SOFTWARE DEVELOPMENT 1

Terms and Programming Concepts

2018W
A. Ferscha (Institute of Pervasive Computing, JKU Linz)
ACTIVITIES IN SOFTWARE DEVELOPMENT

- Activities in Software Development:
  - Requirements Analysis
  - Specification
  - Design
  - Programming
  - Testing
  - Commissioning

- Object-oriented Software Development:
  - Object-oriented Analysis (OOA)
  - Object-oriented Design (OOD)
    - Analysis / Design is supported by object-oriented modelling languages
      - OMT (Object Modelling Technique)
      - UML (Unified Modelling Language)
  - Object-oriented Programming (OOP)
    - Programming with object-oriented programming language
      - Smalltalk, C++, Objective C, Java, Classic Ada, Delphi, Eiffel, Beta
    - high potential for reuse
DEFINITION: PROGRAMMING

What is a programming language?

A programming language is used to implement requirements, specifications and designs in machine readable form.

What is a machine?

To be precise: To get to expressions that can be translated automatically to machine executable instructions.

What means translate/compile?
// Application to compute area and hypotenuse of a rectangular triangle

class Pythagoras {
    public static void main(String[] args) {
        double a = 3.5, b = 5.5;
        double area, hypotenuse;

        area = a * b / 2;
        hypotenuse = Math.sqrt(a * a + b * b);

        System.out.println("catheti: " + a + " cm and " + b + " cm");
        System.out.println("area: " + area + " qcm");
        System.out.println("hypotenuse: " + hypotenuse + " cm");
    }
}

Output:

    catheti: 3.5 cm and 5.5 cm
    area: 9.625 qcm
    hypotenuse: 6.519202405202649 cm

→ SWE1.01 / Pythagoras.java
LANGUAGES

Languages are used by humans to exchange information.

We need to formulate algorithms in words, that can be understood and executed by a computer.

=> Programming language

Language in this case does not mean “spoken”, but “written communication” (text, graphical symbols, …)

A programming language includes syntax and semantics:
- Syntax determines, which expressions are valid and can be checked by the computer (more precisely: by the compiler)
- Semantics means the meaning of these expressions. Within limits this can be formally verified, meaning, it is possible to prove if the program does what has been specified
ARTIFICIAL LANGUAGES

■ Also known as formal languages.

■ Is defined by an exact specification; there is no “fuzziness” or ambiguities as it is common in natural languages.

■ A formal language is comprised of a set of words, which are composed from the underlying alphabet by applying syntactical rules (grammar).

■ The meaning of “alphabet” and “word” differs slightly from the usual meaning in natural languages:
  □ An alphabet contains the smallest units of the language. In programming languages these are „tokens“ (e.g. reserved words like if, for, int, or variable identifiers); although tokens are again described by a simpler formal language, and are usually made up by letters.
  □ A word is a composition of several elements of the alphabet that fulfill all the syntactic rules. In a programming language, a word is a program.
DEFINITION OF ARTIFICIAL LANGUAGES

- **Alphabet**
  - is an ordered finite set of distinguishable symbols
  - A is the set of available symbols and ≤ a linear order over A
  - (A, ≤) is called alphabet

- A* is called free monoid over A
  - A* is a triple (A, +, ε)
  - + is the operator for the concatenation of symbols
  - ε is the empty word and central element of the monoid
  - A* contains the set of all words that can be constructed from A and the empty word ε

- **Language**
  - each partial set of L ⊆ A* is called language over A
  - a specific partial set L (language) is defined by a grammar

e.g. A = { H, S, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 }

Rule (Grammar): Only words that start with HS and are followed by 1 or 2 numbers.
L = { HS1, HS2, HS3, ... }
GRAMMAR

- Grammar
  - defines the syntax of a language ("rules")
  - comprised of rules, that define which words are part of this language
  - for natural languages this is difficult to specify; e.g. there pop up new words and expressions all the time
  - in computer science a formal basis for the definition of a formal language

- Grammar of a formal language contains:
  - Terminal symbols are characters or sequences of characters and appear in words (programs) of the language
  - Non-terminal symbols do not appear in words (programs); they are just needed to coordinate the production rules
  - Production rules define how a sequence of terminal and non-terminal symbols can be replaced by another sequence of terminal and non-terminal symbols
  - The starting symbol is a special non-terminal symbol. It defines, which rule has to be applied first to produce a valid word (program)
SYNTAX NOTATION

- How is a programming language defined?
  - Grammar rules become confusing in larger languages
  - Description of the language with a meta language (formal language)
  - e.g. syntax diagrams or Backus-Naur Form (BNF)

- Backus-Naur Form (BNF)
  - Meta language to describe context-free grammars
  - Used by J. Backus and P. Naur to describe ALGOL 60
  - Meta symbols of BNF:
    - `< >` mark a non-terminal symbol
    - `::=` defined as
    - `|` or
SYNTAX NOTATION

- Example: BNF for Identifier

\[
\begin{align*}
\langle Number \rangle & ::= \ 0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ 5 \ | \ 6 \ | \ 7 \ | \ 8 \ | \ 9 \\
\langle Letter \rangle & ::= \ A \ | \ B \ | \ C \ | \ ... \ | \ Z \\
\langle Identifier \rangle & ::= \ < \ Letter > \ | \ < \ Letter > \ < \ String > \\
\langle String \rangle & ::= \ < \ Letter > \ | \ < \ Number > \ | \\
\langle Letter \rangle \ < \ String > \ | \\
\langle Number \rangle \ < \ String > \ \\
\end{align*}
\]
SYNTAX NOTATION

- Extended Backus-Naur-Form (EBNF)
  - More short forms
  - Additional meta symbols (different notation than in BNF)
    - "abc" mark terminal symbols
    - x = y x is defined as y
    - (x|y) there has to be either x or y (exactly one of the two)
    - [x] x may be (or not be) there
    - {x} x may be there several times (not at all, once or multiple times)
    - . end of production

- Example: EBNF for Identifier
  - BNF rules 3 and 4 from before become:

    Identifier = Letter { Letter | Number } .
SYNTAX NOTATION

- Syntax diagram

  - Graphical notation of context-free grammars
  - Introduced by N. Wirth in PASCAL at the beginning of the 70s
  - The grammar of the programming language MODULA-2 is completely written in syntax diagrams
  - Assign each non-terminal symbol a directed graph with an additional input- and output-edge

- Meta-symbols
  - Oval: Terminal symbol (A)
  - Rectangle: Non-terminal symbol (Identifier)

- A run through the graph from the input- to the output edge produces a possible production for this non-terminal symbol
SYNTAX NOTATION

- Example: Syntax diagram for Identifier

- Number

- Letter

- Identifier

- Letter

- Number
SYNTAX NOTATION

Syntax of a student list \{ (FirstName, LastName, StudentID), ... \}
Example: \{ ("Max", "Mustermann", 0123456), ("Wilhelm", "Bush", 0700001) \}
SYNTAX NOTATION

- Example: Syntax diagram for Assignment

- Example: Floating-point number

- Example: Recursion in a syntax diagram
Grammars are production systems for formal languages. Having specified a valid program in such a language, we need a recognition system that can process this program. In other terms: a program that takes a program as input.

- **Compiler und Interpreter**
  - Are programs, that check if a given program is part of the language, meaning it is syntactically correct
  - Convert the given program in a representation that can be processed by the machine
  - **Compilers** first transform the program to machine code, then execute the code
  - **Interpreters** take instruction by instruction from the input program and execute each one immediately
COMPILATION

- Example: Steps of a compiler

```
a = 3 + 4;
```

1. Lexical Analysis

```
[ident a] [assign] [lit 3] [plus] [lit 4] [semikolon]
```

2. Syntactical Anal.

```
a = +
3 4
```

3. Code Generation

```
mov #3, R1
mov #4, R2
add R1, R2
mov R2, a
```

4. Code Optimization

```
mov #3, a
add #4, a
```
MACHINES, DATA AND INSTRUCTIONS

Level of abstraction

Actual significance for SD

Low

High

Turing Machine

Von Neumann Machine

Java Virtual Machine
THE TURING TEST

“A person communicates with a computer through a terminal. When the person is unable to decide whether he is talking to a computer or another person, the computer can safely be said to possess all important characteristics of intelligence.”

Alan Turing (1912 - 1954)
TURING MACHINE

- Universal Machine Model
  - proposed by the English mathematician A.M. Turing 1936
  - just a theoretical model, not a physical processor

- Formal Definition:
  \[ M = (B, I, Q, \alpha, \Delta) \]

- \( B \) Bandalphabet \{a,b,c,...\} * \{z\}
- \( z \) Leerzeichen
- \( I \) Eingabealphabet \( I \subseteq B \setminus \{z\} \)
- \( Q \) endliche Zustandsmenge \{\alpha,\beta,\chi,...\}
- \( \alpha \) Anfangszustand
- \( \Delta \) Übergangsfunktion \( Q \times B \rightarrow Q \times B \times \{L, R, H\} \)
- \( L,R,H \) Bewegungsrichtung: Links, Rechts, Halt
Der Turing-Maschine Simulator

Bei der nachfolgenden Implementation eines Turing-Simulators wird das Alphabet $A$ auf 0, 1, $x$ beschränkt. # stellt das Leerzeichen dar. Eine Zeile der Turing-Tabelle wird durch $S, 0 \rightarrow s_1, r$ dargestellt. $S$ steht für den Anfangszustand, $s_1 \ldots s_n$ steht für Zustand 1 bis n. halt steht für den Haltezustand.


Weitere Quellen zu Turing

Wer sich eingehender mit der Biographie von Alan Turing beschäftigen möchte, dem sei die deutsche Übersetzung des Buches von Andrew Hodges (Hod94) empfohlen. Es kann direkt beim Springer Verlag bestellt werden. Andrew Hodges unterhält eine eigene Seite über Alan Turing (in Englisch).

VON NEUMANN ARCHITECTURE

John von Neumann
(1903 - 1957)
VON NEUMANN ARCHITECTURE

- Universal architecture

- To solve a specific problem, a program has to be entered

- Five working units: processing unit, control unit, memory, input-, output-mechanism

- Memory:
  - programs, data and results are all stored in the same memory
  - separated in cells of identical size
  - numbered sequentially (addresses)
  - value of a cell can be accessed through it’s address

- Execution
  - consecutive instructions are put in consecutive memory cells
  - load next instruction by increasing the instruction counter by 1
  - jump instructions allow deviations from the normal execution order
  - there are arithmetic, logical, transport and jump instructions
JAVA VIRTUAL MACHINE

- Java VM specifies
  - Java Instruction-Set
  - Data types
  - Operands-Stack
  - Constants-Pool
  - Methods-Area
  - Heap for Runtime-Data
  - Class File Format

- Java Instruction-Set
  - up to 256 OpCodes (each 8bit, 51 not in use)
  - 25 variations
  - 3 reserved OpCodes
  - 0, 1, 2, 3, ... Operands
  - no register specification
  - elementary und complex instructions
  - local variables relative to base pointer
  - highly compacted code (~1.8 bytes)

- Data types
  - byte
  - short
  - integer
  - long
  - float
  - double
  - char
  - object
  - returnAddress
Java Code

Compiler: Code → JVM Byte Code

Byte Code

Interpreter

Operating System

Any CPU

Java Code

Compiler: Code → JVM Byte Code

Byte Code

JIT (Just-In-Time) Compiler: Byte Code → Machine Code

Binary Machine Code

Operating System

Any CPU
PROCESSORS

- Turing machine is just a theoretical model
  - in reality there are no infinite tapes
  - Turing applications are “somewhat” unmanageable

- Actual processors (Von-Neumann machine, real processors, JVM) have a different structure and procedures:
  - they understand basic instructions, e.g.
    ```
    load ACC, #294
    store REG1, 4567
    add ACC, REG1
    cmp 4321
    jump #4711
    ```
  - this is still rather difficult to work with; at a very low level of programming
  - complex, prone to errors, difficult to modify
  - higher level of abstraction is desirable
  - problem oriented program code is better
**HISTORY OF PROGRAMMING LANGUAGES**

1. **Binary Machine Code**
   - 001001
   - e.g.: MC68000-Assembler
   - Constructs: MOVE, ADD, DBP

2. **Assembler**
   - e.g.: MC68000-Assembler
   - Constructs: MOVE, ADD, DBP

3. **Imperative/Procedural Languages**
   - e.g.: Fortran, Cobol, Algol, PL1, Ada, C
   - Constructs: procedures, while, if

4. **Functional Languages**
   - e.g.: LISP, ML, Miranda
   - Constructs: Functions

5. **Logical Languages**
   - e.g.: Prolog
   - Constructs: Facts, Rules

6. **Object-Oriented Languages**
   - e.g.: C++, C#, Smalltalk, Eiffel, Java

**Time**
Programming Concepts (demonstrated in Java)
**NAMES AND NAMING CONVENTIONS**

Names (also Identifiers, Bezeichner): identify variables, literals, types etc.

- can be of any length and all characters are significant
- have to start with a letter ('A' to 'Z', 'a' to 'z', '_', '$')
- may contain any Unicode-character
- are case sensitive (upper-/lowercase matters)
- symbols like %, *, @ are usually reserved for operators
- should always be meaningful, e.g. concatenate several words to indicate what a variable contains, a method does, etc.

Valid names:
```plaintext
a, hypotenuse, MyFirstVariable, x_4711
```

Invalid names:
```plaintext
4a, hypo%use, a-b, ;-)```

KEYWORDS (SCHLÜSSELWÖRTER)

Keywords are words that have a special meaning in a programming language and can not be used as an identifier.

Java reserves the following words as keywords:

abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, false, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while
Everything in-between /* and */ or in-between // and the end of the line is ignored by the compiler and seen as a comment, intended for humans.

// This is a single comment line

/* This is a multi-line comment.
   Multi-line comments may not contain other multi-line comments! */

/**
 * This is a Javadoc comment, which can be used to generate
 * <em>Documentation</em> from the source files. You can use
 * HTML in Javadoc comments.
 */

In Java sources you will often find comments formatted like the one above. These are Javadoc comments, used to automatically generate documentation from the source code files. A good example for this is the Java API itself: all the documentation for the Java API is generated with Javadoc.
METHODS

Executable parts of a program (instructions) are defined in “methods”.

```java
// Application to compute area and hypotenuse of a right-angled triangle
class Pythagoras {
    public static void main(String[] args) {
        double a = 3.5, b = 5.5;
        double area, hypotenuse;
        area = a * b / 2;
        hypotenuse = Math.sqrt(a * a + b * b);
        System.out.println("catheti: \" + a + \" cm and \" + b + \" cm");
        System.out.println("area: \" + area + \" qcm");
        System.out.println("hypotenuse: \" + hypotenuse + \" cm");
    }
}
```

More about methods later in “Classes and Objects”.

Modifiers

Return type

Main Method

Parameters

Method Body: Instructions
EXPRESSIONS

- Expressions (Ausdrücke) are instructions (Anweisungen)
  - Expressions always return a result (contrary to instructions)
  - The result has to have a specific type and can be used in further expressions or commands

- Typical expressions:
  - Assignments: $i = 3$
  - Operators: $j = (k = i * 7)$
  - Variables, Literals: String $s = "..."
  - Method calls: $i = s.length()$
  - Object instantiation: LinkedList list = new LinkedList();
VARIABLES

- Variables are used to store and process values. These values can be modified during program execution.

  - Compare to a cell in the memory of a von Neumann machine
  - A variable needs to have a data type. It can only store data of this type
  - Before a variable can be used, its name and type have to be specified. This is called declaration (Deklaration)
  - After the declaration an initial value can be assigned to the variable. This is called initialization (Initialisierung)
  - Some script languages (like JavaScript) do not require a declaration or specification of a data type. Although this seems easier at a glance, it also introduces a whole bunch of possible runtime errors. The compiler (or interpreter) can only guess what kind of data will be assigned to the variable and therefore can not check if an operation is valid.
Variables are declared by specifying their type, followed by a name.

- Several variables of the same type can be declared by adding more names, separated by commas (,).
- Declarations are instructions and need to be closed with a semi-colon (;).

Example (see "first program"):

```java
double a = 3.5, b = 5.5;
double area, hypotenuse;

area = a * b / 2;
hypotenuse = Math.sqrt(a * a + b * b);
```
INITIALIZATION

- An initial value is assigned with the assignment-operator = (Zuweisungs-operator). The value to the right is assigned to the variable on the left.
  - Initializations are instructions and need to be closed with a semi-colon (;)
  - Assignments require both sides to be of the same type

- Example (see “first program”):
  ```java
  double a = 3.5, b = 5.5;
  double area, hypotenuse;
  area = a * b / 2;
  hypotenuse = Math.sqrt(a * a + b * b);
  ```

Declaration and initialization can be combined in a single statement.

area and hypotenuse are initialized with according values.
DECLARATION AND INITIALIZATION WRAP-UP

- Declaration of a new variable:
  ```
  type name;
  ```

- Declaration of multiple variables of identical type:
  ```
  type name1, name2, name3;
  ```

- (Initial) Assignment of a value to a variable:
  ```
  name = value;
  ```

- (Initial) Assignment of a value of a different type than the variable:
  ```
  name = (type)value;
  ```

- Declaration combined with initialization:
  ```
  type name = value;
  ```

- Scope of the declaration:
  - (Local) Variables are only valid within the block `{}` in which they are declared.
  - Later in „Classes and Objects“ we will discuss other variables than local variables, which are accessible from other parts of the program.
Data Types
DATA TYPES

- The data type of a variable defines:
  - range of values
  - valid operations

- **Primitive Data Types:** defined by the language
  - Integer numbers:
    - byte (8 bit), short (16 bit), int (32 bit), long (64 bit)
  - Floating-point numbers:
    - float (32 bit), double (64 bit)
  - Logical types (true or false):
    - boolean
  - Characters:
    - char

- **Complex Data Types (Classes):** constructed from primitive data types
  - Strings:
    - String
  - Arrays:
    - e.g. char[]
DATA TYPES: LITERALS

- **Literals** are constant values: numbers, characters or sequences.

  - can be used to initialize variables
  - a literal always has a certain primitive data type, which has to be considered in assignments
  - literals are only available for:
    - primitive data types: `int`, `long`, `float`, `double`, `char`, `boolean`
    - character sequences: `String`
    - arrays: more about that later
LOGICAL TYPES: BOOLEAN

- The data type boolean only stores the logical values true or false

- Operators: & (bitwise and), | (bitwise or), ^ (xor), ! (negation), && (logical and), || (logical or)

- Literals: The only valid literals for booleans are true and false

- Example: De Morgan's Law
  
  ```java
  boolean isMale = true, isDrunk = true;
  // deMorgan's Law: (!p) && (!q) = !(p || q)
  // and:            (!p) || (!q) = !(p && q)
  boolean isFemale = !isMale;
  boolean isSober = !isDrunk;
  boolean deMorgan1 = (isFemale && isSober) == !(isMale || isDrunk);
  boolean deMorgan2 = (isFemale || isSober) == !(isMale && isDrunk);
  ```
  
  → SWE1.01 / BooleanBsp.java
Gottfried Wilhelm Leibnitz:
*De Progressione Dyadica (Zur diadischen Progression)*
Leipzig, 15. März 1679
DE L’ARITMÉTIQUE BINAIRE
INTEGER NUMBERS (BYTE, SHORT, INT, LONG)

Value range: depends on number of bits; 8, 16, 32, 64 bits.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value Range</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>-128 .. 127</td>
<td>8 Bit</td>
</tr>
<tr>
<td>short</td>
<td>-32768 .. 32767</td>
<td>16 Bit</td>
</tr>
<tr>
<td>int</td>
<td>-2147483648 .. 2147483647</td>
<td>32 Bit</td>
</tr>
<tr>
<td>long</td>
<td>-9223372036854775808 .. 9223372036854775807</td>
<td>64 Bit</td>
</tr>
</tbody>
</table>

Encoding

Positive numbers in the binary system: \( x = \sum_{i=0}^{n-1} d_i \times 2^i \)

Example: print a number in binary format:

```java
while (x != 0) {
    if (x % 2 == 0)
        print('0');
    else
        print('1');
    x = x / 2;
}
```
# INTEGER NUMBERS

- To encode (positive) numbers: binary value encoding

<table>
<thead>
<tr>
<th>$d_3$</th>
<th>$d_2$</th>
<th>$d_1$</th>
<th>$d_0$</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**INTEGER NUMBERS**

- To encode signed (negative) numbers: Two's complement

<table>
<thead>
<tr>
<th></th>
<th>d$_3$</th>
<th>d$_2$</th>
<th>d$_1$</th>
<th>d$_0$</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-7</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-8</td>
</tr>
</tbody>
</table>
INTEGER NUMBERS

Example: Compute two's complement of a negative number

- Given
  -x
  -4

- Find \(d_i\) for x
  - Invert bits
    - 0100
  - Add 1
    - 1011
  - Result
    - 1100
**INTEGER NUMBERS**

- Example: Compute two's complement of a negative number

  - **Given**
    - \(-x\) -4
  - **Find d_i for x**
    - 0100
    - Invert bits
      - 1011
    - Add 1
      - 0001
    - Result
      - 1100

  - **Advantage**: Negative numbers in two's complement do not need special treatment in arithmetic computations:

    \[
    \begin{array}{c}
    \text{0011} \\
    + \text{1011} \\
    \hline
    = \text{1110}
    \end{array}
    \]
    
    \[
    \begin{array}{c}
    3 \\
    -5 \\
    \hline
    -2
    \end{array}
    \]
Subtractions can be reduced to a negation and addition.

Problem: Watch for overflows!

\[
\begin{array}{c}
0111 & 7 \\
+ 0001 & 1 \\
\hline
= 1000 & -8 \text{ wrong}
\end{array}
\]

Trick: Duplicate sign-bits, check if they are identical after the operation:

\[
\begin{array}{c}
00111 & 7 \\
+ 00001 & 1 \\
\hline
= 01000 & ?
\end{array}
\]

\[
\begin{array}{c}
00011 & 3 \\
+ 11011 & -5 \\
\hline
11110 & -2
\end{array}
\]

result undefined \hspace{1cm} \text{result ok!}
INTEGER NUMBERS: LITERALS

- Integer numbers can simply be stated by their decimal value:
  0, 20, 4332, -323, +4238, ...

- By prepending "0x" it is possible to state numbers in the hexadecimal system.
  The letters A to F can be both upper- or lowercase:
  0x3AF, 0xfff178, ...

- Prepending a "0" indicates a number in the octal system:
  04123, 01114, ...

- By default an integer literal is of the type int. To enter a long-literal, append the letter “L” (either upper- or lowercase) to the number:
  0L, 100000000000L

  Warning: It is easy to confuse 1 with 1 (one), therefore its better to the uppercase representation “L”.

- byte and short do not need literals of their own. You may assign any int-literal as long as the number is small enough.
INTEGER NUMBERS: EXAMPLE

// create 32 bit integer with largest possible value
int largeNumber = Integer.MAX_VALUE;
System.out.println(largeNumber);

// increase by 1 -> overflow!
largeNumber = largeNumber + 1;
System.out.println(largeNumber);

// create 8 bit integer and assign the far too large number
byte smallNumber = (byte)largeNumber;
System.out.println(smallNumber);

■ Output:
  2147483647
  -2147483648
  0

→ SWE1.01 / IntBsp.java
FLOATING POINT NUMBERS (FLOAT, DOUBLE)

- **float** and **double** store floating point numbers according to the IEEE- Standard 754-1985 in the format: sign, exponent, significand

- Encoding: 32 Bits (float) are split in
  - sign 1 Bit,
  - exponent 8 Bits,
  - significand 23 Bits.

- Value = \[
\begin{align*}
(-1)^s \times 2^{e-127} \times 1.f & \quad \text{if } 0 < e < 255 \quad \text{(normalized)} \\
(-1)^s \times 2^{-126} \times 0.f & \quad \text{if } e=0 \quad \text{(subnormal)}
\end{align*}
\]

  - If normalized, the value of the significand 1.f lies in the interval [1,2]
  - Binary numbers after the floating point 0.d_{-1} d_{-2} d_{-3} ... d_{k} mean:
    \[
    \sum_{i=-k}^{-1} d_i \cdot 2^i
    \]
FLOATING POINT NUMBERS

- 64 bit floating point numbers (double) are split in
  - sign: 1 Bit
  - exponent: 11 Bits
  - significand: 52 Bits

<table>
<thead>
<tr>
<th>X</th>
<th>xxxxxxxxxxxx</th>
<th>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign s</td>
<td>exponent e</td>
<td>significand f</td>
</tr>
</tbody>
</table>

- Value:
  \[
  (\begin{cases} 
  (-1)^s \times 2^{e-1023} \times 1.f & \text{if } 0 < e < 2047 \\
  (-1)^s \times 2^{-1022} \times 0.f & \text{if } e=0 
  \end{cases}
  \]
  (normalized)
  (subnormal)

- The largest positive number that can be stored is
  \[
  2 \cdot 2^{2048-1023} \approx 10^{308}
  \]

- The smallest positive number is
  \[
  2^{-52} \cdot 2^{-1022} = 2^{-1074} \approx 10^{-324}
  \]
FLOATING POINT NUMBERS

■ Floating point numbers always have the same precision (52 bits for doubles), but the value range can be adjusted by the exponent:

\[
\begin{align*}
\text{e} &= 1021 & \text{-0.5} & \text{0.5} \\
\text{e} &= 1024 & \text{-4} & \text{-2} & \text{2} & \text{4} \\
\text{e} &= 1031 & \text{-512} & \text{-256} & \text{256} & \text{512}
\end{align*}
\]

■ Since the precision of floating point numbers is limited (the larger the range, the less precise), it is necessary to approximate numbers that have no exact representation.

\[
\begin{align*}
\text{1.3 (decimal)} & \quad \text{s} = + & s = 0b \\
& \text{e} = 1023 & e = 0111111111b \\
& \text{f} = 0.3 & f = 0.100110011001100110011001100110011001100110011001100b \\
& = 1.2999999999999982236431605997495353221893310546875
\end{align*}
\]
FLOATING POINT NUMBERS: EXAMPLE

```java
public class DoubleBsp {
    public static void main(String[] args) {
        // create 3 different 64-Bit floating point numbers
        double a = 1.5, b = 1.3, c = 1.4;
        // commutative property: both results should be 4.2
        double result1 = (a + b) + c;
        double result2 = a + (b + c);
        // output results: surprise! ;)
        System.out.println(result1);
        System.out.println(result2);
        System.out.println(result1 == result2);
    }
}
```

Output:
4.199999999999999
4.2
false

→ SWE1.01 / DoubleBsp.java
FLOATING POINT NUMBERS: LITERALS

- **floats** are defined by a trailing \( f \) or uppercase \( F \).
  
  e.g. \( 7f \), \( 3.8F \)

- **doubles** may include a trailing \( d \) or uppercase \( D \).
  
  e.g. \( 121D \), \( 1.4E7d \)

- Numbers that do not include any of \( f \), \( F \), \( d \) or \( D \) but are obviously floating point numbers, are assumed to be of the type **double**.

- Numbers are recognized as floating point if one of these conditions is met:
  - there is an exponent, e.g. \( 1E3 \)
  - there is a decimal point, e.g. \( 3.0 \)
  - and of course if the number ends with \( f \), \( F \), \( d \) or \( D \)
CHARACTERS (CHAR)

- Value range:
  - all characters from the Unicode-Charset

- Encoding:
  - each character is encoded in a 16 bit value

- Literals:
  - Character literals are enclosed in apostrophes, e.g. 'Q', '5' or '?'
  - special chars can be entered with the following escape sequences:

    - \b: backspace BS (Zeichen zurück) (ASCII: 8)
    - \t: horizontal tab HT (horizontaler Tabulator) (ASCII: 9)
    - \n: line feed LF (Zeilenvorschub) (ASCII: 10)
    - \f: form feed FF (Seitenvorschub) (ASCII: 12)
    - \r: carriage return CR (Wagenrücklauf) (ASCII: 13)
    - \": double quote (doppeltes Anführungszeichen)
    - \': single quote (einfaches Anführungszeichen)
    - \\: backslash (Rückwärts-Schrägstrich)
    - \015: octal form feed FF
    - \u000D: hexadecimal form feed FF
STRING LITERALS

- Strings are character sequences.
  - String-literals are enclosed in double quotes
  - Characters can be entered identical to char-literals
  - Examples:
    - "Use double quotes to start and end a String"
    - "This is a String\nincluding a line feed"
SUMMARY: LITERALS

Examples:

```java
byte b;        b = 123;
short i;      i = -1234;
int i;        i = -1234;
long i;       i = -1234L;
float x, y;   x = -123.45F; y = 1.0E-23;
double x, y;  x = -123.45; y = 1.0E-23;
boolean a, b; b = true;   b = false;
char c;       c = 'A';
String s;     s = "Abc";   s = null;
String h;     h = "Hundert \u20ac";
```
SOFTWARE DEVELOPMENT 1

Terms and Programming Concepts
2018W
A. Ferscha (Institute of Pervasive Computing, JKU Linz)