösung auch ohne ein if-Konstrukt finden. Diese ist aber nicht praktikabel. Die ersten 2 Zeilen lauten:

```
dec TeigAusrollen:PizzaTeig -> PizzaTeig;
--- TeigAusrollen(5,4.5,geknetet) <= TeigAusrollen(7,2.25,geknetet);
--- TeigAusrollen(7,2.25,geknetet) <= TeigAusrollen(9,1.125,geknetet);
```

usw.

### Antwort 2.4b

Nach der Theorie würde das Resultat zufällig gewählt. Dies, weil das Programmstück nicht deterministisch ist. Implementationen von funktionalen Sprachen gehen hier aber deterministisch vor und wählen von oben her das erste «Pattern» das «matcht». Die Tomate würde also in diesem Beispiel gewaschen.

# 3 Lösungen zur Studenten-Lernkontrolle

## Lösung 3.1

Die Funktion *listsum* berechnet die Summe einer Liste von ganzen Zahlen.

```
dec listsum: list(num) -> num;
--- listsum([]) <= 0;
--- listsum(erstesElement::rest) <= erstesElement + listsum(rest);
```

Wie ein System die Funktion auswerten würde:

listsum([3 2 1]) = 3 + listsum([2 1]) = 3 + 2 + listsum([1]) = 3 + 2 + 1 + listsum([]) =

3 + 2 + 1 + 0 = 3 + 2 + 1 = 3 + 3 = 6

## Lösung 3.2

Die Funktion *f* entfernt ein Element *x* aus einer Liste von ganzen Zahlen.

```
dec f: list(num)#num -> list(num);
--- f([],x) <= [];
--- f(erstesElement::rest,x) <=
        if erstesElement = x then
           f(rest,x)
        else
           erstesElement::f(rest,x);
```

Auswertung der Funktion durch ein System:

f([3 2 1],2) = 3::f([2 1],2) = 3::f([1],2) = 3::1::f([],2) = 3::1::[] = 3::[1] = [3 1]

# Logik-Paradigma: Arbeitsblätter fürs Selbststudium

## 0 So gehen Sie vor

### Pizzabeispiel

Es wurden Ihnen zwei Beispielprogramme abgegeben. Beide beschreiben, wie eine Pizza zubereitet wird. Das eine ist im imperativen, das andere im Logik-Paradigma formuliert. Lesen Sie zuerst das imperative Beispiel durch. Dadurch bekommen Sie eine Idee, wie die Pizza gemacht werden soll. Dies sollte Ihnen keine Schwierigkeiten bereiten. Gehen Sie danach zum Logik-Pizzabeispiel über. Studieren Sie auch dieses und versuchen Sie, die Fragen zu beantworten. Dazu müssen Sie das Beispiel nicht im Detail verstanden haben.

Beachten Sie die untenstehenden Bemerkungen zum Pizzabeispiel. Die Lehrperson steht Ihnen nur zur Verfügung, falls Sie Verständnisschwierigkeiten haben.

Für diesen Teil sollten Sie nicht mehr als 30 Minuten einsetzen.

### Vermittlung des Lernstoffes

In diesem Teil lernen Sie die wichtigsten Eigenschaften und Begriffe des Logik-Paradigmas kennen. Schauen Sie jeweils die Lösung nach, wenn Sie eine Kontrollfrage beantwortet haben. Gehen Sie erst weiter, wenn Sie den Stoff verstanden haben.

Nehmen Sie sich für diesen Teil 75 Minuten Zeit.

### Studenten-Lernkontrolle

Hier können Sie überprüfen, ob Sie den Lernstoff verstanden haben. Seien Sie ehrlich zu sich selber: Falls Sie die Lösung nicht selber finden, gehen Sie zurück zum Abschnitt «Vermittlung des Lernstoffes» und suchen Sie dort nach Ideen.

Sie haben für diesen Teil 30 Minuten zur Verfügung.

## 1 Bemerkungen zum Pizzabeispiel

Das Prozentzeichen % leitet einen Kommentar ein.

Variablen sind gross geschrieben: *Liste, Rest, Zustand, …*

[Kopf | Rest] stellt eine Liste mit unbekannter Anzahl von Elementen dar, wobei *Kopf* das erste Element in der Liste ist. Der *Rest* der Liste ist immer eine Liste, die unter Umständen jedoch leer sein kann [ ].

Falls die Umlaute durch entsprechende Schriftzeichen ersetzt werden, kann dieses Programm auf einem Prolog-System eingegeben und gestartet werden.

## 2 Vermittlung des Lernstoffes

Die Logik-Programmierung beruht darauf, dass wir das vorhandene Wissen über ein Problem in Fakten und Regeln darstellen. Dabei kümmern wir uns nur beschränkt (im Idealfall gar nicht) um den Kontrollfluss, das heisst wir geben nur den logischen Teil eines Algorithmus' vor und das System erledigt den Rest. Die Kontrollfluss-Konstrukte in einer imperativen Programmiersprache legen fest, welches die nächste Anweisung sein wird, die der Computer abarbeitet.

### 2.1 Fakten und Regeln

Ein Logik-Programm besteht aus Fakten («Tom ist ein Vater.») und aus Wenn-Dann-Regeln («Für alle X gilt, wenn X ein Vater ist, dann ist X männlich.»). Auf das Pizzabeispiel bezogen heisst das, sowohl der Ausgangs- und Endzustand als auch die notwendigen Arbeiten zur Erstellung einer Pizza sind bekannt und können somit als Fakten definiert werden.

Regeln setzen sich aus einem Kopf («X ist männlich») und einem Körper («X ist ein Vater») zusammen. Wenn der Regelkörper erfüllt ist, so auch der Regelkopf. Wiederum auf unser Bei-

spiel bezogen heisst das, die Pizza Margherita kann nur dann erfüllt werden, wenn der Startzustand und das Pizzabacken erfüllt werden können.

Aus den Fakten und Regeln lässt sich neues Wissen ableiten, wie zum Beispiel: «Tom ist männlich».

Fakten können auch als ein Spezialfall von Regeln aufgefasst werden, bei denen der Regelkörper immer erfüllt ist.

**Kontrollfrage 2.1**

Markieren Sie im Pizzabeispiel die Regeln und die Fakten mit zwei unterschiedlichen Farben. Geben Sie zusätzlich an, woran man äusserlich erkennen kann, dass es sich um eine Regel handelt.

## 2.2 Hornklauseln und Implikation

Sie kennen den Begriff der konjunktiven Normalform aus der Booleschen Algebra. Ein Beispiel: (abc)(abc)(abc). Die konjunktive Normalform besteht aus der Konjunktion () von Klauseln, wobei Klauseln Disjunktionen () von Atomen sind. Atome haben nur zwei Werte: Wahr (true) oder Falsch (false true).

Eine Hornklausel ist eine spezielle Form der Klausel, die höchstens *ein* nicht-negiertes Atom enthält (zweite Klausel im Beispiel).

Die Implikation () ist eine Funktion von zwei Variablen, welche wie folgt umgeschrieben werden kann:

a, b Atome: a b a b («not a or b»)

**Bemerkung**

Die meisten Logik-Programmiersprachen basieren auf Hornklauseln.

**Kontrollfrage 2.2**

Nehmen Sie die Regel *pizzaBacken(...) :- arbeiten(...), lösche(...), pizzaBacken(...)* aus dem Pizzabeispiel. Wandeln Sie diese Regel in eine Formel um, indem Sie die Kommas durch («and») und :- durch ersetzen. Führen Sie anschliessend diese Formel auf eine Hornklausel zurück.

**Hilfe**

In unserem Pizzabeispiel haben die Atome folgende Form: *name(*$_1$, .., $_n$*)*, wobei die $_i$ Variablen, Konstanten oder wiederum Atome sein können. Dazu ein Beispiel:

*pizzaBacken(startZustand(Zustand), [], Zutaten)*

## 2.3 Unifikation und Goal

Die Abarbeitung eines Logikprogramms beginnt mit der Angabe eines Goals. Dieses Goal kann ein Faktum oder ein Regelkopf sein. (Anbei: Goal wird mit *Ziel* übersetzt.) Bei unserem Pizzabeispiel geben Sie *pizzaMargherita(Arbeiten, Zutaten)* als Goal an.

Der erste Schritt des Systems besteht nun darin, das Programm durchzukämmen und ein Faktum oder eine Regel zu finden, die den selben Namen hat. Wird nichts gefunden, so antwortet das System mit der Meldung, dass es das Goal nicht beweisen konnte. Im positiven Fall beginnt der eigentliche Prozess der Unifikation, den ich durch das folgende Beispiel charakterisieren möchte:

Der Aufruf *datum(Tag1, Monat, 1994)* unifiziert mit dem Faktum *datum(Tag2, 6, Jahr)* aus folgenden Gründen:

Der Name *datum(...)* ist bei beiden Ausdrücken identisch.

Alle Variablen (gross geschrieben) erhalten entweder einen Wert oder werden mit einer anderen Variable gleichgesetzt, so dass beide Ausdrücke gleich sind und dass keine Widersprüche entstehen.

Das Resultat einer Unifikation ist somit eine Wertbelegung für die Variablen. Auf unser Beispiel bezogen heisst das: *Tag1 = Tag2, Monat = 6, Jahr = 1994*

Dazu ein weiteres Beispiel:

*s(p(1, 2), E)* unifiziert mit *s(p(1, Y), p(X, 4))*, so dass gilt: *E = p(X, 4), Y = 2*

**Kontrollfrage 2.3**

Unifiziert das Goal *lösche([keineZutat, gewürz(verstreut), tomaten(gehackt)], Zutaten)* mit einem oder mehreren Ausdrücken (Fakten oder Regelköpfen) aus dem Pizzabeispiel? Wenn ja, dann geben Sie bitte die Wertebelegung der Variablen an.

**Hilfe**

Eine Liste, gekennzeichnet durch die eckigen Klammern, kann mit einer Variable oder mit dem Konstrukt [Kopf | Rest] unifizieren.

## 2.4 Beweisprozess

Wir gehen davon aus, dass das vom Benutzer eingegebene Goal unifiziert werden konnte. Falls es mit einem Faktum unifizierte, gibt das System eine Meldung zurück, dass das Goal bewiesen worden ist. Für allfällige Variablen im Goal werden zudem die resultierenden Werte aus der Unifikation zurückgegeben.

Ein wenig komplizierter wird das Ganze, falls das Goal mit einem Regelkopf unifizierte. In diesem Fall wird aus jedem Atom des Regelkörpers ein Subgoal. Damit das Goal nun bewiesen werden kann, müssen alle Subgoals mit Fakten oder Regeln bewiesen werden. Dabei ist zu beachten, dass zum Beweis von Subgoals eventuell wiederum neue Subgoals bewiesen werden müssen.

**Bemerkung**

Bei den meisten Logik-Programmiersprachen hat der Programmierer die Freiheit, mehrere Regeln und Fakten mit dem gleichen Namen anzugeben. Dadurch stellt sich für das System die Frage, welche Regel zuerst zu wählen ist, falls mehrere unifzieren würden. Die meisten Systeme wählen in diesem Fall die erstmögliche aus.

**Kontrollfrage 2.4**

Gegeben sei folgendes Prolog-Programm:

```
p(X) :- q(X), r(Y).
q(Y) :- s(X), t(Y), u(z).
r(1).
s(2).
t(3).
u(4).
```

Der Benutzer gibt *p(X)* als zu beweisendes Goal an. Kann dieses Goal bewiesen werden? Falls ja, welchen Wert hat dann die Variable *X*. Zeichnen Sie mit Hilfe eines Suchbaumes den Weg des Beweisprozesses auf.

**Hilfe**

Damit *p(X)* bewiesen werden kann, muss zuerst das Subgoal *q(X)* und nachher das Subgoal *r(Y)* bewiesen werden. *q(X)* kann bewiesen werden, falls das Atom *q(...)* im Programm als Faktum oder Regelkopf auftritt und die beiden unifizieren.

## 2.5 Backtracking

Es stellt sich nun die Frage, was passiert, falls ein Subgoal nicht bewiesen werden kann. Folgt daraus direkt, dass das Goal nicht beweisbar ist? Nein, denn sehr oft existieren mehrere verschiedene Regelkörper zum gleichen Regelkopf. In diesem Fall muss sich das System ja

bekanntlich für einen Regelkörper entscheiden. Wenn nun ein Subgoal fehlschlägt (nicht beweisbar ist), dann geht das System automatisch zur Regel zurück, wo es sich zum letzten Mal entscheiden musste und wählt dieses Mal eine Alternative. Falls alle diese Alternativen wiederum fehlschlagen, geht das System noch einen Schritt weiter zurück und beginnt von neuem. Diesen Prozess nennen wir Backtracking.

Backtracking hat also zur Folge, dass *alle* denkbaren Beweiswege durchprobiert werden, bis einer zum Erfolg führt. Nur wenn kein einziger Weg erfolgreich ist, gibt das System die Meldung zurück, dass dieses Goal nicht beweisbar sei.

Backtracking ist nicht die einzige Methode diese Problematik zu lösen; aber die am weitesten verbreitete.

### Kontrollfrage 2.5

Im Pizzabeispiel spielt das Atom *zustand(...)*, welche den Tomaten-, den Mozzarella-, den Teig- und auch den Pizzazustand charakterisiert, eine wichtige Rolle. Damit das Subgoal *pizzaBacken(...)* erfüllt werden kann, müssen im Wesentlichen mehrere *arbeiten(...)* durchgeführt werden, so dass der Zustand von *startZustand(...)* nach *endZustand(...)* übergeht.

Gehen Sie nun davon aus, dass folgende Zustandsübergänge bereits stattgefunden haben:

| | | | | |
|---|---|---|---|---|
| Startzustand: | *zustand(frisch,* | *frisch,* | *geknetet,* | *unbearbeitet)* |
| 1: | *zustand(frisch,* | *frisch,* | *ausgerollt,* | *unbearbeitet)* |
| 2: | *zustand(geschält,* | *frisch,* | *ausgerollt,* | *unbearbeitet)* |
| 3: | *zustand(gehackt,* | *frisch,* | *ausgerollt,* | *unbearbeitet)* |
| 4: | *zustand(gehackt,* | *gerieben,* | *ausgerollt,* | *unbearbeitet)* |

Das System befindet sich also in Zustand 4. Nun stellt sich die Frage, ob das System weitere Zustandsübergänge findet, so dass der Endzustand erreicht werden kann? Falls nicht, so kommt die Strategie des Backtrackings zum Zuge.

An welcher Stelle in der Liste der Zustandsübergänge muss eine Alternative gewählt werden? Schreiben Sie auch die darauffolgenden Zustandsübergänge auf.

## 3 Studenten-Lernkontrolle

Sie haben im Selbststudium die wichtigsten Begriffe und Konzepte des Logik-Paradigmas kennengelernt und diese bereits in kleinen Übungen angewendet. In der folgenden Studenten-Lernkontrolle geben wir Ihnen die Möglichkeit, dieses Wissen zu sichern und zu festigen, denn Sie müssen ja auch in der Lage sein, das Wissen Ihren Mitstudenten weiterzugeben.

Für die Lernkontrolle haben Sie 30 Minuten Zeit. Sie machen diese Übung nur für sich selbst, das heisst sie wird nicht bewertet. Falls Sie Probleme bei der Lösung dieser Aufgaben haben, gehen Sie bitte nochmals zu den entsprechenden Abschnitten in «Vermittlung des Lernstoffes» zurück.

### Aufgabe 3.1

Gegeben sind Fakten der Form:

```
vater(X, Y)      % X ist Vater von Y
mutter(X, Y)  % X ist Mutter von Y
```

sowie eine kleine Datenbasis:

```
vater(tom, andi)          mutter(sarah, nadia)
vater(tom, nadia)         mutter(sarah, hans)
vater(tom, hans)          mutter(moni, andi)
vater(peter, dani)     mutter(nadia, dani)
vater(peter, chris)       mutter(nadia, chris)
```

Formulieren Sie folgende Regeln (gleiche Syntax wie im Pizzabeispiel verwenden):

a) *istMutter(X)* X ist eine Mutter
b) *istVater(X)* X ist ein Vater
c) *grossvater(X, Y)* X ist Grossvater von Y
d) *vorfahre(X, Y)* X ist ein Vorfahre von Y

Ihre Lösung ist genügend, wenn Sie die Regeln a) - c) korrekt aufstellen können. Falls Sie auch die Regel d) richtig haben, so sind Sie bereits in der Lage auch raffiniertere Programme zu schreiben.

#### Hilfe

Überlegen Sie sich zuerst alle Bedingungen, die gelten müssen, damit zum Beispiel *X* der Grossvater von *Y* ist. Schauen Sie sich nochmals Abschnitt 2.1 und das Pizzabeispiel an, denn dort sehen Sie, wie Regeln korrekt aufgestellt werden. Für die Regel *vorfahre(...)* benötigen Sie den rekursiven Aufruf einer Regel.

### Aufgabe 3.2

Beantworten Sie mit Hilfe obenstehender Datenbasis folgende Fragen:

a) ?- istVater(hans).
b) ?- istMutter(X).
c) ?- grossvater(X, Y).
d) Wer sind die Vorfahren von *dani*?

Versuchen Sie beim Auffinden der Lösungen gleich vorzugehen wie ein Logik-System.

Ihre Lösung ist genügend, wenn Sie alle richtigen Lösungen der Teilaufgaben a) - c) gefunden haben.

# Logik-Paradigma: Lösungsblätter zum Selbststudium

## 1 Lösungen zum Pizzabeispiel

### Antwort 1.1

Der Pizzaiolo muss für seine Margherita folgenden Zutaten einkaufen: *Tomaten, Mozzarella, Teig, Gewürz*

### Antwort 1.2

Die Tomaten machen folgende Zustände durch: *frisch, geschält, gehackt*.

### Antwort 1.3

Die Pizza wird im Zustand *belegt* gewürzt.

**Hilfe:**

Richten Sie Ihr Augenmerk auf die Zeilen mit dem Wort *arbeiten(...)*.

Schauen Sie sich noch einmal genau die erste Kommentarzeile im Programm an. Dort wird die Struktur eines Zustandes genau definiert.

Die Struktur der Anweisung *arbeiten* besteht aus vier Teilen: Eingabe- und Ausgabezustand, ausgeführte Aktion, verwendete Zutat.

## 2 Lösungen der Kontrollfragen

### Antwort 2.1

Die einzigen Regeln sind: *pizzaBacken(...)* und *pizzaMargherita(...)*. In Prolog erkennt man die Regeln daran, dass sie aus einem Kopf und einem Körper und dem dazwischenliegenden Zeichen :- bestehen.

Wie Sie leicht feststellen können, existieren zwei verschiedene Regeln *pizzaBacken(...)*. Diese beiden Regeln gehören jedoch zusammen und werden deshalb direkt untereinander geschrieben. Falls das System die Regel *pizzaBacken(...)* anwenden will, muss es sich für eine der beiden Ausführungen entscheiden.

Auch in imperativen Programmiersprachen gibt es Wenn-Dann-Regeln. Diese unterscheiden sich jedoch von den hier besprochenen dadurch, dass eine *Aktion* ausgeführt wird, falls eine Bedingung erfüllt ist.

### Antwort 2.2

Regel: *arbeiten(...), lösche(...), pizzaBacken(...) -: pizzaBacken(...)*
Formeln: *arbeiten(...) lösche(...) pizzaBacken(...) pizzaBacken(...)*
*(arbeiten(...) lösche(...) pizzaBacken(...)) pizzaBacken(...)*
Hornklausel: *arbeiten(...) lösche(...) pizzaBacken(...) pizzaBacken(...)*

Wir gehen davon aus, dass in einem Logikprogramm nur Dinge stehen, die wahr sind, das heisst also, alles was nicht explizit aufgeschrieben worden ist oder hergeleitet werden kann ist falsch.

Die Hornklauseln stellen somit eine sinnvolle Einschränkung der konjunktiven Normalform dar, weil sie es gestatten, Regeln und Fakten im Programm ohne Negation darzustellen.

Beispiel: Faktum mit Hilfe einer Hornklausel darstellen:

Faktum: *istVater(tom).*
Regel: *istVater(tom) :- true.*
Formel: *true istVater(tom)*
Hornklausel: *true istVater(tom)* ist identisch zu: *false istVater(tom)*

Da in einem Logikprogramm nur wahre Aussagen stehen, muss *istVater(tom)* wahr sein. Somit reduziert sich die Hornklausel auf: *istVater(tom)*.

Ein weiterer positiver Aspekt der Hornklauseln ist, dass sie es erlauben, in nützlicher Zeit Werte für die Variablen zu finden, welche eine Formel wahr machen. Das heisst, der Aufwand für den Computer, eine Lösung zu finden, ist vertretbar.

### Antwort 2.3

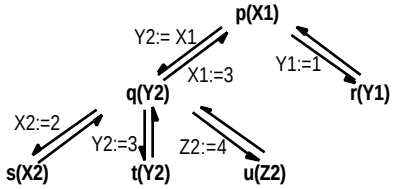
*lösche([keineZutat, gewürz(verstreut), tomaten(gehackt)], Zutaten)* unifiziert mit zwei Fakten im Pizzabeispiel:

1. Unifikation mit *lösche([keineZutat|Rest], Rest)*, so dass gilt:
   *Rest = Zutaten = [gewürz(verstreut), tomaten(gehackt)]*.
2. Unifikation mit *lösche(Liste, Liste)*, so dass gilt:
   *Liste = Zutaten = [keineZutat, gewürz(verstreut), tomaten(gehackt)]*.

Falls Sie mit dieser Aufgabe Mühe hatten, sollten Sie den Abschnitt 2.3 und die Bemerkungen zum Pizzabeispiel nochmals sorgfältig durchlesen.

### Antwort 2.4

Das Goal *p(X)* kann bewiesen werden und die Variable *X* erhält dabei den Wert 3. Der Suchbaum des Beweisprozesses sieht folgendermassen aus:




Es ist nicht notwendig, dass der Benutzer *p(X)* als Goal angibt. Zum Beispiel kann er auch *p(3)* als Goal eingeben. Da *p(3)* beweisbar ist, erhält er eine positive Meldung vom System zurück. Falls er aber *p(1)* eingibt, meldet das System, dass dies nicht beweisbar ist.

### Antwort 2.5

Vom momentanen *zustand(gehackt, gerieben, ausgerollt, unbearbeitet)* gibt es keinen legalen Übergang zu einem Folgezustand. Dies hat zur Folge, dass Backtracking stattfindet: Das System geht einen Schritt zurück und versucht eine Alternative zum obenstehenden Zustand zu finden. Es findet den neuen Zustand 4, welcher schliesslich auch zum Endzustand führt:

| | | | | |
|---|---|---|---|---|
| 4: | *zustand(gehackt,* | *verkleinert,* | *ausgerollt,* | *unbearbeitet)* |
| 5: | *zustand(gehackt,* | *verkleinert,* | *ausgerollt,* | *belegt)* |
| 6: | *zustand(gehackt,* | *verkleinert,* | *ausgerollt,* | *gewürzt)* |
| Endzustand: | *zustand(gehackt,* | *verkleinert,* | *ausgerollt,* | *gebacken)* |

# 3 Lösungen zur Studenten-Lernkontrolle

## Lösung 3.1

Die ersten zwei Regeln sind analog zueinander. Wenn *X* eine Mutter sein soll, so muss *X* als erster Parameter im Faktum *mutter(...)* aufgeführt sein, denn wir haben *mutter(X, Y)* so definiert, dass *X* die Mutter von *Y* sei. Der Parameter *Y* steht für ein Kind und ist deshalb hier nicht weiter von Bedeutung.

```
istMutter(X) :- mutter(X, Kind)
istVater(X) :- vater(X, Kind)
```

Die Regel *grossvater(...)* ist schon etwas komplizierter. Denn hier müssen zwei verschiedene Fälle vorgesehen werden: erstens kann *X* der Vater der Mutter von *Y* sein und zweitens kann *X* der Vater des Vaters von *Y* sein.

```
grossvater(X, Y) :- vater(X, Z1), mutter(Z1, Y).
grossvater(X, Y) :- vater(X, Z2), vater(Z2, Y).
```

Die Variablen *Z1* und *Z2* schaffen die notwendige Beziehungen zwischen den «and»-verknüpften Bedingungen der Regeln.

Bei der Regel *vorfahre(...)* wird die Möglichkeit der Rekursion ausgenutzt. Da wir die Rekursion bis jetzt nicht explizit erwähnt haben, möchten wir dies hier nachholen. Rekursion ist die wichtigste Art, um in Logik-Programmen Schleifen zu formulieren.

```
vorfahre(X, Y) :- vater(X, Y).
vorfahre(X, Y) :- mutter(X, Y).
vorfahre(X, Y) :- vater(X, Z), vorfahre(Z, Y).
vorfahre(X, Y) :- mutter(X, Z), vorfahre(Z, Y).
```

*X* ist dann ein Vorfahre von *Y*, wenn *X* der Vater oder die Mutter von *Y* ist oder wenn *X* der Vater oder die Mutter eines Vorfahren von *Y* ist. Diese Definition des Vorfahren nennen wir rekursiv, weil Vorfahre in der eigenen Definition vorkommt. Damit die Rekursion endet braucht es eine Verankerung in der Form, wie Sie sie in den ersten beiden Zeilen dieses Beispiels sehen.

Auch im Pizzabeispiel ist Rekursion verwendet worden, um das wiederholte Testen von möglichen Arbeiten zu realisieren.

## Lösung 3.2

a) ?- istVater(hans). Die Frage ob *hans* ein Vater ist, muss mit Nein beantwortet werden, da kein Faktum *vater(hans, ...)* existiert. Es gibt zwar ein Faktum *vater(tom, hans)*, doch dies ist per Definition so zu interpretieren, dass *tom* der Vater von *hans* ist.

b) ?- istMutter(X). Hier wird nach allen *X* gefragt, die Mutter sind. Das System gibt folgende Personen an: *sarah, moni, nadia*

c) ?- grossvater(X, Y). Es werden Wertepaare gesucht, von denen *X* der Grossvater von *Y* ist. Das System findet folgende zwei Paare: *(tom, chris)* und *(tom, dani)*

d) Wer sind die Vorfahren von *dani*? In der hier verwendeten Schreibweise würde die Anfrage wie folgt aussehen: *vorfahre(X, dani)*. Es werden also alle *X* gesucht, die Vorfahre von *dani* sind. Das System findet Danis Eltern *peter* und *nadia*, sowie Nadias Eltern *tom* und *sarah*.

# Datenfluss-Paradigma: Arbeitsblätter fürs Selbststudium

## 0 So gehen Sie vor

### Pizzabeispiel

Es wurden Ihnen zwei Beispielprogramme abgegeben. Beide beschreiben, wie eine Pizza zubereitet wird. Das eine ist im imperativen, das andere im Datenfluss-Paradigma formuliert. Lesen Sie zuerst das imperative Beispiel durch. Dadurch bekommen Sie eine Idee, wie die Pizza gemacht werden soll. Dies sollte Ihnen keine Schwierigkeiten bereiten. Gehen Sie danach zum Datenfluss-Pizzabeispiel über. Studieren Sie auch dieses und versuchen Sie, die Fragen zu beantworten. Dazu müssen Sie das Beispiel nicht im Detail verstanden haben.

Beachten Sie die untenstehenden Bemerkungen zum Pizzabeispiel. Die Lehrperson steht Ihnen nur zur Verfügung, falls Sie Verständnisschwierigkeiten haben.

Für diesen Teil sollten Sie nicht mehr als 30 Minuten einsetzen.

### Vermittlung des Lernstoffes

In diesem Teil lernen Sie die wichtigsten Eigenschaften und Begriffe des Datenfluss-Paradigmas kennen. Schauen Sie jeweils die Lösung nach, wenn Sie eine Kontrollfrage beantwortet haben. Gehen Sie erst weiter, wenn Sie den Stoff verstanden haben.

Nehmen Sie sich für diesen Teil 75 Minuten Zeit.

### Studenten-Lernkontrolle

Hier können Sie überprüfen, ob Sie den Lehrstoff verstanden haben. Seien Sie ehrlich zu sich selber: Falls Sie die Lösung nicht selber finden, gehen Sie zurück zum Abschnitt «Vermittlung des Lernstoffes» und suchen Sie dort nach Ideen.

Sie haben für diesen Teil 30 Minuten Zeit.

## 1 Bemerkungen zum Pizzabeispiel

Funktionsnamen sind **fett** geschrieben. Beispiel: **TeigAusrollen**

Daten-Bezeichner sind *kursiv* geschrieben. Beispiel: *teig*

Datenwerte sind normal geschrieben. Beispiel: ausgerollt

Nicht weiter spezifizierte Funktionsnamen sind klein geschrieben:

| | |
|---|---|
| **if**: | entscheidet in Abhängigkeit des Datenwertes, welcher der Ausgänge das Datum enthält |
| **split**: | teilt die Daten auf mehrere separate Datenpfade auf |
| **join**: | verbindet mehrere Datenpfade zu einem |
| **append**: | wie **join**, jedoch muss eine bestimmte Reihenfolge eingehalten werden |
| **schälen:** | schält eine Tomate |
| **hacken:** | hackt eine Tomate |
| **verkleinern**: | verkleinert einen Mozzarella |

*head* ist das erste Element einer Liste und *tail* der Rest. *nil* bezeichnet die leere Liste.

# 2 Vermittlung des Lernstoffes

Das Datenfluss-Paradigma ist eigentlich nichts Neues, denn es dient schon lange als Grundlage von verschiedenen Software-Entwurfsmethoden. Mit dem grossen Aufkommen von grafischen Arbeitsstationen in den letzten Jahren rückt dieses Konzept jedoch wieder in den Vordergrund, denn die grafischen Darstellungen werden jetzt nicht mehr nur zu Dokumentationszwecken verwendet, sondern sie repräsentieren direkt das abzuarbeitende Programm.

Es existieren aber auch textuelle Programmiersprachen, welche das Datenfluss-Paradigma erfüllen. Diese haben eine sehr grosse Ähnlichkeit mit funktionalen Programmiersprachen.

## 2.1 Datenfluss-Diagramm

Das Datenfluss-Paradigma lässt sich am besten durch einen gerichteten Graphen, dem Datenfluss-Diagramm, visualisieren. Dieses Diagramm setzt sich aus Knoten (im Pizzabeispiel als Ellipsen gezeichnet) und Kanten (Verbindungen zwischen den Knoten) zusammen. In diesem Netzwerk von Knoten und Kanten fliessen Datenströme. Dabei konsumiert jeder Knoten die einfliessenden Daten und berechnet neue Daten, die zum nächsten Knoten fliessen.

Die Aufgabe des Programmierers ist es also, das Datenfluss-Diagramm aufzustellen und die Gleichungen in den Knoten festzulegen.

### Kontrollfrage 2.1

Versuchen Sie ein Datenfluss-Diagramm zu zeichnen, das folgende Berechnung durchführt:

$x(t+2) := (a(t) + b(t))*(c(t) + d(t))$

Der Parameter t soll hier lediglich die Zeit zu den diskreten Zeitpunkten (t, t+1, t+2, ...) darstellen.

### Hinweis

Datenstrom und Datenpaket sind zwei wichtige Begriffe, die fortan gebraucht werden. Sie haben folgende Bedeutung:

> Datenpaket: Ein Datenpaket ist ein Verbund von einzelnen Datenwerten. In einer imperativen Programmiersprache wäre ein Datenpaket ein Record.
>
> Datenstrom: Ein Datenstrom ist eine zeitlich geordnete Menge von Datenpaketen. Ein Knoten konsumiert ein Datenpaket des Eingabe-Datenstromes und produziert ein neues Datenpaket des Ausgabe-Datenstromes. Auf einer Kante zwischen zwei Knoten kann sich nur ein Datenpaket pro Mal befinden.

## 2.2 Knotengleichungen und Ablaufsteuerung

Die Gleichungen in den Knoten des Datenfluss-Diagramms bestimmen durch ihre gegenseitige Abhängigkeit die Abarbeitungsreihenfolge des Programms. Damit das Ganze jedoch berechenbar bleibt, muss folgende Einschränkung gelten: Der Ausgabewert eines Knotens zum Zeitpunkt t darf lediglich von Eingabewerten abhängen, die vor t berechnet wurden.

Diese Einschränkung verunmöglicht aber keineswegs die Rückführung eines Ausgangs zum Eingang desselben Knotens. Das heisst durch Rückführung ist es möglich, Schleifen (Iterationen) zu programmieren. Trotzdem wird die Iteration selten benutzt, da sie durch die elegantere Rekursion ersetzt werden kann.

**Kontrollfrage 2.2**

Überlegen Sie sich, woran man Rekursion in einem Datenfluss-Programm erkennt. Finden Sie im Pizzabeispiel diejenigen Stellen, wo Rekursion eingesetzt wird.

**Hilfe**

Rekursion in prozeduralen Sprachen (imperatives Paradigma) erkennt man daran, dass in der Prozedur *p* die Prozedur *p* aufgerufen wird.

## 2.3 Ausführungsmodelle

Im Datenfluss-Paradigma sind grundsätzlich zwei verschiedene Ausführungsmodelle denkbar:

*data-driven*: Ein Knoten berechnet die Ausgabedaten, sobald *alle* benötigten Eingabedaten vorhanden sind.

*demand-driven*: Ein Knoten berechnet die Ausgabedaten erst, wenn alle benötigten Eingabedaten vorhanden sind und die Ausgabedaten von einem anderen Knoten *verlangt* werden.

**Kontrollfrage 2.3**

Beide Ausführungsmodelle haben ihre Vor- und Nachteile. Schreiben Sie diese auf. Berücksichtigen Sie dabei vor allem die Aspekte der Parallelität und der Überproduktion.

## 2.4 Vergleich mit anderen Paradigmen

Das Datenfluss-Paradigma hat grosse Ähnlichkeiten mit dem funktionalen Paradigma. (Näheres über das funktionale Paradigma erfahren Sie in der Unterrichtsrunde.) Viele textuelle Datenfluss-Programmiersprachen basieren auf funktionalen Sprachen. Trotzdem gibt es aus der Sicht der Paradigmen gewisse Unterschiede:

Ein funktionales Programm kann durch eine einzige Funktion dargestellt werden; ein Datenfluss-Programm ist eine Menge von Gleichungen.

Datenfluss-Knoten operieren auf Datenströmen; Funktionen im funktionalen Paradigma operieren auf Datenpaketen.

Im Gegensatz zum imperativen Paradigma wird die Abarbeitungsreihenfolge nicht explizit festgelegt, was zur Folge hat, dass beim Datenfluss-Paradigma auf Kontrollfluss-Konstrukte verzichtet werden kann.

**Kontrollfrage 2.4**

Eines der häufigsten Kontrollfluss-Konstrukte im imperativen Paradigma ist das *if-then*. Im Datenfluss gibt es auch ein *if*. Weshalb wird hier das *if* als Datenfluss- und nicht als Kontrollfluss-Konstrukt angesehen? Versuchen Sie die Unterschiede zum *if-then* aufzuzeigen.

**Hilfe**

Im Abschnitt 2.2 ist erwähnt worden, dass in jedem Knoten eine Gleichung stehen muss, welche die Ausgangsdaten als Funktion der Eingangsdaten darstellt. Überlegen Sie sich, ob Sie eine solche Gleichung auch für das *if* aufstellen können.

Das Datenfluss-*if* kann als ein Spezialfall des *if-then* betrachtet werden. Welche Einschränkungen müssen für den *if*-Teil gelten und was lässt sich über den *then*-Teil sagen?

## 3 Studenten-Lernkontrolle

Sie haben im Selbststudium die wichtigsten Begriffe und Konzepte des Datenfluss-Paradigmas kennengelernt und diese bereits in kleinen Übungen angewendet. In der folgenden Studenten-Lernkontrolle geben wir Ihnen die Möglichkeit, dieses Wissen zu sichern und zu festigen, denn Sie müssen ja auch in der Lage sein, das Wissen Ihren Mitstudenten weiterzugeben.

Für die Lernkontrolle haben Sie 30 Minuten Zeit. Sie machen diese Übung nur für sich selbst, das heisst sie wird nicht bewertet. Falls Sie Probleme bei der Lösung dieser Aufgaben haben, gehen Sie bitte nochmals zu den entsprechenden Abschnitten in «Vermittlung des Lernstoffes» zurück.

### Aufgabe 3.1

Gegeben ist der folgende vereinfachte QuickSort-Algorithmus, welcher eine Liste von Zahlen sortiert. Wenden Sie diesen auf die Liste [3, 5, 1, 3, 2, 4, 2] an. Schreiben Sie dazu alle Zwischenschritte genau auf.

Vereinfachter QuickSort:

1. Nehmen Sie das erste Element aus der Liste heraus. Wir nennen dieses Element das Pivot.
2. Teilen Sie die Liste in zwei kleinere auf, so dass die eine Liste nur Zahlen enthält, die kleiner als das Pivot sind und die andere Liste nur Zahlen enthält, die gleich oder grösser als das Pivot sind. Es kann sein, dass eine Liste leer ist.
3. Wenden Sie das gleiche Prinzip auf beide Teillisten an, dann auf die Teile der neuen Zerlegung, usw., bis jeder Teil nur noch maximal ein Element umfasst.
4. Fügen Sie alle Teile in der richtigen Reihenfolge zusammen:
   (sortierte Liste mit Zahlen < Pivot) Pivot (sortierte Liste mit Zahlen  Pivot)

**Hilfe**

QuickSort ist ein rekursiver Algorithmus.

Liste mit nur einem Element wird nicht mehr sortiert.

### Aufgabe 3.2

Ihre Aufgabe ist es nun ein Datenfluss-Diagramm aufzuzeichnen, welches Zahlenlisten nach dem Prinzip von QuickSort sortiert.

Benutzen Sie die vordefinierten Funktionen *split, append* und *if* unter der Annahme, dass *split* die Liste in ihre drei Teile (Schritte 1 und 2) aufteilt. Gehen Sie weiter davon aus, dass *append* die Daten in der richtigen Reihenfolge zusammenfügt (Schritt 4).

Da das Verfahren rekursiv ist. müssen Sie die notwendigen Vorkehrungen treffen, damit die Rekursion abbricht. Verwenden Sie dazu die Funktion *if*.

Ihre Lösung ist gut, wenn das Prinzip von QuickSort klar zum Ausdruck kommt.

**Hilfe**

Schauen Sie sich den Knoten *M-Bearbeiten* im Pizzabeispiel an. Der Knoten *QuickSort* wird sehr ähnlich aussehen. In den Abschnitten 2.1 und 2.2 finden Sie die notwendige Theorie.

# Datenfluss-Paradigma: Lösungsblätter zum Selbststudium

## 1 Lösungen zum Pizzabeispiel

### Antwort 1.1

Der Pizzaiolo muss für seine Margherita folgenden Zutaten einkaufen: *Tomaten, Mozzarella, Teig, Gewürz*

### Antwort 1.2

a) Folgende Aktionen werden auf die Tomaten angewendet, falls sie frisch sind: *schälen*, *hacken*.

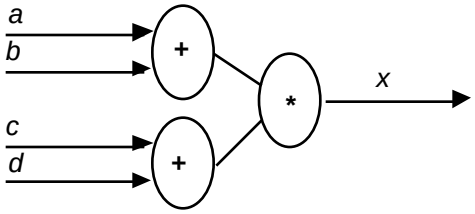
b) Die Tomaten sind ein Teil der Ingredienzen und somit auch Bestandteil der Pizza. Ihr Farbstiftstrich muss beim Input *pizza* in **Backen** beginnen, bei **PizzaBelegen** die Ellipse verlassen, dann zu **PizzaBelegen** gehen, diese Ellipse bei **Vorbereitung** verlassen, dann zu **Vorbereitung** gehen, diese Ellipse bei **T-Bearbeiten** verlassen, dann **T-Bearbeiten** pro Tomate einmal durchqueren und danach den ganzen Weg wieder zurück, bis **Backen** vollständig durchgearbeitet ist.

## 2 Lösungen der Kontrollfragen

### Antwort 2.1

Es ist ersichtlich, dass *x* wirklich zum Zeitpunkt t+2 berechnet werden kann, vorausgesetzt, dass eine Operation nicht länger als eine Einheit dauert.

Das Datenfluss-Diagramm sieht nun folgendermassen aus:




**Bemerkungen**

Die Variablen *a, b, c, d* und *x* repräsentieren Datenströme, die sich im Laufe der Zeit verändern können.

Das Datenfluss-Diagramm stellt einen dynamischen Prozess dar: Jeder Knoten berechnet einen neuen Wert, sobald neue Inputwerte vorhanden sind.

Aus einem Datenfluss-Diagramm ist schnell ersichtlich, welche Operationen gleichzeitig (parallel) durchgeführt werden können.

### Antwort 2.2

Die Rekursion ist ein sehr wichtiges Ausdrucksmittel in der Beschreibung von Datenflüssen. Im Datenfluss-Diagramm ist sie daran erkennbar, dass im Knoten *k* mindestens ein Datenpfad zum Knoten *k* führt. Oder anders gesagt, der Knoten *k* enthält einen Subknoten, der wiederum mit *k* bezeichnet ist.

In unserem Pizzabeispiel wird in den Knoten *Backen*, *Teigausrollen*, *T-Bearbeiten* und *M-Bearbeiten* Rekursion eingesetzt.

Falls Sie in Aufgabe 1.2c etwas Mühe mit dem Aufzeichnen des «Tomaten-Lebensweges» hatten, so könnte dies mit der Rekursion zusammenhängen.

### Antwort 2.3

Das *data-driven*-Modell ermöglicht die parallele Abarbeitung von allen Knotenprozessen, deren Eingabedaten vorhanden sind. Wenn die Eingabedaten Datenströme sind, kann Pipeline-Verarbeitung angewendet werden. Der Nachteil dabei ist, dass eine Überproduktion von Ausgabedaten stattfinden kann.

Beim *demand-driven*-Modell ist es gerade umgekehrt. Überproduktion kann nicht stattfinden, weil die Ausgabedaten nur erstellt werden, falls sie benötigt werden. Dadurch wird aber die Pipeline-Verarbeitung verunmöglicht.

### Antwort 2.4

Beim Datenfluss-Diagramm stehen in den Knoten Gleichungen, welche die Ausgangsdaten als Funktion der Eingangsdaten beschreiben. Auch für das *if* lässt sich eine solche Gleichung aufstellen:

Ausgabedaten = f(Eingabedaten) mit $f(x) = \begin{cases} x & \text{für } x = \text{Bedingung}(x) \\ \text{keine Daten}, & \text{sonst} \end{cases}$

Wie man sieht, muss die Bedingung alleine von den Eingabedaten abhängen. Im imperativen *if-then* ist dies jedoch nicht der Fall, denn dort kann sowohl eine beliebige Bedingung als auch eine beliebige Aktion gewählt werden. Anders gesagt, das *if* steuert den Datenfluss und das imperative *if-then* kontolliert die Reihenfolge der Programmabarbeitung und damit natürlich auch den Datenfluss. Somit kann gesagt werden, dass das Datenfluss-*if* ein Spezialfall des imperativen *if-then* ist.

Das Datenfluss-*if* kann auch als programmierte Weiche aufgefasst werden: Falls der herannahende Zug ein IC ist, schaltet sie auf geradeaus, sonst auf abzweigen. So wird die Weiche automatisch für jeden Zug neu gestellt. Dabei ist es aber unerlässlich, dass die Weiche für jeden Wagen des Zuges in der gleichen Stellung verharrt. In dieser Analogie stellt ein Zug ein Datenpaket, ein Wagen eine Komponente des Datenpaketes und der gesamte Schienenverkehr dieser Strecke den Datenstrom dar.

# 3 Lösungen zur Studenten-Lernkontrolle

## Lösung 3.1

Das Sortierverfahren QuickSort angewendet auf die Liste [3, 5, 1, 3, 2, 4, 2] spielt sich wie folgt ab:

1. pivot := 3
2. k-list := [1, 2, 2]; g-list := [5, 3, 4]
3. k-list zerlegen :
   - 3.1 k-piv := 1
   - 3.2 kk-list := []; kg-list := [2, 2]
   - 3.3 kg-list zerlegen
     - 3.3.1 kg-piv := 2
     - 3.3.2 kgk-list := []; kgg-list := [2]

   g-list zerlegen:
   - g-piv := 5
   - gk-list := [3, 4]; gg-list := []
   - gk-list zerlegen
     - gk-piv := 3
     - gkk-list := []; gkg-list := [4]
4. alle Teile richtig zusammenfügen, so dass wieder eine Liste entsteht:
   ((kk-list k-piv (kgk-list kg-piv kgg-list)) pivot ((gkk-list gk-piv gkg-list) g-piv gg-list))
   [1, 2, 2, 3, 3, 4, 5]

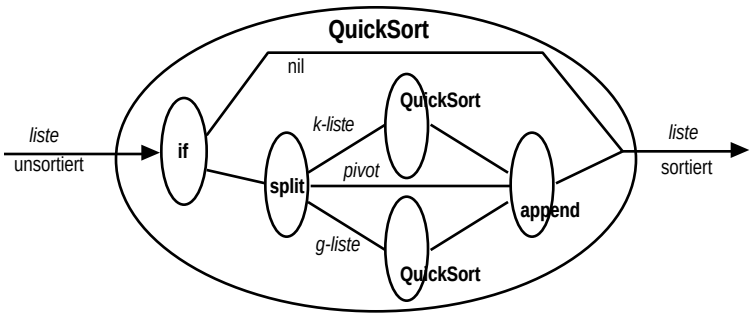
In der Aufgabenstellung ist bemerkt worden, dass QuickSort ein rekursiver Sortieralgorithmus ist. Das bedeutet, dass das gleiche Sortierprinzip auf die ursprüngliche Liste wie auch auf die Teillisten angewendet wird. Diese Teillisten (k-list, g-list, kk-list, kgk-list, ...) werden immer im zweiten Teilschritt gebildet. Sobald eine dieser Teillisten leer ist ([]) oder nur noch ein Element enthält (Bsp.: kgg-list = [2]) wird sie nicht mehr weiter aufgesplittet, denn sie ist bereits sortiert. Im vierten Schritt werden dann alle Teillisten in der richtigen Reihenfolge aneinadergefügt. Dabei genügt es, die sortierten xk-list und xg-list mit dem entsprechenden x-piv richtig zusammenzufügen (x steht für eine Kombination von k und g).

### Bemerkung

Wie man deutlich sieht, können k-list und g-list zur gleichen Zeit sortiert werden. Dass parallele Verarbeitung möglich ist, wird im Datenfluss-Diagramm besonders deutlich sichtbar.

## Lösung 3.2

In der Lösungsbeschreibung zur Aufgabe 3.1 ist erwähnt worden, dass Listen, die nur ein Element enthalten nicht aufgesplittet werden müssen, weil sie bereits sortiert sind. Der Einfachheit wegen wird im untenstehenden Datenfluss-Diagramm auf diese Verbesserung verzichtet, denn im *if* wird lediglich getestet, ob eine Liste leer (nil) ist.




### Bemerkungen

*k-liste* bezeichnet diejenige Liste, deren Elemente kleiner als *pivot* sind

*g-liste* bezeichnet diejenige Liste, deren Elemente grösser oder gleich *pivot* sind

# Mini-Didaktik (Expertenrunde)

## Wie vermittle ich mein erworbenes Expertenwissen an meine Mitstudenten

Sie haben sich während des Selbststudiums viel neues Wissen angeeignet. Dieses Expertenwissen gilt es nun in der Unterrichtsrunde weiterzugeben. Das ist jedoch nicht ganz einfach. Um es Ihnen etwas zu vereinfachen, haben wir Ihnen sechs Tips aufgeschrieben. Lesen Sie diese sorgfältig durch:

### 1. Überblick in drei Sätzen

Geben Sie den anderen Studenten in drei Sätzen einen Überblick über das, was Sie bei Ihrem Stoffgebiet gelernt haben. Sagen Sie Ihnen nur, was für Sie das Wichtigste war.

### 2. Was wissen oder können die Zuhörer nachher

Sagen Sie Ihren Mitstudenten, was sie nach der Unterrichtsrunde von Ihrem Stoffgebiet wissen und was sie nachher können müssen. Orientieren Sie sich dafür an den Lernzielen vorne in den Unterlagen.

### 3. Unterrichtsblock

Jetzt wird es ernst! Sie tragen nun den anderen vor, was Sie im Selbststudium gelernt haben. Vielleicht machen Sie eine kleine Zeichnung oder Sie verweisen auf bereits besprochene Beispiele. Wichtig ist, dass sich Ihre Zuhörer etwas vorstellen können.

### 4. Zusammenfassung

Am Schluss fassen Sie das Wichtigste des Paradigmas nochmals kurz zusammen. Zwei, drei Sätze reichen!

### 5. Keine unnötigen Fremdwörter

Sprechen Sie einfach! Verwenden Sie nur wenige Fremd- und Fachwörter. Ihre Mitstudenten haben nicht so viel Zeit, um sich an den neuen Wortschatz zu gewöhnen.

### 6. Zeitreserve für Fragen

Sie müssen damit rechnen, dass Ihre Zuhörer Ihnen einen Haufen Fragen stellen werden. Achten Sie also bei der Zeiteinteilung Ihres Unterrichtes darauf, dass Sie genügend Reserve für diese Fragen einplanen.

## Reihenfolge in der Unterrichtsrunde

Die Reihenfolge in der Unterrichtsrunde ist durch die Themen vorbestimmt: objektorientiertes Paradigma, funktionales Paradigma, Logik-Paradigma, Datenfluss-Paradigma.

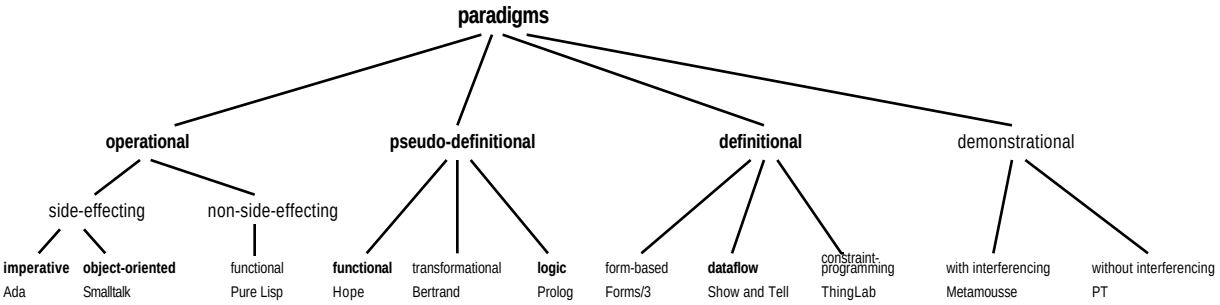
# Verwendete Quellen

Ambler A. L., Burnett M. M., Zimmerman B. A.: Operational Versus Definitional: A Perspective on Programming Paradigms. In: IEEE Computer, September 1992. Seiten 28 bis 42.

# Projektbeschrieb für die Lehrperson

## 1 Im Puzzle behandelte Paradigmen

Die folgende Grafik fasst die wichtigsten Paradigmen von Programmiersprachen zu einer hierarchischen Struktur zusammen. Auf der untersten Zeile sind typische Programmiersprachen angegeben.




In unserer Arbeit haben wir die folgenden berücksichtigt:

imperative

object-oriented

functional (pseudo-definitional)

logic

dataflow

### Gründe für die Wahl

Grundsätzlich haben wir bei der Wahl darauf geachtet, dass die Paradigmen in der Praxis auch tatsächlich eingesetzt werden, das heisst dass Programmiersprachen zu diesen Paradigmen existieren.

Eine Ausnahme bildet dabei das Datenfluss-Paradigma. Wir haben dieses gewählt um zu zeigen, wie «man auch noch programmieren könnte». Ausserdem ist es interessant, weil Case Tools ähnlich arbeiten.

Die Unterschiede zwischen «operational functional» und «pseudo-definitional functional» sind zu klein, als das sie in unserer Unterrichtseinheit exakt ausgearbeitet werden könnten. Wir haben uns deshalb auf «pseudo-definitional functional» beschränkt.

## 2 Die Lernziele

Die folgende Liste enthält die Lernziele, die wir als wichtig erachten und die wir in den Stoff einbringen wollen. Sie sind aber nicht identisch mit den Lernzielen, die explizit im Unterrichtsmaterial erscheinen.

Wissen, was der Ausdruck «Paradigmen von Programmiersprachen» bedeutet.

Wissen, dass verschiedene Paradigmen von Programmiersprachen existieren.

Die Studenten wissen von den besprochenen Paradigmen:
- die Namen
- wie stark der Kontrollfluss durch das Paradigma vorgegeben ist
- wie ein Programm optisch aussieht
- welches die Grundkonstrukte sind

Kennen der wesentlichen Unterschiede zum imperativen Paradigma.

Einfache Programme in einer zum Paradigma passenden Syntax formulieren können.

## 3 Zeitaufwand

Mindestens 8 Lektionen.

## 4 Welches Vorwissen bringen die Studenten mit

Die Studenten arbeiten seit mindestens einem Jahr mit einer imperativen Programmiersprache. Sie sind in der Lage, kleinere Programme zu schreiben und zu analysieren.

Es wird nicht vorausgesetzt, dass die Studenten die Prädikatenlogik erster Stufe beherrschen. Für das Logik-Paradigma sind jedoch gewisse Kenntnisse der Logik notwendig. Es genügt aber, wenn die Studenten die Grundzüge der Aussagenlogik kennen (disjunktive, konjunktive Normalform) und die de-Morganschen Gesetze anwenden können.

## 5 Gruppeneinteilung

Es können maximal vier Gruppen gebildet werden. Falls nötig, kann auch mit drei Gruppen gearbeitet werden.

## 6 Bemerkungen zum Selbststudium

### Pizzabeispiel

Im ersten Abschnitt des Selbststudiums lesen die Studenten ein in dem zu behandelnden Paradigma formuliertes Programm, die «digitale Pizza». Dazu bekommen Sie nur die nötigsten Hinweise. Parallel dazu können sie im imperativen Beispiel nachlesen, wie die Pizza gemacht werden soll. Dies sollte es ihnen ermöglichen, die Fragen zu diesem Abschnitt zu beantworten. Es geht uns hier vor allem darum, dass die Studenten sich zuerst einmal mit dem Stoff bekanntmachen und einige Überlegungen dazu anstellen. Dazu müssen sie das Beispiel nicht im Detail verstehen.

### Vermittlung des Lernstoffes

Im zweiten Abschnitt werden den Studenten die wichtigsten Punkte der jeweiligen Paradigmen vorgestellt. Dabei haben wir darauf geachtet, dass die Studenten entdeckend lernen können. Falls die Studenten bei einzelnen Fragen Probleme haben, können sie unter «Hilfe» ein paar Tips nachlesen. Bei den einzelnen Punkten werden auch immer wieder Vergleiche mit dem imperativen Paradigma gemacht.

### Studenten-Lernkontrolle

Im letzten Abschnitt können die Studenten prüfen, ob sie den Stoff wirklich verstanden haben.

## 7 Bemerkungen zur Expertenrunde

In der Expertenrunde werden die Ergebnisse des Selbststudiums diskutiert und allfällige Missverständnisse ausgeräumt. Hier haben Sie als Lehrperson auch die Möglichkeit, allen Experten einer Gruppe gewisse Ergänzungen und wichtige Hinweise mitzuteilen.

## 8 Bemerkungen zur Unterrichtsrunde

Wir schlagen vor, dass die Paradigmen in der Unterrichtsrunde in der folgenden Reihenfolge behandelt werden: objektorientiert, funktional, Logik, Datenfluss. Der Grund: Der Kontrollfluss wird bei den einzelnen Paradigmen unterschiedlich streng durch das Paradigma selber festgelegt. Beim imperativen wie auch beim objektorientierten Paradigma wird ein streng sequentieller Ablauf verlangt. Beim funktionalen und beim Logik-Paradigma hingegen kommt die Sequenz nur noch limitiert zum tragen. Und beim Datenfluss-Diagramm kann man überhaupt nicht mehr von Sequenz sprechen.

## 9 Lehrer-Lernkontrolle/Test

Wir erachten den Wissensvorsprung der Experten gegenüber den anderen Studenten als enorm. Deshalb ist es schwierig, Aufgaben auszuarbeiten, die beiden gerecht werden. Für einen Teil der Aufgaben sollten alle Studenten Expertenniveau haben. Deshalb schlagen wir Ihnen vor, den Stoff weiter zu vertiefen, wenn Sie die Aufgaben unverändert übernehmen wollen.

## 10 Welches Material wann abgeben

Jeder Student erhält zu Beginn des Puzzles:

Arbeitsanleitung und Lernziele

Imperatives Pizzabeispiel

Alle Mitglieder der betreffenden Expertenrunde erhalten zu Beginn des Selbststudiums:

Arbeitsblätter fürs Selbststudium

Lösungsblätter zum Selbststudium

Pizzabeispiel des zu bearbeitenden Paradigmas

Die Aufgaben und Lösungen der Studenten-Lernkontrolle können Sie auch erst am Ende des Selbststudiums abgeben.

Damit sich die Studenten gut auf die Unterrichtsrunde vorbereiten können, geben Sie ihnen vor der Expertenrunde das Merkblatt «Mini-Didaktik» ab.

## 11 Wo Probleme auftreten könnten

Wir glauben, dass jene Studenten, welche das Logik-Paradigma in der Expertenrunde erhalten, die schwierigste Aufgabe haben. Es scheint uns daher naheliegend, diese Aufgabe den interessiertesten und besseren Studenten zu geben.

Die zur Verfügung stehende Zeit von 8 Lektionen ist eher knapp bemessen.

Es ist denkbar, dass die Studenten für das Selbststudienmaterial unterschiedlich viel Zeit benötigen.

## 12 Paradigmen und Programmiersprachen

Die in den Beispielen verwendeten Syntaxen lehnen sich stark an bestehende Programmiersprachen an. Beim imperativen Paradigma an Ada, beim objektorientierten an Smalltalk, beim funktionalen an Hope und beim logischen Paradigma an Prolog.

Teilweise werden im Selbststudienmaterial Punkte behandelt, die nicht zum eigentlichen Paradigma, sondern zu einer Programmiersprache gehören. Dies ist zum Beispiel beim Backtracking des logischen Paradigmas der Fall. Das logische Paradigma schreibt Backtracking nicht vor. Es könnten auch andere Strategien gewählt werden. Trotzdem wollten wir den Studenten zeigen, wie beim logischen Paradigma eine Lösung gefunden wird. Deshalb waren wir gezwungen, uns auf eine Methode festzulegen.

# Lehrer-Lernkontrolle/Test Serie A

## Lehrertest Objektorientiertes Paradigma: Serie A

### Aufgabe 1.1

Es gibt viele Möglichkeiten, wie man ein Problem darstellen und lösen kann. Die verschiedenen Paradigmen liefern unterschiedliche Ansätze dazu.

Erklären Sie die grundlegenden Ideen, die hinter dem *objektorientierten* Paradigma stehen. Beschreiben Sie dann zwei zentrale Punkte des objektorientierten Paradigmas.

Ihre Antwort beinhaltet drei Teile. Im ersten Teil beschreiben Sie allgemein die wesentlichen Grundzüge des objektorientierten Paradigmas. In den beiden anderen Teilen gehen Sie auf die beiden wichtigen Punkte ein. Jeder Teil beinhaltet ungefähr zwei bis fünf Sätze.

Falls Sie nur auf einen Punkt eingehen, den ersten Teil aber richtig beantwortet haben, wird Ihre Lösung als genügend bewertet.

### Aufgabe 1.2

Objektorientierte Programmiersprachen eignen sich zur Implementation von grafischen Editoren und Benutzeroberflächen.

Entwerfen Sie eine Klassenhierarchie für einen grafischen Editor. Der Editor kennt die Objekte Dreieck, Kreis und Rechteck.

Für alle Objekte gilt: Eine Bounding Box, bestehend aus Position x,y sowie Breite b und Höhe h, wird gespeichert. Die Bounding Box ist ein Rechteck, das die betreffende Figur umschliesst. Für einen Kreis mit Radius r wären somit b=2r und h=2r. Die Koordinaten x,y bezeichnen die linke untere Ecke der Bounding Box. Dicke und Farbe der Linien, mit denen das Objekt gezeichnet wird, werden gespeichert. Die Operationen *Zeichnen* und *Verschieben* sollen von allen Objekten ausgeführt werden können.

Für den Kreis gilt: Die Koordinaten des Kreismittelpunktes müssen festgehalten werden. Kreise sollen skaliert werden können .

Rechteck: Die Koordinaten für die Eckpunkte sollen gespeichert werden. Als spezielle Operationen wird *Rotieren* verlangt.

Dreieck: Koordinaten für die Eckpunkte.

Sie müssen das Beispiel nicht programmieren. Stellen Sie die Klassenhierarchie dar. Verwenden Sie dazu eine Baumstruktur. Geben Sie zu jeder Klasse die Variablen und die Methoden an.

Hier wird geprüft, ob Sie das Prinzip der Vererbung verstanden haben. Es wird keine detaillierte Ausarbeitung verlangt.

### Hinweis

Sie können davon ausgehen, dass die Operationen *Zeichnen* und *Verschieben* für alle Klassen identisch sind.

# Lehrertest Funktionales Paradigma: Serie A

## Aufgabe 2.1

Es gibt viele Möglichkeiten, wie man ein Problem darstellen und lösen kann. Die verschiedenen Paradigmen liefern unterschiedliche Ansätze dazu.

Erklären Sie die grundlegenden Ideen, die hinter dem *funktionalen* Paradigma stehen. Beschreiben Sie dann zwei zentrale Punkte des funktionalen Paradigmas.

Ihre Antwort beinhaltet drei Teile. Im ersten Teil beschreiben Sie allgemein die wesentlichen Grundzüge des funktionalen Paradigmas. In den beiden anderen Teilen gehen Sie auf die beiden wichtigen Punkte ein. Jeder Teil beinhaltet ungefähr zwei bis fünf Sätze.

Falls Sie nur auf einen Punkt eingehen, den ersten Teil aber richtig beantwortet haben, ist Ihre Lösung als genügend bewertet.

## Aufgabe 2.2

Das folgende Programm widerspricht dem funktionalen Paradigma. Suchen Sie die fehlerhaften Stellen. Finden Sie den zentralen Punkt, in welchem das funktionale Paradigma verletzt wird. Erklären Sie in Ihrer Antwort diesen Punkt.

```
var found: boolean;

dec f: list(num)#num -> list(num);
--- f([],x) <= [];
--- f(erstesElement::rest,x) <=
        if erstesElement = x then
           f(rest,x); found:=true;
        else
           found:=false; erstesElement::f(rest,x);
```

Als Antwort werden zwei bis fünf Sätze erwartet.

Ohne Erklärung ist Ihre Antwort nicht genügend.

## Hinweis

Das Programm soll ein Element *x* aus einer Liste entfernen. Falls sich das Element in der Liste befindet, wird *found* gleich true gesetzt.

# Lehrertest Logik-Paradigma: Serie A

## Aufgabe 3.1

Es gibt viele Möglichkeiten, wie man ein Problem darstellen und lösen kann. Die verschiedenen Paradigmen liefern unterschiedliche Ansätze dazu.

Erklären Sie die grundlegenden Ideen, die hinter dem *Logik*-Paradigma stehen. Beschreiben Sie dann drei zentrale Punkte des Logik-Paradigmas.

Ihre Antwort beinhaltet vier Teile. Im ersten Teil beschreiben Sie allgemein, die grundlegende Idee, die hinter dem Logik-Paradigma steht. In den drei anderen Teilen gehen Sie auf drei wichtige Punkte ein. Jeder Teil beinhaltet ungefähr zwei bis fünf Sätze.

Falls Sie nur auf zwei Punkte eingehen, den ersten Teil aber richtig beantwortet haben, wird Ihre Lösung als genügend bewertet.

## Aufgabe 3.2

Mit Hilfe von logischen Programmiersprachen kann auf einfache Art getestet werden, ob ein Ausdruck, die verlangte Form einhält oder nicht.

Ihre Aufgabe ist es, ein Logikprogramm zu schreiben, welches einfache mathematische Ausdrücke auf ihre Korrektheit untersucht. Die zu untersuchenden Ausdrücke bestehen lediglich aus Zahlen, +, * und =, wobei folgende Form als korrekt betrachtet wird:

```
Ausdruck  ::=   Summe '='.
Summe     ::=   Faktor {'+' Faktor}.
Faktor    ::=   Zahl {'*' Zahl}.
```

Mit den geschweiften Klammern wird angedeutet, dass deren Inhalt keinmal oder beliebig oft vorkommen kann.

Beispiel eines *korrekten* Ausdruckes: 2*3+4*5+6+7*8*9=
Beispiel eines *unkorrekten* Ausdruckes: 2*3+*5+6+7*8*9=

In diesem Beispiel geht es vor allem darum, dass Sie für die Lösung einen richtigen Ansatz wählen. Auch mit Syntax- oder sonstigen kleinen Fehlern wird sie immer noch als genügend benotet.

### Hinweise

Verwenden Sie zum Testen von Zahlen die vordefinierte Regel *numeric(X)*.

Eine Variable *X* kann mit einem Ausdruck 2*3+4*5 unifizieren.

# Lehrertest Datenfluss-Paradigma: Serie A

## Aufgabe 4.1

Es gibt viele Möglichkeiten, wie man ein Problem darstellen und lösen kann. Die verschiedenen Paradigmen liefern unterschiedliche Ansätze dazu.

Erklären Sie die grundlegenden Ideen, die hinter dem *Datenfluss*-Paradigma stehen. Beschreiben Sie dann zwei zentrale Punkte des Datenfluss-Paradigmas.

Ihre Antwort beinhaltet drei Teile. Im ersten Teil beschreiben Sie allgemein, die grundlegende Idee, die hinter dem Datenfluss-Paradigma steht. In den beiden anderen Teilen gehen Sie auf die beiden wichtigen Punkte ein. Jeder Teil beinhaltet ungefähr zwei bis fünf Sätze.

Falls Sie nur auf einen Punkt eingehen, den ersten Teil aber richtig beantwortet haben, wird Ihre Lösung als genügend bewertet.

## Aufgabe 4.2

Schon in der Grundschule mussten Sie in der Lage sein, von zwei natürlichen Zahlen den grössten gemeinsamen Teiler (ggT) zu bestimmen. Solange diese Zahlen klein sind, ist dies ein einfaches Unterfangen; bei grossen Zahlen kann es aber recht aufwendig werden.

Zeichnen Sie ein Datenfluss-Diagramm, welches aus einem Datenpaket (a, b) den ggT rekursiv bestimmt.

In diesem Beispiel geht es vor allem darum, dass Sie für die Lösung einen richtigen Ansatz wählen. Auch mit kleinen Fehlern wird sie immer noch als genügend benotet.

### Hinweise

Der ggT von (a, b) kann wie folgt bestimmt werden: Annahme: a ist grösser als b. Man subtrahiert b solange von a bis beide gleich gross sind oder a kleiner b ist. Wenn dann b grösser als a ist, so subtrahiert man a von b, bis beide gleich sind oder a grösser b ist. Im zweiten Fall beginnt das Spiel wieder von vorne.

Ein Datenpaket ist ein Verbund von einzelnen Datenwerten. In einer imperativen Programmiersprache wäre ein Datenpaket ein Record.

## Lösungen zur Serie A

### Antwort 1.1 (K2)

Grundlegende Ideen (2 Punkte)

Beim objektorientierten Paradigma versucht man, «real existierende Objekte» in das Programm abzubilden. Objekte können Operationen ausführen. Die Operationen werden ausgeführt, indem die Methoden des Objektes gestartet werden. Zu diesem Zweck schickt man dem Objekt eine Nachricht.

Von den nachfolgenden Punkten sollten zwei erwähnt werden.

Klassenhierarchie (1 Punkt)

Objekte mit ähnlichen Eigenschaften werden zu einer Hierarchie zusammengefasst. Es wird eine Oberklasse definiert, in der die gemeinsamen Variablen und Methoden deklariert werden. Hier werden also die gemeinsamen Eigenschaften der Klassen festgelegt. In den Subklassen können nun spezielle Variablen und Methoden, die nur für diese Klasse gültig sind, deklariert werden. So werden die Unterschiede zwischen den Klassen ausgedrückt.

Vererbung (1 Punkt)

Die Vererbung hängt eng mit der Klassenhierarchie zusammen. Variablen und Methoden der Oberklasse werden an die Unterklassen vererbt. Dies bedeutet, dass auch Objekte der Unterklassen sämtliche Variablen und Methoden der Oberklasse zur Verfügung haben. Spezielle Variablen und Methoden der Unterklasse sind Objekten der betreffenden Unterklasse vorbehalten.

Nachrichten und Methoden (1 Punkt)

Jedes Objekt besitzt eine Anzahl Methoden. Die Methoden werden dazu verwendet, die Instanzvariablen des Objektes zu verändern, oder sonstige Manipulationen durchzuführen. Instanzen verschiedener Klassen können Methoden gleichen Namens haben. Die Operationen dieser Methoden sind aber völlig unabhängig voneinander.

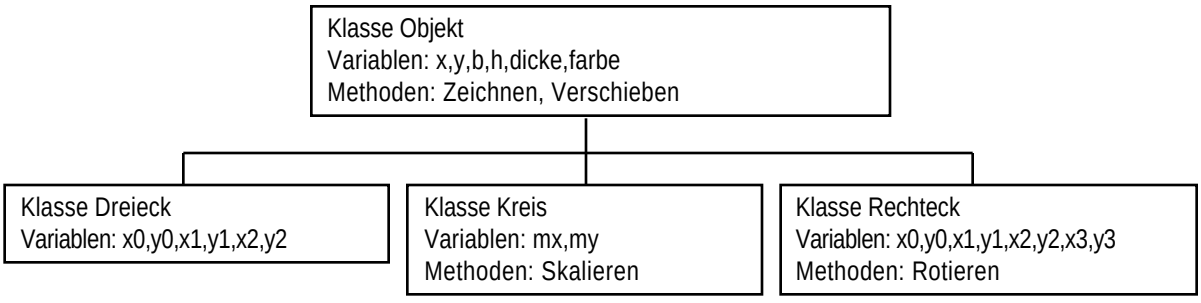
Eine Methode wird gestartet, indem man dem betreffenden Objekt eine Nachricht sendet.

Einkapselung (1 Punkt)

Auf die Variablen eines Objektes kann nur mittels Methoden zugegriffen werden. Die Variablen sind gegen aussen nicht sichtbar, sondern «eingekapselt».

### Antwort 1.2 (K3)

Es können maximal 3 Punkte vergeben werden.



### Antwort 2.1 (K2)

Grundlegende Ideen (2 Punkte)

Beim funktionalen Paradigma wird die Lösung mit einer Sammlung von Funktionen spezifiziert. Die Funktionen haben die gleiche Bedeutung wie in der Mathematik. Sie repräsentieren einen Wert. Die Reihenfolge der Funktionsdeklaration hat keinen Einfluss auf die Lösung (Annahme: Deterministische Funktionen).

Zwei der folgenden Punkte sollten erwähnt werden:

Definieren von Funktionen (1 Punkt)

Beim funktionalen Paradigma wird ein Problem formuliert, indem man es als eine Sammlung von Funktionen darstellt. Diese Funktionen haben einen Definitions- und einen Wertebereich, genau wie in der Mathematik.

Pattern Matching (1 Punkt)

Will man eine Funktion aufrufen, so muss der Funktionsaufruf mit dem «Kopf» einer Funktion im Programm übereinstimmen. Funktionsaufruf und Kopf der Funktion müssen das gleiche Pattern aufweisen.

Der Ersatz für den Loop (1 Punkt)

Das funktionale Paradigma kennt keine for- oder while-Konstrukte. Statt Iteration wird Rekursion verwendet.

Auswertungsreihenfolge (1 Punkt)

Grundsätzlich braucht man sich beim funktionalen Programmieren nicht um die Reihenfolge der Funktionsdeklaration und der Funktionsauswertung zu kümmern. Mit Pattern Matching und dem if-Konstrukt wird implizit eine Auswertungsreihenfolge festgelegt.

Side-Effects (1 Punkt)

Eine Wertzuweisung an eine Variable bezeichnet man als side-effect. Beim funktionalen Paradigma gibt es nur Variablen für Zwischenwerte. Beim imperativen Paradigma können die Variablen während der Berechnung laufend neue Werte annehmen. Bei der Termination des Programms ist das Resultat durch die Variablenbelegung gegeben. Im Gegensatz dazu ist das funktionale Paradigma non-side-effecting.

### Antwort 2.2 (K2)

Das funktionale Paradigma erlaubt keine side-effects. Somit sind die *kursiv* hervorgehobenen Ausdrücke nicht korrekt. (1,5 Punkte)

```
var found: boolean;

dec f:list(num)#num -> list(num);
--- f([],x) <= [];
--- f(erstesElement::rest,x) <=
        if erstesElement = x then
           f(rest,x); found:=true;
        else
           found:=false; erstesElement::f(rest,x);
```

Für die Erklärung können maximal 1,5 Punkte vergeben werden.

Eine Wertzuweisung an eine Variable bezeichnet man als side-effect. Beim funktionalen Paradigma gibt es nur Variablen für Zwischenwerte. Es existieren keine Variablen wie zum Beispiel beim imperativen Paradigma, die bei Termination des Programms das Resultat der Berechnung darstellen. Funktionale Sprachen sind non-side-effecting.

## Antwort 3.1 (K2)

(4 Punkte)

Grundlegende Ideen (2 Punkte)

Ein Logikprogramm enthält nur die Logik-Komponente eines Algorithmus'. Die Logik wird durch Fakten und Regeln dargestellt. Das System versucht ein vom Benutzer eingegebenes Goal mit Hilfe der Fakten und Regeln zu beweisen. Dabei wird Unifikation und Backtracking verwendet.

Von den nachfolgenden Punkten sollten drei erwähnt werden. Die maximalen 4 Punkte können nur erteilt werden, wenn die grundlegenden Ideen richtig wiedergegeben wurden.

Fakten und Regeln (1 Punkt)

Regeln bestehen aus einem Kopf und einem Körper. Fakten sind ein Spezialfall von Regeln, deren Körper immer true ist. Damit ein Regelkopf erfüllt (true) ist, muss zuerst der Regelkörper erfüllt werden. Aus den Fakten und Regeln lässt sich neues Wissen ableiten.

Hornklauseln (1 Punkt)

Fakten, Regeln und Anfragen werden in Form von Hornklauseln formuliert. In einem Logikprogramm stehen nur Dinge, von denen man annimmt, dass sie wahr sind. Hornklauseln ermöglichen es, in nützlicher Zeit eine Variablenbelegung zu finden, welche eine Formel wahr macht.

Unifikation und Goal (1 Punkt)

Damit ein Goal oder Subgoal bewiesen werden kann, muss zumindest ein Faktum oder ein Regelkopf im Programm vorhanden sein, der mit dem Goal unifiziert. Die Unifikation selber ist ein Pattern-Matching, bei dem Variablen mit einem Wert versehen werden oder Variablen an eine andere Variable gebunden werden.

Beweisprozess (1 Punkt)

Der Benutzer stellt eine Anfrage an das System (Goal). Falls die Anfrage Variablen enthält, versucht das System eine Wertebelegung zu finden, welche das Goal erfüllt. Enthält die Anfrage nur Konstanten, so ist es die Aufgabe des Systems zu überprüfen, ob das Goal erfüllbar ist. Im Falle dass ein Goal mit einem Regelkopf unifiziert, muss der Regelkörper bewiesen werden. Dazu werden neue Subgoals lanciert.

Backtracking (1 Punkt)

Sehr oft gibt es für ein Faktum oder einen Regelkopf mehrere Ausführungen. Das System muss sich also immer für eine Ausführung entscheiden. Wenn ein Subgoal fehlschlägt, geht das System dorthin zurück, wo es sich das letzte Mal entscheiden musste und wählt eine Alternative. Falls alle Alternativen fehlschlagen, geht es noch einen Schritt weiter zurück. Backtracking hat zur Folge, dass alle denkbaren Beweiswege durchprobiert werden.

## Antwort 3.2 (K3)

(5 Punkte)

Das untenstehende Logikprogramm wiederspiegelt die Definition der Korrektheit in der Aufgabenstellung sehr deutlich.

```
rechnung(R=) :- summe(R).
```

```
summe(S) :- faktor(S).
summe(S+S1) :- faktor(S), summe(S1).

faktor(F) :- numeric(F).
faktor(F*F1) :- numeric(F), faktor(F1).
```

Für diese Aufgabe können Sie maximal 5 Punkte vergeben. Falls nur Syntaxfehler vorhanden sind, werden mindestens 4 Punkte erteilt.

### Antwort 4.1 (K2)

(4 Punkte)

Grundlegende Ideen (2 Punkte)

Im Datenfluss-Paradigma stehen die Daten und deren Verarbeitung im Vordergrund. Ein Datenpaket fliesst in einen Knoten hinein, wird bearbeit und fliesst wieder hinaus. Die Aufgabe des Programmierers ist es, die Gleichungen in den Knoten und die Verbindungen zwischen den Knoten zu spezifizieren. Datenfluss-Programme werden am natürlichsten mittels Datenfluss-Diagrammen dargestellt.

Von den nachfolgenden Punkten sollten zwei erwähnt werden. Die maximalen 4 Punkte können nur erteilt werden, wenn die grundlegenden Ideen richtig wiedergegeben wurden.

Datenfluss-Diagramm (1 Punkt)

Das Datenfluss-Diagramm ist ein gerichteter Graph. In diesem Netzwerk von Knoten und Kanten fliessen Datenströme. Dabei konsumiert jeder Knoten die einfliessenden Daten und berechnet neue Daten, die zum nächsten Knoten fliessen.

Knotengleichungen und Ablaufsteuerung (1 Punkt)

Die Gleichungen in den Knoten bestimmen durch ihre gegenseitige Abhängigkeit die Abarbeitungsreihenfolge des Programmes. Kontrollfluss-Konstrukte, wie sie in imperativen Programmiersprachen verwendet werden, sind hier überflüssig. Die Ausgangsdaten lassen sich als Funktion der Eingangsdaten auffassen, wobei die natürliche Einschränkung gilt, dass nur bereits produzierte Daten als Funktionsparameter verwendet werden dürfen.

Ausführungsmodelle (1 Punkt)

Es existieren zwei verschiedene Ausführungsmodelle:

data-driven: Ein Knoten berechnet die Ausgabedaten, sobald *alle* benötigten Eingabedaten vorhanden sind.

demand-driven: Ein Knoten berechnet nur auf verlangen die Ausgabedaten. Auch hier müssen alle Eingabedaten vorhanden sein.

Vergleich mit anderen Paradigmen

Datenfluss: Programm ist eine Menge von Gleichungen
Funktional: Programm lässt sich als eine Gleichung auffassen

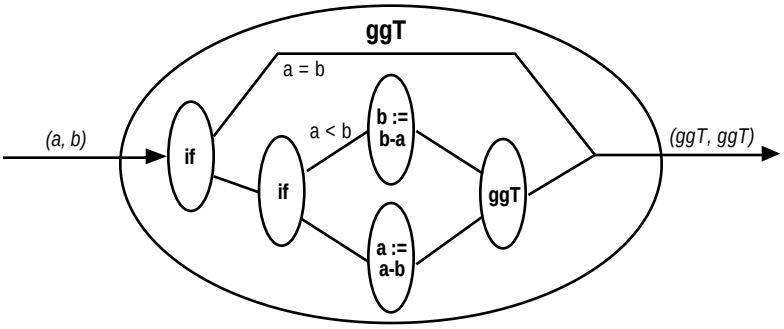
Datenfluss: Knoten operieren auf Datenströme
Funktional: Funktionen operieren auf einem Datenpaket

Datenfluss: Kontrollfluss-Konstrukte sind überflüssig
Imperativ: Kontrollfluss-Konstrukte steuern den Ablauf des Programmes

## Antwort 4.2

(5 Punkte)

Das Datenfluss-Diagramm zeigt grafisch auf, wie der grösste gemeinsame Teiler *ggT* von zwei natürlichen Zahlen *a* und *b* bestimmt wird.




Für diese Aufgabe können Sie maximal 5 Punkte vergeben. Es ist denkbar, dass der Student eine anderes korrektes Datenfluss-Diagramm aufzeichnet, welches jedoch nicht optimal ist. In diesem Fall sind mindestens 4 Punkte zu erteilen.

# Lehrer-Lernkontrolle/Test Serie B

## Lehrertest Objektorientiertes Paradigma: Serie B

### Aufgabe 1.1

Beim *objektorientierten* Paradigma können Objekte Nachrichten empfangen und Methoden ausführen. Aber erst durch Klassenhierarchie und Vererbung kommen die Stärken des objektorientierten Paradigmas richtig zum Ausdruck.

Erklären Sie die Begriffe Klassenhierarchie und Vererbung.

Ihre Antwort besteht aus fünf bis zehn Sätzen und kann, falls nötig, durch ein Beispiel ergänzt werden.

Falls Sie nur auf einen Punkt eingehen, ist Ihre Lösung nicht genügend.

### Aufgabe 1.2

Warteschlangen werden in vielen Programmen verwendet. So zum Beispiel bei der Prozessorzuteilung bei Mehrprozesssystemen. Jeder Prozess hat eine bestimmt Rechenzeit zur Verfügung. Ist diese abgelaufen, wird der Prozess in eine Warteschlange einfügt. Dann beginnt ein anderer Prozess aus der Warteschlange zu laufen. In diesem Beispiel soll eine Klassenhierarchie für verschiedene Warteschlangen erstellt werden.

Es gibt drei Typen von Warteschangen:

FIFO: First In First Out. Das Element, das zuerst eingefügt worden ist, wird auch zuerst entfernt.

LIFO: Last In First Out. Das Element, das zuletzt eingefügt worden ist, wird zuerst entfernt.

Priority: Jedes Element in der Warteschlange besitzt eine Priorität. Das Element mit der höchsten Priorität wird zuerst entfernt.

Operationen für alle Typen:

Abfragen der Anzahl Elemente in der Warteschlange.

Einfügen eines Elementes in die Warteschlange.

Entfernen des «nächsten Elementes» gemäss der Strategie (FIFO, LIFO, Priority) der Warteschlange.

Zusätzlich für Priority:

Entfernen eines Elementes mit einer bestimmten Priorität.

a) Entwerfen Sie eine Klassenhierarchie für dieses Problem. Sie müssen das Beispiel nicht programmieren. Stellen Sie nur die Klassenhierarchie dar. Verwenden Sie dazu eine Baumstruktur. Geben Sie zu jeder Klasse die Methoden an.

b) Beschreiben Sie die Aktionen der Methoden mit einigen Stichworten.

Hier wird geprüft, ob Sie das Prinzip der Vererbung verstanden haben. Es wird keine detaillierte Ausarbeitung verlangt.

Ihre Ausarbeitung muss nicht vollständig korrekt sein. Wenn Sie aber nur a) oder b) beantworten, ist Ihre Lösung nicht genügend.

### Hinweis

Überlegen Sie sich, ob die Operationen *Einfügen* und *Entfernen* für die verschiedenen Typen von Warteschlangen mit gemeinsamen oder separaten Methoden implementiert werden könnten.

## Lehrertest Funktionales Paradigma: Serie B

### Aufgabe 2.1

Beim imperativen Paradigma wird durch das zeilenweise, sequentielle Abarbeiten des Programms eine Lösung berechnet. Beim *funktionalen* Ansatz wird eine grundlegend andere Strategie verfolgt.

Beschreiben Sie, wie beim funktionalen Ansatz die Lösung gefunden wird. Gehen Sie, falls notwendig, auch auf die wesentlichen Grundzüge des funktionalen Paradigmas ein.

Als Antwort werden fünf bis zehn Sätze erwartet. Falls nötig, können Sie bei Ihrer Erklärung auch die Unterschiede zum imperativen Paradigma hervorheben.

Wenn Sie den Stoff verstanden haben, sollte diese Frage für Sie kein Problem darstellen. Es werden deshalb nur null Punkte oder die maximale Anzahl Punkte vergeben.

### Aufgabe 2.2

Lesen Sie den untenstehenden Text durch. Stimmt die Aussage?

Falls Sie der Meinung sind, dass der Text falsch ist, so begründen Sie Ihre Antwort mit zwei bis fünf Sätzen. Wenn Sie der Ansicht sind, das der Text korrekt ist, müssen Sie Ihre Antwort nicht begründen.

«Beim funktionalen Paradigma wird die Lösung mit einer Sammlung von Funktionen spezifiziert. Das Resultat einer Funktion ist vom Zeitpunkt der Auswertung abhängig.

Will man eine Funktion aufrufen, so muss der Funktionsaufruf mit dem «Kopf» einer Funktion im Programm übereinstimmen. Funktionsaufruf und Kopf der Funktion müssen das gleiche Pattern aufweisen.

Grundsätzlich braucht man sich beim funktionalen Programmieren nicht um die Reihenfolge der Funktionsdeklaration und der Funktionsauswertung zu kümmern. Mit Pattern Matching und dem if-Konstrukt wird implizit eine Auswertungsreihenfolge festgelegt (Annahme: Deterministische Funktionen).

Die Lösung wird beim funktionalen Paradigma durch Iteration gefunden.

Die Wertzuweisung an eine Variable bezeichnet man als side-effect. Bei funktionalen Paradigmen gibt es nur Variablen für Zwischenwerte. Es existieren keine Variablen wie zum Beispiel beim imperativen Paradigma, die bei Termination des Programms das Resultat der Berechnung darstellen. Funktionale Sprachen sind non-side-effecting.»

## Lehrertest Logik-Paradigma: Serie B

### Aufgabe 3.1

Beim imperativen Paradigma wird durch das zeilenweise, sequentielle Abarbeiten des Programms eine Lösung berechnet. Beim *Logik-Paradigma* wird ein grundlegend anderer Ansatz verfolgt.

Beschreiben Sie, wie beim Logik-Paradigma die Lösung gefunden wird. Gehen Sie, falls notwendig, auch auf die wesentlichen Grundzüge dieses Paradigmas ein.

Als Antwort werden etwa fünf bis zehn Sätze erwartet.

Wenn Sie den Stoff verstanden haben, sollte diese Frage für Sie kein Problem darstellen. Ihre Antwort wird als genügend bewertet, wenn Sie die Schlagwörter im richtigen Kontext wiedergeben können.

### Aufgabe 3.2

Mit Hilfe von logischen Programmiersprachen kann auf einfache Art die Ableitung von mathematischen Ausdrücken symbolisch bestimmt werden.

Ihre Aufgabe ist es, ein Logikprogramm zu schreiben, welches einfache mathematische Ausdrücke symbolisch ableitet. Die abzuleitenden Ausdrücke bestehen lediglich aus Zahlen, +, * und der Variable *x*. Im abgeleiteten Ausdruck dürfen auch Klammern vorkommen.

Beispiel: $\frac{d(x^2+2x+1)}{dx} = 2 \cdot x \cdot x + 3 \cdot (1 \cdot x + x \cdot 1) + 0 \cdot (x + 2 \cdot 1) + 0 = 6 \cdot x + 2$

Schreiben Sie eine Regel *d(U, V)*, welche dU/dx in *V* zurückgibt. Auf die Vereinfachung des Ausdruckes in V können Sie verzichten. Als kleine Hilfe geben wir Ihnen die Ableitung von *x* vor: *d(x, 1)*.

In diesem Beispiel geht es vor allem darum, dass Sie für die Lösung einen richtigen Ansatz wählen. Auch mit Syntax- oder sonstigen kleinen Fehlern wird sie immer noch als genügend benotet.

#### Hinweise

Verwenden Sie zum Testen von Zahlen die vordefinierte Regel *numeric(X)*.

Produkteregel (Ableiten von Produkten): $\frac{d(f(x) \cdot g(x))}{dx} = f(x) \cdot \frac{dg(x)}{dx} + g(x) \cdot \frac{df(x)}{dx}$

# Lehrertest Datenfluss-Paradigma: Serie B

## Aufgabe 4.1

Sie haben die wichtigsten Konzepte des funktionalen und des Datenfluss-Paradigmas kennengelernt. Dabei ist Ihnen sicher aufgefallen, dass zwischen diesen beiden Paradigmen eine gewisse Verwandtschaft besteht.

Erklären Sie in wenigen Sätzen die Gemeinsamkeiten und die Unterschiede dieser beiden Paradigmen.

Ihre Anwort ist gut, falls Sie mindestens zwei Unterschiede und zwei Gemeinsamkeiten finden.

## Aufgabe 4.2

Die Fibonacci-Zahlen sind eine bekannte Zahlenfolge, die sowohl in der Biologie als auch in der Mathematik anzutreffen ist. Sie werden wie folgt rekursiv definiert:

$Fib(0) = 1$
$Fib(1) = 1$
$Fib(n) = Fib(n-1) + Fib(n-2)$, für $n > 1$

Zeichnen Sie ein Datenfluss-Diagramm, welches die n-te Fibonacci-Zahl berechnet.

In diesem Beispiel geht es vor allem darum, dass Sie für die Lösung einen richtigen Ansatz wählen. Auch mit kleinen Fehlern wird sie immer noch als genügend benotet.

## Lösungen zur Serie B

### Antwort 1.1 (K2)

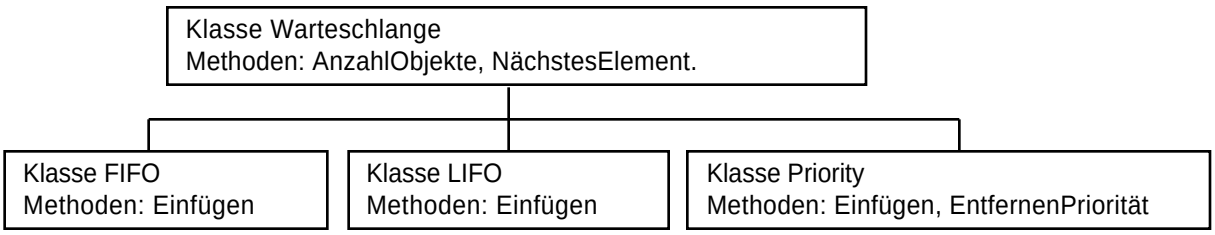
Klassenhierarchie (1,5 Punkte)

Objekte mit ähnlichen Eigenschaften werden zu einer Hierarchie zusammengefasst. Es wird eine Oberklasse definiert, in der die gemeinsamen Variablen und Methoden deklariert werden. Hier werden also die gemeinsamen Eigenschaften der Klassen festgelegt. In den Subklassen können nun spezielle Variablen und Methoden, die nur für diese Klasse gültig sind, deklariert werden. So werden die Unterschiede zwischen den Klassen ausgedrückt.

Vererbung (1,5 Punkte)

Die Vererbung hängt eng mit der Klassenhierarchie zusammen. Variablen und Methoden der Oberklasse werden an die Unterklassen vererbt. Dies bedeutet, dass auch Objekte der Unterklassen sämtliche Variablen und Methoden der Oberklasse zur Verfügung haben. Spezielle Variablen und Methoden der Unterklassen sind den Objekten der betreffenden Unterklasse vorbehalten.

### Antwort 1.2 (K3)




Für die Erklärung können maximal 1,5 Punkte vergeben werden.

Es existiert nur eine Methode *NächstesElement*. Diese nimmt das *erste* Element aus der Warteschlange. Deshalb brauchen alle Unterklassen eigene Methoden *Einfügen*. Die Methoden fügen die Elemente gemäss der Strategie der Warteschlange ein. Als Beispiel: Bei FIFO wird ein Element am Ende der Schlange eingefügt.

### Antwort 2.1 (K2)

Für die Erklärung können maximal 2 Punkte vergeben werden.

Grundsätzlich braucht man sich beim funktionalen Paradigma nicht um die Reihenfolge der Funktionsdeklaration und der Funktionsauswertung zu kümmern. Mit Pattern Matching und dem if-Konstrukt wird implizit eine Auswertungsreihenfolge festgelegt. Dies gilt unter der Voraussetzung, dass die Funktionen deterministisch sind.

Im Unterschied dazu wird beim imperativen Paradigma ein sequentielles Abarbeiten der Programmzeilen vorgeschrieben.

### Antwort 2.2 (K2)

Folgende Aussagen sind falsch:

«Die Lösung wird beim funktionalen Paradigma durch Iteration gefunden.»

Beim funktionalen Paradigma wird die Lösung durch Rekursion, nicht durch Iteration gefunden. Konstrukte wie «while...loop» oder «for...loop» existieren beim funktionalen Paradigma nicht. (1,5 Punkte)

«Beim funktionalen Paradigma wird die Lösung mit einer Sammlung von Funktionen spezifiziert. Das Resultat einer Funktion ist vom Zeitpunkt der Auswertung abhängig.»

Richtig ist nur der erste Satz. Der zweite ist falsch. Die Funktionen haben die gleiche Bedeutung wie in der Mathematik. Sie repräsentieren einen Wert. Das Resultat ist somit *nicht* abhängig vom Zeitpunkt der Auswertung.

### Antwort 3.1 (K2)

(4 Punkte)

Bei einem Logikprogramm wird nur der logische Teil eines Algorithmus' angegeben. Die Abarbeitungsstrategie ist im System vorgegeben und somit problemunabhängig.

Ein Logikprogramm besteht aus *Fakten* und *Regeln*, wobei Fakten ein Spezialfall von Regeln darstellen, deren Regelkörper immer erfüllt sind. Das System versucht ein vom Benutzer eingegebenes *Goal* mit Hilfe der Fakten und Regeln zu *beweisen*. Dabei wird *Unifikation* und *Backtracking* verwendet. Falls das Goal *Variablen* enthält, so werden diese bei einer möglichen Unifikation mit Werten versehen. Backtracking kommt dann zum Zuge, wenn es mehrere Regeln (Fakten) zur Auswahl hat, das System sich für eine entscheiden muss und dabei in eine Sackgasse gerät. Backtracking hat zur Folge, dass alle möglichen Beweiswege ausprobiert werden.

Für die Antwort können Sie maximal 4 Punkte vergeben. Falls alle Schlagwörter im richtigen Kontext wiedergegeben werden, so sind mindestens 3 Punkte zu erteilen.

### Antwort 3.2 (K3)

(5 Punkte)

Das folgende Logikprogramm ist in der Lage einfache (in der Aufgabenstellung charakterisiert) aber beliebige mathematische Ausdrücke symbolisch abzuleiten.

```
d(x, 1).
d(N, 0) :- numeric(N).
d(U+V, U1+V1) :- d(U, U1), d(V, V1).
d(U*V, (U1*V+U*V1)) :- d(U, U1), d(V, V1).
```

Für diese Aufgabe können Sie maximal 5 Punkte vergeben. Falls nur Syntaxfehler vorhanden sind, werden mindestens 4 Punkte erteilt.

### Antwort 4.1 (K4)

(5 Punkte)

Gemeinsamkeiten (3 Punkte):

Ein Programm ist eine Komposition von Funktionen (Ausgabedaten der einen Funktion sind Eingabedaten der nächsten Funktion).

Abarbeitungsreihenfolge wird nicht explizit festgelegt.

Parallele Abarbeitung von Teilsequenzen ist möglich.

Unterschiede (2 Punkte):

Datenfluss: Programm ist eine Menge von Gleichungen.
Funktional: Programm lässt sich als eine Funktion auffassen.

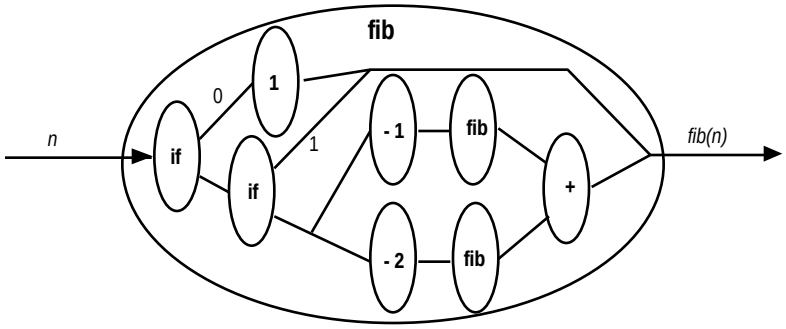
Datenfluss: Knoten operieren auf Datenströme.
Funktional: Funktionen operieren auf einem Datenpaket.

Für diese Aufgabe können Sie maximal 5 Punkte vergeben. Für je zwei richtige Unterschiede und Gemeinsamkeiten sollten Sie 4 Punkte erteilen.

**Antwort 4.2 (K3)**

(5 Punkte)

Das Datenfluss-Diagramm zeigt grafisch, wie die n-te Fibonacci-Zahl berechnet wird. Die mit 1 bezeichnete Ellipse gibt stets 1 als Ausgabewert zurück.




Für diese Aufgabe können Sie maximal 5 Punkte vergeben. Es ist denkbar, dass der Student eine anderes korrektes Datenfluss-Diagramm aufzeichnet, welches jedoch nicht optimal ist. In diesem Fall sind mindestens 4 Punkte zu erteilen.

# Imperatives Paradigma: Pizzabeispiel

```
procedure PizzaMargherita is

   type Tomate is record
      zustand: (frisch, geschält,
gehackt) := frisch;
   end Tomate;

   type Mozzarella is record
      zustand: (frisch, gerieben,
verkleinert) := frisch;
   end Mozzarella;

   type Ingredienzen is record
      tomaten: array 2 of Tomate;
      käse   : array 2 of Mozzarella;
      gewürz : set of (pfeffer, paprika,
basilikum);
   end Ingredienzen;

   type PizzaTeig is record
      radius : Mass := 5;
      dicke  : Mass := 4.5;
      zustand: (geknetet, ausgerollt) :=
geknetet;
   end PizzaTeig;

   type Pizza is record
      teig    : PizzaTeig;
      beigaben: Ingredienzen;
      zustand : (unbearbeitet, belegt,
gebacken) := unbearbeitet;
   end Pizza;

   procedure Vorbereitung (zutaten: in
out Ingredienzen) is
   begin
      for t in zutaten.tomaten'range loop
         -- Tomate schälen --
         zutaten.tomaten[t].zustand :=
geschält;
         -- Tomate hacken --
         zutaten.tomaten[t].zustand :=
gehackt;
      end loop;
      for k in zutaten.käse'range loop
         -- Käse verkleinern --
         zutaten.käse[k].zustand :=
verkleinert;
      end loop;
      zutaten.gewürz := {pfeffer,
paprika, basilikum};
   end Vorbereitung;

   procedure TeigAusrollen (radius,
dicke: in Mass;
                            teig : in out
PizzaTeig) is
   begin
      while (teig.radius < radius) and
(teig.dicke > dicke) loop
         -- Teig ausrollen --
         teig.radius := teig.radius + 2;
         teig.dicke := teig.dicke/1.5;
      end loop;
      teig.zustand := ausgerollt;
   end TeigAusrollen;
```

```
procedure TeigBelegen (zutaten:
   in Ingredienzen;
                          teig:       in
PizzaTeig;
                          margherita: in
out Pizza) is
   begin
      margherita.teig := teig;
      for t in zutaten.tomaten'range loop
            -- Teig mit gehackten Tomaten
belegen --

   margherita.beigaben.tomaten[t] :=
zutaten.tomaten[t];
      end loop;
      for k in zutaten.käse'range loop
            -- Teig mit verkleinertem
Käse belegen --
            margherita.beigaben.käse[k]
:= zutaten.käse[k];
      end loop;
      -- Pizza würzen --
      margherita.beigaben.gewürz :=
zutaten.gewürz;
      margherita.zustand := belegt;
   end TeigBelegen;

   procedure PizzaBacken (temperatur,
dauer: in Mass;
                              margherita:
in out Pizza) is
      zeit: Mass;
   begin
      zeit := 0;
      while zeit < dauer loop
         -- Pizza bei einer Temperatur
von temperatur Grad backen --
         zeit := zeit + 1;
      end loop;
      margherita.zustand := gebacken;
   end PizzaBacken;

   teig       : PizzaTeig;
   zutaten    : Ingredienzen;
   margherita : Pizza;

begin
   Vorbereitung (zutaten);
   TeigAusrollen (15, 0.5, teig);
   TeigBelegen (zutaten, teig,
margherita);
   PizzaBacken (250, 20, margherita);
end PizzaMargherita;
```

# Objektorientiertes Paradigma: Pizzabeispiel

```
classname                 Zugaben
superclass                -
instanceVariableNames     zustand
instanceMethods
   Zustand: anfangszustand
      zustand:=anfangszustand.

classname                 Tomate
superclass                Zugaben
instanceVariableNames     -
instanceMethods
   Schälen
      zustand:=geschält.
   Hacken
      zustand:=gehackt.

classname                 Mozzarella
superclass                Zugaben
instanceVariableNames     -
instanceMethods
   Verkleinern
      zustand:=verkleinert.

classname                 Ingredienzen
superclass                -
instanceVariableNames     tomaten käse
gewürz
instanceMethods
   Würzen: gewürztMit
      gewürz:=gewürztMit.
   Einkaufen
      for i=0 to 2
         [tomaten at: i put: Tomate new.
         tomaten at: i receive Zustand:
frisch.
         käse at: i put: Mozzarella new.
         käse at: i receive Zustand:
frisch.]
   Vorbereiten
      for i=0 to 2
         [tomaten at: i receive: Schälen.
         tomaten at: i receive: Hacken.
         käse at: i receive:
Verkleinern.]

classname                 PizzaTeig
superclass                -
instanceVariableNames     radius dicke
zustand
instanceMethods
   Einkaufen
      radius:=5.
      dicke:=4.5.
      zustand:=geknetet.
   AusrollenAufRadius: sollradius
UndDicke: solldicke
      while ((radius < sollradius) AND
(dicke > solldicke))
         [dicke:=dicke/1.5.
radius:=radius+2.]
      zustand:=ausgerollt.
```

```
classname                Pizza
superclass               -
instanceVariableNames    margheritaTeig
beigaben zustand backzeit
   ofentemperatur
instanceMethods
   Anfangszustand: anfangszustand.
      zustand:=anfangszustand.
   Belegen: teig Mit: ingredienzen
      margheritaTeig:=teig.
      beigaben:=ingredienzen.
      zustand:=belegt.
      ingredienzen Würzen: pfeffer.
   Minuten: dauer BackenBeiTemperatur:
temperatur
      ofentemperatur:=temperatur.
      backzeit:=0.
      while (backzeit < dauer)
         [backzeit:=backzeit+1.]
      zustand:=gebacken.

« Programm-Aufruf »
teig:=PizzaTeig new.
teig Einkaufen.
zutaten:=Ingredienzen new.
zutaten Einkaufen.
margherita:=Pizza new.
margherita Anfangszustand: unbearbeitet.
zutaten Vorbereiten.
teig AusrollenAufRadius: 15 UndDicke:
0.5.
margherita Belegen: teig Mit: zutaten.
margherita Minuten: 20
BackenBeiTemperatur: 250.
```

## Einige Fragen zum Aufwärmen.

1.1 Wie bei einer echten, so werden auch bei der digitalen Pizza die Tomaten nicht einfach so auf den Teig gelegt. Welche Zustände müssen die armen Tomaten durchmachen, bevor sie auf dem Teig landen? Notieren Sie bitte diese Zustände.

1.2 Unser Pizzaiolo braucht gewisse Zutaten für die Pizza, die er alle einkaufen muss. Stellen Sie bitte eine Einkaufsliste zusammen (Artikel und Menge).

1.3 Mozzarella und Tomaten sind eher fade. Darum wird gepfeffert. Natürlich machen wir das nicht zu Beginn, sondern erst, nachdem schon einiges vorbereitet ist. Was ist mit dem Teig angestellt worden, bis die Pizza gepfeffert wird? Notieren Sie sich bitte die Zustände, die der Teig bis zu diesem Zeitpunkt angenommen hat.

## Funktionales Paradigma: Pizzabeispiel

```
data TomatenZustand ==
frisch++geschält++gehackt;
type Tomate == TomatenZustand;

data MozzarellaZustand ==
frisch++gerieben++verkleinert;
type Mozzarella == MozzarellaZustand;

data Gewürz ==
pfeffer++paprika++basilikum;

type Ingredienzen ==
list(Tomate)#list(Mozzarella)#list(Gewürz
);

data TeigZustand ==
(geknetet,ausgerollt);
type PizzaTeig ==
Radius#Dicke#TeigZustand;

data PizzaZustand ==
unbearbeitet++belegt++gebacken;
type Pizza ==
PizzaTeig#Ingredienzen#PizzaZustand;

dec TomatenRüsten:Tomate -> Tomate;
--- TomatenRüsten(frisch) <=
TomatenRüsten(geschält);
--- TomatenRüsten(geschält) <= gehackt;

dec Verkleinern:Mozzarella -> Mozzarella;
--- Verkleinern(frisch) <= verkleinert;

dec Bearbeite:list(anytype)#(anytype ->
anytype) -> list(anytype);
--- Bearbeite([],bearbeitung) <= [];
---
Bearbeite(erstesElement::rest,bearbeitung
) <=

   bearbeitung(erstesElement)::Bearbeite(
rest,bearbeitung));

dec Würzen:list(Gewürz) -> list(Gewürz);
--- Würzen(erstesElement::rest) <=
[erstesElement];

dec Vorbereitung:Ingredienzen ->
Ingredienzen;
---
Vorbereitung(tomaten,mozzarella,gewürz)
<=

   (Bearbeite(tomaten,TomatenRüsten),

   Bearbeite(mozzarella,Verkleinern),
         Würzen(gewürz)
         );

dec TeigAusrollen:PizzaTeig -> PizzaTeig;
--- TeigAusrollen(rad,dicke,geknetet) <=
         if (rad < 15) and (dicke > 0.5)
then
            TeigAusrollen (rad + 2,dicke
/ 1.5,geknetet)
         else
            (rad,dicke,ausgerollt);

dec PizzaBelegen:Pizza -> Pizza;
---
PizzaBelegen(teig,zutaten,unbearbeitet)
<=

   (TeigAusrollen(teig),Vorbereitung(zuta
ten),belegt);
```

```
dec Backen:Pizza#BackZeit#Temperatur ->
Pizza;
--- Backen((teig,zutaten,belegt),0,t) <=
(teig,zutaten,gebacken);
---
Backen((teig,zutaten,belegt),dauer,temper
atur) <=
        Backen
((teig,zutaten,belegt),dauer -
1,temperatur);
---
Backen((teig,zutaten,unbearbeitet),d,t)
<=

   Backen(PizzaBelegen(teig,zutaten,unbea
rbeitet),d,t);


% Programm-Aufruf: %

Backen
(
   (
      (5,4.5,geknetet),
      (
         (frisch,frisch),
         (frisch,frisch),
         (pfeffer,paprika,basilikum)
      ),
      unbearbeitet
   ),
   20,
   250
)
```

## Einige Fragen zum Aufwärmen.

1.1 Wie bei einer echten, so werden auch bei der digitalen Pizza die Tomaten nicht einfach so auf den Teig gelegt. Welche Zustände müssen die armen Tomaten durchmachen, bevor sie auf dem Teig landen? Notieren Sie sich bitte diese Zustände.

1.2 Unser Pizzaiolo braucht gewisse Zutaten für die Pizza, die er alle Einkaufen muss. Stellen Sie bitte eine Einkaufsliste zusammen.

1.3 Mozzarella und Tomaten sind eher fade. Darum wird gewürzt. Natürlich machen wir das nicht zu Beginn, sondern erst, nachdem schon einiges vorbereitet ist. Was ist mit dem Teig angestellt worden, bis die Pizza gepfeffert wird? Notieren Sie sich bitte die Zustände, die der Teig bis zu diesem Zeitpunkt angenommen hat.

## Logik-Paradigma: Pizzabeispiel

```
% zustand(TomatenZustand,
MozzarellaZustand, TeigZustand,
PizzaZustand)
startZustand(zustand(frisch, frisch,
geknetet, unbearbeitet)).
endZustand(zustand(TomatenZ, MozzarellaZ,
TeigZ, gebacken)).

% lösche(in Liste, out Liste)
lösche([keineZutat|Rest], Rest).
lösche(Liste, Liste).

% arbeiten(in Zustand, out Aktion, out
Zutat, out Zustand)
arbeiten(zustand(TomatenZ, MozzarellaZ,
TeigZ, gewürzt),
        pizzaBacken,
        keineZutat,
        zustand(TomatenZ, MozzarellaZ,
TeigZ, gebacken)).

arbeiten(zustand(TomatenZ, MozzarellaZ,
TeigZ, belegt),
        pizzaWürzen,
        gewürz(verstreut),
        zustand(TomatenZ ,MozzarellaZ,
TeigZ, gewürzt)).

arbeiten(zustand(gehackt, verkleinert,
ausgerollt, unbearbeitet),
        pizzaBelegen,
        keineZutat,
        zustand(TomatenZ, MozzarellaZ,
TeigZ, belegt)).

arbeiten(zustand(TomatenZ, MozzarellaZ,
geknetet, unbearbeitet),
        teigAusrollen,
        teig(ausgerollt),
        zustand(TomatenZ, MozzarellaZ,
ausgerollt, unbearbeitet)).

arbeiten(zustand(TomatenZ, frisch, TeigZ,
unbearbeitet),
        mozzarellaReiben,
        mozzarella(gerieben),
        zustand(TomatenZ, gerieben,
TeigZ, unbearbeitet)).

arbeiten(zustand(TomatenZ, frisch, TeigZ,
unbearbeitet),
        mozzarellaVerkleinern,
        mozzarella(verkleinert),
        zustand(TomatenZ, verkleinert,
TeigZ, unbearbeitet)).

arbeiten(zustand(geschält, MozzarellaZ,
TeigZ, unbearbeitet),
        tomatenHacken,
        tomaten(gehackt),
        zustand(gehackt, MozzarellaZ,
TeigZ, unbearbeitet)).

arbeiten(zustand(frisch, MozzarellaZ,
TeigZ, unbearbeitet),
        tomatenSchälen,
        tomaten(geschält),
        zustand(geschält, MozzarellaZ,
TeigZ, unbearbeitet)).
```

```
% pizzaBacken(in Zustand, out
AktionenListe, out ZutatenListe)
pizzaBacken(Zustand,[], []):-
        endZustand(Zustand).
pizzaBacken(Zustand,
[Aktion|WeitereAktionen], Zutaten):-
        arbeiten(Zustand, Aktion, Zutat,
NeuerZustand),
        lösche([Zutat|WeitereZutaten],
Zutaten),
        pizzaBacken(NeuerZustand,
WeitereAktionen, WeitereZutaten).

% pizzaMargherita(out AktionenListe, out
ZutatenListe)
pizzaMargherita(Aktionen, Zutaten):-
        startZustand(Zustand),
        pizzaBacken(Zustand, Aktionen,
Zutaten).


% Programm-Aufruf:
?- pizzaMargherita(Arbeiten, Zutaten).
```

## Einige Fragen zum Aufwärmen.

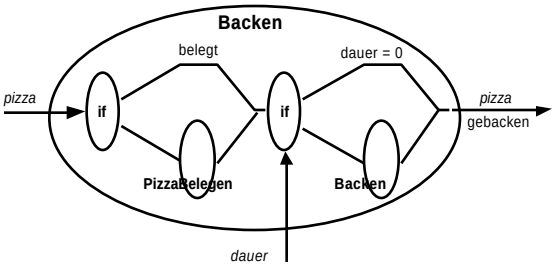
1.1 Unser Pizzaiolo braucht gewisse Zutaten für die Pizza, die er alle einkaufen muss. Stellen Sie bitte eine Einkaufsliste zusammen.

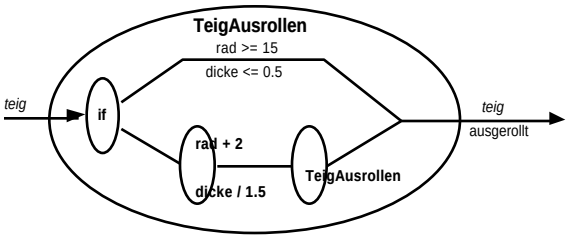
1.2 Wie bei einer echten, so werden auch bei der digitalen Pizza die Tomaten nicht einfach auf den Teig gelegt.

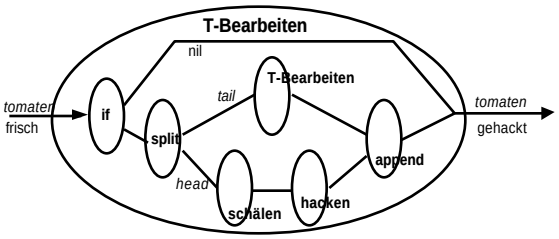
Welche Zustände müssen die armen Tomaten durchmachen, bevor sie auf dem Teig landen?

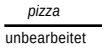
1.3 Mozzarella und Tomaten sind eher fade. Darum wird gewürzt. Natürlich macht dies der Pizzaiolo nicht zu Beginn, sondern erst, nachdem schon einiges vorbereitet ist. In welchem Zustand befindet sich also die Pizza zum Zeitpunkt des Würzens?

# Datenfluss-Paradigma: Pizzabeispiel






## Einige Fragen zum Aufwärmen

1.1 Unser Pizzaiolo braucht gewisse Zutaten für die Pizza, die er alle einkaufen muss. Stellen Sie bitte eine Einkaufsliste zusammen.

1.2 Wie bei einer echten, so werden auch bei der digitalen Pizza die Tomaten nicht einfach auf den Teig gelegt.

a) Welche Aktionen wendet das Programm auf die Tomaten an?

b) Versuchen Sie mit einem Farbstift den «Lebensweg» einer Tomate nachzuzeichen. **Tip:** Sie dürfen nur bei vier der sechs grossen Ellipsen vorbeikommen.

**Bemerkung:** Die Ingredienzen setzen sich aus Gewürz, Mozzarella und Tomaten zusammen, wobei Mozzarella und Tomaten Listen sind.