Design and Implementation of Programming Languages
Overview of Language Design

- Elements language design
  - Syntax
  - Operational semantics (expressions, declarations)
  - Static semantics (type checking, type inference, static analyses)
  - Modularization mechanisms

- Goals and principles
  - Expressivity (computational power, expressive typing, flexibility of modularization)
  - Declarativeness (capture design intention, what instead of how, intuitive abstractions)
  - Orthogonality (independent mechanisms for independent features)
  - Regularity (regular rules, without exceptions)
  - Conciseness (concise syntax, inference, implicit conversions)
  - Implementation properties (efficiency, independent compilation and distribution)
Overview of Language Implementation

- Industrial-strength implementations typically split into **front-end** and **back-end**
  - **Front-end**: compiler
    - Parses source code
    - Resolves references, analyses for errors
    - Applies transformations
    - Emits target code (bytecode or machine code)
  - **Back-end**: virtual machine
    - Reads intermediate representation (bytecode)
    - Abstracts over hardware: memory, I/O
    - Provides services: garbage collection, dynamic linking
    - Executes program (interpretation or just-in-time compilation)
    - Applies (adaptive) optimizations
Grading & Deliverables

- **Term Paper (40%)**
  - 1 paper per groups of 1, 2, or 3 students
  - 4, 6, or 8 pages, respectively (ACM SIGPLAN style)
  - Collectively graded

- **Reviews (20%)**
  - 2 reviews per student
  - 500 words minimum per review
  - Individually graded

- **Talk (40%)**
  - 1 talk per group
  - 15, 25, or 35 minutes of uninterrupted talk, respectively
  - 5-15 minutes questions & answers
  - Individually graded
Hints

- **Term Paper**
  - Correct
  - Focussed (on certain questions or issues)
  - Clear and precise (avoid vague concepts and statements)
  - Comprehensible (Background, figures, examples)
  - Well-structured (Few, if any, forward-references)
  - Meaningful references
  - **Never** commit plagiarism!
Hints (cont’d)

- **Reviews**
  - Constructive
  - Summary (short)
  - List major positive and negative critique points
  - Discusses details
    - (structural issues, missing references, spelling, etc.)

- **Talk**
  - Used PowerPoint to good (rather than ill) effect?
  - Well-structured?
  - Level of detail appropriate for target audience?
  - No superfluous “5 seconds” slides?
  - Too long? Too short?
The Good, the Bad, and the Ugly

- **Research:**
  - “I’ve used all my initial references. That should be about enough.”
  - “I have googled a bit to find further relevant sources.”
  - “First, I have used Google to get an overview. Then, I have consulted the websites of publishers, conferences, and individual authors. Here, I have followed both forward- and backward-references.”

- **Structure:**
  - “Structure? The one used by the original authors should’ve survived copy-and-paste just fine.”
  - “Each topic covered gets a section of its own.”
  - “The structure is chosen to best meet the goals of the paper (e.g. the questions to be answered). There is a clearly recognisable argumentation line connecting the sections of the paper.”
The Good, the Bad, and the Ugly (cont‘t)

- **Style:**
  - “All sentence consist of subject, verb, and object.”
  - “I’ve written the paper just like my high school essays.”
  - “I know how the papers I’ve read are structured; I will do likewise. Others have helped me to eliminate mistakes and ambiguities.”

- **Scientific Contribution:**
  - “Wikipedia in LaTeX doesn’t look so bad!”
  - “I’ve read a number of papers and summarized them for my fellow students.”
  - “I’ve found all truly relevant sources to my topic, which I have then analysed to answer the most important questions on the topic. This process also led to some new insights, which I have presented in the paper.”
Resources

- ACM Digital Library
  <http://portal.acm.org/dl.cfm/>

- Springer Link
  <http://www.springerlink.de/>

- IEEE
  <http://elib.tu-darmstadt.de/ieee/>

- Citeseer
  <http://citeseer.ist.psu.edu/>

- Google Scholar
  <http://scholar.google.com/>
Important Dates

- **This week:** assignment of topics
- **Next week:** meeting with your supervisor
- **31 May (Tuesday):** initial submission of paper
- **14 June (Tuesday):** submission of reviews
- **5 July (Tuesday):** final submission of paper
- **19 July (Tuesday):** presentation (blockseminar)
- **Any time:** questions
#1: Extensible Module Systems
Betreuer: Tom Dinkelaker

- **Background:**
  - Modules (aka packages) contain interfaces and classes, which again contain their members. We use modules to encapsulate certain related functionality.
  - Most popular programming languages only have a restricted module system that comes with one specific semantics.
  - There are special programming languages in which one can extend the concept of modules.

- **Assignment:**
  - Explain the different concepts of existing module systems and what benefits modules bring (encapsulation, information hiding, ...)
  - Explain extension mechanisms for building extensible module systems
  - Present example extensions for module systems (e.g. AOP)

- **Initial References:**
#2: Symbolic Execution
Betreuer: Eric Bodden

- **Background:**
  - Symbolic execution means executing a program not with real values (int, double, float, ...) but with symbolic values, e.g. value ranges such as “x<=5” instead
  - Allows automatic reasoning about programs
  - Used for test-case generation and model checking

- **Assignment:**
  - Read and summarize three 2011 papers, one survey paper and two ISSTA papers that show how to make symbolic execution practical for Java programs; *What are the current open challenges?*

- **Initial References:**
  - Symbolic Execution for Software Testing in Practice–Preliminary Assessment (Cadar et al., ICSE 2011)
  - Enabling Dynamic Symbolic Execution of Real-World Java Programs (Anand et al., ISSTA 2011)
  - Symbolic Execution with Mixed Concrete-Symbolic Solving (Pasareanu et al., ISSTA 2011)
    (The first paper is online; Eric Bodden can provide copies of the latter two papers)
#3: Automated Learning of API Specifications

**Betreuer: Eric Bodden**

- **Background:**
  - API specifications define how programmers use an API correctly
  - Problem: nobody likes writing specifications
  - Idea: infer specifications by looking at typical API usages

- **Assignment:**
  - Read and summarize three current papers that take different approaches to this problem. Answer: What assumptions do the approaches make? What is the idea behind each approach? How do the approaches compare?

- **Initial References:**
#4: Constraint Logic Programming
Betreuer: Vaidas Gasiunas

- **Background:**
  - Extends logic programming (Prolog) with constraints
  - A declarative way to formulate constraint satisfaction problems

- **Assignment:**
  - Investigate motivation, goals, advantages and limitations of CLP
  - Give an overview of the concepts and the major domains of CLP
  - Try out a couple of existing implementations

- **Initial References:**
  - Thom W. Frühwirth et al. Contraint Logic Programming - An Informal Introduction
#5: Constructors for Multiple Inheritance  
Betreuer: Jan Sinschek, Vaidas Gasiunas

- **Background:**
  - Object initialization is difficult in presence of multiple inheritance (diamond problem, traits)

- **Assignment:**
  - Give an overview of the problems related to object initialization and the proposed solutions
  - Illustrate the problem and the proposed solutions by examples
  - Analyze the advantages and limitations of these proposals.

- **Initial References:**
  - Ulrich W. Eisenecker, Frank Blinn, and Krzysztof Czarnecki: A Solution to the Constructor-Problem of Mixin-Based Programming in C++
  - Donna Malayeri, Jonathan Aldrich: CZ: Multiple Inheritance without Diamonds
#6: Refinement Types

Betreuer: Jan Sinschek, Vaidas Gasiunas

- **Background:**
  - Refinement Types are types of the form \( \{x:B \mid C(x)\} \), where B is a base type and \( C(x) \) a predicate constraining its instances.
  - Initial implementation exist in novel languages such as Microsoft’s F# and IBM’s X10.

- **Assignment:**
  - Explain the concepts and implementation principles of ref. types.
  - Implement examples demonstrating their typical uses in F# and X10.
  - Discuss limitations and challenges.

- **Initial References:**
#7: Materialized Views
Betreuer: Ralf Mitschke

**What are materialized views?**

derived relations in a database

stored back in database for efficiency (caching; indexing)

updated incrementally on data changes

**Used in semantics of logic languages (e.g. Datalog)**

considerable problem for recursive queries

ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).
Assignment:
Compare different algorithms for:
  - efficiency
  - supported maintenance operations (e.g. only insertions)
  - computation semantics of maintenance
  - Relational algebra (good for parallelization)
  - Recursive algorithm
  - size of auxiliary data

Starting point:
Incremental Maintenance of Recursive Views: A survey
(Dong - 1998) – In Materialized Views (Book)
Background:
As the compiler for the Scala programming language emits Java bytecode, it has to encode ("dumb-down") many of the more advanced features of Scala in Java bytecode.

Assignment:
Give a brief overview of interesting language features and the encodings used to compile them.
Experiment to find out where the encodings currently in use diverge from the one originally described – and why.

Initial References:
Burak Emir. Object-Oriented Pattern Matching. 2007
M. Schinz. Compiling Scala for the Java Virtual Machine. 2005
#9: Actors on the JVM

Betreuer: Andreas Sewe

Background:
Multiple implementations of actor on the JVM exit. For Scala alone, several libraries implement actor-based concurrency models.

Assignment:
Give an overview of actors and their properties.
Compare the different implementation approaches and their respective tradeoffs.

Initial References: