

Helmut Seidl

Informatik 2

TU München

Sommersemester 2005

Inhaltsverzeichnis

1	Graphische Benutzer-Oberflächen	4
1.1	Schon mal ein Anfang ...	7
1.2	Einfache AWT-Komponenten	10
1.3	Ereignisse	17
1.4	Schachtelung von Komponenten	23
2	Tic-Tac-Toe	31
2.1	Überblick über das Tic-Tac-Toe-System	39
2.2	Berechnung einer Strategie	39
2.3	Die Klasse Game	42
2.4	Die graphische Benutzer-Oberfläche	44
2.5	Beantwortung eines Zugs	45
2.6	Ein Dialog-Fenster	47
2.7	Effizienz	49
3	Korrektheit von Programmen	54
3.1	Verifikation von Programmen	56
3.2	Korrektheit	65
3.3	Optimierung	71
3.4	Terminierung	74
3.5	Der Hoare-Kalkül	78
3.6	Behandlung von Prozeduren	82
3.7	Prozeduren mit lokalen Variablen	89
3.8	Prozeduren mit Parametern	89
4	Grundlagen	94
4.1	Die Interpreter-Umgebung	94
4.2	Ausdrücke	94
4.3	Wert-Definitionen	95
4.4	Komplexere Datenstrukturen	96
4.5	Listen	98
4.6	Definitionen von Funktionen	99
4.7	Benutzerdefinierte Typen	102
5	Funktionen – näher betrachtet	107
5.1	Rekursionsarten	107
5.2	Funktionen höherer Ordnung	109
5.3	Funktionen als Daten	112
5.4	Einige Listen-Funktionen	113
5.5	Polymorphe Funktionen	114

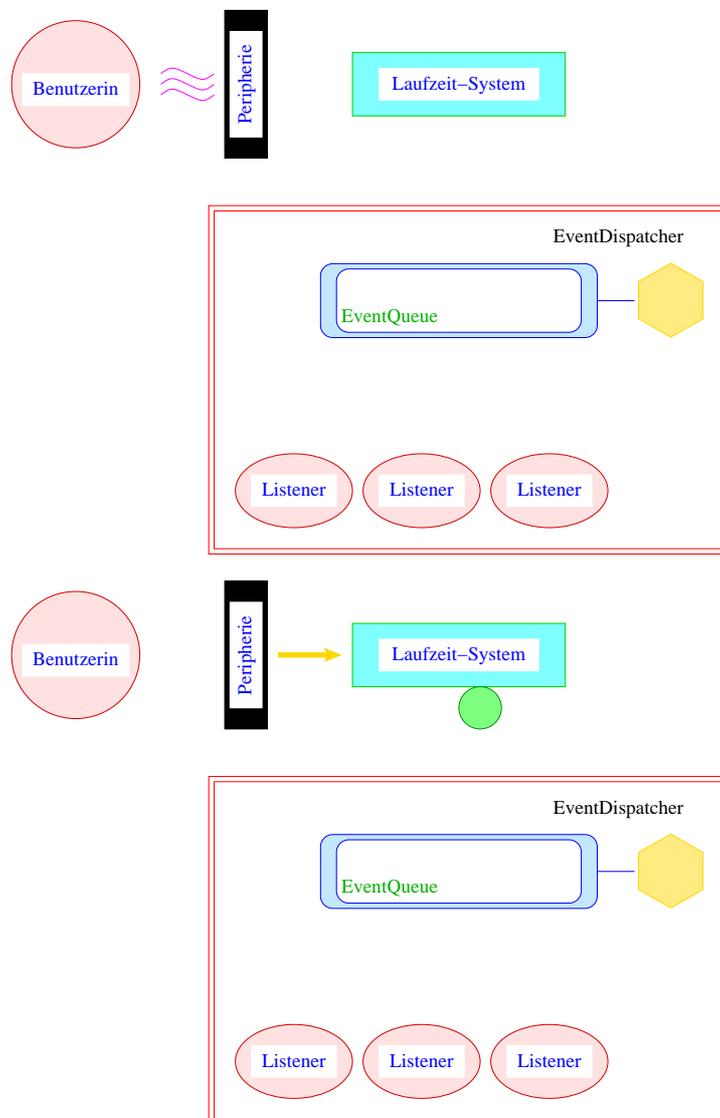
5.6	Polymorphe Datentypen	115
5.7	Anwendung: Queues	117
5.8	Namenlose Funktionen	118
5.9	Unendliche Datenstrukturen	120
6	Größere Anwendung: Balancierte Bäume	122
7	Praktische Features in Ocaml	134
7.1	Ausnahmen (Exceptions)	134
7.2	Imperative Features im Ocaml	138
8	Anwendung: Grundlegende Graph-Algorithmen	142
8.1	Gerichtete Graphen	142
8.2	Erreichbarkeit und DFS	143
8.3	Topologisches Sortieren	145
8.4	Kürzeste Wege	148
9	Das Modulsystem von OCAML	152
9.1	Module oder Strukturen	153
9.2	Modul-Typen oder Signaturen	156
9.3	Information Hiding	158
9.4	Funktoren	160
9.5	Constraints	162
9.6	Getrennte Übersetzung	164
10	Formale Methoden für Ocaml	165
10.1	MiniOcaml	165
10.2	Eine Semantik für MiniOcaml	166
10.3	Beweise für MiniOcaml-Programme	174
11	Parallele Programmierung	183
11.1	Kanäle	184
11.2	Selektive Kommunikation	191
11.3	Threads und Exceptions	196
11.4	Gepufferte Kommunikation	197
11.5	Multicasts	198
12	Datalog: Rechnen mit Relationen	202
12.1	Beantwortung von Anfragen	207
12.2	Operationen auf Relationen	210

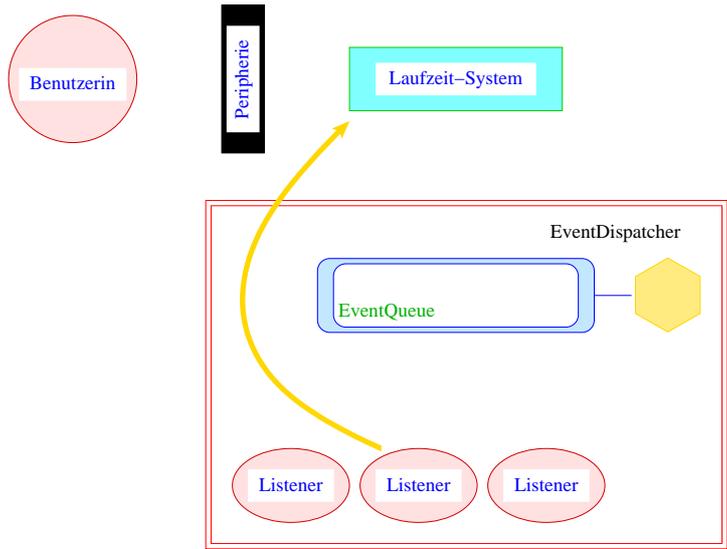
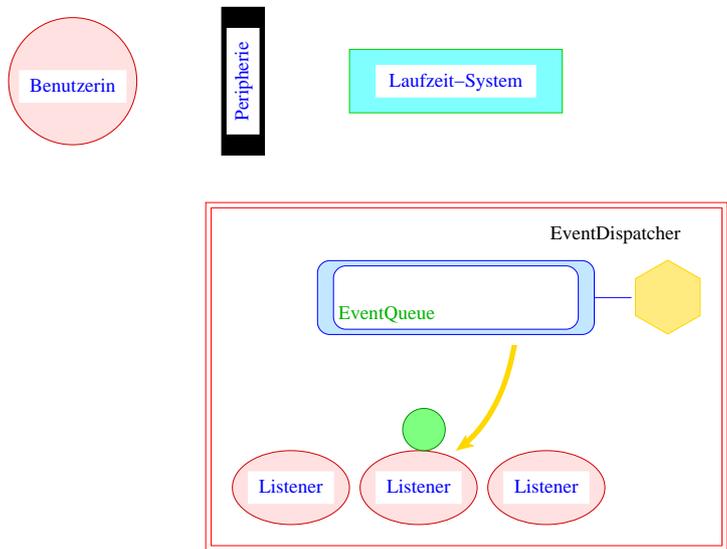
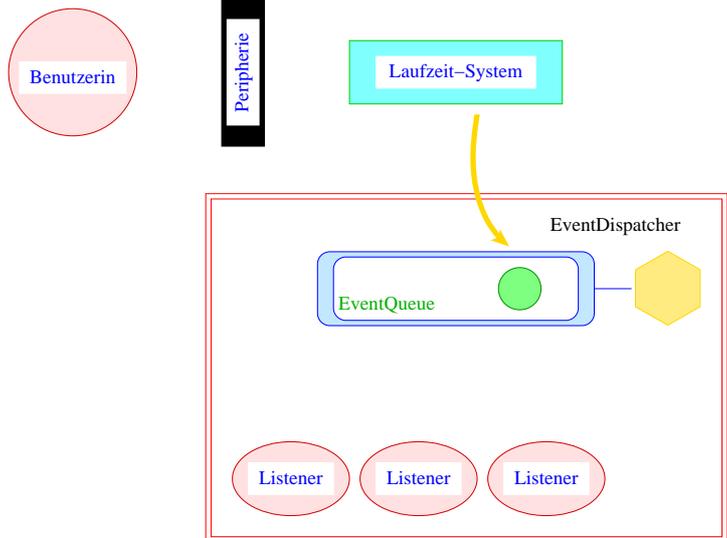
1 Graphische Benutzer-Oberflächen

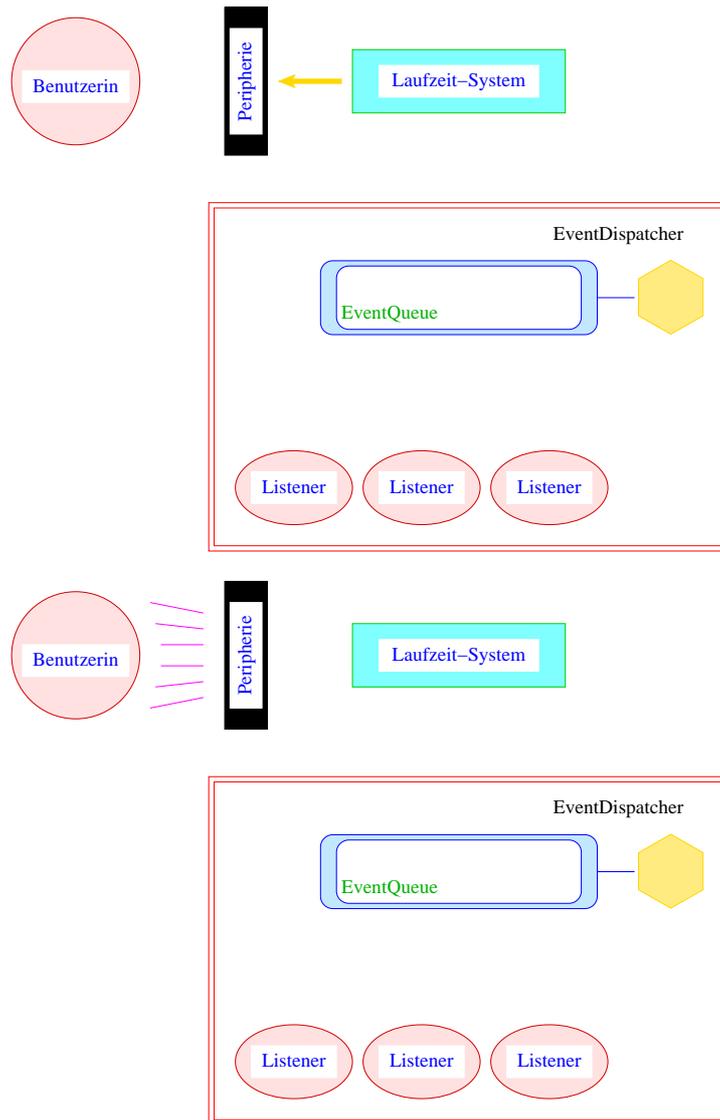
Eine graphische Benutzer-Oberfläche (GUI) ist i.a. aus mehreren Komponenten zusammen gesetzt, die einen (hoffentlich :-} intuitiven Dialog mit der Benutzerin ermöglichen sollen.

Idee:

- Einzelne Komponenten bieten der Benutzerin Aktionen an.
- Ausführen der Aktionen erzeugt **Ereignisse**.
- Ereignisse werden an die dafür zuständigen Listener-Objekte weiter gereicht
↑ **Ereignis-basiertes Programmieren.**







- Maus-Bewegungen und -Klicks, Tastatur-Eingaben etc. werden von der Peripherie registriert und an das \uparrow Betriebssystem weitergeleitet.
- Das Java-Laufzeit-System nimmt die Signale vom Betriebssystem entgegen und erzeugt dafür AWTEvent-Objekte.
- Diese Objekte werden in eine AWTEventQueue eingetragen \implies **Producer!**
- Die Ereignis-Schlange verwaltet die Ereignisse in der Reihenfolge, in der sie entstanden sind, kann aber auch mehrere ähnliche Ereignisse zusammenfassen ...
- Der AWTEvent-Dispatcher ist ein weiterer Thread, der die Ereignis-Schlange abarbeitet \implies **Consumer!**
- Abarbeiten eines Ereignisses bedeutet:
 1. Weiterleiten des AWTEvent-Objekts an das Listener-Objekt, das vorher zur Bearbeitung solcher Ereignisse **angemeldet** wurde;

2. Aufrufen einer speziellen Methode des Listener-Objekts.

- Die Objekt-Methode des Listener-Objekts hat für die Reaktion des Applets zu sorgen.



- Ereignisse können jederzeit eintreten.
- Ihre Abarbeitung erfolgt sequentiell.

1.1 Schon mal ein Anfang ...

Beispiel: Ein Knopf :-)

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class FirstButton extends Applet
    implements ActionListener {
    Label label;
    Button button;
    ...

    public void init() {
        setBackground(Color.orange);
        setFont(new Font("SansSerif", Font.BOLD, 18));
        label = new Label("This is my first button :-");
        add(label);
        button = new Button("Knopf");
        button.setBackground(Color.white);
        button.addActionListener(this);
        add(button);
    }
    ...
}
```

- Das Applet enthält zwei weitere Komponenten:
(1) ein Label; (2) einen Button.
- Objekte dieser Klassen besitzen eine Aufschrift :-)

- Wie bei Applet-Objekten kann der Hintergrund gesetzt werden.
- `public void addActionListener(ActionListener listener)`
registriert ein Objekt `listener` als dasjenige, das die von der Komponente ausgelösten `ActionEvent`-Objekte behandelt, hier: das Applet selber :-).
- `ActionListener` ist ein [Interface](#).
Zu seiner Implementierung muss die Methode
`void actionPerformed(ActionEvent e)`
bereitgestellt werden.
- Die Objekt-Methoden:
 `void add(Component c)`
 `void add(Component c, int i)`
... fügen die Komponente `c` zum Applet hinten
(bzw. an der Stelle `i`) hinzu.

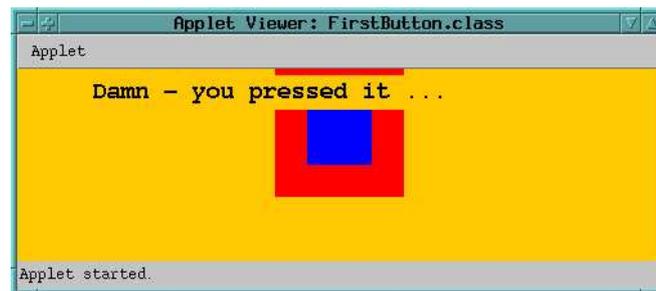
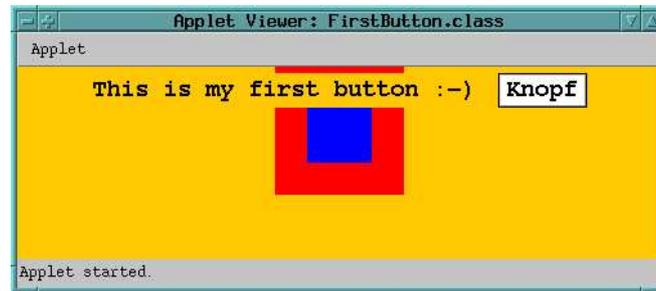
```

...
public void paint(Graphics page) {
    page.setColor(Color.red);
    page.fillRect(200,0,100,100);
    page.setColor(Color.blue);
    page.fillRect(225,25,50,50);
}
public void actionPerformed(ActionEvent e) {
    label.setText("Damn - you pressed it ...");
    System.out.println(e);
    remove(button);
}
} // end of Applet FirstButton

```

- Die Methode
`public void actionPerformed(ActionEvent e)`
setzt den Text des Labels auf einen neuen Wert und beseitigt anschließend den Knopf mithilfe der Objekt-Methode
`public void remove(Component c);`
- Beim Drücken des Knopfs passiert das Folgende:
 1. ein `ActionEvent`-Objekt `action` wird erzeugt und in die Ereignis-Schlange eingefügt.
 2. Der `AWTEvent-Dispatcher` holt `action` wieder aus der Schlange. Er identifiziert das Applet `app` selbst als das für `action` zuständige Listener-Objekt. Darum ruft er `app.actionPerformed(action);` auf.

- Wären **mehrere** Objekte als listener registriert worden, würden sukzessive auch für diese entsprechende Aufrufe abgearbeitet werden.



Beachte:

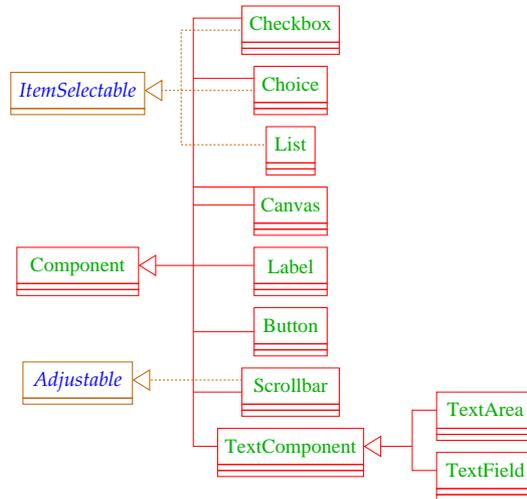
- Die in den Labels verwendete Schriftart richtet sich nach der des umgebenden Applets (zumindest in der Größe :-).
- Die Objekt-Methoden:


```
public String getText();
public void setText(String text);
```

 ... gestatten den Zugriff auf den Text eines Label- (Button- oder TextComponent) Objekts.
- Die hinzugefügten Komponenten liegen im Applet-Feld **vor** der graphischen Darstellung, die die `paint()`-Methode erzeugt.
- Jede Komponente weiss, wie sie neu gemalt werden muss ...

1.2 Einfache AWT-Komponenten

Eine Übersicht ...



Canvas eine Fläche zum Malen ...

Label zeigt eine Text-Zeile.

Button einzelner Knopf, um eine Aktion auszulösen.

Scrollbar Schieber zur Eingabe von (kleinen) int-Zahlen.

Canvas eine Fläche zum Malen ...

Label zeigt eine Text-Zeile.

Button einzelner Knopf, um eine Aktion auszulösen.

Scrollbar Schieber zur Eingabe von (kleinen) int-Zahlen.

Beispiel:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class ScalingFun extends Applet
    implements AdjustmentListener {
    private Scrollbar sHeight, sWidth;
    private Image image;
    ...
}
```

```

private int width = 600; private int height = 500;
public void init() {
    setBackground(Color.white);
    image = getImage(getDocumentBase(),"df991230.jpg");
    setLayout(new BorderLayout());
    sHeight = new Scrollbar(Scrollbar.VERTICAL,
                           500,50,0,550);
    sHeight.addAdjustmentListener(this);
    add(sHeight,"West");
    sWidth = new Scrollbar(Scrollbar.HORIZONTAL,
                           600,50,0,650);
    sWidth.addAdjustmentListener(this);
    add(sWidth,"South");
}
...

```

- Ein Scrollbar-Objekt erzeugt AdjustmentEvent-Ereignisse.
- Entsprechende Listener-Objekte müssen das Interface AdjustmentListener implementieren.
- Dieses Interface verlangt die Implementierung einer Methode
`public void adjustmentValueChanged(AdjustmentEvent e);`
- Die `init()`-Methode des Applets legt zwei Scrollbar-Objekte an, eines horizontal, eines vertikal.
Dafür gibt es in der Klasse Scrollbar die int-Konstanten HORIZONTAL und VERTICAL.
- Der Konstruktor `public Scrollbar(int dir,int init, int slide, int min, int max);`
erzeugt ein Scrollbar der Ausrichtung `dir` mit Anfangsstellung `init`, Dicke des Schiebers `slide`, minimalem Wert `min` und maximalem Wert `max`.
Aufgrund der Dicke des Schiebers ist der **wirkliche** Maximal-Wert `max - slide`.
- `void addAdjustmentListener(AdjustmentListener adj);`
registriert das AdjustmentListener-Objekt als Listener für die AdjustmentEvent-Objekte des Scrollbars.
- Das selbe Objekt kann mehrmals als Listener auftreten ;-)

Bleibt, das Geheimnis um Layout und West bzw. South zu lüften ...

- Jeder Container, in den man weitere Komponenten schachteln möchte, muss über eine Vorschrift verfügen, wie die Komponenten anzuordnen sind.

- Diese Vorschrift heißt **Layout**.

Zur Festlegung des Layouts stellt **Java** das Interface `LayoutManager` zur Verfügung sowie nützliche implementierende Klassen ...

- Eine davon ist das `BorderLayout`.
- Mithilfe der String-Argumente:

```
BorderLayout.NORTH = "North",
BorderLayout.SOUTH = "South",
BorderLayout.WEST = "West",
BorderLayout.EAST = "East" und
BorderLayout.CENTER = "Center"
```

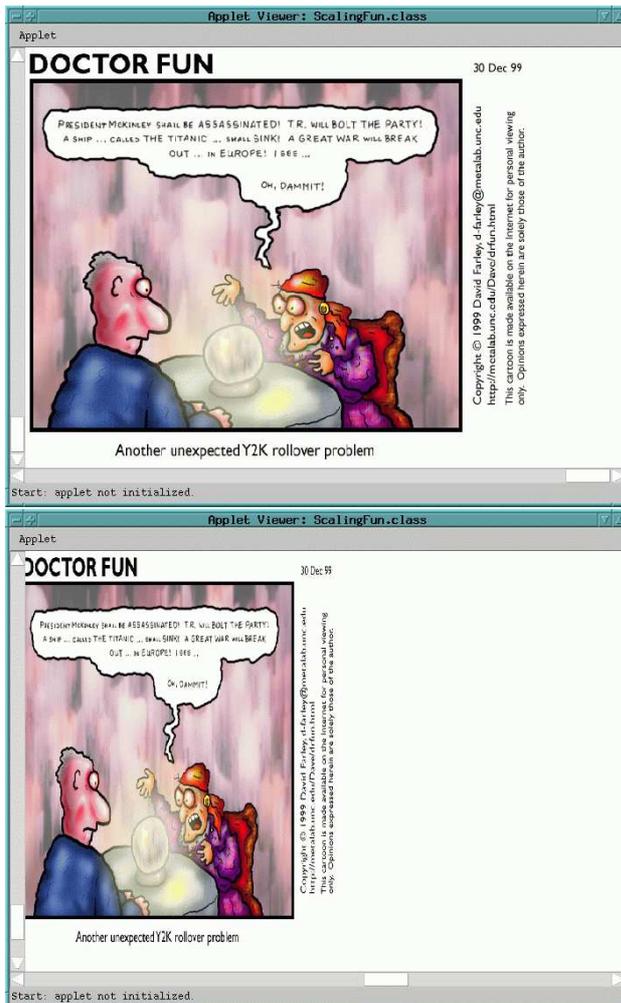
kann man **genau eine** Komponente am bezeichneten Rand bzw. der Mitte positionieren.

```
public void paint(Graphics page) {
    page.drawImage(image,0,0,width,height,this);
}
public void adjustmentValueChanged(AdjustmentEvent e) {
    Adjustable s = e.getAdjustable();
    int value = e.getValue();
    if (s == sHeight) height = value;
    else width = value;
    repaint();
}
} // end of Applet ScalingFun
```

- Um `AdjustmentEvent`-Objekte behandeln zu können, implementieren wir die Methode `AdjustmentValueChanged(AdjustmentEvent e);`
- Jedes `AdjustmentEvent`-Objekt verfügt über die Objekt-Methoden:


```
public Adjustable getAdjustable();
public int getValue();
```

 ... mit denen das auslösende Objekt sowie der eingestellte `int`-Wert abgefragt werden kann.
- Dann sieht das Applet so aus ...



Texteingabe:

TextField zeigt eine Text-Zeile, die vom Benutzer modifiziert werden kann.

TextArea zeigt mehrere modifizierbare Text-Zeilen.

Auswahl aus mehreren Alternativen:

List scrollbare Liste wählbarer Items;

Choice analog List – nur mit Anzeige des ausgewählten Items.

Checkbox kann nur die Werte true oder false annehmen. Mehrere davon können jedoch in einer CheckBoxGroup zusammengefasst werden.

Beispiel: Eine Choice-Liste ...

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class FirstChoice extends Applet
    implements ItemListener {
    private Color color = Color.white;
    private Choice colorChooser;
    ...
    public void init() {
        setFont(new Font("Serif", Font.PLAIN, 18));
        colorChooser = new Choice();
        colorChooser.setBackground(Color.white);
        colorChooser.add("white");
        colorChooser.add("red");
        colorChooser.add("green");
        colorChooser.add("blue");
        colorChooser.addItemListener(this);
        add(colorChooser);
        setBackground(Color.orange);
    }
    ...
}
```

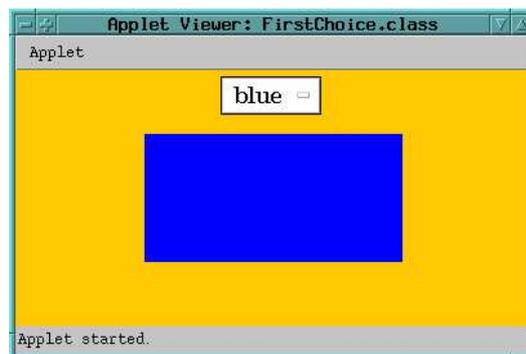
- `public Choice();` legt ein neues Choice-Objekt an;
- Zu diesem Objekt können beliebig viele Items hinzugefügt werden.
Dazu dient die ObjektMethode:
`public void add(String str);`
- `public void addItemListener(ItemListener listener);`
registriert das Objekt listener für die erzeugten ItemEvent-Objekte.
- ItemListener ist ein Interface ähnlich wie ActionListener.
- Wieder fügen wir die neue Komponente mithilfe von `void add(Component comp)` dem Applet hinzu.

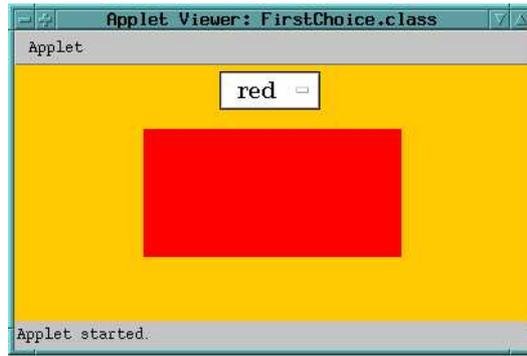
```

...
public void paint(Graphics g) {
    g.setColor(color);
    g.fillRect(100,50,200,100);
}
public void itemStateChanged(ItemEvent e) {
    String s = (String) e.getItem();
    switch(s.charAt(0)) {
        case 'w':    color = Color.white; break;
        case 'r':    color = Color.red; break;
        case 'g':    color = Color.green; break;
        case 'b':    color = Color.blue;
    }
    repaint();
}
} // end of Applet FirstChoice

```

- Das Interface `ItemListener` verlangt die Implementierung einer Methode `public void itemStateChanged(ItemEvent e);`
- Diese Methode ist für die Behandlung von `ItemEvent`-Objekten zuständig.
- `ItemEvent`-Objekte bieten (u.a. :-)) die folgenden Methoden an:
 - `public ItemSelectable getItemSelectable();`
— liefert den Selektions-Knopf;
 - `public Object getItem();`
— liefert den Text des Items.
- Dann sieht das Ganze so aus:

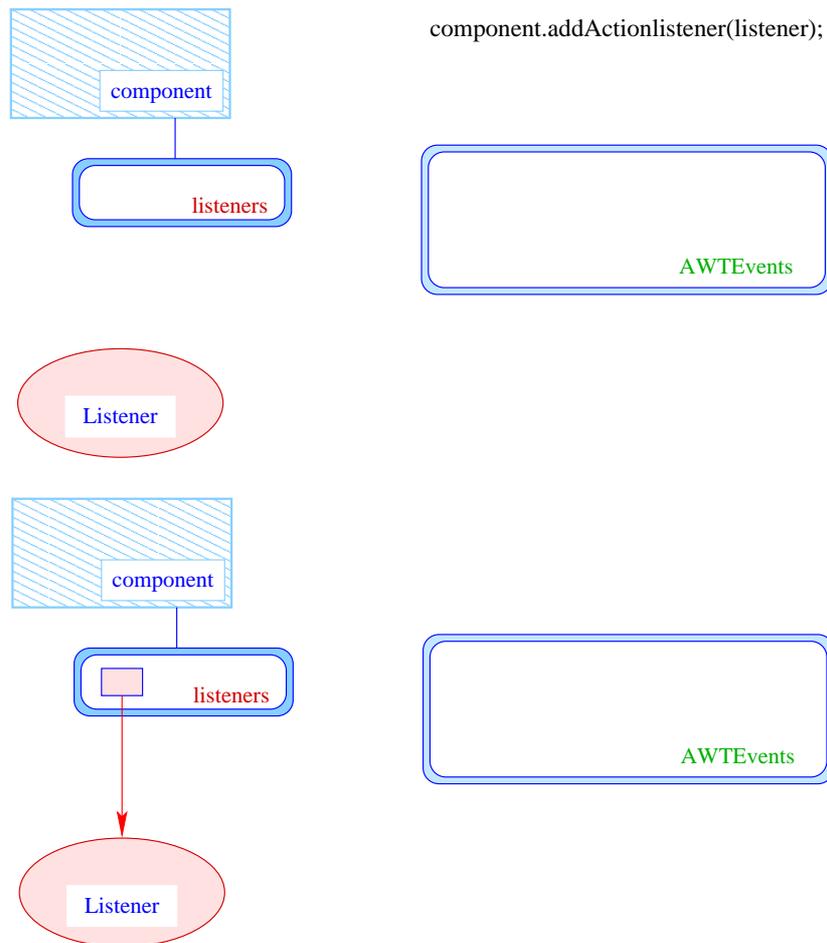


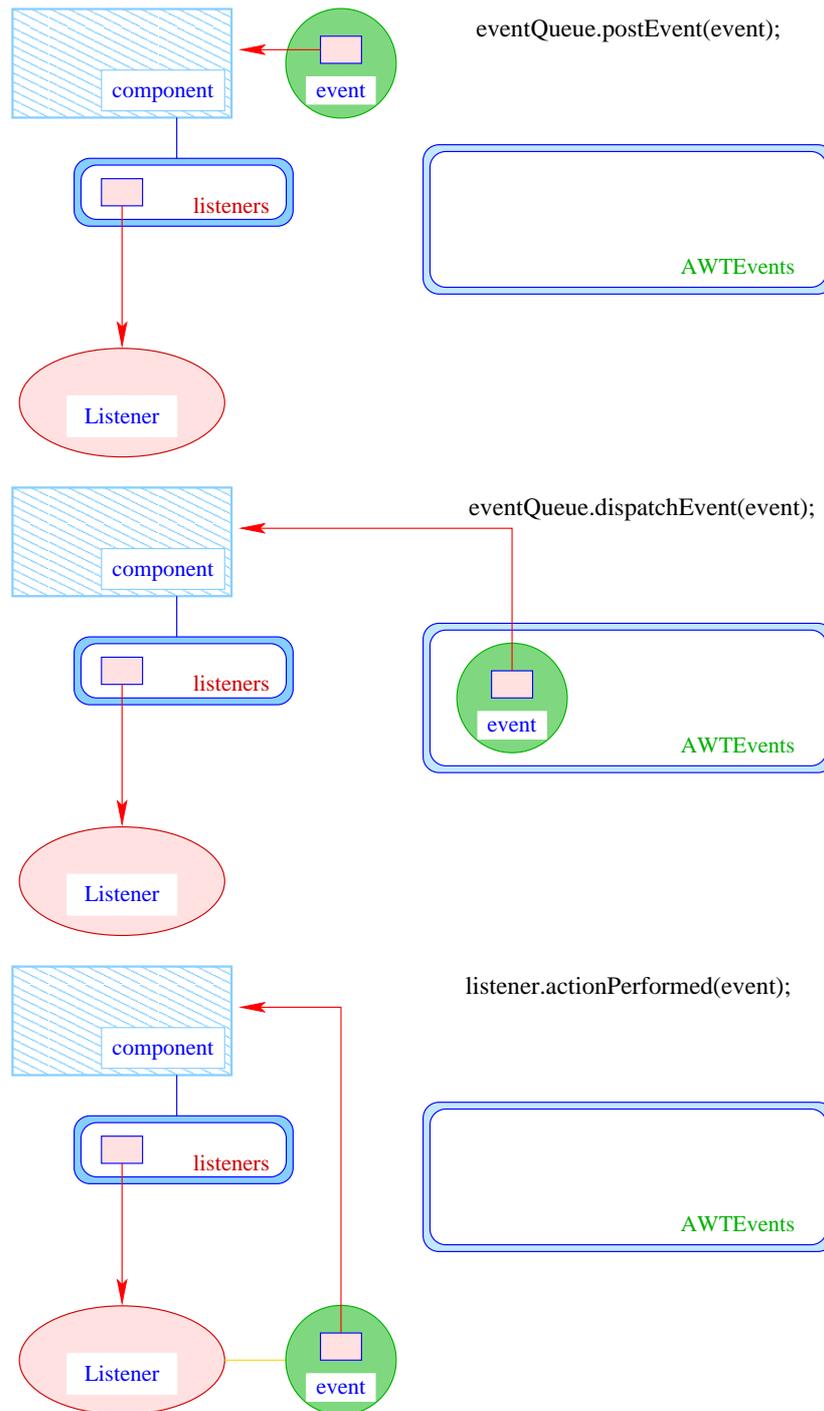


1.3 Ereignisse

- Komponenten erzeugen Ereignisse;
- Listener-Objekte werden an Komponenten für Ereignis-Klassen registriert;
- Ereignisse werden entsprechend ihrer Herkunft an Listener-Objekte weitergereicht.

Im Detail:





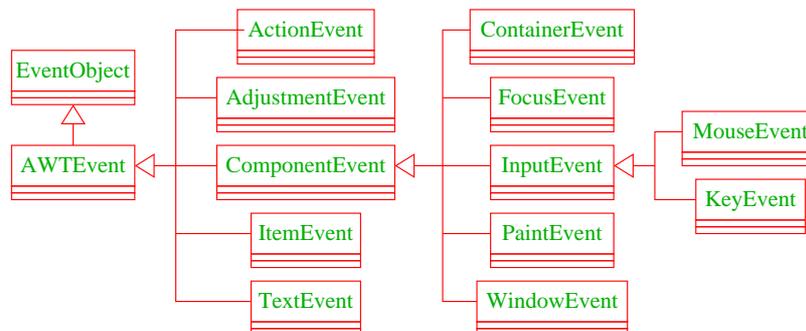
- Jedes AWTEvent-Objekt verfügt über eine **Quelle**, d.h. eine Komponente, die dieses Ereignis erzeugte.
`public Object getSource();` (der Klasse `java.util.EventObject`) liefert dieses Objekt.
- Gibt es verschiedene Klassen von Komponenten, die Ereignisse der gleichen Klasse erzeugen können, werden diese mit einem geeigneten Interface zusammengefasst.

Beispiele:

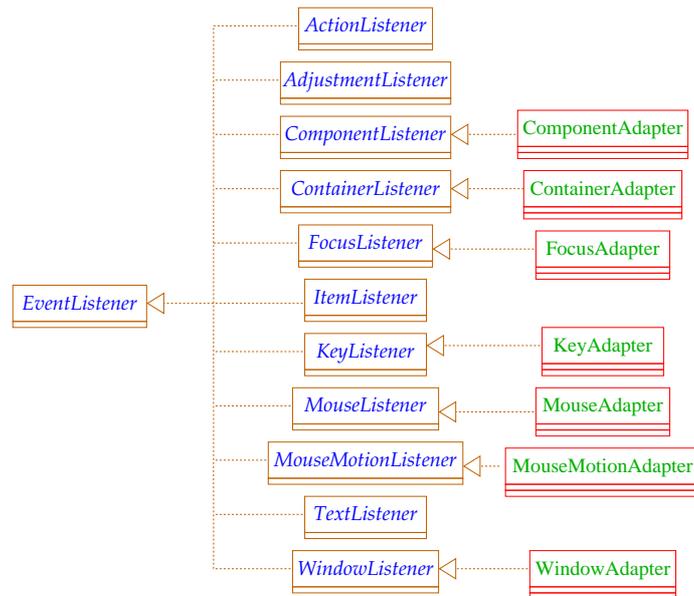
Ereignis-Klasse	Interface	Objekt-Methode
ItemEvent	ItemSelectable	ItemSelectable getItemSelectable();
AdjustmentEvent	Adjustable	Adjustable getAdjustable()

- Eine Komponente kann Ereignisse **verschiedener** AWTEvent-Klassen erzeugen.
- Für jede dieser Klassen können getrennt Listener-Objekte registriert werden ...
- Man unterscheidet zwei Sorten von Ereignissen:
 1. **semantische** Ereignis-Klassen — wie ActionEvent oder AdjustmentEvent;
 2. **low-level** Ereignis-Klassen — wie WindowEvent oder MouseEvent.

Ein Ausschnitt der Ereignis-Hierarchie ...



- Zu jeder Klasse von Ereignissen gehört ein Interface, das die zuständigen Listener-Objekte implementieren müssen.
- Manche Interfaces verlangen die Implementierung **mehrerer** Methoden.
- In diesem Fall stellt **Java Adapter**-Klassen zur Verfügung.
- Die Adapterklasse zu einem Interface implementiert sämtliche geforderten Methoden auf **triviale** Weise ;-)
- In einer Unterklasse der Adapter-Klasse kann man sich darauf beschränken, nur diejenigen Methoden zu implementieren, auf die man Wert legt.



Beispiel: ein MouseListener

- Das Interface `MouseListener` verlangt die Implementierung der Methoden:
 - `void mousePressed(MouseEvent e);`
 - `void mouseReleased(MouseEvent e);`
 - `void mouseEntered(MouseEvent e);`
 - `void mouseExited(MouseEvent e);`
 - `void mouseClicked(MouseEvent e);`
- Diese Methoden werden bei den entsprechenden Maus-Ereignissen der Komponente aufgerufen.
- Unser Beispiel-Applet soll bei jedem Maus-Klick eine kleine grüne Kreisfläche malen
...

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
class MyMouseListener extends MouseAdapter {
    private Graphics gBuff;
    private Applet app;
    public MyMouseListener(Graphics g, Applet a) {
        gBuff = g; app = a;
    }
    public void mouseClicked(MouseEvent e) {
        int x = e.getX(); int y = e.getY();
        gBuff.setColor(Color.green);
        gBuff.fillOval(x-5,y-5,10,10);
        app.repaint();
    }
} // end of class MyMouseListener

```

- Wir wollen nur die Methode `mouseClicked()` implementieren. Darum definieren wir unsere `MouseListener`-Klasse `MyMouseListener` als Unterklasse der Klasse `MouseAdapter`.
- Die `MouseEvent`-Methoden:

```
public int getX();    public int getY();
```

 liefern die Koordinaten, an denen der Mouse-Klick erfolgte ...
- an dieser Stelle malen wir einen gefüllten Kreis in den Puffer.
- Dann rufen wir für das Applet die Methode `repaint()` auf, um die Änderung sichtbar zu machen ...

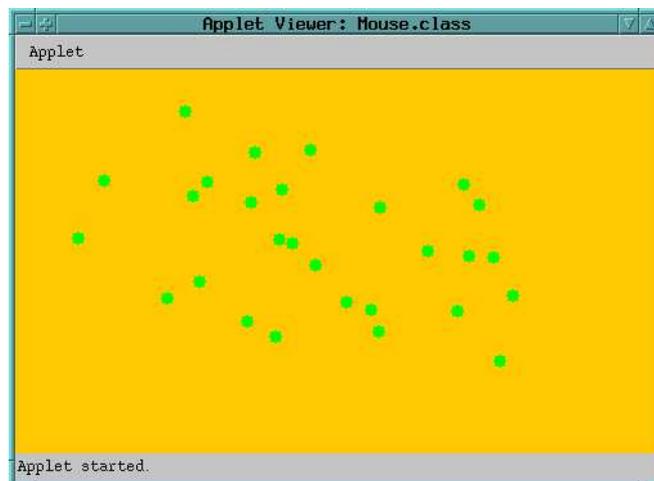
```

public class Mouse extends Applet {
    private Image buffer; private Graphics gBuff;
    public void init() {
        buffer = createImage(500,300);
        gBuff = buffer.getGraphics();
        addMouseListener(new MyMouseListener(gBuff,this));
    }
    public void start() {
        gBuff.setColor(Color.orange);
        gBuff.fillRect(0,0,500,300);
    }
    public void paint(Graphics page) {
        page.drawImage(buffer,0,0,500,300,this);
    }
} // end of class Mouse

```

- Die Methode `init()` legt den Puffer an, in dem die kleinen grünen Scheiben gemalt werden. Dann erzeugt sie ein `MouseListener`-Objekt und registriert es als `MouseListener` des Applets.
- Die Methode `start()` malt den Puffer orange.
- Die Methode `paint()` überträgt den Puffer auf die Applet-fläche.

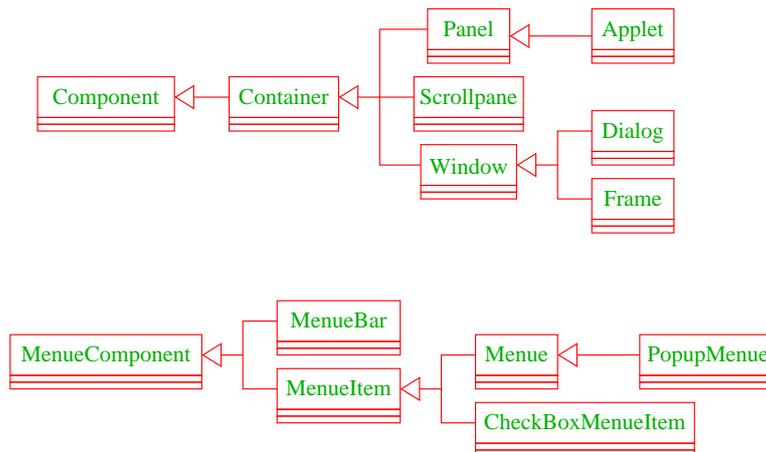
... der Effekt:



1.4 Schachtelung von Komponenten

- Komponenten, die andere Komponenten aufnehmen können, heißen **Container**.
- Der **LayoutManager** des Containers bestimmt, wie Komponenten innerhalb eines Containers angeordnet werden.

Ein Ausschnitt der Container-Hierarchie:



Container: Abstrakte Oberklasse aller Komponenten, die andere als Bestandteil enthalten können.

Panel: Konkrete Container-Klasse zum Gruppieren von Komponenten.

Applet: Unterklasse von Panel für das Internet.

Window: Ein nackter Container zur Benutzung in normalen Programmen.
Kein Rand, kein Titel, keine Menue-Leiste.

Frame: Ein Window mit Rand und Titel-Zeile. Unterstützt die Benutzung von Menues.

Dialog: Spezielles Window, das sämtlichen sonstigen Benutzer-Input blockieren kann, bis das Dialog-Fenster geschlossen wurde.

Beispiel: Das Basic Applet als Frame

- Statt der Klasse Applet benutzen wir die (Ober-)Klasse Panel.

Der Grund:

Applet ist eine Unterklasse – allerdings mit zusätzlichen Multimedia-Features, über die Panel nicht verfügt – wie z.B. Bilder aus dem internet zu laden.

Indem wir nur Panel-Methoden zulassen, garantieren wir, dass die Extra-Features nicht benutzt werden.

Da wir nur auf eine Fläche malen wollen, würde (hier) auch ein Canvas-Objekt reichen.

- Das Panel-Objekt passen wir in einen Frame ein.
- Ein Frame-Objekt ist normalerweise **unsichtbar**.

Um es sichtbar zu machen,

rufen wir `public void setVisible(boolean b)` mit `b==true` auf.

```
import java.awt.*;
class BasicPanel extends Panel {
    public BasicPanel() {
        setBackground(Color.orange);
    }
    public void paint(Graphics g) {
        g.setColor(Color.red);
        g.fillRect(50,50,200,100);
        g.setColor(Color.blue);
        g.fillRect(100,100,200,100);
        g.setColor(Color.green);
        g.fillRect(150,150,200,100);
    }
} // end of class BasicPanel
...
```

- Was in den Methoden `init()` bzw. `start()` passierte, erfolgt nun in den Konstruktoren des Panel-Objekts ...
- Der Methode `destroy()` entspricht die Methode `public void finalize();` die aufgerufen wird, wenn das Objekt freigegeben wird (deren Existenz wir bisher verschwiegen haben :-).
- Die `paint()`-Methode entspricht derjenigen des Applets und wird entsprechend automatisch aufgerufen, wenn die Fläche neu bemalt werden soll.

```

...
public class Basic extends Frame {
    public Basic(int x, int y) {
        setLocation(x,y);
        setSize(500,300);
        add(new BasicPanel());
    }
    public static void main(String[] args) {
        (new Basic(50,50)).setVisible(true);
        (new Basic(600,600)).setVisible(true);
    }
} // end of class Basic

```

- Mithilfe der Objekt-Methoden


```

void setLocation(int x, int y);
void setSize(int width, int height);

```

 ... kann ein Fenster positioniert bzw. dimensioniert werden.
- Der Standard-LayoutManager der Klasse Frame ist BorderLayout.
Für diesen fügt `void add(Component c)` die Komponente `c` in der Mitte ein (sind die Ränder unbesetzt, breitet sich die Mitte über die ganze Fläche aus :-)
- Die Klassen-Methode `main()` legt zwei Basic-Objekte an verschiedenen Stellen des Bildschirms an ...

Achtung: Um auf Schließen des Fensters adequat reagieren zu können, empfiehlt es sich, einen `WindowListener` für das Frame zu implementieren !!!

Mögliche Anordnungen von Komponenten:

Frage: was passiert, wenn das Fenster (evt.) redimensioniert wird?

Frage: Wie kann man erreichen, dass das Erscheinungsbild exakt unserer Vorstellung entspricht???

Flexible Proportionierung mithilfe eines LayoutManagers:

- Das Layout der Komponenten passt sich der Größe der zur Verfügung stehenden Fläche an :-)
- Knöpfe vergrößern sich unförmig oder wandern von einer Zeile in die nächste :-)
- Die eigene Vorstellung muss (evt. relativ mühsam) mithilfe vorgefertigter Manager realisiert werden :-{

Absolute Positionierung und Dimensionierung:

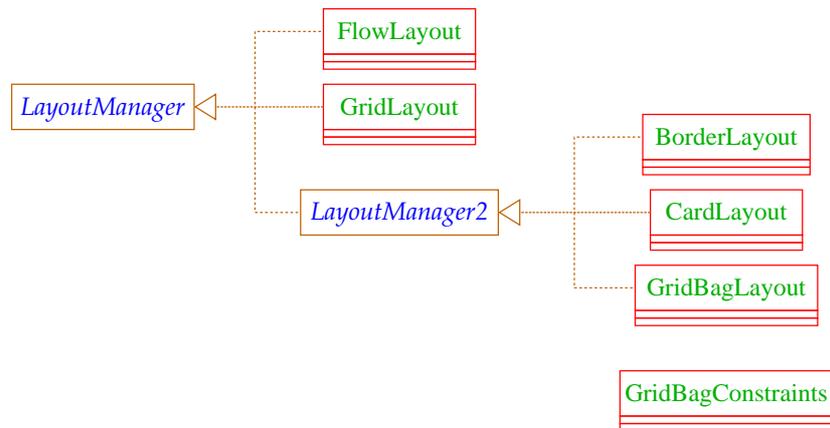
- ... kennen wir bereits für Bildschirm-Fenster und graphische Elemente einer Fläche.

- Die belegte Fläche erhält bei Fenster-Deformierung einen unbenutzten Rand oder wird (teilweise) unsichtbar :-)
- Um sie für Komponenten in Container-Objekten zu ermöglichen, kann man mittels `setLayout(null);` den aktuellen Layout-Manager **ausschalten** :-)
- Sukzessive Aufrufe von `void add(Component c);` erzeugen einen **Stack** von Komponenten, deren Größe und Position mittels

```
public void setSize(int width, int height);
public void setLocation(int x, int y);
```

... modifiziert werden kann :-}

Einige vorgefertigte Layout-Manager:



FlowLayout: Das Default-Layout der Klasse `Panel`. Komponenten werden von links nach rechts zeilenweise abgelegt; passt eine Komponente nicht mehr in eine Zeile, rückt sie in die nächste.

BorderLayout: Das Default-Layout der Klasse `Window`. Die Fläche wird in die fünf Regionen `North`, `South`, `West`, `East` und `Center` aufgeteilt, die jeweils von einer Komponente eingenommen werden können.

CardLayout: Die Komponenten werden wie in einem Karten-Stapel abgelegt. Der Stapel ermöglicht sowohl den Durchgang in einer festen Reihenfolge wie den Zugriff auf spezielle Elemente.

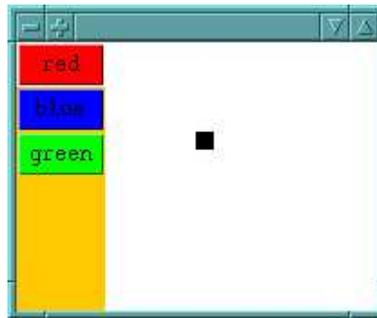
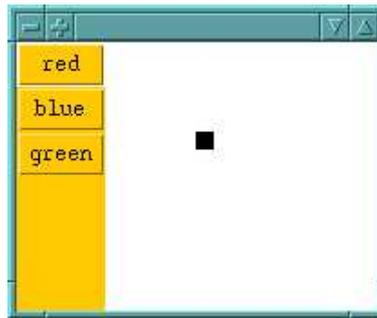
GridLayout: Die Komponenten werden in einem Gitter mit gegebener Zeilen- und Spalten-Anzahl abgelegt.

GridBagLayout: Wie `GridLayout`, nur flexibler, indem einzelne Komponenten auch mehrere Felder des Gitters belegen können.

Beispiel:

Ziel:

- Eine orange Knopfleiste am linken Rand;
- drei Knöpfe in der oberen Hälfte der Leiste;
- auf Knopf-Druck soll die Farbe des Knopfs wechseln;
- links daneben eine weiße Fläche mit einem schwarzen Quadrat ...



Implementierung:

- Die Knöpfe legen wir mittels GridLayout in ein Panel-Objekt.
- Die weiße Fläche mit schwarzem Quadrat malen wir auf ein Canvas-Objekt.
- Ein Frame-Objekt besitzt bereits das BorderLayout.
- Das Panel-Objekt legen wir im Westen des Frames ab, das Canvas-Objekt in der Mitte.

```
import java.awt.*;
import java.awt.event.*;
class MyPanel extends Panel implements ActionListener {
    Button b0, b1, b2;
    public MyPanel() {
        setBackground(Color.orange);
        b0 = new Button("red");
        b1 = new Button("blue");
        b2 = new Button("green");
        b0.addActionListener(this);
        b1.addActionListener(this);
        b2.addActionListener(this);
        setLayout(new GridLayout(6,1));
        add(b0); add(b1); add(b2);
    }
    ...
}
```

- Der Konstruktor `public GridLayout(int row, int col);` teilt die zur Verfügung stehende Fläche in ein Raster von gleich großen Feldern ein, die in `row` vielen Zeilen und `col` vielen Spalten angeordnet sind.
- die Felder werden sukzessive von links oben nach rechts unten aufgefüllt.
- Nicht alle Felder müssen tatsächlich belegt werden.
Im Beispiel bleibt die Hälfte frei ...
- Gemeinsamer ActionListener für alle drei Knöpfe ist (hier) das Panel selbst ...

```

public void actionPerformed(ActionEvent e) {
    Button b = (Button) e.getSource();
    if (b.getBackground() == Color.orange) {
        if (b == b0) b0.setBackground(Color.red);
        else if (b == b1) b1.setBackground(Color.blue);
        else b2.setBackground(Color.green);
    } else
        b.setBackground(Color.orange);
}
} // end class MyPanel
...

```

- Der Aufruf `e.getSource();` liefert das **Objekt**, das das `ActionEvent`-Objekt erzeugte, hier ein `Button`-Objekt.
- Falls die Hintergrunds-Farbe `orange` ist, modifizieren wir die Farbe. Ansonsten setzen wir sie auf `orange` zurück.

```

public class Nest extends Frame {
    public Nest() {
        setSize(200,150); setLocation(500,500);
        add(new MyPanel(), "West");
        add(new MyCanvas(), "Center");
    }
    public static void main(String[] args) {
        (new Nest()).setVisible(true); }
} // end of class Nest
class MyCanvas extends Canvas {
    public MyCanvas() { setBackground(Color.white);}
    public void paint(Graphics page) {
        page.setColor(Color.black);
        page.fillRect(50,50,10,10);
    }
} // end of class MyCanvas

```

- Ein neues Canvas-Objekt besitzt eigentlich Breite und Höhe 0 — und ist damit **unsichtbar!**
- Im BorderLayout wird eine Komponente jedoch auf die gesamte zur Verfügung stehende Fläche ausgedehnt.
- Wie in der Klasse Applet wird zum (Neu-)Malen der Canvas-Fläche implizit die Objekt-Methode `public void paint(Graphics page);` aufgerufen.

Was es sonst noch so in Java gibt:

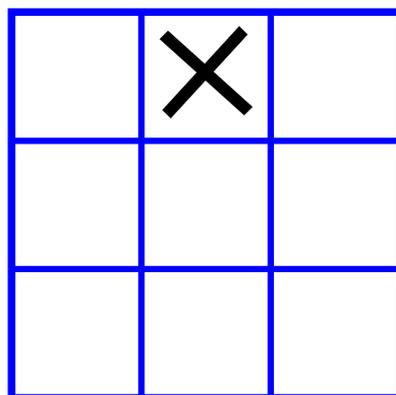
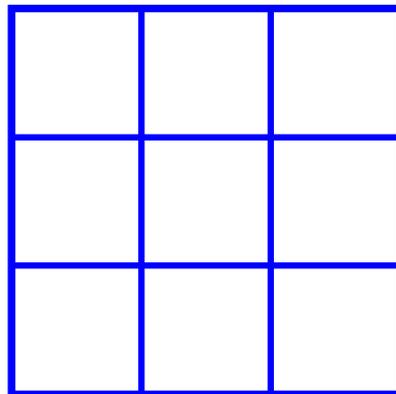
... an nützlichem:	innere Klassen;
... an mysteriösem:	Klassen zur Selbst-Reflektion;
... an komfortablem:	Malen mit Swing;
... an technischem:	Networking mit Sockets, RMI, Jini, Corba, ...
... an sonstigem:	... :-)

2 Tic-Tac-Toe

Regeln:

- Zwei Personen setzen abwechselnd **Steine** auf ein (3×3) -Spielfeld.
- Wer zuerst drei Steine in einer **Reihe** erreicht, gewinnt.
- Zeilen, Spalten und Haupt-Diagonalen sind Reihen.

Beispiel:

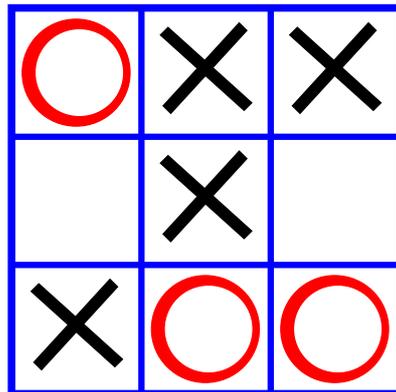
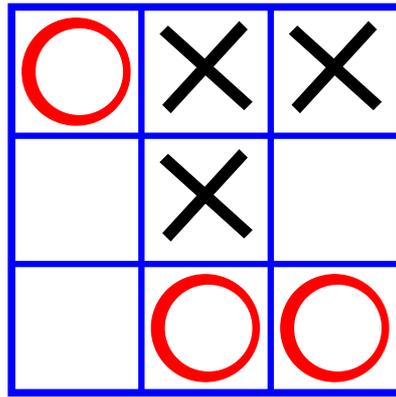


	X	
		O

	X	X
		O

O	X	X
		O

O	X	X
	X	
		O



... offenbar hat die anziehende Partei gewonnen.

Fragen:

- Ist das immer so? D.h. kann die anziehende Partei immer gewinnen?
- Wie implementiert man ein **Tic-Tac-Toe**-Programm, das
 - ... möglichst oft gewinnt?
 - ... eine **ansprechende** Oberfläche bietet?

Hintergrund:

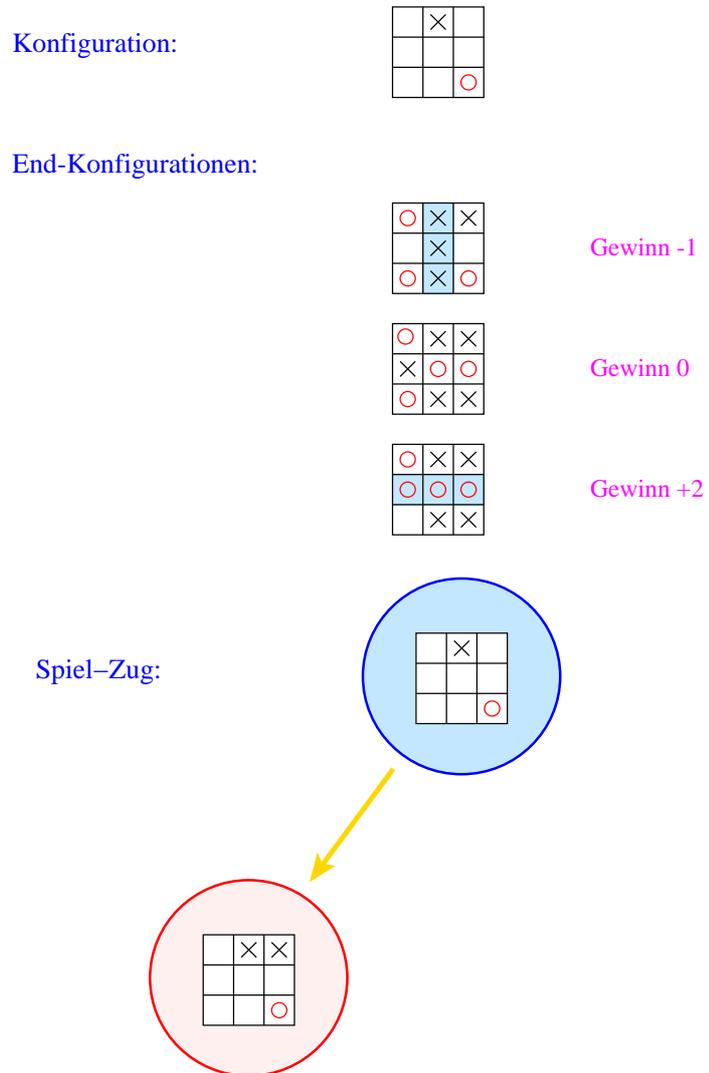
Tic-Tac-Toe ist ein endliches **Zwei-Personen-Nullsummen-Spiel**.

Das heißt:

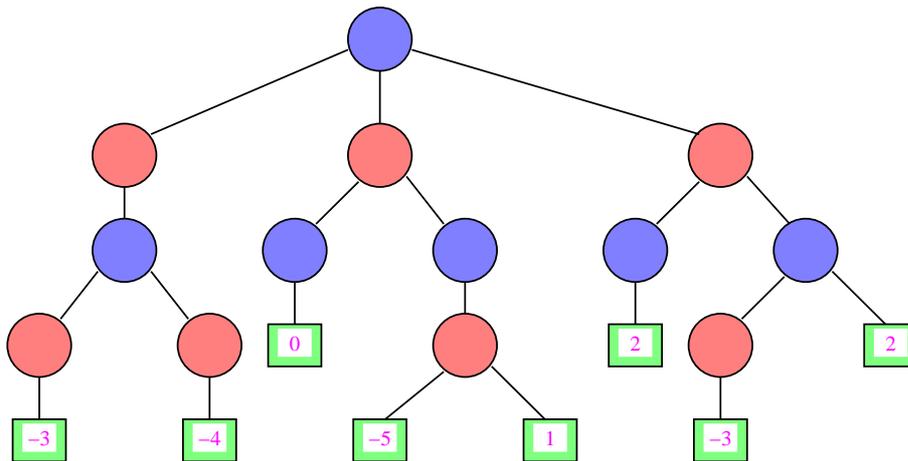
- Zwei Personen spielen gegeneinander.
- Was der eine gewinnt, verliert der andere.
- Es gibt eine endliche Menge von **Spiel-Konfigurationen**.
- Die Spieler ziehen abwechselnd.
Ein **Zug** wechselt die Konfiguration, bis eine **End-Konfiguration** erreicht ist.
- Jede End-Konfiguration ist mit einem **Gewinn** aus \mathbb{R} bewertet.

- Person 0 hat gewonnen, wenn eine End-Konfiguration erreicht wird, deren Gewinn negativ ist, Person 1, wenn er positiv ist.

... im Beispiel:



Alle möglichen Spiel-Verläufe lassen sich als ein Baum darstellen:



Knoten des Spielbaums	==	Konfigurationen
Kanten	==	Spiel-Züge
Blätter	==	End-Konfigurationen

Frage:

Wie finden wir (z.B. als **blaue** Person) uns im Spiel-Baum zurecht?

Was müssen wir tun, um **sicher** ein negatives Blatt zu erreichen?

Idee:

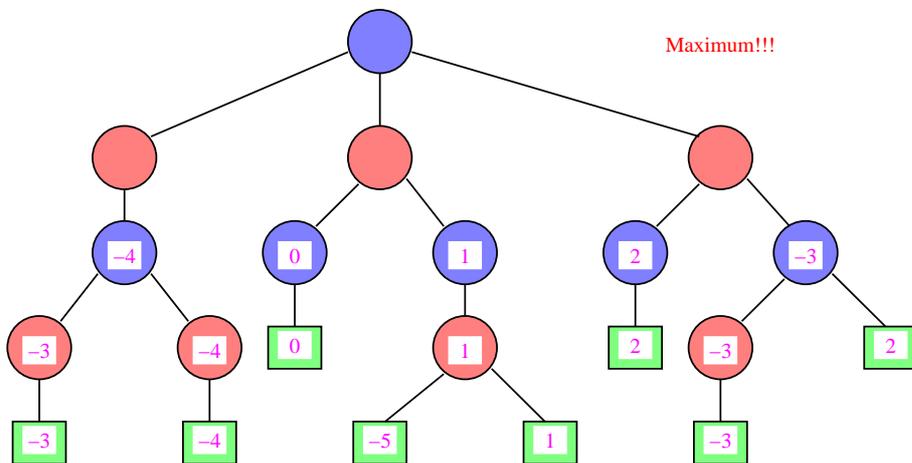
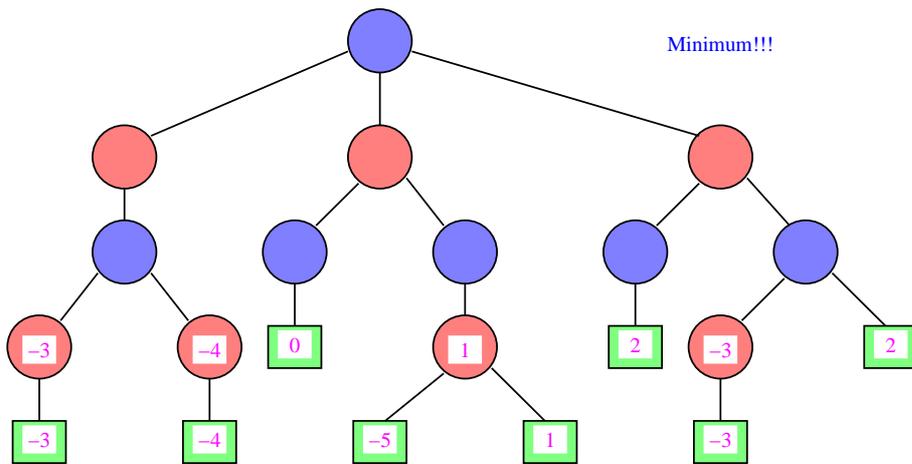
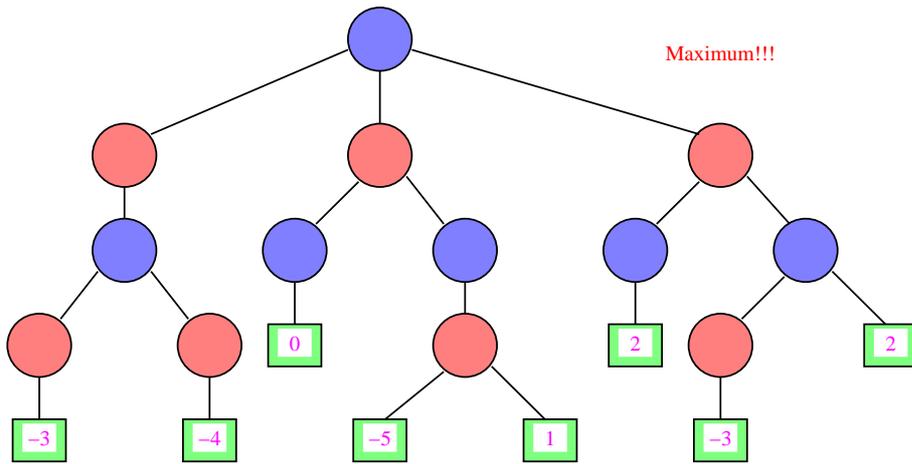
- Wir ermitteln für jede Konfiguration den jeweils **besten** zu erzielenden Gewinn.
- Seien die Gewinne für sämtliche Nachfolger einer Konfiguration bereits ermittelt.

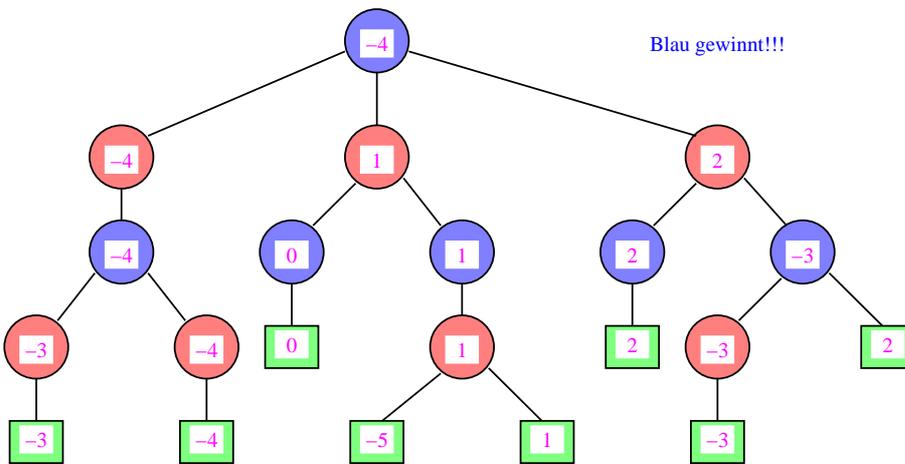
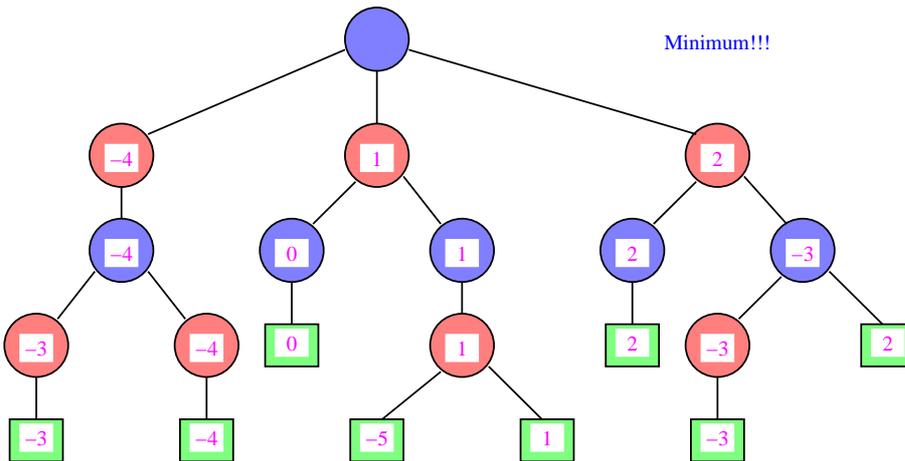
Fall 1: Die Konfiguration ist **blau**: wir sind am Zug.

Wir können garantiert das **Minimum** der Gewinne der Söhne erzielen.

Fall 2: Die Konfiguration ist **rot**: der Gegner ist am Zug.

Er kann garantiert das **Maximum** der Gewinne der Söhne erzielen.

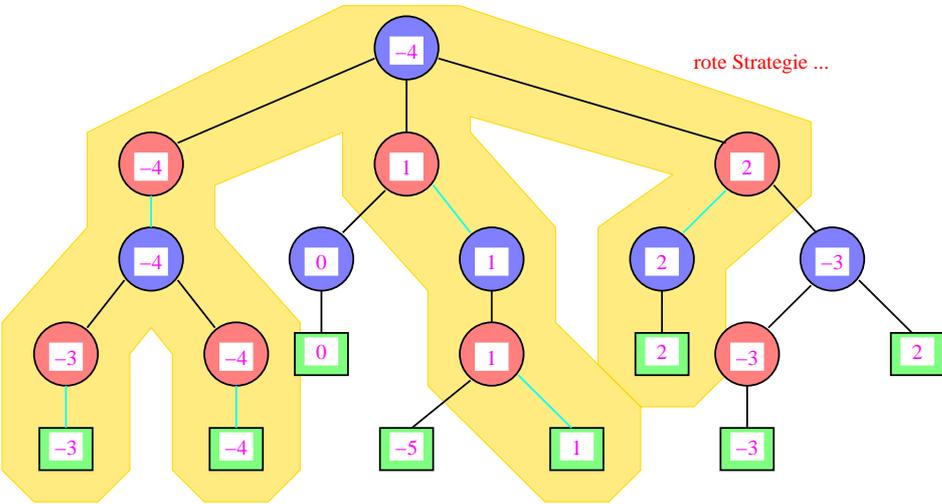
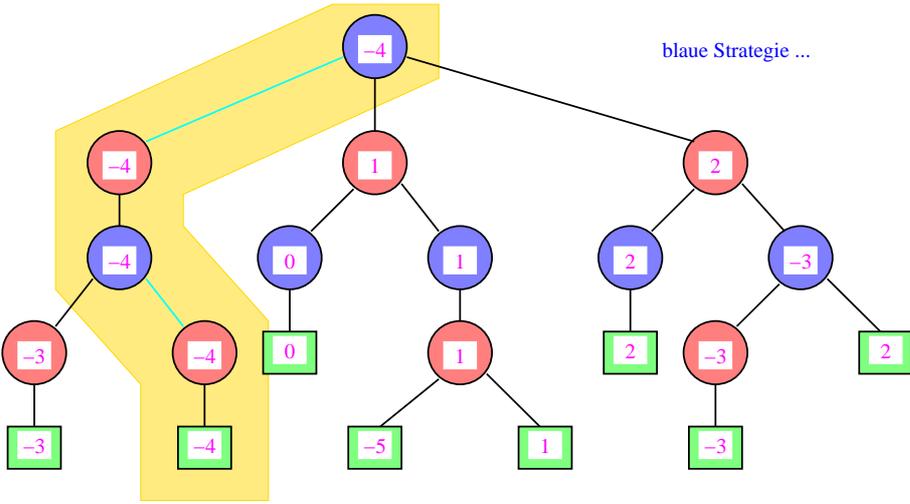
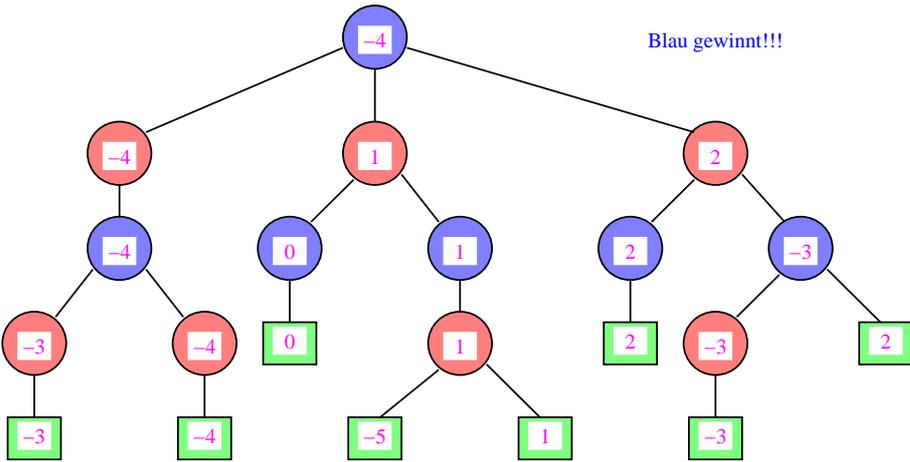




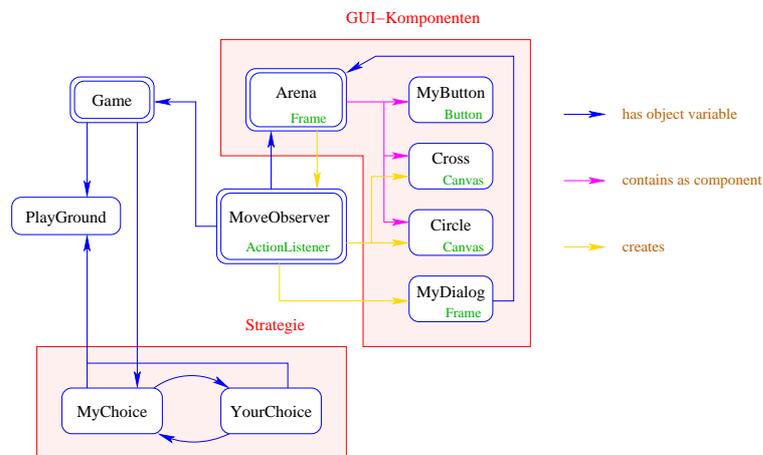
Eine **Strategie** ist eine Vorschrift, die uns in jeder (erreichbaren) Konfiguration mitteilt, welchen Nachfolger wir auswählen sollen.

Eine **optimale** Strategie ist eine, deren Anwendung garantiert zu einer End-Konfiguration führt, deren Wert mindestens so groß ist wie der berechnete garantierte Gewinn.

Eine **akzeptable** Strategie ist eine, deren Anwendung einen Verlust des Spiels verhindert, wann immer das möglich ist ...



2.1 Überblick über das Tic-Tac-Toe-System



2.2 Berechnung einer Strategie

- Die Knoten des Spiel-Baums sind aus den Klassen `YourChoice` und `MyChoice`.
- `MyChoice` implementiert Knoten, in denen das Programm zieht.
- `YourChoice` implementiert Knoten der Spielerin.
- Jeder Knoten enthält:
 - das aktuelle Spielbrett `ground`,
 - einen garantierten Gewinn `value`
 - sowie (Verweise auf) die Nachfolger-Knoten.
- `MyChoice`-Knoten enthalten zusätzlich den empfohlenen Zug `acceptableChoice`.

```

public class YourChoice implements PlayConstants {
    private PlayGround ground; private int value;
    private MyChoice[] answer;
    public YourChoice(PlayGround g, int place) {
        ground = new PlayGround(g,place,ME);
        answer = new MyChoice[9];
        value = ME;
        PossibleMoves moves = new PossibleMoves(ground);
        for(int choice = moves.next(); choice!=-1; choice = moves.next()) {
            if (ground.won(choice,YOU)) { value = YOU; continue;}
            answer[choice] = new MyChoice(ground,choice);
            int win = answer[choice].value();
            if (win < value) value = win;
        }
    }
    ...
}

```

- Das Interface `PlayConstants` fasst durchgängig benutzte Konstanten zusammen.
 - `YOU < NONE < ME` repräsentieren die drei möglichen Ausgänge eines Spiels bzw. Belegungen eines Felds des Spielbretts (-1, 0 und 2).
 - Die Klasse `PlayGround` repräsentiert Spielbretter.
 - Das aktuelle Spielbrett wird aus demjenigen des Vater-Knotens und dem von `ME` gewählten Zug gestimmt.
 - `MyChoice answers[choice]` enthält den Teilbaum für Zug `choice`.
 - Das Objekt `PossibleMoves moves` enumeriert alle aktuell freien Positionen für Steine (mindestens eine :-)
- Gewinnt der Zug `choice`, wird `value = YOU` gesetzt. Andernfalls wird `value` auf das Minimum des Werts des Unterbaums `answers[choice]` und des alten Werts gesetzt.

```

...
    public int value() { return value;}
    public MyChoice answer(int place) {
        return answer[place];
    }
} // end of class YourChoice

```

- `public int value();` liefert den Spielwert des aktuellen Spielbaums.
- `public MyChoice answer(int place);` liefert den Spiel-Teilbaum für den Zug `place`.

```

public class MyChoice implements PlayConstants {
    private PlayGround ground;
    private YourChoice yours;
    private int acceptableChoice, value;
    public MyChoice(PlayGround g, int place) {
        ground = new PlayGround(g,place,YOU);
        if (ground.won(place,YOU)) { value = YOU; return; }
        acceptableChoice = ground.force(ME);
        if (acceptableChoice != -1) { value = ME; return; }
        acceptableChoice = ground.force(YOU);
        if (acceptableChoice != -1) {
            yours = new YourChoice(ground,acceptableChoice);
            value = yours.value(); return;
        }
        ...
    }
}

```

- Die Objekt-Methode `int force(int who);` liefert einen Zug, mit dem `who` unmittelbar gewinnt – sofern ein solcher existiert :-), andernfalls -1.
- Der Aufruf `ground.force(ME)` ermittelt einen solchen Gewinnzug für das Programm.
- Falls kein Gewinnzug existiert, liefert der Aufruf `ground.force(YOU)` eine Position, mit der der Gegner in einem Zug gewinnen könnte (was verhindert werden muss :-)
- Nur wenn keiner dieser Fälle auftritt, überprüfen wir sämtliche Zug-Möglichkeiten ...

```

...
PossibleMoves moves = new PossibleMoves(ground);
int tmp = moves.next();
if (tmp == -1) { value = NONE; return;}
value = YOU; do { acceptableChoice = tmp;
    yours = new YourChoice(ground, acceptableChoice);
    if (yours.value() > YOU) { value = yours.value(); return; }
    tmp=moves.next();
} while (tmp != -1);
}
public int value() { return value;}
public int acceptableChoice() { return acceptableChoice; }
public MyChoice select(int place) {
    return yours.answer(place);
}
} // end of class MyChoice

```

- Bei der Iteration über alle Zug-Möglichkeiten geben wir uns bereits mit einem **akzeptablen** Zug zufrieden, genauer gesagt: dem ersten, der mindestens **unentschieden** liefert.

Der Grund:

als **nachziehender** können wir (bei Tic-Tac-Toe und gegen einen optimalen Gegner) nichts besseres erreichen :-)

- `public int value();` liefert wieder den Spiel-Wert des Spielbaums.
- `public int acceptableChoice()` liefert einen akzeptablen Zug und
- `public MyChoice select(int place)` liefert bei gegebenem Spielzug für den akzeptablen Zug und die Antwort place des Gegners den nächsten Teilbaum.

2.3 Die Klasse Game

Die Klasse Game sammelt notwendige Datenstrukturen und Methoden zur Durchführung eines Spiels.

```

public class Game implements PlayConstants {
    public PlayGround ground = new PlayGround();
    private int count = 0;
    private MyChoice gameTree;
    public int nextMove(int place) {

```

```

    if (count == 1)
        gameTree = new MyChoice(ground,place);
    else gameTree = gameTree.select(place);
    return gameTree.acceptableChoice();
}
...

```

- PlayGround ground enthält die jeweils aktuelle Spiel-Konfiguration.
- int count zählt die Anzahl der bereits ausgeführten Züge.
- MyChoice gameTree enthält eine Repräsentation der ME-Strategie.
- public int nextMove(int place)
liefert den Antwort-Zug auf den YOU-Zug place:
 - War place der erste YOU-Zug, wird ein MyChoice-Objekt gameTree für das verbleibende Spiel angelegt.
 - Bei jedem weiteren Zug wird gameTree auf die gemäß place ausgewählte Teil-Strategie gesetzt.
 - Den nächsten ME-Zug liefert jeweils acceptableChoice in gameTree.

```

...
public boolean possibleMove(int place) {
    if (count%2 == 1) return false;
    return (ground.free(place));
}
public boolean finished() { return (count == 9);}
public void move(int place,int who) {
    count++;
    ground.move(place,who);
}
} // end of class Game

```

Weitere nützliche Methoden:

- boolean possibleMove(int place)
überprüft, ob der Zug place überhaupt möglich ist;
- public boolean finished() überprüft, ob bereits 9 Züge gespielt wurden;
- public void move(int place,int who)
erhöht den Zug-Zähler und führt den Zug auf dem Spielfeld ground aus.

2.4 Die graphische Benutzer-Oberfläche

Idee:

- Unbesetzte Felder repräsentieren wir durch Button-Komponenten.
- Die Knöpfe werden in einem (3×3) -Gitter angeordnet.
- Wird ein Zug ausgeführt, wird der entsprechende Knopf durch eine (bemale) Canvas-Fläche ersetzt.

```
import java.awt.*;
import java.awt.event.*;
class MyButton extends Button {
    private int number;
    public MyButton(int i) {
        number = i;
        setBackground(Color.white);
    }
    public int getNumber() { return number;}
} // end of class MyButton
```

- MyButton erweitert die Klasse Button um die Objekt-Variable int number.
- number wird vom Konstruktor gesetzt und mit int getNumber(); abgefragt.

```
public class Arena extends Frame {
    private MoveObserver moveObserver;
    public Arena() {
        setBackground(Color.blue);
        setSize(300,300); setLocation(400,400);
        setLayout(new GridLayout(3,3,3,3)); start();
    }
    public void start() {
        removeAll();
        moveObserver = new MoveObserver(this);
        for (int i=0; i<9;i++) {
            MyButton b = new MyButton(i);
            b.addActionListener(moveObserver); add(b);
        }
    }
}
```

- Arena ist die Unterklasse von `Frame`, in der wir das Spielfeld anlegen.
- Dazu wählen wir das `GridLayout` als `Layout-Manager`.
- `public GridLayout(int row, int col, int cs, int rs);`
konstruiert ein `GridLayout`-Objekt mit `row` Zeilen, `col` Spalten sowie den Abständen `cs` und `rs` zwischen den Spalten bzw. Zeilen.
- Zu der Arena fügen wir neun (durch-numerierete) `MyButton`-Komponenten hinzu.
- Gemeinsamer `ActionListener` für sämtliche Knöpfe ist das `MoveObserver`-Objekt `moveObserver`.
- `public void removeAll();` beseitigt sämtliche Komponenten eines Containers.

```

...
public void move(int place, Canvas canvas) {
    remove(place); add(canvas, place);
}
public static void main(String[] args) {
    Arena arena = new Arena(); arena.setVisible(true);
}
} //end of class Arena

```

- `public void move(int place, Canvas canvas);`
ersetzt die Komponente an der Stelle `place` durch die `Canvas`-Komponente `canvas`.
- `public void remove(int i);`
beseitigt die `i`-te Komponente eines Container-Objekts.
- `public void add(Component c, int i);`
fügt die Komponente `c` als `i`-te Komponente ein.

2.5 Beantwortung eines Zugs

Beachte:

- Ziehen der Tic-Tac-Toe-Spielerin heißt Drücken eines Knopfs.
- Drücken eines Knopfs löst einen `ActionEvent` aus.
- Die Klasse `MoveObserver` verarbeitet die Ereignisse.
- Aufgabe eines `MoveObserver`-Objekts ist es, das Spielfeld zu modifizieren und einen Antwort-Zug zu liefern, d.h. zu finden und dann anzuzeigen.

```

import java.awt.*;
import java.awt.event.*;
public class MoveObserver implements
        ActionListener, PlayConstants {
    private Game game; private Arena frame;
    public MoveObserver(Arena a) {
        game = new Game(); frame = a;
    }
    public void actionPerformed(ActionEvent e) {
        MyButton button = (MyButton) e.getSource();
        int place = button.getNumber();
        if (!game.possibleMove(place)) return;
        frame.move(place,new Cross());
        game.move(place,YOU);
        frame.setVisible(true);
        ...
    }
}

```

- Wurde das Feld place ermittelt, das die Gegnerin setzte, wird der Knopf an dieser Stelle durch ein neues Cross-Element ersetzt.
- Dann wird der Zug auch auf der internen Repräsentation des Spielbretts ausgeführt.
- `frame.setVisible(true);` macht diese Änderung auf dem Bildschirm sichtbar.

```

...
if (game.ground.won(place,YOU)) {
    game.ground.clear();
    (new MyDialog(frame,"You won ...")).setVisible(true);
    return;
}
if (game.finished()) {
    (new MyDialog(frame,"No winner ...")).setVisible(true);
    return;
}
place = game.nextMove(place);
frame.move(place,new Circle());
game.move(place,ME);
frame.setVisible(true);
...

```

- Die Objekt-Methode `public void clear();` der Klasse `PlayGround` verhindert weitere Züge, indem sämtliche Felder auf 3 gesetzt werden (**Hack** – sorry!).

- Falls die Spielerin YOU gewonnen hat, beenden wir das Spiel und erzeugen ein Dialog-Fenster der Klasse MyDialog.
- Ähnlich verfahren wir, wenn das Spiel nicht verloren, aber zuende ist ...
- Ist das Spiel weder verloren noch zuende, bestimmen wir unseren nächsten Zug.
- Diesen führen wir auf unserer Intern-Darstellung des Spielbretts wie in dem Arena-Frame aus.

```

...
    if (game.ground.won(place,ME)) {
        game.ground.clear();
        (new MyDialog(frame,"I won ...")).setVisible(true);
        return;
    }
}
} // end of class MoveObserver

```

- Falls unser Zug zum Gewinn führte, beenden wir das Spiel und erzeugen ein MyDialog-Fenster.

2.6 Ein Dialog-Fenster

Aufgaben:

1. Anzeige des Spiel-Ausgangs;
2. Ermöglichen der Beendigung bzw. Wiederholung des Spiels.

```

import java.awt.*;
import java.awt.event.*;
public class MyDialog extends Frame implements ActionListener {
    private Button repeat, kill; private Arena arena;
    public MyDialog(Arena frame, String string) {
        arena = frame;
        setLocation(200,200); setSize(250,50);
        setBackground(Color.yellow);
        setForeground(Color.blue);
        setLayout(new FlowLayout());
        add(new Label(string));
        repeat = new Button("new");
        repeat.addActionListener(this); add(repeat);
        kill = new Button("kill");
        kill.addActionListener(this); add(kill);
    }
}

```

- `public void setForeground(Color c);`
setzt die aktuelle Vordergrund-Farbe auf c. Schrift erscheint dann z.B. in dieser Farbe.
- Wir legen ein neues Fenster mit gelbem Hintergrund und blauem Vordergrund an.
- Wir benutzen das `FlowLayout`, um Elemente im Fenster zu positionieren.
- Den Kommentar zum Spiel-Ergebnis übergeben wir dem Konstruktor, der es in einem Label-Element anzeigt.
- Weiterhin fügen wir zwei Knöpfe für Wiederholung bzw. Beendigung hinzu.
- Gemeinsamer `ActionListener` beider Knöpfe ist das Fenster-Objekt selbst ...

```

...
    public void actionPerformed(ActionEvent e) {
        Button b = (Button) e.getSource();
        if (b == kill) { arena.setVisible(false); System.exit(0);}
        else {
            arena.start(); arena.setVisible(true);
            setVisible(false);
        }
    }
} // end of class MyDialog

```

- `public void exit(int x);`
ist eine Klassen-Methode der Klasse `System`, die die Applikation (mit Rückgabe-Wert x) beendet.

- Falls der "kill"-Knopf gedrückt wurde, wird die Applikation beendet.
- Falls der "new"-Knopf gedrückt wurde, wird für arena die `start()`-Methode aufgerufen, d.h. ein neues Game-Objekt angelegt und die Komponenten in arena neu zu Knöpfen initialisiert.
- `setVisible(false);`
lässt das Fenster verschwinden.

2.7 Effizienz

Problem:

Spielbäume können **RIESIG** werden!!

Unsere Lösung:

- Wir erzeugen die ME-Strategie nicht für alle möglichen Spiel-Verläufe, sondern erst **nach dem ersten Zug** der Gegnerin.
Spart ...**Faktor 9**
- Wir berücksichtigen **Zug-Zwang**.
Spart ...red??!!...:-)
- Wir sind mit **akzeptablen** ME-Zügen zufrieden.
Spart ungefähr ...**Faktor 2**

Achtung:

- Für Tic-Tac-Toe reicht das vollkommen aus: pro Spielverlauf werden zwischen **126** und **1142** MyChoice-Knoten angelegt ...
- Für komplexere Spiele wie **Dame**, **Schach** oder gar **Go** benötigen wir weitere Ideen ...

1. Idee: Eröffnungen

- Tabelliere Anfangs-Stücke optimaler Spiel-Verläufe.
- Konstruiere die Strategie erst ab der ersten Konfiguration, die von den tabellierten Eröffnungen abweicht ...

Beispiel: Tic-Tac-Toe

Wir könnten z.B. beste Antworten auf jeden möglichen Eröffnungs-Zug tabellieren:

```
public interface Opening {
    int[] OPENING = {
        4, 4, 4, 4, 2, 4, 4, 4, 4
    };
}
```

- Die Funktion `int nextMove(int place);` schlägt dann den ersten Antwort-Zug in `OPENING` nach.
- Erst bei der zweiten Antwort (d.h. für den vierten Stein auf dem Brett) wird die ME-Strategie konstruiert.
- Dann bleiben grade mal höchstens $6! = 720$ Spiel-Fortsetzungen übrig ... die Anzahl der tatsächlich benötigten `MyChoice`-Knoten scheint aber nur noch zwischen 9 und 53 zu schwanken (!!!)

2. Idee: Bewertungen

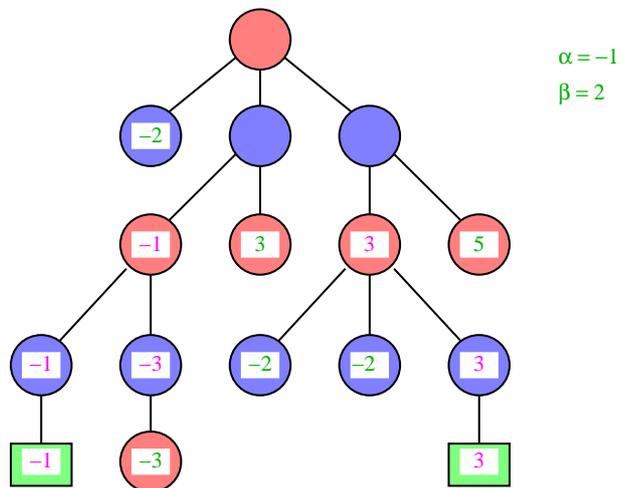
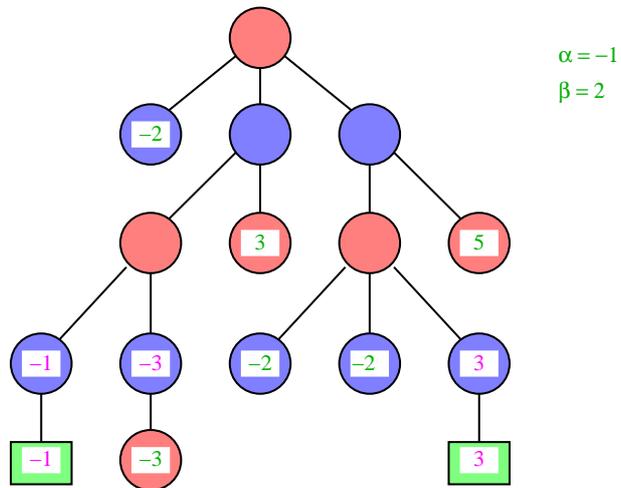
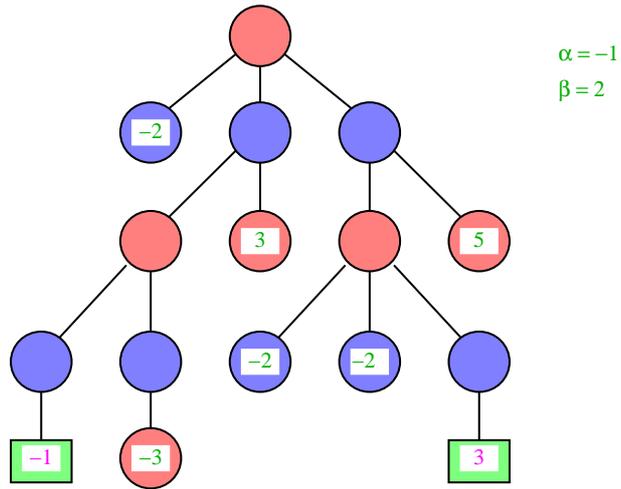
Finde eine geeignete Funktion `advice`, die die Erfolgsaussichten einer Konfiguration direkt abschätzt, d.h. ohne Aufbau eines Spielbaums.

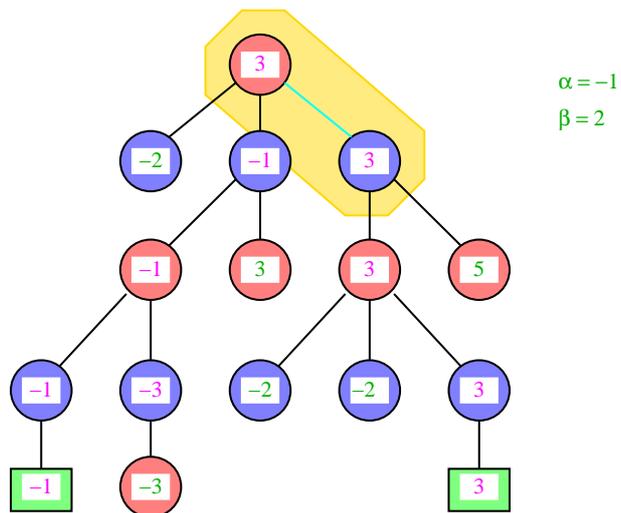
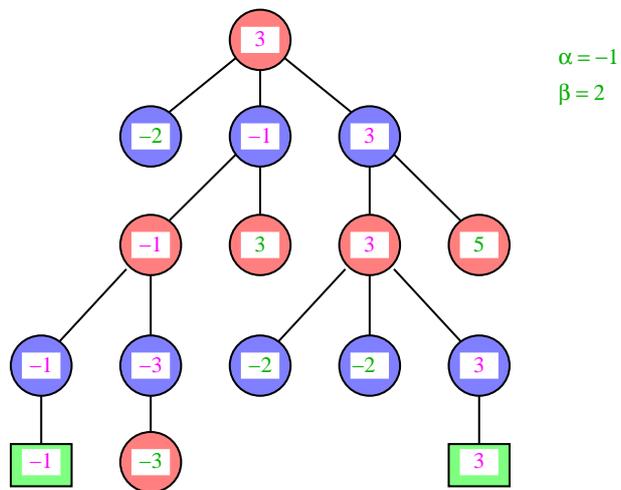
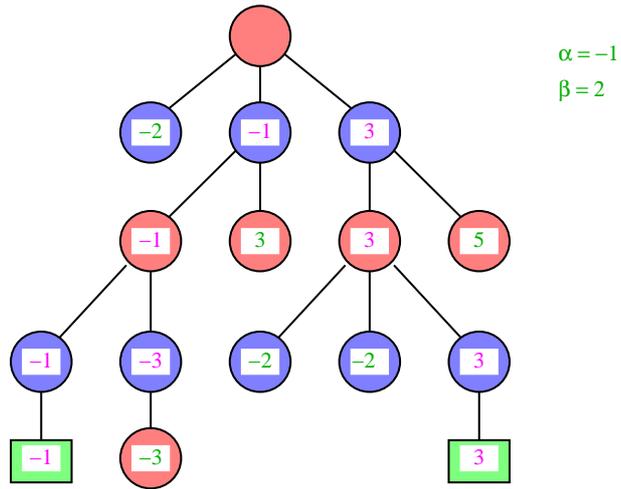
Achtung:

- I.a. ist eine solche Funktion nicht bekannt :-)
- Man muss mit unpräzisen bis fehlerhaften Bewertungs-Funktionen zurecht kommen ...

3. Idee: (α, β) -Pruning

- Wir nehmen an, wir hätten eine halbwegs zuverlässige Bewertungsfunktion `advice`, d.h. es gibt Zahlen $\alpha < 0 < \beta$ so dass für Konfigurationen `conf` gilt:
 - Ist `advice(conf) < α` , gewinnt voraussichtlich die Gegnerin;
 - Ist `$\beta < advice(conf)$` , gewinnt voraussichtlich das Programm.
- Zur Bestimmung unseres nächsten Zugs, betrachten wir sukzessive alle Nachfolger-Konfigurationen `conf`.
 - Ist `$\beta < advice(conf)$` , ist der Zug akzeptabel.
 - Gibt es keinen akzeptablen Zug, betrachten wir rekursiv die Nachfolger aller Konfigurationen `conf`, für die `$\alpha < advice(conf)$` .
 - Für gegnerische Konfigurationen gehen wir dual vor ...





Vorteil:

Die Anzahl der zu untersuchenden Konfigurationen wird (hoffentlich ;-)
beträchtlich eingeschränkt!

Nachteil:

Ist die Bewertungsfunktion offensichtlich fehlerhaft, lässt sich das Programm austricksen ...

Frage:

Wie findet man eine Bewertungsfunktion deren Fehlerhaftigkeit nicht so offensichtlich ist???

Ausblick:

- Nicht alle 2-Personen-Spiele sind endlich.
- Gelegentlich hängt der Effekt eines Zugs zusätzlich vom Zufall ab.
- Eventuell ist die aktuelle Konfiguration nur partiell bekannt.

⇒ Spieltheorie

3 Korrektheit von Programmen

- Programmierer machen Fehler :-)
- Programmierfehler können **teuer** sein, z.B. wenn eine Rakete explodiert, ein firmenwichtiges System für **Stunden** ausfällt ...
- In einigen Systemen dürfen **keine** Fehler vorkommen, z.B. Steuerungssoftware für Flugzeuge, Signalanlagen für Züge, Airbags in Autos ...

Problem:

Wie können wir sicherstellen, dass ein Programm das **richtige** tut?

Ansätze:

- Sorgfältiges Vorgehen bei der Software-Entwicklung;
- Systematisches Testen
 \implies formales Vorgehensmodell (**Software Engineering**)
- Beweis der Korrektheit
 \implies **Verifikation**

Hilfsmittel: Zusicherungen

Beispiel:

```
public class GGT extends MiniJava {
    public static void main (String[] args) {
        int x, y, a, b;
        a = read(); b = read();
        x = a; y = b;
        while (x != y)
            if (x > y) x = x - y;
            else     y = y - x;

        assert(x != y);

        write(x);
    } // Ende der Definition von main();
} // Ende der Definition der Klasse GGT;
```

Kommentare:

- Die statische Methode `assert()` erwartet ein Boolesches Argument.
- Bei normaler Programm-Ausführung wird jeder Aufruf `assert(e);` ignoriert :-)
- Starten wir **Java** mit der Option: `-ea` (**enable assertions**), werden die `assert`-Aufrufe ausgewertet:
 - ⇒ Liefert ein Argument-Ausdruck `true`, fährt die Programm-Ausführung fort.
 - ⇒ Liefert ein Argument-Ausdruck `false`, wird ein **Fehler** `AssertionError` geworfen.

Achtung:

Der Laufzeit-Test soll eine **Eigenschaft** des Programm-Zustands bei Erreichen eines Programm-Punkts überprüfen.

Der Test sollte **keineswegs** den Programm-Zustand verändern !!!
Sonst zeigt das beobachtete System ein anderes Verhalten als das unbeobachtete ???

Tipp:

Um Eigenschaften komplizierterer Datenstrukturen zu überprüfen, empfiehlt es sich, getrennt **Inspector**-Klassen anzulegen, deren Objekte eine Datenstruktur **störungsfrei** besichtigen können :-)

Problem:

- Es gibt i.a. sehr viele Programm-Ausführungen :-)
- Einhalten der Zusicherungen kann das **Java**-Laufzeit-System immer nur für eine Programm-Ausführung überprüfen :-)



Wir benötigen eine generelle Methode, um das Einhalten einer Zusicherung zu **garantieren** ...

3.1 Verifikation von Programmen



Robert W Floyd, Stanford U. (1936 – 2001)

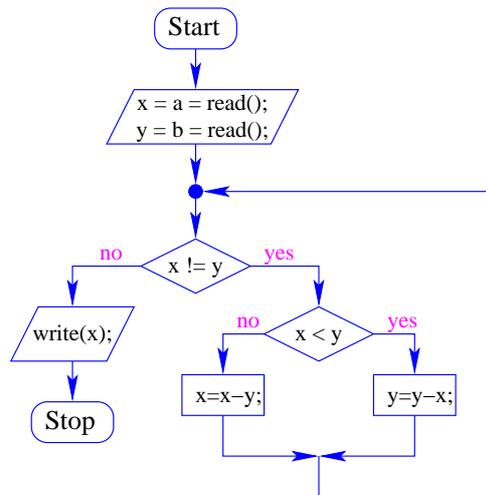
Vereinfachung:

Wir betrachten erst mal nur **MiniJava** ;-)

Idee:

- Wir schreiben eine Zusicherung an **jeden** Programmpunkt :-)
- Wir überprüfen **lokal**, dass die Zusicherungen von den einzelnen Anweisungen im Programm eingehalten werden.

Unser Beispiel:



Diskussion:

- Die Programmpunkte entsprechen den **Kanten** im Kontrollfluss-Diagramm :-)
- Wir benötigen eine Zusicherung pro Kante ...

Hintergrund:

$d \mid x$ gilt genau dann wenn $x = d \cdot z$ für eine ganze Zahl z .
Für ganze Zahlen x, y sei $ggT(x, y) = 0$, falls $x = y = 0$
und andernfalls die größte ganze Zahl d , die x und y teilt.
Dann gelten unter anderem die folgenden Gesetze:

$$\begin{aligned} ggT(x, 0) &= |x| \\ ggT(x, x) &= |x| \\ ggT(x, y) &= ggT(x, y - x) \\ ggT(x, y) &= ggT(x - y, y) \end{aligned}$$

Idee für das Beispiel:

- Am Anfang gilt nix :-)
- Nach `a=read(); x=a;` gilt $a = x$:-)

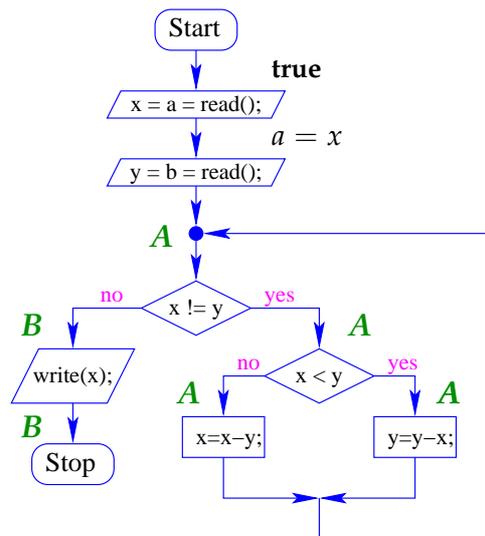
- Vor Betreten und während der Schleife soll gelten:

$$A \equiv ggT(a, b) = ggT(x, y)$$

- Am Programm-Ende soll gelten:

$$B \equiv A \wedge x = y$$

Unser Beispiel:



Frage:

Wie überprüfen wir, dass Zusicherungen lokal zusammen passen?

Teilproblem 1: Zuweisungen

Betrachte z.B. die Zuweisung: $x = y + z;$

Damit **nach** der Zuweisung gilt: $x > 0,$ // Nachbedingung

muss **vor** der Zuweisung gelten: $y + z > 0.$ // Vorbedingung

Allgemeines Prinzip:

- Jede Anweisung transformiert eine Nachbedingung B in eine **minimale** Anforderung, die **vor** Ausführung erfüllt sein muss, damit B **nach** der Ausführung gilt :-)

- Im Falle einer Zuweisung $x = e;$ ist diese **schwächste Vorbedingung** (engl.: **weakest precondition**) gegeben durch

$$\mathbf{WP}[[x = e;]](B) \equiv B[e/x]$$

Das heißt: wir **substituieren** einfach in B überall x durch e !!!

- Eine beliebige Vorbedingung A für eine Anweisung s ist **gültig**, sofern

$$A \Rightarrow \mathbf{WP}[[s]](B)$$

// A **impliziert** die schwächste Vorbedingung für B .

Beispiel:

Zuweisung:	$x = x - y;$
Nachbedingung:	$x > 0$
schwächste Vorbedingung:	$x - y > 0$
stärkere Vorbedingung:	$x - y > 2$
noch stärkere Vorbedingung:	$x - y = 3$:-)

... im GGT-Programm (1):

Zuweisung:	$x = x - y;$
Nachbedingung:	A
schwächste Vorbedingung:	

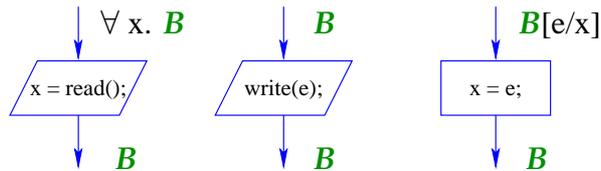
$$\begin{aligned} A[x - y/x] &\equiv \text{ggT}(a, b) = \text{ggT}(x - y, y) \\ &\equiv \text{ggT}(a, b) = \text{ggT}(x, y) \\ &\equiv A \end{aligned}$$

... im GGT-Programm (2):

Zuweisung:	$y = y - x;$
Nachbedingung:	A
schwächste Vorbedingung:	

$$\begin{aligned}
A[y - x/y] &\equiv \text{ggT}(a, b) = \text{ggT}(x, y - x) \\
&\equiv \text{ggT}(a, b) = \text{ggT}(x, y) \\
&\equiv A
\end{aligned}$$

Zusammenstellung:

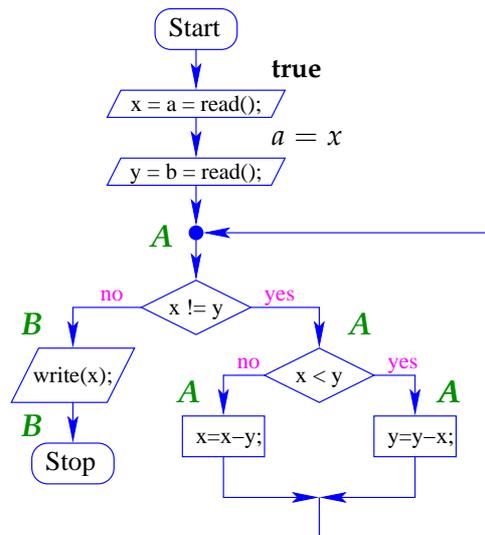


$$\begin{aligned}
\text{WP}[\text{;}](B) &\equiv B \\
\text{WP}[\text{x = e;}](B) &\equiv B[e/x] \\
\text{WP}[\text{x = read();}](B) &\equiv \forall x. B \\
\text{WP}[\text{write(e);}](B) &\equiv B
\end{aligned}$$

Diskussion:

- Die Zusammenstellung liefert für alle Aktionen jeweils die **schwächsten** Vorbedingungen für eine Nachbedingung B .
- Eine Ausgabe-Anweisung ändert keine Variablen.
Deshalb ist da die schwächste Vorbedingung B selbst ;-)
- Eine Eingabe-Anweisung x=read(); ändert die Variable x auf unvorhersehbare Weise.
Damit nach der Eingabe B gelten kann, muss B **vor** der Eingabe für jedes mögliche x gelten ;-)

Orientierung:



Für die Anweisungen: `b = read(); y = b;` berechnen wir:

$$\begin{aligned} \mathbf{WP}[\mathit{y} = \mathit{b};] (A) &\equiv A[b/y] \\ &\equiv \mathit{ggT}(a, b) = \mathit{ggT}(x, b) \end{aligned}$$

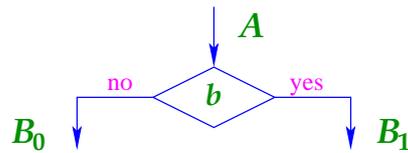
$$\begin{aligned} \mathbf{WP}[\mathit{b} = \mathit{read}();] (\mathit{ggT}(a, b) = \mathit{ggT}(x, b)) \\ &\equiv \forall b. \mathit{ggT}(a, b) = \mathit{ggT}(x, b) \\ &\Leftarrow a = x \quad \text{: -) } \end{aligned}$$

Für die Anweisungen: `a = read(); x = a;` berechnen wir:

$$\begin{aligned} \mathbf{WP}[\mathit{x} = \mathit{a};] (a = x) &\equiv a = a \\ &\equiv \mathbf{true} \end{aligned}$$

$$\begin{aligned} \mathbf{WP}[\mathit{a} = \mathit{read}();] (\mathbf{true}) &\equiv \forall a. \mathbf{true} \\ &\equiv \mathbf{true} \quad \text{: -) } \end{aligned}$$

Teilproblem 2: Verzweigungen



Es sollte gelten:

- $A \wedge \neg b \Rightarrow B_0$ und
- $A \wedge b \Rightarrow B_1$.

Das ist der Fall, falls A die **schwächste Vorbedingung** der Verzweigung:

$$\mathbf{WP}[[b]](B_0, B_1) \equiv ((\neg b) \Rightarrow B_0) \wedge (b \Rightarrow B_1)$$

impliziert :-)

Die schwächste Vorbedingung können wir umschreiben in:

$$\begin{aligned} \mathbf{WP}[[b]](B_0, B_1) &\equiv (b \vee B_0) \wedge (\neg b \vee B_1) \\ &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1) \vee (B_0 \wedge B_1) \\ &\equiv (\neg b \wedge B_0) \vee (b \wedge B_1) \end{aligned}$$

Beispiel:

$$B_0 \equiv x > y \wedge y > 0 \qquad B_1 \equiv x > 0 \wedge y > x$$

Sei b die Bedingung $y > x$.

Dann ist die schwächste Vorbedingung:

$$\begin{aligned} &(x > y \wedge y > 0) \vee (x > 0 \wedge y > x) \\ &\equiv x > 0 \wedge y > 0 \wedge x \neq y \end{aligned}$$

... im GGT-Beispiel:

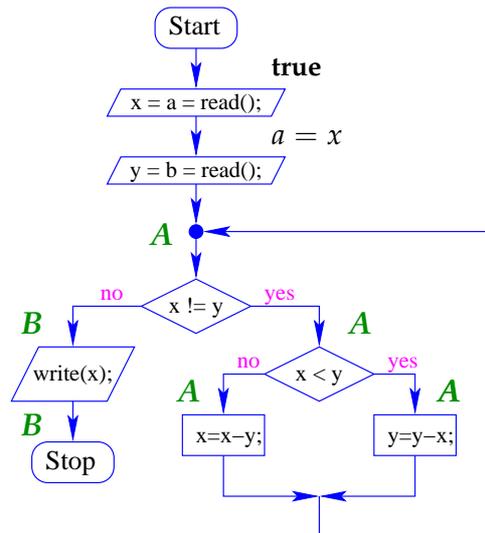
$$\begin{aligned} b &\equiv y > x \\ \neg b \wedge A &\equiv x \geq y \wedge \text{ggT}(a, b) = \text{ggT}(x, y) \\ b \wedge A &\equiv y > x \wedge \text{ggT}(a, b) = \text{ggT}(x, y) \end{aligned}$$

⇒ Die schwächste Vorbedingung ist:

$$ggT(a, b) = ggT(x, y)$$

... also genau A :-)

Orientierung:



Analog argumentieren wir für die Zusicherung vor der Schleife:

$$b \equiv y \neq x$$

$$\neg b \wedge B \equiv B$$

$$b \wedge A \equiv A \wedge x \neq y$$

⇒ $A \equiv (A \wedge x = y) \vee (A \wedge x \neq y)$ ist die schwächste Vorbedingung für die Verzweigung :-)

Zusammenfassung der Methode:

- Annotiere jeden Programmpunkt mit einer Zusicherung.

- Überprüfe für jede Anweisung s zwischen zwei Zusicherungen A und B , dass A die schwächste Vorbedingung von s für B impliziert, d.h.:

$$A \Rightarrow \mathbf{WP}[[s]](B)$$

- Überprüfe entsprechend für jede Verzweigung mit Bedingung b , ob die Zusicherung A vor der Verzweigung die schwächste Vorbedingung für die Nachbedingungen B_0 und B_1 der Verzweigung impliziert, d.h.

$$A \Rightarrow \mathbf{WP}[[b]](B_0, B_1)$$

Solche Annotierungen nennen wir **lokal konsistent**.

3.2 Korrektheit

Fragen:

- Welche Programm-Eigenschaften können wir mithilfe lokal konsistenter Annotierungen garantieren ?
- Wie können wir nachweisen, dass unser Verfahren **keine falschen Ergebnisse** liefert ??

Erinnerung (1):

- In **MiniJava** können wir ein Zustand σ aus einer **Variablen-Belegung**, d.h. einer Abbildung der Programm-Variablen auf ganze Zahlen (ihren Werten), z.B.:

$$\sigma = \{x \mapsto 5, y \mapsto -42\}$$

- Ein Zustand σ **erfüllt** eine Zusicherung A , falls

$$A[\sigma(x)/x]_{x \in A}$$

// wir substituieren jede Variable in A durch ihren Wert in σ
eine **wahre** Aussage ist, d.h. äquivalent **true**.

Wir schreiben: $\sigma \models A$.

Beispiel:

$$\begin{aligned} \sigma &= \{x \mapsto 5, y \mapsto 2\} \\ A &\equiv (x > y) \\ A[5/x, 2/y] &\equiv (5 > 2) \\ &\equiv \mathbf{true} \end{aligned}$$

$$\begin{aligned} \sigma &= \{x \mapsto 5, y \mapsto 12\} \\ A &\equiv (x > y) \\ A[5/x, 12/y] &\equiv (5 > 12) \\ &\equiv \mathbf{false} \end{aligned}$$

Triviale Eigenschaften:

$\sigma \models \mathbf{true}$ für jedes σ
 $\sigma \models \mathbf{false}$ für kein σ

$\sigma \models A_1$ und $\sigma \models A_2$ ist äquivalent zu
 $\sigma \models A_1 \wedge A_2$
 $\sigma \models A_1$ oder $\sigma \models A_2$ ist äquivalent zu
 $\sigma \models A_1 \vee A_2$

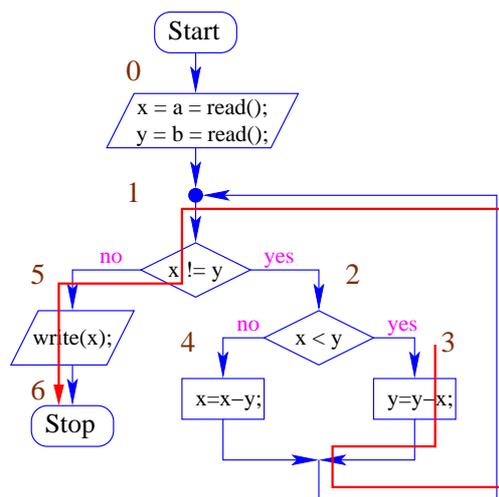
Erinnerung (2):

- Eine Programmausführung π durchläuft einen Pfad im Kontrollfluss-Graphen :-)
 - Sie beginnt in einem Programmpunkt u_0 in einem Anfangszustand σ_0 und führt in einen Programmpunkt u_m und einen Endzustand σ_m .
 - Jeder Schritt der Programm-Ausführung führt eine Aktion aus und ändert Programmpunkt und Zustand :-)
- \implies Wir können π als Folge darstellen:

$$(u_0, \sigma_0) s_1 (u_1, \sigma_1) \dots s_m (u_m, \sigma_m)$$

wobei die s_i Elemente des Kontrollfluss-Graphen sind, d.h. Anweisungen oder Bedingungen ...

Beispiel:



Nehmen wir an, wir starten in Punkt 3 mit $\{x \mapsto 6, y \mapsto 12\}$.

Dann ergibt sich die **Programmausführung**:

$$\begin{aligned}\pi = & (3, \{x \mapsto 6, y \mapsto 12\}) \quad y = y-x; \\ & (1, \{x \mapsto 6, y \mapsto 6\}) \quad (x \neq y) \\ & (5, \{x \mapsto 6, y \mapsto 6\}) \quad \text{write}(x); \\ & (6, \{x \mapsto 6, y \mapsto 6\})\end{aligned}$$

Satz:

Sei p ein **MiniJava**-Programm, Sei π eine Programmausführung, die im Programmpunkt u startet und zum Programmpunkt v führt.

Annahmen:

- Die Programmpunkte von p seien lokal konsistent mit Zusicherungen annotiert.
- Der Programmpunkt u sei mit A annotiert.
- Der Programmpunkt v sei mit B annotiert.

Dann gilt:

Erfüllt der Anfangszustand von π die Zusicherung A , dann erfüllt der Endzustand die Zusicherung B .

Bemerkungen:

- Ist der Startpunkt des Programms mit **true** annotiert, dann erfüllt **jede** Programmausführung, die den Programmpunkt v erreicht, die Zusicherung an v :-)
- Zum Nachweis, dass eine Zusicherung A an einem Programmpunkt v gilt, benötigen wir eine lokal konsistente Annotierung mit zwei Eigenschaften:
 - (1) der Startpunkt ist mit **true** annotiert;
 - (2) Die Zusicherung an v **impliziert** A :-)
- Unser Verfahren gibt (vorerst) keine Garantie, dass v überhaupt erreicht wird !!!
- Falls ein Programmpunkt v mit der Zusicherung **false** annotiert werden kann, kann v **nie** erreicht werden :-))

Beweis:

Sei $\pi = (u_0, \sigma_0) s_1(u_1, \sigma_1) \dots s_m(u_m, \sigma_m)$

Gelte: $\sigma_0 \models A$.

Wir müssen zeigen: $\sigma_m \models B$.

Idee:

Induktion nach der Länge m der Programmausführung :-)

$m = 0$:

Der Endpunkt der Ausführung ist gleich dem Startpunkt

$\implies \sigma_0 = \sigma_m$ und $A \equiv B$

\implies Behauptung gilt :-)

$m > 0$:

Annahme: Die Behauptung gilt bereits für $m - 1$.

Sei B' die Zusicherung am Punkt u_{m-1} .

$\implies \sigma_{m-1} \models B'$

Behandeln wir erst einmal Tests $s_m \equiv b$.

Dann ist insbesondere $\sigma_{m-1} = \sigma_m$:-)

1. Möglichkeit: $\sigma_m \models b$

$\implies B' \Rightarrow \mathbf{WP}[[b]](C, B)$ wobei
 $\mathbf{WP}[[b]](C, B) \equiv (\neg b \Rightarrow C) \wedge (b \Rightarrow B)$

$\implies \sigma_m \models b \wedge (b \Rightarrow B)$

$\implies \sigma_m \models B$:-)

Behandeln wir erst einmal Tests $s_m \equiv b$.

Dann ist insbesondere $\sigma_{m-1} = \sigma_m$:-)

2. Möglichkeit: $\sigma_m \models \neg b$

$\implies B' \Rightarrow \mathbf{WP}[[b]](B, C)$ wobei
 $\mathbf{WP}[[b]](B, C) \equiv (\neg b \Rightarrow B) \wedge (b \Rightarrow C)$

$$\begin{aligned} \implies \sigma_m &\models \neg b \wedge (\neg b \Rightarrow B) \\ \implies \sigma_m &\models B \quad (-) \end{aligned}$$

Nun behandeln wir Anweisungen.

1. Möglichkeit: $s_m \equiv ;$

Dann gilt:

- $\sigma_{m-1} = \sigma_m$
- $\mathbf{WP}[\text{;}](B) \equiv B$

$$\begin{aligned} \implies B' &\Rightarrow B \\ \implies \sigma_{m-1} = \sigma_m &\models B \quad (-) \end{aligned}$$

2. Möglichkeit: $s_m \equiv \text{write}(e);$

Dann gilt:

- $\sigma_{m-1} = \sigma_m$
- $\mathbf{WP}[\text{;write}(e)](B) \equiv B$

$$\begin{aligned} \implies B' &\Rightarrow B \\ \implies \sigma_{m-1} = \sigma_m &\models B \quad (-) \end{aligned}$$

3. Möglichkeit: $s_m \equiv x = \text{read}();$

Dann gilt:

- $\sigma_m = \sigma_{m-1} \oplus \{x \mapsto c\}$ für ein $c \in \mathbb{Z}$
- $\mathbf{WP}[x = \text{read}();](B) \equiv \forall x. B$

$$\begin{aligned} \implies B' &\Rightarrow \forall x. B \Rightarrow B[c/x] \\ \implies \sigma_m &\models B \quad (-) \end{aligned}$$

4. Möglichkeit: $s_m \equiv x = e;$ Dann gilt:

- $\sigma_m = \sigma_{m-1} \oplus \{x \mapsto c\}$ für $c = \text{Wert von } e \text{ in } \sigma_{m-1}$
- $\mathbf{WP}[x = e;](B) \equiv B[e/x]$

$$\begin{aligned} \implies B' &\Rightarrow B[e/x] \implies \sigma_{m-1} \models B[c/x] \\ \implies \sigma_m &\models B \quad (-) \end{aligned}$$

Fazit:

- Das Verfahren nach Floyd ermöglicht uns zu beweisen, dass eine Zusicherung B bei Erreichen eines Programmpunkts stets (bzw. unter geeigneten Zusatzannahmen $-)$ gilt
- ...

- Zur Durchführung benötigen wir:
 - Zusicherungen an jedem Programmpunkt :-)
 - Nachweis, dass die Zusicherungen lokal konsistent sind
⇒ Logik, automatisches Beweisen

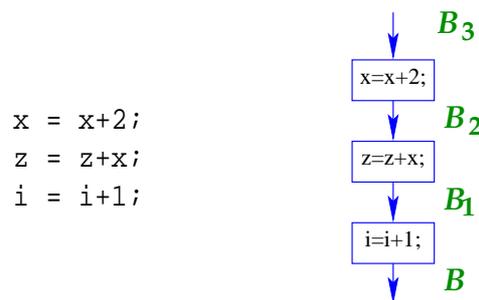
3.3 Optimierung

Ziel: Verringerung der benötigten Zusicherungen

Beobachtung:

Hat das Programm **keine Schleifen**, können wir für jeden Programmpunkt eine hinreichende Vorbedingung **ausrechnen !!!**

Beispiel:



Beispiel (Fort.):

Sei $B \equiv z = i^2 \wedge x = 2i - 1$
 Dann rechnen wir:

$$\begin{aligned}
 B_1 &\equiv \mathbf{WP}[[i = i+1;]](B) &&\equiv z = (i + 1)^2 \wedge x = 2(i + 1) - 1 \\
 &&&\equiv z = (i + 1)^2 \wedge x = 2i + 1 \\
 B_2 &\equiv \mathbf{WP}[[z = z+x;]](B_1) &&\equiv z + x = (i + 1)^2 \wedge x = 2i + 1 \\
 &&&\equiv z = i^2 \wedge x = 2i + 1 \\
 B_3 &\equiv \mathbf{WP}[[x = x+2;]](B_2) &&\equiv z = i^2 \wedge x + 2 = 2i + 1 \\
 &&&\equiv z = i^2 \wedge x = 2i - 1 \\
 &&&\equiv B \quad ;-)
 \end{aligned}$$

Idee:

- Für jede Schleife wähle **einen** Programmpunkt aus.
Sinnvolle Auswahlen:

- Vor der Bedingung;
- Am Beginn des Rumpfs;
- Am Ende des Rumpfs ...
- Stelle für jeden gewählten Punkt eine Zusicherung bereit
 - ⇒ **Schleifen-Invariante**
- Für alle übrigen Programmpunkte bestimmen wir Zusicherungen mithilfe $WP[\dots]() :-)$

Beispiel:

```

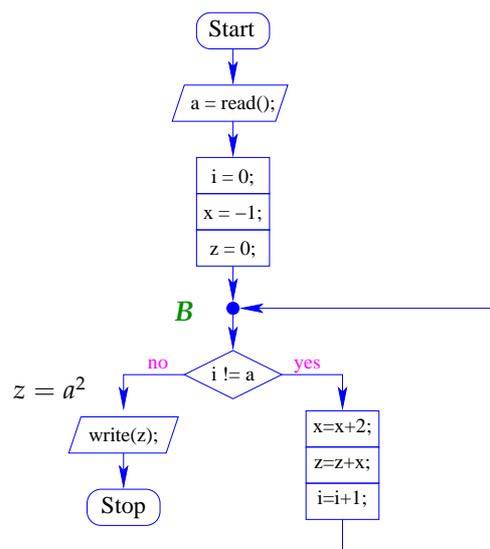
int a, i, x, z;
a = read();
i = 0;
x = -1;
z = 0;
while (i != a) {
    x = x+2;
    z = z+x;
    i = i+1;
}

```

assert(z==a*a);

write(z);

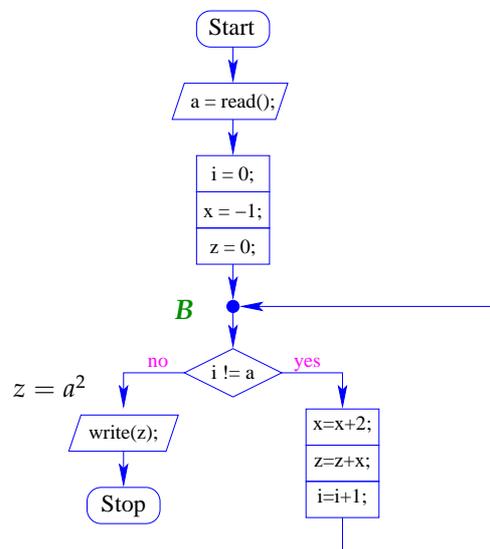
Beispiel:



Wir überprüfen:

$$\begin{aligned}
 \mathbf{WP}[\![i \neq a]\!](z = a^2, B) & \\
 \equiv (i = a \wedge z = a^2) \vee (i \neq a \wedge B) & \\
 \equiv (i = a \wedge z = a^2) \vee (i \neq a \wedge z = i^2 \wedge x = 2i - 1) & \\
 \Leftarrow (i \neq a \wedge z = i^2 \wedge x = 2i - 1) \vee (i = a \wedge z = i^2 \wedge x = 2i - 1) & \\
 \equiv z = i^2 \wedge x = 2i - 1 \equiv B & \quad \text{:-)
 \end{aligned}$$

Orientierung:



Wir überprüfen:

$$\begin{aligned}
 \mathbf{WP}[\![z = 0;]\!](B) & \equiv 0 = i^2 \wedge x = 2i - 1 \\
 & \equiv i = 0 \wedge x = -1 \\
 \mathbf{WP}[\![x = -1;]\!](i = 0 \wedge x = -1) & \equiv i = 0 \\
 \mathbf{WP}[\![i = 0;]\!](i = 0) & \equiv \mathbf{true} \\
 \mathbf{WP}[\![a = \text{read}();]\!](\mathbf{true}) & \equiv \mathbf{true} \quad \text{:-)
 \end{aligned}$$

3.4 Terminierung

Problem:

- Mit unserer Beweistechnik können wir nur beweisen, dass eine Eigenschaft gilt wann immer wir einen Programmpunkt erreichen !!!
- Wie können wir aber garantieren, dass das Programm immer terminiert ?
- Wie können wir eine Bedingung finden, unter der das Programm immer terminiert ??

Beispiele:

- Das ggT-Programm terminiert nur für Eingaben a, b mit: $a > 0$ und $b > 0$.
- Das Quadrier-Programm terminiert nur für Eingaben $a \geq 0$.
- `while (true) ;` terminiert nie.
- Programme ohne Schleifen terminieren immer :-)

Lässt sich dieses Beispiel verallgemeinern ??

Beispiel:

```
int i, j, t;
t = 0;
i = read();
while (i>0) {
    j = read();
    while (j>0) { t = t+1; j = j-1; }
    i = i-1;
}
write(t);
```

- Die gelesene Zahl i (falls positiv) gibt an, wie oft eine Zahl j eingelesen wird.
- Die Gesamtlaufzeit ist (im wesentlichen :-)
die Summe der positiven für j gelesenen Werte
⇒ das Programm terminiert immer !!!

Programme nur mit for-Schleifen der Form:

```
for (i=n; i>0; i--) {...}
// im Rumpf wird i nicht modifiziert
```

... terminieren ebenfalls immer :-))

Frage:

Wie können wir aus dieser Beobachtung eine Methode machen, die auf beliebige Schleifen anwendbar ist ?

Idee:

- Weise nach, dass jede Schleife nur endlich oft durchlaufen wird ...
- Finde für jede Schleife eine Kenngröße r , die zwei Eigenschaften hat:
 - (1) Wenn immer der Rumpf betreten wird, ist $r > 0$;
 - (2) Bei jedem Schleifen-Durchlauf wird r kleiner :-)
- Transformiere das Programm so, dass es neben der normalen Programmausführung zusätzlich die Kenngrößen r mitberechnet.
- Verifiziere, dass (1) und (2) gelten :-)

Beispiel: Sicheres ggT-Programm

```
int a, b, x, y;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
else {
    while (x != y)
        if (y > x) y = y-x;
        else x = x-y;
    write(x);
}
```

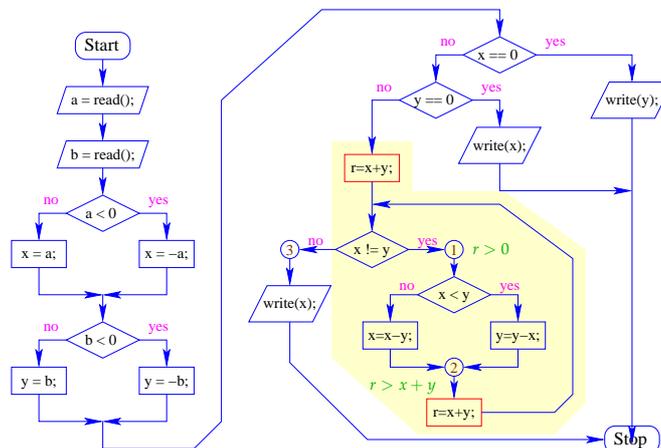
Wir wählen: $r = x + y$

Transformation:

```

int a, b, x, y, r;
a = read(); b = read();
if (a < 0) x = -a; else x = a;
if (b < 0) y = -b; else y = b;
if (x == 0) write(y);
else if (y == 0) write(x);
else { r = x+y;
      while (x != y) {
        if (y > x) y = y-x;
        else      x = x-y;
        r = x+y; }
      write(x);
}

```



An den Programmpunkten 1, 2 und 3 machen wir die Zusicherungen:

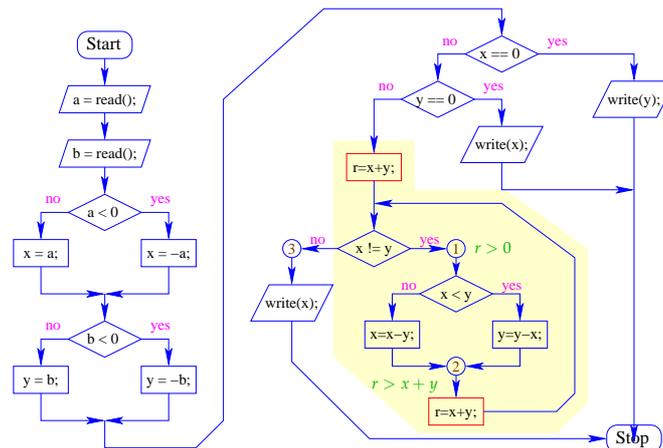
- (1) $A \equiv x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y$
- (2) $B \equiv x > 0 \wedge y > 0 \wedge r > x + y$
- (3) **true**

Dann gilt:

$$A \Rightarrow r > 0 \quad \text{und} \quad B \Rightarrow r > x + y$$

$$\begin{aligned}
\mathbf{WP}[x \neq y](\mathbf{true}, A) &\equiv x = y \vee A \\
&\Leftarrow x > 0 \wedge y > 0 \wedge r = x + y \\
&\equiv C \\
\mathbf{WP}[r = x+y](C) &\equiv x > 0 \wedge y > 0 \\
&\Leftarrow B \\
\mathbf{WP}[x = x-y](B) &\equiv x > y \wedge y > 0 \wedge r > x \\
\mathbf{WP}[y = y-x](B) &\equiv x > 0 \wedge y > x \wedge r > y \\
\mathbf{WP}[y > x](\dots, \dots) &\equiv (x > y \wedge y > 0 \wedge r > x) \vee \\
&\quad (x > 0 \wedge y > x \wedge r > y) \\
&\Leftarrow x \neq y \wedge x > 0 \wedge y > 0 \wedge r = x + y \\
&\equiv A
\end{aligned}$$

Orientierung:



Weitere Propagation von C durch den Kontrollfluss-Graphen komplettiert die lokal konsistente Annotation mit Zusicherungen \Leftarrow

Wir schließen:

- An den Programmpunkten 1 und 2 gelten die Zusicherungen $r > 0$ bzw. $r > x + y$.
- In jeder Iteration wird r kleiner, bleibt aber stets positiv.
- Folglich wird die Schleife nur endlich oft durchlaufen \implies das Programm terminiert \Leftarrow

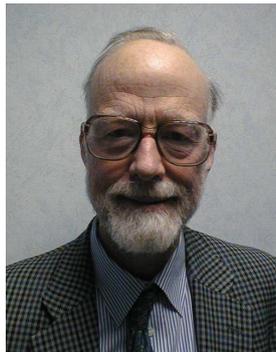
Allgemeines Vorgehen:

- Für jede vorkommende Schleife `while (b) s` erfinden wir eine neue Variable `r`.
- Dann transformieren wir die Schleife in:

```
r = e0;
while (b) {
  assert(r>0);
  s
  assert(r > e1);
  r = e1;
}
```

für geeignete Ausdrücke `e0, e1` :-)

3.5 Der Hoare-Kalkül



Tony Hoare, Microsoft Research, Cambridge

Idee (1):

- Organisiere den Korrektheitsbeweis entsprechend der Struktur des Programms !
- Modularisiere ihn so, dass Beweise für Teilprogramme zu einem Beweis für das gesamte Programm zusammen gesetzt werden können !

Idee (2):

Betrachte Aussagen der Form:

$$\{A\} \quad p \quad \{B\}$$

... das heißt:

Gilt **vor** der Ausführung des Programmstücks p Eigenschaft A und terminiert die Programm-Ausführung, dann gilt **nach** der Ausführung von p Eigenschaft B .

A : Vorbedingung
 B : Nachbedingung

Beispiele:

$$\{x > y\} \quad z = x-y; \quad \{z > 0\}$$

$$\{\mathbf{true}\} \quad \text{if } (x < 0) \quad x = -x; \quad \{x \geq 0\}$$

$$\{x > 7\} \quad \text{while } (x \neq 0) \quad x = x-1; \quad \{x = 0\}$$

$$\{\mathbf{true}\} \quad \text{while } (\mathbf{true}); \quad \{\mathbf{false}\}$$

Ableitung korrekter Aussagen:

- Bestimme eine Menge von **Grundaussagen** oder **Axiome**, die gültige Tripel für einfache Anweisungen beschreiben.
- Gib **Regeln** an, wie aus bereits abgeleiteten gültigen Tripeln weitere gültige Tripel hergeleitet werden können ...

Axiome:

leere Anweisung: $\{B\} \quad ; \quad \{B\}$

Zuweisung: $\{B[e/x]\} \quad x = e; \quad \{B\}$

Eingabe: $\{\forall x. B\} \quad x = \text{read}(); \quad \{B\}$

Ausgabe: $\{B\} \quad \text{write}(e); \quad \{B\}$

Regeln:

$$\text{Komposition: } \frac{\{A\} \text{ s } \{B\} \quad \{B\} \text{ ss } \{C\}}{\{A\} \text{ s ss } \{C\}}$$

- Oberhalb des Strichs stehen die Voraussetzungen für die Regel-Anwendung.
- Unterhalb des Strichs steht die Schlussfolgerung \therefore



Den Beweis eines Tripels für das zusammengesetzte Programm $s \text{ ss}$ setzen wir aus Beweisen für Aussagen über die Programmbestandteile s und ss zusammen \therefore

Beispiel:

Für: $x=x+2; z=z+x; i=i+1;$ konstruieren wir:

$$\frac{\begin{array}{c} \{z = (i+1)^2 \wedge x = 2i+1\} \text{ i=i+1; } \{z = i^2 \wedge x = 2i-1\} \\ \{z = i^2 \wedge x = 2i+1\} \text{ z=z+x; } \{z = (i+1)^2 \wedge x = 2i+1\} \\ \{z = i^2 \wedge x = 2i-1\} \text{ x=x+2; } \{z = i^2 \wedge x = 2i+1\} \quad \{z = i^2 \wedge x = 2i+1\} \text{ z=z+x; i=i+1; } \{z = i^2 \wedge x = 2i-1\} \end{array}}{\{z = i^2 \wedge x = 2i-1\} \text{ x=x+2; z=z+x; i=i+1; } \{z = i^2 \wedge x = 2i-1\}}$$

Regeln (Forts.):

$$\text{Bedingung: } \frac{\{\neg b \wedge A\} \text{ ss0 } \{B\} \quad \{b \wedge A\} \text{ ss1 } \{B\}}{\{A\} \text{ if } (b) \{ss0\} \text{ else } \{ss1\} \{B\}}$$

- Diese Regel entspricht unserer bisherigen Behandlung bedingter Verzweigungen ...
- Falls $\{A_0\} \text{ ss0 } \{B\}$ und $\{A_1\} \text{ ss1 } \{B\}$ gelten, können wir A auch wählen als

$$A \equiv (\neg b \Rightarrow A_0) \wedge (b \Rightarrow A_1) \quad \therefore$$

Beispiel:

Wir wollen

$$\{A\} \text{ if } (y>x) \text{ y=y-x; else } \text{ x=x-y; } \{B\}$$

beweisen für:

$$A \equiv x > 0 \wedge y > 0 \wedge x \neq y$$

$$B \equiv x > 0 \wedge y > 0$$

Dazu konstruieren wir:

$$\frac{\frac{\{x > 0 \wedge y > x\} \quad y=y-x; \quad \{B\}}{\{A \wedge y > x\} \quad y=y-x; \quad \{B\}} \quad \frac{\{x > y \wedge y > 0\} \quad x=x-y; \quad \{B\}}{\{A \wedge y \leq x\} \quad x=x-y; \quad \{B\}}}{\{A\} \quad \text{if } (y>x) \text{ } y=y-x;\text{else } x=x-y; \quad \{B\}}$$

Regeln (Forts.):

$$\text{Schleife: } \frac{\{b \wedge I\} \text{ ss } \{I\}}{\{I\} \text{ while } (b) \{ \text{ss} \} \{ \neg b \wedge I \}}$$

- Die Zusicherung I entspricht unserer **Schleifen-Invariante** :-)
- Bei Betreten des Rumpfs wissen wir zusätzlich, dass die Bedingung b gilt :-)
- Bei Verlassen der Schleife wissen wir zusätzlich, dass die Bedingung b **nicht** mehr gilt :-))

Regeln (Forts.):

- Gelegentlich **passen** die Vor- bzw. Nachbedingungen der bereits als korrekt bewiesenen Tripel nicht genau zusammen ...
- Dann kann man die Vorbedingung **verstärken** oder die Nachbedingung **abschwächen**:

$$\text{Abschwächung: } \frac{A \Rightarrow A_1 \quad \{A_1\} \text{ ss } \{B_1\} \quad B_1 \Rightarrow B}{\{A\} \text{ ss } \{B\}}$$

3.6 Behandlung von Prozeduren

Modulare Verifikation können wir benutzen, um die Korrektheit auch von Programmen mit Funktionen nachzuweisen :-)

Vereinfachung:

Wir betrachten nur

- **Prozeduren**, d.h. statische Methoden ohne Rückgabewerte;
- nur **globale Variablen**, d.h. alle Variablen sind ebenfalls `static`.
// werden wir später verallgemeinern :-)

Beispiel:

```
int a, b, x, y;          void mm() {
                        if (a>b) {
void main () {          x = a;
                        y = b;
    a = read();         } else {
    b = read();         y = a;
    mm();              x = b;
    write (x-y);       }
}                      }
```

Kommentar:

- Der Einfachheit halber haben wir alle Vorkommen von `static` gestrichen :-)
- Die Prozedur-Definitionen sind nicht rekursiv.
- Das Programm liest zwei Zahlen ein.
- Die Prozedur `minmax` speichert die größere in `x`, die kleinere in `y` ab.
- Die Differenz von `x` und `y` wird ausgegeben.
- Wir wollen zeigen, dass gilt:

$$\{a \geq b\} \text{ mm}(); \{a = x\}$$

Vorgehen:

- Für jede Prozedur $f()$ stellen wir ein Tripel bereit:

$$\{A\} f(); \{B\}$$

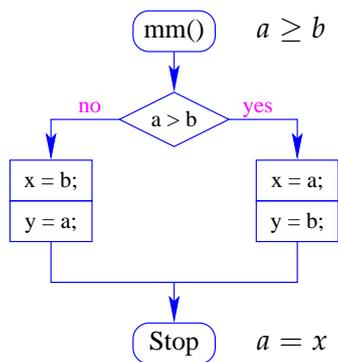
- Unter dieser **globalen Hypothese** H verifizieren wir, dass sich für jede Prozedurdefinition $\text{void } f() \{ ss \}$ zeigen lässt:

$$\{A\} ss \{B\}$$

- Wann immer im Programm ein Prozeduraufruf vorkommt, benutzen wir dabei die Tripel aus $H \dots$

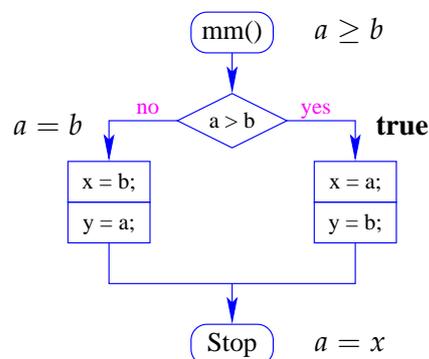
... im Beispiel:

Wir überprüfen:



... im Beispiel:

Wir überprüfen:



Diskussion:

- Die Methode funktioniert auch, wenn die Prozedur einen Rückgabewert hat: den können wir mit einer globalen Variable return simulieren, in die das jeweilige Ergebnis geschrieben wird :-)
- Es ist dagegen nicht offensichtlich, wie die Vor- und Nachbedingung für Prozeduraufrufe gewählt werden soll, wenn eine Funktion an **mehreren** Stellen aufgerufen wird ...
- Noch schwieriger wird es, wenn eine Prozedur **rekursiv** ist: dann hat sie potentiell unbeschränkt viele verschiedene Aufrufe !?

Beispiel:

```
int x, m0, m1, t;

void main () {
    x = read();
    m0 = 1; m1 = 1;
    if (x > 1) f();
    write (m1);
}

void f() {
    x = x-1;
    if (x>1) f();
    t = m1;
    m1 = m0+m1;
    m0 = t;
}
```

Kommentar:

- Das Programm liest eine Zahl ein.
- Ist diese Zahl höchstens 1, liefert das Programm 1 ...
- Andernfalls berechnet das Programm die **Fibonacci-Funktion** fib :-)
- Nach einem Aufruf von f enthalten die Variablen m0 und m1 jeweils die Werte fib($i - 1$) und fib(i) ...

Problem:

- Wir müssen in der Logik den i -ten vom $(i + 1)$ -ten Aufruf zu unterscheiden können :-)
- Das ist **einfacher**, wenn wir **logische Hilfsvariablen** $\underline{l} = l_1, \dots, l_n$ zur Verfügung haben, in denen wir (ausgewählte) Werte **vor** dem Aufruf retten können ...

Im Beispiel:

$\{A\} f(); \{B\}$ wobei

$$A \equiv x = l \wedge x > 1 \wedge m_0 = m_1 = 1$$

$$B \equiv l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}$$

Allgemeines Vorgehen:

- Wieder starten wir mit einer **globalen Hypothese** H , die für jeden Aufruf $f()$; nun eine Beschreibung:

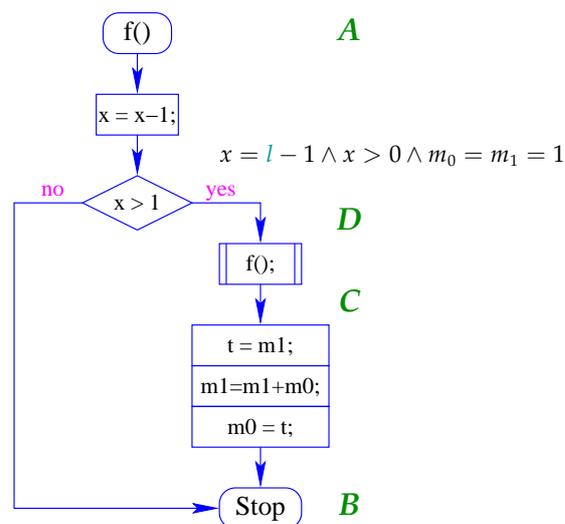
$\{A\} f(); \{B\}$

// sowohl A wie B können l_i enthalten :-)

- Unter dieser **globalen Hypothese** H verifizieren wir, dass für jede Funktionsdefinition `void f() { ss }` gilt:

$\{A\} ss \{B\}$

... im Beispiel:



- Wir starten von der Zusicherung für den Endpunkt:

$$B \equiv l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}$$

- Die Zusicherung C ermitteln wir mithilfe von **WP**[...] und **Abschwächung** ...

$$\begin{aligned}
& \text{WP}[[t=m_1; m_1=m_1+m_0; m_0=t;]] (B) \\
& \equiv l-1 > 0 \wedge m_1 + m_0 \leq 2^l \wedge m_1 \leq 2^{l-1} \\
& \leftarrow l-1 > 0 \wedge m_1 \leq 2^{l-1} \wedge m_0 \leq 2^{l-2} \\
& \equiv C
\end{aligned}$$

Frage:

Wie nutzen wir unsere globale Hypothese, um einen konkreten Prozeduraufruf zu behandeln ???

Idee:

- Die Aussage $\{A\} f(); \{B\}$ repräsentiert eine Wertetabelle für $f()$:-)
- Diese Wertetabelle können wir logisch repräsentieren als die Implikation:

$$\forall l. (A[h/\underline{x}] \Rightarrow B)$$

// h steht für eine Folge von Hilfsvariablen

Die Werte der Variablen \underline{x} vor dem Aufruf stehen in den Hilfsvariablen :-)

Beispiele:

Funktion: `void double () { x = 2*x; }`
Spezifikation: $\{x = l\} \text{double}(); \{x = 2l\}$
Tabelle: $\forall l. (h = l) \Rightarrow (x = 2l)$
 $\equiv (x = 2h)$

Für unsere Fibonacci-Funktion berechnen wir:

$$\begin{aligned}
& \forall l. (h > 1 \wedge h = l \wedge h_0 = h_1 = 1) \Rightarrow \\
& \quad l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1} \\
& \equiv (h > 1 \wedge h_0 = h_1 = 1) \Rightarrow m_1 \leq 2^h \wedge m_0 \leq 2^{h-1}
\end{aligned}$$

Ein anderes Paar (A_1, B_1) von Zusicherungen liefert ein gültiges Tripel $\{A_1\} f(); \{B_1\}$, falls die zugehörige Tabelle impliziert wird ...

Regel:

$$\text{Adaption: } \frac{\{A\} f(); \{B\} \in H \quad (\forall l. A[h/\underline{x}] \Rightarrow B) \Rightarrow (A_1[h/\underline{x}] \Rightarrow B_1)}{\{A_1\} f(); \{B_1\}}$$

Praktisch müssen wir zeigen:

$$\frac{\forall l. A[l/x] \Rightarrow B \quad A_1[l/x]}{B_1}$$

Beispiel: double()

$$\begin{aligned} A &\equiv x = l & B &\equiv x = 2l \\ A_1 &\equiv x \geq 3 & B_1 &\equiv x \geq 6 \end{aligned}$$

Wir überprüfen:

$$\frac{x = 2h \quad h \geq 3}{x \geq 6}$$

:-)

Bemerkungen:

Gültige Paare (A_1, B_1) erhalten wir z.B.,

- indem wir die logischen Variablen substituieren:

$$\frac{\{x = l\} \text{ double()}; \{x = 2l\}}{\{x = l - 1\} \text{ double()}; \{x = 2(l - 1)\}}$$

- indem wir die Vorbedingung verstärken bzw. die Nachbedingung abschwächen:

$$\frac{\{x = l\} \text{ double()}; \{x = 2l\}}{\{x > 0 \wedge x = l\} \text{ double()}; \{x = 2l\}}$$

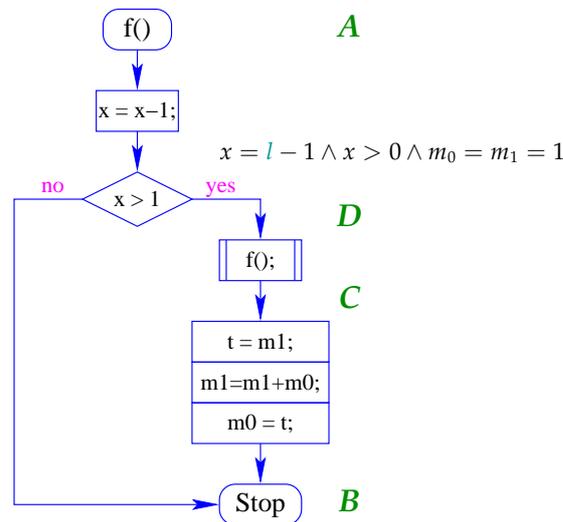
Anwendung auf Fibonacci:

Wir wollen beweisen: $\{D\} f(); \{C\}$

$$\begin{aligned} A &\equiv x > 1 \wedge l = x \wedge m_0 = m_1 = 1 \\ A[(l-1)/l] &\equiv x > 1 \wedge l-1 = x \wedge m_0 = m_1 = 1 \\ &\equiv D \end{aligned}$$

$$\begin{aligned} B &\equiv l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1} \\ B[(l-1)/l] &\equiv l-1 > 1 \wedge m_1 \leq 2^{l-1} \wedge m_0 \leq 2^{l-2} \\ &\equiv C \quad \quad \quad :-)) \end{aligned}$$

Orientierung:



Für die bedingte Verzweigung verifizieren wir:

$$\mathbf{WP}[[x>1]](B, D) \equiv (x \leq 1 \wedge l > 1 \wedge m_1 \leq 2^l \wedge m_0 \leq 2^{l-1}) \vee (x > 1 \wedge x = l - 1 \wedge m_1 = m_0 = 1)$$

$$\Leftarrow x > 0 \wedge x = l - 1 \wedge m_0 = m_1 = 1$$

:-))

3.7 Prozeduren mit lokalen Variablen

- Prozeduren $f()$ modifizieren globale Variablen.
- Die Werte der lokalen Variablen des Aufrufers **vor** und **nach** dem Aufruf sind unverändert :-)

Beispiel:

```
{int y= 17; double(); write(y);}
```

Vor und nach dem Aufruf von `double()` gilt: $y = 17$:-)

- Der Erhaltung der lokalen Variablen tragen wir **automatisch** Rechnung, wenn wir bei der Adaptionregel in

$$(\forall l. A[h/x] \Rightarrow B) \Rightarrow (A_1[h/x] \Rightarrow B_1)$$

beachten:

- Die Vor- und Nachbedingungen: $\{A\}, \{B\}$ für Prozeduren sprechen nur über globale Variablen !
- Die h werden nur für die **globalen** Variablen eingesetzt !!
- Als neuen Spezialfall der Adaption erhalten wir:

$$\frac{\{A\} f(); \{B\}}{\{A \wedge C\} f(); \{B \wedge C\}}$$

falls C nur über lokale Variablen des Aufrufers spricht :-)

3.8 Prozeduren mit Parametern

- Unsere Behandlung von lokalen Variablen ermöglicht uns, **call-by-value** Parameter-Übergabe von **Java** zu simulieren :-)
- Ein formaler Parameter y ist eine spezielle lokale Variable :-)
- Für die geregelte Parameter-Übergabe spendieren wir uns für y eine zusätzliche **globale Variable** Y .
- **Vor dem Aufruf** erhält Y den aktuellen Wert für y .
- **Am Anfang** des Rumpfs wird Y nach y kopiert ...

Aus einer Definition: void f(int y) { ss }
 wird: void f() {int y=Y; ss }

Aus einem Aufruf: f(e);
 wird: Y=e; f();

Um diese Transformation nicht syntaktisch ausführen zu müssen, führen wir die entsprechende Transformation nur in den Zusicherungen aus ...

- Als Spezifikation für f erlauben wir Tripel:

$$\{A\} f(\text{int } y) \{B\}$$

wobei A über globale Variablen und Y spricht :-)

- Zur Korrektheit verifizieren wir:

$$\{A \wedge Y = y\} \text{ss } \{B\}$$

- Bei einem Aufruf folgern wir:

$$\frac{\{A\} f(\text{int } y) \{B\} \in H}{\{A[e/Y]\} f(e); \{B\}}$$

Beispiel:

```
int ret;                        void main () {
                                  ret = read();
void inc(int y) {                inc(ret + 1);
    ret = ret + y;               write (ret);
}                                }
```

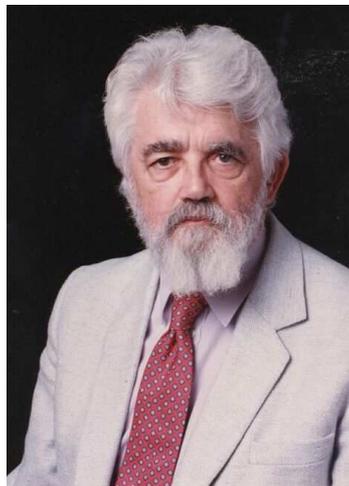
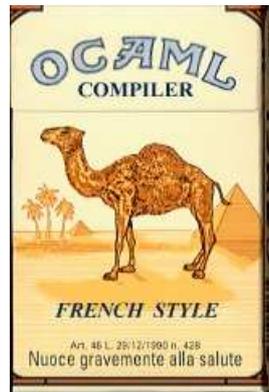
Spezifikation:
 Überprüfung:
 Anwendung:

$$\begin{aligned} & \{\text{ret} = l_1 \wedge Y = l_2\} \text{inc}(\text{int } y) \{\text{ret} = l_1 + l_2\} \\ & \{\text{ret} = l_1 \wedge Y = l_2 = y\} \text{ret} = \text{ret} + y; \{\text{ret} = l_1 + l_2\} \\ & \{\text{ret} = l_1 \wedge \text{ret} + 1 = l_2\} \text{inc}(\text{ret} + 1); \{\text{ret} = l_1 + l_2\} \\ & \{\text{ret} = l_1 \wedge l_1 + 1 = l_2\} \text{inc}(\text{ret} + 1); \{\text{ret} = 2l_1 + 1\} \end{aligned}$$

Abschluss:

- Jedes weitere Sprachkonstrukt erfordert neue Methoden zur Verifikation :-)
- Wie behandelt man dynamische Datenstrukturen, Objekte, Klassen, Vererbung ?
- Wie geht man mit Nebenläufigkeit, Reaktivität um ??
- Erlauben die vorgestellten Methoden alles zu beweisen \implies Vollständigkeit ?
- Wie weit lässt sich Verifikation automatisieren ?
- Wieviel Hilfe muss die Programmiererin und/oder die Verifiziererin geben ?

Funktionale Programmierung



John McCarthy, Stanford



Robin Milner, Edinburgh



Xavier Leroy, INRIA, Paris

4 Grundlagen

- Interpreter-Umgebung
- Ausdrücke
- Wert-Definitionen
- Komplexere Datenstrukturen
- Listen
- Definitionen (Forts.)
- Benutzer-definierte Datentypen

4.1 Die Interpreter-Umgebung

Der Interpreter wird mit `ocaml` aufgerufen...

```
seidl@linux:~> ocaml
Objective Caml version 3.08.2
#
```

Definitionen von Variablen, Funktionen, ... können direkt eingegeben werden :-)
Alternativ kann man sie aus einer Datei einlesen:

```
# #use "Hallo.ml";;
```

4.2 Ausdrücke

```
# 3+4;;
- : int = 7
# 3+
  4;;
- : int = 7
#
```

- Bei `#` wartet der Interpreter auf Eingabe.
- Das `;;` bewirkt Auswertung der bisherigen Eingabe.
- Das Ergebnis wird berechnet und mit seinem Typ ausgegeben.

Vorteil: Das Testen von einzelnen Funktionen kann stattfinden, ohne jedesmal neu zu übersetzen :-)

Vordefinierte Konstanten und Operatoren:

Typ	Konstanten: Beispiele	Operatoren
int	0 3 -7	+ - * / mod
float	-3.0 7.0	+. -. *. /.
bool	true false	not &&
string	"hallo"	^
char	'a' 'b'	

Typ	Vergleichsoperatoren
int	= <> < <= >= >
float	= <> < <= >= >
bool	= <> < <= >= >
string	= <> < <= >= >
char	= <> < <= >= >

```
# -3.0/.4.0;;
- : float = -0.75
# "So"^" "^"geht"^" "^"das";;
- : string = "So geht das"
# 1>2 || not (2.0<1.0);;
- : bool = true
```

4.3 Wert-Definitionen

Mit `let` kann man eine **Variable** mit einem Wert belegen. Die Variable behält diesen Wert **für immer** :-)

```
# let seven = 3+4;;
val seven : int = 7
# seven;;
- : int = 7
```

Achtung: Variablen-Namen werden **klein** geschrieben !!!

Eine erneute Definition für seven weist **nicht** seven einen neuen Wert zu, sondern erzeugt eine **neue** Variable mit Namen seven.

```
# let seven = 42;;
val seven : int = 42
# seven;;
- : int = 42
# let seven = "seven";;
val seven : string = "seven"
```

Die alte Definition wurde **unsichtbar** (ist aber trotzdem noch vorhanden :-)
Offenbar kann die neue Variable auch einen **anderen Typ** haben :-)

4.4 Komplexere Datenstrukturen

- **Paare:**

```
# (3,4);;
- : int * int = (3, 4)
# (1=2,"hallo");;
- : bool * string = (false, "hallo")
```

- **Tupel:**

```
# (2,3,4,5);;
- : int * int * int * int = (2, 3, 4, 5)
# ("hallo",true,3.14159);;
-: string * bool * float = ("hallo", true, 3.14159)
```

Simultane Definition von Variablen:

```
# let (x,y) = (3,4.0);;
val x : int = 3
val y : float = 4.

# val (3,y) = (3,4.0);;
val y : float = 4.0
```

Records: Beispiel:

```
# type person = {vor:string; nach:string; alter:int};;
type person = { vor : string; nach : string; alter : int; }
# let paul = { vor="Paul"; nach="Meier"; alter=24 };;
val paul : person = {vor = "Paul"; nach = "Meier"; alter = 24}
# let hans = { nach="kohl"; alter=23; vor="hans"};;
val hans : person = {vor = "hans"; nach = "kohl"; alter = 23}
# let hans1 = {alter=23; nach="kohl"; vor="hans"}
val hans1 : person = {vor = "hans"; nach = "kohl"; alter = 23}
# hans=hans1;;
- : bool = true
```

Bemerkung:

- ... Records sind Tupel mit benannten Komponenten, deren Reihenfolge irrelevant ist :-)
- ... Als neuer Typ muss ein Record vor seiner Benutzung mit einer `type`-Deklaration eingeführt werden.
- ... Typ-Namen und Record-Komponenten werden `klein` geschrieben :-)

Zugriff auf Record-Komponenten

... per Komponenten-Selektion:

```
# paul.vor;;
- : string = "Paul"
```

... mit Pattern Matching:

```
# let {vor=x;nach=y;alter=z} = paul;;
val x : string = "Paul"
val y : string = "Meier"
val z : int = 24
```

... und wenn einen nicht alles interessiert:

```
# let {vor=x} = paul;;
val x : string = "Paul"
```

Fallunterscheidung: `match` und `if`

```
match n
with 0 -> "Null"
     | 1 -> "Eins"
     | _ -> "Soweit kann ich nicht zaehlen!"

match e
with true  -> e1
     | false -> e2
```

Das zweite Beispiel kann auch so geschrieben werden (`-:`

```
if e then e1 else e2
```

Vorsicht bei redundanten und unvollständigen Matches!

```
# let n = 7;;
val n : int = 7
# match n with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 5, -13).
# match n
  with 0 -> "null"
       | 0 -> "eins"
       | _ -> "Soweit kann ich nicht zaehlen!";;
Warning: this match case is unused.
- : string = "Soweit kann ich nicht zaehlen!"
```

4.5 Listen

Listen werden mithilfe von `[]` und `::` konstruiert.

Kurzschreibweise: `[42; 0; 16]`

```
# let mt = [];;
val mt : 'a list = []
# let l1 = 1::mt;;
val l1 : int list = [1]
# let l = [1;2;3];;
val l : int list = [1; 2; 3]
# let l = 1::2::3::[];;
val l : int list = [1; 2; 3]
```

Achtung:

Alle Elemente müssen den **gleichen** Typ haben:

```
# 1.0::1::[];;
```

This expression has type int but is here used with type float

`tau list` beschreibt Listen mit Elementen vom Typ `tau` **:-)**

Der Typ `'a` ist eine **Typ-Variable**:

`[]` bezeichnet eine leere Liste für **beliebige** Element-Typen **:-))**

Pattern Matching auf Listen:

```
# match l
  with []      -> -1
       | x::xs -> x;;
-: int = 1
```

4.6 Definitionen von Funktionen

```
# let double x = 2*x;;
val double : int -> int = <fun>
# (double 3, double (double 1));;
- : int * int = (6,4)
```

- Nach dem Funktions-Namen kommen die Parameter.
- Der Funktionsname ist damit auch nur eine Variable, deren **Wert** eine Funktion ist **:-)**
- Alternativ können wir eine Variable einführen, deren **Wert** direkt eine Funktion beschreibt **...**

```
# let double = fun x -> 2*x;;
val double : int -> int = <fun>
```

- Diese Funktionsdefinition beginnt mit **fun**, gefolgt von den formalen Parametern.
- Nach `->` kommt die Berechnungsvorschrift.
- Die linken Variablen dürfen rechts benutzt werden **:-)**

Achtung:

Funktionen sehen die Werte der Variablen, die zu ihrem **Definitionszeitpunkt** sichtbar sind:

```
# let faktor = 2;;
val faktor : int = 2
# let double x = faktor*x;;
val double : int -> int = <fun>
# let faktor = 4;;
val faktor : int = 4
# double 3;;
- : int = 6
```

Achtung:

Eine Funktion ist ein **Wert**:

```
# double;;
- : int -> int = <fun>
```

Rekursive Funktionen:

Eine Funktion ist **rekursiv**, wenn sie sich selbst aufruft.

```
# let rec fac n = if n<2 then 1 else n * fac (n-1);;
val fac : int -> int = <fun>
# let rec fib = fun x -> if x <= 1 then 1
                        else fib (x-1) + fib (x-2);;
val fib : int -> int = <fun>
```

Dazu stellt **Ocaml** das Schlüsselwort **rec** bereit :-)

Rufen mehrere Funktionen sich gegenseitig auf, heißen sie **verschränkt rekursiv**.

```
# let rec even n = if n=0 then "even" else odd (n-1)
                  and odd n = if n=0 then "odd" else even (n-1);;
val even : int -> string = <fun>
val odd : int -> string = <fun>
```

Wir kombinieren ihre Definitionen mit dem Schlüsselwort **and** :-)

Definition durch Fall-Unterscheidung:

```
# let rec len = fun x -> match x
                        with [] -> 0
                             | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```

... kann kürzer geschrieben werden als:

```
# let rec len = function [] -> 0
                    | x::xs -> 1 + len xs;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
```

Fall-Unterscheidung bei mehreren Argumenten:

```
# let rec app x y = match x
                    with [] -> y
                         | x::xs -> x :: app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

... kann auch geschrieben werden als:

```
# let rec app = function [] -> fun y -> y
                    | x::xs -> fun y -> x::app xs y;;
val app : 'a list -> 'a list -> 'a list = <fun>
# app [1;2] [3;4];;
- : int list = [1; 2; 3; 4]
```

Lokale Definitionen

Definitionen können mit `let` lokal eingeführt werden:

```
# let    x = 5
  in let sq = x*x
    in   sq+sq;;
- : int = 50
# let facit n = let rec
  iter m yet = if m>n then yet
                else iter (m+1) (m*yet)
  in iter 2 1;;
val facit : int -> int = <fun>
```

4.7 Benutzerdefinierte Typen

Beispiel: Spielkarten

Wie kann man die Farbe und den Wert einer Karte spezifizieren?

1. Idee: Benutze Paare von Strings und Zahlen, z.B.

```
("Karo",10)  ≡  Karo Zehn
("Kreuz",12) ≡  Kreuz Bube
("Pik",1)    ≡  Pik As
```

Nachteile:

- Beim Test auf eine Farbe muss immer ein String-Vergleich stattfinden
→ ineffizient!
- Darstellung des Buben als 12 ist nicht intuitiv
→ unleserliches Programm!
- Welche Karte repräsentiert das Paar ("Kaor",1)?
(Typfehler werden vom Compiler nicht bemerkt)

Besser: Aufzählungstypen von **Ocaml**.

Beispiel: Spielkarten

2. Idee: Aufzählungstypen

```
# type farbe = Karo | Herz | Pik | Kreuz;;
type farbe = Karo | Herz | Pik | Kreuz
# type wert = Neun | Bube | Dame | Koenig | Zehn | As;;
type wert = Neun | Bube | Dame | Koenig | Zehn | As
# Kreuz;;
- : farbe = Kreuz
# let pik_bube = (Pik,Bube);;
val pik_bube : farbe * wert = (Pik, Bube)
```

Vorteile:

- Darstellung ist intuitiv.
- Tippfehler werden erkannt:

```
# (Kaor,As);;
Unbound constructor Kaor
```

- Die interne Repräsentation ist **effizient** :-)

Bemerkungen:

- Durch **type** wird ein **neuer Typ** definiert.
- Die Alternativen heißen **Konstruktoren** und werden durch **|** getrennt.
- Jeder Konstruktor wird groß geschrieben und ist **eindeutig** einem Typ zugeordnet.

Aufzählungstypen (cont.)

Konstruktoren können verglichen werden:

```
# Kreuz < Karo;;
- : bool = false;;
# Kreuz > Karo;;
- : bool = true;;
```

Pattern Matching auf Konstruktoren:

```
# let istTrumpf = function
    (Karo,_)    -> true
  | (_,Bube)   -> true
  | (_,Dame)   -> true
  | (Herz,Zehn) -> true
  | (_,_)      -> false;;

val istTrumpf : farbe * wert -> bool = <fun>
```

Damit ergibt sich z.B.:

```
# istTrumpf (Karo,As);;
- : bool = true
# istTrumpf (Pik,Koenig);;
- : bool = false
```

Eine andere nützliche Funktion:

```
# let string_of_farbe = function
      Karo  -> "Karo"
    | Herz  -> "Herz"
    | Pik   -> "Pik"
    | Kreuz -> "Kreuz";;
val string_of_farbe : farbe -> string = <fun>
```

Beachte:

Die Funktion `string_of_farbe` wählt für eine Farbe in **konstanter Zeit** den zugehörigen String aus (der Compiler benutzt – hoffentlich – **Sprungtabellen** :-)
Jetzt kann **Ocaml** schon fast Karten spielen:

```
# let sticht = function
      ((Herz,Zehn),_)      -> true
    | (_, (Herz,Zehn))    -> false
    | ((f1,Dame),(f2,Dame)) -> f1 > f2
    | (_, Dame),_        -> true
    | (_, (_, Dame))     -> false
    | ((f1,Bube),(f2,Bube)) -> f1 > f2
    | (_, Bube),_        -> true
    | (_, (_, Bube))     -> false
    | ((Karo,w1),(Karo,w2)) -> w1 > w2
    | ((Karo,_),_)       -> true
    | (_, (Karo,_))      -> false
    | ((f1,w1),(f2,w2))  -> if f1=f2 then w1 > w2
                              else false;;
```

```

...
# let nimm (karte2,karte1) =
    if sticht (karte2,karte1) then karte2 else karte1;;

# let stich (karte1,karte2,karte3,karte4) =
    nimm (karte4, nimm (karte3, nimm (karte2,karte1))));;

# stich ((Herz,Koenig),(Karo,As), (Herz,Bube),(Herz,As));;
- : farbe * wert = (Herz, Bube)
# stich((Herz,Koenig),(Pik,As), (Herz,Koenig),(Herz,As));;
- : farbe * wert = (Herz, As)

```

Summentypen

Summentypen sind eine Verallgemeinerung von Aufzählungstypen, bei denen die Konstruktoren **Argumente** haben.

Beispiel: Hexadezimalzahlen

```

type hex = Digit of int | Letter of char;;
let char2dez c = if c >= 'A' && c <= 'F'
    then (Char.code c)-55
    else if c >= 'a' && c <= 'f'
    then (Char.code c)-87
    else -1;;
let hex2dez = function
    Digit n -> n
    | Letter c -> char2dez c;;

```

Char ist ein **Modul**, der Funktionalität für **char** sammelt :-)

Ein Konstruktor, der mit `type t = Con of <typ> | ...` definiert wurde, hat die Funktionalität `Con : <typ> -> t`
 — muss aber stets **angewandt** vorkommen ...

```

# Digit;;
The constructor Digit expects 1 argument(s),
but is here applied to 0 argument(s)
# let a = Letter 'a';;
val a : hex = Letter 'a'
# Letter 1;;
This expression has type int but is here used with type char
# hex2dez a;;
- : int = 10

```

Datentypen können auch **rekursiv** sein:

```

type folge = Ende | Dann of (int * folge)

# Dann (1, Dann (2, Ende));;
- : folge = Dann (1, Dann (2, Ende))

```

Beachte die Ähnlichkeit zu Listen ;-)

Rekursive Datentypen führen wieder zu rekursiven Funktionen:

```

# let rec n_tes = function
    (_,Ende) -> -1
  | (0,Dann (x,_)) -> x
  | (n,Dann (_, rest)) -> n_tes (n-1,rest);;
val n_tes : int * folge -> int = <fun>

# n_tes (4, Dann (1, Dann (2, Ende)));;
- : int = -1
# n_tes (2, Dann (1, Dann(2, Dann (5, Dann (17, Ende)))));;
- : int = 5

```

Anderes Beispiel:

```

# let rec runter = function
    0 -> Ende
  | n -> Dann (n, runter (n-1));;
val runter : int -> folge = <fun>

# runter 4;;
- : folge = Dann (4, Dann (3, Dann (2, Dann (1, Ende))));;
# runter -1;;
Stack overflow during evaluation (looping recursion?).

```

Der Options-Datentyp

Ein eingebauter Datentyp in **Ocaml** ist `option` mit den zwei Konstruktoren `None` und `Some`.

```

# None;;
- : 'a option = None
# Some 10;
- : int option = Some 10

```

Er wird häufig benutzt, wenn eine Funktion nicht für alle Eingaben eine Lösung berechnet:

```

# let rec n_tes = function
    (n,Ende) -> None
    | (0, Dann (x,_)) -> Some x
    | (n, Dann (_,rest)) -> n_tes (n-1,rest);;
val n_tes : int * folge -> int option = <fun>

# n_tes (4,Dann (1, Dann (2, Ende)));;
- : int option = None
# n_tes (2, Dann (1, Dann (2, Dann (5, Dann (17, Ende)))));;
- : int option = Some 5

```

5 Funktionen – näher betrachtet

- Rekursionsarten
- Funktionen höherer Ordnung
 - Currying
 - Partielle Anwendung
- Polymorphe Funktionen
- Polymorphe Datentypen
- Namenlose Funktionen
- Unendliche Datenstrukturen

5.1 Rekursionsarten

Je nach Art der rekursiven Aufrufe unterscheiden wir folgende Arten von Rekursion:

1. **End-Rekursion:** Jeder rekursive Aufruf liefert gleichzeitig den Rückgabewert.

```

let rec fac1 = function
    (1,acc) -> acc
    | (n,acc) -> fac1 (n-1,n*acc);;

```

```

let rec loop x = if x<2 then x
                 else if x mod 2 = 0 then loop (x/2)
                 else loop (3*x+1);;

```

2. **Repetitive Rekursion:** Es gibt in jedem Zweig höchstens einen rekursiven Aufruf.

```

let rec fac = function
    1 -> 1
    | n -> n * fac (n-1);;

```

3. Allgemeine Rekursion:

```
let rec fib = function
  0 -> 1
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2);;

let rec f n = if n<2 then n
             else f (f (n/2));;
```

Das **beste Speicherverhalten** hat die End-Rekursion.

Es wird zur allgemeinen Rekursion hin immer schlechter.

Der typische Fall von repetitiver Rekursion hat die folgende Form:

```
let rec f x = if t(x) then v else e(x,f(g(x)))
```

Beispiel: Fakultät

```
let rec fac x = if x=1 then 1 else x * fac (x-1)
```

Hier ist $t(x) \equiv x=1$, $v \equiv 1$, $g(x) \equiv x-1$ und $e(x,y) \equiv x*y$.

Wie wir gesehen haben, gibt es auch eine end-rekursive Variante:

```
let rec fac1 (x,a) = if x=1 then a else fac1 (x-1,x*a)
```

Im allgemeinen Fall kann man f ersetzen durch:

```
let rec f1 (x,a) = if t(x) then a else f1 (g(x),e(x,a))
```

Dann ist $f1(x,v)$ gleich $f\ x$ — aber nur dann,
wenn e eine **assoziative, kommutative** Funktion ist:

$$n*(\dots*(2*1)) = 1*(2*(\dots*n))$$

Der zusätzliche Parameter a heißt auch **Akkumulator**.

Vorsicht bei Listenfunktionen:

```

let rec f x      = if x=0 then [] else x :: f (x-1)
let rec f1 (x,a) = if x=0 then a  else f1 (x-1,x::a)

```

Der Listenkonstruktor `::` ist weder kommutativ noch assoziativ.
Deshalb berechnen `f x` und `f1 (x, [])` nicht den gleichen Wert:

```

# f 5;;
- : int list = [5; 4; 3; 2; 1]
# f1 (5,[]);;
- : int list = [1; 2; 3; 4; 5]

```

Selbst allgemeine Rekursion kann manchmal linearisiert werden:

```

let rec fib = function
  | 0 -> 1
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2);;

let fib1 n =
  let rec iter x (m0,m1) =
    if x<=1 then m1 else iter (x-1) (m1,m0+m1)
  in iter n (1,1)

```

Das lässt sich aber nicht **so einfach** verallgemeinern :-)

5.2 Funktionen höherer Ordnung

Betrachte die beiden Funktionen

```

let f (a,b) = a+b+1;;
let g a b   = a+b+1;;

```

Auf den ersten Blick scheinen sich `f` und `g` nur in der Schreibweise zu unterscheiden.
Aber sie haben einen **anderen Typ**:

```

# f;;
- : int * int -> int = <fun>
# g;;
- : int -> int -> int = <fun>

```

- Die Funktion f hat ein Argument, welches aus dem **Paar** (a, b) besteht. Der Rückgabewert ist $a+b+1$.
- g hat ein Argument a vom Typ `int`. Das Ergebnis einer Anwendung auf a ist **wieder eine Funktion**, welche, angewendet auf ein weiteres Argument b , das Ergebnis $a+b+1$ liefert:

```
# f (3,5);;
- : int = 9
# let g1 = g 3;;
val g1 : int -> int = <fun>
# g1 5;;
- : int = 9
```



Haskell B. Curry, 1900–1982

Das Prinzip heißt nach seinem Erfinder Haskell B. Curry **Currying**.

- g heißt Funktion **höherer Ordnung**, weil ihr Ergebnis wieder eine Funktion ist.
- Die Anwendung von g auf ein Argument heißt auch **partiell**, weil das Ergebnis nicht vollständig ausgewertet ist, sondern eine weitere Eingabe erfordert.

Das Argument einer Funktion kann auch wieder selbst eine Funktion sein:

```
# let apply f a b = f (a,b);;
val apply ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
...
```

```
...
# let plus (x,y) = x+y;;
val plus : int * int -> int = <fun>
# apply plus;;
- : int -> int -> int = <fun>
# let plus2 = apply plus 2;;
val plus2 : int -> int = <fun>
# let plus3 = apply plus 3;;
val plus3 : int -> int = <fun>
# plus2 (plus3 4);;
- : int = 9
```

5.3 Funktionen als Daten

Funktionen sind **Daten** und können daher in Datenstrukturen vorkommen:

```
# ((+), plus3) ;
- : (int -> int -> int) * (int -> int) = (<fun>, <fun>);;
# let rec plus_list = function
    [] -> []
  | x::xs -> (+) x :: plus_list xs;;
val plus_list : int list -> (int -> int) list = <fun>
# let l = plus_list [1;2;3];;
val l : (int -> int) list = [<fun>; <fun>; <fun>]
```

```
// (+) : int -> int -> int ist die Funktion zum Operator +
```

...

```
# let do_add n =
    let rec add_list = function
        [] -> []
      | f::fs -> f n :: add_list fs
    in add_list ;;
val do_add : 'a -> ('a -> 'b) list -> 'b list = <fun>
# do_add 5 l;;
- : int list = [6;7;8]

# let rec sum = function
    [] -> 0
  | f::fs -> f (sum fs);;
val sum : (int -> int) list -> int = <fun>
# sum l;;
- : int = 6
```

5.4 Einige Listen-Funktionen

```
let rec map f = function
  [] -> []
  | x::xs -> f x :: map f xs

let rec fold_left f a = function
  [] -> a
  | x::xs -> fold_left f (f a x) xs

let rec fold_right f = function
  [] -> fun b -> b
  | x::xs -> fun b -> f x (fold_right f xs b)

let rec find_opt f = function
  [] -> None
  | x::xs -> if f x then Some x
             else find_opt f xs
```

Beachte:

- Diese Funktionen abstrahieren von dem Verhalten der Funktion f . Sie spezifizieren das Rekursionsverhalten gemäß der Listenstruktur, unabhängig von den Elementen der Liste.
- Daher heißen solche Funktionen **Rekursions-Schemata** oder (Listen-)**Funktionale**.
- Listen-Funktionale sind unabhängig vom Typ der Listenelemente. (Diesen muss nur die Funktion f kennen :-)
- Funktionen, die gleich strukturierte Eingaben verschiedenen Typs verarbeiten können, heißen **polymorph**.

5.5 Polymorphe Funktionen

Das **Ocaml**-System inferiert folgende Typen für diese Funktionale:

```
map : ('a -> 'b) -> 'a list -> 'b list
fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
find_opt : ('a -> bool) -> 'a list -> 'a option
```

- 'a und 'b sind **Typvariablen**. Sie können durch jeden Typ ersetzt (**instanziiert**) werden (aber an jedem Vorkommen durch den gleichen Typ).
- Durch partielle Anwendung auf eine Funktion können die Typvariablen instanziiert werden:

```
# Char.chr;;
val : int -> char = <fun>
# map Char.chr;;
- : int list -> char list = <fun>

# fold_left (+);;
val it : int -> int list -> int = <fun>
```

- Wenn man einem Funktional eine polymorphe Funktion als Argument gibt, ist das Ergebnis wieder polymorph:

```
# let cons_r xs x = x::xs;;
val cons_r : 'a list -> 'a -> 'a list = <fun>
# let rev l = fold_left cons_r [] l;;
val rev : 'a list -> 'a list = <fun>
# rev [1;2;3];;
- : int list = [3; 2; 1]
# rev [true;false;false];;
- : bool list = [false; false; true]
```

Ein paar der einfachsten polymorphen Funktionen:

```
let compose f g x = f (g x)
let twice f x = f (f x)
let iter f g x = if g x then x else iter f g (f x);;

val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
val iter : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a = <fun>

# compose neg neg;;
- : bool -> bool = <fun>
# compose neg neg true;;
- : bool = true;;
# compose Char.chr plus2 65;;
- : char = 'C'
```

5.6 Polymorphe Datentypen

Man sich auch selbst polymorphe Datentypen definieren:

```
type 'a tree = Leaf of 'a
             | Node of ('a tree * 'a tree)
```

- `tree` heißt **Typkonstruktor**, weil er aus einem anderen Typ (seinem Parameter `'a`) einen neuen Typ erzeugt.
- Auf der rechten Seite dürfen nur die Typvariablen vorkommen, die auf der linken Seite als Argument für den Typkonstruktor stehen.
- Die Anwendung der Konstruktoren auf Daten instanziiert die Typvariable(n):

```
# Leaf 1;;
- : int tree = Leaf 1
# Node (Leaf ('a',true), Leaf ('b',false));;
- : (char * bool) tree = Node (Leaf ('a', true),
                             Leaf ('b', false))
```

Funktionen auf polymorphen Datentypen sind typischerweise wieder polymorph ...

```
let rec size = function
  Leaf _      -> 1
  | Node(t,t') -> size t + size t'

let rec flatten = function
  Leaf x      -> [x]
  | Node(t,t') -> flatten t @ flatten t'

let flatten1 t = let rec doit = function
  (Leaf x, xs) -> x :: xs
  | (Node(t,t'), xs) -> let xs = doit (t',xs)
                        in doit (t,xs)
  in doit (t,[])
end
...

...
val size : 'a tree -> int = <fun>
val flatten : 'a tree -> 'a list = <fun>
val flatten1 : 'a tree -> 'a list = <fun>

# let t = Node(Node(Leaf 1,Leaf 5),Leaf 3);;
val t : int tree = Node (Node (Leaf 1, Leaf 5), Leaf 3)

# size t;;
- : int = 3
# flatten t;;
val : int list = [1;5;3]
# flatten1 t;;
val : int list = [1;5;3]
```

5.7 Anwendung: Queues

Gesucht:

Datenstruktur 'a queue, die die folgenden Operationen unterstützt:

```
enqueue : 'a -> 'a queue -> 'a queue
dequeue : 'a queue -> 'a option * 'a queue
is_empty : 'a queue -> bool
queue_of_list : 'a list -> 'a queue
list_of_queue : 'a queue -> 'a list
```

1. Idee:

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial :-)

- Entnehmen heißt Zugreifen auf das erste Element der Liste:

```
let dequeue = function
  [] -> (None, [])
  | x::xs -> (Some x, xs)
```

- Einfügen bedeutet hinten anhängen:

```
let enqueue x xs = xs @ [x]
```

Diskussion:

- Der Operator `@` konkateniert zwei Listen.
- Die Implementierung ist sehr einfach :-)
- Entnehmen ist sehr billig :-)
- Einfügen dagegen kostet so viele rekursive Aufrufe von `@` wie die Schlange lang ist :-(
- Geht das nicht besser ??

2. Idee (Fortsetzung):

- Einfügen erfolgt deshalb in der zweiten Liste:

```
let enqueue x (Queue (first,last)) =  
    Queue (first, x::last)
```

- Entnahme bezieht sich dagegen auf die erste Liste :-)
Ist diese aber leer, wird auf die zweite zugegriffen ...

```
let dequeue = function  
    Queue ([],last) -> (match List.rev last  
        with [] -> (None, Queue ([],[]))  
             | x::xs -> (Some x, Queue (xs,[])))  
    | Queue (x::xs,last) -> (Some x, Queue (xs,last))
```

Diskussion:

- Jetzt ist Einfügen billig :-)
- Entnehmen dagegen kann so teuer sein, wie die Anzahl der Elemente in der zweiten Liste :-)
- Gerechnet aber auf jede Einfügung, fallen nur **konstante** Zusatzkosten an !!!

⇒ **amortisierte Kostenanalyse**

5.8 Namenlose Funktionen

Wie wir gesehen haben, sind Funktionen **Daten**. Daten, z.B. [1;2;3] können verwendet werden, ohne ihnen einen Namen zu geben. Das geht auch für Funktionen:

```
# fun x y z -> x+y+z;;  
- : int -> int -> int -> int = <fun>
```

- **fun** leitet eine **Abstraktion** ein.
Der Name kommt aus dem **λ -Kalkül**.
- **->** hat die Funktion von **=** in Funktionsdefinitionen.
- **Rekursive** Funktionen können so nicht definiert werden,
denn ohne Namen kann eine Funktion nicht in ihrem Rumpf vorkommen :-)



Alonzo Church, 1903–1995

- Um Pattern Matching zu benutzen, kann man `match ... with` für das entsprechende Argument einsetzen.
- Bei einem einzigen Argument bietet sich `function` an ...

```
# function None    -> 0
      | Some x    -> x*x+1;;
- : int option -> int = <fun>
```

Namenlose Funktionen werden verwendet, wenn sie nur **einmal** im Programm vorkommen. Oft sind sie **Argument für Funktionale**:

```
# map (fun x -> x*x) [1;2;3];;
- : int list = [1; 4; 9]
```

Oft werden sie auch benutzt, um eine Funktion **als Ergebnis** zurückzuliefern:

```
# let make_undefined () = fun x -> None;;
val make_undefined : unit -> 'a -> 'b option = <fun>
# let def_one (x,y) = fun x' -> if x=x' then Some y
      else None;;
val def_one : 'a * 'b -> 'a -> 'b option = <fun>
```

5.9 Unendliche Datenstrukturen

Wie können wir die Menge aller natürlichen Zahlen darstellen?

1. Versuch:

```
let rec all_nats n = n :: all_nats (n+1);;
```

`all_nats` berechnet eine unendliche Liste.

Sie versucht es zumindest:

die unendliche Berechnung kommt nie zum Ende `:-(`

2. Versuch:

Wir benutzen **Abstraktionen** `:-)`

Dazu definieren wir zunächst einen Datentyp:

```
type 'a inflist = Lazy of 'a * (unit -> 'a inflist)
```

Dieser Datentyp kann keine endlich großen Daten repräsentieren, denn es fehlt der Konstruktor für das Ende der Liste `:-?`

Das zweite Argument für `Lazy` (Rest der Liste) ist vom Typ:

```
(unit -> 'a inflist)
```

Es ist also eine Funktion, die, angewendet auf `() : unit`, den Rest der Liste ergibt `:-)`

Das nutzen wir aus, um unendliche Listen zu repräsentieren:

```
# let rec all_nats n = Lazy (n, fun () -> all_nats (n+1));;
val all_nats : int -> int inflist = <fun>
```

Aufrufe der Funktion `all_nats` terminieren:

```
# let nats = all_nats 0;;
val nats : 'a inflist = Lazy (0, <fun>)
```

Um auf die Komponenten von `inflist`'s zuzugreifen definieren wir:

```
# let head (Lazy (x,f)) = x;;
val head : 'a inflist -> 'a = <fun>
# let tail (Lazy (_,f)) = f();;
val tail : 'a inflist -> 'a inflist = <fun>
let rec nth = function (0, Lazy (x,_)) -> x
                    | (n, Lazy (_,f)) -> nth (n-1, f());;
```

Achtung:

Der Rest der Liste in `tail` und `nth` muss auf den Wert `()` angewendet werden. Erst dadurch wird das nächste Element (und die Funktion, die den weiteren Rest der Liste darstellt) erzeugt.

```
# head nats;;
- : int = 0
# tail nats;;
- : int inflist = Lazy (1,<fun>)
# head (tail (tail (tail nats)));;
- : int = 3
```

`()` ist der einzige Wert vom Typ `unit`.

Er wird häufig zusammen mit Abstraktionen für die **Verzögerung** von Berechnungen verwendet.

Extrahieren einer endlichen Teilliste:

```
# let rec first_n = function
    (0,_)          -> []
  | (n, Lazy (x,f)) -> x :: first_n (n-1,f());;

# first_n (10,nats);;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Funktionale auf unendlichen Listen:

```
let rec inf_map h (Lazy (x,f)) =
  Lazy (h x, fun () -> inf_map h (f()));;
let rec inf_filter p (Lazy (x,f)) =
  if p x then Lazy (x, fun () -> inf_filter p (f()))
  else inf_filter p (f());;
```

Dann erhalten wir etwa:

```
val inf_map : ('a -> 'b)
  -> 'a inflist -> 'b inflist
val inf_filter : ('a -> bool)
  -> 'a inflist -> 'a inflist
```

Jetzt können wir z.B. die unendliche Liste aller geraden Zahlen oder aller Quadratzahlen berechnen:

```
# first_n (10,inf_filter (fun x -> x mod 2=0) nats);;
- : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18]
# first_n (7,inf_map (fun x -> x*x) nats);;
- : int list = [0; 1; 4; 9; 16; 25; 36]
```

So können wir auch die Liste aller Primzahlen berechnen (Sieb des Eratosthenes):

```
# let all_primes () =
  let rec sieve (Lazy (n,f)) =
    Lazy (n, (fun () -> sieve (inf_filter
      (fun x -> x mod n<>0)
      (f()))))
  in sieve (all_nats 2);;

# first_n (10, all_primes());;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
```

6 Größere Anwendung: Balancierte Bäume

Erinnerung: Sortiertes Array:

2	3	5	7	11	13	17
---	---	---	---	----	----	----

Eigenschaften:

- **Sortierverfahren** gestatten Initialisierung mit $\approx n \cdot \log(n)$ vielen Vergleichen :-)
// n = Größe des Arrays
- **Binäre Suche** erlaubt Auffinden eines Elements mit $\approx \log(n)$ vielen Vergleichen :-)

- Arrays unterstützen weder Einfügen noch Löschen einzelner Elemente :-)

Gesucht:

Datenstruktur 'a d, die dynamisch eine Folge von Elementen sortiert hält, d.h. die die Operationen unterstützt:

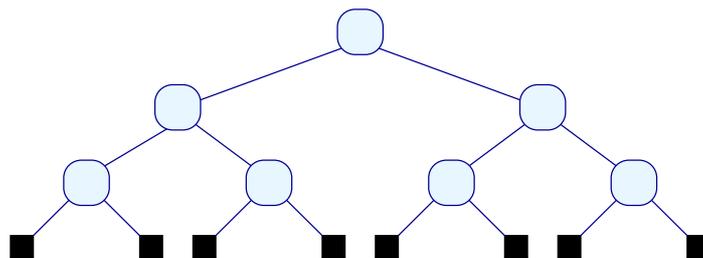
```

insert :      'a -> 'a d -> 'a d
delete :     'a -> 'a d -> 'a d
extract_min : 'a d -> 'a option * 'a d
extract_max : 'a d -> 'a option * 'a d
extract : 'a * 'a -> 'a d -> 'a list * 'a d
list_of_d :   'a d -> 'a list
d_of_list :   'a list -> 'a d

```

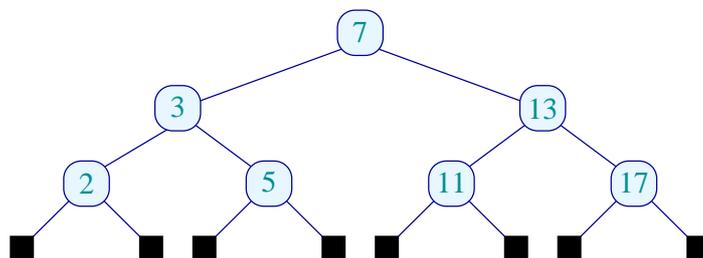
1. Idee:

Benutze **balancierte Bäume** ...



1. Idee:

Benutze **balancierte Bäume** ...



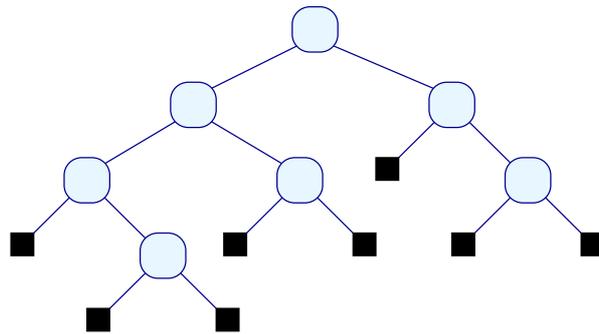
Diskussion:

- Wir speichern unsere Daten in den **inneren** Knoten :-)
- Ein **Binärbaum** mit n Blättern hat $n - 1$ innere Knoten :-)
- Zum Auffinden eines Elements müssen wir mit allen Elementen auf einem Pfad vergleichen ...
- Die **Tiefe** eines Baums ist die maximale Anzahl innerer Knoten auf einem Pfad von der Wurzel zu einem Blatt.
- Ein **vollständiger balancierter** Binärbaum mit $n = 2^k$ Blättern hat Tiefe $k = \log(n)$:-)
- Wie fügen wir aber weitere Elemente ein ??
- Wie können wir Elemente löschen ???

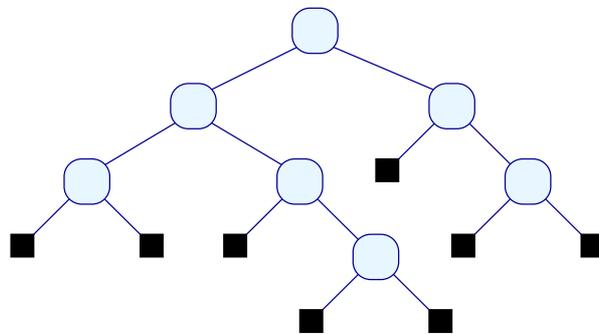
2. Idee:

- Statt balancierter Bäume benutzen wir **fast** balancierte Bäume ...
- An jedem Knoten soll die Tiefe des rechten und linken Teilbaums **ungefähr** gleich sein :-)
- Ein **AVL**-Baum ist ein Binärbaum, bei dem an jedem inneren Knoten die Tiefen des rechten und linken Teilbaums maximal um 1 differieren ...

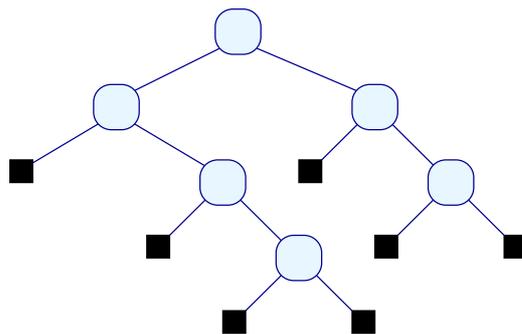
Ein AVL-Baum:

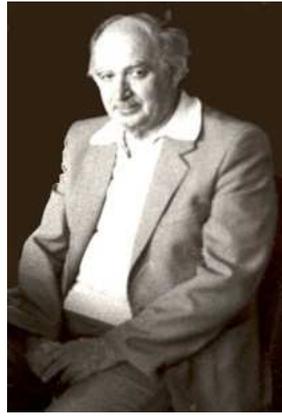


Ein AVL-Baum:



Kein AVL-Baum:





G.M. Adelson-Velskij, 1922

E.M. Landis, Moskau, 1921-1997

Wir vergewissern uns:

(1) Jeder AVL-Baum der Tiefe $k > 0$ hat mindestens

$$\text{fib}(k) \geq A^{k-1}$$

Knoten für $A = \frac{\sqrt{5} + 1}{2}$ // goldener Schnitt :-)

(2) Jeder AVL-Baum mit $n > 0$ inneren Knoten hat Tiefe maximal

$$\frac{1}{\log(A)} \cdot \log(n) + 1$$

Beweis: Wir zeigen nur (1) :-)

Sei $N(k)$ die minimale Anzahl der inneren Knoten eines AVL-Baums der Tiefe k .

Induktion nach der Tiefe $k > 0$:-)

$k = 1$: $N(1) = 1 = \text{fib}(1) = A^0$:-)

$k = 2$: $N(2) = 2 = \text{fib}(2) \geq A^1$:-)

$k > 2$: Gelte die Behauptung bereits für $k - 1$ und $k - 2$...

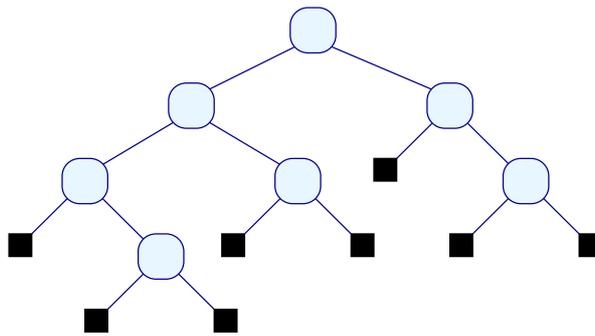
$$\begin{aligned} \implies N(k) &= N(k-1) + N(k-2) + 1 \\ &\geq \text{fib}(k-1) + \text{fib}(k-2) \\ &= \text{fib}(k) \end{aligned} \quad \text{:-)}$$

$$\begin{aligned}
 \text{fib}(k) &= \text{fib}(k-1) + \text{fib}(k-2) \\
 &\geq A^{k-2} + A^{k-3} \\
 &= A^{k-3} \cdot (A+1) \\
 &= A^{k-3} \cdot A^2 \\
 &= A^{k-1}
 \end{aligned}
 \quad :-\))$$

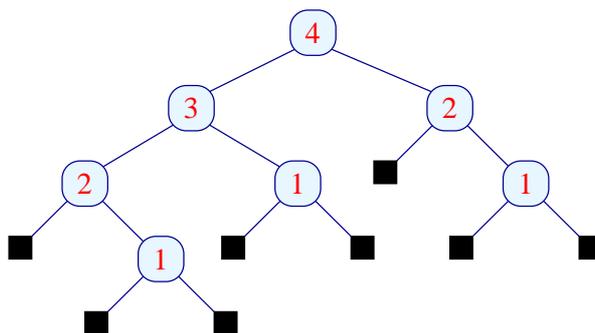
2. Idee (Fortsetzung)

- Fügen wir ein weiteres Element ein, könnte die **AVL-Eigenschaft** verloren gehen :-)
- Entfernen wir ein Element ein, könnte die **AVL-Eigenschaft** verloren gehen :-)
- Dann müssen wir den Baum so umbauen, dass die **AVL-Eigenschaft** wieder hergestellt wird :-)
- Dazu müssen wir allerdings an jedem inneren Knoten wissen, wie tief die linken bzw. rechten Teilbäume sind ...

Repräsentation:



Repräsentation:



3. Idee:

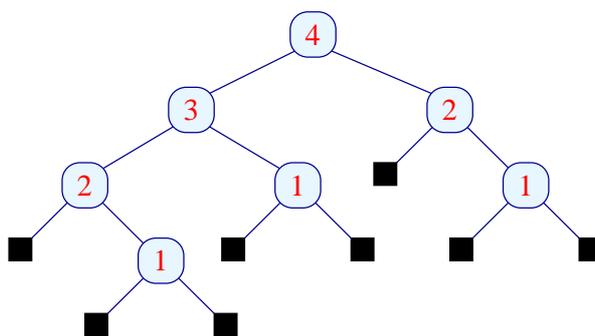
- Anstelle der **absoluten** Tiefen speichern wir an jedem Knoten nur, ob die Differenz der Tiefen der Teilbäume negativ, positiv oder ob sie gleich sind !!!
- Als Datentyp definieren wir deshalb:

```

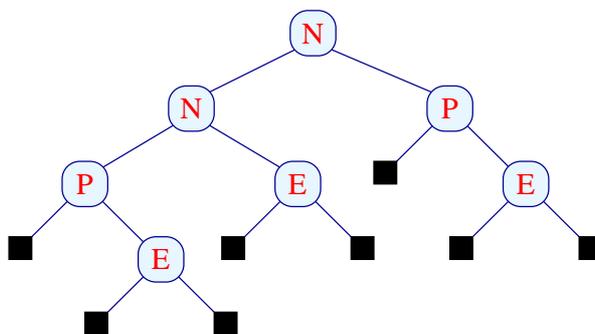
type 'a avl = Null
           | Neg of 'a avl * 'a * 'a avl
           | Pos of 'a avl * 'a * 'a avl
           | Eq  of 'a avl * 'a * 'a avl

```

Repräsentation:



Repräsentation:



Einfügen:

- Ist der Baum ein Blatt, erzeugen wir einen neuen **inneren** Knoten mit zwei neuen leeren Blättern.
- Ist der Baum nicht-leer, vergleichen wir den einzufügenden Wert mit dem Wert an der Wurzel.

- Ist er größer, fügen wir rechts ein.
- Ist er kleiner, fügen wir links ein.

- **Achtung:** Einfügen kann die Tiefe erhöhen und damit die **AVL**-Eigenschaft zerstören !
- Das müssen wir reparieren ...

```
let rec insert x avl = match avl
with Null          -> (Eq (Null,x,Null), true)
  | Eq (left,y,right) -> if x < y then
      let (left,inc) = insert x left
      in if inc then (Neg (left,y,right), true)
         else      (Eq (left,y,right), false)
    else let (right,inc) = insert x right
      in if inc then (Pos (left,y,right), true)
         else      (Eq (left,y,right), false)
  ...
```

- Die Funktion `insert` liefert außer dem neuen **AVL**-Baum die Information, ob das Ergebnis **tiefer** ist als das Argument `:-)`
- Erhöht sich die Tiefe nicht, braucht die Markierung der Wurzel nicht geändert werden.

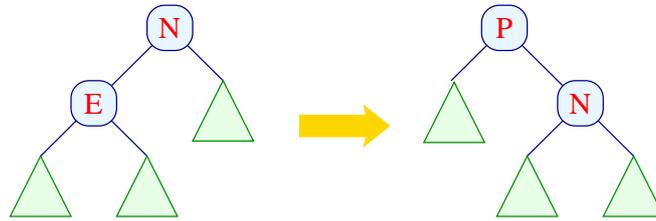
```
| Neg (left,y,right) -> if x < y then
  let (left,inc) = insert x left
  in if inc then let (avl,_) = rotateRight (left,y,right)
      in (avl,false)
    else      (Neg (left,y,right), false)
else let (right,inc) = insert x right
  in if inc then (Eq (left,y,right), false)
     else      (Neg (left,y,right), false)
| Pos (left,y,right) -> if x < y then
  let (left,inc) = insert x left
  in if inc then (Eq (left,y,right), false)
     else      (Pos (left,y,right), false)
else let (right,inc) = insert x right
  in if inc then let (avl,_) = rotateLeft (left,y,right)
      in (avl,false)
    else      (Pos (left,y,right), false);;
```

Kommentar:

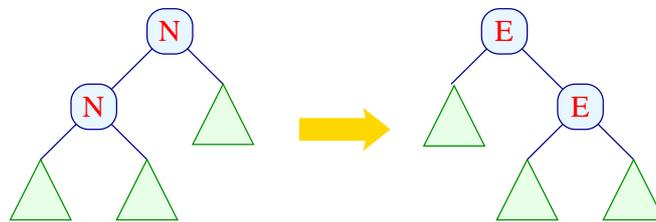
- Einfügen in den flacheren Teilbaum erhöht die Gesamttiefe nie `:-)`
Gegebenenfalls werden aber beide Teilbäume **gleich** tief.
- Einfügen in den **tieferen** Teilbaum kann dazu führen,
dass der Tiefenunterschied auf `2` anwächst `:-(`

Dann **rotieren** wir Knoten an der Wurzel, um die Differenz auszugleichen ...

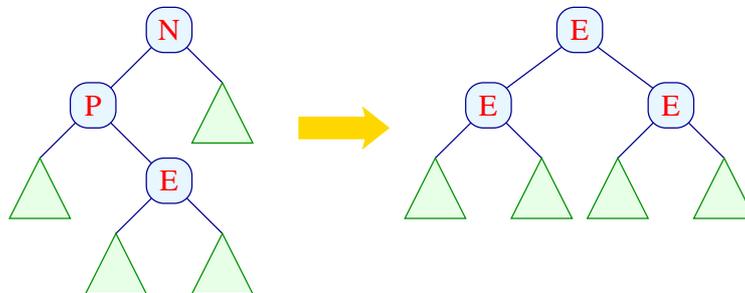
rotateRight:



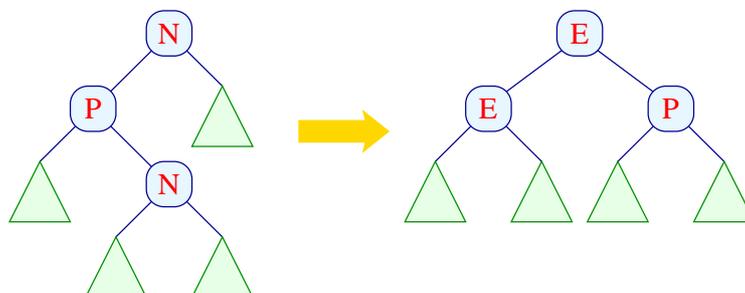
rotateRight:



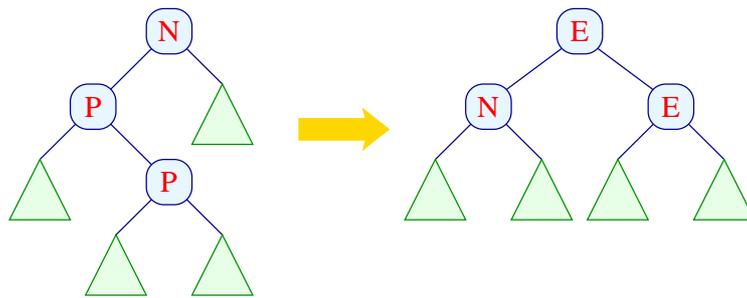
rotateRight:



rotateRight:



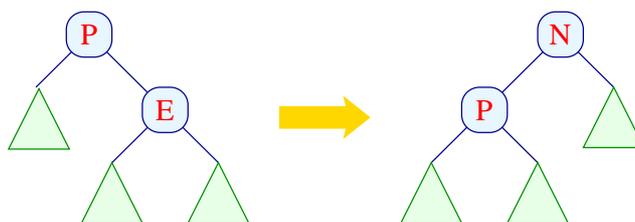
rotateRight:



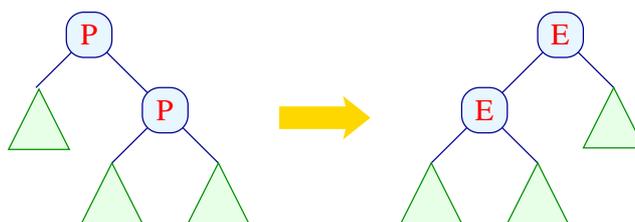
```
let rotateRight (left, y, right) = match left
  with Eq (l1,y1,r1) -> (Pos (l1, y1, Neg (r1,y,right)), false)
  | Neg (l1,y1,r1) -> (Eq (l1, y1, Eq (r1,y1,right)), true)
  | Pos (l1, y1, Eq (l2,y2,r2)) ->
    (Eq (Eq (l1,y1,l2), y2, Eq (r2,y,right)), true)
  | Pos (l1, y1, Neg (l2,y2,r2)) ->
    (Eq (Eq (l1,y1,l2), y2, Pos (r2,y,right)), true)
  | Pos (l1, y1, Pos (l2,y2,r2)) ->
    (Eq (Neg (l1,y1,l2), y2, Eq (r2,y,right)), true)
```

- Das zusätzliche Bit gibt diesmal an, ob der Baum nach der Rotation in der Tiefe **abnimmt**
...
- Das ist nur dann nicht der Fall, wenn der tiefere Teilbaum von der Form `Eq (...)` ist — was hier nie vorkommt :-)

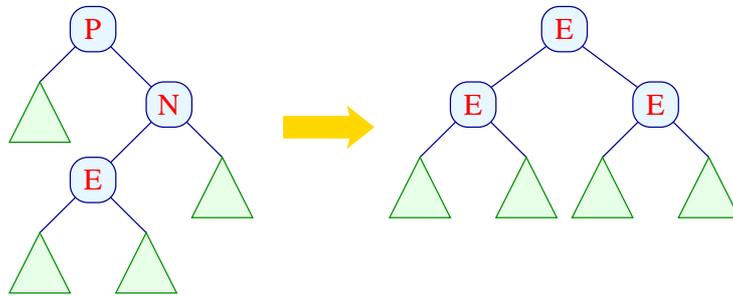
rotateLeft:



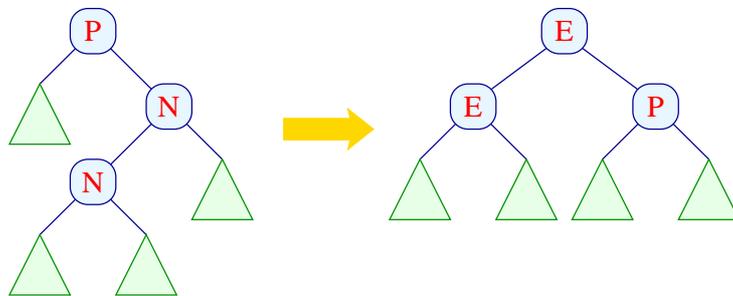
rotateLeft:



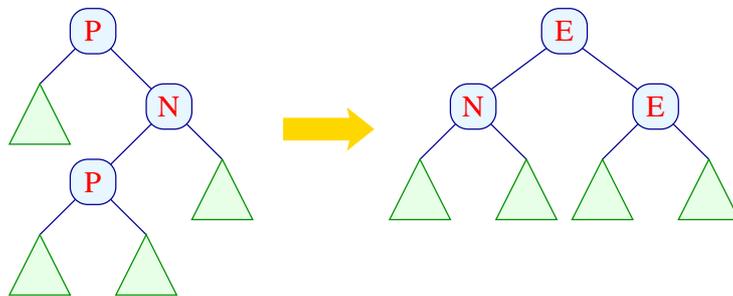
rotateLeft:



rotateLeft:



rotateLeft:



```
let rotateLeft (left, y, right) = match right
with Eq (l1,y1,r1) -> (Neg (Pos (left,y,l1), y1, r1), false)
| Pos (l1,y1,r1) -> (Eq (Eq (left,y,l1), y1, r1), true)
| Neg (Eq (l1,y1,r1), y2 ,r2) ->
    (Eq (Eq (left,y,l1),y1, Eq (r1,y2,r2)), true)
| Neg (Neg (l1,y1,r1), y2 ,r2) ->
    (Eq (Eq (left,y,l1),y1, Pos (r1,y2,r2)), true)
| Neg (Pos (l1,y1,r1), y2 ,r2) ->
    (Eq (Neg (left,y,l1),y1, Eq (r1,y2,r2)), true)
```

- `rotateLeft` ist analog zu `rotateRight` — nur mit den Rollen von `Pos` und `Neg` vertauscht.
- Wieder wird fast immer die Tiefe verringert :-)

Diskussion:

- Einfügen benötigt höchstens soviele Aufrufe von `insert` wie der Baum tief ist.
- Nach Rückkehr aus dem Aufruf für einen Teilbaum müssen maximal drei Knoten umorganisiert werden.
- Der Gesamtaufwand ist darum `proportional` zu `log(n)` :-)
- Im allgemeinen sind wir aber nicht an dem Zusatz-Bit bei jedem Aufruf interessiert. Deshalb definieren wir:

```
let insert x tree = let (tree,_) = insert x tree
                    in tree
```

Extraktion des Minimums:

- Das Minimum steht am `linksten` inneren Knoten.
- Dieses finden wir mithilfe eines rekursiven Besuchs des jeweils linken Teilbaums :-)
- Den linksten Knoten haben wir gefunden, wenn der linke Teilbaum `Null` ist :-))
- Entfernen eines Blatts könnte die Tiefe verringern und damit die `AVL`-Eigenschaft zerstören.
- Nach jedem Aufruf müssen wir darum den Baum lokal reparieren ...

```
let rec extract_min avl = match avl
with Null          -> (None, Null, false)
  | Eq (Null,y,right) -> (Some y, right, true)
  | Eq (left,y,right) -> let (first,left,dec) = extract_min left
                        in if dec then (first, Pos (left,y,right), false)
                           else      (first, Eq (left,y,right), false)
  | Neg (left,y,right) -> let (first,left,dec) = extract_min left
                        in if dec then (first, Eq (left,y,right), true)
                           else      (first, Neg (left,y,right), false)
  | Pos (Null,y,right) -> (Some y, right, true)
  | Pos (left,y,right) -> let (first,left,dec) = extract_min left
                        in if dec then let (avl,b) = rotateLeft (left,y,right)
                                       in (first,avl,b)
                           else      (first, Pos (left,y,right), false)
```

Diskussion:

- Rotierung ist nur erforderlich, wenn aus einem Baum der Form `Pos (...)` extrahiert wird und sich die Tiefe des linken Teilbaums verringert :-)
- Insgesamt ist die Anzahl der rekursiven Aufrufe beschränkt durch die Tiefe. Bei jedem Aufruf werden maximal drei Knoten umgeordnet.
- Der Gesamtaufwand ist darum proportional $\log(n)$:-)
- Analog konstruiert man Funktionen, die das Maximum bzw. das letzte Element aus einem Intervall extrahieren ...

7 Praktische Features in Ocaml

- Ausnahmen
- Imperative Konstrukte
 - modifizierbare Record-Komponenten
 - Referenzen
 - Sequenzen
 - Arrays und Strings

7.1 Ausnahmen (Exceptions)

Bei einem Laufzeit-Fehler, z.B. Division durch Null, erzeugt das Ocaml-System eine **exception** (Ausnahme):

```
# 1 / 0;;
Exception: Division_by_zero.
# List.tl (List.tl [1]);;
Exception: Failure "tl".
# Char.chr 300;;
Exception: Invalid_argument "Char.chr".
```

Hier werden die Ausnahmen `Division_by_zero`, `Failure "tl"` bzw. `Invalid_argument "Char.chr"` erzeugt.

Ein anderer Grund für eine Ausnahme ist ein **unvollständiger Match**:

```
# match 1+1 with 0 -> "null";;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
Exception: Match_failure ("", 2, -9).
```

In diesem Fall wird die Exception `Match_failure ("", 2, -9)` erzeugt :-)
Vordefinierte Konstruktoren für Exceptions:

Division_by_zero	Division durch Null
Invalid_argument of string	falsche Benutzung
Failure of string	allgemeiner Fehler
Match_failure of string * int * int	unvollständiger Match
Not_found	nicht gefunden :-)
Out_of_memory	Speicher voll
End_of_file	Datei zu Ende
Exit	für die Benutzerin ...

Eine Exception ist ein **First Class Citizen**, d.h. ein Wert eines Datentyps `exn ...`

```
# Division_by_zero;;
- : exn = Division_by_zero
# Failure "Kompletter Quatsch!";;
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` **erweitert** wird ...

```
# exception Hell;;
exception Hell
# Hell;;
- : exn = Hell
# Division_by_zero;;
- : exn = Division_by_zero
# Failure "Kompletter Quatsch!";;
- : exn = Failure "Kompletter Quatsch!"
```

Eigene Exceptions werden definiert, indem der Datentyp `exn` **erweitert** wird ...

```
# exception Hell of string;;
exception Hell of string
# Hell "damn!";;
- : exn = Hell "damn!"
```

Ausnahmebehandlung:

Wie in **Java** können Exceptions ausgelöst und behandelt werden:

```
# let teile (n,m) = try Some (n / m)
  with Division_by_zero -> None;;

# teile (10,3);;
- : int option = Some 3
# teile (10,0);;
- : int option = None
```

So kann man z.B. die member-Funktion neu definieren:

```
let rec member x l = try if x = List.hd l then true
  else member x (List.tl l)
  with Failure _ -> false

# member 2 [1;2;3];;
- : bool = true
# member 4 [1;2;3];;
- : bool = false
```

Das Schlüsselwort **with** leitet ein Pattern Matching auf dem Ausnahme-Datentyp **exn** ein:

```
try <exp>
with <pat1> -> <exp1> | ... | <patN> -> <expN>
```

⇒ Man kann mehrere Exceptions gleichzeitig abfangen :-)

Der Programmierer kann selbst Exceptions auslösen.

Das geht mit dem Schlüsselwort **raise ...**

```
# 1 + (2/0);;
Exception: Division_by_zero.
# 1 + raise Division_by_zero;;
Exception: Division_by_zero.
```

Eine Exception ist ein Fehlerwert, der jeden Ausdruck ersetzen kann.

Bei Behandlung wird sie durch einen anderen Ausdruck (vom richtigen Typ) ersetzt — oder durch eine andere Exception :-)

Exception Handling kann nach jedem beliebigen Teilausdruck, auch geschachtelt, stattfinden:

```
# let f (x,y) = x / (y-1)
# let g (x,y) = try let n = try f (x,y)
                  with Division_by_zero ->
                    raise (Failure "Division by zero")
                  in string_of_int (n*n)
                  with Failure str -> "Error: "^str

# g (6,1);;
- : string = "Error: Division by zero"
# g (6,3);;
- : string = "9"
```

7.2 Imperative Features im Ocaml

Gelegentlich möchte man Werte **destruktiv** verändern **;-)**
Dazu benötigen wir neue Konzepte ...

Modifizierbare Record-Komponenten:

- Records fassen benannte Werte zusammen **;-)**
- Einzelne Komponenten können als **modifizierbar** deklariert werden ...

```
# type cell = {owner: string; mutable value: int};;
type cell = { owner : string; mutable value : int; }

...
# let x = {owner="me"; value=1};;
val x : cell = {owner = "me"; value = 1}
# x.value;;
- : int = 1
# x.value <- 2;;
- : unit = ()
# x.value;;
- : int = 2
```

- Modifizierbare Komponenten werden mit **mutable** gekennzeichnet.
- Die Initialisierung erfolgt wie bei einer normalen Komponente.
- Der Ausdruck **x.value <- 2** hat den Wert **()**,
aber modifiziert die Komponente **value** als **Seiteneffekt !!!**

Spezialfall: Referenzen

Eine Referenz `tau ref` auf einen Typ `tau` ist ein Record mit der einzigen Komponente `mutable contents: tau`:

```
# let ref_hallo = {contents = "Hallo!"};;
val ref_hallo : string ref = {contents = "Hallo!"}
# let ref_1 = {contents = 1};;
val ref_1 : int ref = {contents = 1}
```

Deshalb kann man auf den Wert einer Referenz mit Selektion zugreifen:

```
# ref_hallo.contents;;
- : string = "Hallo!"
# ref_1.contents;;
- : int = 1
```

Eine andere Möglichkeit ist der **Dereferenzierungs-Operator** `!`:

```
# !ref_hallo;;
- : string = "Hallo!"
```

Der Wert, auf den eine Referenz zeigt, kann mit `<-` oder mit `:=` verändert werden:

```
# ref_1.contents <- 2;;
- : unit = ()
# ref_1 := 3;;
- : unit = ()
```

Gleichheit von Referenzen

Das Setzen von `ref_1` mittels `:=` erfolgt als **Seiteneffekt** und hat keinen Wert, d.h. ergibt `()`.

```
# (:=);;
- : 'a ref -> 'a -> unit = <fun>
```

Zwei Referenzen sind **gleich**, wenn sie auf **den gleichen** Wert zeigen:

```
# let x = ref 1
  let y = ref 1;;
val x : int ref = {contents = 1}
val y : int ref = {contents = 1}
# x = y;;
- : bool = true
```

Sequenzen

Bei Updates kommt es nur auf den Seiteneffekt an :-)

Bei Seiteneffekten kommt es auf die Reihenfolge an :-)

Mehrere solche Aktionen kann man mit dem **Sequenz-Operator** `;` hintereinander ausführen:

```
# ref_1 := 1; ref_1 := !ref_1 +1; ref_1;;
- : int ref = {contents = 2}
```

In **Ocaml** kann man sogar **Schleifen** programmieren ...

```
# let x = ref 0;;
val x : int ref = {contents = 1}
# while !x < 10 do x := !x+1 done;;
- : unit = ()
# x;;
- : int ref = contents = 10
```

Ein wichtiges Listenfunktional ist `List.iter`:

```
# let rec iter f = function
  [] -> ()
  | x::xs -> f x; iter f xs;;

val iter : ('a -> unit) -> 'a list -> unit = <fun>
```

Arrays und Strings

Ein Array ist ein Record fester Länge, auf dessen modifizierbare Elemente mithilfe ihres Index in **konstanter Zeit** zugegriffen wird:

```
# let arr = [|1;3;5;7|];;
val arr : int array = [|1; 3; 5; 7|]
# arr.(2);;
- : int = 5
```

Zugriff außerhalb der Array-Grenzen löst eine Exception aus:

```
# arr.(4);;
Invalid_argument "index out of bounds"
```

Ein Array kann aus einer Liste oder als **Wertetabelle** einer Funktion erzeugt werden ...

```
# Array.of_list [1;2;3];;
- : int array = [|1; 2; 3|]
# Array.init 6 (fun x -> x*x);;
- : int array = [|0; 1; 4; 9; 16; 25|]
```

... und wieder zurück in eine Liste transformiert werden:

```
Array.fold_right (fun x xs -> x::xs)
                 [|0; 1; 4; 9; 16; 25|] [];;
- : int list = [0; 1; 4; 9; 16; 25]
```

Modifizierung der Array-Einträge funktioniert analog der Modifizierung von Record-Komponenten:

```
# arr.(1) <- 4;;
- : unit = ()
# arr;;
- : int array = [|1; 4; 5; 7|]
# arr.(5) <- 0;;
Exception: Invalid_argument "index out of bounds".
```

Ähnlich kann man auch Strings manipulieren :-)

```
# let str = "Hallo";  
val str : string = "Hallo"  
# str.[2];;  
- : char = 'l'  
# str.[2] <- 'r';;  
- : unit = ()  
# str;;  
- : string = "Harlo"
```

Für Arrays und Strings gibt es übrigens auch die Funktionen `length` und `concat` (und weitere :-).

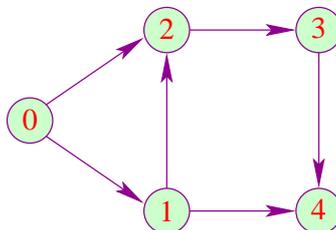
8 Anwendung: Grundlegende Graph-Algorithmen

- Gerichtete Graphen
- Erreichbarkeit und DFS
- Topologische Sortierung
- Kürzeste Wege

8.1 Gerichtete Graphen

Beobachtung:

- Viele Probleme lassen sich mit Hilfe **gerichteter Graphen** repräsentieren ...



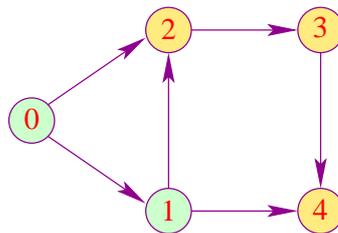
	Knoten	Kanten
Betrieblicher Prozess	Bearbeitungsschritt	Abfolge
Software	Zustand	Änderung
Systemanalyse	Komponente	Einfluss

- Oft sind die Kanten mit Zusatz-Informationen **beschriftet** :-)

8.2 Erreichbarkeit und DFS

Einfaches Problem:

- Gegeben ein Knoten.
- Welche anderen Knoten im Graphen kann man erreichen?



Eine Datenstruktur für Graphen:

- Wir nehmen an, die Knoten sind fortlaufend mit 0, 1, ... durch-nummeriert.
- Für jeden Knoten gibt es eine **Liste** ausgehender Kanten.
- Damit wir schnellen Zugriff haben, speichern wir die Listen in einem **Array** ab ...

```

# type node      = int
# type 'a graph = ('a * node) list array

# let example = [| [((,1); ((,2))]; [((,2); ((,4))];
                 [((,3))]; [((,4))]; [ ] |] ;

```

Idee:

- Wir verwalten eine Datenstruktur `reachable : bool array`.
- Am Anfang haben alle Einträge `reachable.(x)` den Wert `false`.
- Besuchen wir einen Knoten `x`, der noch nicht erreicht wurde, setzen wir `reachable.(x) <- true` und besuchen alle Nachbarn von `x` :-)
- Kommen wir zu einem Knoten, der bereits besucht wurde, tun wir nichts :-))

```
# let fold_li f a arr = let n = Array.length arr
                        in let rec doit i a = if i = 0 then a
                                                else doit (i-1) (f i a arr.(i))
                        in doit (n-1) a;;
```

```
# let reach edges x =
  let n = Array.length edges
  in let reachable = Array.make n false
  in let extract_result () = fold_li
    (fun i acc b -> if b then i::acc
                    else acc) [] reachable
  ...
```

Kommentar:

- `fold_li : (int -> 'a -> 'b -> 'a) -> 'a -> 'b array -> 'a` erlaubt, über ein Array zu iterieren unter Berücksichtigung des Index wie der Werte für den Index :-)
- `Array.make : int -> 'a -> 'a array` legt ein neues initialisiertes Array an.
- Die entscheidende Berechnung erfolgt in der rekursiven Funktion `dfs ...`
- Die Technik heißt **Depth First Search** oder **Tiefensuche**, weil die Nachfolger eines Knotens **vor** den Nachbarn besucht werden :-)

```
...
in let rec dfs x = if not reachable.(x) then (
  reachable.(x) <- true;
  List.iter (fun (_,y) -> dfs y)
            edges.(x)
)
```

```

in dfs x;
  extract_result ();;

```

Kommentar (Forts.):

Der Test am Anfang von `dfs` liefert für jeden Knoten nur **einmal** `true`.

⇒ Jede Kante wird maximal **einmal** behandelt.

⇒ Der Gesamtaufwand ist proportional zu $n + m$

// n == Anzahl der Knoten

// m == Anzahl der Kanten

8.3 Topologisches Sortieren

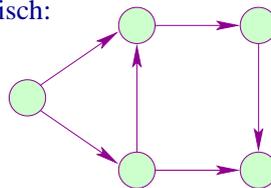
Ein **Pfad** oder **Weg** der Länge n in einem gerichteten Graphen ist eine Folge von Kanten:

$$(v_0, \rightarrow v_1)(v_1, \rightarrow v_2) \dots (v_{n-1}, \rightarrow v_n)$$

Ein Pfad der Länge $n > 0$ von v nach v heißt **Kreis**.

Ein gerichteter Graph ohne Kreise heißt **azyklisch ...**

azyklisch:

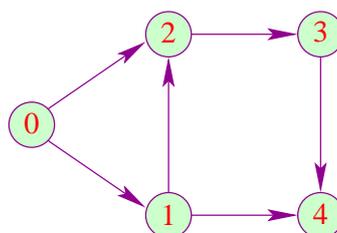


Aufgabe:

Gegeben: ein gerichteter Graph.

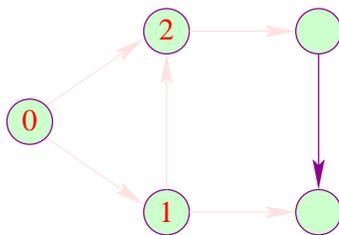
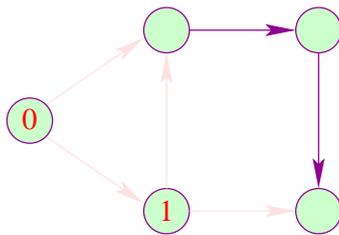
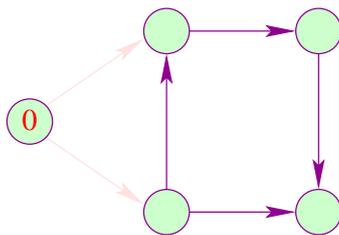
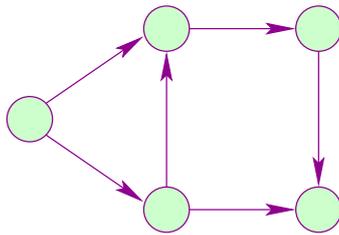
Frage: Ist der Graph azyklisch?

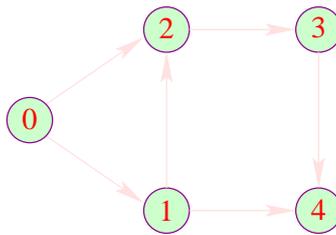
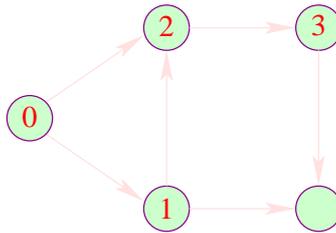
Wenn ja: Finde eine **lineare Anordnung** der Knoten!



Idee:

- Suche einen Knoten **ohne eingehende Kanten**.
- Besitzt jeder Knoten eingehende Kanten, muss der Graph einen Kreis enthalten ;-)
- Gibt es einen Knoten ohne eingehende Kanten, können wir diesen als kleinstes Element auswählen :-)
- Entferne den Knoten mit allen seinen **ausgehenden Kanten** !
- Löse das verbleibende Problem ...





Implementierung:

- Für jeden Knoten x benutzen wir einen **Zähler** `count.(x)`, der die Anzahl der eingehenden Kanten zählt.
- Haben wir einen Knoten, dessen Zähler 0 ist, nummerieren wir ihn **:-)**
- Dann durchlaufen wir seine ausgehenden Kanten.
- Für jeden Nachbarn dekrementieren wir den Zähler und fahren rekursiv fort.

⇒ wieder **Depth-First-Search**

```

# let top_sort edges =
  let n = Array.length edges
  in let count = Array.make n 0
  in let inc y = count.(y) <- count.(y) + 1
  in let dec y = count.(y) <- count.(y) - 1
  in for x=0 to n-1 do
    List.iter (fun (_,y) -> inc y) edges.(x)
  done ;
  let value = Array.make n (-1)
  ...

```

```

in let next = ref 0
in let number y = value.(y) <- !next;
    next := !next + 1
in let rec dfs x = if count.(x) = 0 then (
    number x;
    List.iter (fun (_,y) ->
        dec y;
        dfs y)
    edges.(x))
in for x=0 to n-1 do
    if value.(x) = -1 then dfs x
done; value;;

```

Diskussion:

- `for y=... to ... do ... done` ist eine vereinfachte `while`-Schleife ;-)
- Für die Initialisierung des Arrays `count` iterieren wir einmal über alle Knoten und für jeden Knoten einmal über alle ausgehenden Kanten.

⇒ Aufwand: proportional $n + m$

- Die Hauptschleife behandelt jeden Knoten maximal einmal.
- `dfs` wird maximal einmal für jeden Knoten und jede Kante aufgerufen und behandelt jede Kante maximal einmal.

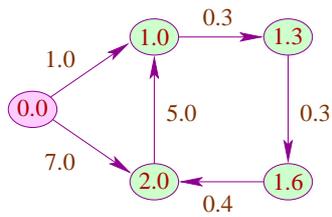
⇒ Aufwand: proportional $n + m$

⇒ Der Gesamtaufwand ist linear :-)

8.4 Kürzeste Wege

Gegeben:

- Ein gerichteter Graph mit **Kosten** an den Kanten;
- ein **Startknoten** im Graphen.



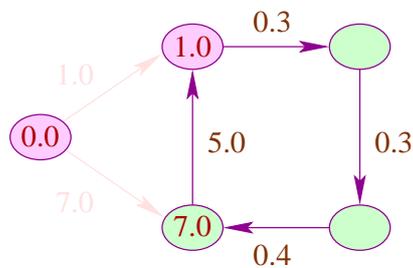
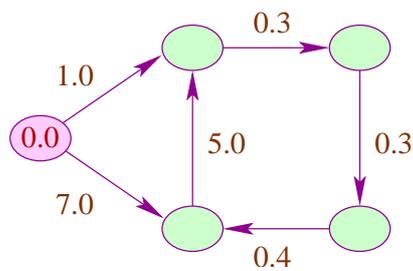
Gesucht:

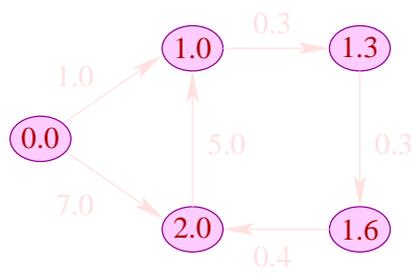
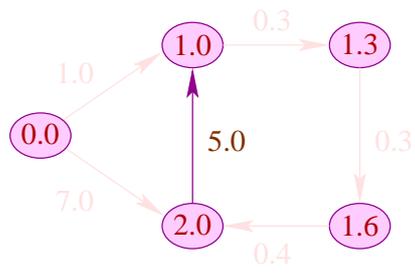
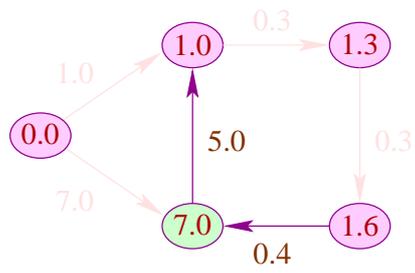
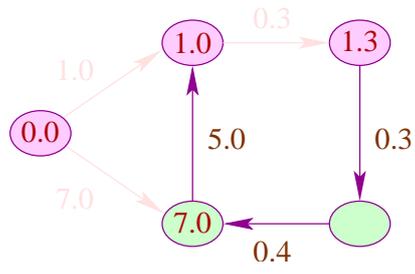
Die **minimalen Kosten**, um die anderen Knoten zu erreichen ...

Idee: Dijkstras Algorithmus

- Verwalte für jeden Knoten **mögliche Kosten**.
- Da die Kosten **nicht-negativ** sind, können die Kosten des **billigsten** Knotens können weiter reduziert werden !

- ⇒
- Wir entfernen diesen Knoten;
 - Wir benutzen seine ausgehenden Kanten, um die oberen Schranken der anderen Knoten evt. zu verbessern;
 - Wir entfernen die Kanten und fahren mit dem nächst billigen Knoten fort ...







Edsger Dijkstra, 1930-2002

Implementierung:

- Wir versuchen, stets die **lokal günstigste** Entscheidung zu treffen
 \implies **greedy Algorithmus**
- Wir verwalten die Menge der Paare (c, x) von bereits besuchten, aber noch nicht selektierten Knoten x mit aktuellen Kosten $c : \text{float}$ in einer Datenstruktur.
- Die Datenstruktur d sollte die folgenden Operationen unterstützen:

```
extract_min : (float * node) d -> (float * node) option  
insert      : (float * node) -> (float * node) d -> (float * node) d  
delete     : (float * node) -> (float * node) d -> (float * node) d
```

\implies wir können **AVL**-Bäume benutzen :-)

```
# let dijkstra edges x = let n = Array.length edges  
  in let dist = Array.make n None  
  in let _ = dist.(x) <- Some 0.  
  in let avl = Eq (Null, (0.,x), Null)  
  in let do_edge (d,x) avl (a,y) = match dist.(y)  
    with None   -> dist.(y) <- Some (d+.a) ;  
              insert (d+.a,y) avl  
    | Some old  -> if d+.a < old then (  
      dist.(y) <- Some (d+.a);  
      insert (d+.a,y) (  
        delete (old,y) avl))  
    else avl  
  ...
```

```

...
in let rec bfs = function
  | Null -> dist
  | avl -> let (Some (d,x),avl) = extract_min avl
           in let avl = List.fold_left
                (do_edge (d,x))
                avl
                edges.(x)
           in bfs avl
in bfs avl;;

```

Diskussion:

- Wurde x einmal als Minimum extrahiert, wird es nie wieder in avl eingetragen
 \implies es gibt maximal n `extract_min`
- Jede ausgehende Kante von x wird darum auch nur einmal behandelt `;-)`
 \implies es gibt maximal m `insert`
 \implies es gibt maximal m `delete`
- Der **Gesamtaufwand** ist folglich proportional zu:

$$n + m \cdot \log(n)$$

// das lässt sich mit Hilfe von **Fibonacci-Heaps** verbessern `;-)`

9 Das Modulsystem von OCAML

- Strukturen
- Signaturen
- Information Hiding
- Funktoren
- Constraints
- getrennte Übersetzung

9.1 Module oder Strukturen

Zur Strukturierung großer Programmsysteme bietet [Ocaml Strukturen](#) an:

```
module Pairs =  
  struct  
    type 'a pair = 'a * 'a  
    let pair (a,b) = (a,b)  
    let first (a,b) = a  
    let second (a,b) = b  
  end
```

Auf diese Eingabe antwortet der Compiler mit dem Typ der Struktur, einer [Signatur](#):

```
module Pairs :  
  sig  
    type 'a pair = 'a * 'a  
    val pair : 'a * 'b -> 'a * 'b  
    val first : 'a * 'b -> 'a  
    val second : 'a * 'b -> 'b  
  end
```

Die Definitionen innerhalb der Struktur sind außerhalb **nicht sichtbar**:

```
# first;;  
Unbound value first
```

Zugriff auf Komponenten einer Struktur:

Über den Namen greift man auf die Komponenten einer Struktur zu:

```
# Pairs.first;;  
- : 'a * 'b -> 'a = <fun>
```

So kann man z.B. [mehrere Funktionen](#) gleichen Namens definieren:

```

# module Triples = struct
  type 'a triple = Triple of 'a * 'a * 'a
  let first (Triple (a,_,_)) = a
  let second (Triple (_,b,_)) = b
  let third (Triple (_,_,c)) = c
end;;
...

...
module Triples :
sig
  type 'a triple = Triple of 'a * 'a * 'a
  val first : 'a triple -> 'a
  val second : 'a triple -> 'a
  val third : 'a triple -> 'a
end
# Triples.first;;
- : 'a Triples.triple -> 'a = <fun>

```

... oder [mehrere Implementierungen](#) der gleichen Funktion:

```

# module Pairs2 =
  struct
    type 'a pair = bool -> 'a
    let pair (a,b) = fun x -> if x then a else b
    let first ab = ab true
    let second ab = ab false
  end;;

```

Öffnen von Strukturen

Um nicht immer den Strukturnamen verwenden zu müssen, kann man [alle](#) Definitionen einer Struktur auf einmal sichtbar machen:

```

# open Pairs2;;
# pair;;
- : 'a * 'a -> bool -> 'a = <fun>
# pair (4,3) true;;
- : int = 4

```

Sollen die Definitionen des anderen Moduls **Bestandteil** des gegenwärtigen Moduls sein, dann macht man sie mit `include` verfügbar ...

```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
  open A
  let y = 2
end;;
module B : sig val y : int end
# module C = struct
  include A
  include B
end;;
module C : sig val x : int val y : int end
```

Geschachtelte Strukturen

Strukturen können selbst wieder Strukturen enthalten:

```
module Quads = struct
  module Pairs = struct
    type 'a pair = 'a * 'a
    let pair (a,b) = (a,b)
    let first (a,_) = a
    let second (_,b) = b
  end
  type 'a quad = 'a Pairs.pair Pairs.pair
  let quad (a,b,c,d) =
    Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
  ...

  ...
  let first q = Pairs.first (Pairs.first q)
  let second q = Pairs.second (Pairs.first q)
  let third q = Pairs.first (Pairs.second q)
  let fourth q = Pairs.second (Pairs.second q)
end
```

```

# Quads.quad (1,2,3,4);;
- : (int * int) * (int * int) = ((1,2),(3,4))
# Quads.Pairs.first;;
- : 'a * 'b -> 'a = <fun>

```

9.2 Modul-Typen oder Signaturen

Mithilfe von **Signaturen** kann man einschränken, was eine Struktur nach außen exportiert.

Beispiel:

```

module Count =
  struct
    let count = ref 0
    let setCounter n = count := n
    let getCounter () = !count
    let incCounter () = count := !count+1
  end

```

Der Zähler ist nach außen sichtbar, zugreifbar und veränderbar:

```

# !Count.count;;
- : int = 0
# Count.count := 42;;
- : unit = ()

```

Will man, daß nur die Funktionen der Struktur auf ihn zugreifen können, benutzt man einen Modul-Typ oder **Signatur**:

```

module type Count =
  sig
    val setCounter : int -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> int
  end

```

Die Signatur enthält den Zähler selbst nicht :-)

Mit dieser Signatur wird die Schnittstelle der Struktur eingeschränkt:

```
# module SafeCount : Count = Count;;
module SafeCount : Count
# SafeCount.count;;
Unbound value Safecount.count
```

Die Signatur bestimmt, welche Definitionen exportiert werden.
Das geht auch direkt bei der Definition:

```
module Count1 : Count =
  struct
    val count = ref 0
    fun setCounter n = count := n
    fun getCounter () = !count
    fun incCounter () = count := !count+1
  end

# !Count1.count;;
Unbound value Count1.count
```

Signaturen und Typen

Die in der Signatur angegebenen Typen müssen **Instanzen** der für die exportierten Definitionen inferierten Typen sein :-)

Dadurch werden deren Typen spezialisiert:

```
module type A1 = sig
  val f : 'a -> 'b -> 'b
end
module type A2 = sig
  val f : int -> char -> int
end
module A = struct
  let f x y = x
end
```

```
# module A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
                        is not included in A1

Values do not match:
  val f : 'a -> 'b -> 'a
is not included in
  val f : 'a -> 'b -> 'b
# module A2 : A2 = A;;
module A2 : A2
# A2.f;;
- : int -> char -> int = <fun>
```

9.3 Information Hiding

Aus Gründen der Modularität möchte man oft verhindern, dass die Struktur exportierter Typen einer Struktur von außen sichtbar ist.

Beispiel:

```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
```

```

    [] -> true | _ -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end

```

Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```

module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end

```

```

# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue

```



Das Einschränken per Signatur genügt, um die **wahre Natur** des Typs `queue` zu verschleiern :-)

Soll der Datentyp mit seinen Konstruktoren dagegen exportiert werden, **wiederholen** wir seine Definition in der Signatur:

```

module type Queue =
sig
  type 'a queue = Queue of ('a list * 'a list)
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a -> 'a queue -> 'a queue
  val dequeue : 'a queue -> 'a option * 'a queue
end

```

9.4 Funktoren

Da in **Ocaml** fast alles höherer Ordnung ist, wundert es nicht, dass es auch Strukturen höherer Ordnung gibt: die **Funktoren**.

- Ein Funktor bekommt als Parameter eine Folge von Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Argumente des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.

Wir legen zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
module type Enum = sig
  type enum
  val null : enum
  val incr : enum -> enum
  val int_of_enum : enum -> int
end
module type Counter = sig
  type counter
  val incCounter : unit -> unit
  val getCounter : unit -> counter
  val int_of_counter : counter -> int
end
...

...
# module GenCounter (Enum : Enum) : Counter =
  struct
    open Enum
    type counter = enum
    let count = ref null
    let incCounter() = count := incr(!count)
    let getCounter() = !count
    let int_of_counter x = int_of_enum x
  end;;
module GenCounter : functor (Enum : Enum) -> Counter
```

Jetzt können wir den Funktor auf eine Struktur [anwenden](#) und erhalten eine neue Struktur ...

```
module PlusEnum = struct
  type enum = int
  let null = 0
  let incr x = x+1
  let int_of_enum x = x
end

# module PlusCounter = GenCounter (PlusEnum);;
module PlusCounter : Counter

# PlusCounter.incCounter ();
PlusCounter.incCounter ();
PlusCounter.int_of_counter (PlusCounter.getCounter ());;
- : int = 2
```

Eine erneute Anwendung des Funktors erzeugt eine [neue Struktur](#):

```
module MalEnum = struct
  type enum = int
  let null = 1
  let incr x = 2*x
  let int_of_enum x = x
end

# module MalCounter = GenCounter (MalEnum);;
module MalCounter : Counter

# MalCounter.incCounter();
MalCounter.incCounter();
MalCounter.int_of_counter (MalCounter.getCounter());;
- : int = 4
```

Mit einem Funktor kann man sich auch [mehrere Instanzen](#) des gleichen Moduls erzeugen:

```
# module CountApples = GenCounter (PlusEnum);;
module CountApples : Counter
# module CountPears = GenCounter (PlusEnum);;
module CountPears : Counter
...
```

```

...
# CountApples.incCounter();;
- : unit = ()
# CountApples.incCounter();;
- : unit = ()
# CountPears.incCounter ();;
- : unit = ()
# CountApples.int_of_counter (CountApples.getCounter());;
- : int = 2
# CountPears.int_of_counter (CountPears.getCounter());;
- : int = 1

```

9.5 Constraints

Manchmal möchten wir zwei Typen, die in verschiedenen Strukturen definiert sind, [identifizieren](#):

Beispiel:

```

module type PrintCounter = sig
  type counter
  val incCounter : unit -> unit
  val getCounter : unit -> counter
  val int_of_counter : counter -> int
  val printCounter : unit -> unit
end

```

Nehmen wir an, wir hätten eine weitere Signatur:

```

module type Printable = sig
  type printable
  val print : printable -> unit
end

```

Wir wollen einen Funktor definieren, der eine Struktur produziert, die zählen und den Zählerstand ausdrucken kann:

```

module GenPrintCounter (E : Enum) (P : Printable)
    : PrintCounter = struct
  module Count = GenCounter (E)
  include Count
  let printCounter() = P.print (getCounter())
end

```

Leider bekommen wir den Fehler:

```

# module GenPrintCounter (E : Enum) (P : Printable)
    : PrintCounter = struct
  module Count = GenCounter (E)
  include Count
  let printCounter() = P.print (getCounter())
end
his expression has type counter = GenCounter(E).counter
but is here used with type P.printable

```

Der Compiler kann nicht wissen, dass der Typ `Print.printable` gleich `GenCounter(E).counter` sein soll `:-)`

... das muss man ihm durch **Constraints** mitteilen:

```

module GenCounter (E : Enum) : Counter
    with type counter = E.enum =
  struct open E
    type counter = enum
    let count = ref null
    let incCounter() = count := incr(!count)
    let getCounter() = !count
    let int_of_counter x = int_of_enum x
  end
  ...

```

Der Typ `counter` in der Rückgabe des Funktors `genCounter` soll gleich dem Typ `enum` des Arguments sein `:-)`

Außerdem soll der Typ `printable` des zweiten Arguments gleich dem Typ `enum` des ersten Arguments sein `...`

```

module GenPrintCounter (E : Enum)
    (P : Printable
     with type printable = E.enum)
    : PrintCounter = struct
  module Count = GenCounter (E)
  type counter = P.printable
  include Count
  let printCounter() = P.print (getCounter())
end

```

Der Funktor darf nun allerdings nur noch auf solche Strukturen angewendet werden, bei denen die Constraints erfüllt sind :-)

9.6 Getrennte Übersetzung

- Eigentlich möchte man **Ocaml**-Programme nicht immer in der interaktiven Umgebung starten :-)

- Dazu gibt es u.a. den Compiler `ocamlc ...`

```
> ocamlc Test.ml
```

interpretiert den Inhalt der Datei `Test.ml`

als Folge von Definitionen einer Struktur `Test`.

- Als Ergebnis der Übersetzung liefert `ocamlc` die Dateien:

<code>Test.cmo</code>	Bytecode für die Struktur
<code>Test.cmi</code>	Bytecode für das Interface
<code>a.out</code>	lauffähiges Programm

- Gibt es eine Datei `Test.mli`

wird diese als Definition der Signatur für `Test` aufgefasst. Dann rufen wir auf:

```
> ocamlc Test.mli Test.ml
```

- Benutzt eine Struktur `A` eine Struktur `B`, dann sollte diese mit übersetzt werden:

```
> ocamlc B.mli B.ml A.mli A.ml
```

- Möchte man auf die Neuübersetzung von `B` verzichten, kann man `ocamlc` auch die vor-übersetzte Datei mitgeben:

```
> ocamlc B.cmo A.mli A.ml
```

- Zur praktischen Verwaltung von benötigten Neuübersetzungen nach Änderungen von Dateien bietet **Linux** das Kommando `make` an. Das Protokoll der auszuführenden Aktionen steht dann in einer Datei `Makefile` :-)

10 Formale Methoden für Ocaml

Frage:

Wie können wir uns versichern, dass ein Ocaml-Programm das macht, was es tun soll ???

Wir benötigen:

- eine formale Semantik;
- Techniken, um Aussagen über Programme zu beweisen ...

10.1 MiniOcaml

Um uns das Leben leicht zu machen, betrachten wir nur einen kleinen Ausschnitt aus Ocaml.

Wir erlauben ...

- nur die Basistypen `int`, `bool` sowie Tupel und Listen;
- rekursive Funktionsdefinitionen nur auf dem Top-Level :-)

Wir verbieten ...

- veränderbare Datenstrukturen;
- Ein- und Ausgabe;
- lokale rekursive Funktionen :-)

Dieses Fragment von Ocaml nennen wir MiniOcaml.

Ausdrücke in MiniOcaml lassen sich durch die folgende Grammatik beschreiben:

$$\begin{aligned} E ::= & \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid \\ & (E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid \\ & \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid \\ & \text{fun name} \rightarrow E \mid E_1 E_2 \\ \\ P ::= & \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2 \end{aligned}$$

Achtung:

- Die Menge der **erlaubten** Ausdrücke muss weiter eingeschränkt werden auf diejenigen, die **typkorrekt** sind, d.h. für die der **Ocaml**-Compiler einen Typ herleiten kann ...
 - (1, [true; false]) **typkorrekt**
 - (1 [true; false]) **nicht typkorrekt**
 - ([1; true], false) **nicht typkorrekt**
- Wir verzichten auf `if ... then ... else ...`, da diese durch `match ... with true -> ... | false -> ...` simuliert werden können :-)
- Wir hätten auch auf `let ... in ...` verzichten können (wie?)

Ein **Programm** besteht dann aus einer Folge wechselseitig rekursiver globaler Definitionen von Variablen f_1, \dots, f_m :

```
let rec f1 = E1
    and f2 = E2
    ...
    and fm = Em
```

10.2 Eine Semantik für MiniOcaml

Frage:

Zu welchem **Wert** wertet sich ein Ausdruck E aus ??

Ein **Wert** ist ein Ausdruck, der nicht weiter ausgerechnet werden kann :-)

Die Menge der Werte lässt sich ebenfalls mit einer Grammatik beschreiben:

$$V ::= \text{const} \mid \text{fun name} \rightarrow E \mid (V_1, \dots, V_k) \mid [] \mid V_1 :: V_2$$

Ein MiniOcaml-Programm ...

```
let rec comp = fun f -> fun g -> fun x -> f (g x)
  and map = fun f -> fun list -> match list
    with [] -> []
         | x::xs -> f x :: map f xs
```

Beispiele für Werte ...

```
1
(1, [true; false])
fun x -> 1 + 1
[fun x -> x+1; fun x -> x+2; fun x -> x+3]
```

Idee 1:

- Wir definieren einen **Schritt** der Berechnung als eine Relation:
 $e \rightarrow e'$ zwischen Ausdrücken.
- Eine **Berechnung** ist eine **Folge** von Berechnungsschritten
 \implies **Small-Step operationelle Semantik**
- Auf Werten ist kein weiterer Berechnungsschritt möglich,
d.h. bei Erreichen eines Wert endet eine Berechnung **:-)**
- Einzelne Berechnungsschritte definieren wir mit Hilfe geeigneter Axiome und Regeln
...

// v, v_1, \dots stehen für Werte,
// e, e_1, \dots bezeichnen beliebige Ausdrücke

Tupel:

$$\frac{e_i \rightarrow e'_i}{(v_1, \dots, v_{i-1}, e_i, \dots, e_k) \rightarrow (v_1, \dots, v_{i-1}, e'_i, e_{i+1}, \dots, e_k)}$$

Listen:

$$\frac{e_1 \rightarrow e'_1}{e_1 :: e_2 \rightarrow e'_1 :: e_2} \qquad \frac{e_2 \rightarrow e'_2}{v_1 :: e_2 \rightarrow v_1 :: e'_2}$$

Globale Definitionen:

$$\frac{f = e}{f \rightarrow e}$$

Lokale Definitionen:

$$\frac{e_1 \rightarrow e'_1}{\text{let } x = e_1 \text{ in } e_0 \rightarrow \text{let } x = e'_1 \text{ in } e_0}$$
$$\text{let } x = v \text{ in } e_0 \rightarrow e_0[v/x]$$

Funktionsaufrufe:

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2}$$
$$(\text{fun } x \rightarrow e_0) v \rightarrow e_0[v/x]$$

Pattern Matching:

$$\frac{e_0 \rightarrow e'_0}{\text{match } e_0 \text{ with } \textit{plist} \rightarrow \text{match } e'_0 \text{ with } \textit{plist}}$$

$$\frac{v \equiv p_i[v_1/x_1, \dots, v_k/x_k]}{\text{match } v \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m \rightarrow e_i[v_1/x_1, \dots, v_k/x_k]}$$

— sofern v auf keines der Muster p_1, \dots, p_{i-1} passt :-)

// “ \equiv ” bezeichnet **syntaktische Gleichheit**

Eingebaute Operatoren:

$$\frac{e_1 \rightarrow e'_1}{e_1 \text{ op } e_2 \rightarrow e'_1 \text{ op } e_2} \quad \frac{e_2 \rightarrow e'_2}{v \text{ op } e_2 \rightarrow v \text{ op } e'_2}$$

$$\frac{v_1 \text{ op } v_2 = v}{v_1 \text{ op } v_2 \rightarrow v}$$

Die unären Operatoren behandeln wir analog :-)

Beispiel:

```
let f = fun x -> x+1
let s = fun y -> y*y
```

```
f 16 + s 2  → (fun x -> x+1) 16 + s 2
             → (16+1) + s 2
             → 17 + s 2
             → 17 + (fun y -> y*y) 2
             → 17 + (2*2)
             → 17 + 4
             → 21
```

Beispiel:

```
let rec app = fun x -> fun y -> match x
  with [] -> y
       | h::t -> h :: app t y
```

Frage:

Welchen Wert liefert `app (1::[]) (2::[])` ?

```
app (1::[]) (2::[])
  → (fun x -> fun y -> ...) (1::[]) (2::[])
  → (fun y -> match ...) (2::[])
  → match 1::[] with ...
  → 1 :: app [] (2::[])
  → 1 :: (fun x -> fun y -> ...) [] (2::[])
  → 1 :: (fun y -> match ...) (2::[])
  → 1 :: match [] with ...
  → 1 :: 2 :: []
```

Diskussion:

- Die Small-Step operationelle Semantik ist gut geeignet, um eine einzelne Berechnung nach zu vollziehen :-)
- Sie ist weniger gut geeignet, um nachzuweisen, dass verschiedene Ausdrücke den selben Wert liefern :-)

Idee 2:

- Wir definieren eine Relation: $e \Rightarrow v$ zwischen Ausdrücken und ihren Werten = \Rightarrow **Big-Step operationelle Semantik**.
- Diese Relation definieren wir erneut mit Hilfe von Axiomen und Regeln, die sich an der **Struktur** von e orientieren :-)
- Offenbar gilt stets: $v \Rightarrow v$ für jeden Wert v :-))

Tupel:

$$\frac{e_1 \Rightarrow v_1 \quad \dots \quad e_k \Rightarrow v_k}{(e_1, \dots, e_k) \Rightarrow (v_1, \dots, v_k)}$$

Listen:

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 :: e_2 \Rightarrow v_1 :: v_2}$$

Globale Definitionen:

$$\frac{f = e \quad e \Rightarrow v}{f \Rightarrow v}$$

Lokale Definitionen:

$$\frac{e_1 \Rightarrow v_1 \quad e_0[v_1/x] \Rightarrow v_0}{\text{let } x = e_1 \text{ in } e_0 \Rightarrow v_0}$$

Funktionsaufrufe:

$$\frac{e_1 \Rightarrow \text{fun } x \rightarrow e_0 \quad e_2 \Rightarrow v_2 \quad e_0[v_2/x] \Rightarrow v_0}{e_1 e_2 \Rightarrow v_0}$$

Pattern Matching:

$$\frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \dots, v_k/x_k] \quad e_i[v_1/x_1, \dots, v_k/x_k] \Rightarrow v}{\text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m \Rightarrow v}$$

— sofern v' auf keines der Muster p_1, \dots, p_{i-1} passt :-)

Eingebaute Operatoren:

$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad v_1 \text{ op } v_2 = v}{e_1 \text{ op } e_2 \Rightarrow v}$$

Die unären Operatoren behandeln wir analog :-)

Beispiel:

```
let f = fun x -> x+1
let s = fun y -> y*y
```

$$\frac{\frac{f = \text{fun } x \rightarrow x+1}{f \Rightarrow \text{fun } x \rightarrow x+1} \quad \frac{16+1 = 17}{16+1 \Rightarrow 17}}{f \ 16 \Rightarrow 17} \quad \frac{\frac{s = \text{fun } y \rightarrow y*y}{s \Rightarrow \text{fun } y \rightarrow y*y} \quad \frac{2*2 = 4}{2*2 \Rightarrow 4}}{s \ 2 \Rightarrow 4} \quad 17+4 = 21}{f \ 16 + g \ 2 \Rightarrow 21}$$

// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen :-)

Beispiel:

```
let rec app = fun x -> fun y -> match x
  with [] -> y
       | h::t -> h :: app t y
```

Behauptung: $\text{app } (1::[]) (2::[]) \Rightarrow 1::2::[]$

Beweis:

$$\frac{\frac{\frac{\text{app} = \text{fun } x \rightarrow \text{fun } y \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \rightarrow \text{fun } y \rightarrow \dots}}{\text{app } (1::[]) \Rightarrow \text{fun } y \rightarrow \text{match } \dots} \quad \frac{\frac{\frac{\frac{\text{app} = \text{fun } x \rightarrow \text{fun } y \rightarrow \dots}{\text{app} \Rightarrow \text{fun } x \rightarrow \text{fun } y \rightarrow \dots}}{\text{app } (1::[]) \Rightarrow \text{fun } y \rightarrow \text{match } \dots} \quad \frac{2::[] \Rightarrow 2::[]}{\text{match } [] \dots \Rightarrow 2::[]}}{\text{app } [] (2::[]) \Rightarrow 2::[]}}{\frac{1 :: \text{app } [] (2::[]) \Rightarrow 1::2::[]}{\text{match } 1::[] \dots \Rightarrow 1::2::[]}}}{\text{app } (1::[]) (2::[]) \Rightarrow 1::2::[]}$$

// Benutzungen von $v \Rightarrow v$ haben wir i.a. weggelassen :-)

Diskussion:

- Die **Big-Step operationelle Semantik** ist nicht sehr gut geeignet, um Schritt für Schritt nahezu vollziehen, was ein **MiniOcaml**-Programm macht :-)
- Wir können damit aber überprüfen, dass **optimierende Transformationen** korrekt sind :-)
- Schließlich können wir sie benutzen, um die Korrektheit von Aussagen über funktionale Programme zu beweisen !
- Zuerst müssen wir aber zeigen, dass sie zur **Small-Step operationellen Semantik äquivalent** ist ...

Notation:

$e_1 \xrightarrow{*} e_2$ heißt $e_1 \equiv e_2$ (textuelle Gleichheit)
 oder es gibt eine Folge von Berechnungsschritten: $e_1 \rightarrow \dots \rightarrow e_2$

Satz

Für jedes **MiniOcaml**-Programm gilt:

$$e \xrightarrow{*} v \quad \text{genau dann wenn} \quad e \Rightarrow v$$

// e Ausdruck, v Wert

Diskussion:

- Der Satz lässt sich mit Hilfe von Induktion beweisen einmal nach der Länge von Folgen von Berechnungsschritten, einmal nach der Tiefe von Ableitungen der Big-Step operationellen Semantik :-)
- Die Big-Step operationelle Semantik legt nahe, Ausdrücke als **Beschreibungen** von Werten aufzufassen.
- Ausdrücke, die sich zu den **gleichen** Werten auswerten, sollten deshalb austauschbar sein ...

Achtung:

- **Gleichheit** zwischen Werten kann in **MiniOcaml** nur getestet werden, wenn diese keine Funktionen enthalten !!
- Solche Werte nennen wir **vergleichbar**. Sie haben die Form:

$$C ::= \text{const} \mid (C_1, \dots, C_k) \mid [] \mid C_1 :: C_2$$

- Offenbar ist ein **MiniOcaml**-Wert genau dann vergleichbar, wenn sein **Typ** funktionsfrei, d.h. einer der folgenden Typen ist:

$$c ::= \text{bool} \mid \text{int} \mid \text{unit} \mid c_1 * \dots * c_k \mid c \text{ list}$$

:-)

Für Ausdrücke e_1, e_2, e mit funktionsfreien Typen können wir **Schlussregeln** angeben ...

Substitutionslemma:

$$\frac{e_1 \Rightarrow v' \quad e_2 \Rightarrow v' \quad e[e_1/x] \Rightarrow v}{e[e_2/x] \Rightarrow v}$$

Beachte:

$$e_1 = e_2 \Rightarrow \text{true} \iff e_1 \Rightarrow v' \wedge e_2 \Rightarrow v' \text{ für ein } v' \quad :-)$$

Wir folgern:

$$\frac{e_1 = e_2 \Rightarrow \text{true} \quad e[e_1/x] \text{ terminiert}}{e[e_1/x] = e[e_2/x] \Rightarrow \text{true}}$$

Diskussion:

- Das Lemma besagt damit, dass wir in **jedem Kontext** alle Vorkommen eines Ausdrucks e_1 durch einen Ausdruck e_2 ersetzen können, sofern e_1 und e_2 die selben Werte liefern :-)
- Das Lemma lässt sich mit Induktion über die Tiefe der benötigten Herleitungen zeigen (was wir uns sparen :-))
- Der Austausch von als gleich erwiesenen Ausdrücken ist die Grundlage unserer Methode zum Nachweis der **Äquivalenz** von Ausdrücken ...

10.3 Beweise für MiniOcaml-Programme

Beispiel 1:

```
let rec app = fun x -> fun y -> match x
                        with [] -> y
                        | x::xs -> x :: app xs y
```

Wir wollen nachweisen:

- (1) $\text{app } x \ [] = x$ für alle Listen x .
- (2) $\text{app } x \ (\text{app } y \ z) = \text{app } (\text{app } x \ y) \ z$
für alle Listen x, y, z .

Idee: Induktion nach der Länge n von x

$n = 0$: Dann gilt: $x = []$

Wir schließen:

$$\begin{aligned} \text{app } x \ [] &= \text{app } [] \ [] \\ &= [] \\ &= x \quad \text{:-)} \end{aligned}$$

$n > 0$: Dann gilt: $x = h::t$ wobei t Länge $n - 1$ hat.

Wir schließen:

$$\begin{aligned} \text{app } x \ [] &= \text{app } (h::t) \ [] \\ &= h :: \text{app } t \ [] \\ &= h :: t \quad \text{nach Induktionsannahme} \\ &= x \quad \text{:-))} \end{aligned}$$

Analog gehen wir für die Aussage (2) vor ...

$n = 0$: Dann gilt: $x = []$

Wir schließen:

$$\begin{aligned} \text{app } x \text{ (app } y \text{ } z) &= \text{app } [] \text{ (app } y \text{ } z) \\ &= \text{app } y \text{ } z \\ &= \text{app (app } [] \text{ } y) \text{ } z \\ &= \text{app (app } x \text{ } y) \text{ } z \quad \text{:)} \end{aligned}$$

$n > 0$: Dann gilt: $x = h::t$ wobei t Länge $n - 1$ hat.

Wir schließen:

$$\begin{aligned} \text{app } x \text{ (app } y \text{ } z) &= \text{app (h::t) (app } y \text{ } z) \\ &= h :: \text{app } t \text{ (app } y \text{ } z) \\ &= h :: \text{app (app } t \text{ } y) \text{ } z \quad \text{nach Induktionsannahme} \\ &= \text{app (h :: app } t \text{ } y) \text{ } z \\ &= \text{app (app (h::t) } y) \text{ } z \\ &= \text{app (app } x \text{ } y) \text{ } z \quad \text{:))} \end{aligned}$$

Diskussion:

- Bei den Gleichheitsumformungen haben wir einfache Zwischenschritte weggelassen :-)
- Eine Aussage: $\text{exp1} = \text{exp2}$
steht für: $\text{exp1} = \text{exp2} \Rightarrow \text{true}$
und schließt damit die Terminierung der Auswertungen von exp1 , exp2 ein.
- Zur Korrektheit unserer Induktionsbeweise benötigen wir, dass sämtliche vorkommenden Funktionsaufrufe **terminieren**.
- Im Beispiel reicht es zu zeigen, dass für alle x , y ein v existiert mit:

$$\text{app } x \text{ } y \Rightarrow v$$

... das geht natürlich wieder mit **Induktion** :-)

$n = 0 :$ Dann gilt: $x = []$

Wir schließen:

$$\begin{aligned} \text{app } x \ y &= \text{app } [] \ y \\ &\Rightarrow y \end{aligned}$$

Das heißt, die Behauptung gilt für $v = y$:-)

$n > 0 :$ Dann gilt: $x = h :: t$ für ein t der Länge $n - 1$.
Nach **Induktionsannahme** gibt es darum ein v' mit:

$$\text{app } t \ y \Rightarrow v'$$

Wir schließen ...

$$\frac{\frac{\text{app } = \text{fun } x \rightarrow \dots}{\text{app } \Rightarrow \text{fun } x \rightarrow \dots} \quad \frac{\text{app } t \ y \Rightarrow v'}{h :: \text{app } t \ y \Rightarrow h :: v'}}{\text{app } x \Rightarrow \text{fun } y \rightarrow \dots \quad \text{match } x \dots \Rightarrow h :: v'}}{\text{app } x \ y \Rightarrow h :: v'}$$

// Benutzung von Axiomen $z \Rightarrow z$ haben wir weggelassen :-)

Das heißt, die Behauptung gilt für $v = h :: v'$:-))

Beispiel 2:

```
let rec rev = fun x -> match x
  with [] -> []
  | x::xs -> app (rev xs) [x]
let rec rev1 = fun x -> fun y -> match x
  with [] -> y
  | x::xs -> rev1 xs (x::y)
```

Behauptung:

$\text{rev } x = \text{rev1 } x []$ für alle Listen x .

Allgemeiner:

$\text{app } (\text{rev } x) y = \text{rev1 } x y$ für alle Listen x, y .

Beweis: Induktion nach der Länge n von x

$n = 0$: Dann gilt: $x = []$

Wir schließen:

$$\begin{aligned} \text{app } (\text{rev } x) y &= \text{app } (\text{rev } []) y \\ &= \text{app } [] y \\ &= y \\ &= \text{rev1 } [] y \\ &= \text{rev1 } x y \quad \text{:)} \end{aligned}$$

$n > 0$: Dann gilt: $x = h::t$ wobei t Länge $n - 1$ hat.

Wir schließen:

$$\begin{aligned} \text{app } (\text{rev } x) y &= \text{app } (\text{rev } (h::t)) y \\ &= \text{app } (\text{app } (\text{rev } t) [h]) y \\ &= \text{app } (\text{rev } t) (\text{app } [h] y) \quad \text{wegen Beispiel 1} \\ &= \text{app } (\text{rev } t) (h::y) \\ &= \text{rev1 } t (h::y) \quad \text{nach Induktionsvoraussetzung} \\ &= \text{rev1 } (h::t) y \\ &= \text{rev1 } x y \quad \text{:))} \end{aligned}$$

Diskussion:

- Wieder haben wir implizit die Terminierung der Funktionsaufrufe von `app`, `rev` und `rev1` angenommen :-)
- Deren Terminierung können wir jedoch leicht mittels Induktion nach der Tiefe des ersten Arguments nachweisen.
- Die Behauptung:
folgt aus:
$$\begin{aligned} \text{rev } x &= \text{rev1 } x [] \\ \text{app } (\text{rev } x) \ y &= \text{rev1 } x \ y \end{aligned}$$

indem wir: `y = []` setzen und Aussage (1) aus **Beispiel 1** benutzen :-)

Beispiel 3:

```
let rec sorted = fun x -> match x
  with x1::x2::xs -> (match x1 <= x2
    with true -> sorted (x2::xs)
      | false -> false)
    | _ -> true

and merge = fun x -> fun y -> match (x,y)
  with ([],y) -> y
    | (x,[]) -> x
    | (x1::xs,y1::ys) -> (match x1 <= y1
      with true -> x1 :: merge xs y
        | false -> y1 :: merge x ys
```

Behauptung:

$\text{sorted } x \wedge \text{sorted } y \rightarrow \text{sorted } (\text{merge } x \ y)$
für alle Listen `x`, `y`.

Beweis: Induktion über die **Summe** n der Längen von `x`, `y` :-)

Gelte $\text{sorted } x \wedge \text{sorted } y$.

$n = 0$: Dann gilt: $x = [] = y$

Wir schließen:

$$\begin{aligned} \text{sorted } (\text{merge } x \ y) &= \text{sorted } (\text{merge } [] \ []) \\ &= \text{sorted } [] \\ &= \text{true} \quad \text{:-)} \end{aligned}$$

$n > 0$:

Fall 1: $x = []$.

Wir schließen:

```
sorted (merge x y) = sorted (merge [] y)
                  = sorted y
                  = true   :-)
```

Fall 2: $y = []$ analog :-)

Fall 3: $x = x1::xs \wedge y = y1::ys \wedge x1 \leq y1$.

Wir schließen:

```
sorted (merge x y) = sorted (merge (x1::xs) (y1::ys))
                  = sorted (x1 :: merge xs y)
                  = ...
```

Fall 3.1: $xs = []$

Wir schließen:

```
... = sorted (x1 :: merge [] y)
    = sorted (x1 :: y)
    = sorted y
    = true   :-)
```

Fall 3.2: $xs = x2::xs' \wedge x2 \leq y1$.

Insbesondere gilt: $x1 \leq x2 \wedge \text{sorted } xs$.

Wir schließen:

```
... = sorted (x1 :: merge (x2::xs') y)
    = sorted (x1 :: x2 :: merge xs' y)
    = sorted (x2 :: merge xs' y)
    = sorted (merge xs y)
    = true nach Induktionsannahme :-)
```

Fall 3.3: $xs = x2 :: xs' \wedge x2 > y1$.

Insbesondere gilt: $x1 \leq y1 < x2 \wedge \text{sorted } xs$.

Wir schließen:

```
... = sorted (x1 :: merge (x2::xs') (y1::ys))
     = sorted (x1 :: y1 :: merge xs ys)
     = sorted (y1 :: merge xs ys)
     = sorted (merge xs y)
     = true nach Induktionsannahme :-)
```

Fall 4: $x = x1 :: xs \wedge y = y1 :: ys \wedge x1 > y1$.

Wir schließen:

```
sorted (merge x y) = sorted (merge (x1::xs) (y1::ys))
                  = sorted (y1 :: merge x ys)
                  = ...
```

Fall 4.1: $ys = []$

Wir schließen:

```
... = sorted (y1 :: merge x [])
     = sorted (y1 :: x)
     = sorted x
     = true :-)
```

Fall 4.2: $ys = y2 :: ys' \wedge x1 > y2$.

Insbesondere gilt: $y1 \leq y2 \wedge \text{sorted } ys$.

Wir schließen:

```
... = sorted (y1 :: merge x (y2::ys'))
     = sorted (y1 :: y2 :: merge x ys')
     = sorted (y2 :: merge x ys')
     = sorted (merge x ys)
     = true nach Induktionsannahme :-)
```

Fall 4.3: $ys = y2 :: ys' \wedge x1 \leq y2$.

Insbesondere gilt: $y1 < x1 \leq y2 \wedge \text{sorted } ys$.

Wir schließen:

```
... = sorted (y1 :: merge (x1::xs) (y2::ys'))
     = sorted (y1 :: x1 :: merge xs ys)
     = sorted (x1 :: merge xs ys)
     = sorted (merge x ys)
     = true nach Induktionsannahme :-))
```

Diskussion:

- Wieder steht der Beweis unter dem Vorbehalt, dass alle Aufrufe der Funktionen `sorted` und `merge` terminieren :-)
- Als zusätzliche Technik benötigten wir **Fallunterscheidungen** über die verschiedenen Möglichkeiten für Argumente in den Aufrufen :-)
- Die Fallunterscheidungen machten den Beweis länglich :-(
// Der Fall $n = 0$ ist tatsächlich überflüssig,
// da er in den Fällen 1 und 2 enthalten ist :-)

Ausblick:

- In Programmoptimierungen möchten wir gelegentlich **Funktionen** austauschen, z.B.

$$\text{comp } (\text{map } f) (\text{map } g) = \text{map } (\text{comp } f g)$$

- Offenbar stehen rechts und links des **Gleichheitszeichens** Funktionen, deren Gleichheit **Ocaml** nicht überprüfen kann

⟹

Die Logik benötigt einen **stärkeren** Gleichheitsbegriff :-)

- Wir **erweitern** die **Ocaml**-Gleichheit $=$ so dass gilt:

□ $x = y$ falls x und y nicht terminieren :-)

□ $f = g$ falls $f x = g x$ für alle x

⟹ **extensionale Gleichheit**

Wir haben:

- Seien der Typ von e_1, e_2 **funktionsfrei**. Dann gilt:

$$\frac{e_1 = e_2 \quad e_1 \text{ terminiert}}{e_1 = e_2 \Rightarrow \text{true}}$$

- Außerdem gibt es eine stark vereinfachte **Substitutionsregel**:

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

// Die Annahmen über Terminierung entfallen :-)

\implies Die Beweistechnik vereinfacht sich :-))

11 Parallele Programmierung

Die Bibliothek `threads.cma` unterstützt die Implementierung von Systemen, die mehr als einen Thread benötigen :-)

Beispiel:

```
module Echo = struct open Thread
  let echo () = print_string (read_line () ^ "\n")
  let main    = let t1 = create echo ()
                in join t1;
                print_int (id (self ()));
                print_string "\n"
end
```

Kommentar:

- Die Struktur `Thread` versammelt Grundfunktionalität zur Erzeugung von Nebenläufigkeit :-)
- Die Funktion `create: ('a -> 'b) -> 'a -> t` erzeugt einen neuen Thread mit den folgenden Eigenschaften:
 - der Thread wertet die Funktion auf dem Argument aus;
 - der erzeugende Thread erhält die Thread-Id zurück und läuft unabhängig weiter.
 - Mit den Funktionen: `self : unit -> t` bzw. `id : t -> int` kann man die eigene Thread-Id abfragen bzw. in ein `int` umwandeln.

Weitere nützliche Funktionen:

- Die Funktion `join: t -> unit` hält den aktuellen Thread an, bis die Berechnung des gegebenen Threads beendet ist.
- Die Funktion: `kill: t -> unit` beendet einen Thread;
- Die Funktion: `delay: float -> unit` verzögert den aktuellen Thread um eine Zeit in Sekunden;
- Die Funktion: `exit: unit -> unit` beendet den aktuellen Thread.

Achtung:

- Die interaktive Umgebung funktioniert nicht mit Threads !!
- Stattdessen muss man mit der Option: `-thread` compilieren:
`> ocamlc -thread unix.cma threads.cma Echo.ml`
- Die Bibliothek `threads.cma` benötigt dabei Hilfsfunktionalität der Bibliothek `lueunix.cma` :-)
`//` unter Windows sieht die Sache vermutlich anders aus :-))
- Das Programm testen können wir dann durch Aufruf von:
`> ./a.out` :-)

```
> ./a.out
> abcdefghijk
> abcdefghijk
> 0
>
```

- **Ocaml**-Threads werden vom System nur simuliert :-)
- Die Erzeugung von Threads ist **billig** :-))
- Die Programm-Ausführung endet mit der Terminierung des Threads mit der Id `0` .

11.1 Kanäle

Threads kommunizieren über Kanäle :-)

Für Erzeugung, Senden auf und Empfangen aus einem Kanal stellt die Struktur `Event` die folgende Grundfunktionalität bereit:

```
type 'a channel
new_channel : unit -> 'a channel
type 'a event
always : 'a -> 'a event
sync : 'a event -> 'a
send : 'a channel -> 'a -> unit event
receive : 'a channel -> 'a event
```

- Jeder Aufruf `new_channel()` erzeugt einen anderen Kanal.
- Über einen Kanal können **beliebige** Daten geschickt werden !!!
- `always` wandelt einen Wert in ein **Ereignis** um.
- Senden und Empfangen sind erzeugen **Ereignisse ...**
- **Synchronisierung** auf Ereignisse liefert deren **Wert :-)**

```

module Exchange = struct open Thread open Event
let thread ch = let x = sync (receive ch)
                in print_string (x ^ "\n");
                sync (send ch "got it!")
let main = let ch = new_channel () in create thread ch;
           print_string "main is running ... \n";
           sync (send ch "Greetings!");
           print_string ("He " ^ sync (receive ch) ^ "\n")
end

```

Diskussion:

- `sync (send ch str)` macht das **Ereignis** des Sendens der Welt offenbar und **blockiert** den Sender, bis jemand den Wert aus dem Kanal ausgelesen hat ...
- `sync (receive ch)` blockiert den Empfänger, bis ein Wert im Kanal enthalten ist. Dann liefert der Ausdruck diesen Wert **:-)**
- Synchrone Kommunikation ist eine Alternative zum Austausch von Daten zwischen Threads bzw. zur Organisation von Nebenläufigkeit \implies **Rendezvous**
- Insbesondere kann sie benutzt werden, um asynchrone Thread-Kooperation zu implementieren **:-)**

Im Beispiel spaltet `main` einen Thread ab. Dann sendet sie diesem einen String und wartet auf Antwort. Entsprechend wartet der Thread auf Übertragung eines **string**-Werts auf dem Kanal. Sobald er ihn erhalten hat, sendet er auf dem **selben Kanal** eine Antwort.

Achtung!

Ist die Abfolge von `send` und `receive` nicht sorgfältig designet, können Threads leicht blockiert werden ...

Die Ausführung des Programms liefert:

```
> ./a.out
main is sending ...Greetings!
He got it!
>
```

Beispiel: Eine globale Speicherzelle

Eine globale Speicherzelle, insbesondere in Anwesenheit mehrerer Threads sollte die Signatur `Cell` implementieren:

```
module type Cell = sig
  type 'a cell
  val new_cell : 'a -> 'a cell
  val get : 'a cell -> 'a
  val put : 'a cell -> 'a -> unit
end
```

Dabei muss sichergestellt werden, die verschiedenen `get`- und `put`-Aufrufe sequenzialisiert ausgeführt werden.

Diese Aufgabe erfüllt ein `Server`-Thread, mit dem `get` und `put` kommunizieren:

```
type 'a req = Get of 'a channel | Put of 'a
type 'a cell = 'a req channel
```

Der Kanal transportiert Requests an die Speicherzelle, welche entweder den zu setzenden Wert oder den Rückkanal enthalten ...

```
let get req = let reply = new_channel ()
              in sync (send req (Get reply));
              sync (receive reply)
```

Die Funktion `get` sendet einen neuen Rückkanal auf `req`. Ist dieser angekommen, wartet sie auf die Antwort.

```
let put req x = sync (send req (Put x))
```

Die Funktion `put` sendet ein `Put`-Element, das den neuen Wert der Speicherzelle enthält.

Spannend ist jetzt nur noch die Implementierung der Zelle selbst:

```
let new_cell x = let req = new_channel ()
  in let rec serve x = match sync (receive req)
    with Get reply -> sync (send reply x);
      serve x
    | Put y      -> serve y
  in
    create serve x;
    req
```

Beim Anlegen der Zelle mit dem Wert `x` wird ein Server-Thread abgespalten, der den Aufruf `serve x` auswertet.

Achtung:

der Server-Thread ist potentiell nicht-terminierend!

Nur deshalb kann er beliebig viele Requests bearbeiten :-)

Nur weil er `end-rekursiv` programmiert ist, schreibt er dabei nicht sukzessive den gesamten Speicher voll ...

```
let main = let x = new_cell 1
  in print_int (get x); print_string "\n";
    put x 2;
    print_int (get x); print_string "\n"
```

Dann liefert die Ausführung:

```
> ./a.out
1
2
>
```

Anstelle von `get` und `put` könnte man auch kompliziertere Anfrage- bzw. Update-Funktionen vom `cell`-Server ausführen lassen ...

Beispiel: Locks

Oft soll von mehreren möglicherweise aktiven Threads nur ein einziger auf eine bestimmte Resource zugreifen. Um solchen **wechselseitigen Ausschluss** zu implementieren, kann man Locks verwenden:

```
module type Lock = sig
  type lock
  type ack
  val new_lock : unit -> lock
  val acquire : lock -> ack
  val release : ack -> unit
end
```

Ausführen der Operation `acquire` liefert ein Element des Typs `ack`, dessen Rückgabe den Lock wieder frei gibt:

```
type ack = unit channel
type lock = ack channel
```

Der Einfachheit halber wählen wir `ack` einfach als den Kanal, über den die Freigabe des Locks erfolgen soll :-)

```
let acquire lock = let ack = new_channel ()
                  in sync (send lock ack);
                  ack
```

Der Freigabe-Kanal wird von `acquire` hergestellt :-)

```
let release ack = sync (send ack ())
```

... und von der Operation `release` benutzt.

```

let new_lock () = let lock = new_channel ()
                  in let rec ack_server () =
                      rel_server (sync (receive lock))
                    and rel_server ack =
                      sync (receive ack);
                      ack_server ()
                  in create ack_server ();
                  lock

```

Herzstück der Implementierung sind die beiden wechselseitig rekursiven Funktionen `ack_server` und `rel_server`.

`ack_server` erwartet ein `ack`-Element, d.h. einen Kanal, und ruft nach Erhalt `rel_server` auf.

`rel_server` erwartet über den erhaltenen Kanal ein Signal, dass die Resource auch freigegeben wurde ...

Jetzt sind wir in der Lage, einen anständigen `Deadlock` zu programmieren:

```

let dead = let l1 = new_lock ()
           in let l2 = new_lock ()
           ...

in let th (l1,l2) = let a1 = acquire l1
                  in let _ = delay 1.0
                  in let a2 = acquire l2
                  in release a2; release a1;
                  print_int (id (self ()));
                  print_string " finished\n"
in let t1 = create th (l1,l2)
in let t2 = create th (l2,l1)
in join t1

```

Das Ergebnis ist:

```
> ./a.out
```

Ocaml wartet und wartet :-)

Beispiel: Semaphore

Gelegentlich gibt es mehr als ein Exemplar einer Resource.
Dann sind [Semaphore](#) ein geeignetes Hilfsmittel ...

```
module type Sema = sig
  type sema
  new_sema : int -> sema
  up : sema -> unit
  down : sema -> unit
end
```

Idee:

Wir implementieren wieder einen Server, der in einem akkumulierenden Parameter die Anzahl der noch zur Verfügung stehenden Ressourcen bzw. eine Schlange der wartenden Threads enthält ...

```
module Sema = struct open Thread Event
  type sema = unit channel option channel
  let up sema = sync (send sema None)
  let down sema = let ack = (new_channel : unit channel)
                  in sync (send sema (Some ack));
                  sync (receive ack)
  ...
  ...
  let new_sema n = let sema = new_channel ()
                  in let rec serve (n,q) =
                      match sync (receive sema)
                      with None -> (match dequeue q
                                   with (None,q) -> serve (n+1,q)
                                    | (Some ack,q) -> sync (send ack ());
                                    serve (n,q))
                      | Some ack -> if n>0 then sync (send ack ());
                                   serve (n-1,q)
```

```

                else serve (n,enqueue ack q)
            in create serve (n,new_queue()); sema
        end
    end

```

Offensichtlich verwalten wir in der Schlange nicht die Threads selbst, sondern ihre jeweiligen Rückantwort-Kanäle :-)

11.2 Selektive Kommunikation

Häufig weiß ein Thread nicht, welche von mehreren möglichen Kommunikations-Rendezvous zuerst oder überhaupt eintrifft.

Nötig ist eine **nicht-deterministische Auswahl** zwischen mehreren Aktionen ...

Beispiel: Die Funktion

```
add : int channel * int channel * int channel -> unit
```

soll Integers aus zwei Kanälen lesen und die Summe auf dem dritten senden.

1. Versuch:

```

let forever f init = let
    let rec loop x = loop (f x)
    in create loop init
end
let add1 (in1, in2, out) = forever (fun () ->
    sync (send out (sync (receive in1) +
        sync (receive in2)))
    )) ()

```

Nachteil:

Kommt am zweiten Input-Kanal bereits früher ein Wert an, muss der Thread trotzdem warten.

2. Versuch:

```

let add (in1, in2, out) = forever (fun () ->
    let (a,b) = select [

```

```

wrap (receive in1) (fun a -> (a, sync (receive in2)));
wrap (receive in2) (fun b -> (sync (receive in1), b))
]
in sync (send out (a+b))
) ()

```

Dieses Programm müssen wir uns langsam auf der Zunge zergehen lassen :-)

Idee:

- Initiierung von Input- wie Output-Operationen erzeugen **Events**.
- Events sind Daten-Objekte des Typs: `'a event`.
- Die Funktion:

```
wrap : 'a event -> ('a -> 'b) -> 'b event
```

wendet eine Funktion **nachträglich** auf den Wert eines Events – sollte er denn eintreffen – an.

Die Liste enthält damit `(int, int)`-Events.

Die Funktionen:

```
choose : 'a event list -> 'a event
select : 'a event list -> 'a
```

wählen **nicht-deterministisch** ein Event einer Event-Liste aus.
`select` synchronisiert anschließend auf den ausgewählten Event,
d.h. stellt die nötige Kommunikation her und liefert den Wert zurück:

```
let select = comp sync choose
```

Typischerweise wird dabei dasjenige Event ausgewählt, das zuerst einen Partner findet :-)

Weitere Beispiele:

Die Funktion

```
copy : 'a channel * 'a channel * 'a channel -> unit
```

soll ein gelesenes Element auf zwei Kanäle kopieren:

```

let copy (in, out1, out2) = forever (fun () ->
  let x = sync (receive in)
  in select [
    wrap (send out1 x)
      (fun () -> sync (send out2 x));
    wrap (send out2 x)
      (fun () -> sync (send out1 x))
  ]
) ()

```

Offenbar kann die Event-Liste auch aus **Sende-Events** bestehen
– oder sogar beide Sorten enthalten :-)

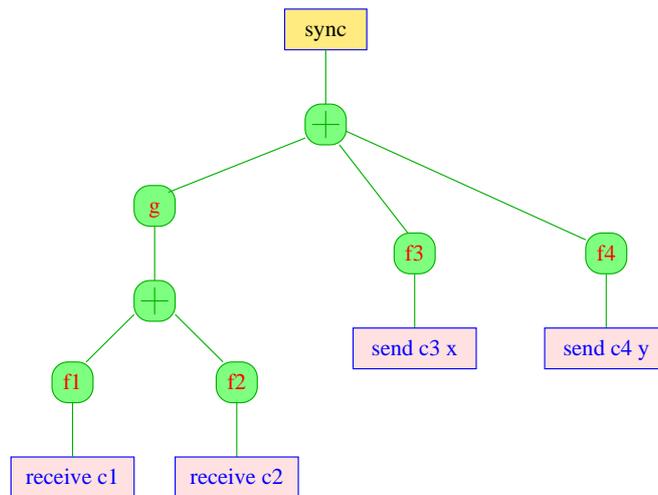
```

type 'a cell = 'a channel * 'a channel
...

...
let get (get_chan,_) = sync (receive get_chan)
let put (_,put_chan) x = sync (send put_chan x)
let new_cell x = let get_chan = new_channel ()
  in let put_chan = new_channel ()
  in let serve x = select [
    wrap (send get_chan x) (fun () -> serve x);
    wrap (receive put_chan) serve
  ]
  in
  create serve x;
  (get_chan, put_chan)

```

Im allgemeinen kann ein Event-Baum vorliegen:



- Die Blätter sind Basis-Events.
- Auf ein Event kann eine Wrapper-Funktion angewandt werden.
- Mehrere Ereignisse des gleichen Typs können mit einem Auswahl-Knoten zusammengefasst werden.
- Synchronisierung auf einen Event-Baum aktiviert ein Blatt-Event.
Als Ergebnis wird der Wert geliefert, den die Komposition der Wrapper-Funktionen auf dem Weg zur Wurzel liefert.

Beispiel: Ein Swap-Channel

Ein Swap-Channel soll beim Rendezvous die Werte der beiden beteiligten Threads austauschen.

Die Signatur lautet:

```
module type Swap = sig
  type 'a swap
  val new_swap : unit -> 'a swap
  val swap : 'a swap -> 'a -> 'a event
end
```

Jeder beteiligte Thread muss in einer Implementierung mit normalen Kanälen sowohl die Möglichkeit zu empfangen wie zu senden anbieten.

Sobald ein Thread sein Senden erfolgreich beendete (d.h. der andere auf ein `receive`-Event synchronisierte), muss noch der zweite Wert übermittelt werden.
 Mit dem ersten Wert übertragen wir deshalb einen Kanal für den zweiten Wert:

```

module Swap =
struct open Thread open Event
  type 'a swap = ('a * 'a channel) channel
  let new_swap () = new_channel ()
  ...

  ...
  let swap ch x = let c = new_channel ()
    in choose [
      wrap (receive ch) (fun (y,c) ->
        sync (send c x); y);
      wrap (send ch (x,c)) (fun () ->
        sync (receive c))
    ]
  ]

```

Einen konkreten Austausch können wir implementieren, indem wir `choose` in `select` umwandeln.

Timeouts

Oft dauert unsere Geduld nur eine Weile :-)
 Dann wollen wir ein angefangenes Sende- oder Empfangswarten beenden ...

```

module type Timer = sig
  set_timer : float -> unit event
  timed_receive : 'a channel -> float -> 'a option event
  timed_send : 'a channel -> 'a -> float -> unit option event
end

```

```

module Timer = struct open Thread Event
  let set_timer t = let ack = new_channel ()
    in let serve () = delay t;
      sync (receive ack)
    in create serve (); send ack ()
  ]

```

```

let timed_receive ch time = choose [
  wrap (receive ch) (fun a -> Some a);
  wrap (set_timer time) (fun () -> None)
]

let timed_send ch x time = choose [
  wrap (send ch x) (fun a -> Some ());
  wrap (set_timer time) (fun () -> None)
]
end

```

11.3 Threads und Exceptions

Eine Exception muss immer innerhalb des Threads behandelt werden, in der sie erzeugt wurde.

```

module Blow = struct open Thread
let thread x = (x / 0);
  print_string "thread terminated regularly ...\n"
let main = create thread 0; delay 1.0;
  print_string "main terminated regularly ...\n"
end

```

... liefert:

```

> ./a.out
Thread 1 killed on uncaught exception Division_by_zero
main terminated regularly ...

```

Der Thread wurde getötet, das **Ocaml**-Programm terminierte trotzdem.

Auch ungefangene Exceptions innerhalb einer Wrapper-Funktion beenden den laufenden Thread:

```

module BlowWrap = struct open Thread open Event open Timer
let main = try sync (wrap (set_timer 1.0) (fun () -> 1 / 0))
  with _ -> 0;
  print_string "... this is the end!\n"
end

```

Dann liefert:

```
> ./a.out
Fatal error: exception Division_by_zero
```

Achtung:

Exceptions können nur im Rumpf der Wrapper-Funktion selbst, nicht aber hinter dem `sync` abgefangen werden !

11.4 Gepufferte Kommunikation

Ein Kanal für gepufferte Kommunikation erlaubt **nicht blockierendes** Senden. Empfangen dagegen blockiert, sofern keine Nachrichten vorhanden ist. Für solche Kanäle implementieren wir die Struktur `Mailbox` zur Verfügung:

```
module type Mailbox = sig
  type 'a mbox
  val new_mailbox : unit -> 'a mbox
  val send : 'a mbox -> 'a -> unit
  val receive : 'a mbox -> 'a event
end
```

Zur Implementierung benutzen wir einen Server, der eine Queue der gesendeten, aber noch nicht erhaltenen Nachrichten verwaltet.

Damit implementieren wir:

```
module Mailbox =
struct open Thread open Queue open Event
  type 'a mbox = 'a channel * 'a channel
  let send (in_chan,_) x = sync (send in_chan x)
  let receive (_,out_chan) = receive out_chan
  let new_mailbox () = let in_chan = new_channel ()
                       and out_chan = new_channel ()
  ...
end
```

```

...
    in let rec serve q = if (is_empty q) then
        serve (enqueue (
            sync (Event.receive in_chan)) q)
        else select [
            wrap (Event.receive in_chan)
                (fun y -> serve (enqueue y q));
            wrap (Event.send out_chan (first q))
                (fun () -> let (_,q) = dequeue q
                    in serve q)
        ]
    in create serve (new_queue ());
        (in_chan, out_chan)
end

```

... wobei `first : 'a queue -> 'a` das erste Element einer Schlange liefert,
ohne es zu entfernen `:-)`

11.5 Multicasts

Für das Versenden einer Nachricht an **viele** Empfänger stellen wir die Struktur `Multicast` zur Verfügung, die die Signatur `Multicast` implementiert:

```

module type Multicast = sig
  type 'a mchannel and 'a port
  val new_mchannel : unit -> 'a mchannel
  val new_port : 'a mchannel -> 'a port
  val multicast : 'a mchannel -> 'a -> unit
  val receive : 'a port -> 'a event
end

```

Die Operation `new_port` erzeugt einen neuen Port,
an dem eine Nachricht empfangen werden kann.

Die Operation `multicast` sendet (nicht-blockierend) an sämtliche registrierten Ports.

```

module Multicast = struct open Thread open Event
module M = Mailbox
type 'a port = 'a M.mbox
type 'a mchannel = 'a channel * unit channel
                    * 'a port channel
let new_port (_, req, port) = sync (send req ());

```

```

                                sync (receive port)
let multicast (send_ch,_,_) x = sync (send send_ch x)
let receive mbox                = M.receive mbox
...

```

Die Operation `multicast` sendet die Nachricht auf dem Kanal `send_ch`.
 Die Operation `receive` liest aus der Mailbox des Ports.
 Der Multicast-Kanal selbst wird von einem Server-Thread überwacht,
 der eine Liste der zu bedienenden Ports verwaltet:

```

let new_mchannel () = let send_ch = new_channel ()
                      in let req = new_channel ()
                      in let port = new_channel ()
                      in let send_port x mbox = M.send mbox x
...

...
in let rec serve ports = select [
  wrap (Event.receive req) (fun () ->
    let p = M.new_mailbox ()
    in sync (send port p);
    serve (p :: ports));
  wrap (Event.receive send_ch) (fun x ->
    create (iter (send_port x)) ports;
    serve ports)
]
in create serve [];
(send_ch, req, port)
...

```

Beachte, dass der Server-Thread sowohl auf Port-Requests auf dem Kanal `req` wie auf Sende-Aufträge auf dem Kanal `send_ch` gefasst sein muss.

Achtung:

Unsere Implementierung gestattet zwar das Hinzufügen,
 nicht aber das Entfernen nicht mehr benötigter Ports.
 Zum Ausprobieren benutzen wir einen Test-Ausdruck `main`:

```

...
let main = let mc = new_mchannel ()
  in let thread i = let p = new_port mc
    in while true do let x = sync (receive p)
      in print_int i; print_string ": ";
        print_string (x^"\n")
    done
  in create_thread 1; create_thread 2;
    create_thread 3; delay 1.0;
    multicast mc "Hallo!";
    multicast mc "World!";
    multicast mc "... the end.";
    delay 10.0
end
end

```

Dann haben wir:

```

- ./a.out
3: Hallo!
2: Hallo!
1: Hallo!
3: World!
2: World!
1: World!
3: ... the end.
2: ... the end.
1: ... the end.

```

Fazit:

- Die Programmiersprache **Ocaml** bietet komfortable Möglichkeiten an, nebenläufige Programme zu schreiben :-)
- Kanäle mit synchroner Kommunikation können Konzepte der Nebenläufigkeit wie asynchrone Kommunikation, globale Variablen, Locks für wechselseitigen Ausschluss und Semaphore simulieren :-)
- Nebenläufige funktionale Programme können deshalb genauso undurchsichtig und schwer zu verstehen sein wie nebenläufige **Java**-Programme :-)
- Es werden Methoden benötigt, um systematisch die Korrektheit solcher Programme zu verifizieren ...

Abschluss:

- Jenseits der hier besprochenen Sprachkonzepte gibt es in **Ocaml** einige weitere Konzepte, die insbesondere **objekt-orientierte** Programmierung ermöglichen.
- Darüberhinaus bietet **Ocaml** elegante Möglichkeiten, Betriebssystemsfunktionalität auszunutzen, graphische Bibliotheken anzusteuern, mit anderen Rechnern zu kommunizieren ...

⇒ **Ocaml** ist eine interessante Alternative zu **Java**.

12 Datalog: Rechnen mit Relationen

Beispiel 1: Das Lehrangebot einer TU



⇒ Entity-Relationship Diagram

Diskussion:

- Viele Anwendungsbereiche lassen sich mit Hilfe von Entity-Relationship-Diagrammen beschreiben.
- Entitäten im Beispiel: **Dozent**, **Vorlesung**, **Student**.
- Die Menge aller **vorkommenden** Entitäten d.h. Instanzen lassen sich mit einer Tabelle beschreiben ...

Dozent :

Name	Telefon	Email
Brauer	17204	brauer@in.tum.de
Nipkow	17302	nipkow@in.tum.de
Seidl	18155	seidl@in.tum.de

Vorlesung:

Titel	Raum	Zeit
Diskrete Strukturen II	MI 1	Do 12:15-13, Fr 10-11:45
Perlen der Informatik II	MI 3	Do 8:30-10
Einführung in die Informatik II	MW 1802	Di 12-14, Fr 11:45-13:15
Compilerbau	MI 2	Mo 12-14, Mi 10-12

Student:

Matr.nr.	Name	Sem.
123456	Hans Dampf	02
007042	Fritz Schluri	10
543345	Anna Blume	02
131175	Effi Briest	04

Diskussion (Forts.):

- Die Zeilen entsprechen den Instanzen.
- Die Spalten entsprechen den **Attributen**.
- **Annahme:** das erste Attribut **identifiziert** die Instanz
⇒ **Primärschlüssel**
Folgerung: Beziehungen sind ebenfalls Tabellen ...

liest:

Name	Titel
Brauer	Diskrete Strukturen II
Nipkow	Perlen der Informatik II
Seidl	Einführung in die Informatik II
Seidl	Compilerbau

hört:

Name	Titel
123456	Einführung in die Informatik II
123456	Compilerbau
123456	Diskrete Strukturen II
543345	Einführung in die Informatik II
543345	Diskrete Strukturen II
131175	Compilerbau

Mögliche Anfragen:

- In welchen Semestern sind die Studierenden der Vorlesung “Diskrete Strukturen II” ?
- Wer hört eine Vorlesung bei Dozent “Seidl” ?
- Wer hört sowohl “Diskrete Strukturen II” wie “Einführung in die Informatik II” ?

⇒ Datalog

Idee: Tabelle ⇔ Relation

Eine Relation R ist eine Menge von Tupeln, d.h.

$$R \subseteq \mathcal{U}_1 \times \dots \times \mathcal{U}_n$$

wobei \mathcal{U}_i die Menge aller möglicher Werte für die i -te Komponente ist.

In unserem Beispiel kommen etwa vor:

`int`, `string`, möglicherweise Aufzähltypen
// Einstellige Relationen sind Mengen :-)

Relationen können durch Prädikate beschrieben werden ...

Prädikate können wir definieren durch Aufzählung von Fakten ...

... im Beispiel:

```
liest ("Brauer", "Diskrete Strukturen II").  
liest ("Nipkow", "Perlen der Informatik II").  
liest ("Seidl", "Einführung in die Informatik II").  
liest ("Seidl", "Compilerbau").
```

```
hört (123456, "Compilerbau").  
hört (123456, "Einführung in die Informatik II").  
hört (123456, "Diskrete Strukturen").  
hört (543345, "Einführung in die Informatik II").  
hört (543345, "Diskrete Strukturen II").  
hört (131175, "Compilerbau").
```

Wir können aber auch Regeln benutzen, mit denen weitere Fakten abgeleitet werden können

...

... im Beispiel:

```
hat_Hörer (X,Y) :- liest (X,Z), hört (M,Z), student (M,Y,_).  
semester (X,Y) :- hört (Z,X), student (Z,_,Y).
```

- :- bezeichnet die logische Implikation “ \Leftarrow ”.
- Die komma-separierte Liste sammelt die Voraussetzungen.
- Die linke Seite, der Kopf der Regel, ist die Schlussfolgerung.
- Die Variablen werden groß geschrieben.
- Die anonyme Variable _ bezeichnet irrelevante Werte :-)

An die Wissensbasis aus Fakten und Regeln können wir jetzt Anfragen stellen ...

... im Beispiel:

```
?- hat_Hörer ("Seidl", Z).
```

- Datalog findet alle Werte für Z, für die die Anfrage aus den gegebenen Fakten mit Hilfe der Regeln beweisbar ist :-)
- In unserem Beispiel ist das:

```
Z = "Hans Dampf"
Z = "Anna Blume"
Z = "Effi Briest"
```

Weitere Anfragen:

```
?- semester ("Diskrete Strukturen II", X).
```

```
X = 2
```

```
X = 4
```

```
?- hört (X, "Einführung in die Informatik II"),
```

```
hört (X, "Diskrete Strukturen II").
```

```
X = 123456
```

```
X = 543345
```

Achtung:

Natürlich kann die Anfrage auch gar keine oder mehr als eine Variable enthalten :-)

Ein Beispiel-Beweis:

Die Regel:

```
hat_Hörer (X,Y) :- liest (X,Z), hört (M,Z), student (M,Y,_).
```

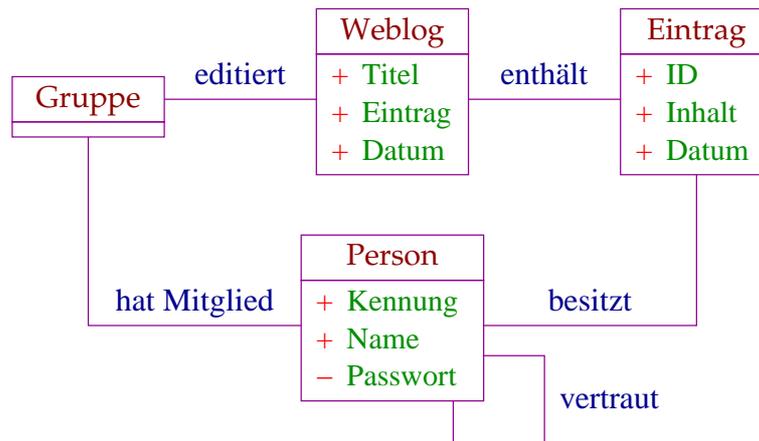
gilt für alle X, M, Y, Z . Mit Hilfe der Substitution:

```
"seidl"/X "Einführung ..."/Z 543345/M "Anna Blume"/Y
```

können wir schließen:

```
liest ("Seidl", "Einführung ...")
      hört (543345, "Einführung ....")
      student (543345, "Anna Blume", 2)
-----
hat_Hörer ("Seidl", "Anna Blume")
```

Beispiel 2: Ein Weblog



Aufgabe: Festlegung der Zugriffsberechtigung

- Jedes Mitglied der editierenden Gruppe darf einen neuen Eintrag hinzufügen.
- Nur die Besitzerin eines Eintrags darf ihn löschen.
- Modifizieren darf ihn jeder, dem die Besitzerin traut.
- Lesen darf ihn jedes Mitglied der Gruppe und jeder ihrer mittelbar Vertrauten ...

Spezifikation in Datalog:

```
darf_hinzufügen (X,W) :- editiert (Z,W),
                        hat_Mitglied (Z,X).
darf_löschen (X,E) :- besitzt (X,E).
darf_modifizieren (X,E) :- besitzt (X,E).
darf_modifizieren (X,E) :- darf_modifizieren (Y,E),
                        vertraut (Y,X).
darf_lesen (X,E) :- enthält (W,E),
                  darf_hinzufügen (X,W).
darf_lesen (X,E) :- darf_lesen (Y,E),
                  vertraut (Y,X).
```

Beachte:

- Zur Definition neuer Prädikate dürfen wir selbstverständlich alle vorhandenen benutzen oder sogar Hilfsprädikate definieren.
- Offenbar können Prädikatsdefinitionen auch **rekursiv** sein :-)
- Mit einer Person X , die einen Eintrag modifizieren darf, dürfen auch alle Personen modifizieren, denen X traut.

12.1 Beantwortung von Anfragen

Gegeben: eine Menge von Fakten und Regeln.

Gesucht: die Menge aller ableitbaren Fakten.

Problem:

```
equals (X,X).
```

\implies Die Menge aller ableitbaren Fakten ist nicht endlich :-)

Satz:

Sei W eine endliche Menge von Fakten und Regeln mit den folgenden Eigenschaften:

- (1) Fakten enthalten keine Variablen.
- (2) Jede Variable im Kopf kommt auch im Rumpf vor.

Dann ist die Menge der ableitbaren Fakten **endlich**.

Beweisskizze:

Man zeigt für jedes beweisbare Faktum $p(a_1, \dots, a_k)$, dass jede Konstante a_i bereits in W vorkommt :-))

Berechnung aller ableitbaren Fakten:

Berechne sukzessiv Mengen $R^{(i)}$ der Fakten, die mithilfe von Beweisen der Tiefe maximal i abgeleitet werden können ...

$$R^{(0)} = \emptyset \quad R^{(i+1)} = \mathcal{F}(R^{(i)})$$

wobei der Operator \mathcal{F} definiert ist durch:

$$\mathcal{F}(M) = \{h[\underline{a}/\underline{X}] \mid \exists h :- l_1, \dots, l_k. \in W : \\ l_1[\underline{a}/\underline{X}], \dots, l_k[\underline{a}/\underline{X}] \in M\}$$

// $[\underline{a}/\underline{X}]$ eine **Substitution** der Variablen \underline{X}
// k kann auch 0 sein :-)

Es gilt: $R^{(i)} = \mathcal{F}^i(\emptyset) \subseteq \mathcal{F}^{i+1}(\emptyset) = R^{(i+1)}$

Die Menge R aller implizierten Fakten ist gegeben durch:

$$R = \bigcup_{i \geq 0} R^{(i)} = R^{(n)}$$

für ein geeignetes n — da R endlich ist :-)

Beispiel:

```
edge (a,b).
edge (a,c).
edge (b,d).
edge (d,a).
t (X,Y) :- edge (X,Y).
t (X,Y) :- edge (X,Z), t (Z,Y).
```

Relation `edge` :

	a	b	c	d
a	blue	red	red	blue
b	blue	blue	blue	red
c	blue	blue	blue	blue
d	red	blue	blue	blue

$t^{(0)}$	a	b	c	d	$t^{(1)}$	a	b	c	d
a	blue	blue	blue	blue	a	blue	red	red	blue
b	blue	blue	blue	blue	b	blue	blue	blue	red
c	blue	blue	blue	blue	c	blue	blue	blue	blue
d	blue	blue	blue	blue	d	red	blue	blue	blue

$t^{(2)}$	a	b	c	d	$t^{(3)}$	a	b	c	d
a	blue	red	red	red	a	red	red	red	red
b	red	blue	blue	red	b	red	red	red	red
c	blue	blue	blue	blue	c	blue	blue	blue	blue
d	red	red	red	blue	d	red	red	red	red

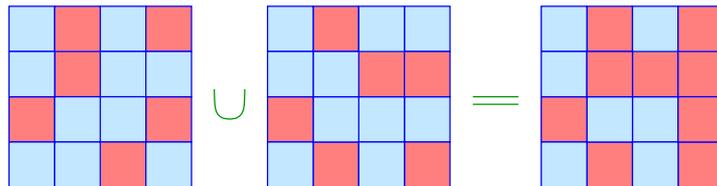
Diskussion:

- Unsere Überlegungen reichen aus, um für ein **Datalog**-Programm die Menge aller implizierten Fakten zu berechnen :-)
- Aus diesen können wir die Antwort-Substitutionen für die Anfrage ablesen :-))
- Die naive Vorgehensweise ist allerdings **hoffnungslos ineffizient** :-((
- Intelligenterere Verfahren versuchen, Mehrfachberechnungen immer der gleichen Fakten zu vermeiden ...
- Insbesondere braucht man ja auch nur solche Fakten abzuleiten, die zur Beantwortung der Anfrage **nützlich** sind \implies **Compilerbau, Datenbanken**

12.2 Operationen auf Relationen

- Wir benutzen Prädikate, um Relationen zu beschreiben.
- Auf Relationen gibt es natürliche **Operationen**, die wir gerne in **Datalog**, d.h. für Prädikate definieren möchten :-)

1. Vereinigung:



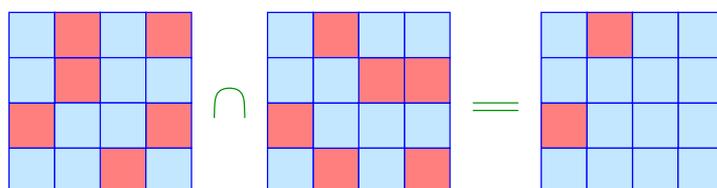
... in Datalog:

$$\begin{aligned} r(X_1, \dots, X_k) & :- s_1(X_1, \dots, X_k). \\ r(X_1, \dots, X_k) & :- s_2(X_1, \dots, X_k). \end{aligned}$$

Beispiel:

```
hört_Brauer_oder_Seidl (X) :- hat_Hörer ("Brauer", X).  
hört_Brauer_oder_Seidl (X) :- hat_Hörer ("Seidl", X).
```

2. Durchschnitt:



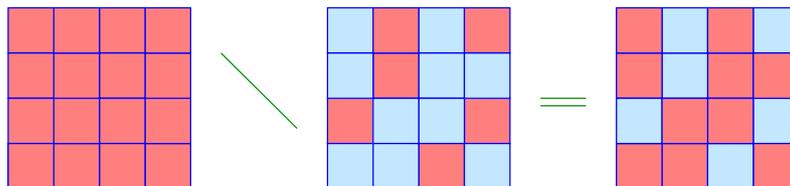
... in Datalog:

$$\begin{aligned} r(X_1, \dots, X_k) & :- s_1(X_1, \dots, X_k), \\ & \quad s_2(X_1, \dots, X_k). \end{aligned}$$

Beispiel:

```
hört_Brauer_und_Seidl (X) :- hat_Hörer ("Brauer", X),  
                             hat_Hörer ("Seidl", X).
```

3. Relatives Komplement:



... in Datalog:

$$r(X_1, \dots, X_k) \text{ :- } s_1(X_1, \dots, X_k), \text{ not}(s_2(X_1, \dots, X_k)).$$

d.h., $r(a_1, \dots, a_k)$ folgt, wenn sich $s_1(a_1, \dots, a_k)$,
aber **nicht** $s_2(a_1, \dots, a_k)$ beweisen lässt :-)

Beispiel:

```
hört_nicht_Seidl (X) :- student (X, _, _),  
                          not (hat_Hörer ("Seidl", X)).
```

Achtung:

Die Anfrage:

```
p("Hallo!").  
?- not (p(X)).
```

führt zu **unendlich vielen** Antworten :-)

⇒ wir erlauben negierte Literale nur, wenn links davon alle
Variablen in nicht-negierten Literalen vorkommen :-)

```

p("Hallo!").
q("Damn ...").
?- q(X), not (p(X)).
   X = "Damn ..."

```

Achtung (Forts.):

Negation ist nur **sinnvoll**, wenn s nicht rekursiv von r abhängt ...

```
p(X) :- not (p(X)).
```

... ist **nicht leicht** zu interpretieren.

⇒ Wir erlauben $\text{not}(s(\dots))$ nur in Regeln für

Prädikate r , von denen s nicht abhängt

⇒ **stratifizierte Negation**

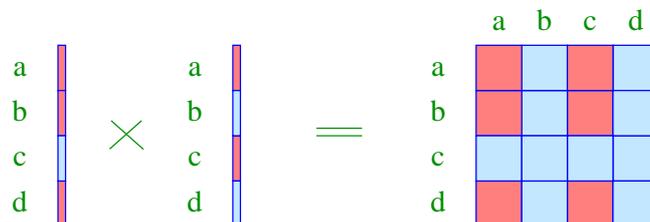
// Ohne rekursive Prädikate ist jede Negation stratifiziert :-)

4. Cartesisches Produkt:

$$S_1 \times S_2 = \{(a_1, \dots, a_k, b_1, \dots, b_m) \mid (a_1, \dots, a_k) \in S_1, (b_1, \dots, b_m) \in S_2\}$$

... in Datalog:

$$r(X_1, \dots, X_k, Y_1, \dots, Y_m) \text{ :- } s_1(X_1, \dots, X_k), s_2(Y_1, \dots, Y_m).$$



Beispiel:

```
dozent_student (X,Y) :- dozent (X,_,_),
                        student (Y,_,_).
```

Bemerkung:

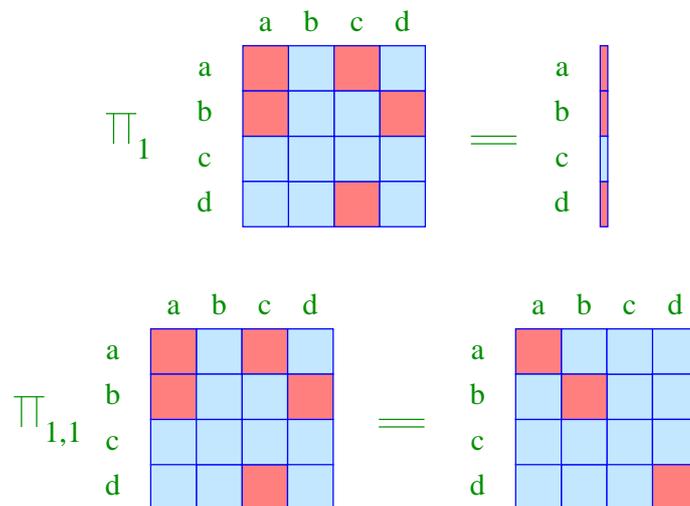
- Das Produkt unabhängiger Relationen ist sehr **teuer** :-)
- Man sollte es nach Möglichkeit **vermeiden** ;-)

5. Projektion:

$$\pi_{i_1, \dots, i_k}(S) = \{(a_{i_1}, \dots, a_{i_k}) \mid (a_1, \dots, a_m) \in S\}$$

... in Datalog:

```
r(X_{i_1}, \dots, X_{i_k}) :- s(X_1, \dots, X_m).
```



6. Join:

$$S_1 \bowtie S_2 = \{(a_1, \dots, a_k, b_1, \dots, b_m) \mid (a_1, \dots, a_{k+1}) \in S_1, \\ (b_1, \dots, b_m) \in S_2, \\ a_{k+1} = b_1 \}$$

... in Datalog:

$$r(X_1, \dots, X_k, Y_1, \dots, Y_m) \quad :- \quad s_1(X_1, \dots, X_k, Y_1), s_2(Y_1, \dots, Y_m).$$

Diskussion:

Joins können durch die anderen Operationen definiert werden ...

$$S_1 \bowtie S_2 = \pi_{1, \dots, k, k+2, \dots, k+1+m} (S_1 \times S_2 \cap \mathcal{U}^k \times \pi_{1,1}(\mathcal{U}) \times \mathcal{U}^{m-1})$$

// Zur Vereinfachung haben wir angenommen, \mathcal{U} sei das
// gemeinsame Universum aller Komponenten :-)

Joins erlauben oft, teure cartesische Produkte zu vermeiden :-)

Die vorgestellten Operationen auf Relationen bilden die Grundlage der relationalen Algebra

...

Hintergrund:

Relationale Algebra ...

+ ist die Basis für Anfragesprachen relationaler Datenbanken
 \implies SQL

+ erlaubt Optimierung von Anfragen.

Idee: Ersetze aufwändig zu berechnende Teilausdrücke der Anfrage durch billigere mit der gleichen Semantik !

- ist ziemlich kryptisch.

- erlaubt keine rekursiven Definitionen.

Beispiel:

Das **Datalog**-Prädikat:

```
semester (X,Y) :- hört (Z,X), student (Z,_,Y)
```

... lässt sich in **SQL** so ausdrücken:

```
SELECT hört.Titel, Student.Semester  
FROM   hört, Student  
WHERE  hört.Matrikelnummer = Student.Matrikelnummer
```

Ausblick:

- Außer einer Anfragesprache muss eine praktische Datenbank-Sprache auch die Möglichkeit zum **Einfügen / Modifizieren / Löschen** anbieten :-)
- Die **Implementierung** einer Datenbank muss nicht nur Spielanwendungen wie unsere Beispiele bewältigen, sondern mit **gigantischen Datenvolumen** umgehen können !!!
- Sie muss viele **parallel ablaufende Transaktionen** zuverlässig abwickeln, ohne sie durcheinander zu bringen.
- Eine Datenbank sollte auch einen Stromausfall überstehen

⇒ **Datenbank-Vorlesung**