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
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.

Sometimes 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 – Plankalkül
- 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 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
- Block 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.
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.
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.
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;
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.
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
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.
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)
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 = "OK";
        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
  - ...

Programming languages
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.
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 exists 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.
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

```plaintext
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".
“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:
  ```plaintext
  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 (e.g. "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:
  ```plaintext
  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.
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
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.
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, automatons,...

● What correct program should do?
  ● SEMANTICS
  ● Lambda calculus, Attributed grammars,...
Specification of programming languages - Formal languages

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

- **Words over an alphabet $\Sigma$**
  - Set of symbols from $\Sigma$ ($\Sigma^*$)
  - Empty set - $\varepsilon$
  - Examples: 1001, pjp, a*(b+b)

- **Language over an alphabet $\Sigma$**
  - A subset of words over an alphabet $\Sigma$
  - Finite or infinite languages
  - Examples:
    - $\{0, 00, 11, 000, 011, 101, 110, 0000, 0011, \ldots\}$
    - $\{\text{int, double, char}\}$
    - $\{a, b, a+a, a+b, b+a, b+b, \ldots, a*(b+b), \ldots \}$
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 – automatons**
   - 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
    $P \subseteq (N \times T)^* N (N \times T)^* \times (N \times T)^*$
    $\alpha \rightarrow \beta$  $\alpha$ - left side, $\beta$ - right side
  - $S$ – start symbol $S \in N$
Specification of programming languages– Grammars (2)

● Binary numbers

● $N = \{S, D\}$  \hspace{1cm} $T = \{0, 1\}$

● $P: \begin{align*}
S & \rightarrow \ D \ | \ S \ D \\
D & \rightarrow \ 0 \ | \ 1 \\
S & \rightarrow S \ D \rightarrow S \ 0 \rightarrow S \ D \ 0 \rightarrow D \ D \ 0 \rightarrow 1 \ D \ 0 \rightarrow 1 \ 1 \ 0 \\
S & \rightarrow^* 1 \ 1 \ 0
\end{align*}$
Specification of programming languages - Grammar’s derivation tree

- \[ S \Rightarrow S \Rightarrow S \Rightarrow S \Rightarrow D \Rightarrow D \Rightarrow 0 \Rightarrow D \Rightarrow D \Rightarrow 1 \Rightarrow D \Rightarrow 0 \Rightarrow 1 \Rightarrow 0 \]
Specification of programming languages -
Chomsky Language Classification (1)

- Type 0 – Unrestricted languages
  \[ \alpha \rightarrow \beta \quad \alpha, \beta \text{ all possibilities} \]
- 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 automatons

- **Type 3 – Regular languages**
  Even more effective methods to analyze them
  Finite automatons
Specification of programming languages- Finite automata

States + transitions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
Specification of programming languages - Finite automatons

\[ A = (Q, \Sigma, \delta, q_0, F) \]

- **Q**: a finite set of states
- **\( \Sigma \)**: input alphabet
- **\( \delta \)**: state transition function
  \[ \delta_{\text{NFA}} : Q \times \Sigma \rightarrow 2^Q \quad \delta_{\text{DFA}} : Q \times \Sigma \rightarrow Q \]
- **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 automatons
  - 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)
  
  \[
  \begin{align*}
  <\text{decl}> & \rightarrow \text{‘DEF’} \ <\text{ident}> \ \text{‘=} \ <\text{expr}> \ <\text{expr1}> \\
  & \quad | \quad \text{‘TYPE’} \ <\text{ident}> \ \text{‘=} \ <\text{type}> \\
  <\text{expr1}> & \rightarrow \text{‘;’} \ <\text{expr}> \ <\text{expr1}> \\
  & \quad | \quad \text{e}
  \end{align*}
  \]

- Example: DEF a = 1;

- Extended Backus-Naur Form (EBNF)
  - Extended with regular expression's operators
    
    \[
    \begin{align*}
    <\text{decl}> & \rightarrow \text{‘DEF’} \ <\text{ident}> \ \text{‘=} \ <\text{expr}> \ ( <\text{expr}> )^* \\
    & \quad | \quad \text{‘TYPE’} \ <\text{ident}> \ \text{‘=} \ <\text{type}>
    \end{align*}
    \]
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}` while R do S `{Q \land \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 describe 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 expresses what computer should do and when
  - Statement has a side effect
  - 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** (× 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 – \(\lambda\)-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, ...

- λ-abstraction
  - (λx . e)

- Application
  - (e₁ e₂)

- Parentheses convention
  - λx . λy . e₁ e₂ = (λx . (λy . e₁ e₂))
  - e₁ e₂ e₃ = ((e₁ e₂) e₃)
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 \( x \ a \ y \)
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.

- $(\lambda x y . f x y) [g z / f] = \lambda x y . (g z) x y$
- $(\lambda x y . f x y) [g z / x] = \lambda x y . f x y$
- $(\lambda x y . f x y) [g y / f] = error$ in substitution
Functional programming – Evaluation of $\lambda$-expressions

- $\alpha$-reduction
  - $\lambda x . e \leftrightarrow \lambda y . e[y / x]$
  - Renaming of a captured variable

- $\beta$-reduction
  - $(\lambda x . e_1) e_2 \leftrightarrow e_1[e_2/x]$
  - “function’s call” – replacing a parameter with an argument

- $\eta$-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](http://haskell.org)
    - Language specification and other resources
  - [http://haskell.org/hugs](http://haskell.org/hugs)
    - Installation packages (Win / Unix)
    - User’s manual (is a part of installation)
Haskell – Hugs Interpret

- Basic evaluation: calculator
  
  $ \texttt{hugs}
  
  Prelude> 2*(3+5)
  16

- Script: containing user’s definitions
  
  $ \texttt{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**
  
  ```haskell
  module Example where
  -- Function computing sum of two numbers
  sum x y = x + y
  ```

- **Example.lhs**

  ```haskell
  > 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 -> 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):
  - \( f \ pat11 \ pat12 \ldots = rhs1 \)
  - \( f \ pat21 \ pat22 \ldots = rhs2 \)
  - \ldots\)

- First corresponding equation is chosen.

- If there is none \(\rightarrow\) 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
  - nl (Leaf _) = 1
  - nl (Node _ l r) = (nl l) + (nl 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**
  - \( \text{fakt1} \ n = \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fakt1} (n-1) \)
  - \( \text{fakt2} \ 0 = 1 \)
  - \( \text{fakt2} \ n = n \times \text{fakt2} (n-1) \)
  - \( \text{fakt3} \ 0 = 1 \)
  - \( \text{fakt3} \ (n+1) = (n+1) \times \text{fakt3} \ n \)
  - \( \text{fakt4} \ n | n == 0 = 1 \)
  - \( | \text{otherwise} = n \times \text{fakt4} (n-1) \)

- **Fibonacci numbers**
  \( \text{fib} :: \text{Int} \rightarrow \text{Int} \)
  \( \text{fib} \ 0 = 0 \)
  \( \text{fib} \ 1 = 1 \)
  \( \text{fib} \ (n+2) = \text{fib} \ n + \text{fib} \ (n+1) \)
Haskell – Example

- List length
  - `length [] = 0`
  - `length (x:xs) = 1 + 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*
  
  \[
  f \ x \ y = \text{let } p = x + y \\
  \qquad q = x - y \\
  \text{in } p * q
  \]

- Construction *where*
  
  \[
  f \ x \ y = p * q \\
  \text{where } p = x + y \\
  \qquad q = x - y
  \]
Haskell – Partial function application

- \( \text{inc} \ x = 1 + x \)
- \( \text{inc} \ x = \text{add} \ 1 \ x \)
- \( \text{inc} = \text{add} \ 1 \)
- \( \text{inc} = (+1) = (1+) \)
- \( \text{add} = (+) \)

- Eta reduction

- Point free programming
  - \( \text{lcaseString} \ s = \text{map} \ \text{toLower} \ s \)
  - \( \text{lcaseString} = \text{map} \ \text{toLower} \)
Haskell – Lambda abstraction

- Using function like a parameter
  
  ```haskell
  nonzero xs = filter p xs
  where p x = x /= 0
  ```

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

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

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

- Example creating a list of squared numbers
  
  ```haskell
  dm [] = []
  dm (x:xs) = sq x : dm xs
  where sq x = x * x
  ```

- List’s ordering (quicksort)
  
  ```haskell
  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

- \([m..n]\)
  - \([1..5] = [1,2,3,4,5]\)
- \([m1,m2..n]\)
  - \([1,3..10] = [1,3,5,7,9]\)
- \([m..]\)
  - \([1..] = [1,2,3,4,5,\ldots]\)
- \([m1,m2..]\)
  - \([5,10..] = [5,10,15,20,25,\ldots]\)
Haskell – Function filter

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

```haskell
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
```

Programming languages
Haskell – Function map

- List’s elements
  
  \[
  \text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
  \]
  
  \[
  \text{map } f \ [\ ] \ = \ [\ ]
  \]
  
  \[
  \text{map } f \ (x:xs) \ = \ f \ x : \text{map} \ f \ xs
  \]
  
  \[
  \text{map} \ (+1) \ [1,2,3] \ = \ [2,3,4]
  \]
  
  \[
  \text{map } \text{toUpper} \ \text{"abcd"} \ = \ \text{"ABCD"}
  \]
  
  \[
  \text{squares} \ x \ = \ \text{map} \ (\lambda x \rightarrow x \ast x) \ [1..]
  \]
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
- \[ f \ x \mid x \leftarrow xs \] = map f xs
- \[ x \mid x \leftarrow xs, \ p \ x \] = filter p xs

- \[ (x,y) \mid x\leftarrow xs, \ y\leftarrow ys \] =
  \[ (x_1,y_1),(x_1,y_2),(x_1,y_3),..., \]
  \[ (x_2,y_1),(x_2,y_2),(x_2,y_3),..., \]
  ... ]
Haskell – Example

- Set’s operation using list’s generators
  - Intersection
    \[
    \text{intersect } xs \ ys = [y \mid y \leftarrow ys, \text{elem } y \ xs]
    \]
  - Union
    \[
    \text{union } xs \ ys = xs ++ [y \mid y \leftarrow ys, \text{notElem } y \ xs]
    \]
  - Difference
    \[
    \text{diff } xs \ ys = [x \mid x \leftarrow xs, \text{notElem } x \ ys]
    \]
  - Subset
    \[
    \text{subset } xs \ ys = [x \mid x \leftarrow xs, \text{notElem } x \ ys] == []
    \]
    \[
    \text{subset } xs \ ys = \text{all } (\lambda x \rightarrow \text{elem } x \ ys) \ xs
    \]
**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

```haskell
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

<table>
<thead>
<tr>
<th>Type Class</th>
<th>Methods and Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq a</td>
<td>(==), (/=)</td>
</tr>
<tr>
<td>Eq a =&gt; Ord a</td>
<td>(&lt;), (&lt;=), (&gt;), (&gt;=), min, max</td>
</tr>
<tr>
<td>Enum a</td>
<td>succ, pred</td>
</tr>
<tr>
<td>Read a</td>
<td>readsPrec</td>
</tr>
<tr>
<td>Show a</td>
<td>showsPres, show</td>
</tr>
<tr>
<td>(Eq a, Show a) =&gt; Num a</td>
<td>(+), (-), (*), negate, abs</td>
</tr>
<tr>
<td>(Num a) =&gt; Fractional a</td>
<td>(/), recip</td>
</tr>
<tr>
<td>(Fractional a) =&gt; Floating a</td>
<td>pi, exp, log, sqrt, (**), …</td>
</tr>
</tbody>
</table>
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 form 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 \textit{main}

- Represents main program
- Action returning nothing:
  
  \begin{itemize}
    \item \texttt{main :: IO ()}
    \item \texttt{main = do c <- getChar}
    \item \texttt{putChar c}
  \end{itemize}

1. Reads character and marks it \( c \).
2. Write character \( c \).
3. Returns the result of last action – \texttt{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)

\[(xs ++ ys) ++ zs = xs ++ (ys ++ zs)\]

\[
\begin{align*}
[] ++ ys &= ys \\
(x:xs) ++ ys &= x: (xs ++ ys)
\end{align*}
\]

(++.1) (++.2)

a) \[ \] \Rightarrow xs

\[
\begin{align*}
([] ++ ys) ++ zs &= ys ++ zs \\
&= [] ++ (ys ++ zs)
\end{align*}
\]

(++ 1) (++.1)
Haskell – Example – Associativity of ++ (2)

\[(xs ++ ys) ++ zs = xs ++ (ys ++ zs)\]

\[\begin{align*}
[] ++ ys &= ys & (++.1) \\
(x:xs) ++ ys &= x: (xs ++ ys) & (++.2)
\end{align*}\]

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

\[\begin{align*}
((x:xs)++ys)++zs &= x:(xs++ys)++zs & (++.2) \\
&= x:((xs++ys)++zs) & (++.2) \\
&= x:(xs++(ys++zs)) & \text{(assumption)} \\
&= (x:xs)++(ys++zs) & (++.2)
\end{align*}\]
Haskell – Example – length (xs++ys) (1)

length (xs++ys) = length xs + length ys

length [] = 0 (len.1)
length (_:xs) = 1 + length xs (len.2)
a) [] => xs
   length ([] ++ ys)
       = length ys (++.1)
       = 0 + length ys (base case +)
       = length [] + length ys (len.1)
Haskell – Example – length (xs++ys) (2)

length (xs++ys) = length xs + length ys

\[
\begin{align*}
\text{length } [] & = 0 \quad \text{(len.1)} \\
\text{length } (_\text{::}xs) & = 1 + \text{length } xs \quad \text{(len.2)}
\end{align*}
\]

b) (x:xs) => xs

\[
\begin{align*}
\text{length } ((x:xs) \text{ ++ } ys) & \\
& = \text{length } (x:(xs++ys)) \quad \text{(++ .2)} \\
& = 1 + \text{length } (xs++ys) \quad \text{(len.2)} \\
& = 1 + (\text{length } xs + \text{length } ys) \quad \text{(Assumption)} \\
& = (1 + \text{length } xs) + \text{length } ys \quad \text{(+ Associativity)} \\
& = \text{length } (x:xs) + \text{length } ys \quad \text{(len.2)}
\end{align*}
\]