

Programming languages and compilers

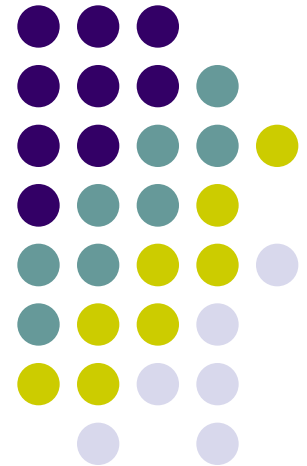
Programming languages

Ing. Marek Běhálek
FEI VŠB-TUO

A-1018 / 597 324 251

<http://www.cs.vsb.cz/behalek>
marek.behalek@vsb.cz

This presentation is based on original course materials coming from doc. Ing Miroslav Beneš
Ph.D.





Overview

- Introduction
- History
- Classification of programming languages
- Specification of programming languages
- Declarative programming
- Functional programming - Haskell
- Logical programming languages
- Script languages
- „Non traditional“ object oriented languages

Introduction - What is a programming language?



- Many definitions
 - A programming language is a **machine-readable** artificial language designed to express **computations** that can be performed by a machine, particularly a **computer**.
 - Programming languages can be used to create programs that specify the behavior of a machine, to express algorithms **precisely**, or as a mode of human communication.
- Wikipedia – Programming languages

Introduction - Definitions



- **Function** – a language used to write computer programs, which involve a computer performing some kind of computation or algorithm.
- **Target** - Programming languages differ from natural languages, they are build to allow humans to communicate instructions to machines.
 - Some programming languages are used by one device to control another.
- **Constructs** - Programming languages may contain constructs for defining and manipulating data structures or controlling the flow of execution.
- **Expressive power** - The theory of computation classifies languages by the computations they are capable of expressing.
 - All Turing complete languages can implement the same set of algorithms.
 - ANSI/ISO SQL and Charity are examples of languages that are not Turing complete, yet often called programming languages.
- Sometime is term "programming language" restricted to those languages that can express all possible algorithms.
 - Sometimes the term "computer language" is used for more limited artificial languages.
- Non-computational languages, such as markup languages like HTML or formal grammars like BNF, are usually not considered programming languages.



History – First Languages

Theoretical beginnings - 30s

- **Alonzo Church** - lambda calculus – theory of computations
- **Alan Turing** – show that a machine can solve a “problem”.
- **John von Neumann** – defined computer’s architecture (relevant even for today’s computers).

Around 1946 Konrad Zuse – Plankalkul

- Used also for a chess game
- Not published until 1972, never implemented

1949 John Mauchly - Short Code

- First language actually used on an electronic device.
- Used for equations definition.
- “hand compiled” language.

1951 Grace Murray Hopper

- Enforcement of usage of high level programming languages.
- Work on a design of first compiler.

History – First Compilers



Term “compiler”

- early 50s - Grace Murray Hopper
- Program’s compilation like a “compilation” of sequences of programs form a library.
- “automatic programming” – compilation in today’s meaning assumed to be impossible to perform.

1954-57 FORTRAN (FORMula TRANslator)

- John Backus, IBM
- Problem’s oriented, machine independent language
- Fortran shows advantages of high level compiled programming languages.
- Ad hoc structures – components and technologies were work out during development
- That day’s people believes compilers are to complex, hard to understand and very expensive. (**18 humans years** –one of the greatest projects of that times)

History – FORTRAN



```
C
C      Function computing a factorial
C
      INTEGER FUNCTION FACT(N)
      IMPLICIT NONE
      INTEGER N, I, F
      F = 1
      DO 10 I = 1,N
         F = F * I
10     CONTINUE
      FACT = F
      END

      PROGRAM P1
      IMPLICIT NONE
      INTEGER N, F, FACT
      READ(*,*) N
      F = FACT(N)
      WRITE(*,*) "Fact = ", F
      END
```

History – High level programming languages(1)



1958-59 LISP 1.5 (List Processing)

- John McCarthy, M. I. T.
- First functional programming language – implementation of lambda calculus
- Also possibility of usage of a imperative style of programming

1958-60 ALGOL 60 (Algorithmic Language)

- J. Backus, P. Naur
- Blok structure, composed statements, recursion.
- Syntax formally described by a grammar (BNF) for the first time.
- Most popular language in Europe in late 60s.
- Base for other programming languages.

History – ALGOL 60



```
begin
  integer N;
  ReadInt(N);

  begin
    real array Data[1:N];
    real sum, avg;
    integer i;
    sum:=0;
    for i:=1 step 1 until N do
      begin real val;
        ReadReal(val);
        Data[i]:=if val<0 then -val else val
      end;
    for i:=1 step 1 until N do
      sum:=sum + Data[i];
    avg:=sum/N;
    PrintReal(avg)
  end
end
```

History – High level programming languages(2)



1960 COBOL (Common Business Oriented Language)

- COBOL is one of the oldest programming languages still in active use.
- Its primary domain in business, finance, and administrative systems for companies and governments.
- COBOL 2002 standard includes support for object-oriented programming and other modern language features.

1964 BASIC (Beginners All-Purpose Symbolic Instruction Code)

- John G. Kemeny, Thomas E. Kurz, Dartmouth University
- 1975 Tiny BASIC running on a computer with 2KB RAM
- 1975 Bill Gates, Paul Allen sells it to a company MITS

1963-64 PL/I (Programming Language I)

- Combination of languages: COBOL, FORTRAN, ALGOL 60
- Developed to contain “everything for everybody” => too complex
- Present constructions for concurrent execution and exceptions.

History – COBOL



```
IDENTIFICATION DIVISION.  
PROGRAM-ID. Iter.  
AUTHOR. Michael Coughlan.  
  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
    01 Num1 PIC 9 VALUE ZEROS.  
    01 Num2 PIC 9 VALUE ZEROS.  
    01 Result PIC 99 VALUE ZEROS.  
    01 Operator PIC X VALUE SPACE.  
  
PROCEDURE DIVISION. Calculator.  
    PERFORM 3 TIMES  
        DISPLAY "Enter First Number      : "  
        ACCEPT Num1  
        DISPLAY "Enter Second Number    : "  
        ACCEPT Num2  
        DISPLAY "Enter operator (+ or *) : "  
        ACCEPT Operator  
        IF Operator = "+" THEN  
            ADD Num1, Num2 GIVING Result  
        END-IF  
        IF Operator = "*" THEN  
            MULTIPLY Num1 BY Num2 GIVING Result  
        END-IF  
        DISPLAY "Result is = ", Result  
    END-PERFORM.  
STOP RUN.
```

History – PL/I



```
FINDSTRINGS: PROCEDURE OPTIONS(MAIN)
  /* načte STRING a poté vytiskne každý
     následující shodující se řádek */
  DECLARE PAT VARYING CHARACTER(100),
           LINEBUF VARYING CHARACTER(100),
           (LINENO, NDFILE, IX) FIXED BINARY;
  NDFILE = 0; ON ENDFILE(SYSIN) NDFILE=1;
  GET EDIT(PAT) (A);
  LINENO = 1;
  DO WHILE (NDFILE=0);
    GET EDIT(LINEBUF) (A);
    IF LENGTH(LINEBUF) > 0 THEN DO;
      IX = INDEX(LINEBUF, PAT);
      IF IX > 0 THEN DO;
        PUT SKIP EDIT (LINENO,LINEBUF) (F(2),A)
        END;
      END;
      LINENO = LINENO + 1;
    END;
  END FINDSTRINGS;
```

History – High level programming languages(3)



1968 ALGOL 68

- Widely used version of ALGOL 60
- A little bit too complex to understand and to implement
- Structured data types, pointers
- Formal syntax and semantics definition
- Dynamic memory management, garbage collection, modules

1966 LOGO

- Logo is a computer programming language used for functional programming.
- Today, it is known mainly for its turtle graphics
- Development goal was to create a math land where kids could play with words and sentences.

History – Structured programming languages



1968-71 Pascal

- Niklaus Wirth, ETH Zurich
- Developed to be a small and efficient language intended to encourage good programming practices using structured programming and data structuring.

1972 C

- Dennis Ritchie
- C was designed for writing architecturally independent system software.
- It is also widely used for developing application software.

History – Pascal



```
program P3;

var
  F: Text;
  LineNo: Integer;
  Line: array [1..60] of Char;

begin
  if ParamCount < 1 then begin
    WriteLn('Pouziti: opis <inp>');
    Halt;
  end;

  Reset(F, ParamStr(1));
  LineNo := 1;
  while not Eof(F) do begin
    ReadLn(F, Line);
    WriteLn(LineNo:4, ': ', Line);
    LineNo := LineNo + 1;
  end;
end.
```

History – Modular programming



1980 Modula-2

- Support of modularity, strong type control, dynamic arrays, co programs

1980-83 Ada

- Jean Ichibah, Honeywell Bull for US DoD
- Ada was originally targeted at embedded and real-time systems.
- Ada is strongly typed and compilers are validated for reliability in mission-critical applications, such as avionics software.
- Properties: strong typing, modularity mechanisms (packages), run-time checking, parallel processing (tasks), exception handling, and generics, dynamic memory management

History – Modula-2



```
DEFINITION MODULE Storage;

VAR
  ClearOnAllocate : BOOLEAN;

PROCEDURE Allocate( VAR a: ADDRESS; size: CARDINAL );
PROCEDURE Free( VAR a: ADDRESS );
PROCEDURE Deallocate( VAR a: ADDRESS; size: CARDINAL );
PROCEDURE Reallocate( VAR a: ADDRESS; size: CARDINAL );

PROCEDURE MemorySize( a : ADDRESS ): CARDINAL;
TYPE
  TMemoryStatus = RECORD
    MemoryLoad      : LONGCARD;  (* percent of memory in use *)
    TotalPhys       : LONGCARD;  (* bytes of physical memory *)
  END;

PROCEDURE GetMemoryStatus( VAR MemoryStatus : TMemoryStatus );

END Storage.
```

History – Ada



```
with TEXT_IO; use TEXT_IO;

procedure faktorial is
  package IIO is new INTEGER_IO(Integer);
  use IIO;

  cislo: Integer;

  function f(n : Integer) return Integer is
  begin
    if n < 2 then
      return 1;
    else
      return n*f(n-1);
    end if;
  end f;

begin
  PUT("Zadejte cislo:");
  GET(cislo);
  PUT(f(cislo));
  SKIP_LINE;
end faktorial;
```

History – Object oriented languages(1)



1964-67 SIMULA 67

- Ole Dahl, Kristen Nygaard (Norsko)
- For simulation of discrete models
- Abstract data types, classes, simple inheritance – base for object oriented languages

1972 Smalltalk

- Alan Kay, Xerox
- Originally only experimental language.
- Pure object oriented language – everything is achieved with message transition.
- First language supporting GUI with windows.
- Interpreted at the beginning. Now translated into abstract machine code or Just-in-time compiled.

1982-85 C++

- Bjarne Stroustrup, AT&T Bell Labs
- Developed from C => many dangerous futures like dynamic memory management without GC, pointer arithmetic
- 1997 ISO a ANSI standard

History – Object oriented languages(2)



1984-85 Objective C

- Brad J. Cox
- C language extension, for OOP defined new constructions
- Widely considered to be better than C++, freely available compilers come to late...
- Main programming language for Apple NeXT and OS Rhapsody

1994-95 Java

- James Gosling, Sun Microsystems
- Originally developed for embedded devices, later widely used for other areas like WWW.
- Machine independent code (Java Bytecode), use just-in-time compilation

2000-02 C#

- Anders Hejlsberg, Microsoft
- One of the basics languages of .NET
- Implemented even for Linux (project Mono) a BSD Unix (project Rotor)

History – C#



```
using System;
using System.Windows.Forms;
using System.Drawing;
public class Sample : Form {
    [STAThread]
    public static int Main(string[] args) {
        Application.Run(new Sample());
        return 0;
    }
    public Sample() {
        Button btn = new Button();
        btn.Text = "  ";
        Controls.Add(btn);
    }
}
```

Language Classification- Introduction



- Many different criteria for a classification of programming languages.
 - Implemented paradigm of programming.
 - Object oriented paradigm
 - Declarative style of programming
 - Aspect oriented programming
 - ...
 - Implemented type system
 - Weak vs. Strong Typing
 - Dynamic vs. Static Types
 - ...
 - Generation (“level”) of programming language
 - High vs. low level programming languages
 - Machine dependent programming languages
 - ...

Language Classification- Paradigm of programming (1)



- A programming paradigm is a fundamental style of computer programming.
 - Compare with a methodology, which is a style of solving specific software engineering problems.
 - Paradigms differ in the concepts and abstractions used to represent the elements of a program.
 - objects, functions, variables, constraints, etc.
 - steps that compose a computation (assignment, evaluation, continuations, data flows, etc.).
 - Example : In object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations.

Language Classification-

Paradigm of programming(2)



- A programming language can support multiple paradigms.
 - Smalltalk supports object-oriented programming.
 - Java supports imperative, generic, reflective, object-oriented (class-based) programming.
- Many programming paradigms are as well known for what techniques they forbid as for what they enable.
 - For instance, pure functional programming disallows the use of side-effects.
 - Structured programming disallows the use of the *goto* statement.

Language Classification- Examples of Programming paradigms (1)



- Annotative programming (as in Flare language)
- Aspect-oriented programming (as in AspectJ)
- Attribute-oriented programming (might be the same as annotative programming) (as in Java 5 Annotations, pre-processed by the XDoclet class; C# Attributes)
- Class-based programming, compared to Prototype-based programming (within the context of object-oriented programming]
- Concept-oriented programming is based on using concepts as the main programming construct.
- Constraint programming, compared to Logic programming
- Data-directed programming
- Dataflow programming (as in Spreadsheets)
- Flow-driven programming, compared to Event-driven programming
- Functional programming
- Imperative programming, compared to Declarative programming
- Intentional Programming
- Logic programming (as in Mathematica)

Language Classification- Examples of Programming paradigms (2)



- Message passing programming, compared to Imperative programming
- Object-Oriented Programming (as in Smalltalk)
- Pipeline Programming (as in the UNIX command line)
- Policy-based programming
- Procedural programming, compared to Functional programming
- Process oriented programming a parallel programming model.
- Recursive programming, compared to Iterative programming
- Reflective programming
- Scalar programming, compared to Array programming
- Component-oriented programming (as in OLE)
- Structured programming, compared to Unstructured programming
- Subject-oriented programming
- Tree programming
- Value-level programming, compared to Function-level programming

Language Classification- Basic programming paradigms (1)



- **Imperative**

- Programs are sequences of statement (mostly assignments).
- Programs flow can be changed using control statements like loops.
 - Control statement define which statement will be performed and in what order.
- *C, Pascal, Fortran, JSI*

- **Object oriented**

- Program are collections of interacting objects.
- Often uses inheritance or polymorphism.
- *Simula, Smalltalk-80, C++, Java, C#*

Language Classification – Language and computer's architecture



- Programming languages are limited by an architecture of today's computer.
 - Effective implementation must exist if we want to use them to create real life applications.
- Von Neumann's architecture
 - Model of today's mainstream computers
 - Widely used languages like Java or C/C++/C# are closely related to this architecture.
- Functional languages
 - Backus (1977, Turing Award) Can Programming Be Liberated From the von Neumann Style?
 - Criticized attempt „from architecture to language“
 - For example functional languages are considered to be superior to imperative languages.
 - We can prove some properties.
 - Easy to parallelize
 - Based on algebraic rules
 - On the other hand they are not as effective as imperative languages on Von Neumann's architecture based computers.
 - Massive optimizations needed (Ocaml - nearly as effective as C)
 - Result => Not so often used like for example Java.

Language Classification- Basic programming paradigms (2)



Declarative languages

– source code describes what to compute not how

- **Logic programming languages**

- Programs are a collection of predicates in some concrete logic (most often predicate logic).
- Defining feature of logic programming is that sets of formulas can be regarded as programs and proof search can be given a computational meaning.
- *Prolog, Goedel*

- **Functional programming languages**

- Treats computation as the evaluation of mathematical functions and avoids state and mutable data.
- It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state.
- *FP, LISP, Scheme, ML, Haskell*

Language Classification- Basic programming paradigms (3)



- **Concurrent programming languages**
 - Programs are designed as collections of interacting computational processes that may be executed in parallel.
 - Concurrent (parallel) programming languages are programming languages that use language constructs for concurrency.
 - Some versions of language Modula-2, Ada
 - Today's programming languages often use some sort of library for concurrent programming MPI, PVM.

Language Classification- Type system



- Type system definition
 - Strict:
 - A tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.
 - Loosely:
 - A type system associates one (or more) type(s) with each program value.
 - By examining the flow of these values, a type system attempts to prove that no "type errors" can occur.
- Type system's main functions
 - Assigning data types (typing) gives meaning to collections of bits.
 - Types usually have associations either with values in memory or with objects such as variable.
 - Safety - Use of types may allow a compiler to detect meaningless or probably invalid code.
 - Abstraction (or modularity) - Types allow programmers to think about programs at a higher level than the bit or byte, not bothering with low-level implementation.
 - Optimizations, documentation,...
- Type theory studies type systems.

Language Classification- Type checking



- The process of verifying and enforcing the constraints of types – *type checking* .
- Different ways to categorize the type checking.
 - The terms are not used in a strict sense!
 - Compile-time (a static check) / Run-time (a dynamic check)
 - Strongly typed / Weakly typed
 - Safely and unsafely typed systems

Language Classification-

Categorizing type checking (1)



- Static typing
 - Type checking is performed during compile-time as opposed to run-time.
 - Ada, C, C++, C#, Java, Fortran, ML, Pascal, or Haskell.
 - Static typing is a limited form of program verification
 - However it allows many errors to be caught early in the development cycle.
 - Program execution may also be made more efficient (i.e. faster or taking reduced memory).
 - Static type checkers are conservative.
 - They will reject some programs that may be well-behaved at run-time, but that cannot be statically determined to be well-typed.
 - Some statically typed languages enables programmers to write pieces of code that circumvent the default verification performed by a static type checker.
 - For example, Java and most C-style languages have type conversion.

Language Classification-

Categorizing type checking (2)



- Dynamic typing
 - Majority of its type checking is performed at run-time.
 - Groovy, JavaScript, Lisp, Clojure, Objective-C, Perl, PHP, Prolog, Python, Ruby, or Smalltalk.
 - Dynamic typing can be more flexible than static typing.
 - For example by allowing programs to generate types based on run-time data.
 - Run-time checks can potentially be more sophisticated, since they can use dynamic information as well as any information that was present during compilation.
 - On the other hand, runtime checks only assert that conditions hold in a particular execution of the program, and are repeated for every execution of the program.

Language Classification-

Categorizing type checking (3)



- Strongly typed languages (also term memory safe is used)
 - Definition involves preventing success for an operation on arguments which have the wrong type.
 - Strongly typed languages that do not allow undefined operations to occur.
 - For example, a memory-safe language will check array bounds (resulting to compile-time and perhaps runtime errors).
- Weak typing means that a language implicitly converts (or casts) types when used.
- Example

```
var x := 5; // (1) (x is an integer)
var y := "37"; // (2) (y is a string)
x + y; // (3) (?)
```

- It is not clear what result one would get in a weakly typed language.
 - Visual Basic, would produce run able code producing the result 42.
 - JavaScript would produce the result "537".

Language Classification-

Categorizing type checking (4)



- “Type-safe“ is language if it does not allow operations or conversions which lead to erroneous conditions.
 - Let us again have a look at the pseudocode example:

```
var x := 5;           // (1)
var y := "37";       // (2)
var z := x + y;      // (3)
```

 - In languages like Visual Basic variable z in the example acquires the value 42.
 - The programmer may or may not have intended this, the language defines the result specifically, and the program does not crash or assign an ill-defined value to z.
 - If the value of y was a string that could not be converted to a number (eg "hello world"), the results would be undefined.
 - Such languages are type-safe (in that they will not crash) but can easily produce undesirable results.
 - Now let us look at the same example in C:

```
int x = 5;
char y[] = "37";
char* z = x + y;
```

 - In this example z will point to a memory address five characters beyond y.
 - Might lie outside addressable memory.
 - The mere computation of such a pointer may result in undefined behavior.
 - We have a well-typed, but not memory-safe program.
 - A condition that cannot occur in a type-safe language.

Language Classification- Other Type System's Futures



- Polymorphism
 - The ability of code (in particular, methods or classes) to act on values of multiple types.
 - Or the ability of different instances of the same data-structure to contain elements of different types.
 - Type systems that allow polymorphism generally do so in order to improve the potential for code re-use.
 - In a language with polymorphism, programmers need only implement a data structure such as a list or an associative array once.

Language Classification- Level of programming language (1)



- Low-level programming languages (machine dependent programming languages).
 - language that provides little or no abstraction from a computer's instruction set architecture.
 - The first-generation programming language, or 1GL, is machine code.
 - It is the only language a microprocessor can understand directly.
 - Example: A function in 32-bit x86 machine code to calculate the nth Fibonacci number:

```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008D0419 83FA0376 078BD98B
C84AEBF1 5BC3
```

Language Classification- Level of programming language (2)



- The second-generation programming language, or 2GL, is assembly language.
 - It is considered a second-generation language because while it is not a microprocessor's native language, an assembly language programmer must still understand the microprocessor's unique architecture (such as its registers and instructions).
 - These simple instructions are then assembled directly into machine code.
- Part of program computing Fibonacci numbers above, but in x86 assembly language using MASM syntax:

```
mov edx, [esp+8]
cmp edx, 0
ja @f
mov eax, 0
ret
```

Language Classification- Level of programming language (3)



- High level programming languages
 - Such languages hide the details of CPU operations such as memory access models and management of scope.
 - May use natural language elements, be easier to use, or more portable across platforms.
 - A compiler is needed when used for programming of real-life applications.
 - This greater abstraction and hiding of details is generally intended to make the language user-friendly.
 - A high level language isolates the execution semantics of a computer architecture from the specification of the program, making the process of developing a program simpler and more understandable with respect to a low-level language.
- The amount of abstraction provided defines how 'high level' a programming language is (3GL, 4GL? 5GL??).

Language Classification- Level of programming language (4)



- A very high-level programming language (VHLL) is a programming language with a very high level of abstraction, used primarily as a professional programmer productivity tool.
 - Very high-level programming languages are usually limited to a very specific application, purpose, or type of task.
 - Due to this limitation in scope, they might use syntax that is never used in other programming languages, such as direct English syntax.
 - For this reason, very high-level programming languages are often referred to as goal-oriented programming languages.

Language Classification- Level of programming language (5)



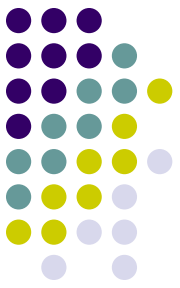
- A third-generation language (3GL)
 - Where as a second generation language is more aimed to fix logical structure to the language, a third generation language aims to refine the usability of the language in such a way to make it more user friendly.
 - First introduced in the late 1950s, Fortran, ALGOL and COBOL are early examples of this sort of language.
 - Most "modern" languages (BASIC, C, C++, C#, Pascal, and Java) are also third-generation languages.
 - Most 3GLs support structured programming.

Language Classification- Level of programming language (6)



- A fourth-generation programming language (1970s-1990, 4GL)
 - Is a programming language or programming environment designed with a specific purpose in mind.
 - In the evolution of computing, the 4GL followed the 3GL in an upward trend toward higher abstraction and statement power.
 - 3GL development methods can be slow and error-prone.
 - Some applications could be developed more rapidly by adding a higher-level programming language and methodology which would generate the equivalent of very complicated 3GL instructions with fewer errors.
 - 4GL and 5GL projects are more oriented toward problem solving and systems engineering.
 - Fourth-generation languages have often been compared to domain-specific programming languages (maybe a sub-set of DSLs).
 - Given the persistence of assembly language even now in advanced development environments, one expects that a system ought to be a mixture of all the generations, with only very limited use of the first.
 - Examples: SQL, IDL

Language Classification- Level of programming language (7)



- A fifth-generation programming language (5GL)
 - Is a programming language based around solving problems using constraints given to the program, rather than using an algorithm written by a programmer.
 - Fifth-generation languages are used mainly in artificial intelligence research.
 - While 4GL are designed to build specific programs, 5GL are designed to make the computer solve a given problem without the programmer.
- However, as larger programs were built, the flaws of the approach became more apparent.
 - It turns out that, starting from a set of constraints defining a particular problem, deriving an efficient algorithm to solve it is a very difficult problem in itself.
 - This crucial step cannot yet be automated and still requires the insight of a human programmer.
 - Today are mostly used in academic circles for research.
- Example: Prolog, OPS5, and Mercury

Specification of programming languages- What we want to describe?



- **How correct program should look like?**
 - SYNTAX
 - Formal languages, grammars, automaton,...
- **What correct program should do?**
 - SEMANTICS
 - Lambda calculus, Attributed grammars,...

Specification of programming languages- **Formal languages**



- **Alphabet**

- Finite set of symbols Σ
- Example: $\{0,1\}$, $\{a, b, c, \dots, z\}$, $\{a,b,+, *,(,)\}$

- **Words over an alphabet Σ**

- Set of symbols from Σ (Σ^*)
- Empty set - ε
- Examples: 1001, pjp, $a*(b+b)$

- **Language over an alphabet Σ**

- A subset of words over an alphabet Σ
- Finite or infinite languages
- Examples:
 $\{0, 00, 11, 000, 011, 101, 110, 0000, 0011, \dots\}$
 $\{\text{int, double, char}\}$
 $\{a, b, a+a, a+b, b+a, b+b, \dots, a*(b+b), \dots\}$

Specification of programming languages- How we can describe a language?



a) **Elements list**

- Finite languages only.

b) **Description in “spoken” language**

- vague, can not be used for computations, complex

c) **Generative systems – grammars**

- Instructions, how we can generate all words in a language.

d) **Detection systems – automaton**

- Instructions, how we can check if a word belongs to a language or does not.

Specification of programming languages– **Grammars (1)**



- $G = (N, T, P, S)$
 - **N – non-terminal symbols**
Can be transformed to a different set of symbols.
 - **T – terminal symbols**
Can not be transformed future.
 - **P – production rules**
 $P \subseteq (N \times T)^* N (N \times T)^* \times (N \times T)^*$
 $\alpha \rightarrow \beta$ α - left side, β - right side
 - **S – start symbol** $S \in N$

Specification of programming languages– Grammars (2)



- Binary numbers

- $N = \{S, D\}$ $T = \{0, 1\}$

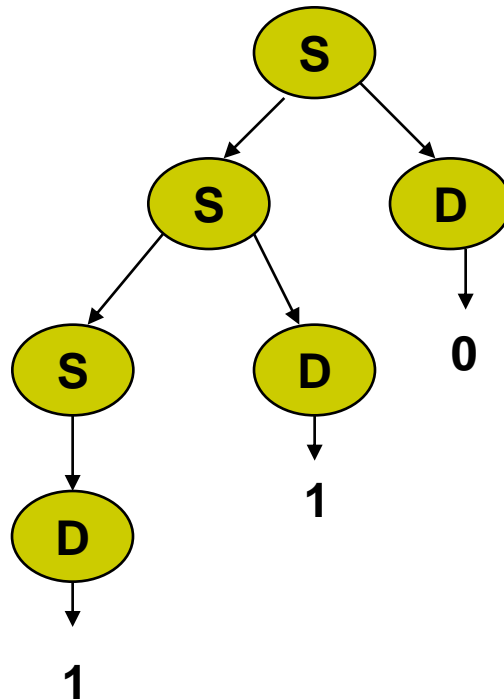
- P: $S \rightarrow D \mid S D$
 $D \rightarrow 0 \mid 1$

- $S \Rightarrow S D \Rightarrow S 0 \Rightarrow S D 0 \Rightarrow D D 0 \Rightarrow 1 D 0 \Rightarrow 1 1 0$
 $S \Rightarrow^* 1 1 0$

Specification of programming languages- Grammar's derivation tree



- $S \Rightarrow SD \Rightarrow S0 \Rightarrow SD0 \Rightarrow DD0 \Rightarrow 1D0 \Rightarrow 110$



Specification of programming languages- Chomsky Language Classification (1)



- Type 0 – Unrestricted languages
 $\alpha \rightarrow \beta$ α, β all possibilities
- Type 1 – Context languages
 $\omega_1 \alpha \omega_2 \rightarrow \omega_1 \beta \omega_2$
- Type 2 – Context free languages
 $A \rightarrow \beta$
- Type 3 – Regular languages
 $A \rightarrow b C$
 $A \rightarrow b$

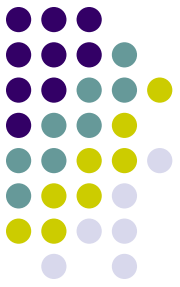
Specification of programming languages-

Chomsky Language Classification (2)

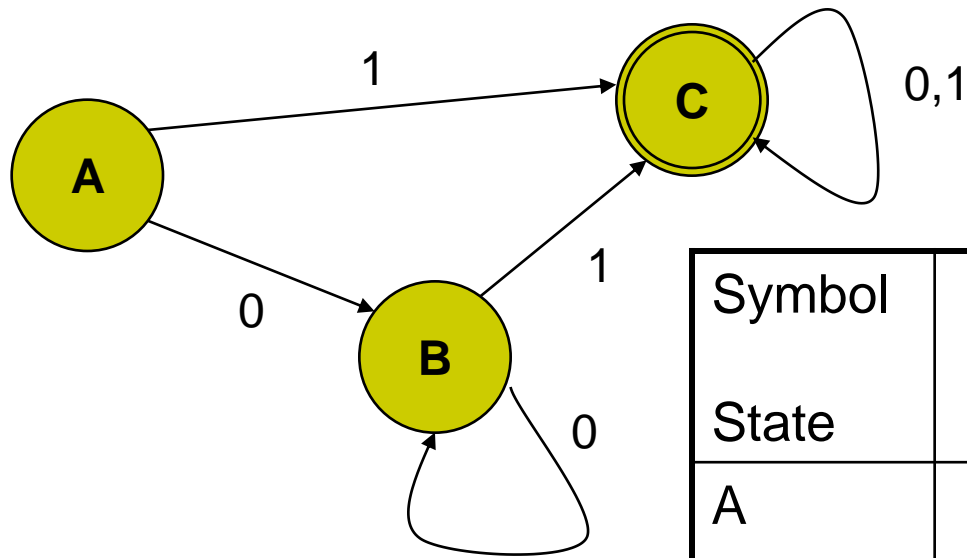


- **Type 0 – Unrestricted languages**
We are unable to compute if word belongs to some language.
Turing's machines
- **Type 1 – Context languages**
Containing real programming languages.
Are unable to analyze effectively
Linearly bound Turing's machines
- **Type 2 – Context free languages**
Can be analyzed very effectively
Pushdown automaton
- **Type 3 – Regular languages**
Even more effective methods to analyze them
Finite automaton

Specification of programming languages- **Finite automaton**



States + transitions



Symbol	0	1
State		
A	B	C
B	B	C
C	C	C

Specification of programming languages- **Finite automaton**s



$$A = (Q, \Sigma, \delta, q_0, F)$$

- Q : a finite set of states
- Σ : input alphabet
- δ : state transition function

$$\delta_{\text{NFA}} : Q \times \Sigma \rightarrow 2^Q$$

$$\delta_{\text{DFA}} : Q \times \Sigma \rightarrow Q$$

- q_0 : initial state
- F : a set of final states

Specification of programming languages – **Syntax's description**



- Three levels of syntax's description
 - Lexical structure (identifiers, numbers, strings)
 - Regular expressions, finite automaton
 - Context free syntax
 - Context free grammars
 - Common programming languages are not context free languages.
 - If - else
 - Context restrictions

Specification of programming languages – Syntax description's methods



- Syntactic graph
- Backus-Naur Form (BNF)

```
<decl>    -> 'DEF' <ident> '=' <expr> <expr1>  
          | 'TYPE' <ident> '=' <type>  
<expr1>   -> ';' <expr> <expr1>  
          | e
```

- Example: DEF a = 1;

- Extended Backus-Naur Form (EBNF)

- Extended with regular expression's operators

```
<decl>    -> 'DEF' <ident> '=' <expr> ( <expr> )*  
          | 'TYPE' <ident> '=' <type>
```


Specification of programming languages- Language's semantics specification



- Semantics reflects the meaning of programs or functions.
- Many different frameworks, none of them considered to be “standard”
- Three main approaches
 - **Axiomatic semantics**
 - Specific properties of the effect of executing the constructs as expressed as *assertions*.
 - Thus there may be aspects of the executions that are ignored.
 - $\{P\} \text{ while } R \text{ do } S \{Q \wedge \neg R\}$
 - **Operational semantics**
 - The meaning of a construct is specified by the computation it induces when it is executed on a machine.
 - In particular, it is of interest *how* the effect of a computation is produced.
 - **Denotation semantics**
 - Meanings are modeled by mathematical objects that represent the effect of executing the constructs.
 - Thus *only* the effect is of interest, not how it is obtained.
 - $E : \text{Expr} \rightarrow (\text{String} \rightarrow \text{Int}) \rightarrow \text{Int}$

Functional programming – Differences between imperative and declarative programming languages



- Imperative languages
 - Imperative languages describes computation in terms of statements that change a program state.
 - Imperative programs define sequences of commands for the computer to perform
 - Explicit term sequence of commands – it express what computer should do and when
 - Statement has a side effects
 - Based on actual (Von Neumann's) computer's architecture
 - Simple and effective implementation
- Declarative languages
 - Programs are likely composed from expressions not from statements.
 - Expresses *what* needs to be done, without prescribing *how* to do it.
 - In terms of sequences of actions to be taken.
 - There is no sequence of commands given.
 - For effective implementation complex optimizations must be performed.
- **Functional and logical programming languages** are characterized by a declarative programming style.

Functional programming – Functional programming languages(1)



- Based on lambda calculus – basic computation's model is a mathematical term function. Functions are applied on arguments and compute results.
- Programs are composed from functions without side effects.
- Functions are considered to be „first-class values“.
- Functional languages have better abstraction mechanisms.
 - High order functions may be used.
 - Function's composition
 - Programs often much shorter
- Functional languages do not contain assignments, cycles, ...
 - Recursion is used instead.
 - Assignment has a mathematical meaning.
 - Variable has the same value in a given context.

Functional programming – Functional programming languages(2)



- Functional languages allow to use new algebraic approaches.
 - **Lazy evaluation** (x **eager evaluation**)
 - We could use infinite structures.
 - We could separate data from execution order – for example for parallelization.
- Functional languages allows new approaches for a application's development.
 - Proofing properties of programs.
 - Possibility to transform program based on algebraic properties.
- Easier parallelization
 - Easy to find parts which can be evaluated in parallel.
 - Functions has no side effects!
 - Often to many parallelisms.
 - We can create new parallel program simply by composing two parallel programs.

Functional programming – λ -calculus



- 1930 Alonzo Church
 - Lambda calculus is a formal system designed to investigate function definition, function application and recursion.
 - Part of an investigation into the foundations of mathematics
- Base for functional languages
- Some constructions present even in imperative languages (for example Python or C#).

Functional programming – Lambda calculus (1)



- Variables
 - x, y, z, f, g, \dots
- λ -abstraction
 - $(\lambda x . e)$
- Application
 - $(e_1 e_2)$
- Parentheses convention
 - $\lambda x . \lambda y . e_1 e_2 = (\lambda x . (\lambda y . e_1 e_2))$
 - $e_1 e_2 e_3 = ((e_1 e_2) e_3)$

Functional programming – Lambda calculus (1)



- λ -Abstraction
 - $\lambda x . e$
 - A function with a parameter x and a body e
 - $\lambda x y . e$
 - A function with parameters x, y and a body e
 - Is equivalent to a notation $\lambda x . (\lambda y . e)$
 - $\lambda e . e (\lambda f x (f x x)) (\lambda f x (f x x))$
- Application
 - $(e_1 e_2)$
 - Application of the function e_1 to the argument e_2
 - $(f x y)$
 - Application of the function $(f x)$ to the argument y
 - Application of the function f to arguments x and y

Functional programming - Substitution



- $e_1 [e_2/x]$
 - replacement of a variable X by expression e_2 every place it is free within e_1
 - Substitution must be correct.
 - We must be careful in order to avoid accidental variable capture.
- $(\lambda x y . f x y) [g z / f] = \lambda x y . (g z) x y$
- $(\lambda x y . f x y) [g z / x] = \lambda x y . f x y$
- $(\lambda x y . f x y) [g y / f] = \textit{error in substitution}$

Functional programming – Evaluation of λ -expressions



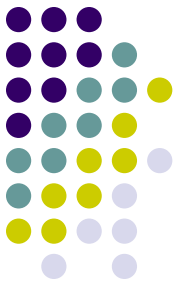
- α -reduction
 - $\lambda x . e \leftrightarrow \lambda y . e[y / x]$
 - Renaming of a captured variable
- β -reduction
 - $(\lambda x . e_1) e_2 \leftrightarrow e_1 [e_2 / x]$
 - “function’s call” – replacing a parameter with an argument
- η -reduction
 - $\lambda x . f x \leftrightarrow f$
 - Removing of an abstraction
 - Variable x must not be free in f
 - *Two functions are the same if and only if they give the same result for all arguments.*
- Substitution must be correct!

Functional programming -

Example

- $(\lambda f x . f x x) (\lambda x y . p y x)$
 $=_{\beta} \lambda x . (\lambda x y . p y x) x x$
 $=_{\alpha} \lambda z . (\lambda x y . p y x) z z$
 $=_{\beta} \lambda z . (\lambda y . p y z) z$
 $=_{\beta} \lambda z . p z z$

- $(\lambda f x . f x x) (\lambda x y . p y x)$
 $=_{\eta} (\lambda f x . f x x) (\lambda y . p y)$
 $=_{\eta} (\lambda f x . f x x) p$
 $=_{\beta} \lambda x . p x x$



Functional programming – Reduction strategies



- *redex* --- **reducible expression**
 - Expression that can be reduced further; α -redex, β -redex.
- *Expression's normal form*
 - Any expression containing no β -redex.
- Reduction strategies - The distinction between reduction strategies relates to the distinction in functional programming languages between eager evaluation and lazy evaluation.
 - **Applicative order**
 - The rightmost, innermost redex is always reduced first.
 - Intuitively this means a function's arguments are always reduced before the function itself.
 - **Eager evaluation** – This is essentially using applicative order, call by value reduction
 - **Normal order**
 - The leftmost, outermost redex is always reduced first.
 - **Call by name**
 - As normal order, but no reductions are performed inside abstractions.
 - **Call by value**
 - Only the outermost redexes are reduced: a redex is reduced only when its right hand side has reduced to a value (variable or lambda abstraction).
 - **Call by need**
 - As normal order, but function applications that would duplicate terms instead name the argument, which is then reduced only "when it is needed" - **lazy evaluation**.



Haskell - Haskell

- September 1991 – Gofer
 - Experimental language
 - Mark P. Jones
- February 1995 – Hugs
- Hugs98
 - Nearly full implementation of programming language Haskell 98
 - Some extension implemented
- Basic resources
 - <http://haskell.org>
 - Language specification and other resources
 - <http://haskell.org/hugs>
 - Installation packages (Win / Unix)
 - User's manual (is a part of installation)



Haskell –Hugs Interpret

- Basic evaluation: calculator
 - `$ hugs`
 - `Prelude> 2*(3+5)`
 - `16`
- Script: containing user's definitions
 - `$ hugs example.hs`
- Editing of source code
 - `:edit [file.hs]`
 - `:e`
- Loading of source code
 - `:load [file.hs]`
 - `:reload`
- Exiting work
 - `:quit`
- Help
 - `?:`



Haskell – Script

- `example.hs`

```
module Example where
-- Function computing sum of two numbers
sum x y = x + y
```

- `Example.lhs`

```
> module Example where
```

```
Function computing factorial
```

```
> f n = if n == 0 then 1 else n * f (n-1)
```



Haskell – Data types(1)

- Basic data types
 - `1::Int`
 - `'a'::Char`
 - `True,False::Bool`
 - `3.14::Float`
- Lists `[a]`
 - Empty list `[]`
 - Non-empty list `(x:xs)`
 - `1:2:3:[] :: [Int]`
 - `[1,2,3] :: [Int]`
- Ordered tuples `(a,b,c,...)`
 - `(1,2) :: (Int,Int)`
 - `(1,['a','b'])::(Int, [Char])`
 - `() :: ()`



Haskell – Data types(2)

- Function $a \rightarrow b$
 - `factorial :: Int -> Int`
 - `sum :: Int -> Int -> Int`
 - `plus :: (Int, Int) -> Int`
- User defined data types
 - `data Color = Black
 | white`
 - `data Tree a = Leaf a
 | Node a (Tree a) (Tree a)`
 - `type String = [Char]`
 - `type Table a = [(String, a)]`



Haskell – Type classes

- Type class – set of types with specific operations
 - Num: +, -, *, abs, negate, signum, ...
 - Eq: ==, /=
 - Ord: >, >=, <, <=, min, max
- Constrains, type class specification
 - `elem :: Eq a => a -> [a] -> Bool`
 - `minimum :: Ord a => [a] -> a`
 - `sum :: Num a => [a] -> a`



Haskell – Function definition

- Equation and pattern unification (pattern matching):
 - `f pat11 pat12 ... = rhs1`
 - `f pat21 pat22 ... = rhs2`
 - ...
- First corresponding equation is chosen.
- If there is none → error



Haskell – Patterns

- variable
 - `inc x = x + 1`
- constant
 - `not True = False`
 - `not False = True`
- List
 - `length [] = 0`
 - `length (x:xs) = 1 + length xs`
- tuples
 - `plus (x,y) = x+y`
- User's type constructor
 - `n1 (Leaf _) = 1`
 - `n1 (Node _ l r) = (n1 l) + (n1 r)`
- Named pattern's parts
 - `duphd p@(x:xs) = x:p`
- Another patterns - $n+k$
 - `fact 0 = 1`
 - `fact (n+1) = (n+1)*fact n`



Haskell – Example

- Factorial

- $\text{fakt1 } n = \text{if } n == 0 \text{ then } 1$
 $\text{else } n * \text{fakt1 } (n-1)$

- $\text{fakt2 } 0 = 1$
 $\text{fakt2 } n = n * \text{fakt2 } (n-1)$

- $\text{fakt3 } 0 = 1$
 $\text{fakt3 } (n+1) = (n+1) * \text{fakt3 } n$

- $\text{fakt4 } n \mid n == 0 = 1$
 $\mid \text{otherwise} = n * \text{fakt4 } (n-1)$

- Fibonacci numbers

```
fib :: Int -> Int
```

```
fib 0 = 0
```

```
fib 1 = 1
```

```
fib (n+2) = fib n + fib (n+1)
```



Haskell – Example

- List length
 - $\text{length } [] = 0$
 $\text{length } (x:xs) = 1 + \text{length } xs$
- Comment: be aware of name conflict with previously defined functions!
 - ```
module Example where
import Prelude hiding(length)

length [] = 0
length (_:xs) = 1 + length xs
```



# Haskell – Local definition

- Construction *let ... in*
  - $f\ x\ y = \text{let } p = x + y$   
                                   $q = x - y$   
                                  in  $p * q$
- Construction *where*
  - $f\ x\ y = p * q$   
                  where  $p = x + y$   
                           $q = x - y$

# Haskell – Partial function application



- `inc x = 1 + x`
- `inc x = add 1 x`
- `inc = add 1`
- `inc = (+1) = (1+)`
- `add = (+)`
  
- Eta reduction
  
- Point free programming
  - `lcaseString s = map toLower s`
  - `lcaseString = map toLower`



# Haskell – Lambda abstraction

- **Using function like a parameter**

```
nonzero xs = filter p xs
 where p x = x /= 0
```

```
nonzero xs = filter (/= 0) xs
```

```
nonzero xs = filter (\x -> x/=0) xs
```

- $\lambda x \rightarrow e \quad \dots \quad \lambda x . e$

- $\text{inc} = \lambda x \rightarrow x+1$
- $\text{plus} = \lambda(x,y) \rightarrow x + y$
- $\text{dividers } n = \text{filter } (\lambda m \rightarrow n \text{ `mod` } m == 0) [1..n]$





# Haskell – Example

- Example creating a list of squared numbers

- ```
dm [] = []  
dm (x:xs) = sq x : dm xs  
           where sq x = x * x
```

- List's ordering (quicksort)

- ```
qs [] = []
qs (x:xs) =
 let ls = filter (< x) xs
 rs = filter (>=x) xs
 in qs ls ++ [x] ++ qs rs
```

# Haskell – Functions manipulating with lists(1)



- Access to list's elements
  - `head [1,2,3] = 1`
  - `tail [1,2,3] = [2,3]`
  - `last [1,2,3] = 3`
  - `init [1,2,3] = [1,2]`
  - `[1,2,3] !! 2 = 3`
  - `null [] = True`
  - `length [1,2,3] = 3`

# Haskell – Functions manipulating with list (2)



- List's union
  - $[1,2,3] ++ [4,5] = [1,2,3,4,5]$
  - $[[1,2],[3],[4,5]] = [1,2,3,4,5]$
  - $\text{zip } [1,2] [3,4,5] = [(1,3),(2,4)]$
  - $\text{zipwith } (+) [1,2] [3,4] = [4,6]$
- List's aggregation
  - $\text{sum } [1,2,3,4] = 10$
  - $\text{product } [1,2,3,4] = 24$
  - $\text{minimum } [1,2,3,4] = 1$
  - $\text{maximum } [1,2,3,4] = 4$

# Haskell – Functions manipulating with list (3)



- Selecting list's parts
  - `take 3 [1,2,3,4,5] = [1,2,3]`
  - `drop 3 [1,2,3,4,5] = [4,5]`
  - `takeWhile (>0) [1,3,0,4] = [1,3]`
  - `dropWhile (> 0) [1,3,0,4] = [0,4]`
  - `filter (>0) [1,3,0,2,-1] = [1,3,2]`
- List's transformations
  - `reverse [1,2,3,4] = [4,3,2,1]`
  - `map (*2) [1,2,3] = [2,4,6]`



# Haskell – Arithmetic rows

- $[m..n]$ 
  - $[1..5] = [1,2,3,4,5]$
- $[m1,m2..n]$ 
  - $[1,3..10] = [1,3,5,7,9]$
- $[m..]$ 
  - $[1..] = [1,2,3,4,5,\dots]$
- $[m1,m2..]$ 
  - $[5,10..] = [5,10,15,20,25,\dots]$



# Haskell – Function filter

Obtaining a part of list corresponding to given rule  
(predicate)

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter _ [] = []
```

```
filter p (x:xs) | p x = x : filter p xs
 | otherwise = filter p xs
```

```
filter even [1..10] = [2,4,6,8]
```

```
filter (> 0) [1,3,0,2,-1] = [1,3,2]
```

```
dividers n = filter deli [1..n]
```

```
 where deli m = n `mod` m == 0
```



# Haskell – Function map

- List's elements

```
map :: (a -> b) -> [a] -> [b]
```

```
map f [] = []
```

```
map f (x:xs) = f x : map f xs
```

```
map (+1) [1,2,3] = [2,3,4]
```

```
map toUpper "abcd" = "ABCD"
```

```
squares x = map (\x -> x * x) [1..]
```



# Haskell – List's generators

Example: A set of even numbers from 1 to 10

- $\{x \mid x \in 1..10, x \text{ is even}\}$
- $[x \mid x \leftarrow [1..10], \text{even } x]$
  
- $[x \mid x \leftarrow xs] = xs$
- $[f\ x \mid x \leftarrow xs] = \text{map } f\ xs$
- $[x \mid x \leftarrow xs, p\ x] = \text{filter } p\ xs$
  
- $[(x, y) \mid x \leftarrow xs, y \leftarrow ys] =$   
     $[(x_1, y_1), (x_1, y_2), (x_1, y_3), \dots,$   
     $(x_2, y_1), (x_2, y_2), (x_2, y_3), \dots,$   
     $\dots ]$





# Haskell – Example

- Set's operation using list's generators

- Intersection

```
intersect xs ys = [y | y <-ys, elem y xs]
```

- Union

```
union xs ys = xs ++ [y | y<-ys, notElem y xs]
```

- Difference

```
diff xs ys = [x | x<-xs, notElem x ys]
```

- Subset

```
subset xs ys = [x | x<-xs, notElem x ys] == []
subset xs ys = all (\x -> elem x ys) xs
```

# Haskell – Definition of user's types



- `data Color = Red | Green | Blue`
  - Color –type's constructor
  - Red / Green / Blue – data constructor
- `data Point = Point Float Float`
  - `dist (Point x1 y1) (Point x2 y2) =  
sqrt ((x2-x1)**2 + (y2-y1)**2)`
  - `dist (Point 1.0 2.0) (Point 4.0 5.0) = 5.0`
- `data Point a = Point a a`
  - Polymorphism
  - Constructor `Point :: a -> a -> Point a`

# Haskell – Recursive data types



## Tree

```
data Tree1 a = Leaf a
 | Branch (Tree1 a) (Tree1 a)
```

```
data Tree2 a = Leaf a
 | Branch a (Tree2 a) (Tree2 a)
```

```
data Tree3 a = Null
 | Branch a (Tree3 a) (Tree3 a)
```

```
t2l (Leaf x) = [x]
```

```
t2l (Branch lt rt) = (t2l lt) ++ (t2l rt)
```



# Haskell – Type's Synonyms

- `type String = [Char]`

```
type Name = String
data Address = None | Addr String
type Person = (Name, Address)
```

```
type Table a = [(String, a)]
```

- They are equivalent to original types
- They represent only a shortcuts



# Haskell – Basic type classes

|                              |                                |
|------------------------------|--------------------------------|
| Eq a                         | (==), (/=)                     |
| Eq a => Ord a                | (<), (<=), (>), (>=), min, max |
| Enum a                       | succ, pred                     |
| Read a                       | readsPrec                      |
| Show a                       | showsPres, show                |
| (Eq a, Show a) => Num a      | (+), (-), (*), negate, abs     |
| (Num a) => Fractional a      | (/), recip                     |
| (Fractional a) => Floating a | pi, exp, log, sqrt, (**), ...  |



# Haskell – Type class *Show*

- Values that can be converted to a string
  - `type Shows = String -> String`  
`class Show a where`
    - `showsPrec :: Int -> a -> Shows`
    - `show :: a -> String`
    - `showList :: [a] -> Shows`
  - `showPoint :: Point -> String`  
`showPoint (Point x y) =`  
`“(“ ++ show x ++ ”;” ++ show y ++ ”)”`
  - `instance Show Point where`
    - `show p = showPoint p`



# Haskell – Type class *Read*

- Values readable from a string
  - `type Reads a = String -> [(a,String)]`  
`class Read a where`
    - `readsPrec :: Int -> Reads a`
    - `readList :: Reads [a]`
  - `readsPoint :: Reads Point`  
`readsPoint ('(' : s) =`
    - `[ (Pt x y, s'') |`
      - `(x, ';' : s') <- reads s,`
      - `(y, ')' : s;;) <- reads s']`
  - `instance Read Point where`
    - `readsPrec _ = readsPoint`

# Haskell – Programming with Actions



- Imperative languages
  - Program is a sequence of statements
    - Straight forward and clear sequence of actions
    - Side effects
  - We can for example easily use global variables, read and write file,...
- Haskell (simplified)
  - Actions are divided from pure functional code.
  - *Monadic operators*
  - Actions is a function which's result is of type: **(IO a)**.



# Haskell – Programming with Actions Example



- Char's read and write
  - `getChar :: IO Char`  
`putChar :: Char -> IO ()`
- Transformation of a function to a action
  - `return :: a -> IO a`
- Test: y/n check – sequence of actions
  - `ready :: IO Bool`  
`ready = do c <- getChar`  
`return (c == 'y')`



# Haskell – Function *main*

- Represents main program
  - Action returning nothing:
    - ```
main :: IO ()  
main = do c <- getChar  
          putChar c
```
1. Reads character and marks it `c`.
 2. Write character `c`.
 3. Returns the result of last action – `IO()`.

Haskell – Line reader example



1. Program reads first character.
2. If program reads end of line character then program returns readied string.
3. Otherwise program adds readied character to a result.

```
getLine :: IO String
getLine = do x <- getChar
             if x=='\n' then return ""
             else do xs <- getLine
                    return (x:xs)
```



Haskell – Writing of a *string*

- We can use function `putChar` on every character.
For example:
 - `map putChar xs`
 - The result is a list of actions.
 - `map :: (a -> b) -> [a] -> [b]`
`putChar :: Char -> IO ()`
`map putChar s :: [IO ()]`
- Can be transformed to a single action.
 - `sequence :: [IO()] -> IO ()`
`putStr :: String -> IO ()`
`putStr s = sequence (map putChar s)`

Haskell – Proving using mathematical induction



- The simplest and most common form of mathematical induction proves that a statement involving a natural number n holds for all values of n .
 - The proof consists of two steps:
 - The **basis (base case)**: showing that the statement holds when $n = 0$.
 - The **inductive step**: showing that *if* the statement holds for some n , *then* the statement also holds when $n + 1$ is substituted for n .
- Structural induction for lists.
 - a) We prove a statement for empty list - []
 - b) If a statement holds for xs , then we show that it also holds for $(x:xs)$.

Haskell – Example – Associativity of ++ (1)



$$(xS ++ yS) ++ zS = xS ++ (yS ++ zS)$$

$$[] ++ yS = yS \quad (++) .1$$

$$(x:xS) ++ yS = x: (xS ++ yS) \quad (++) .2$$

a) [] => xS

$$\begin{aligned} & ([] ++ yS) ++ zS \\ &= yS ++ zS \quad (++) .1 \end{aligned}$$

$$= [] ++ (yS ++ zS) \quad (++) .1$$

Haskell – Example – Associativity of ++ (2)



$$(XS ++ YS) ++ ZS = XS ++ (YS ++ ZS)$$

$$[] ++ YS = YS \quad (++) . 1)$$

$$(X:XS) ++ YS = X: (XS ++ YS) \quad (++) . 2)$$

b) $(X:XS) \Rightarrow XS$

$$\begin{aligned} & ((X:XS) ++ YS) ++ ZS \\ &= X:(XS ++ YS) ++ ZS && (++) . 2) \\ &= X:((XS ++ YS) ++ ZS) && (++) . 2) \\ &= X:(XS ++ (YS ++ ZS)) && \text{(assumption)} \\ &= (X:XS) ++ (YS ++ ZS) && (++) . 2) \end{aligned}$$

Haskell – Example – length ($xs++ys$) (1)



$\text{length } (xs++ys) = \text{length } xs + \text{length } ys$

$\text{length } [] = 0$ (len.1)

$\text{length } (_ : xs) = 1 + \text{length } xs$ (len.2)

a) $[] \Rightarrow xs$

$\text{length } ([] ++ ys)$

$= \text{length } ys$ (++ .1)

$= 0 + \text{length } ys$ (base case +)

$= \text{length } [] + \text{length } ys$ (len.1)

Haskell – Example – length ($xs++ys$) (2)



$\text{length } (xs++ys) = \text{length } xs + \text{length } ys$

$\text{length } [] = 0$ (len.1)

$\text{length } (_ : xs) = 1 + \text{length } xs$ (len.2)

b) $(x:xs) \Rightarrow xs$

$\text{length } ((x:xs) ++ ys)$
= $\text{length } (x:(xs++ys))$ (++.2)
= $1 + \text{length } (xs++ys)$ (len.2)
= $1 + (\text{length } xs + \text{length } ys)$ (Assumption)
= $(1 + \text{length } xs) + \text{length } ys$ (+ Associativity)
= $\text{length } (x:xs) + \text{length } ys$ (len.2)



Logické jazyky - Introduction

- Hlavní myšlenka:
Využití počítače k vyvozování důsledků na základě deklarativního popisu
- Postup:
 - reálný svět →
 - zamýšlená interpretace →
 - logický model →
 - program
- Výpočet - určení splnitelnosti či nespłnitelnosti **cíle**, případně včetně vhodných **substitucí**.
- Pro Example použít jazyk Prolog

Logické jazyky - Logický program



- **Fakta**

```
vek (petr, 30) .
```

```
vek (jana, 24) .
```

- **Pravidla**

```
starsi (X, Y) :-
```

```
    vek (X, V1) , vek (Y, V2) , V1 > V2 .
```

- **Dotazy**

```
?- starsi (petr, jana) .
```

```
Yes
```

Logické jazyky - Co je to dotaz?



- Odpověď na dotaz vzhledem k programu = určení, zda je dotaz logickým důsledkem programu.
- Logické důsledky se odvozují aplikací dedukčních pravidel, např.
$$P \vdash P \quad (\text{identita})$$
- Je-li nalezen fakt identický dotazu, dostaneme Yes.
- Odpověď No znamená pouze to, že z programu nelze platnost dotazu vyvodit.

Logické jazyky - Předpoklad uzavřeného světa



```
zvire (pes) .
```

```
zvire (kocka) .
```

```
?- zvire (pes) .
```

Yes

```
?- zvire (zirafa) .
```

No

=> Předpokládáme platnost pouze toho, co je uvedeno v programu.

Logické jazyky - Logická proměnná



- Představuje nespecifikovaný objekt
- Jméno začíná velkým písmenem

?- vek(jana, X) .

X = 24 .

?- vek(pavla, X) .

No

- Existuje X takové, že vek(jana, X) lze odvodit z programu? Pokud ano, jaká je hodnota X?

Logické jazyky - **Kvantifikátory**



- likes(X , beer).
Pro všechna X platí likes(X , beer).

- ?- likes(X , beer).
Existuje X takové, že likes(X , beer)?



Logické jazyky - Term

- Datová struktura definovaná rekurzivně:
 - Konstanty a proměnné jsou termy
 - Struktury jsou termy: $funktor(arg1, arg2, \dots)$
 - *funktor*: jméno začínající malým písmenem
 - *argument*: term
- Funktor je určen **jménem a aritou**
 - $f(t_1, t_2, \dots, t_n) \dots f/n$
- Příklad:
 - $z/0 \quad s/1 \quad \dots \quad z, s(z), s(s(z)), s(s(s(z)))$

Logické jazyky - Substituce



- **Základní term** (ground term)
 - neobsahuje proměnné $s(s(z))$
- **Substituce**
 - Konečná množina dvojic ve tvaru $X_i=t_i$
 - Aplikace substituce θ na term $A \dots A\theta$
 $f(X, a) \{X=g(z), Y=b\} = f(g(z), a)$
- **Instance termu**
 - A je instancí B , existuje-li substituce θ taková, že
 $A = B\theta$
 $f(g(z), a)$ je instancí termu $f(X, a)$

Logické jazyky - Konjunktivní dotazy



- ?- zvire(pes), zvire(kocka).
Yes

Sdílení proměnných:

- ?- vek(X, V), vek(Y, V).
Existují X a Y se stejným věkem?



Logické jazyky - Pravidla

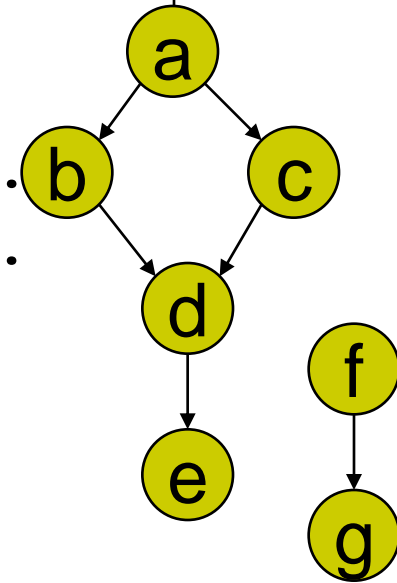
- $A :- B_1, B_2, \dots, B_n$.
A = **hlava** pravidla
 B_1, B_2, \dots, B_n = **tělo** pravidla
- $\text{syn}(X, Y) :- \text{otec}(Y, X), \text{muz}(X)$.
 $\text{deda}(X, Y) :- \text{otec}(X, Z), \text{otec}(Z, Y)$.
- Proměnné jsou univerzálně kvantifikované.
- Platnost proměnných je celé pravidlo.

Logické jazyky - Rekurzivní pravidla



- **Definice grafu**

`edge (a, b) . edge (a, c) . edge (b, d) .`
`edge (c, d) . edge (d, e) . edge (f, g) .`



- `connected (N, N) .`

`connected (N1, N2) :-`

`edge (N1, L) , connected (L, N2) .`

Skriptovací jazyky - Obsah



- Co jsou to skriptovací jazyky
- Výhody a nevýhody skriptovacích jazyků
- Hlavní oblasti použití
- Example jazyků: Perl, Python, Java Script

- Jazyk PHP - Introduction

Skriptovací jazyky - Skriptovací jazyky (1)



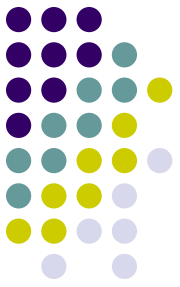
- Jazyky určené k rozšíření nebo propojení existujících aplikací a komponent
 - Uživatelem definované Function (např. editory)
 - Grafické uživatelské rozhraní (Tcl, VB)
 - Webový server (PHP) nebo klient (Java Script)
- Nepoužívají se obvykle ke složitým výpočtům nebo k práci se složitými datovými strukturami

Skriptovací jazyky - Skriptovací jazyky (2)



- Obvykle netyповané (nebo slabě typované)
 - Automatická konverze typů
 - Proměnné mohou obsahovat cokoliv
- Obvykle interpretované
 - Nevyžadují samostatný překlad
 - Možnost měnit části programu za běhu
- Vestavěné složitější typy a operátory
 - Seznamy, vyhledávací tabulky

Skriptovací jazyky - Výhody skriptovacích jazyků



- **Rychlý vývoj aplikací**
- **Jednoduchá instalace aplikací**
 - často stačí pouze zkopírovat zdrojové soubory
- **Integrace s existujícími technologiemi**
 - např. komponentní technologie
- **Jednoduchost určení a použití**
- **Dynamické vlastnosti**
 - např. typování, rozsahy polí, konverze

Skriptovací jazyky - Example



- `select | grep scripting | wc` *(sh)*
- `button .b -text Hello! -font {Times 16}`
`-command {puts hello} (Tcl)`
- Java: 7 řádků
- C++ (MFC): 25 řádků

Skriptovací jazyky - Nevýhody skriptovacích jazyků



- **Neúplnost**
 - předpokládá se spolupráce s „normálními“ jazyky
- **Nesoulad s pravidly „dobrého“ návrhu**
 - strukturování programu
 - objektově orientované programování
- **Zaměření na konkrétní oblast**
 - např. PHP pro dynamické WWW stránky

Skriptovací jazyky - Použití skriptovacích jazyků



- **Správa systému**
 - Řízení startu a ukončení činnosti systému
 - Základní systémové operace – např. archivace
 - Provádění dávkových operací
 - Shell – JCL, COMMAND/CMD, bash
- **Automatizace tvorby programů**
 - Často se opakující činnosti (překlad, instalace)
 - Ant – uživatelem definované činnosti

Skriptovací jazyky - Použití skriptovacích jazyků



- Přizpůsobení aplikací
 - Windows Scripting Host (WSH) – integrováno do operačního systému (VBScript, JScript)
 - Makra v textových editorech – VBA (MS Office), Office Basic (Sun StarOffice), eLISP (emacs)
- Přizpůsobení zařízení
 - Měřicí přístroje s vestavěnými Tcl

Skriptovací jazyky - Hlavní oblasti použití



- GUI – grafické uživatelské rozhraní
 - Visual Basic, Tcl/Tk
- Internet
 - Perl, JavaScript, PHP
- Komponentní technologie
 - Visual Basic

Skriptovací jazyky – Perl (1)



- Practical Extraction and Report Language
- <http://www.perl.com/>
- Populární mezi administrátory Unixu
- Obtížně čitelná syntaxe, mnoho implicitních vlastností

Skriptovací jazyky - Python



- <http://www.python.org/>
- Původně vyvinutá jako komponenta operačního systému Amoeba
- Jednodušší syntaxe
- Jython – běží pod JVM

Skriptovací jazyky – Javascript



- Netscape Corp. – pro prohlížeč
- „Java...” je zavádějící – mnoho odlišností
 - Java: jazyk založený na třídách a dědičnosti
 - JS: jazyk založený na prototypech
- JScript (MS), ECMAScript (European Computer Manufacturer’s Association)
- Sun StarOffice, Macromedia Flash

Skriptovací jazyky – Porovnávání jazyků



Perl

```
for $i (0 .. 6000-1) {  
    %x=();  
    for $j (0 .. 1000-1) {  
        x{$j}=$i;  
        $x{$j}  
    }  
}
```

Python

```
for i in range (6000) :  
    x={}  
    for j in range (1000):  
        x[j]=I  
        x[j]
```

Java

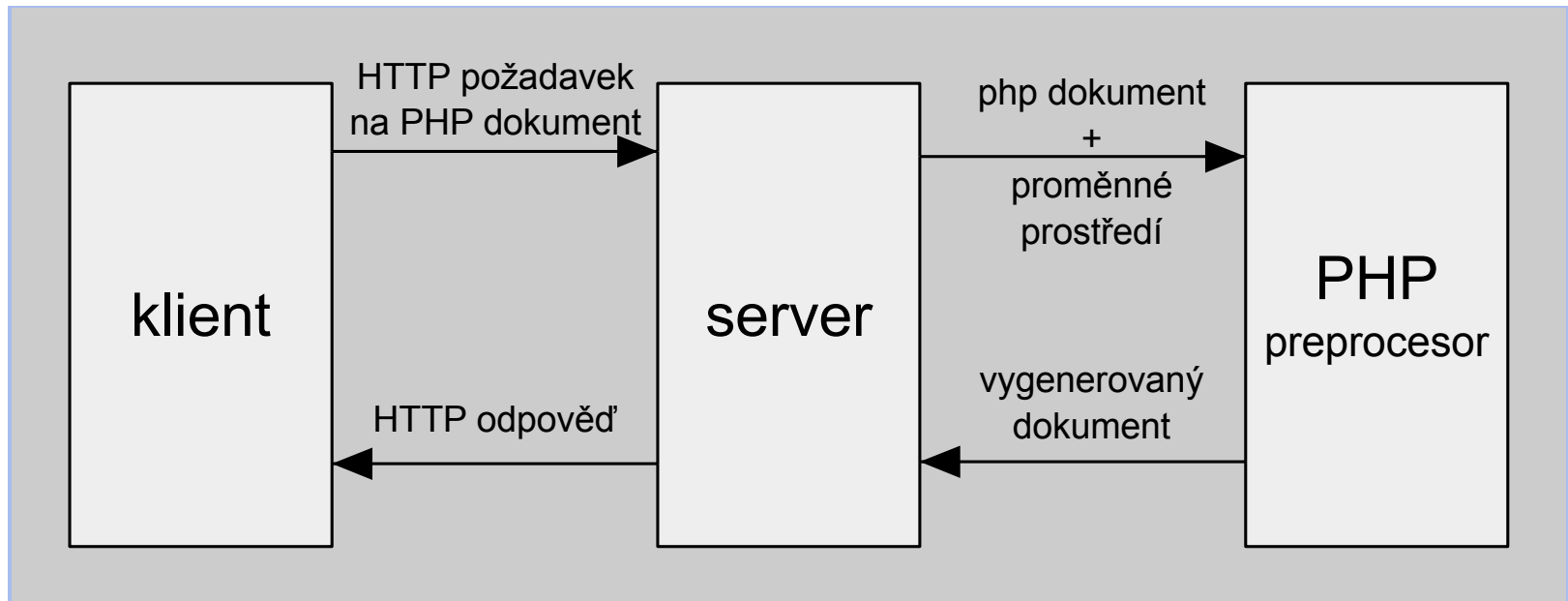
```
import java.util.*;  
public class Test {  
  
    public static void main(  
        String[] args) {  
        for(int i=0; i<6000;i++) {  
            Map x=new HashMap();  
            for (int j=0; j<1000; j++){  
                Integer I=new Integer(i);  
                Integer J=new Integer(j);  
                x.put(I, J); x.get(I);  
            }  
        }  
    }  
}
```

Skriptovací jazyky – Internet



- Protokol HTTP
 - protokol pro přenos dat mezi klientem a webovým serverem
 - typy požadavků
 - GET, POST, HEAD
- Statické stránky
 - Protokol HTML (XHTML)
 - Soubor s příponou .html, .htm

Skriptovací jazyky – Webové aplikace



Skriptovací jazyky – HTTP

požadavek GET



```
GET /index.html HTTP/1.0
User-Agent: Mozilla/5.0
Accept: text/plain, text/xml, text/html, ...
Accept-Language: en
```

```
HTTP/1.0 200 OK
Date: Sun, 02 October 2005 20:19:32 GMT
Content-Type: text/plain
Content-Length: 32
```

Toto je obsah souboru `index.html`



Skriptovací jazyky – PHP

- <http://www.php.net/>
- Původně pro návrh WWW stránek (Personal Home Page)
- K dispozici zdarma pro všechny OS
- Syntaxe podobná C/C++
- Hlavní oblasti
 - Skripty na straně serveru
 - Skripty spouštěné z příkazového řádku



Skriptovací jazyky – PHP

- Verze PHP5: kompletní objektový model
- Spolupráce s mnoha databázemi
 - MySQL, PostgreSQL, ODBC, Oracle, DB2, ...
- Přístup k dalším službám
 - LDAP, IMAP, SNMP, NNTP, POP3, HTTP, ...
- Napojení na jiné technologie
 - Java, COM
- Silná podpora zpracování textu, regulární výrazy, XML, komprese dat, ...

Skriptovací jazyky – PHP a Internet



- Zdrojový text je HTML obsahující úseky programu v PHP:

```
<p><?php echo "ahoj";?></p>
```

```
<p><? echo date('Y-m-d') ?></p>
```
- Skripty jsou umístěny někde v adresáři `~/public_html/` s příponou `.php` (linux456)
- Je třeba zajistit, aby měl webový server právo číst soubory `.php` (příkaz `chmod`)

Skriptovací jazyky – PHP - Proměnné



- Uživatelské proměnné
 - Nedeklarují se
 - Jejich jméno začíná znakem \$
 - `$x = 10;`
if (`$x > 0`) echo “\$x je kladné”;
- Systémové proměnné
 - `$GLOBALS`
 - `$_REQUEST`, `$_SERVER`, `$_SESSION`, ...

Skriptovací jazyky – PHP – Pole (1)



- Indexovaná pole
 - `$a = array();`
 - `$a[0] = 10; $a[1] = 5;`
 - `$a = array (0=>10, 1=>5);`
- Asociativní pole
 - `$a = array();`
 - `$a["Po"] = "Pondělí";`
 - `$a = array ("Po"=>"Pondělí", "Ut"=>"Úterý", ...)`

Skriptovací jazyky – PHP – Pole (2)



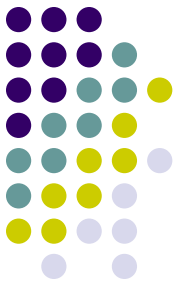
- Průchod polem
 - for (\$i = 0; \$i < **count**(\$a); \$i++)
 echo “a[\$i] = {\$a[\$i]}\n”;
 - **foreach** (\$a **as** \$i => \$v) echo “a[\$i] = \$v\n”;
 - **foreach** (\$a **as** \$v) echo “\$v”;

Skriptovací jazyky – PHP - Příklad – generování tabulky



```
<table border="1">
<?
    for($i = 0; $i < 10; $i++) {
        echo "<tr>\n";
        echo "    <td>$i</td>\n";
        echo "    <td>", $i * $i, "</td>\n";
        echo "</tr>\n";
    }
?>
</table>
```

Skriptovací jazyky – PHP - Další řídicí konstrukce



- *if (podmínka) příkaz*
- *if (podmínka) příkaz else příkaz*
- *while (podmínka) příkaz;*
- *do příkaz while (podmínka);*
- *break;*
- *continue;*
- *switch (výraz) příkaz*
- *include “soubor”; require “soubor”;*

Skriptovací jazyky – PHP - Příklad



```
<? if ( $pocet > 0 ) { ?>  
<p> Počet = <? echo $pocet ?> </p>  
<? } ?>
```

```
<? Switch ( $den ) {  
    case "So": case "Ne":  
        $vikend = true;  
        break;  
    default:  
        $vikend = false;  
        break;  
} ?>
```

Skriptovací jazyky – PHP - Function



- ```
function soucet ($x, $y = 1) {
 return $x +$y;
}
```

- Všechny proměnné jsou lokální, globální proměnné se musí deklarovat:

```
global $g;
```

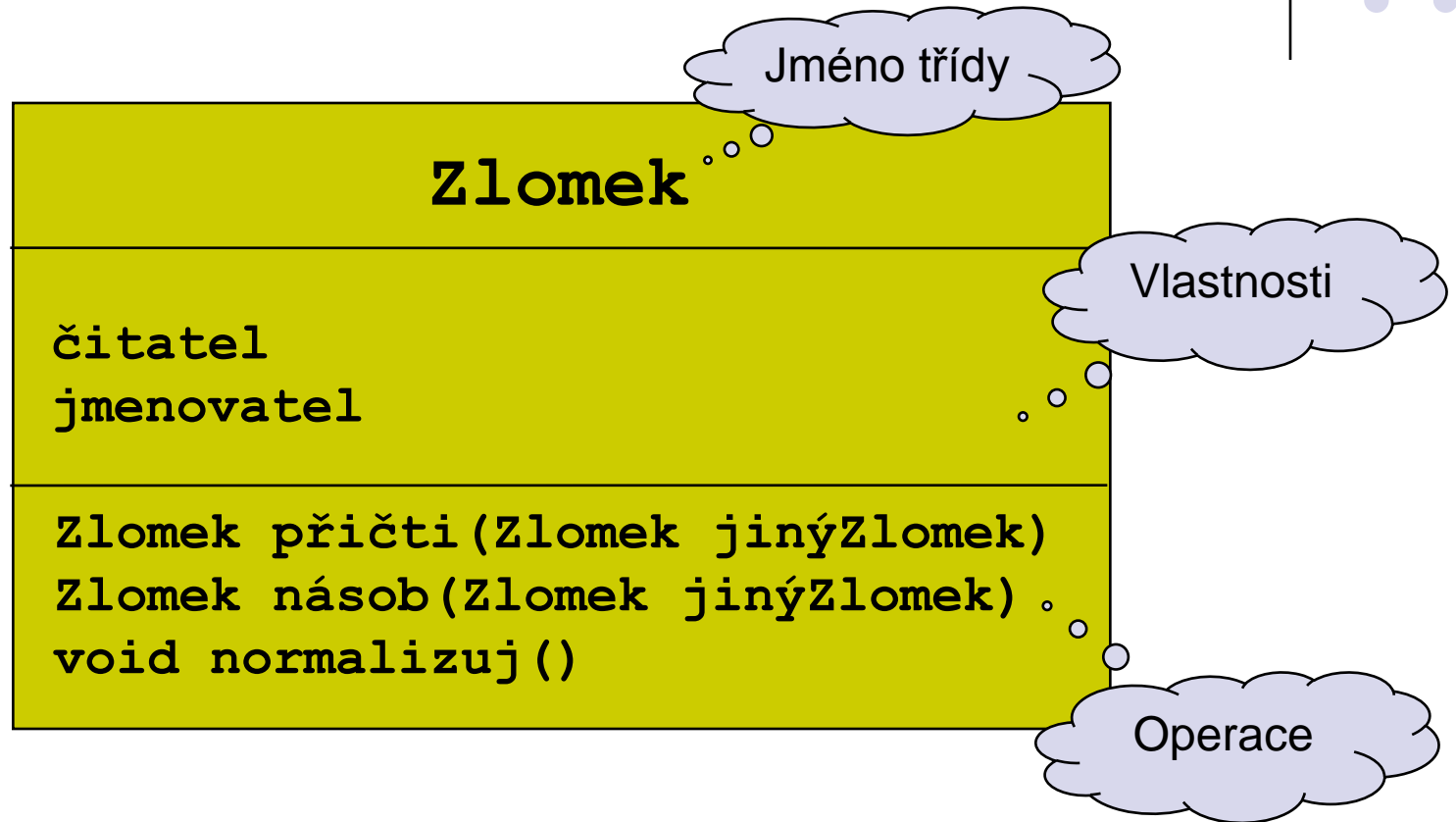
# Skriptovací jazyky – PHP - Příklad



```
function table_row ($row) {
 echo "<tr>\n";
 foreach ($row as $v) {
 echo " <td>$v</td>\n";
 }
 echo "</tr>\n";
}

echo "<table border=`1`>\n";
 table_row(array(1,2,3,4,5));
echo "</table>\n";
```

# Skriptovací jazyky – PHP - Třídy





# Skriptovací jazyky – PHP -

## Třídy v jazyce Java



```
class Zlomek {
// instanční proměnné
 int cit;
 int jm;
 public Zlomek(int jm, int cit) {
 this.jm=jm;
 this.cit=cit;
 }
// metody
 Zlomek soucin(Zlomek jiny)
 {
 citatel *= jiny.citatel;
 jmenovatel *= jiny.jmenovatel;
 }
}
```

# Skriptovací jazyky – PHP -

## Třídy a objekty



```
//PHP4
```

```
class Zlomek {
 var $cit, &jm;

 function Zlomek($c, $j) {
 $this->cit = $c;
 $this->jm = $j;
 }

 function soucin($z) {
 $this->cit *= $z->cit;
 $this->jm *= $z->jm;
 }

}
```

```
//PHP5
```

```
Class Zlomek {
 public $cit, $jm;

 function __constructor($c,
$j)
 {
 $this->cit = $c;
 $this->jm = $j;
 }

 ...
}
```

# Skriptovací jazyky – PHP - Třídy a objekty



- Vytvoření instance třídy

```
$z = new Zlomek(3, 5);
```

- Přístup k atributům a metodám objektu

```
$z->soucin(new Zlomek(2, 3));
echo "$z->cit / $z->jm";
```

# Skriptovací jazyky – PHP - Dědičnost



```
class LepsiZlomek extends Zlomek
{
 function LepsiZlomek($c=1, $j=1)
 {
 //konstruktor předka se nevolá automaticky!
 Zlomek::Zlomek($c, $j);
 }
 ...
}
```

# Skriptovací jazyky – PHP - Novinky v PHP5



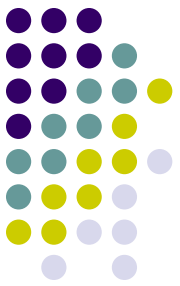
- Konstruktory a destruktory
  - `__construct()` `__destruct()`
- Viditelnost atributů a metod
  - `public`, `protected`, `private`
- Statické atributy a metody
  - `public static $x = "abcd";`
  - ... `Třída::$x`
- Abstraktní třídy a metody, rozhraní
- Reflexe

# Skriptovací jazyky – PHP - Reference na objekt (1)



- PHP4
  - operátor = vytváří kopii objektu
  - operátor =& vytváří referenci na objekt
- PHP5
  - operátor = vytváří kopii reference na objekt (jako v Javě)
  - operátor =& pořád vytváří referenci na objekt

# Skriptovací jazyky – PHP - Reference na objekt (2)



```
$zlomek1 = new Zlomek(1, 2);
$zlomek2 = $zlomek1;
$zlomek1->soucín(new Zlomek(1, 2));
$echo "$zlomek2->cit/$zlomek2->jm";
//výsledek bude 1/2 v PHP4
//výsledek bude 1/4 v PHP5

//úprava 2. řádku pro stejný výsledek v
 PHP4 i PHP5
$zlomek2 =& $zlomek1
```

# Skriptovací jazyky – PHP - Reference na objekt (3)



```
$zlomek1 = new Zlomek (1, 2) ;
$zlomek2 =& $zlomek1 ;
$zlomek2 = new Zlomek (1, 3) ;

// $zlomek1 bude uvolněn !
```



# Skriptovací jazyky – PHP - Výjimky (PHP5)



```
try {
 $error = 'Always throw this error' ;
 throw new Exception($error) ;

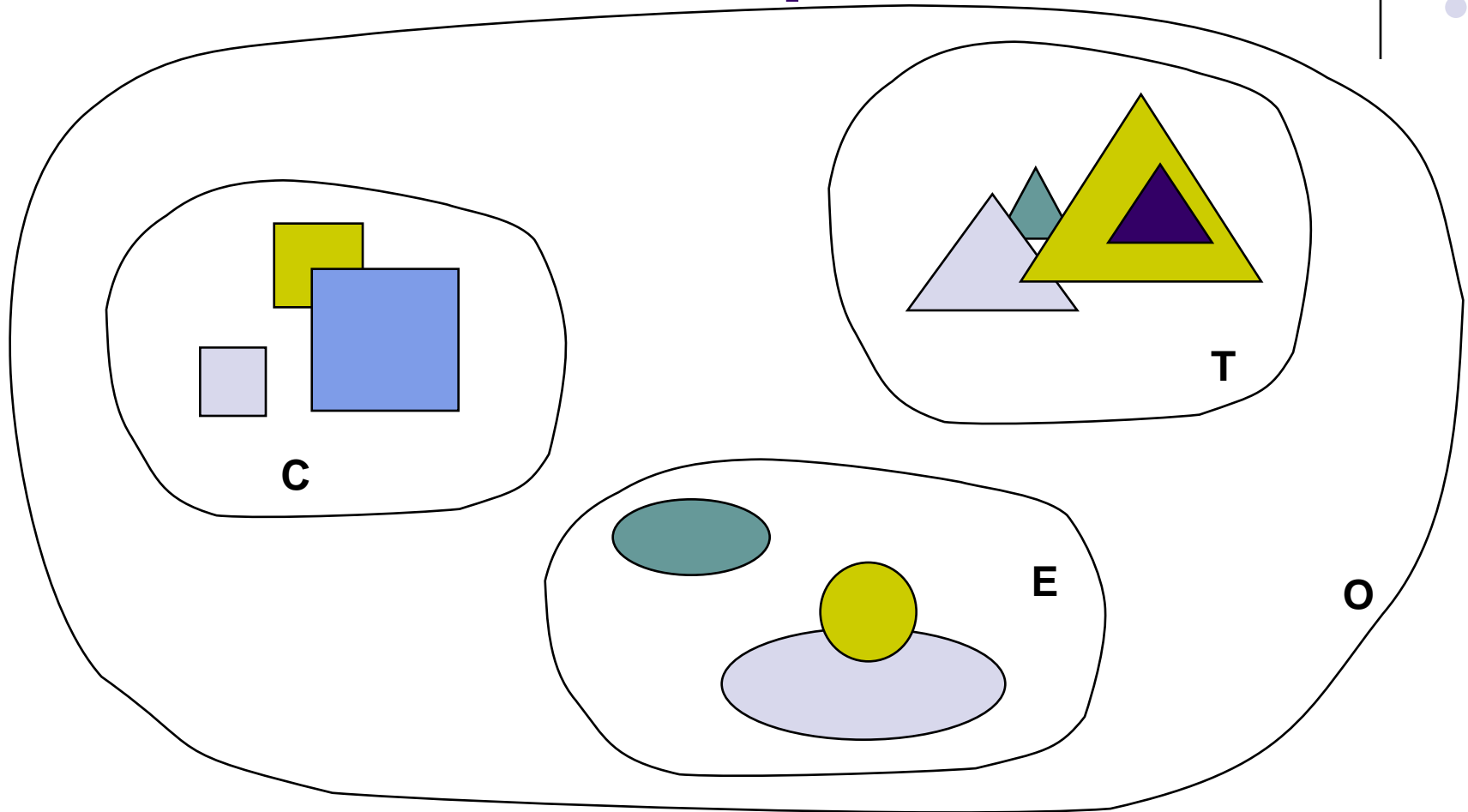
 echo 'Never executed' ;
} catch (Exception e) {
 echo 'Caught exception: ',
 $e->getMessage() , "\n" ;
}
```

# Skriptovací jazyky – PHP - Parametry z požadavků



- `/app/predmet.php?kod=456-513/1&arg=1`
  - `$_REQUEST["kod"] = '456-513/1'`
  - `$_REQUEST["arg"] = '1'`
  - `<a href="go.php?action=del">Odstranit</a>`
- Speciální proměnné
  - `$_REQUEST`, `$_GET`, `$_POST`, `$_FILES`
  - `$_COOKIE`
  - `$_SESSION`
  - `$_SERVER`, `$_ENV`

# OOP – Motivační příklad





# OOP – Motivační příklad (2)

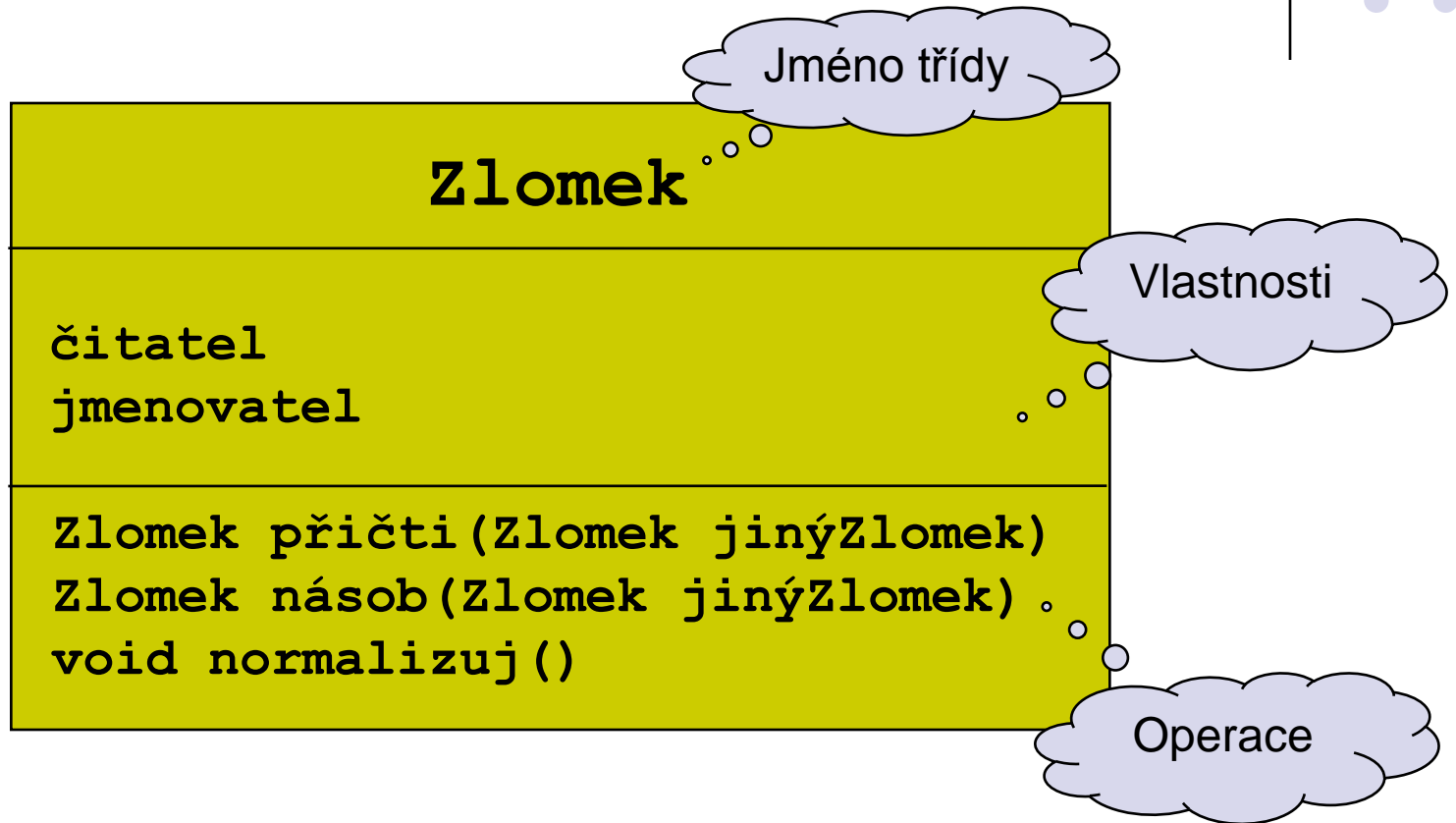
- **Vlastnosti (stav)**
  - souřadnice středu  $x, y$
  - barva
  - obsah, obvod
- **Operace (chování)**
  - přesunutí na jinou pozici
  - $n$ -násobné zvětšení a zmenšení
  - vykreslení na obrazovku
- **Vztahy**
  - susedí, překrývají se, ...



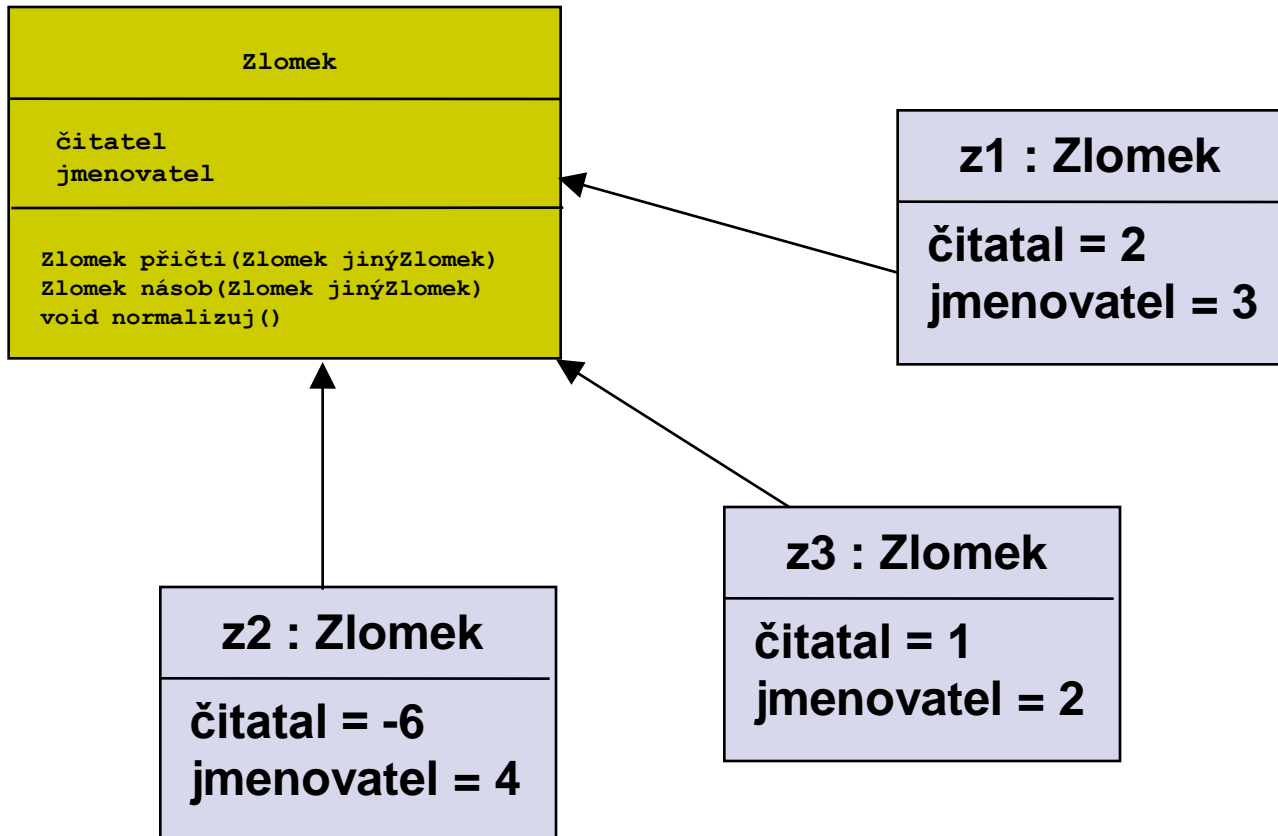
# OOP – Motivační příklad (3)

- **Druh obrazce**
  - čtverec, trojúhelník, elipsa
- **Specifické vlastnosti**
  - délka strany čtverce
  - velikosti poloos elipsy
- **Hodnoty vlastností**
  - konkrétní souřadnice, barva, ...
- **Způsob provedení operací**
  - vykreslení na obrazovku

# OOP – Grafická reprezentace třídy



# OOP – Objekt = instance třídy





# OOP – Třídny v jazyce Java

```
class Zlomek {
 // instanční proměnné
 int citatel;
 int jmenovatel;

 // metody
 Zlomek nasob(Zlomek jiny)
 {
 citatel *= jiny.citatel;
 jmenovatel *= jiny.jmenovatel;
 }
}
```



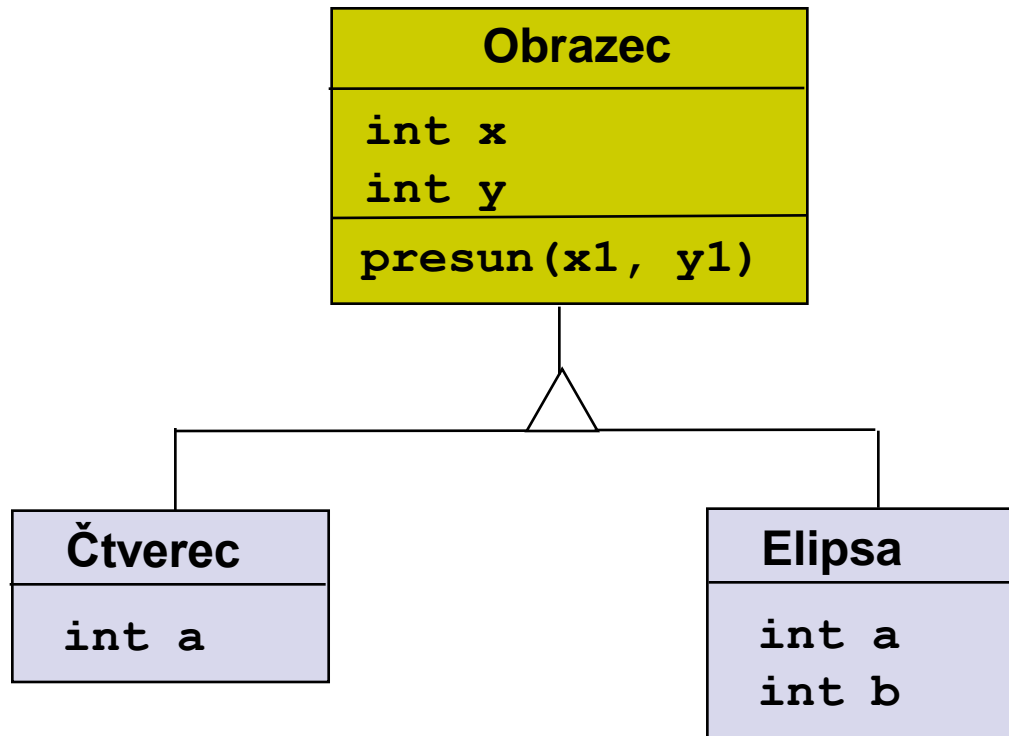
# OOP – Vytvoření instance třídy



```
public static void main(String[] args)
{
 Zlomek z = new Zlomek;
 // nastavení instančních proměnných
 z.citatel = 2;
 z.jmenovatel = 3;
 // volání metody
 z.nasob(z); // z *= z
}
```

reference

# OOP – Dědičnost





# OOP – Dědičnost

```
class Obrazec {
 int x, y;
 void presun(int x, int y) {
 this.x = x; this.y = y;
 }
}

class Ctverec extends Obrazec {
 int a;
}
```



# OOP – Další vlastnosti OOP

- Zapouzdření prvků třídy
  - Soukromé proměnné/metody – private
  - Chráněné proměnné/metody – protected
  - Veřejné proměnné/metody – public
  
- Polymorfismus

# OOP – Smalltalk - Introduction do jazyka Smalltalk



- Existují i jiné objektově orientované jazyky než Java...
- "When I invented the term 'object-oriented' I did not have C++ in mind." -- Alan Kay
- Jedním z „jiných“ objektově orientovaných jazyků je jazyk Smalltalk
  - Ve skutečnosti neexistuje jazyk Smalltalk. Existuje celá řada „variant“ jazyků obsahujících Smalltalk v jejich názvu.
  - Obvykle je pod pojmem Smalltalk rozuměn jazyk Smalltalk-80.

# OOP – Smalltalk - History

## jazyka Smalltalk (1)



- 1968: SIMULA – první „objektově orientovaný“ jazyk
- 1973: Xerox Alto computer
  - Používal Smalltalk (implementovaný v jazyce BASIC)
  - implementoval „želví“ grafiku (LOGO)
  - Třídy (žádná hierarchie), instance, `self`
- 1974:
  - Zlepšení výkonnosti
  - První meta-objekty
- 1976
  - Vše je objekt
  - Hierarchie tříd, `super`
  - Implementováno procházení a ladění zdrojových kódů (code browser, inspector, debugger)

# OOP – Smalltalk - History

## jazyka Smalltalk (2)



- Smalltalk-80 :
  - Jazykový standard
  - Masivně používá MVC
  - Celá řada variant. Volně dostupné jsou:
    - Squeak (open source) - <http://www.squeak.org/>
    - Smalltalk/X – volně dostupný pro nekomerční použití: <http://www.exept.de/>
    - Cincom Smalltalk: volně dostupný pro nekomerční použití: <http://www.parcplace.com/>
    - Strongtalk (typovaný Smalltalk) <http://www.cs.ucsb.edu/projects/strongtalk/pages/index.html>
- Celá řada jazyků vychází s jazyka Smalltalk
  - S# - určený pro tvorbu skriptů
  - Python, Ruby - jazyky postavené na stejných ideách jako Smalltalk, syntaxe jazyka se více blíží Javě a C.

# OOP – Smalltalk - Základní koncepty jazyka Smalltalk (1)



- Smalltalk je *čistě* objektově orientovaný jazyk.
  - Koncepce tříd a objektů
  - Základní myšlenkou je, že vše je objekt a objekty spolu komunikují prostřednictvím zpráv
  - Výjimkou jsou proměnné (jejich obsah proměnnou je).
  - Nejsou v něm „hodnotové“ datové typy.
- Objekty mohou v jazyce Smalltalk provádět právě tři činnosti
  - Udržovat stav (reference na další objekty)
  - Přijímat zprávy od sebe a nebo od jiných objektů
  - V rámci reakce na zprávu posílat zprávy jiným objektům.
- Objekty mohou o jiných objektech zjišťovat informace (nebo měnit stav jiných objektů) posíláním zpráv.



# OOP – Smalltalk - Základní koncepty jazyka Smalltalk (2)



- Vše v jazyce Smalltalk je objekt.
- Každý objekt je instancí nějaké třídy. Třídy jsou také objekty.
- Každá třída je instancí nějaké *metatřídy*.
- Metatřídy jsou všechny instancí třídy `Metaclass`.
- Blok zdrojového kódu je taky objekt
  - Například tělo metody – zprávy
- Výhody tohoto přístupu jsou například:
  - Dynamický typový systém
  - Striktní hierarchie tříd
  - Silný mechanismus reflexe

# OOP – Smalltalk - Mechanismus reflexe (1)



- Smalltalk-80 plně implementuje mechanismus reflexe.
- Strukturální reflexe – *Třídy a metody, které definují systém jsou také objekty a jsou součástí systému, který pomáhají definovat.*
  - Systém se chová „živý“. Nové třídy jsou zkompilovány a přidány do systému (třída *CompiledMethod*).
  - Můžeme se ptát na „otázky“ jako:
    - Jaké metody implementuje třída XY?
    - Jaké třídy jsou definované v systému?

# OOP – Smalltalk - Mechanismus reflexe (2)



- Výpočetní reflexe – schopnost pozorovat aktuální stav systému, průběh výpočtu programu.
  - Můžeme získat odpovědi na otázky jako: Kdo poslal objektu X zprávu Y?

# OOP – Smalltalk - Prostředí jazyka Smalltalk (1)



- Liší se dle specifické implementace konkrétní varianty jazyka Smalltalk
- Obvykle realizuje „image-based persistence“
  - Prostředí pro jazyky jako Java odděluje zdrojový kód od stavu programu.
    - Zdrojový kód je nahrán při startu aplikace.
    - Po ukončení jsou ztraceny všechny data kromě těch, které byly explicitně uloženy.
  - V jazyce Smalltalk je vše objektem, tedy například i třídy, a vše je uloženo jako jeden „image“.
  - Ten může být snadno „obnoven“.

# OOP – Smalltalk - Prostředí jazyka Smalltalk (2)



- Pro spuštění aplikace je obvykle použit virtuální stroj.
  - Obvykle využívá JIT
  - Instalace aplikace je pak dodání „image“ spolu se spustitelnou (binární) verzí virtuálního stroje.
- Vývojové prostředí je obvykle součástí prostředí. Není využíván žádný „externí“ nástroj.
- Výhody
  - Velmi dobré prostředky pro ladění aplikace.
  - Možnost měnit chod aplikace za jejího běhu.
    - Můžeme měnit hierarchii tříd
    - Můžeme měnit vlastní IDE
    - Můžeme měnit činnost garbage collectoru
    - `true become: false` 😊

# OOP – Smalltalk – Syntaxe jazyka



- Čistě objektové jazyky jsou ze své podstaty velmi jednoduché.
- Jazyk Smalltalk má velmi jednoduchou syntaxi.
  - Obsahuje pět klíčových slov:
    - **true**, **false**, **nil**, **self** a **super**
  - Podporuje tvorbu a zasílání zpráv.
  - Obsahuje tři operátory := (přiřazení), = (rovnost), == (identita)
  - Umožňuje realizaci několika typů „literálů“.
  - Poznámky jsou v úvozovkách...

# OOP – Smalltalk – Definice literálů (1)



- Čísla
  - 42
  - -42
  - 123.45
  - 1.2345e2
  - 2r10010010
  - 16rA000
- Znaky
  - Začínají znakem \$ - \$A
- Řetězce
  - Jsou v jednoduchých úvozovkách - 'Hello, world!'

# OOP – Smalltalk – Definice literálů (2)



- Symbols
  - Dva stejné řetězce mohou být uloženy na dvou místech v paměti – může jít o různé objekty
  - Smalltalk obsahuje jiný „typ“ řetězce. Symbol je sekvence znaků a je garantováno, že bude unikátní, právě jedna v systému.
  - Symbol je definován za znakem # - #foo
- Pole
  - # ( 1 2 3 4)



# OOP – Smalltalk – Proměnné



- Implementace proměnných se liší podle konkrétní verze jazyka.
- Obvyklé dělení je na instanční proměnné a dočasné proměnné.
  - Dočasné proměnné jsou v rámci bloku kódu – deklarují se v bloku ohraničeném | |
  - Instanční proměnné (Squeak)
    - Třídní proměnné
    - Globální proměnné
    - Pool variables (Sdílené?)



# OOP – Smalltalk – Zprávy (1)

- Smalltalk „nemá“ žádná klíčová slova

## Java / C++ verze:

```
Transformation t;
float a;
Vector v;
t->rotate(a,v); // for C++
t.rotate(a,v); // for Java
```

## Smalltalk:

```
| t a v |
"lepší!"
| aTransformation angle aVector |
```

```
t rotateBy: a around: v
```

Jde o SmallTalk!



# OOP – Smalltalk – Zprávy (2)

- Definice „těla“ metody `rotateBy`

```
rotateBy: angle around: aVector
| result |
result := do some computations.
^result
```

```
makeWindow
| window |
window := Window new.
window label: 'Hello'.
window open.
```

- Alternativy:

- `rotateAround: aVector by: angle`
- `rotate: angle and: aVector („špatně“)`
- „klíčová“ slova udávají pořadí parametrů



# OOP – Smalltalk – Zprávy (3)

- Obecná struktura zprávy

```
keyword1: param1 keyword2: param2 ...
| local variables |
expressions
```

- Jednotlivé výrazy končí tečkou (kromě poslední).
- Hodnota může být ihned vrácena použitím:
  - operátoru  $\wedge$  -  $\wedge$ value
- Která zpráva bude vybrána je rozhodnuto na základě selektoru: **keyword1:keyword2:... – keyword messages**

# OOP – Smalltalk – Zprávy (4)



- Sémantika posílání zpráv
  - Zprávy jsou posílány objektům
  - První element ve výraze je vždy objekt
  - Výsledek činnosti je také objekt
  - Pořadí vyhodnocování je zleva do prava
  - Pořadí vyhodnocování může být změněno použitím závorek.
  - Smalltalk nemá implicitní volání na sebe sama.

## Java:

```
myMethod();
this.myMethod();
```

## Smalltalk:

```
self myMethod
```

# OOP – Smalltalk – Výrazy (1)



- Aritmetické výrazy
  - Smalltalk neobsahuje unární operátory
  - Speciální binární operátory jako +, -, ...
  - Výsledkem výrazu `4 sqrt` bude 2.0
  - Výsledkem výrazu `1 + 2 * 3` bude 9 (zprávy jsou vyhodnocovány zleva do prava)
  - Výsledkem výrazu `1 + (2 * 3)` bude in 7

# OOP – Smalltalk – Výrazy (2)



- Priorita vyhodnocení zasílaných zpráv ve výraze je následující:
  - Nejvyšší prioritu mají unární zprávy
  - Následují binární operátory
  - Potom *keywords messages*
  - Pořadí vyhodnocování je zleva do prava
- Výraz:  
`3 factorial + 4 factorial between: 10 and: 100`
- Bude vyhodnocen:
  - 3 receives the message "factorial" and answers 6
  - 4 receives the message "factorial" and answers 24
  - 6 receives the message "+" with 24 as the argument and answers 30
  - 30 receives the message "between:and:" with 10 and 100 as arguments and answers true

# OOP – Smalltalk – Výrazy (3)



- Sekvence zpráv určená jednomu objektu může být zapsána jako „kaskáda“ (cascade)

- Místo:

```
| p |
p := Client new.
p name: 'Jack'.
p age: 32.
p address: 'Earth,
```

- Můžeme použít:

```
| p |
p := Client new name: 'Jack'; age: '32';
address: 'Earth'
```





# OOP – Smalltalk – Bloky

- Blok kódu (anonymní Function ) může také být chápán jako literál.
- Syntaxe: [ :params | <message-expressions> ]
- Příklad bloku: [ :x | x + 1 ]
- Můžeme chápat jako:
  - $f(x) = x + 1$
  - $\lambda x.(x+1)$
- Blok je také objekt.
  - Můžeme ho „nechat“ vypočítat svou hodnotu voláním zprávy :value.
  - Může být předán jako parametr:

```
positiveAmounts := allAmounts select: [:amt | amt
isPositive]
```
- Bloky se používají pro realizaci řady uživatelsky definovaných řídicích struktur.

# OOP – Smalltalk – Řídicí struktury (1)



- Řídicí struktury nemají speciální syntaxi v jazyce Smalltalk.
- *Jsou realizovány prostřednictvím zpráv!*
- Například „podmínka“ je realizována voláním zprávy `isTrue` na objekt typu `Boolean`. Argument (blok kódu) se provede jen tehdy, pokud je jeho hodnota `true`.

```
result := a > b
 ifTrue:['greater']
 ifFalse:['less']
```

# OOP – Smalltalk – Řídicí struktury (2)



- Cykly typu „while“

```
| a |
```

```
a := 100 atRandom.
```

```
[a = 42] whileFalse: [a := 100 atRandom]
```

- Cyklus typu „for“

```
100 timesRepeat: [Transcript show: 'Hello
world.'; cr]
```

```
1 to: 100 do: [:i | Transcript show: i; cr]
```

```
1 to: 100 by: 10 do: [:i | Transcript show:
i; cr]
```

```
100 to: 1 by: -1 do: [...] 0.5 to: 7.3 by:
1.1 do: [...]
```

# OOB – Smalltalk – Definice tříd



- **Obecné schéma:**

```
Object subclass: #MessagePublisher
 instanceVariableNames: ''
 classVariableNames: ''
 poolDictionaries: ''
 category: 'Smalltalk Examples'
```

- **Obecná nadtřída je třída Object**
- **Vytvoření nové třídy je vlastně posílání zprávy subclass**
- **Vytvoření nového objektu posláním zprávy new**
  
- **Na závěr Hello world program (omlouvám se, že nebyl někde na začátku☺):**  
Transcript show: 'Hello, world!'.

# OOP – Self – Jazyky založené na prototypch (1)



- Dalším „stupněm“ vývoje objektově orientovaných jazyků jsou jazyky založené na prototypch.
- Jako příklad těchto jazyků může být Self, Io nebo JavaScript.
- Tradiční objektově orientované jazyky obsahují:
  - Třídy – zobecňují vlastnosti a chování množiny objektů
  - Objekty – konkrétní skutečné případy, které třídy zobecňují
- Vývoj jazyků založených na prototypch byl motivován:
  - Je těžké definovat hierarchii tříd, pokud neznáme přesné vlastnosti všech objektů (někdy i pokud je známe).
  - Můžeme použít refaktorizaci, ale v principu by se nám hodil nějaký lepší mechanismus, jak měnit strukturu tříd.
  - Jazyky založené na prototypch tento problém eliminují eliminací duality mezi instancemi objektů a třídami.

# OOP – Self – Jazyky založené na prototypch (2)



- V jazycích založených na prototypch nejsou objekty instancemi tříd
- Nové objekty vznikají klonováním objektů stávajících
  - Prototypy – objekty, které slouží zejména jako vzor pro klonování nových objektů.
  - Pokud chceme vytvořit unikátní typ objektu s právě jednou instancí, nemusíme vytvořit dvě entity – třídu a objekt.
- Tato technika přináší výrazné zjednodušení

# OOP – Self – Popis jazyka Self



- Jazyk Self vychází z jazyka Smalltalk
- Základní vlastnosti:
  - Self obsahuje pouze objekty.
  - Objekty v jazyce self jsou kolekcí „slotů“.
    - Self nerozlišuje instanční proměnné a metody.
  - Do každého slotu můžeme umístit nějaký objekt a tento objekt jsem schopni pak také získat.  
`myPerson name` – vrátí hodnotu uloženou v objektu `myPerson` ve slotu pojmenovaném `name`.  
`myPerson name: 'Marek'` – vloží do slotu novou hodnotu.
  - Self používá bloky kódu (jako Smalltalk).
  - Metody jsou objekty, které kromě slotů obsahují navíc i kód.
    - Ve slotech metody jsou uloženy parametry a dočasné proměnné.
  - Posílání zpráv je základem syntaxe jazyka Self. V principu je řada zpráv posílána implicitně na `self` (jako v Javě `this`).



# OOP – Self – Popis syntaxe

- Obdobně jako ve Smalltalku existují tři typy zpráv:
  - unary - *receiver* slot\_name
  - binary - *receiver* + *argument*
  - Keyword - *receiver* keyword: *arg1* With: *arg2*
    - Klíčové slovo (selektor) začíná malým písmenem, argumenty velkým!
- Příklad použití:
  - `valid: base bottom between: ligature bottom + height And: base top / scale factor.`
  - `valid: ((base bottom) between: ((ligature bottom) + height) And: ((base top) / (scale factor))) .`
- Příklad Hello world programu:  
`'Hello, World!' print.`



# OOP – Self – Vytváření nových objektů (1)



- Nové objekty v jazyce Self vznikají kopírováním `labelWidget copy label: 'Hello, World!'`.
  - Objekt je samostatná entita. Neexistují žádné třídy či meta-třídy.
  - Vytvořený objekt udržuje vazbu na „rodičovský“ objekt.
  - Jeden slot (parent) obsahuje odkaz na rodičovský objekt a může být použit k delegování zpráv na tento objekt.
    - Tímto způsobem je „realizována dědičnost“.
    - Stejný princip lze využít k realizaci jmenných prostorů.
  - Pokud potřebujeme změnit chování objektu, můžeme přidávat či jinak modifikovat sloty.



# OOP – Self – Prostředí

- Prostředí pro realizaci a spuštění programu v jazyce Self je podobné jazyku Smalltalk
  - Programy nejsou „samostatné“ entity.
  - Je použit virtuální stroj.
  - Program je přenášen jako obraz paměti (snapshot).
  - Umožňuje jednoduše modifikovat program za běhu.
  - Snadné ladění aplikace.