Design and Implementation of Programming Languages
Overview of Language Implementation

- Industrial-strength implementations typically split into **front-end** and **back-end**

  - **Front-end**: compiler
    - Parses source code
    - Analyses source code
    - 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?
  - Comprehensible? (Background, figures, examples)
  - Well-structured? (Few, if any, forward-references)
  - Meaningful references?
  - **Never** commit plagiarism!
Hints (cont’d)

- Reviews
  - Constructive?
  - Summary (short?)
  - Names *The good, the bad, and the ugly*?
  - 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 builds upon the similarities I have identified in the various sources. (I’ve also pointed out differences where necessary.)”
The Good, the Bad, and the Ugly (cont‘t)

- **Style:**
  - “All sentences 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 categorized. 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

- **Today:** (attempt to) assign topics.
  - If all else fails, we will resort to preferences send by mail
- **Next week:** detailed task descriptions (from your advisors)
- **16 December:** initial submission of paper
- **21 December 2009 – 8 January 2010:** *Christmas break*
- **20 January:** submission of reviews
- **3 February:** final submission of paper
- **12 February:** *end of lectures*
- **17 February:** draft of presentation
- **~3 March:** presentation (blockseminar)
- **Any time:** questions
#1: Generational Garbage Collection and Pretenuring

- **Background:**
  - The (weak) generational hypothesis: "Most objects die young"
  - Generational garbage collection (GC) exploits this by collecting young and old (tenured) generations separately.
  - But even very long-lived objects have once been young – needlessly.
  - Pretenuring tries to avoid the resultant overhead

- **Assignment:**
  - Overview of generational GC and pretenuring
  - Classification of pretenuring approaches and their accuracy/coverage

- **Initial References:**
  - Blackburn et. al.: Profile-Based Pretenuring, 2007
  - Marion, Jones, Ryder: Decrypting the Java Gene Pool: Predicting objects' lifetimes with micro-patterns, 2007

Andreas Sewe <sewe@st.informatik.tu-darmstadt.de>
The Design of Modern Garbage Collectors

Background:
- The fundamental techniques in garbage collections are decades old: reference counting, marking, and sweeping, copying, or compacting.
- Still, these building blocks can be composed in new ways.
- Immix, a recent GC algorithm, is one such composition.

Assignment:
- Survey of GC building blocks
- Description and classification of the Immix algorithm

Initial References:
- Blackburn et. al.: Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance, 2008
- Bacon, Cheng, Rajan: A Unified Theory of Garbage Collection, 2004

Andreas Sewe <sewe@st.informatik.tu-darmstadt.de>
#3: Static and Dynamic Race Detection

- **Background:**
  - Concurrent programs suffer from data races: concurrent accesses to the same memory location where at least one of them is a “write”.
  - Researchers have developed a multitude of algorithms to detect races either at compile time (statically) or at runtime (dynamically).

- **Assignment:**
  - Survey of existing approaches to Data-Race detection
  - Description the relative strengths and weaknesses

- **Initial References:**

Eric Bodden <bodden@cs.tu-darmstadt.de>
#4: Inference of Typestate Properties

- **Background:**
  - Typestate properties describe which operations are allowed on an object, depending on the object’s internal state, the typestate.
  - Researchers have developed algorithms that try to infer valid typestate properties by analyzing program code or program runs.

- **Assignment:**
  - Survey of existing approaches to the inference of typestate props.
  - Description the relative strengths and weaknesses

- **Initial Reference:**

Eric Bodden <bodden@cs.tu-darmstadt.de>
#5: Unification Algorithms

- **Background:**
  - Unification is at the heart of logic programming languages
  - Naïve implementations take exponential time and space, but efficient algorithms do exist

- **Assignment:**
  - Discussing and presenting common terminology
  - Presenting an efficient algorithm

- **Initial Reference:**
  - An Efficient Unification Algorithm; Alberto Martelli and Ugo Montanari; ACM ToPLaS; Vol 4., No. 2, 1982
  - Unification theory; Franz Baader; LNCS 572; Springer 1992

Michael Eichberg <eichberg@cs.tu-darmstadt.de>
#6: Warren’s Abstract Machine

- **Background:**
  - The Warren Abstract Machine (WAM) has become the de facto standard for implementing Prolog compilers.
  - The WAM pioneered a large number of different optimizations

- **Assignment:**
  - Discussing the high-level architecture of the WAM
  - Discussing the properties of the WAM (e.g., the performed optimizations,...)

- **Initial Reference:**
  - Warren’s Abstract Machine - A TUTORIAL RECONSTRUCTION; Hassan Ait-Kaci; 1999

  Michael Eichberg <eichberg@cs.tu-darmstadt.de>
#7: Type inference for dynamically typed languages

- **Background:**
  - Dynamically typed language become more and more popular and used in large-scale industrial projects (Python, Ruby, Groovy)
  - Knowing statically the runtime type, and or an approximation (e.g. a super-class) is useful for building advanced IDEs (e.g. navigate between code elements, search for usages, bug finding)

- **Assignment:**
  - Discuss the different type inference systems of programming language compilers, interpreters, IDEs (how many types can be found, reliability, applicability to other programming languages, etc)

- **Initial References:**
  - Type inference for Python: <http://lambda-the-ultimate.org/node/1519>
  - Type inference for Smalltalk: Roel Wuyts' RoelTyper, <http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/>

*Martin Monperrus <monperrus@cs.tu-darmstadt.de>*
#8: LePUS3 – A Visual Design Description Language (1-2 Students)

Background:

- Visual description language for object-oriented design
- Formal specification with minimal vocabulary
- Axiomatized in decidable subset of the first-order predicate logic
- Abstraction mechanisms can specify design patterns

Assignment:

- Discuss used formalization
- Discuss application of formalization for expressing design patterns

Initial Reference:

http://www.cs.concordia.ca/faculty/eden/lepus/

http://www.nicholsonweb.co.uk/publications/nfm09.pdf

Ralf Mitschke <mitschke@st.informatik.tu-darmstadt.de>
#8: LePUS3 – A Visual Design Description Language (1-2 Students)