INFORMATIK II
am D-BAUG

Kurzzusammenfassung zur Vorlesung 252-0846-00
Frühjahrssemster 2016, ETH Zürich

Felix Friedrich

1 Introduction

An overview over the planned topics of the course was presented: object oriented programming, data structures and algorithms, data bases. I pointed out that the focus of the course is on *solving problems* and not primarily on learning a programming language. This is underpinned by the discussion of *case studies* per learned concepts in the course.

We shortly recapitulated the concept of computers and programming. In particular we looked at the concept of the **Turing Machine** and demonstrated its working principle with the oldest known non-trivial algorithm, the Euclidean Algorithm. Even if we did not refer to the Turing machine too often later, it constitutes the base of nearly all what followed in the course.

We treated the programming language for this course, Java, as an imperative language first by comparing it with Pascal. We learned how basic expressions and statements are translated from Pascal to Java and already looked at the major differences: existence of classes (constituting reference typed) and methods as opposed to (value typed) records and procedures. We learned that Java does not provide reference parameters. We learned what the simplest Java program looks like ("hello world").

```
public class Hello {
    public static void main(String[] args)
    {
        System.out.println("Hello World.");
    }
}
```

Without explicit reference, we learned first principles for the implementation of an algorithm: abstract simple representation (readability and maintainability), removal of redundant execution steps by code transformation (performance), encapsulation into reusable blocks (genericity).

2 Java I

We introduced Java from scratch, starting from names and qualified identifiers, variables, (arithmetic) assignments, constants, fundamental types, numerical operations such as addition, subtraction, multiplication, division and modulus. The modulus turned out later to be important for wrap-around semantics when accessing data on an array, for example for circular buffers or hashing.

We learned that for positive numbers a and $b > 0$, the modulus $a\%b$ modulo provides the rest of a division defined by

$$a\%b = a - \lfloor a/b \rfloor \cdot b,$$

where $\lfloor a/b \rfloor$ denotes integer division such as it occurs in the Java language when two values of type `int` are divided.

We discussed rules for the exact determination of expression evaluation in terms of precedence and associativity and discussed increment and decrement operators in pre- and postfix form. We discussed type compatibilities, explicit and implicit conversion. Java is strongly typed and therefore assignments between symbols of different type require type conversions. When converting to a type with larger domain an *implicit* conversion takes place. The compiler detects static incompatibilities while the runtime can detect additional dynamic incompatibilities.

We treated control structures in Java, such as `if`-, `while`-, `do-while` and in particular `for`-loops, looked at the commonalities and differences between Pascal and Java. The most remarkable difference we identified at the `for`-loop as it provides more versatile implementations of loops.

```
for (int i = 0; i<10; ++i)
{
    System.out.println(i + "*" + i + "=" + i*i);
}
for (int x = 1; x<=128; x*= 2)
{
    System.out.println ("x= "+ x);
}
```

As an application of modulus, we looked at the principle of **Monte Carlo Simulation** and its foundation, the generation of pseudo random numbers with a linear congruence generator.

Compilation units in Java consist of *classes* that can be contained in *packages*. *Static* methods play the role of procedures in Pascal. A self-contained program provides a class containing a method of the form `public static void main(String args[])`

We used such static methods in order to implement a class for random number generation.

```
class RandomNumbers {
        static final long m = 2147483647;
        static final long b = 12345;
        static final long a = 1103515245;
        static long u = System.currentTimeMillis();
        // returns a pseudo random number in interval [0,1)
        public static double Uniform(){
                u = (u * a + b) % m;
                return(double)u/m;
        }
}
```

3 Java II

Arrays are dynamic objects in Java that can be (re-)allocated during runtime. As a consequence of reference semantics in Java, an assignment of variables of type array does not imply a data copy but only a copy of the *reference* to the data.

```
int [] x = new int[10];
int [] y;
y = x;     // copies the reference, not the array data
y[3] = 10; // modifies y and consequently also x !
```

Array bounds are checked at runtime.

Strings are objects that store a sequence of characters. If a literal is assigned to a string, the memory allocation happens implicitly. We learned that strings, just like arrays, cannot be compared element-wise by using the built-in operators <, >, ==, etc. but instead methods available on the strings have to be called, such as `x.equals(y)`.

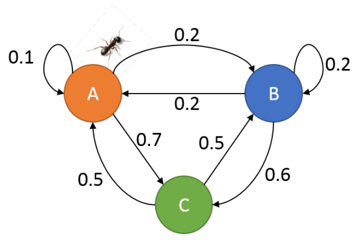
We discussed the details of pass-by-value by ways of various examples that showed that object content can change in a method, even if Java is pass-by-value.

We learned about streams such as `System.in`, `System.err` and `System.out` that can be used in order to in- or output data in Java.

We discussed random number transformation in some details in order to gain an understanding for generic random number generation from a given discrete distribution. Of central relevance is the method to draw from an unfair dice that we developed in this chapter:

```
static int UnfairDice(double[] p){
        double u = LCG.Uniform();
        double sum = p[0];
        int res= 0;
        while (res < p.length-1 && sum<u)       {
                res++;
                sum += p[res];
        }
        return res;
}
...

double[] p = {0.1,0.3,0.1,0.2,0.1,0.2};
int dice = UnfairDice(p);
```

This method was central for the implementation of the journey of the ant and its follow-up, the random surfer algorithm. A random surfer starts at an arbitrary page on the internet and continues iteratively choosing outgoing links with equal probabilities. We modeled the behavior of the random surfer as a Markov Chain with transition matrix $\mathbf{P} := (P_{ij})_{0\leq i,j\leq n}$. Entries P_{ij} stand for the probabilities to continue on page j when having started on page i. The method to draw from an unfair dice was iteratively applied taking respective rows of the transition matrix $\mathbf{P}$ in order to simulate the behavior of the random surfer. We computed the *page rank* as visiting frequency for each page.



4 Classes

Classes in Java contain data *and* code. Classes have reference semantics, i.e. they have to be allocated with `new`. Deallocation is not necessary as Java comes with a Garbage collector. Before we could actually talk about memory allocation in more detail, we visited the concept of a stack in the context of recursion. Then we discussed dynamic memory allocation with `new` in Java in order to have the minimum available to start talking about classes.

We motivated the usage of classes with the example of a pump that should be implemented with certain constraints in mind, such as a dependency of its internal state variables describing pumping head and delivery rate. We started with using classes as pure data containers.

Technically, we learned that a class can provide several constructors with different *signature*. A constructor is called when `new(...)` is executed according to its parameters, following the principle of method *overloading*.

We discussed the first principle of object-orientation: encapsulation, made possible in Java with hiding the objects state behind so called getter and setter methods. In our example, we were able to parameterize the pump such that it keeps its internal state always in-line with its characteristic curve. Only much later in the course did we revisiti this example in the context of computing pipe networks.

```
class Pumpe{
        public Pumpe (double H, double Q) { ... }
        public Pumpe (double[] H, double[] Q) {...}
        public double GetH() { ... }
        public double GetQ() { ... }
        public double SetH(double H) { ... }
        public double SetQ(double Q) { ... }
}
```

Encapsulation is an important concept which helps to give guarantees regarding invariants. By hiding implementation details behind a defined interface it permits to provide a sufficient level of abstraction.

Methods have access to the data (variables) of their containing class via the implicit `this` parameter. If `this` is not specified in a reference, it is implicitly added provided that the symbol refers to the class variables.

Passing a variable of class type as parameter means passing a *reference* to an object.

Case Study: Online Statistics

Wanted: object providing values such as mean or variance without costly computation.

The *circular buffer* is a concept to store a limited amount of data. Using this concept mean and variance can be computed in linear time. Central to the circular buffer was the reuse of data in an array by maintaining a pointer with so called wrap-around semantics.

```
public void Put(double value){
        data[pos] = value;
        if (n < data.length)
                n++;
        pos = (pos + 1) % data.length; // wrap around semantics
}
```

Using the concept of a *dynamic (growing) array*, the amount of stored data can be derestricted.

For online computation of mean and variance, the provisional means algorithm turned out to be much better. It updates the mean according to the formula $\mu_{n+1} = \mu_n + \frac{x_{n+1} - \mu_n}{n+1}$. The median is a selection problem that requires a more complicated treatment.

The Statistics class was a good example of how *getters* and *setters* are used in Object Oriented Programming in order to hide implementation details.

```
public class Statistics {
    int n = 0;    double mean = 0;    double ssq = 0;
    public void Put(double value){
        n++; double oldMean = mean;
        mean = oldMean + (value - oldMean) / n;
        ssq = ssq + (ssq - oldMean) * (value - mean);
    }
    public double Mean(){ return mean; }
    public double Variance() { if (n==0) return 0; return ssq}
}
```

Hashing

Starting from a data set storing name and telephone number of a friend we discussed how to provide an object that is capable of storing and quickly retreiving data sets by some key (such as the name). We started this discussion from the assumption that we have a computer with unlimited storage capacity and that we can deal with integers of arbitrary size (a so called “Random Access Machine”). We considered different strategies to map a name to a storage location in a conceptually unlimited array.

We found a function that indeed maps different names of the form $s_0 s_1 ... s_{l-1}$ to different integers (provided that constant $b > 0$ is large enough):

$$h_b(s) = \sum_{i=0}^{l-1} s_i \cdot b^i = s_0 + s_1 \cdot b + s_2 \cdot b + \cdots + s_{l-1} \cdot b^{b-1}$$

We found out that constraining the outcome of this function to a finite index domain introduces the problem of collisions, i.e. the fact that different data sets might be mapped to the same index location.

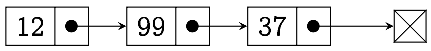
We gave the definition of hash table and hash function as follows: A *hash function* is a mapping from the set of possible key values to the set of possible indices in an array. A *hash table* is a data structure where references to the data are stored in an array together with a hash function that delivers array indices from key values.

Going from there we developed an algorithm to store data sets in a table by carefully treating collisions. Collisions occur for a hash function h when $h(k_1) = h(k_2)$ for two distinct used key values k_1, k_2. We shortly discussed so called *open addressing* with linear probing: if entry i is occupied with a data set of differing key value the next possible entry was chosen in the array adopting wrap-around semantics.

```
class Friends{
        String[] key;  // Schl ssel
        Entry[] data;  // Daten
        ...

        int Probe(String k){
                int index = Hash(k);
                while(key[index] != null && !key[index].equals(k))
                        index = (index + 1) % key.length;
                return index;
        }
}
```

5 Dynamic Data Structures I

We motivated dynamic data structures with the observation that for arrays it is computationally expensive to insert or remove elements ”in the middle”. *Linked lists* provide a solution where elements are stored anywhere in memory (and not consecutively as arrays do it). In addition to the *key* (or value), each list element also stores a pointer to its successor:



For the implementation of a singly linked list, we introduced a dynamic data structure

```
class Node
{
    double value;
    Node next;
    Node (double v, Node nxt){
        value=v; next = nxt;
    }
}
```

and used it in order to implement *stack* and *queue*. We learned that *insert and remove* operations require a careful treatment of *special cases* such as "empty list". Moreover, we learned how to use a *"running pointer"* in order to traverse a list. Sorted insertion requires keeping a reference to the previous element when searching for the insertion position.

Stepwise Refinement

With stepwise refinement we discussed a top-down approach to solving programming problems by formulating rough solutions first with comments and fictitious functions. The refinement step comprises of replacing comments by program text and fictitious functions by real function definitions. As an example we discussed insertion into a sorted list.

```
public void Insert (int value)
{
        if (first == null || value <= first.value)
                first = new Node(value, first);
        else {
                Node prev = first;
                Node v = prev.next;
                while (v != null && v.value > value){
                        prev = v;
                        v = v.next;
                }
                prev.next = new Node(value, v);
        }
}
```

Case Study: Point-In-Polygon Algorithm

The Jordan curve theorem implies: in order to identify if a point p is in the inner of a polygon, it is sufficient to count intersections of the polygon with an arbitrary half-line l starting in p. The main difficulty with implementations arise at the special cases where l cuts the polygon on its vertices.

We implemented a polygon as a singly linked list of vertices. The PointInPolygon algorithm was implemented firstly such that it operated on horizontal lines taking into account the mentioned special cases. It used floating point arithmetics. An important tool used during the implementation of complicated algorithms such as the PointInPolygon algorithm was the consideration of **invariants** that hold for a polygon.

Motivated by potential ambiguities, lacking computing efficiency and by the requirement to draw the polygon, we discussed line drawing on a pixel grid as an interesting *discretization* task. We treated a slightly simpler naive line drawing algorithm to quite some detail in order to understand the process of discretization. The *Bresenham algorithm* is a highly efficient algorithm to draw an arbitrary discretized line on a pixel grid. It works without floating point arithmetic. The Bresenham algorithm keeps the absolute value of $err = dx(d - y_0) - dy(x - x_0)$ small when advancing in horizontal or vertical direction.

6 Object Oriented Programming

Motivation: build a graphics library for drawing geometric figures. A solution with the procedural approach was sketched and problems were identified: waste of memory resources, administrative overhead, hindered "non-invasive" code extension.

One key to the solution of such problems is the concept of *inheritance*: common properties of figures can stay in a base class ("generalisation") and distinguishing properties can be expressed in the inheriting classes ("specialisation"). Inheritance implies possibility of code reuse operating on common properties. We introduced the notions *base class*, *inheriting class*.

```
public class Figure {
  ...
  public void draw(BufferenImage img) { }
}

public class Circle extends Figure {
  ...
  public void draw(BufferedImage img) {...} // draw circle
}

public class Rectangle extends Figure {
  ...
  public void draw(BufferedImage img) {  ... } // draw rect
}
```

In order to take the dynamic type of an object into account, the concept of *polymorphism* was introduced. When a method with same type and signature as in the base class is defined in the inheriting class then the method to be executed is chosen *at* *runtime*. Object Orientation requires encapsulation, inheritance and polymorphism.

Case Study: Numerical Integration

Objective was the development of a generic software framework for numerical integration of an arbitrary real valued function. We introduced abstract classes

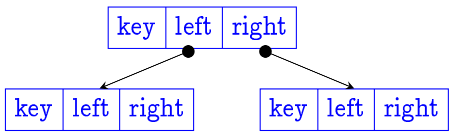
```
public abstract class Function {
    public abstract double Evaluate(double x);
}

public abstract class Integrator {
    int n;
    public void SetNumberPieces(int pieces) {
        n = pieces;
    }
    public abstract double Integrate(Function f, double x0, double x1);
}
```

and experimented with different functions and integrators by specializing. In the course we considered the parabola $f(x) = x^2$ and the density of the normal distribution and applied *Rectangle rule*, *Trapezoidal rule*, *Simpson rule* and a *Monte Carlo* integrator. We derived Simpsons's rule and experimentally verified the error terms of $O(\Delta^3)$ for Rectangle/Trapezoidal rule and and $O(\Delta^5)$ for Simpson Rule.

7 Dynamic Data Structures II

Trees are a natural generalization of lists: nodes have more than one successor. A *binary search tree* is a tree of order two with $\text{key}_{x.\text{left}} < \text{key}_x < \text{key}_{x.\text{right}}$ for each node x

```
public class SearchNode {
    int key;
    SearchNode left;
    SearchNode right;
    ...
}
```



Insertion in a binary search tree requires *tree traversal* according to the key order until an empty child node is found. For node removal several cases have to be considered. If the node has two non-empty children then it has to be replaced by a *symmetric successor*, e.g. the leftmost node in its right subtree.

Search trees can *degenerate* to linked lists which implies a worst case complexity $O(n)$. To guarantee $O(\log n)$ in worst case, update operations require additional balancing, which we did not treat in this course.

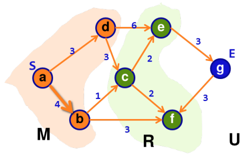
Heaps

A *Min-Heap* is a binary tree with the Min-Heap property: the key of a child node is always greater than the key of the parent node. A heap is thus a data structure for fast retreival of the minimum of a data set. The heap data structure can be easily stored in an array with indices $\mathrm{parent}(i) = \lfloor (i-1)/2 \rfloor$ and $\mathrm{children}(i) = \{2i+1, 2i+2\}$. In order to insert an element in a heap, the element is inserted at the first free position and the heap property is ensured by "raising" the element to its proper position. In order to retreive and delete the minimum element, the root is replaced by the last node and it is "lowered" until the heap property is reasserted.

The median of a data set can be kept up to date with a worst case update complexity $O(\log n)$ by employing a min- and a max-heap around the median.

Case Study: Dijkstra's Shortest Path

Given a directed graph provided with positive weights at the edges, objective is finding a path with lowest accumulated weights leading from starting point S to end point E.



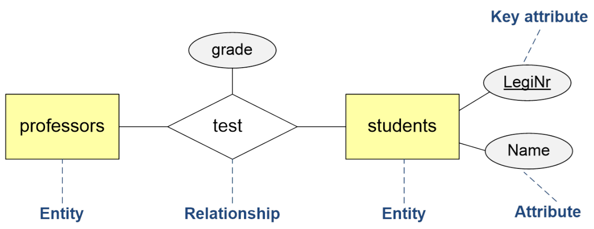
In order to formulate the iterative algorithm of Dijkstra, we considered three sets of nodes. M: nodes that are part of a shortest path, initially $M = \{S\}$. R: all nodes not in M that can be reached via one edge from M and U: all other remaining nodes.

At each update step a node n from R is chosen with minimal path length amongst all nodes in R. Node n is added to M. Then the neighbours of n are added to R and all path lengths of elements in R are updated. In the implementation we used a Min-Heap to store R in order to quickly identify the minimum. During the update step of path lengths of elements in R it was required to find elements in the middle of the heap, which is why a hash table had to be maintained for the stored elements in a heap.

8 Databases: Entity-Relationship Model

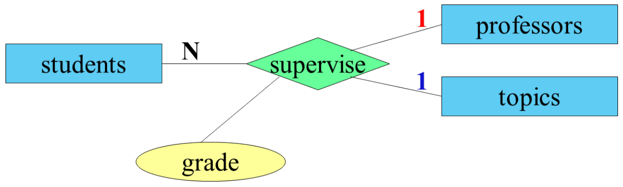
The Entity-Relationship Model provides a means to conceptually model a part of the world for a database in a graphical way.

It consists of *entities* and *relationships* which both can be characterized with *attributes*. Entities can provide a *key attribute* that models a unique identifier of an identity. Relationships can play *roles* with respect to entities.



An additional tool in modeling relationships is the specification of how many items may stand in relation to how many other items. Such *functionalities* are provided in the form "1:1", "1:N", "N:M" for binary relations, and "1:N:M:..." for n-ary relations. A functionality containing a "1" somewhere provides the information that the relationship can be written as a partial function with codomain of the entity type at the "1".

If, for eample, entities e_1, e_2 and e_3 are related as "1:N:1", then their relation is a subset $R \subset E$ of the cartesian product of their domains $E = E_1 \times E_2 \times E_3$ and can be understood as both partial function $f_R : E_2 \times E_3 \nrightarrow E_1$ and as partial function $g_R : E_1 \times E_2 \nrightarrow E_3$. This often helps with the interpretation of relationships.



supervise: professors $\times$ students $\nrightarrow$ topics
supervise: topics $\times$ students $\nrightarrow$ professors

9 Databases: Relational Model

Using the relational model a database is described as a collection of *relations* (tables) $R \subset D_1 \times \cdots \times D_n$ over domains D_i of *attributes*. A *tuple* $t \in R$ constitutes an element of a relation, i.e. a row in a table. *Schemas* of such tables are described in the form

$$\{[A_1, \ldots, A_n]\}$$

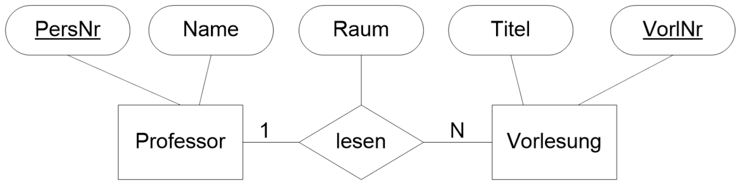
where attributes A_i are usually described in a name:domain form. A *key* is a minimal set of attributes that uniquely identifies a tuple in a relation.

When translating from an ER-Model to a relational model, entity attributes translate to relation attributes. Key attribute translate to *primary keys* of relations. When translating from a relationship, the attributes of all entities together with the attribute of the relationship form the attributes of the relation

$$\{[A_{11}, \ldots, A_{1n_1}, A_{21}, \ldots, A_{2n_2}, \ldots, A_{m1}, \ldots, A_{mn_m}, A_1^R, \ldots, A_{n_r}^R]\}$$

The key of the relation is then formed by all keys of all attributes plus the keys of the relationship. A renaming of attributes may be necessary.

After entities and relationships have been translated into relations, (only) relations with the same key can be *merged*. This implies that relations from $N : M$ relationships cannot be merged.



$$\text{Professor }\{[\underline{\text{PersNr}}, \text{Name}]\}, \text{Vorlesung }\{[\text{Titel}, \underline{\text{VorlNr}}]\}, \text{lesen }\{[\text{PersNr}, \underline{\text{VorlNr}}, \text{Raum}]\}$$
$$\Rightarrow \text{Professor }\{[\underline{\text{PersNr}}, \text{Name}]\}, \text{Vorlesung }\{[\text{Titel}, \underline{\text{VorlNr}}], \text{gelesenVon}, \text{Raum}\},$$

We discussed the following operations from relational algebra:

1. *Selection* operation $\sigma_p(R)$ selects tuples (rows) from a relation (table) R that fulfil the selection predicate p.
2. *Projection* operation $\pi_a(R)$ projects to the named attributes (columns) a of a relation (table) R
3. *Cartesian product* operation $R_1 \times R_2$ yields all possible pairs $r_1 r_2$ of tuples of R_1 and R_2
4. *Renaming* operation $\rho_S(R)$ assigns a new name S to a relation R, renaming $\rho_{A_1 \to B_1}(R)$ renames an attribute (column) of a relation (table).
5. The *Join* operation $R \bowtie S$ selects tuples from $R \times S$ that have equal values on all attributes with the same names and merges attributes with same names.
6. The *Theta-Join* operation $R \bowtie_\theta S$ coincides with $\sigma_\theta(R \times S)$.

Although the relational algebra is not used per se in practice, it provides a very important understanding for databases: queries operate on sets and therefore have to be perceived as set-valued operations.

10 Databases: SQL

SQL (Structured Query Language) is a language used in order to define, manipulate and formulate queries on data bases. In terms of querying it provides a mapping of relational algebra to a formalized natural language.

A typical simple *query* in SQL looks like

```
select s.Name, v.Titel
from Studenten s, hoeren h, Vorlesungen v
where s. Legi= h.Legi and h.VorlNr = v.VorlNr
```

A select statement in SQL corresponds to projection, the from part specifies (cartesian product of) participating tables and the where clause provides a selection predicate.

Aggregate functions can be used in order to compute aggregate values over columns of a table, returning a single value per column. If aggregation is used together with *grouping*, it returns a tuple for each group.

```
select v.gelesenVon, p.Name, sum(v.KP)
from Vorlesungen v, Professoren p
where v.gelesenVon = p.PersNr and p.Rang = 'FP'
group by v.gelesenVon, p.Name
    having avg(v.KP) >= 3
```

For nested statements and grouping, it is important to understand possible execution orders. The previous query is executed as

1. from Vorlesungen, Professoren where gelesenVon = PersNr and Rang = FP
2. group by gelesenVon, Name
3. having avg (KP) >= 3
4. select gelesenVon, Name, sum (KP)

It is possible to use queries in a *nested* way, i.e. the result of a query can serve as table within a query.

```
select c.name from
(
select l.countrycode, l.language, count(l.language)
from countrylanguage l
where l.isofficial = TRUE
group by l.countrycode
having count(l.language)=1
) as p, country c
where p.language = "German" and c.code = p.countrycode
```

SQL defines various *data types* for typical database entries such as characters, numbers, dates, raw data. As Data Definition Language (DDL), SQL provides statements such as `create table`, `drop table`, `alter table`. As Data Manipulation Language(DML), SQL provides statements such as `insert into`, `delete` and `update`.